eBook Gratuit

APPRENEZ C++

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#c++

Table des matières

À propos 1
Chapitre 1: Démarrer avec C ++ 2
Remarques 2
Versions 2
Examples 2
Bonjour le monde 2
Une analyse 2
commentaires 4
Commentaires sur une seule ligne 4
C-Style / Bloc Commentaires 4
Importance des commentaires 5
Marqueurs de commentaires utilisés pour désactiver le code 6
Fonction 6
Déclaration de fonction 6
Appel de fonction 7
Définition de fonction 8
Fonction de surcharge 8
Paramètres par défaut 8
Appels de fonctions spéciales - Opérateurs 9
Visibilité des prototypes et déclarations de fonctions 9
Le processus de compilation C ++ standard 11
Préprocesseur 12
Chapitre 2: Algorithmes de bibliothèque standard 14
Examples 14
std :: for_each 14
std :: next_permutation 14
std :: accumuler 15
std :: find 17
std :: count 18

std :: count_if 19

std :: find_if 21

std :: min_element 22

Utiliser std :: nth_element pour trouver la médiane (ou d'autres quantiles) 24

Chapitre 3: Alignement 25

Introduction 25

Remarques 25

Examples 25

Interroger l'alignement d'un type 25

Contrôle de l'alignement 26

Chapitre 4: Arithmétique en virgule flottante 27

Examples 27

Les nombres à virgule flottante sont étranges 27

Chapitre 5: auto 29

Remarques 29

Examples 29

Échantillon automatique de base 29

auto et modèles d'expression 30

auto, const et références 30

Type de retour 31

Lambda générique (C ++ 14) 31

objets auto et proxy 32

Chapitre 6: Bit Manipulation 34

Remarques 34

Examples 34

Mettre un peu 34

Manipulation de bits de style C 34

Utiliser std :: bitset 34

Effacer un peu 34

Manipulation de bits de style C 34

Utiliser std :: bitset 35

En changeant un peu....35
Manipulation de bits de style C....35
Utiliser std :: bitset....35
Vérification un peu....35
Manipulation de bits de style C....35
Utiliser std :: bitset....36
Changer le nième bit en x....36
Manipulation de bits de style C....36
Utiliser std :: bitset....36
Définir tous les bits....36
Manipulation de bits de style C....36
Utiliser std :: bitset....36
Supprimer le bit mis à l'extrême droite....37
Manipulation de bits de style C....37
Jeu de bits de comptage....37
Vérifiez si un entier est une puissance de 2....38
Application de manipulation de bits: Lettre minuscule à majuscule....38
Chapitre 7: Boucles....40
Introduction....40
Syntaxe....40
Remarques....40
Examples....40
Basé sur la gamme pour....40
Pour la boucle....43
En boucle....46
Déclaration de variables dans des conditions....46
Boucle Do-while....47
Instructions de contrôle de boucle: Break and Continue....48
Portée pour une sous-gamme....49
Chapitre 8: C incompatibilités....51
Introduction....51

Examples .51
Mots-clés réservés .51
Pointeurs faiblement typés 51
aller ou changer .51
Chapitre 9: Catégories de valeur 52
Examples .52
Signification des catégories de valeur .52
valeur .52
xvalue .53
lvalue .53
glvalue .54
rvalue .54
Chapitre 10: Champs de bits 56
Introduction 56
Remarques .56
Examples .57
Déclaration et utilisation 57
Chapitre 11: Classes / Structures 59
Syntaxe .59
Remarques .59
Examples .59
Les bases de la classe .59
Spécificateurs d'accès .60
Héritage .61
Héritage Virtuel 63
Héritage multiple .64
Accéder aux membres de la classe 66
Contexte .67
Héritage privé: restriction de l'interface de la classe de base .67
Classes finales et structures .68
Relation amicale .69
Classes / Structures imbriquées .70

Types de membres et alias 74
Membres de la classe statique 78
Fonctions de membre non statiques 83
Structure / classe sans nom 85
Chapitre 12: Compiler et construire 87
Introduction 87
Remarques 87
Examples 87
Compiler avec GCC 87
Liaison avec les bibliothèques: 89
Compilation avec Visual C ++ (ligne de commande) 89
Compiler avec Visual Studio (interface graphique) - Hello World 93
Compiler avec Clang 100
Compilateurs en ligne 100
Le processus de compilation C ++ 102
Compiler avec Code :: Blocks (interface graphique) 104
Chapitre 13: Comportement défini par la mise en œuvre 110
Examples 110
Char peut être non signé ou signé 110
Taille des types intégraux 110
Taille de char 110
Taille des types d'entiers signés et non signés 110
Taille de char16_t et char32_t 112
Taille de bool 112
Taille de wchar_t 113
Modèles de données 113
Nombre de bits dans un octet 114
Valeur numérique d'un pointeur 114
Plages de types numériques 115
Représentation de la valeur des types à virgule flottante 116
Débordement lors de la conversion d'un entier en entier signé 117

Type sous-jacent (et donc taille) d'un enum....117
Chapitre 14: Comportement non défini....118
Introduction....118
Remarques....118
Examples....119
Lecture ou écriture à travers un pointeur nul....119
Aucune déclaration de retour pour une fonction avec un type de retour non vide....120
Modifier un littéral de chaîne....120
Accéder à un index hors limites....120
Division entière par zéro....121
Dépassement d'entier signé....121
Utilisation d'une variable locale non initialisée....122
Plusieurs définitions non identiques (la règle de définition unique)....123
Appariement incorrect de l'allocation de mémoire et de la désallocation....124
Accéder à un objet avec le mauvais type....124
Débordement en virgule flottante....125
Appel de membres virtuels (purs) à partir d'un constructeur ou d'un destructeur....125
Suppression d'un objet dérivé via un pointeur sur une classe de base sans destructeur virt....126
Accéder à une référence en suspens....126
Extension de l'espace de noms `std` ou` posix`....127
Débordement lors de la conversion vers ou à partir du type à virgule flottante....128
Jet statique de base à dérivé invalide....128
Appel de fonction via le type de pointeur de fonction incompatible....128
Modifier un objet const....128
Accès à un membre inexistant via un pointeur sur un membre....129
Conversion de base en base invalide pour les pointeurs vers les membres....130
Arithmétique de pointeur invalide....130
Déplacement par un nombre de postes invalide....131
Retourner d'une fonction [[noreturn]]....131
Détruire un objet qui a déjà été détruit....131
Récursion du modèle infini....132
Chapitre 15: Comportement non spécifié....133
Remarques....133

Examples......133

Ordre d'initialisation des globales à travers TU......133

Valeur d'un enum hors gamme......134

Cast statique à partir de la valeur nulle et bidon......134

Résultat de certaines conversions réinterprétées......134

Résultat de certaines comparaisons de pointeurs......135

Espace occupé par une référence......135

Ordre d'évaluation des arguments de fonction......136

Déplacé de l'état de la plupart des classes de bibliothèque standard......137

Chapitre 16: Comportements plus indéfinis en C ++......139

Introduction......139

Examples......139

Se référant à des membres non statiques dans les listes d'initialisation......139

Chapitre 17: Concurrence avec OpenMP......140

Introduction......140

Remarques......140

Examples......140

OpenMP: Sections parallèles......140

OpenMP: Sections parallèles......141

OpenMP: Parallel For Loop......142

OpenMP: collecte / réduction en parallèle......142

Chapitre 18: constexpr......144

Introduction......144

Remarques......144

Examples......144

variables constexpr......144

fonctions constexpr......146

Static si déclaration......148

Chapitre 19: Conteneurs C ++......150

Introduction......150

Examples......150

Organigramme des conteneurs C ++......150

Chapitre 20: Contrôle de flux 153
Remarques 153
Examples 153
Cas 153
commutateur 153
capture 154
défaut 154
si 155
autre 155
aller à 155
revenir 156
jeter 156
essayer 157
Structures conditionnelles: si, si..se 158
Instructions de saut: pause, continuer, aller, sortir. 159
Chapitre 21: Conversions de type explicites 163
Introduction 163
Syntaxe 163
Remarques 163
Examples 164
Base à la conversion dérivée 164
Jetant la constance 165
Type de conversion 165
Conversion entre pointeur et entier 166
Conversion par constructeur explicite ou fonction de conversion explicite 167
Conversion implicite 167
Conversions Enum 168
Dérivé de la conversion de base pour les pointeurs en membres 169
annule * à T * 169
Coulée de style C 170
Chapitre 22: Copier Elision 171
Examples 171

But de l'élision de la copie 171
Elision de copie garantie 172
Valeur de retour elision 173
Élision des paramètres 174
Elision de valeur de retour nommée 174
Copie de l'initialisation 175
Chapitre 23: Copier vs assignation 176
Syntaxe 176
Paramètres 176
Remarques 176
Examples 176
Opérateur d'assignation 176
Constructeur de copie 177
Constructeur de constructeur Vs Assignment Construct 178
Chapitre 24: Correct Correct 180
Syntaxe 180
Remarques 180
Examples 180
Les bases 180
Const Correct Design de classe 181
Paramètres de fonction const constants 183
Const correctité comme documentation 185
const CV-Qualified Member Fonctions: 185
const Paramètres de fonction: 187
Chapitre 25: Date et heure en utilisant entête 190
Examples 190
Temps de mesure en utilisant 190
Trouver le nombre de jours entre deux dates 190
Chapitre 26: decltype 192
Introduction 192
Examples 192
Exemple de base 192

Un autre exemple 192

Chapitre 27: déduction de type 194

Remarques 194

Examples 194

Déduction du paramètre de modèle pour les constructeurs 194

Déduction de type de modèle 194

Déduction automatique du type 195

Chapitre 28: Déplacer la sémantique 198

Examples 198

Déplacer la sémantique 198

Déplacer constructeur 198

Déplacer la cession 200

Utiliser std :: move pour réduire la complexité de O (n²) à O (n) 201

Utilisation de la sémantique de déplacement sur les conteneurs 204

Réutiliser un objet déplacé 205

Chapitre 29: Des exceptions 206

Examples 206

Catching exceptions 206

Renvoyer (propager) une exception 207

Fonction Try Blocks In constructeur 208

Fonction Essayez Bloquer pour une fonction régulière 208

Fonction Try Blocks In destructor 209

Meilleure pratique: lancer par valeur, référence par const 209

Exception imbriquée 210

std :: uncaught_exceptions 212

Exception personnalisée 213

Chapitre 30: Disposition des types d'objet 217

Remarques 217

Examples 217

Types de classes 217

Types arithmétiques 220

Types de caractères étroits 220

Types entiers 220
Virgule flottante 220
Tableaux 221
Chapitre 31: Entrée / sortie basique en c ++ 222
Remarques 222
Examples 222
entrée utilisateur et sortie standard 222
Chapitre 32: Énumération 224
Examples 224
Déclaration de dénombrement de base 224
Enumération dans les instructions de commutateur 225
Itération sur un enum 225
Énumérés 226
Énumérer la déclaration en avant en C ++ 11 227
Chapitre 33: Erreurs courantes de compilation / édition de liens (GCC) 229
Examples 229
erreur: '***' n'a pas été déclaré dans cette portée 229
Les variables 229
Les fonctions 229
référence indéfinie à «***» 230
erreur fatale: ***: aucun fichier ou répertoire de ce type 231
Chapitre 34: Espaces de noms 232
Introduction 232
Syntaxe 232
Remarques 232
Examples 233
Que sont les espaces de noms? 233
Faire des espaces de noms 234
Extension des espaces de noms 235
Utiliser la directive 235
Recherche dépendante de l'argument 236
Quand l'ADL ne se produit pas 237

Espace de noms en ligne 237
Espace de noms anonyme / anonyme 239
Espaces de noms imbriqués compacts 240
Aliasing d'un long espace de noms 240
Portée de déclaration d'alias 240
Alias d'espace de noms 241
Chapitre 35: Exemples de serveur client 243
Examples 243
Bonjour TCP Server 243
Bonjour client TCP 246
Chapitre 36: Expressions régulières 248
Introduction 248
Syntaxe 248
Paramètres 248
Examples 249
Exemples de base de regex_match et regex_search 249
Exemple de regex_replace 249
Exemple avec regex_token_iterator 250
Exemple de regex_iterator 250
Fractionner une chaîne 251
Quantificateurs 251
Ancres 253
Chapitre 37: Fichier I / O 254
Introduction 254
Examples 254
Ouvrir un fichier 254
Lecture d'un fichier 255
Ecrire dans un fichier 257
Modes d'ouverture 258
Fermer un fichier 259
Flushing un flux 260
Lire un fichier ASCII dans une chaîne std :: 260

Lecture d'un fichier dans un conteneur 261
Lecture d'un `struct` à partir d'un fichier texte formaté 262
Copier un fichier 263
Vérification de la fin du fichier dans une condition de boucle, mauvaise pratique? 264
Ecriture de fichiers avec des paramètres régionaux non standard 265
Chapitre 38: Fichiers d'en-tête 267
Remarques 267
Examples 267
Exemple de base 267
Fichiers source 267
Le processus de compilation 269
Modèles dans les fichiers d'en-tête 269
Chapitre 39: Filetage 271
Syntaxe 271
Paramètres 271
Remarques 271
Examples 271
Opérations de filetage 271
Passer une référence à un fil 272
Créer un thread std :: 272
Opérations sur le thread en cours 274
Utiliser std :: async au lieu de std :: thread 276
Appel asynchrone d'une fonction 276
Pièges courants 276
S'assurer qu'un fil est toujours joint 276
Réaffectation des objets thread 277
Synchronisation de base 278
Utilisation de variables de condition 278
Créez un pool de threads simple 280
Stockage local 282
Chapitre 40: Flux C ++ 284
Remarques 284

Examples 284
Flux de chaînes 284
Lire un fichier jusqu'à la fin 285
Lecture d'un fichier texte ligne par ligne 285
Lignes sans caractères d'espacement 285
Lignes avec des caractères d'espacement 285
Lecture d'un fichier dans un tampon à la fois 286
Copier des flux 286
Tableaux 287
Impression de collections avec iostream 287
Impression de base 287
Type de distribution implicite 287
Génération et transformation 288
Tableaux 288
Fichiers d'analyse 289
Analyse des fichiers dans des conteneurs STL 289
Analyse de tables de texte hétérogènes 289
Transformation 290
Chapitre 41: Fonction C ++ "appel par valeur" vs. "appel par référence" 291
Introduction 291
Examples 291
Appel par valeur 291
Chapitre 42: Fonction de surcharge 293
Introduction 293
Remarques 293
Examples 293
Qu'est-ce que la surcharge de fonctions? 293
Type de retour en surcharge de fonction 295
Membre Fonction cv-qualifier Surcharge 295
Chapitre 43: Fonctions en ligne 298
Introduction 298

Syntaxe ... 298

Remarques ... 298

Inline comme directive de liaison ... 298

FAQ ... 298

Voir également ... 299

Examples ... 299

Déclaration de fonction en ligne non membre ... 299

Définition de fonction inline non membre ... 299

Fonctions inline membres ... 299

Qu'est-ce que la fonction inline? ... 300

Chapitre 44: Fonctions membres de classe constante ... 301

Remarques ... 301

Examples ... 301

fonction membre constante ... 301

Chapitre 45: Fonctions membres non statiques ... 303

Syntaxe ... 303

Remarques ... 303

Examples ... 303

Fonctions de membres non statiques ... 303

Encapsulation ... 304

Nom Cacher et importer ... 305

Fonctions membres virtuelles ... 307

Correct Correct ... 309

Chapitre 46: Fonctions membres spéciales ... 312

Examples ... 312

Destructeurs virtuels et protégés ... 312

Déplacement et copie implicites ... 313

Copier et échanger ... 313

Constructeur par défaut ... 315

Destructeur ... 317

Chapitre 47: Fonctions membres virtuelles ... 320

Syntaxe ... 320

Remarques.....320
Examples.....320
Utilisation de la substitution avec virtual en C ++ 11 et versions ultérieures.....320
Fonctions membres virtuelles et non virtuelles.....321
Fonctions virtuelles finales.....322
Comportement des fonctions virtuelles dans les constructeurs et les destructeurs.....323
Fonctions virtuelles pures.....324
Chapitre 48: Futures et promesses.....327
Introduction.....327
Examples.....327
std :: future et std :: promis.....327
Exemple asynchrone différé.....327
std :: packaged_task et std :: future.....328
std :: future_error et std :: future_errc.....328
std :: future et std :: async.....329
Classes d'opération asynchrones.....331
Chapitre 49: Génération de nombres aléatoires.....332
Remarques.....332
Examples.....332
Véritable générateur de valeur aléatoire.....332
Générer un nombre pseudo-aléatoire.....333
Utilisation du générateur pour plusieurs distributions.....334
Chapitre 50: Gestion de la mémoire.....335
Syntaxe.....335
Remarques.....335
Examples.....335
Empiler.....335
Stockage gratuit (tas, allocation dynamique ...).....336
Nouveau placement.....338
Chapitre 51: Implémentation du modèle de conception en C ++.....340
Introduction.....340
Remarques.....340

Examples....340
Motif d'observateur....340
Modèle d'adaptateur....343
Modèle d'usine....345
Modèle de générateur avec API Fluent....346
Passer le constructeur autour....348
Variante de conception: objet Mutable....349
Chapitre 52: Internationalisation en C ++....350
Remarques....350
Examples....350
Comprendre les caractéristiques de la chaîne C ++....350
Chapitre 53: Itération....352
Examples....352
Pause....352
continuer....352
faire....352
pour....352
tandis que....353
basé sur la plage pour la boucle....353
Chapitre 54: La gestion des ressources....354
Introduction....354
Examples....354
L'acquisition de ressources est une initialisation....354
Mutexes et sécurité des fils....355
Chapitre 55: La norme ISO C ++....357
Introduction....357
Remarques....357
Examples....358
Projets de travail actuels....358
C ++ 11....358
Extensions de langue....358
Caractéristiques générales....358

Des classes 359
Autres types 359
Modèles 359
Concurrence 359
Fonctions linguistiques diverses 359
Extensions de bibliothèque 360
Général 360
Conteneurs et Algorithmes 360
Concurrence 360
C ++ 14 360
Extensions de langue 361
Extensions de bibliothèque 361
Déconseillé / Supprimé 361
C ++ 17 361
Extensions de langue 361
Extensions de bibliothèque 362
C ++ 03 362
Extensions de langue 362
C ++ 98 362
Extensions linguistiques (en ce qui concerne C89 / C90) 362
Extensions de bibliothèque 363
C ++ 20 363
Extensions de langue 363
Extensions de bibliothèque 363
Chapitre 56: La règle des trois, cinq et zéro 364
Examples 364
Règle de cinq 364
Règle de zéro 365
Règle de trois 366
Protection d'auto-assignation 368
Chapitre 57: Lambdas 370

Syntaxe 370
Paramètres 370
Remarques 371
Examples 371
Qu'est-ce qu'une expression lambda? 371
Spécification du type de retour 374
Capturer par valeur 375
Capture généralisée 376
Capturer par référence 377
Capture par défaut 378
Lambdas génériques 378
Conversion en pointeur de fonction 379
Classe lambda et capture de cette 380
Portage des fonctions lambda en C ++ 03 à l'aide de foncteurs 382
Lambda récursive 383
Utilisez std::function 383
En utilisant deux pointeurs intelligents: 383
Utiliser un combinateur Y 384
Utilisation de lambdas pour le déballage du pack de paramètres en ligne 385
Chapitre 58: Le pointeur 387
Remarques 387
Examples 387
ce pointeur 387
Utiliser le pointeur this pour accéder aux données des membres 389
Utiliser le pointeur this pour différencier les données de membre et les paramètres 390
ce Pointer CV-Qualifiers 391
ce Pointer Ref-Qualifiers 394
Chapitre 59: Les attributs 396
Syntaxe 396
Examples 396
[[non-retour]] 396
[[tomber dans]] 397

[[déconseillé]] et [[déconseillé ("raison")]] 398
[[nodiscard]] 399
[[peut-être_unused]] 399
Chapitre 60: Les itérateurs 401
Examples 401
C itérateurs (pointeurs) 401
Le briser 401
Vue d'ensemble 402
Les itérateurs sont des positions 402
Des itérateurs aux valeurs 402
Itérateurs non valides 404
Naviguer avec les itérateurs 404
Concepts d'itérateur 405
Traits d'itérateur 405
Itérateurs inverses 406
Itérateur de vecteur 407
Itérateur de carte 407
Itérateurs de flux 408
Ecrivez votre propre itérateur avec générateur 408
Chapitre 61: Les portées 410
Examples 410
Étendue de bloc simple 410
Variables globales 410
Chapitre 62: Les références 412
Examples 412
Définir une référence 412
Les références C ++ sont des alias de variables existantes 412
Chapitre 63: Les syndicats 414
Remarques 414
Examples 414
Fonctions de base de l'Union 414

Utilisation typique....414
Comportement non défini....415
Chapitre 64: Littéraux....416
Introduction....416
Examples....416
vrai....416
faux....416
nullptr....416
ce....417
Littéral entier....417
Chapitre 65: Littéraux définis par l'utilisateur....420
Examples....420
Littéraux définis par l'utilisateur avec de longues valeurs doubles....420
Littéraux standard définis par l'utilisateur pour la durée....420
Littéraux standard définis par l'utilisateur pour les chaînes....421
Littéraux standard définis par l'utilisateur pour complexes....421
Littéral auto-créé défini par l'utilisateur pour binaire....422
Chapitre 66: Manipulateurs de flux....424
Introduction....424
Remarques....424
Examples....425
Manipulateurs de flux....426
Manipulateurs de flux de sortie....432
Manipulateurs de flux d'entrée....433
Chapitre 67: Métaprogrammation....435
Introduction....435
Remarques....435
Examples....435
Calcul des factoriels....435
Itération sur un paquet de paramètres....438
Itération avec std :: integer_sequence....439
Expédition de tag....440

Détecter si l'expression est valide .441
Calcul de la puissance avec C ++ 11 (et supérieur) .442
Distinction manuelle des types avec n'importe quel type T .443
If-then-else .444
Min / Max générique avec nombre d'arguments variable .444
Chapitre 68: Métaprogrammation arithmitique .446
Introduction .446
Examples .446
Calcul de la puissance en O (log n) .446
Chapitre 69: Modèle de mémoire C ++ 11 .448
Remarques .448
Opérations atomiques .448
Consistance séquentielle .449
Commande détendue .449
Validation des commandes .449
Commande de sortie-consommation .450
Clôtures .450
Examples .450
Besoin d'un modèle de mémoire .450
Exemple de clôture .452
Chapitre 70: Modèle de modèle curieusement récurrent (CRTP) .454
Introduction .454
Examples .454
Le modèle de modèle curieusement récurrent (CRTP) .454
CRTP pour éviter la duplication de code .456
Chapitre 71: Modèles .458
Introduction .458
Syntaxe .458
Remarques .458
Examples .460
Modèles de fonction .460

Transmission d'argument 461
Modèle de classe de base 462
Spécialisation de template 463
Spécialisation du modèle partiel 464
Valeur du paramètre de modèle par défaut 465
Modèle d'alias 466
Paramètres du modèle de modèle 466
Déclaration des arguments de modèle non-type avec auto 467
Vide deleter personnalisé pour unique_ptr 467
Paramètre de type non-type 468
Structures de données du modèle Variadic 469
Instanciation explicite 472
Chapitre 72: Modèles d'expression 474
Examples 474
Modèles d'expression de base sur des expressions algébriques élémentaires 474
Fichier vec.hh: wrapper pour std :: vector, utilisé pour afficher le journal lorsqu'une co 476
Fichier expr.hh: implémentation de modèles d'expression pour les opérations élémentaires (..... 477
Fichier main.cc: test du fichier src 481
Un exemple de base illustrant des modèles d'expression 483
Chapitre 73: Mot clé ami 488
Introduction 488
Examples 488
Fonction ami 488
Méthode d'ami 489
Classe d'ami 489
Chapitre 74: mot clé const 491
Syntaxe 491
Remarques 491
Examples 491
Variables locales const 491
Pointeurs Const 492
Fonctions de membre Const 492

Éviter la duplication du code dans les méthodes get const et non const....492

Chapitre 75: mot-clé mutable....495

Examples....495

modificateur de membre de classe non statique....495

lambda mutable....495

Chapitre 76: Mots clés....497

Introduction....497

Syntaxe....497

Remarques....497

Examples....499

asm....499

explicite....500

pas d'exception....500

nom_type....502

taille de....502

Mots-clés différents....503

Chapitre 77: Mots-clés de déclaration de variable....508

Examples....508

const....508

decltype....508

signé....509

non signé....509

volatil....510

Chapitre 78: Mots-clés de type....511

Examples....511

classe....511

struct....512

enum....512

syndicat....514

Chapitre 79: Mots-clés de type de base....515

Examples....515

int....515

bool 515
carboniser 515
char16_t 515
char32_t 516
flotte 516
double 516
longue 516
court 517
vide 517
wchar_t 517
Chapitre 80: Mutex récursif 519
Examples 519
std :: recursive_mutex 519
Chapitre 81: Mutexes 520
Remarques 520
Il est préférable d'utiliser std :: shared_mutex que std :: shared_timed_mutex . 520
Le code ci-dessous est l'implémentation MSVC14.1 de std :: shared_mutex. 520
Le code ci-dessous est l'implémentation MSVC14.1 de std :: shared_timed_mutex. 522
std :: shared_mutex traité en lecture / écriture plus de 2 fois plus que std :: shared_tim 525
Examples 528
std :: unique_lock, std :: shared_lock, std :: lock_guard 528
Stratégies pour les classes de verrouillage: std :: try_to_lock, std :: adopt_lock, std :: 529
std :: mutex 530
std :: scoped_lock (C ++ 17) 531
Types de mutex 531
std :: lock 531
Chapitre 82: Objets appelables 532
Introduction 532
Remarques 532
Examples 532
Pointeurs de fonction 532
Classes avec opérateur () (Functors) 533

Chapitre 83: Opérateurs de bits 534
Remarques 534
Examples 534
& - bitwise AND 534
| - bit à bit OU 535
^ - bit à bit XOR (OU exclusif) 535
~ - bitwise NOT (complément unaire) 537
<< - décalage gauche 538
>> - décalage vers la droite 539
Chapitre 84: Optimisation 540
Introduction 540
Examples 540
Expansion Inline / Inlining 540
Optimisation de la base vide 540
Chapitre 85: Optimisation en C ++ 542
Examples 542
Optimisation de la classe de base vide 542
Introduction à la performance 542
Optimiser en exécutant moins de code 543
Supprimer le code inutile 543
Faire du code une seule fois 543
Prévenir les réaffectations inutiles et la copie / déplacement 544
Utiliser des conteneurs efficaces 544
Optimisation des petits objets 545
Exemple 545
Quand utiliser? 546
Chapitre 86: Outils et techniques de débogage et de prévention du débogage C ++ 548
Introduction 548
Remarques 548
Examples 548
Mon programme C ++ se termine par segfault - valgrind 548
Analyse Segfault avec GDB 550

Code propre....551
L'utilisation de fonctions séparées pour des actions séparées....552
Utiliser des mises en forme / constructions cohérentes....553
Faites attention aux parties importantes de votre code....553
Conclusion....553
Analyse statique....553
Avertissements du compilateur....554
Outils externes....554
Autres outils....555
Conclusion....555
Safe-stack (corruptions de piles)....555
Quelles parties de la pile sont déplacées?....555
A quoi sert-il réellement?....556
Comment l'activer?....556
Conclusion....556
Chapitre 87: Pack de paramètres....557
Examples....557
Un modèle avec un pack de paramètres....557
Extension d'un pack de paramètres....557
Chapitre 88: Pimpl Idiom....558
Remarques....558
Examples....558
Idiome de base de Pimpl....558
Chapitre 89: Plier les expressions....560
Remarques....560
Examples....560
Unary Folds....560
Plis binaires....561
Plier sur une virgule....561
Chapitre 90: Pointeurs....563
Introduction....563

Syntaxe 563
Remarques 563
Examples 563
Notions de base sur les pointeurs 563
Créer une variable de pointeur 563
Prendre l'adresse d'une autre variable 564
Accéder au contenu d'un pointeur 565
Déréférencer les pointeurs invalides 565
Opérations de pointeur 566
Arithmétique du pointeur 567
Incrémenter / Décrémenter 567
Addition soustraction 567
Différence de pointeur 567
Chapitre 91: Pointeurs aux membres 569
Syntaxe 569
Examples 569
Pointeurs vers des fonctions membres statiques 569
Pointeurs vers les fonctions membres 570
Pointeurs vers les variables membres 570
Pointeurs vers des variables membres statiques 571
Chapitre 92: Pointeurs intelligents 573
Syntaxe 573
Remarques 573
Examples 573
Partage de propriété (std :: shared_ptr) 573
Partage avec propriété temporaire (std :: faiblesse_ptr) 576
Propriété unique (std :: unique_ptr) 577
Utilisation de paramètres personnalisés pour créer un wrapper vers une interface C 580
Propriété unique sans sémantique de déplacement (auto_ptr) 581
Obtenir un shared_ptr faisant référence à ceci 583
Casting std :: shared_ptr pointeurs 584

Ecrire un pointeur intelligent: value_ptr....584
Chapitre 93: Polymorphisme....587
Examples....587
Définir des classes polymorphes....587
Downcasting sûr....588
Polymorphisme & Destructeurs....590
Chapitre 94: Préprocesseur....591
Introduction....591
Remarques....591
Examples....591
Inclure les gardes....591
Logique conditionnelle et gestion multi-plateforme....592
Macros....594
Messages d'erreur de préprocesseur....598
Macros prédéfinies....598
X-macros....600
#pragma une fois....602
Opérateurs de préprocesseur....602
Chapitre 95: priorité de l'opérateur....604
Remarques....604
Examples....604
Opérateurs arithmétiques....605
Opérateurs logiques ET et OU....605
Logique && et || opérateurs: court-circuit....605
Opérateurs Unaires....606
Chapitre 96: Profilage....608
Examples....608
Profilage avec gcc et gprof....608
Générer des diagrammes callgraph avec gperf2dot....609
Profilage de l'utilisation du processeur avec les outils gcc et Google Perf....610
Chapitre 97: RAII: l'acquisition de ressources est une initialisation....613
Remarques....613

Examples 613
Verrouillage 613
Enfin / ScopeExit 614
ScopeSuccess (c ++ 17) 615
ScopeFail (c ++ 17) 616
Chapitre 98: Recherche de nom dépendante de l'argument 619
Examples 619
Quelles fonctions sont trouvées 619
Chapitre 99: Récursivité en C ++ 621
Examples 621
Utilisation de la récursion de la queue et de la récursion de style Fibonnaci pour résoudr 621
Récursivité avec mémo 621
Chapitre 100: Renvoyer plusieurs valeurs d'une fonction 623
Introduction 623
Examples 623
Utilisation des paramètres de sortie 623
Utiliser std :: tuple 624
Utiliser std :: array 625
Utiliser std :: pair 625
En utilisant struct 626
Fixations structurées 627
Utilisation d'un objet de consommation 628
Utiliser std :: vector 629
Utilisation de l'itérateur de sortie 630
Chapitre 101: Résolution de surcharge 631
Remarques 631
Examples 631
Correspondance exacte 631
Catégorisation de l'argument au coût du paramètre 632
Recherche de nom et contrôle d'accès 633
Surcharge sur la référence de transfert 633
Étapes de la résolution de surcharge 634

Promotions arithmétiques et conversions 636
Surcharge dans une hiérarchie de classes 637
Surcharge de constance et de volatilité 638
Chapitre 102: RTTI: Informations sur le type d'exécution 640
Examples 640
Nom d'un type 640
dynamic_cast 640
Le mot-clé de typeid 640
Quand utiliser qui jette en c ++ 641
Chapitre 103: Sémantique de valeur et de référence 642
Examples 642
Copie en profondeur et support de déplacement 642
Définitions 644
Chapitre 104: Sémaphore 646
Introduction 646
Examples 646
Sémaphore C ++ 11 646
Classe de sémaphore en action 646
Chapitre 105: Séparateurs de chiffres 648
Examples 648
Séparateur de chiffres 648
Chapitre 106: SFINAE (échec de substitution n'est pas une erreur) 650
Examples 650
enable_if 650
Quand l'utiliser 650
void_t 652
le decltype de fin dans les modèles de fonction 653
Qu'est-ce que SFINAE? 654
enable_if_all / enable_if_any 655
est détecté 657
Résolution de surcharge avec un grand nombre d'options 658

Chapitre 107: Side by Side Comparaisons des exemples classiques en C ++ résolus via C ++ v 660
Examples 660
Boucler à travers un conteneur 660
Chapitre 108: Singleton Design Pattern 662
Remarques 662
Examples 662
Initialisation paresseuse 662
Des sous-classes 663
Filet Singeton 664
Singleton de désinitialisation statique 665
Chapitre 109: Spécificateurs de classe de stockage 666
Introduction 666
Remarques 666
Examples 666
mutable 666
registre 667
statique 667
auto 668
externe 669
Chapitre 110: Spécifications de liaison 671
Introduction 671
Syntaxe 671
Remarques 671
Examples 671
Gestionnaire de signal pour un système d'exploitation de type Unix 671
Rendre un en-tête de bibliothèque C compatible avec C ++ 671
Chapitre 111: static_assert 673
Syntaxe 673
Paramètres 673
Remarques 673
Examples 673

static_assert 673

Chapitre 112: std :: any 675

Remarques 675

Examples 675

Utilisation de base 675

Chapitre 113: std :: array 676

Paramètres 676

Remarques 676

Examples 676

Initialisation d'un tableau std :: 676

Accès aux éléments 677

Vérification de la taille du tableau 679

Itérer à travers le tableau 680

Changer tous les éléments du tableau à la fois 680

Chapitre 114: std :: atomics 681

Examples 681

types atomiques 681

Chapitre 115: std :: carte 684

Remarques 684

Examples 684

Accès aux éléments 684

Initialisation d'un std :: map ou std :: multimap 685

Supprimer des éléments 686

Insertion d'éléments 687

Itération sur std :: map ou std :: multimap 688

Recherche dans std :: map ou dans std :: multimap 689

Vérification du nombre d'éléments 690

Types de cartes 690

Carte régulière 690

Multi-Map 691

Hash-Map (carte non ordonnée) 691

Création de std :: map avec les types définis par l'utilisateur comme clé 691

Strict Commande Faible ... 692
Chapitre 116: std :: forward_list ... 693
Introduction ... 693
Remarques ... 693
Examples ... 693
Exemple ... 693
Les méthodes ... 694
Chapitre 117: std :: function: Pour envelopper n'importe quel élément appelable ... 696
Examples ... 696
Usage simple ... 696
std :: function utilisé avec std :: bind ... 696
std :: function avec lambda et std :: bind ... 697
frais généraux de la fonction ... 698
Liaison std :: function à un autre type appelable ... 699
Stocker des arguments de fonction dans std :: tuple ... 701
Chapitre 118: std :: integer_sequence ... 703
Introduction ... 703
Examples ... 703
Tournez un std :: tuple en paramètres de fonction ... 703
Créer un pack de paramètres composé d'entiers ... 704
Transforme une séquence d'indices en copies d'un élément ... 704
Chapitre 119: std :: iomanip ... 706
Examples ... 706
std :: setw ... 706
std :: setprecision ... 706
std :: setfill ... 707
std :: setiosflags ... 707
Chapitre 120: std :: optionnel ... 709
Examples ... 709
introduction ... 709
Autres approches optionnelles ... 709
Facultatif vs pointeur ... 709

Facultatif vs Sentinel 709
Facultatif vs std::pair<bool, T> 709
Utiliser des options pour représenter l'absence de valeur 709
Utiliser des options pour représenter l'échec d'une fonction 710
optionnel comme valeur de retour 711
valeur_ou 712
Chapitre 121: std :: pair 714
Examples 714
Créer une paire et accéder aux éléments 714
Comparer les opérateurs 714
Chapitre 122: std :: set et std :: multiset 716
Introduction 716
Remarques 716
Examples 716
Insérer des valeurs dans un ensemble 716
Insertion de valeurs dans un multiset 717
Changer le type de jeu par défaut 718
Tri par défaut 719
Tri personnalisé 719
Type Lambda 720
Autres options de tri 720
Recherche de valeurs dans set et multiset 720
Suppression de valeurs d'un ensemble 721
Chapitre 123: std :: string 723
Introduction 723
Syntaxe 723
Remarques 724
Examples 724
Scission 724
Remplacement de cordes 725
Remplacer par la position 725
Remplacer les occurrences d'une chaîne par une autre chaîne 726

Enchaînement.....726
Accéder à un personnage.....727
opérateur [] (n).....727
en (n).....727
de face().....727
arrière().....728
Tokenize.....728
Conversion en (const) char *.....729
Recherche de caractère (s) dans une chaîne.....730
Découpe des caractères au début / à la fin.....731
Comparaison lexicographique.....732
Conversion en std :: wstring.....733
Utiliser la classe std :: string_view.....734
En boucle à travers chaque personnage.....735
Conversion en entiers / types à virgule flottante.....735
Conversion entre les encodages de caractères.....737
Vérifier si une chaîne est un préfixe d'un autre.....737
Conversion en std :: string.....738
Chapitre 124: std :: variant.....740
Remarques.....740
Examples.....740
Utilisation de base std :: variant.....740
Créer des pointeurs de pseudo-méthode.....741
Construire un `std :: variant`.....742
Chapitre 125: std :: vector.....743
Introduction.....743
Remarques.....743
Examples.....743
Initialisation d'un std :: vector.....743
Insertion d'éléments.....744
Itération sur std :: vector.....746

Itération dans la direction avant 746
Itération dans le sens inverse 746
Application d'éléments const 747
Une note sur l'efficacité 748
Accéder aux éléments 749
Accès par index: 749
Itérateurs: 751
Utiliser std :: vector comme un tableau C 752
Invalidation de l'itérateur / pointeur 752
Supprimer des éléments 753
Supprimer le dernier élément: 753
Supprimer tous les éléments: 753
Supprimer un élément par index: 754
Supprimer tous les éléments d'une plage: 754
Suppression d'éléments par valeur: 754
Suppression d'éléments par condition: 754
Supprimer des éléments par lambda, sans créer de fonction de prédicat supplémentaire 755
Suppression d'éléments par condition à partir d'une boucle: 755
Suppression d'éléments par condition à partir d'une boucle inverse: 755
Recherche d'un élément dans std :: vector 756
Conversion d'un tableau en std :: vector 758
vecteur : L'exception à tant de règles, tant de règles 758
Taille et capacité du vecteur 760
Vecteurs concaténants 761
Réduire la capacité d'un vecteur 762
Utilisation d'un vecteur trié pour la recherche rapide d'éléments 763
Fonctions de retour de grands vecteurs 764
Rechercher max et min Element et index respectif dans un vecteur 765
Matrices utilisant des vecteurs 766
Chapitre 126: Structures de données en C ++ 768

Examples......768
Implémentation de la liste liée en C ++......768
Chapitre 127: Structures de synchronisation de fil......771
Introduction......771
Examples......771
std :: shared_lock......771
std :: call_once, std :: once_flag......771
Verrouillage d'objet pour un accès efficace......772
std :: condition_variable_any, std :: cv_status......773
Chapitre 128: Surcharge de l'opérateur......774
Introduction......774
Remarques......774
Examples......774
Opérateurs arithmétiques......774
Opérateurs unaires......776
Opérateurs de comparaison......777
Opérateurs de conversion......778
Opérateur d'indice de tableau......779
Opérateur d'appel de fonction......780
Opérateur d'assignation......781
Opérateur binaire NON......782
Opérateurs de décalage de bits pour E / S......782
Nombres complexes revisités......783
Opérateurs nommés......787
Chapitre 129: Surcharge du modèle de fonction......790
Remarques......790
Examples......790
Qu'est-ce qu'une surcharge de modèle de fonction valide?......790
Chapitre 130: Systèmes de construction......792
Introduction......792
Remarques......792
Examples......792

Générer un environnement de construction avec CMake 792
Compiler avec GNU make 793
introduction 793
Règles de base 793
Constructions incrémentielles 795
Documentation 795
Construire avec des SCons 796
Ninja 796
introduction 796
NMAKE (utilitaire de maintenance de programme Microsoft) 797
introduction 797
Autotools (GNU) 797
introduction 797
Chapitre 131: Tableaux 799
Introduction 799
Examples 799
Taille du tableau: tapez safe au moment de la compilation. 799
Tableau brut de taille dynamique 800
Extension du tableau de taille dynamique en utilisant std :: vector. 801
Une matrice de matrice brute de taille fixe (c'est-à-dire un tableau brut 2D). 802
Une matrice de taille dynamique utilisant std :: vector pour le stockage. 803
Initialisation du tableau 805
Chapitre 132: Techniques de refactoring 807
Introduction 807
Examples 807
Refactoring à pied 807
Aller au nettoyage 809
Chapitre 133: Test d'unité en C ++ 811
Introduction 811
Examples 811
Test Google 811

Exemple minimal 811
Capture 812
Chapitre 134: Transmission parfaite 814
Remarques 814
Examples 814
Fonctions d'usine 814
Chapitre 135: Tri 816
Remarques 816
Examples 816
Tri des conteneurs de séquence avec un ordre spécifique 816
Tri des conteneurs de séquence par un opérateur moins surchargé 816
Tri des conteneurs de séquence à l'aide de la fonction de comparaison 817
Tri des conteneurs de séquence à l'aide d'expressions lambda (C ++ 11) 818
Tri et séquence des conteneurs 819
tri avec std :: map (croissant et décroissant) 820
Tri des tableaux intégrés 822
Chapitre 136: Type de retour 823
Syntaxe 823
Remarques 823
Examples 823
Évitez de qualifier un nom de type imbriqué 823
Expressions lambda 823
Chapitre 137: Type de retour Covariance 825
Remarques 825
Examples 825
1. Exemple de base sans retours covariants, montre pourquoi ils sont souhaitables 825
2. Version du résultat de covariant de l'exemple de base, vérification de type statique 826
3. Résultat du pointeur intelligent covariant (nettoyage automatique) 827
Chapitre 138: Type effacement 829
Introduction 829
Examples 829
Mécanisme de base 829

Effacement à un type régulier avec une vtable manuelle 830
Un simple `std :: function` 833
Effacement à un tampon contigu de T 835
Type effacement type effacement avec std :: any 837
Chapitre 139: Type Inférence 843
Introduction 843
Remarques 843
Examples 843
Type de données: Auto 843
Auto lambda 843
Boucles et auto 844
Chapitre 140: Type Traits 845
Remarques 845
Examples 845
Traits de type standard 845
Les constantes 845
Les fonctions 845
Les types 846
Tapez les relations avec std :: is_same 846
Traits de type fondamentaux 847
Propriétés du type 848
Chapitre 141: Typedef et alias de type 850
Introduction 850
Syntaxe 850
Examples 850
Syntaxe de base de typedef 850
Utilisations plus complexes du typedef 851
Déclarer plusieurs types avec typedef 851
Déclaration d'alias avec "using" 851
Chapitre 142: Types atomiques 853
Syntaxe 853

Remarques 853
Examples 853
Accès multithread 853
Chapitre 143: Types sans nom 855
Examples 855
Classes sans nom 855
Membres anonymes 855
Comme alias de type 856
Union anonyme 856
Chapitre 144: Une règle de définition (ODR) 857
Examples 857
Multiplier la fonction définie 857
Fonctions en ligne 857
Violation d'ODR via une résolution de surcharge 859
Chapitre 145: Utiliser la déclaration 860
Introduction 860
Syntaxe 860
Remarques 860
Examples 860
Importation de noms individuellement à partir d'un espace de noms 860
Redéclarer les membres d'une classe de base pour éviter de les masquer 860
Héritage des constructeurs 861
Chapitre 146: Utiliser std :: unordered_map 862
Introduction 862
Remarques 862
Examples 862
Déclaration et utilisation 862
Quelques fonctions de base 862
Chapitre 147: Variables en ligne 864
Introduction 864
Examples 864
Définition d'un membre de données statique dans la définition de classe 864

Crédits 865

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cplusplus

It is an unofficial and free C++ ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C++.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec C ++

Remarques

Le programme 'Hello World' est un exemple courant qui peut être simplement utilisé pour vérifier la présence du compilateur et de la bibliothèque. Il utilise la bibliothèque standard C ++, avec `std::cout` de `<iostream>` , et ne dispose que d'un seul fichier à compiler, ce qui réduit les risques d'erreurs lors de la compilation.

Le processus de compilation d'un programme C ++ diffère intrinsèquement entre les compilateurs et les systèmes d'exploitation. La rubrique Compilation et construction contient les détails sur la façon de compiler du code C ++ sur différentes plates-formes pour différents compilateurs.

Versions

Version	la norme	Date de sortie
C ++ 98	ISO / IEC 14882: 1998	1998-09-01
C ++ 03	ISO / IEC 14882: 2003	2003-10-16
C ++ 11	ISO / IEC 14882: 2011	2011-09-01
C ++ 14	ISO / IEC 14882: 2014	2014-12-15
C ++ 17	À déterminer	2017-01-01
C ++ 20	À déterminer	2020-01-01

Examples

Bonjour le monde

Ce programme imprime `Hello World!` au flux de sortie standard:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

Voyez-le vivre sur Coliru .

Une analyse

Examinons chaque partie de ce code en détail:

- `#include <iostream>` est une **directive de préprocesseur** qui inclut le contenu du fichier d'en-tête C ++ standard `iostream` .

  `iostream` est un **fichier d'en-tête de bibliothèque standard** qui contient les définitions des flux d'entrée et de sortie standard. Ces définitions sont incluses dans l'espace de noms `std` , expliqué ci-dessous.

  Les **flux d'entrée / sortie (E / S) standard** permettent aux programmes d'obtenir des entrées et des sorties vers un système externe - généralement le terminal.

- `int main() { ... }` définit une nouvelle fonction nommée `main` . Par convention, la fonction `main` est appelée lors de l'exécution du programme. Il ne doit y avoir qu'une seule fonction `main` dans un programme C ++, et il doit toujours renvoyer un numéro du type `int` .

  Ici, l' `int` est ce qu'on appelle le type de retour de la fonction. La valeur renvoyée par la fonction `main` est un **code de sortie.**

  Par convention, un code de sortie de programme de `0` ou `EXIT_SUCCESS` est interprété comme un succès par un système qui exécute le programme. Tout autre code de retour est associé à une erreur.

  Si aucune instruction de `return` n'est présente, la fonction `main` (et donc le programme lui-même) renvoie `0` par défaut. Dans cet exemple, nous n'avons pas besoin d'écrire explicitement `return 0;` .

  Toutes les autres fonctions, sauf celles qui renvoient le type `void` , doivent explicitement renvoyer une valeur en fonction de leur type de retour, sinon elles ne doivent pas être renvoyées du tout.

- `std::cout << "Hello World!" << std::endl;` imprime "Hello World!" au flux de sortie standard:

  - `std` est un espace de noms , et `::` est l' **opérateur de résolution de portée** qui permet de rechercher des objets par nom dans un espace de noms.

    Il y a beaucoup d'espaces de noms. Ici, nous utilisons `::` pour montrer que nous voulons utiliser `cout` depuis l'espace de noms `std` . Pour plus d'informations, reportez-vous à Opérateur Résolution de portée - Documentation Microsoft .

  - `std::cout` est l'objet de **flux de sortie standard** , défini dans `iostream` , et il imprime sur la sortie standard (`stdout`).

  - `<<` est, *dans ce contexte* , l' **opérateur d'insertion de flux** , ainsi appelé car il *insère* un objet dans l'objet *flux* .

    La bibliothèque standard définit l'opérateur `<<` permettant d'insérer des données pour

certains types de données dans les flux de sortie. `stream << content` insère du `content` dans le flux et renvoie le même flux, mais mis à jour. Cela permet de chaîner les insertions de flux: `std::cout << "Foo" << " Bar";` imprime "FooBar" sur la console.

- `"Hello World!"` est un **littéral de chaîne de caractères** ou un "littéral de texte". L'opérateur d'insertion de flux pour les littéraux de chaîne de caractères est défini dans le fichier `iostream` .

- `std::endl` est un objet **manipulateur de flux d'E / S** spécial, également défini dans le fichier `iostream` . L'insertion d'un manipulateur dans un flux modifie l'état du flux.

  Le flux manipulateur `std::endl` fait deux choses: d'abord il insère le caractère de fin de ligne, puis il vide le tampon de flux pour forcer le texte à apparaître sur la console. Cela garantit que les données insérées dans le flux apparaissent réellement sur votre console. (Les données de flux sont généralement stockées dans un tampon puis "vidées" par lots, sauf si vous forcez un vidage immédiat.)

  Une autre méthode qui évite le flush est:

  ```
  std::cout << "Hello World!\n";
  ```

  où `\n` est la **séquence d'échappement de caractère** pour le caractère de nouvelle ligne.

- Le point-virgule (`;`) informe le compilateur qu'une instruction est terminée. Toutes les instructions et définitions de classe C ++ nécessitent un point-virgule de fin / de fin.

commentaires

Un **commentaire** est un moyen de placer du texte arbitraire dans le code source sans que le compilateur C ++ l'interprète avec une signification fonctionnelle. Les commentaires permettent de mieux comprendre la conception ou la méthode d’un programme.

Il existe deux types de commentaires en C ++:

Commentaires sur une seule ligne

La double séquence de barre oblique avant `//` marquera tout le texte jusqu'à une nouvelle ligne en tant que commentaire:

```
int main()
{
   // This is a single-line comment.
   int a;  // this also is a single-line comment
   int i;  // this is another single-line comment
}
```

C-Style / Bloc Commentaires

La séquence `/*` est utilisée pour déclarer le début du bloc de commentaires et la séquence `*/` est utilisée pour déclarer la fin du commentaire. Tout le texte entre les séquences de début et de fin est interprété comme un commentaire, même si le texte est une syntaxe C ++ valide. Ceux-ci sont parfois appelés "style C" commentaires, car cette syntaxe de commentaire est héritée du langage prédécesseur de C ++, C:

```
int main()
{
   /*
    *  This is a block comment.
    */
   int a;
}
```

Dans tout commentaire de bloc, vous pouvez écrire tout ce que vous voulez. Lorsque le compilateur rencontre le symbole `*/` , il termine le commentaire de bloc:

```
int main()
{
   /* A block comment with the symbol /*
      Note that the compiler is not affected by the second /*
      however, once the end-block-comment symbol is reached,
      the comment ends.
   */
   int a;
}
```

L'exemple ci-dessus est un code C ++ (et C) valide. Cependant, le fait d’avoir des `/*` supplémentaires dans un commentaire de bloc peut entraîner un avertissement sur certains compilateurs.

Les commentaires de bloc peuvent également commencer et se terminer *sur* une seule ligne. Par exemple:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Importance des commentaires

Comme avec tous les langages de programmation, les commentaires offrent plusieurs avantages:

- Documentation explicite du code pour faciliter la lecture / maintenance
- Explication du but et de la fonctionnalité du code
- Détails sur l'historique ou le raisonnement derrière le code
- Placement des droits d'auteur / licences, notes de projet, remerciements spéciaux, crédits contributeurs, etc. directement dans le code source.

Cependant, les commentaires ont aussi leurs inconvénients:

- Ils doivent être maintenus pour refléter toute modification du code
- Les commentaires excessifs ont tendance à rendre le code *moins* lisible

Le besoin de commentaires peut être réduit en écrivant un code clair et auto-documenté. Un exemple simple est l'utilisation de noms explicatifs pour les variables, les fonctions et les types. La prise en compte des tâches liées logiquement dans des fonctions distinctes va de pair avec cela.

Marqueurs de commentaires utilisés pour désactiver le code

Pendant le développement, les commentaires peuvent également être utilisés pour désactiver rapidement des parties de code sans les supprimer. Ceci est souvent utile à des fins de test ou de débogage, mais ce n'est pas un bon style pour autre chose que des modifications temporaires. C'est ce que l'on appelle souvent «commenter».

De même, conserver les anciennes versions d'un code dans un commentaire à des fins de référence est mal vu car il encombre les fichiers tout en offrant peu de valeur par rapport à l'exploration de l'historique du code via un système de gestion des versions.

Fonction

Une **fonction** est une unité de code qui représente une séquence d'instructions.

Les fonctions peuvent accepter des **arguments** ou des valeurs et **renvoyer** une seule valeur (ou non). Pour utiliser une fonction, un **appel de fonction** est utilisé sur des valeurs d'argument et l'utilisation de l'appel de fonction lui-même est remplacée par sa valeur de retour.

Chaque fonction a une **signature de type** - les types de ses arguments et le type de son type de retour.

Les fonctions sont inspirées par les concepts de la procédure et de la fonction mathématique.

- Note: Les fonctions C ++ sont essentiellement des procédures et ne suivent pas la définition exacte ou les règles des fonctions mathématiques.

Les fonctions sont souvent destinées à effectuer une tâche spécifique. et peut être appelé à partir d'autres parties d'un programme. Une fonction doit être déclarée et définie avant d'être appelée ailleurs dans un programme.

- Remarque: les définitions de fonctions populaires peuvent être masquées dans d’autres fichiers inclus (souvent pour des raisons de commodité et de réutilisation dans de nombreux fichiers). Ceci est une utilisation courante des fichiers d'en-tête.

Déclaration de fonction

Une **déclaration de fonction** déclare l'existence d'une fonction avec son nom et sa signature au compilateur. La syntaxe est la suivante:

```
int add2(int i); // The function is of the type (int) -> (int)
```

Dans l'exemple ci-dessus, la fonction `int add2(int i)` déclare ce qui suit au compilateur:

- Le **type de retour** est `int` .
- Le **nom** de la fonction est `add2` .
- Le **nombre d'arguments** de la fonction est 1:
  - Le premier argument est du type `int` .
  - Le premier argument sera appelé dans le contenu de la fonction par le nom `i` .

Le nom de l'argument est facultatif. la déclaration pour la fonction pourrait également être la suivante:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Selon la **règle à** une **définition** , une fonction avec une certaine signature de type ne peut être déclarée ou définie qu'une seule fois dans une base de code C ++ entière visible par le compilateur C ++. En d'autres termes, les fonctions avec une signature de type spécifique ne peuvent pas être redéfinies - elles ne doivent être définies qu'une seule fois. Ainsi, ce qui suit n'est pas valide C ++:

```
int add2(int i);  // The compiler will note that add2 is a function (int) -> int
int add2(int j);  // As add2 already has a definition of (int) -> int, the compiler
                  // will regard this as an error.
```

Si une fonction ne retourne rien, son type de retour est écrit `void` . S'il ne prend aucun paramètre, la liste de paramètres doit être vide.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Appel de fonction

Une fonction peut être appelée après avoir été déclarée. Par exemple, le programme suivant appelle `add2` avec la valeur `2` dans la fonction de `main` :

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
```

```
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n";  // add2(2) will be evaluated at this point,
                                   // and the result is printed.
    return 0;
}
```

Ici, `add2(2)` est la syntaxe d'un appel de fonction.

Définition de fonction

Une *définition de fonction* * est similaire à une déclaration, sauf qu'elle contient également le code qui est exécuté lorsque la fonction est appelée dans son corps.

Un exemple de définition de fonction pour `add2` pourrait être:

```
int add2(int i)       // Data that is passed into (int i) will be referred to by the name i
{                     // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Fonction de surcharge

Vous pouvez créer plusieurs fonctions avec le même nom mais avec des paramètres différents.

```
int add2(int i)           // Code contained in this definition will be evaluated
{                         // when add2() is called with one parameter.
    int j = i + 2;
    return j;
}

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{                         // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}
```

Les deux fonctions sont appelées du même nom `add2` , mais la fonction réelle appelée dépend directement de la quantité et du type des paramètres de l'appel. Dans la plupart des cas, le compilateur C ++ peut calculer quelle fonction appeler. Dans certains cas, le type doit être explicitement indiqué.

Paramètres par défaut

Les valeurs par défaut des paramètres de fonction ne peuvent être spécifiées que dans les déclarations de fonction.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;               // If multiply() is called with one parameter, the
}                               // value will be multiplied by the default, 7.
```

Dans cet exemple, `multiply()` peut être appelé avec un ou deux paramètres. Si un seul paramètre est donné, `b` aura la valeur par défaut de 7. Les arguments par défaut doivent être placés dans les derniers arguments de la fonction. Par exemple:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

Appels de fonctions spéciales - Opérateurs

Il existe des appels de fonctions spéciaux en C ++ qui ont une syntaxe différente de `name_of_function(value1, value2, value3)` . L'exemple le plus courant est celui des opérateurs.

Certaines séquences de caractères spéciales qui seront réduites à des appels de fonctions par le compilateur, comme `!` , `+` , `-` , `*` , `%` et `<<` et beaucoup plus. Ces caractères spéciaux sont normalement associés à une utilisation autre que la programmation ou sont utilisés pour l'esthétique (par exemple, le caractère `+` est généralement reconnu comme symbole d'ajout à la fois dans la programmation C ++ et dans les mathématiques élémentaires).

C ++ gère ces séquences de caractères avec une syntaxe spéciale; mais, en substance, chaque occurrence d'un opérateur est réduite à un appel de fonction. Par exemple, l'expression C ++ suivante:

```
3+3
```

est équivalent à l'appel de fonction suivant:

```
operator+(3, 3)
```

Tous les noms de fonctions d'opérateur commencent par `operator` .

Alors que dans le prédécesseur immédiat de C ++, C, les noms de fonctions d'opérateurs ne peuvent pas être affectés de différentes significations en fournissant des définitions supplémentaires avec des signatures de types différentes, en C ++, cela est valable. Cacher des définitions de fonctions supplémentaires sous un nom de fonction unique est appelé **surcharge d'opérateur** en C ++ et constitue une convention relativement commune mais non universelle en C ++.

Visibilité des prototypes et déclarations de fonctions

En C ++, le code doit être déclaré ou défini avant utilisation. Par exemple, ce qui suit produit une erreur de compilation:

```
int main()
{
  foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

Il existe deux manières de résoudre ce problème: placer la définition ou la déclaration de `foo()` avant son utilisation dans `main()` . Voici un exemple:

```
void foo(int x) {}  //Declare the foo function and body first

int main()
{
  foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

Cependant, il est également possible de "déclarer à l'avance" la fonction en ne mettant qu'une déclaration "prototype" avant son utilisation, puis en définissant le corps de la fonction ultérieurement:

```
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
  foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

Le prototype doit spécifier le type de retour (`void`), le nom de la fonction (`foo`) et les types de variable de la liste d'arguments (`int`), mais les noms des arguments ne sont PAS requis .

Un moyen courant d'intégrer cela dans l'organisation des fichiers sources est de créer un fichier d'en-tête contenant toutes les déclarations de prototypes:

```
// foo.h
void foo(int); // prototype declaration
```

et ensuite fournir la définition complète ailleurs:

```
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

puis, une fois compilé, liez le fichier objet correspondant `foo.o` au fichier objet compilé où il est utilisé dans la phase de liaison, `main.o` :

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was
beforehand.
// the prototype and body definitions of foo are linked through the object files
```

Une erreur de «symbole externe non résolu» se produit lorsque le *prototype de* fonction et l' *appel* existent, mais le *corps de* la fonction n'est pas défini. Celles-ci peuvent être plus difficiles à résoudre, car le compilateur ne signalera pas l'erreur avant la dernière étape de la liaison et ne sait pas à quelle ligne accéder dans le code pour afficher l'erreur.

Le processus de compilation C ++ standard

Le code du programme C ++ exécutable est généralement produit par un compilateur.

Un **compilateur** est un programme qui traduit le code d'un langage de programmation en une autre forme qui est (plus) directement exécutable pour un ordinateur. L'utilisation d'un compilateur pour traduire du code s'appelle la **compilation.**

C ++ hérite de la forme de son processus de compilation à partir de son langage "parent", C. Ci-dessous une liste des quatre étapes principales de la compilation en C ++:

1. Le préprocesseur C ++ copie le contenu de tous les fichiers d'en-tête inclus dans le fichier de code source, génère du code de macro et remplace les constantes symboliques définies à l'aide de #define par leurs valeurs.
2. Le fichier de code source développé par le préprocesseur C ++ est compilé en langage d'assemblage adapté à la plate-forme.
3. Le code assembleur généré par le compilateur est assemblé en code objet approprié pour la plate-forme.
4. Le fichier de code objet généré par l'assembleur est lié aux fichiers de code objet pour toutes les fonctions de bibliothèque utilisées pour produire un fichier exécutable.

- Remarque: certains codes compilés sont liés entre eux, mais pas pour créer un programme final. Habituellement, ce code "lié" peut également être empaqueté dans un format pouvant être utilisé par d'autres programmes. Ce "paquet de code utilisable et empaqueté" est ce que les programmeurs C ++ appellent une **bibliothèque.**

De nombreux compilateurs C ++ peuvent également fusionner ou désassembler certaines parties du processus de compilation pour plus de facilité ou pour une analyse supplémentaire. De nombreux programmeurs C ++ utiliseront différents outils, mais tous les outils suivront généralement ce processus généralisé lorsqu'ils sont impliqués dans la production d'un programme.

Le lien ci-dessous prolonge cette discussion et fournit un bon graphique pour vous aider. [1]: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Préprocesseur

Le préprocesseur est une partie importante du compilateur.

Il édite le code source, coupe certains bits, en change d'autres et ajoute d'autres choses.

Dans les fichiers source, nous pouvons inclure des directives de préprocesseur. Ces directives indiquent au préprocesseur d'effectuer des actions spécifiques. Une directive commence par un # sur une nouvelle ligne. Exemple:

```
#define ZERO 0
```

La première directive préprocesseur que vous rencontrerez est probablement la

```
#include <something>
```

directif. Ce qu'il fait est prend tous `something` et il insère dans votre dossier où la directive était. Le programme hello world commence par la ligne

```
#include <iostream>
```

Cette ligne ajoute les fonctions et objets vous permettant d'utiliser l'entrée et la sortie standard.

Le langage C, qui utilise également le préprocesseur, n'a pas autant de fichiers d'en-tête que le langage C ++, mais en C ++, vous pouvez utiliser tous les fichiers d'en-tête C.

La prochaine directive importante est probablement la

```
#define something something_else
```

directif. Cela indique au préprocesseur qu'au fur et à mesure qu'il avance dans le fichier, il doit remplacer toutes les occurrences de `something` avec `something_else` . Il peut aussi rendre les choses similaires aux fonctions, mais cela compte probablement comme C ++ avancé.

`something_else` n'est pas nécessaire, mais si vous définissez `something` comme rien, alors en dehors des directives du préprocesseur, toutes les occurrences de `something` disparaîtront.

Cela est utile en raison des directives `#if` , `#else` et `#ifdef` . Le format pour ceux-ci serait le suivant:

```
#if something==true
//code
#else
//more code
#endif

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

Ces directives insèrent le code qui est dans le vrai bit et supprime les faux bits. Cela peut être utilisé pour avoir des morceaux de code qui ne sont inclus que sur certains systèmes d'exploitation, sans avoir à réécrire tout le code.

Lire Démarrer avec C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/206/demarrer-avec-c-plusplus

Chapitre 2: Algorithmes de bibliothèque standard

Examples

std :: for_each

```
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
```

Effets:

Applique `f` à la suite de déréférencement chaque itération dans l'intervalle `[first, last)` à partir de `first` et de procéder à `last - 1` .

Paramètres:

`first, last` - la gamme d'appliquer `f` à.

`f` - objet appelable qui est appliqué au résultat du déréférencement de chaque itérateur de la plage `[first, last)` .

Valeur de retour:

`f` (jusqu'à C ++ 11) et `std::move(f)` (depuis C ++ 11).

Complexité:

Applique `f` exactement `last - first` fois.

Exemple:

c ++ 11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

Applique la fonction donnée à chaque élément du vecteur `v` imprimant cet élément à la `stdout` .

std :: next_permutation

```
template< class Iterator >
bool next_permutation( Iterator first, Iterator last );
template< class Iterator, class Compare >
bool next_permutation( Iterator first, Iterator last, Compare cmpFun );
```

Effets:

Tamisez la séquence de données de la plage [première, dernière] dans la prochaine permutation lexicographique supérieure. Si `cmpFun` est fourni, la règle de permutation est personnalisée.

Paramètres:
`first` - le début de la plage à permuter, inclusivement
`last` - la fin de la gamme à permuter, exclusive

Valeur de retour:
Renvoie true si une telle permutation existe.
Sinon, la plage est convertie en la plus petite permutation lexicographique et renvoie false.

Complexité:
O (n), n est la distance entre le `first` et le `last` .

Exemple :

```
std::vector< int > v { 1, 2, 3 };
do
{
   for( int i = 0; i < v.size(); i += 1 )
   {
       std::cout << v[i];
   }
   std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

imprimer tous les cas de permutation de 1,2,3 dans l'ordre croissant de lexicographie.
sortie:

```
123
132
213
231
312
321
```

std :: accumuler

Défini dans l'en-tête `<numeric>`

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init); // (1)

template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation f); // (2)
```

Effets:

std :: accumulate exécute une opération de pliage en utilisant la fonction `f` sur la plage `[first, last)` commençant par `init` tant que valeur de l'accumulateur.

En effet c'est l'équivalent de:

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

Dans la version (1), `operator+` est utilisé à la place de `f` , donc accumuler sur le conteneur est équivalent à la somme des éléments du conteneur.

Paramètres:

`first, last` - la gamme d'appliquer `f` à.
`init` - valeur initiale de l'accumulateur.
`f` - fonction de pliage binaire.

Valeur de retour:

Valeur cumulée des applications `f` .

Complexité:

O (n × k) , où *n* est la distance entre le `first` et le `last` , *O (k)* est la complexité de la fonction `f` .

Exemple:

Exemple de somme simple:

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

Sortie:

```
10
```

Convertir les chiffres en nombre:

c ++ 11

```
class Converter {
public:
    int operator()(int a, int d) const { return a * 10 + d; }
};
```

et ensuite

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;
```

c ++ 11

```
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
                        0,
                        [](int a, int d) { return a * 10 + d; });
std::cout << n << std::endl;
```

Sortie:

```
123
```

std :: find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

Effets

Trouve la première occurrence de val dans l'intervalle [premier, dernier]

Paramètres

`first` => itérateur pointant au début de la plage `last` => itérateur pointant vers la fin de la plage `val` => valeur à trouver dans la plage

Revenir

Un itérateur qui pointe vers le premier élément dans la plage égale à = (==), l'itérateur pointe sur last si val est introuvable.

Exemple

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> intVec {4, 6, 8, 9, 10, 30, 55,100, 45, 2, 4, 7, 9, 43, 48};

  //define iterators
  vector<int>::iterator  itr_9;
  vector<int>::iterator  itr_43;
  vector<int>::iterator  itr_50;

  //calling find
  itr_9 = find(intVec.begin(), intVec.end(), 9); //occurs twice
  itr_43 = find(intVec.begin(), intVec.end(), 43); //occurs once

  //a value not in the vector
  itr_50 = find(intVec.begin(), intVec.end(), 50); //does not occur
```

```
  cout << "first occurence of: " << *itr_9 << endl;
  cout << "only occurence of: " << *itr_43 << Lendl;


  /*
    let's prove that itr_9 is pointing to the first occurence
    of 9 by looking at the element after 9, which should be 10
    not 43
  */
  cout << "element after first 9: " << *(itr_9 + 1) << ends;

  /*
    to avoid dereferencing intVec.end(), lets look at the
    element right before the end
  */
  cout << "last element: " << *(itr_50 - 1) << endl;

  return 0;
}
```

Sortie

```
first occurence of: 9
only occurence of: 43
element after first 9: 10
last element: 48
```

std :: count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

Effets

Compte le nombre d'éléments égaux à val

Paramètres

`first` => itérateur pointant vers le début de la plage
`last` => itérateur pointant vers la fin de la plage
`val` => L'occurrence de cette valeur dans la plage sera comptée

Revenir

Le nombre d'éléments compris dans la plage (==) à val.

Exemple

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;
```

```
int main(int argc, const char * argv[]) {

  //create vector
  vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

  //count occurences of 9, 55, and 101
  size_t count_9 = count(intVec.begin(), intVec.end(), 9); //occurs twice
  size_t count_55 = count(intVec.begin(), intVec.end(), 55); //occurs once
  size_t count_101 = count(intVec.begin(), intVec.end(), 101); //occurs once

  //print result
  cout << "There are " << count_9  << " 9s"<< endl;
  cout << "There is " << count_55  << " 55"<< endl;
  cout << "There is " << count_101  << " 101"<< ends;

  //find the first element == 4 in the vector
  vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

  //count its occurences in the vector starting from the first one
  size_t count_4 = count(itr_4, intVec.end(), *itr_4); // should be 2

  cout << "There are " << count_4  << " " << *itr_4 << endl;

  return 0;
}
```

Sortie

```
There are 2 9s
There is 1 55
There is 0 101
There are 2 4
```

std :: count_if

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate red);
```

Effets

Compte le nombre d'éléments dans une plage pour laquelle une fonction de prédicat spécifiée est vraie

Paramètres

`first` => itérateur pointant vers le début de la plage `last` => itérateur pointant vers la fin de la plage `red` => fonction du prédicat (retourne true ou false)

Revenir

Nombre d'éléments dans la plage spécifiée pour lesquels la fonction de prédicat a renvoyé la valeur true.

Exemple

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
    Define a few functions to use as predicates
*/

//return true if number is odd
bool isOdd(int i){
  return i%2 == 1;
}

//functor that returns true if number is greater than the value of the constructor parameter
provided
class Greater {
  int _than;
public:
  Greater(int th): _than(th){}
  bool operator()(int i){
    return i > _than;
  }
};

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

  //using a lambda function to count even numbers
  size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); //
>= C++11

  //using function pointer to count odd number in the first half of the vector
  size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

  //using a functor to count numbers greater than 5
  size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

  cout << "vector size: " << myvec.size() << endl;
  cout << "even numbers: " << evenCount << " found" << endl;
  cout << "odd numbers: " << oddCount << " found" << endl;
  cout << "numbers > 5: " << greaterCount << " found"<< endl;

  return 0;
}
```

Sortie

```
vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found
```

std :: find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Effets

Trouve le premier élément d'une plage pour laquelle la fonction de prédicat `pred` renvoie true.

Paramètres

`first` => itérateur pointant vers le début de la plage `last` => itérateur pointant vers la fin de la plage `pred` => fonction prédicat (retourne true ou false)

Revenir

Un itérateur qui pointe vers le premier élément de la plage dans laquelle la fonction de prédicat pred renvoie true pour. L'itérateur pointe sur last si val est introuvable

Exemple

```
#include <iostream>
#include <vector>
#include <algorithm>


using namespace std;

/*
    define some functions to use as predicates
*/

//Returns true if x is multiple of 10
bool multOf10(int x) {
  return x % 10 == 0;
}

//returns true if item greater than passed in parameter
class Greater {
  int _than;

public:
  Greater(int th):_than(th){

  }
  bool operator()(int data) const
  {
    return data > _than;
  }
};


int main()
{

  vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};
```

```
  //with a lambda function
  vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;});
// >= C++11

  //with a function pointer
  vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

  //with functor
  vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

  //not Found
  vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf points
to myvec.end()


  //check if pointer points to myvec.end()
  if(nf != myvec.end()) {
    cout << "nf points to: " << *nf << endl;
  }
  else {
    cout << "item not found" << endl;
  }



  cout << "First item >   10: " << *gt10  << endl;
  cout << "First Item n * 10: " << *pow10 << endl;
  cout << "First Item >    5: " << *gt5   << endl;

  return 0;
}
```

Sortie

```
item not found
First item >   10: 56
First Item n * 10: 10
First Item >    5: 6
```

std :: min_element

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last,Compare comp);
```

Effets

Trouve l'élément minimum dans une plage

Paramètres

`first` - itérateur pointant vers le début de la plage
`last` -iterator pointant vers la fin de la plage `comp` - un pointeur de fonction ou un objet fonction qui

prend deux arguments et renvoie true ou false indiquant si l'argument est inférieur à l'argument 2. Cette fonction ne doit pas modifier les entrées

Revenir

Itérateur à l'élément minimum de la gamme

Complexité

Linéaire en un moins que le nombre d'éléments comparés.

Exemple

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>  //to use make_pair

using namespace std;

//function compare two pairs
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
  return p1.second < p2.second;
}

int main(int argc, const char * argv[]) {

  vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

  vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2),
make_pair("z", 26), make_pair("e", 5) };


  // default using < operator
  auto minInt = min_element(intVec.begin(), intVec.end());

  //Using pairLessThanFunction
  auto minPairFunction = min_element(pairVector.begin(), pairVector.end(),
pairLessThanFunction);


  //print minimum of intVector
  cout << "min int from default: " << *minInt << endl;

  //print minimum of pairVector
  cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

  return 0;
}
```

Sortie

```
min int from default: 6
min pair from PairLessThanFunction: 2
```

Utiliser std :: nth_element pour trouver la médiane (ou d'autres quantiles)

L'algorithme `std::nth_element` prend trois itérateurs: un itérateur au début, *n* ème position et fin. Une fois que la fonction retourne, le *n* ième élément (par ordre) sera le *n* ième élément le plus petit. (La fonction a des surcharges plus élaborées, par exemple, certains prenant des foncteurs de comparaison; voir le lien ci-dessus pour toutes les variations.)

Remarque Cette fonction est très efficace - elle présente une complexité linéaire.

Par souci de cet exemple, définissons la médiane d'une séquence de longueur *n en* tant qu'élément qui serait en position $\lceil n / 2 \rceil$. Par exemple, la médiane d'une séquence de longueur 5 est le troisième élément le plus petit, de même que la médiane d'une séquence de longueur 6.

Pour utiliser cette fonction pour trouver la médiane, nous pouvons utiliser ce qui suit. Disons que nous commençons avec

```
std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// This makes the 2nd position hold the median.
std::nth_element(b, med, e);

// The median is now at v[2].
```

Pour trouver le *p* e quantile , nous changerions quelques - unes des lignes ci - dessus:

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

et recherchez le quantile à la position `pos` .

Lire Algorithmes de bibliothèque standard en ligne: https://riptutorial.com/fr/cplusplus/topic/3177/algorithmes-de-bibliotheque-standard

Chapitre 3: Alignement

Introduction

Tous les types en C ++ ont un alignement. Ceci est une restriction sur l'adresse mémoire que les objets de ce type peuvent être créés à l'intérieur. Une adresse mémoire est valide pour la création d'un objet si la division de cette adresse par l'alignement de l'objet est un nombre entier.

Les alignements de type sont toujours une puissance de deux (dont 1).

Remarques

La norme garantit les éléments suivants:

- L'exigence d'alignement d'un type est un diviseur de sa taille. Par exemple, une classe avec une taille de 16 octets peut avoir un alignement de 1, 2, 4, 8 ou 16, mais pas de 32. (Si les membres d'une classe ne totalisent que 14 octets, mais que la classe doit avoir un besoin d'alignement) de 8, le compilateur va insérer 2 octets de remplissage pour que la taille de la classe soit égale à 16.)

- Les versions signées et non signées d'un type entier ont la même exigence d'alignement.
- Un pointeur à `void` a la même exigence d'alignement qu'un pointeur sur `char` .
- Les versions qualifiées cv et non qualifiées CV d'un type ont la même exigence d'alignement.

Notez que si l'alignement existe en C ++ 03, ce n'est qu'en C ++ 11 qu'il est devenu possible d'interroger l'alignement (en utilisant `alignof`) et de contrôler l'alignement (en utilisant `alignas`).

Examples

Interroger l'alignement d'un type

c ++ 11

L'exigence d'alignement d'un type peut être interrogée à l'aide du mot clé `alignof` tant qu'opérateur unaire. Le résultat est une expression constante de type `std::size_t` , *c'est-à-dire* qu'il peut être évalué au moment de la compilation.

```
#include <iostream>
int main() {
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';
}
```

Sortie possible

> Le besoin d'alignement de int est: 4

Si elle est appliquée à un tableau, elle génère l’alignement requis pour le type d’élément. S'il est appliqué à un type de référence, il génère l'exigence d'alignement du type référencé. (Les références elles-mêmes n'ont aucun alignement, car elles ne sont pas des objets.)

Contrôle de l'alignement

C ++ 11

Le mot-clé `alignas` peut être utilisé pour forcer une variable, un membre de données de classe, une déclaration ou une définition de classe, ou une déclaration ou une définition d'un enum, à avoir un alignement particulier, s'il est pris en charge. Il se présente sous deux formes:

- `alignas(x)` , où `x` est une expression constante, donne à l'entité l'alignement `x` , si elle est supportée.
- `alignas(T)` , où `T` est un type, donne à l'entité un alignement égal à l'exigence d'alignement de `T` , c'est-à-dire `alignof(T)` , si elle est prise en charge.

Si plusieurs spécificateurs d' `alignas` sont appliqués à la même entité, le plus strict s'applique.

Dans cet exemple, le tampon `buf` est correctement aligné pour contenir un objet `int` , même si son type d'élément est un caractère `unsigned char` , ce qui peut nécessiter un alignement plus faible.

```
alignas(int) unsigned char buf[sizeof(int)];
new (buf) int(42);
```

`alignas` ne peut pas être utilisé pour donner à un type un alignement plus petit que le type sans cette déclaration:

```
alignas(1) int i; //Il-formed, unless `int` on this platform is aligned to 1 byte.
alignas(char) int j; //Il-formed, unless `int` has the same or smaller alignment than `char`.
```

`alignas` , quand on leur donne une expression constante, doivent recevoir un alignement valide. Les alignements valides sont toujours des puissances de deux et doivent être supérieurs à zéro. Les compilateurs doivent prendre en charge tous les alignements valides jusqu’à l’alignement du type `std::max_align_t` . Ils *peuvent* prendre en charge des alignements plus importants, mais la prise en charge de l'allocation de mémoire pour ces objets est limitée. La limite supérieure des alignements dépend de l'implémentation.

C ++ 17 dispose d'un support direct dans l' `operator new` pour allouer de la mémoire aux types sur-alignés.

Lire Alignement en ligne: https://riptutorial.com/fr/cplusplus/topic/9249/alignement

Chapitre 4: Arithmétique en virgule flottante

Examples

Les nombres à virgule flottante sont étranges

La première erreur que presque tous les programmeurs font est de présumer que ce code fonctionnera comme prévu:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

Le programmeur novice suppose que cela résume chaque nombre compris entre `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99` , pour donner le résultat `199` , la réponse mathématiquement correcte.

Deux choses se produisent qui rendent cela inexact:

1. Le programme tel qu'il est écrit ne se termine jamais. `a` ne devient jamais égal à `2` et la boucle ne se termine jamais.
2. Si nous réécrivons la logique de boucle pour vérifier `a < 2` place, la boucle se termine, mais le total finit par être différent de `199` . Sur les machines conformes à la norme IEEE754, la somme est généralement d'environ `201` place.

La raison en est que les **nombres à virgule flottante représentent des approximations de leurs valeurs attribuées** .

L'exemple classique est le calcul suivant:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Bien que ce que le programmeur voit, ce sont trois nombres écrits en base10, ce que le compilateur (et le matériel sous-jacent) voient sont des nombres binaires. Parce que `0.1` , `0.2` et `0.3` exigent une division parfaite par `10` - ce qui est assez facile dans un système de base 10, mais impossible dans un système de base 2 - ces nombres doivent être stockés dans des formats imprécis, similaires au nombre `1/3` doit être stocké dans la forme imprécise `0.333333333333333...` en base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111110111001100110011001100110011001100110011001100110011010; //imperfect
```

```
representation of 0.1
double b =     0011111111001001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c =     0011111111010011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that
this is not quite equal to the "canonical" 0.3!
```

Lire Arithmétique en virgule flottante en ligne:
https://riptutorial.com/fr/cplusplus/topic/5115/arithmetique-en-virgule-flottante

Chapitre 5: auto

Remarques

Le mot clé `auto` est un nom de fichier représentant un type déduit automatiquement.

C'était déjà un mot-clé réservé en C ++ 98, hérité de C. Dans les anciennes versions de C ++, il pouvait être utilisé pour indiquer explicitement qu'une variable a une durée de stockage automatique:

```
int main()
{
  auto int i = 5; // removing auto has no effect
}
```

Cette ancienne signification est maintenant supprimée.

Examples

Échantillon automatique de base

Le mot `auto` clé `auto` fournit la déduction automatique du type d'une variable.

C'est particulièrement pratique pour traiter les noms de type long:

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

avec des boucles basées sur la plage :

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

avec lambdas :

```
auto f = [](){ std::cout << "lambda\n"; };
f();
```

pour éviter la répétition du type:

```
auto w = std::make_shared< Widget >();
```

pour éviter des copies surprenantes et inutiles:

```
auto myMap = std::map<int,float>();
myMap.emplace(1,3.14);

std::pair<int,float> const& firstPair2 = *myMap.begin();  // copy!
auto const& firstPair = *myMap.begin();  // no copy!
```

La raison de la copie est que le type renvoyé est en fait `std::pair<const int,float>` !

auto et modèles d'expression

`auto` peut également causer des problèmes lorsque des modèles d'expression entrent en jeu:

```
auto mult(int c) {
    return c * std::valarray<int>{1};
}

auto v = mult(3);
std::cout << v[0]; // some value that could be, but almost certainly is not, 3.
```

La raison en est que l' `operator*` sur `valarray` vous donne un objet proxy qui fait référence à la `valarray` comme moyen d'évaluation `valarray` . En utilisant `auto` , vous créez une référence en suspens. Au lieu de `mult` avait renvoyé un `std::valarray<int>` , alors le code imprimerait définitivement 3.

auto, const et références

Le mot `auto` clé `auto` représente lui-même un type de valeur, similaire à `int` ou `char` . Il peut être modifié avec le mot-clé `const` et le symbole `&` pour représenter respectivement un type const ou un type de référence. Ces modificateurs peuvent être combinés.

Dans cet exemple, `s` est un type de valeur (son type sera infd `std::string`), donc chaque itération de la boucle `for` *copie* une chaîne du vecteur dans `s` .

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

Si le corps de la boucle modifie `s` (comme en appelant `s.append(" and stuff")`), seule cette copie sera modifiée, pas le membre d'origine des `strings` .

Par contre, si `s` est déclaré avec `auto&` ce sera un type de référence (supposé être `std::string&`), donc à chaque itération de la boucle, une *référence* à une chaîne dans le vecteur lui sera attribuée:

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```

Dans le corps de cette boucle, les modifications apportées à `s` affecteront directement l'élément

des `strings` auquel il fait référence.

Enfin, si `s` est déclaré `const auto&` , il s'agira d'un type de référence const, ce qui signifie qu'à chaque itération de la boucle, une *référence const* sera affectée à une chaîne dans le vecteur:

```
for(const auto& s : strings) {
    std::cout << s << std::endl;
}
```

Dans le corps de cette boucle, les `s` ne peuvent pas être modifiés (c.-à-d. Qu'aucune méthode non-const ne peut y être appelée).

Lorsque vous utilisez `auto` avec `for` boucles basées `for` plages, il est généralement recommandé d'utiliser `const auto&` si le corps de la boucle ne modifie pas la structure en boucle, car cela évite les copies inutiles.

Type de retour

`auto` est utilisé dans la syntaxe pour le type de retour de fin:

```
auto main() -> int {}
```

ce qui équivaut à

```
int main() {}
```

Principalement utile combiné avec `decltype` pour utiliser des paramètres au lieu de `std::declval<T>` :

```
template <typename T1, typename T2>
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

Lambda générique (C ++ 14)

C ++ 14

C ++ 14 permet d'utiliser `auto` dans l'argument lambda

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };

print(42);
print("hello world");
```

Ce lambda est essentiellement équivalent à

```
struct lambda {
    template <typename T>
    auto operator ()(const T& arg) const {
        std::cout << arg << std::endl;
```

```
    }
};
```

et alors

```
lambda print;

print(42);
print("hello world");
```

objets auto et proxy

Parfois, l' `auto` peut ne pas se comporter comme prévu par un programmeur. Il en déduit l'expression, même si la déduction de type n'est pas la bonne chose à faire.

Par exemple, lorsque des objets proxy sont utilisés dans le code:

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

Ici, `flag` ne serait pas `bool` , mais `std::vector<bool>::reference` , car pour la spécialisation `bool` du template `vector` l' `operator []` renvoie un objet proxy avec l'opérateur d' `operator bool` conversion `operator bool` défini.

Lorsque `flags.push_back(true)` modifie le conteneur, cette pseudo-référence peut se terminer par un élément qui n'existe plus.

Cela rend également la prochaine situation possible:

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

Le `vector` est éliminé immédiatement, donc `flag` est une pseudo-référence à un élément qui a été supprimé. L'appel à `foo` invoque un comportement indéfini.

Dans des cas comme celui-ci, vous pouvez déclarer une variable avec `auto` et l'initialiser en la convertissant au type que vous voulez déduire:

```
auto flag = static_cast<bool>(getFlags()[5]);
```

mais à ce stade, remplacer simplement `auto` par `bool` plus de sens.

Un autre cas où des objets proxy peuvent causer des problèmes est celui des modèles d'expression . Dans ce cas, les modèles ne sont parfois pas conçus pour durer au-delà de la pleine expression complète pour des raisons d'efficacité, et l'utilisation de l'objet proxy sur le

prochain entraîne un comportement indéfini.

Lire auto en ligne: https://riptutorial.com/fr/cplusplus/topic/2421/auto

Chapitre 6: Bit Manipulation

Remarques

Pour utiliser `std::bitset` vous devrez inclure l'en- tête `<bitset>` .

```
#include <bitset>
```

`std::bitset` surcharge toutes les fonctions d'opérateur pour permettre le même usage que le traitement des bits par c-style.

Les références

- Bit Twiddling Hacks

Examples

Mettre un peu

Manipulation de bits de style C

Un bit peut être défini à l'aide de l'opérateur OU bit à bit (`|`).

```
// Bit x will be set
number |= 1LL << x;
```

Utiliser std :: bitset

`set(x)` ou `set(x,true)` - définit bit à la position `x` sur `1` .

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
```

Effacer un peu

Manipulation de bits de style C

Un bit peut être effacé à l'aide de l'opérateur AND bitwise (`&`).

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Utiliser std :: bitset

`reset(x)` ou `set(x,false)` - efface le bit à la position `x` .

```
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
```

En changeant un peu

Manipulation de bits de style C

Un bit peut être basculé à l'aide de l'opérateur XOR (`^`).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Utiliser std :: bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Vérification un peu

Manipulation de bits de style C

La valeur du bit peut être obtenue en décalant le nombre vers la droite `x` puis en effectuant un bit ET (`&`) dessus:

```
(number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

L'opération de décalage vers la droite peut être mise en œuvre sous la forme d'un décalage arithmétique (signé) ou d'un décalage logique (non signé). Si `number` dans le `number >> x` expression `number >> x` a un type signé et une valeur négative, la valeur résultante est définie par l'implémentation.

Si nous avons besoin de la valeur de ce bit directement sur place, nous pourrions au lieu de cela

déplacer le masque:

```
(number & (1LL << x));  // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

L'un ou l'autre peut être utilisé comme conditionnel, car toutes les valeurs non nulles sont considérées comme vraies.

Utiliser std :: bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
```

Changer le nième bit en x

Manipulation de bits de style C

```
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

Utiliser std :: bitset

`set(n,val)` - met le bit `n` à la valeur `val` .

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Définir tous les bits

Manipulation de bits de style C

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(Voir ici pour expliquer pourquoi cela fonctionne et est en fait la meilleure approche.)

Utiliser std :: bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Supprimer le bit mis à l'extrême droite

Manipulation de bits de style C

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
    return n & (n - 1);
}
```

Explication

- si `n` est zéro, on a `0 & 0xFF..FF` qui est zéro
- sinon `n` peut être écrit `0bxxxxxx10..00` et `n - 1` est `0bxxxxxx011..11` , donc `n & (n - 1)` est `0bxxxxxx000..00` .

Jeu de bits de comptage

Le décompte de population d'une chaîne de bits est souvent nécessaire en cryptographie et dans d'autres applications et le problème a été largement étudié.

La manière naïve nécessite une itération par bit:

```
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
  bits += value & 1;
```

Une bonne astuce (basée sur l' option Retirer le bit le plus à droite) est la suivante:

```
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (; value; ++bits)
  value &= value - 1;
```

Il traverse autant d'itérations qu'il y a de bits définis, donc c'est bien quand on s'attend à ce que la `value` ait peu de bits non nuls.

La méthode a été proposée pour la première fois par Peter Wegner (dans CACM 3/322 - 1960) et elle est bien connue depuis son apparition dans *C Programming Language* par Brian W. Kernighan et Dennis M. Ritchie.

Cela nécessite 12 opérations arithmétiques, dont une multication:

```
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111

  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

Ce type d'implémentation a le meilleur comportement dans le pire des cas (voir Poids de Hamming pour plus de détails).

De nombreux processeurs ont une instruction spécifique (comme le `popcnt` de x86) et le compilateur peut offrir une fonction intégrée spécifique (**non standard**). Par exemple, avec g ++, il y a:

```
int __builtin_popcount (unsigned x);
```

Vérifiez si un entier est une puissance de 2

L'astuce `n & (n - 1)` (voir Remove bit le plus à droite) est également utile pour déterminer si un entier est une puissance de 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Notez que sans la première partie de la vérification (`n &&`), `0` est considéré à tort comme une puissance de 2.

Application de manipulation de bits: Lettre minuscule à majuscule

L'une des nombreuses applications de la manipulation de bits consiste à convertir une lettre du petit au capital ou vice versa en choisissant un **masque** et une **opération de bit** appropriée. Par exemple, la lettre a cette représentation binaire `01(1)00001` tandis que son homologue du capital a `01(0)00001` . Ils diffèrent uniquement par le bit entre parenthèses. Dans ce cas, convertir **une** lettre de petite en majuscule met fondamentalement le bit entre parenthèses à un. Pour ce faire, nous faisons ce qui suit:

```
/****************************************
convert small letter to captial letter.
========================================
     a: 01100001
  mask: 11011111  <-- (0xDF)  11(0)11111
      :---------
a&mask: 01000001  <-- A letter
*****************************************/
```

Le code pour convertir une lettre en lettre A est

```
#include <cstdio>

int main()
{
    char op1 = 'a';  // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c   &   0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

Le résultat est

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Lire Bit Manipulation en ligne: https://riptutorial.com/fr/cplusplus/topic/3016/bit-manipulation

Chapitre 7: Boucles

Introduction

Une instruction de boucle exécute un groupe d'instructions à plusieurs reprises jusqu'à ce qu'une condition soit remplie. Il existe 3 types de boucles primitives en C ++: for, while et do ... while.

Syntaxe

- while *déclaration* (*condition*);
- faire une *déclaration* tandis que (*expression*);
- *(pour* la *-instruction-initialisation, condition; expression) déclaration;*
- pour l' *instruction* (*for-range-declaration* : *for-range-initializer*);
- Pause ;
- continuer ;

Remarques

`algorithm` appels d' `algorithm` sont généralement préférables aux boucles écrites à la main.

Si vous voulez quelque chose qu'un algorithme fait déjà (ou quelque chose de très similaire), l'appel d'algorithme est plus clair, souvent plus efficace et moins sujet aux erreurs.

Si vous avez besoin d'une boucle qui fait quelque chose d'assez simple (mais nécessiterait un enchevêtrement de liants et d'adaptateurs si vous utilisiez un algorithme), alors écrivez simplement la boucle.

Examples

Basé sur la gamme pour

C ++ 11

`for` loops peut être utilisé pour parcourir les éléments d'une plage basée sur un itérateur, sans utiliser d'index numérique ou accéder directement aux itérateurs:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

Cela va parcourir tous les éléments de `v` , avec `val` la valeur de l'élément en cours. La déclaration

suivante:

```
for (for-range-declaration : for-range-initializer ) statement
```

est équivalent à:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

C ++ 17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Cette modification a été introduite pour la prise en charge prévue de Ranges TS en C ++ 20.

Dans ce cas, notre boucle est équivalente à:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

Notez que `auto val` déclare un type de valeur, qui sera une copie d'une valeur stockée dans la plage (nous l'initialisons depuis l'itérateur). Si les valeurs stockées dans la plage sont coûteuses à copier, vous pouvez utiliser `const auto &val` . Vous n'êtes pas non plus obligé d'utiliser `auto` ; Vous pouvez utiliser un nom de fichier approprié, à condition qu'il soit implicitement convertible à partir du type de valeur de la plage.

Si vous avez besoin d’accéder à l’itérateur, le mode basé sur les plages ne peut pas vous aider (au moins pas sans effort).

Si vous souhaitez le référencer, vous pouvez le faire:

```
vector<float> v = {0.4f, 12.5f, 16.234f};
```

```
for(float &val: v)
{
    std::cout << val << " ";
}
```

Vous pouvez itérer sur la référence `const` si vous avez un conteneur `const` :

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

On utilisera des références de transfert lorsque l'itérateur de séquence retourne un objet proxy et que vous devez opérer sur cet objet d'une manière non `const` . Remarque: il est fort probable que cela va dérouter les lecteurs de votre code.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

Le type « plage » fourni à la gamme-base `for` peut être l' un des éléments suivants:

- Tableaux de langues:

  ```
  float arr[] = {0.4f, 12.5f, 16.234f};

  for(auto val: arr)
  {
      std::cout << val << " ";
  }
  ```

  Notez que l'allocation d'un tableau dynamique ne compte pas:

  ```
  float *arr = new float[3]{0.4f, 12.5f, 16.234f};

  for(auto val: arr) //Compile error.
  {
      std::cout << val << " ";
  }
  ```

- Tout type ayant des fonctions membres `begin()` et `end()` , qui renvoient des itérateurs aux éléments du type. Les conteneurs de bibliothèque standard se qualifient, mais les types définis par l'utilisateur peuvent également être utilisés:

  ```
  struct Rng
  {
      float arr[3];
  ```

```
    // pointers are iterators
    const float* begin() const {return &arr[0];}
    const float* end() const   {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end()   {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- Tout type qui a des fonctions `begin(type)` et `end(type)` qui peuvent être trouvées via la recherche dépendante des arguments, en fonction du `type` . Ceci est utile pour créer un type de plage sans avoir à modifier le type de classe lui-même:

```
namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

Pour la boucle

Une boucle `for` exécute des instructions dans le `loop body` la `loop body` , tandis que la `condition` la boucle est vraie. Avant que l' `initialization statement` la boucle ne soit exécutée exactement une fois. Après chaque cycle, la partie d' `iteration execution` l' `iteration execution` est exécutée.

Une boucle `for` est définie comme suit:

```
for (/*initialization statement*/; /*condition*/; /*iteration execution*/)
{
    // body of the loop
}
```

Explication des énoncés de placeholder:

- `initialization statement` : cette instruction n'est exécutée qu'une seule fois, au début de la boucle `for` . Vous pouvez saisir une déclaration de plusieurs variables d'un type, telles que `int i = 0, a = 2, b = 3` . Ces variables ne sont valides que dans le cadre de la boucle. Les variables définies avant la boucle du même nom sont masquées lors de l'exécution de la boucle.
- `condition` : cette instruction est évaluée avant chaque exécution de *corps de boucle* et annule la boucle si elle est évaluée à `false` .
- `iteration execution` : cette instruction est exécutée après le *corps de* la boucle, avant l'évaluation de la *condition* suivante, à moins que la boucle `for` soit abandonnée dans le *corps* (par `break` , `goto` , `return` ou une exception étant lancée). Vous pouvez entrer plusieurs instructions dans la partie d' `iteration execution` l' `iteration execution` , telles que `a++, b+=10, c=b+a` .

L'équivalent approximatif d'une boucle `for` , réécrit en tant `while` boucle while est:

```
/*initialization*/
while (/*condition*/)
{
    // body of the loop; using 'continue' will skip to increment part below
    /*iteration execution*/
}
```

Le cas le plus courant d'utilisation d'une boucle `for` consiste à exécuter des instructions un nombre de fois spécifique. Par exemple, prenez en compte les éléments suivants:

```
for(int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

Une boucle valide est également:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {
    std::cout << a << " " << b << " " << c << std::endl;
}
```

Un exemple de masquage de variables déclarées avant une boucle est:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //we declare a new variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 99
```

Mais si vous souhaitez utiliser la variable déjà déclarée et ne pas la masquer, omettez la partie déclaration:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //we are using already declared variable i
```

```
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 10
```

Remarques:

- Les instructions d'initialisation et d'incrément peuvent effectuer des opérations sans rapport avec la déclaration de condition, ou rien du tout, si vous le souhaitez. Mais pour des raisons de lisibilité, il est recommandé de ne réaliser que des opérations directement liées à la boucle.
- Une variable déclarée dans l'instruction d'initialisation est visible uniquement dans la portée de la boucle `for` et est libérée à la fin de la boucle.
- N'oubliez pas que la variable déclarée dans l' `initialization statement` peut être modifiée pendant la boucle, ainsi que la variable vérifiée dans la `condition` .

Exemple de boucle qui compte de 0 à 10:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Explication des fragments de code:

- `int counter = 0` initialise le `counter` variable à 0. (Cette variable ne peut être utilisée qu'à l'intérieur de la boucle `for` .)
- `counter <= 10` est une condition booléenne qui vérifie si le `counter` est inférieur ou égal à 10. Si c'est `true` , la boucle s'exécute. Si c'est `false` , la boucle se termine.
- `++counter` est une opération d'incrémentation qui incrémente la valeur du `counter` de 1 avant la vérification de condition suivante.

En laissant toutes les instructions vides, vous pouvez créer une boucle infinie:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

Le `while` en boucle équivalente de ce qui précède est la suivante :

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

Cependant, une boucle infinie peut toujours être laissée en utilisant les instructions `break` , `goto` ou `return` ou en lançant une exception.

Le prochain exemple courant d'itération sur tous les éléments d'une collection STL (par exemple, un `vector`) sans utiliser l' `<algorithm>` tête `<algorithm>` est:

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

En boucle

A `while` boucle exécute des instructions à plusieurs reprises jusqu'à ce que la condition donnée est évaluée à `false` . Cette instruction de contrôle est utilisée quand on ne sait pas à l'avance combien de fois un bloc de code doit être exécuté.

Par exemple, pour imprimer tous les nombres de 0 à 9, le code suivant peut être utilisé:

```
int i = 0;
while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console
```

C ++ 17

Notez que depuis C ++ 17, les 2 premières instructions peuvent être combinées

```
while (int i = 0; i < 10)
//... The rest is the same
```

Pour créer une boucle infinie, la construction suivante peut être utilisée:

```
while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}
```

Il existe une autre variante des boucles `while` , à savoir la construction `do...while` . Voir l' exemple de boucle do-while pour plus d'informations.

Déclaration de variables dans des conditions

Dans la condition des boucles `for` et `while` , il est également permis de déclarer un objet. Cet objet sera considéré comme ayant une portée jusqu'à la fin de la boucle et persistera à chaque itération de la boucle:

```
for (int i = 0; i < 5; ++i) {
    do_something(i);
}
// i is no longer in scope.

for (auto& a : some_container) {
    a.do_something();
}
```

```
// a is no longer in scope.

while(std::shared_ptr<Object> p = get_object()) {
   p->do_something();
}
// p is no longer in scope.
```

Cependant, il n'est pas permis de faire la même chose avec une boucle `do...while` while; au lieu de cela, déclarez la variable avant la boucle et (éventuellement) incluez la variable et la boucle dans une étendue locale si vous voulez que la variable sorte de la portée après la fin de la boucle:

```
//This doesn't compile
do {
    s = do_something();
} while (short s > 0);

// Good
short s;
do {
    s = do_something();
} while (s > 0);
```

En effet , la partie *de* la *déclaration* d'une `do...while` la boucle (le corps de la boucle) est évaluée avant la partie *d'expression* (le `while`) est atteinte, et donc, toute déclaration dans *l'expression* ne sera pas visible lors de la première itération de la boucle.

Boucle Do-while

Une boucle *do-while* est très similaire à une boucle *while,* sauf que la condition est vérifiée à la fin de chaque cycle, pas au début. La boucle est donc garantie pour s'exécuter au moins une fois.

Le code suivant affichera `0` , comme condition évaluera à `false` à la fin de la première itération:

```
int i =0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console
```

Note: N'oubliez pas le point-virgule à la fin de `while(condition);` , qui est nécessaire dans la construction *do-while* .

Contrairement à la boucle *do-* while, les éléments suivants n'imprimeront rien, car la condition est `false` au début de la première itération:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
```

```
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Remarque: Une boucle *while* peut être sorti sans la condition de devenir fausse en utilisant une `break` , `goto` ou `return` déclaration.

```
int i = 0;
do
{
    std::cout << i;
    ++i; // Increment counter
    if (i > 5)
    {
        break;
    }
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console
```

Une boucle *do-while* trivial est aussi parfois utilisé pour écrire des macros qui nécessitent leur propre champ (dans ce cas , le point - virgule final est omis de la définition de la macro et doit être fourni par l'utilisateur):

```
#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);
```

Instructions de contrôle de boucle: Break and Continue

Les instructions de contrôle de boucle sont utilisées pour modifier le flux d'exécution à partir de sa séquence normale. Lorsque l'exécution laisse une étendue, tous les objets automatiques créés dans cette étendue sont détruits. Le `break` and `continue` sont des instructions de contrôle de boucle.

L'instruction `break` termine une boucle sans autre considération.

```
for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}
```

Le code ci-dessus sera imprimé:

```
1
2
```

```
3
```

L'instruction `continue` ne quitte pas immédiatement la boucle, mais ignore le reste du corps de la boucle et se dirige vers le haut de la boucle (y compris la vérification de la condition).

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
       does not execute  */
    std::cout << i << " is an odd number\n";
}
```

Le code ci-dessus sera imprimé:

```
1 is an odd number
3 is an odd number
5 is an odd number
```

En raison de ces changements de flux de contrôle sont parfois difficiles pour l' homme de comprendre facilement, `break` et `continue` sont utilisés avec parcimonie. Une implémentation plus simple est généralement plus facile à lire et à comprendre. Par exemple, la première `for` la boucle avec la `break` ci - dessus peut être réécrite comme:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << '\n';
}
```

Le deuxième exemple avec `continue` pourrait être réécrit comme `continue` :

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 != 0) {
        std::cout << i << " is an odd number\n";
    }
}
```

Portée pour une sous-gamme

En utilisant des boucles de base, vous pouvez effectuer une boucle sur une sous-partie d'un conteneur donné ou d'une autre plage en générant un objet proxy qualifié pour les boucles basées sur la plage.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
  Iterator b;
  Sentinel e;
  Iterator begin() const { return b; }
  Sentinel end() const { return e; }
  bool empty() const { return begin()==end(); }
```

```
  range_t without_front( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {std::next(b, count), e};
  }
  range_t without_back( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {b, std::prev(e, count)};
  }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinal e ) {
  return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
  using std::begin; using std::end;
  return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
  auto r = range(c);
  if (r.empty()) return r;
  return r.without_front();
}
```

maintenant nous pouvons faire:

```
std::vector<int> v = {1,2,3,4};

for (auto i : except_first(v))
  std::cout << i << '\n';
```

et imprimer

```
2
3
4
```

Sachez que les objets intermédiaires générés dans la partie `for(:range_expression)` de la boucle `for` auront expiré au moment où la boucle `for` démarre.

Lire Boucles en ligne: https://riptutorial.com/fr/cplusplus/topic/589/boucles

Chapitre 8: C incompatibilités

Introduction

Cela décrit ce que le code C va casser dans un compilateur C ++.

Examples

Mots-clés réservés

Le premier exemple est constitué de mots-clés ayant une utilité particulière en C ++: les éléments suivants sont légaux en C, mais pas en C ++.

```
int class = 5
```

Ces erreurs sont faciles à corriger: renommez simplement la variable.

Pointeurs faiblement typés

En C, les pointeurs peuvent être convertis en un `void*` , qui nécessite une conversion explicite en C ++. Ce qui suit est illégal en C ++, mais légal en C:

```
void* ptr;
int* intptr = ptr;
```

L'ajout d'une distribution explicite fait que cela fonctionne, mais peut causer d'autres problèmes.

aller ou changer

En C ++, vous ne pouvez pas ignorer les initialisations avec `goto` ou `switch` . Ce qui suit est valide en C, mais pas en C ++:

```
goto foo;
int skipped = 1;
foo;
```

Ces bogues peuvent nécessiter une refonte.

Lire C incompatibilités en ligne: https://riptutorial.com/fr/cplusplus/topic/9645/c-incompatibilites

Chapitre 9: Catégories de valeur

Examples

Signification des catégories de valeur

Les expressions en C ++ se voient attribuer une catégorie de valeur particulière, en fonction du résultat de ces expressions. Les catégories de valeur pour les expressions peuvent affecter la résolution de la surcharge de la fonction C ++.

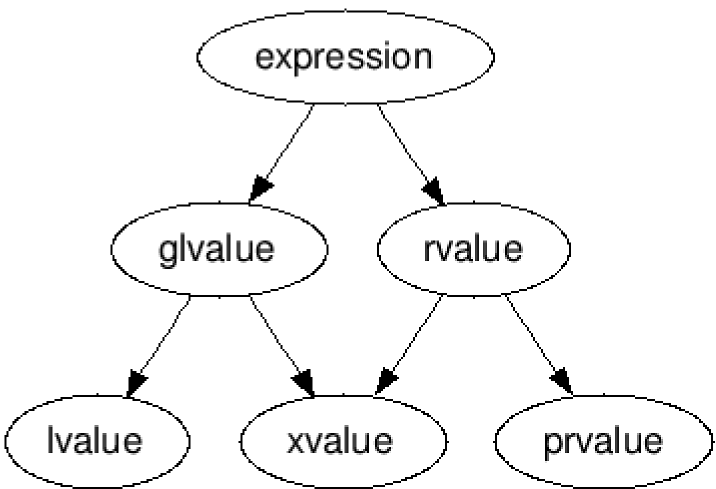
Les catégories de valeur déterminent deux propriétés importantes mais distinctes d'une expression. Une propriété est de savoir si l'expression a une identité. Une expression a une identité si elle fait référence à un objet qui a un nom de variable. Le nom de la variable peut ne pas être impliqué dans l'expression, mais l'objet peut toujours en avoir un.

L'autre propriété est de savoir s'il est légal de déplacer implicitement la valeur de l'expression. Ou plus précisément, si l'expression, utilisée en tant que paramètre de fonction, sera liée aux types de paramètres de valeur r ou non.

C ++ définit 3 catégories de valeurs qui représentent la combinaison utile de ces propriétés: lvalue (expressions avec identité mais non déplaçables à partir de), xvalue (expressions avec identité pouvant être déplacées) et prvalue (expressions sans identité à partir desquelles). C ++ n'a pas d'expressions sans identité et ne peut pas être déplacé.

C ++ définit deux autres catégories de valeurs, chacune basée uniquement sur l'une de ces propriétés: glvalue (expressions avec identité) et rvalue (expressions pouvant être déplacées). Ceux-ci agissent comme des regroupements utiles des catégories précédentes.

Ce graphique sert d'illustration:



valeur

Une expression prvalue (pure-rvalue) est une expression dépourvue d'identité, dont l'évaluation

est généralement utilisée pour initialiser un objet, et qui peut être implicitement déplacée. Ceux-ci incluent, mais ne sont pas limités à:

- Expressions représentant des objets temporaires, tels que `std::string("123")` .
- Une expression d'appel de fonction qui ne renvoie pas de référence
- Un littéral (*sauf* un littéral de chaîne - ce sont des lvalues), tel que `1` , `true` , `0.5f` ou `'a'`
- Une expression lambda

L'adresse intégrée de l'opérateur (`&`) ne peut pas être appliquée à ces expressions.

xvalue

Une expression xvalue (eXpiring value) est une expression ayant une identité et représentant un objet pouvant être implicitement déplacé. L'idée générale des expressions xvalue est que l'objet qu'elles représentent va bientôt être détruit (d'où la partie "eXpiring"), et par conséquent, se déplacer implicitement de ces objets est correct.

Donné:

```
struct X { int n; };
extern X x;

4;                    // prvalue: does not have an identity
x;                    // lvalue
x.n;                  // lvalue
std::move(x);         // xvalue
std::forward<X&>(x);  // lvalue
X{4};                 // prvalue: does not have an identity
X{4}.n;               // xvalue: does have an identity and denotes resources
                      // that can be reused
```

lvalue

Une expression lvalue est une expression qui a une identité, mais ne peut pas être implicitement déplacée. Parmi celles-ci figurent des expressions composées d'un nom de variable, d'un nom de fonction, d'expressions utilisées par l'opérateur de déréférencement intégré et d'expressions faisant référence à des références lvalue.

La lvalue typique est simplement un nom, mais les lvalues peuvent également avoir d'autres saveurs:

```
struct X { ... };

X x;         // x is an lvalue
X* px = &x;  // px is an lvalue
*px = X{};   // *px is also an lvalue, X{} is a prvalue

X* foo_ptr();  // foo_ptr() is a prvalue
X& foo_ref();  // foo_ref() is an lvalue
```

De plus, alors que la plupart des littéraux (par exemple `4` , `'x'` , etc.) sont des valeurs, les littéraux

de chaîne sont des lvalues.

glvalue

Une expression glvalue (une "lvalue généralisée") est une expression qui a une identité, qu'elle puisse ou non être déplacée. Cette catégorie inclut les lvalues (expressions qui ont une identité mais qui ne peuvent pas être déplacées) et les xvalues (expressions qui ont une identité et peuvent être déplacées), mais exclut les valeurs (expressions sans identité).

Si une expression a un *nom* , c'est une glvalue:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
              // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

rvalue

Une expression rvalue est une expression pouvant être implicitement déplacée, qu'elle ait ou non une identité.

Plus précisément, les expressions rvalue peuvent être utilisées comme argument d'une fonction qui prend un paramètre de type `T &&` (où `T` est le type de `expr`). *Seules les* expressions rvalue peuvent être utilisées comme arguments pour ces paramètres de fonction; Si une expression non-rvalue est utilisée, alors la résolution de la surcharge choisira toute fonction n'utilisant pas de paramètre de référence rvalue. Et si aucun n'existe, alors vous obtenez une erreur.

La catégorie des expressions rvalue comprend toutes les expressions xvalue et prvalue, et uniquement ces expressions.

La fonction standard de la bibliothèque `std::move` existe pour transformer explicitement une expression non-rvalue en une rvalue. Plus précisément, il transforme l'expression en une xvalue, car même si c'était une expression de valeur sans identité auparavant, en la passant en paramètre à `std::move` , elle acquiert une identité (le nom du paramètre de la fonction) et devient une valeur x.

Considérer ce qui suit:

```
std::string str("init");                       //1
std::string test1(str);                        //2
std::string test2(std::move(str));             //3

str = std::string("new value");                //4
std::string &&str_ref = std::move(str);        //5
```

```
std::string test3(str_ref);                    //6
```

`std::string` a un constructeur qui prend un seul paramètre de type `std::string&&` , communément appelé "constructeur de déplacement". Cependant, la catégorie de valeur de l'expression `str` n'est pas une valeur (en particulier, c'est une lvalue), elle ne peut donc pas appeler cette surcharge du constructeur. Au lieu de cela, il appelle le `const std::string&` overload, le constructeur de la copie.

La ligne 3 change les choses. La valeur de retour de `std::move` est un `T&&` , où `T` est le type de base du paramètre passé. Donc `std::move(str)` renvoie `std::string&&` . Un appel de fonction dont la valeur de retour est une référence rvalue est une expression rvalue (spécifiquement une xvalue), de sorte qu'il peut appeler le constructeur move de `std::string` . Après la ligne 3, `str` a été déplacé de (dont le contenu est maintenant indéfini).

La ligne 4 transmet un temporaire à l'opérateur d'affectation de `std::string` . Cela a une surcharge qui prend un `std::string&&` . L'expression `std::string("new value")` est une expression rvalue (spécifiquement une valeur), elle peut donc appeler cette surcharge. Ainsi, le temporaire est déplacé dans `str` , remplaçant le contenu non défini par un contenu spécifique.

La ligne 5 crée une référence nommée `str_ref` appelée `str_ref` qui fait référence à `str` . C'est là que les catégories de valeur sont déroutantes.

Voir, alors que `str_ref` est une référence rvalue à `std::string` , la catégorie de valeur de l'expression `str_ref` *n'est pas une rvalue* . C'est une expression de lvalue. Oui vraiment. De ce fait, on ne peut pas appeler le constructeur move de `std::string` avec l'expression `str_ref` . La ligne 6 *copie* donc la valeur de `str` dans `test3` .

Pour le déplacer, il faudrait utiliser `std::move` nouveau.

Lire Catégories de valeur en ligne: https://riptutorial.com/fr/cplusplus/topic/763/categories-de-valeur

Chapitre 10: Champs de bits

Introduction

Les champs de bits compressent étroitement les structures C et C ++ pour réduire la taille. Cela semble indolore: spécifiez le nombre de bits pour les membres, et le compilateur fait le travail de co-mingling bits. La restriction est l'incapacité de prendre l'adresse d'un membre du champ de bits, car elle est stockée de manière concomitante. `sizeof()` est également interdit.

Le coût des champs de bits est un accès plus lent, car la mémoire doit être récupérée et les opérations au niveau des bits appliquées pour extraire ou modifier les valeurs des membres. Ces opérations ajoutent également à la taille de l'exécutable.

Remarques

Quel est le coût des opérations binaires? Supposons une structure de champ non binaire simple:

```
struct foo {
    unsigned x;
    unsigned y;
}
static struct foo my_var;
```

Dans un code ultérieur:

```
my_var.y = 5;
```

Si `sizeof (unsigned) == 4` , alors x est stocké au début de la structure et y est stocké 4 octets. Le code assembleur généré peut ressembler à:

```
loada register1,#myvar     ; get the address of the structure
storei register1[4],#0x05  ; put the value '5' at offset 4, e.g., set y=5
```

C'est simple parce que x n'est pas mêlé à y. Mais imaginez redéfinir la structure avec des champs de bits:

```
struct foo {
    unsigned x : 4; /* Range 0-0x0f, or 0 through 15 */
    unsigned y : 4;
}
```

4 bits seront attribués à `x` et `y` , partageant un seul octet. La structure ainsi occupe 1 octet, au lieu de 8. Considérons l'ensemble pour définir `y` maintenant, en supposant qu'il se retrouve dans le quartet supérieur:

```
loada  register1,#myvar        ; get the address of the structure
loadb  register2,register1[0]  ; get the byte from memory
```

```
andb   register2,#0x0f         ; zero out y in the byte, leaving x alone
orb    register2,#0x50         ; put the 5 into the 'y' portion of the byte
stb    register1[0],register2  ; put the modified byte back into memory
```

Cela peut être un bon compromis si nous avons des milliers ou des millions de ces structures, et cela aide à garder la mémoire dans le cache ou à empêcher le swap - ou pourrait gonfler l'exécutable pour aggraver ces problèmes et ralentir le traitement. Comme pour toutes choses, faites preuve de discernement.

Utilisation du pilote de périphérique: évitez les champs de bits comme stratégie d'implémentation intelligente pour les pilotes de périphérique. Les dispositions de stockage sur champ de bits ne sont pas nécessairement cohérentes entre les compilateurs, ce qui rend ces implémentations non portables. La lecture-modification-écriture pour définir les valeurs peut ne pas faire ce que les périphériques attendent, provoquant des comportements inattendus.

Examples

Déclaration et utilisation

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

Ici, chacun de ces deux champs occupera 1 bit en mémoire. Il est spécifié par `: 1` expression après les noms de variable. Le type de base du champ de bits peut être tout type entier (int 8 bits à 64 bits int). L'utilisation d'un type `unsigned` est recommandée, sinon des surprises peuvent survenir.

Si plus de bits sont requis, remplacez "1" par le nombre de bits requis. Par exemple:

```
struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day:   5;  // 32
};
```

La structure entière n'utilise que 22 bits, et avec les paramètres de compilateur normaux, la `sizeof` cette structure serait de 4 octets.

L'utilisation est assez simple. Déclarez simplement la variable et utilisez-la comme une structure ordinaire.

```
Date d;

d.Year = 2016;
d.Month = 7;
d.Day =  22;
```

```
std::cout << "Year:" << d.Year << std::endl <<
        "Month:" << d.Month << std::endl <<
        "Day:" << d.Day << std::endl;
```

Lire Champs de bits en ligne: https://riptutorial.com/fr/cplusplus/topic/2710/champs-de-bits

Chapitre 11: Classes / Structures

Syntaxe

- variable.member_var = constante;
- variable.member_function ();
- variable_pointer-> member_var = constant;
- variable_pointer-> member_function ();

Remarques

Notez que la ***seule*** différence entre les mots-clés `struct` et `class` est que, par défaut, les variables membres, les fonctions membres et les classes de base d'une `struct` sont `public` , alors que dans une `class` elles sont `private` . Les programmeurs C ++ ont tendance à l'appeler une classe si elle a des constructeurs et des destructeurs, et la possibilité d'imposer ses propres invariants; ou une structure si ce n'est qu'une simple collection de valeurs, mais le langage C ++ lui-même ne fait aucune distinction.

Examples

Les bases de la classe

Une *classe* est un type défini par l'utilisateur. Une classe est introduite avec le mot-clé `class` , `struct` ou `union` . En usage familier, le terme "classe" désigne généralement uniquement les classes non syndiquées.

Une classe est une collection de *membres* de *classe* , qui peuvent être:

- les variables membres (également appelées "champs"),
- fonctions membres (également appelées "méthodes"),
- types de membres ou typedefs (par exemple "classes imbriquées"),
- gabarits de membres (de tout type: variable, fonction, classe ou modèle d'alias)

Les mots `struct` clés `class` et `struct` , appelés *les clés de classe* , sont largement interchangeables, sauf que le spécificateur d'accès par défaut pour les membres et les bases est "private" pour une classe déclarée avec la clé `class` et "public" pour la classe déclarée avec `struct` ou `union` (cf. modificateurs d'accès).

Par exemple, les extraits de code suivants sont identiques:

```
struct Vector
{
    int x;
    int y;
    int z;
};
```

```
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

En déclarant une classe`, un nouveau type est ajouté à votre programme, et il est possible d'instancier des objets de cette classe par

```
Vector my_vector;
```

Les membres d'une classe sont accessibles à l'aide de la syntaxe à points.

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my:vector.z = 7;
```

Spécificateurs d'accès

Trois mots - clés agissent comme **spécificateurs d'accès** . Celles-ci limitent l'accès aux membres de la classe après le spécificateur, jusqu'à ce qu'un autre spécificateur modifie à nouveau le niveau d'accès:

Mot-clé	La description
`public`	Tout le monde a accès
`protected`	Seule la classe elle-même, les classes dérivées et les amis ont accès
`private`	Seule la classe elle-même et ses amis ont accès

Lorsque le type est défini à l'aide du mot `class` clé `class` , le spécificateur d'accès par défaut est `private` , mais si le type est défini à l'aide du mot `struct` clé `struct` , le spécificateur d'accès par défaut est `public` :

```
struct MyStruct { int x; };
class MyClass { int x; };

MyStruct s;
s.x = 9; // well formed, because x is public

MyClass c;
c.x = 9; // ill-formed, because x is private
```

Les spécificateurs d'accès sont principalement utilisés pour limiter l'accès aux champs et méthodes internes, et obligent le programmeur à utiliser une interface spécifique, par exemple pour forcer l'utilisation de getters et de setters au lieu de référencer directement une variable:

```
class MyClass {

public: /* Methods: */

    int x() const noexcept { return m_x; }
    void setX(int const x) noexcept { m_x = x; }

private: /* Fields: */

    int m_x;

};
```

L'utilisation de `protected` est utile pour permettre à certaines fonctionnalités du type d'être uniquement accessibles aux classes dérivées. Par exemple, dans le code suivant, la méthode `calculateValue()` est uniquement accessible aux classes dérivées de la classe de base `Plus2Base` , telle que `FortyTwo` :

```
struct Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* Methods: */
    virtual int calculateValue() noexcept = 0;
};
struct FortyTwo: Plus2Base {
protected: /* Methods: */
    int calculateValue() noexcept final override { return 40; }
};
```

Notez que le mot-clé `friend` peut être utilisé pour ajouter des exceptions d'accès aux fonctions ou aux types d'accès aux membres protégés et privés.

Les mots clés `public` , `protected` et `private` peuvent également être utilisés pour accorder ou limiter l'accès aux sous-objets de classe de base. Voir l'exemple d' héritage .

Héritage

Les classes / structures peuvent avoir des relations d'héritage.

Si une classe / struct `B` hérite d'une classe / struct `A` , cela signifie que `B` a comme parent `A` On dit que `B` est une classe / struct dérivée de `A` , et `A` est la classe / struct de base.

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
{
};
```

Il existe 3 formes d'héritage pour une classe / struct:

- `public`
- `private`
- `protected`

Notez que l'héritage par défaut est identique à la visibilité par défaut des membres: `public` si vous utilisez le mot `struct` clé `struct` et `private` pour le mot `class` clé `class` .

Il est même possible qu'une `class` dérive d'une `struct` (ou vice versa). Dans ce cas, l'héritage par défaut est contrôlé par l'enfant. Par conséquent, une `struct` dérivant d'une `class` sera héritée par défaut et une `class` dérivée d'une `struct` aura un héritage privé par défaut.

héritage `public` :

```
struct B : public A // or just `struct B : A`
{
    void foo()
    {
        p1 = 0; //well formed, p1 is public in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

héritage `private` :

```
struct B : private A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is private in B
        p2 = 0; //well formed, p2 is private in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible
```

héritage `protected` :

```
struct B : protected A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is protected in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
```

```
    }
};

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

Notez que bien que l'héritage `protected` soit autorisé, son utilisation réelle est rare. Un exemple de la façon dont l'héritage `protected` est utilisé dans les applications est la spécialisation de classe de base partielle (généralement appelée «polymorphisme contrôlé»).

Lorsque la POO était relativement nouvelle, on disait souvent que l'héritage (public) modélisait une relation "IS-A". En d'autres termes, l'héritage public est correct uniquement si une instance de la classe dérivée *est également une* instance de la classe de base.

Cela a ensuite été affiné dans le principe de substitution Liskov : l'héritage public ne devrait être utilisé que si / si une instance de la classe dérivée peut être substituée à une instance de la classe de base dans des circonstances possibles (et toujours logique).

On dit généralement que l'héritage privé incarne une relation complètement différente: "est implémenté en termes de" (parfois appelé relation "HAS-A"). Par exemple, une classe `Stack` peut hériter en privé d'une classe `Vector` . L'héritage privé ressemble beaucoup plus à l'agrégation qu'à l'héritage public.

L'héritage protégé n'est presque jamais utilisé et il n'y a pas d'accord général sur le type de relation qu'il incarne.

Héritage Virtuel

Lorsque vous utilisez l'héritage, vous pouvez spécifier le mot clé `virtual` :

```
struct A{};
struct B: public virtual A{};
```

Lorsque la classe `B` a la base virtuelle `A` cela signifie que `A` **résidera dans la classe** d'arbre d'héritage la **plus dérivée** , et que la classe la plus dérivée est également responsable de l'initialisation de cette base virtuelle:

```
struct A
{
    int member;
    A(int param)
    {
        member = param;
    }
};

struct B: virtual A
{
    B(): A(5){}
};
```

```
struct C: B
{
    C(): /*A(88)*/ {}
};

void f()
{
    C object; //error since C is not initializing it's indirect virtual base `A`
}
```

Si nous dé-commentons `/*A(88)*/` nous n'obtiendrons aucune erreur puisque `C` initialise maintenant sa base virtuelle indirecte `A`

Notez également que lorsque nous créons un `object` variable, la classe la plus dérivée est `C` , donc `C` est responsable de la création (appel du constructeur de) `A` et donc la valeur de `A::member` est de `88` et non de `5` (comme ce serait le cas si nous étions créer un objet de type `B`).

C'est utile pour résoudre le problème des diamants .

```
  A                                       A   A
 / \                                      |   |
B   C                                     B   C
 \ /                                       \ /
  D                                         D
virtual inheritance                   normal inheritance
```

`B` et `C` héritent tous deux de `A` , et `D` hérite de `B` et `C` , il **y a** donc **2 instances de `A` dans `D` !** Cela entraîne une ambiguïté lorsque vous accédez à un membre de `A` à `D` , car le compilateur n'a aucun moyen de savoir de quelle classe voulez-vous accéder à ce membre (celui dont `B` hérite ou celui hérité par `C` ?) .

L'héritage virtuel résout ce problème: comme la base virtuelle ne réside que dans la plupart des objets dérivés, il n'y aura qu'une seule instance de `A` dans `D`

```
struct A
{
    void foo() {}
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

La suppression des commentaires résout l'ambiguïté.

Héritage multiple

Mis à part l'héritage unique:

```
class A {};
class B : public A {};
```

Vous pouvez également avoir plusieurs héritages:

```
class A {};
class B {};
class C : public A, public B {};
```

`C` aura maintenant hérité de `A` et `B` en même temps.

Remarque: cela peut entraîner une ambiguïté si les mêmes noms sont utilisés dans plusieurs `class` héritées ou `struct` . Faites attention!

Ambiguïté dans l'héritage multiple

L'héritage multiple peut être utile dans certains cas mais, parfois, une sorte de rencontre étrange de problèmes lors de l'utilisation de l'héritage multiple.

Par exemple: Deux classes de base ont des fonctions avec le même nom qui ne sont pas remplacées dans la classe dérivée et si vous écrivez du code pour accéder à cette fonction en utilisant un objet de classe dérivée, le compilateur affiche une erreur. Voici un code pour ce type d'ambiguïté dans l'héritage multiple.

```
class base1
{
  public:
     void funtion( )
     { //code for base1 function }
};
class base2
{
    void function( )
     { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

  // Error because compiler can't figure out which function to call
  //either function( ) of base1 or base2 .
    obj.function( )
}
```

Mais, ce problème peut être résolu en utilisant la fonction de résolution de portée pour spécifier quelle fonction classer soit base1, soit base2:

```
int main()
{
    obj.base1::function( );  // Function of class base1 is called.
    obj.base2::function( );  // Function of class base2 is called.
}
```

Accéder aux membres de la classe

Pour accéder aux variables membres et aux fonctions membres d'un objet d'une classe, le `.` l'opérateur est utilisé:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
// Accessing member variable a in var.
std::cout << var.a << std::endl;
// Assigning member variable b in var.
var.b = 1;
// Calling a member function.
var.foo();
```

Lors de l'accès aux membres d'une classe via un pointeur, l'opérateur `->` est couramment utilisé. Alternativement, l'instance peut être déréférencée et le `.` opérateur utilisé, bien que ce soit moins fréquent:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
SomeStruct *p = &var;
// Accessing member variable a in var via pointer.
std::cout << p->a << std::endl;
std::cout << (*p).a << std::endl;
// Assigning member variable b in var via pointer.
p->b = 1;
(*p).b = 1;
// Calling a member function via a pointer.
p->foo();
(*p).foo();
```

Lors de l'accès aux membres de classe statiques, l'opérateur `::` est utilisé, mais sur le nom de la classe au lieu d'une instance de celle-ci. Il est également possible d'accéder au membre statique à partir d'une instance ou d'un pointeur vers une instance à l'aide de `.` ou `->` opérateur, respectivement, avec la même syntaxe que l'accès à des membres non statiques.

```
struct SomeStruct {
  int a;
```

```
  int b;
  void foo() {}

  static int c;
  static void bar() {}
};
int SomeStruct::c;

SomeStruct var;
SomeStruct* p = &var;
// Assigning static member variable c in struct SomeStruct.
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Contexte

L'opérateur `->` est nécessaire car l'opérateur d'accès au membre `.` a priorité sur l'opérateur de déréférencement `*`.

On pourrait s'attendre à ce que `*pa` déréférencement `p` (résultant en une référence à l'objet `p` pointe vers) et à accéder ensuite à son membre `a`. Mais en fait, il essaie d'accéder au membre `a` de `p` et le déréférencer. Ie `*pa` équivaut à `*(pa)`. Dans l'exemple ci-dessus, cela entraînerait une erreur de compilation à cause de deux faits: Premièrement, `p` est un pointeur et n'a pas de membre `a`. Deuxièmement, `a` est un entier et ne peut donc pas être déréférencé.

La solution peu commune à ce problème serait de contrôler explicitement la priorité: `(*p).a`

Au lieu de cela, l'opérateur `->` est presque toujours utilisé. C'est un raccourci pour déréférencer le pointeur et y accéder. Ie `(*p).a` est exactement le même que `p->a`.

L'opérateur `::` est l'opérateur de portée, utilisé de la même manière que l'accès à un membre d'un espace de noms. En effet, un membre de classe statique est considéré comme étant dans la portée de cette classe, mais n'est pas considéré comme membre des instances de cette classe. L'utilisation de la normale `.` et `->` est également autorisé pour les membres statiques, même s'ils ne sont pas membres de l'instance, pour des raisons historiques; Ceci est utile pour écrire du code générique dans les modèles, car l'appelant n'a pas besoin de se préoccuper de savoir si une fonction membre donnée est statique ou non.

Héritage privé: restriction de l'interface de la classe de base

L'héritage privé est utile lorsqu'il est nécessaire de restreindre l'interface publique de la classe:

```
class A {
public:
    int move();
    int turn();
```

```
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move();  // compile error
b.turn();  // OK
```

Cette approche empêche efficacement l'accès aux méthodes publiques A en lançant sur le pointeur ou la référence A:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

Dans le cas de l'héritage public, un tel casting donnera accès à toutes les méthodes publiques A, bien qu'il existe d'autres moyens d'empêcher que cela se produise dans B dérivé, comme cacher:

```
class B : public A {
private:
    int move();
};
```

ou privé en utilisant:

```
class B : public A {
private:
    using A::move;
};
```

alors pour les deux cas, il est possible:

```
B b;
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

Classes finales et structures

C ++ 11

Dériver une classe peut être interdit avec le spécificateur `final` . Déclarons une classe finale:

```
class A final {
};
```

Maintenant, toute tentative de sous-classe provoquera une erreur de compilation:

```
// Compilation error: cannot derive from final class:
class B : public A {
```

```
};
```

La classe finale peut apparaître n'importe où dans la hiérarchie des classes:

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

Relation amicale

Le mot - clé `friend` permet aux autres classes et fonctions d'accéder aux membres privés et protégés de la classe, même s'ils sont définis en dehors de la portée de la classe.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;" declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}

int main() {
    Animal animal = {10, 5};
    printWeight(animal);
```

```
    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

```
10
10, 5
Animal height: 5
```

Classes / Structures imbriquées

Une `class` ou une `struct` peut également contenir une autre définition de `class` / `struct` , appelée "classe imbriquée"; dans cette situation, la classe contenant est appelée "classe englobante". La définition de classe imbriquée est considérée comme un membre de la classe englobante, mais est par ailleurs distincte.

```
struct Outer {
    struct Inner { };
};
```

En dehors de la classe englobante, les classes imbriquées sont accessibles à l'aide de l'opérateur scope. A l'intérieur de la classe englobante, cependant, les classes imbriquées peuvent être utilisées sans qualificatifs:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;
```

Comme pour une `class` / `struct` non imbriquée, les fonctions membres et les variables statiques peuvent être définies dans une classe imbriquée ou dans l'espace de noms englobant. Cependant, ils ne peuvent pas être définis dans la classe englobante, car ils sont considérés comme une classe différente de la classe imbriquée.

```
// Bad.
struct Outer {
    struct Inner {
        void do_something();
    };

    void Inner::do_something() {}
};


// Good.
```

```
struct Outer {
    struct Inner {
        void do_something();
    };

};

void Outer::Inner::do_something() {}
```

Comme pour les classes non imbriquées, les classes imbriquées peuvent être déclarées et définies ultérieurement, à condition qu'elles soient définies avant d'être utilisées directement.

```
class Outer {
    class Inner1;
    class Inner2;

    class Inner1 {};

    Inner1 in1;
    Inner2* in2p;

  public:
    Outer();
    ~Outer();
};

class Outer::Inner2 {};

Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}
Outer::~Outer() {
    if (in2p) { delete in2p; }
}
```

C ++ 11

Avant C ++ 11, les classes imbriquées n'avaient accès qu'aux noms de type, `static` membres `static` et aux énumérateurs de la classe englobante; tous les autres membres définis dans la classe englobante étaient hors limites.

C ++ 11

A partir de C ++ 11, les classes imbriquées et leurs membres sont traités comme s'ils étaient des `friend` de la classe englobante et peuvent accéder à tous ses membres, conformément aux règles d'accès habituelles; Si les membres de la classe imbriquée doivent pouvoir évaluer un ou plusieurs membres non statiques de la classe englobante, ils doivent donc passer une instance:

```
class Outer {
    struct Inner {
        int get_sizeof_x() {
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.
        }

        int get_x() {
            return x; // Illegal: Can't access non-static member without an instance.
        }
```

```
        int get_x(Outer& o) {
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private
members.
        }
    };

    int x;
};
```

À l'inverse, la classe englobante n'est *pas* traitée comme une amie de la classe imbriquée et ne peut donc pas accéder à ses membres privés sans obtenir explicitement l'autorisation.

```
class Outer {
    class Inner {
        // friend class Outer;

        int x;
    };

    Inner in;

  public:
    int get_x() {
        return in.x; // Error: int Outer::Inner::x is private.
        // Uncomment "friend" line above to fix.
    }
};
```

Les amis d'une classe imbriquée ne sont pas automatiquement considérés comme des amis de la classe englobante; S'ils doivent également être amis de la classe fermée, cela doit être déclaré séparément. Inversement, comme la classe englobante n'est pas automatiquement considérée comme une amie de la classe imbriquée, les amis de la classe englobante ne seront pas non plus considérés comme des amis de la classe imbriquée.

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Error: int Outer::Inner::i is private.
    int o = out.o; // Good.
}
```

Comme avec tous les autres membres de la classe, les classes imbriquées ne peuvent être nommées qu'en dehors de la classe si elles ont un accès public. Cependant, vous êtes autorisé à y accéder indépendamment du modificateur d'accès, tant que vous ne les nommez pas explicitement.

```
class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

  public:
    static Inner make_Inner() { return Inner(); }
};

// ...

Outer::Inner oi; // Error: Outer::Inner is private.

auto oi = Outer::make_Inner(); // Good.
oi.func();                     // Good.
Outer::make_Inner().func();    // Good.
```

Vous pouvez également créer un alias de type pour une classe imbriquée. Si un alias de type est contenu dans la classe englobante, le type imbriqué et l'alias de type peuvent avoir des modificateurs d'accès différents. Si l'alias de type est en dehors de la classe englobante, il faut que la classe imbriquée, ou un `typedef` correspondant, soit publique.

```
class Outer {
    class Inner_ {};

  public:
    typedef Inner_ Inner;
};

typedef Outer::Inner  ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...

Outer::Inner  oi; // Good.
Outer::Inner_ oi; // Error.
ImOut         oi; // Good.
```

Comme pour les autres classes, les classes imbriquées peuvent être dérivées ou dérivées d'autres classes.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

Cela peut être utile dans les situations où la classe englobante est dérivée d'une autre classe, en permettant au programmeur de mettre à jour la classe imbriquée si nécessaire. Cela peut être combiné avec un typedef pour fournir un nom cohérent pour chaque classe imbriquée:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

  public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---

class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

  public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...

// Calls BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

Dans le cas ci-dessus, `BaseOuter` et `DerivedOuter` fournissent respectivement le type de membre `Inner` , `BaseInner_` et `DerivedInner_` . Cela permet de dériver des types imbriqués sans casser l'interface de la classe englobante, et permet d'utiliser le type imbriqué de manière polymorphe.

Types de membres et alias

Une `class` ou une `struct` peut également définir des alias de type de membre, qui sont des alias

de type contenus dans la classe et traités comme tels.

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

Comme les membres statiques, ces typedefs sont accessibles à l'aide de l'opérateur scope, `::` .

```
IHaveATypedef::MyTypedef i = 5; // i is an int.

IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

Comme pour les alias de type normal, chaque alias de type membre est autorisé à faire référence à tout type défini ou alias avant, mais pas après sa définition. De même, un typedef en dehors de la définition de la classe peut faire référence à tous les typedefs accessibles dans la définition de la classe, à condition qu'il vienne après la définition de la classe.

```
template<typename T>
struct Helper {
    T get() const { return static_cast<T>(42); }
};

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef        i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;                  // ii is an int.
```

Les alias de type de membre peuvent être déclarés avec n'importe quel niveau d'accès et respecteront le modificateur d'accès approprié.

```
class TypedefAccessLevels {
    typedef int PrvInt;

  protected:
    typedef int ProInt;

  public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.
```

```
class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};
```

Cela peut être utilisé pour fournir un niveau d'abstraction, permettant au concepteur d'une classe de modifier son fonctionnement interne sans casser le code qui en dépend.

```
class Something {
    friend class SomeComplexType;

    short s;
    // ...

  public:
    typedef SomeComplexType MyHelper;

    MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();
```

Dans ce cas, si la classe d'assistance est changée de `SomeComplexType` à un autre type, seules les déclarations `typedef` et `friend` devront être modifiées; tant que la classe d'assistance fournit les mêmes fonctionnalités, tout code qui l'utilise comme `Something::MyHelper` au lieu de le spécifier par nom fonctionnera toujours sans aucune modification. De cette manière, nous réduisons la quantité de code à modifier lorsque l’implémentation sous-jacente est modifiée, de sorte que le nom du type ne doit être modifié que dans un emplacement.

Cela peut également être combiné avec `decltype` , si l'on le souhaite.

```
class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

  public:
    typedef decltype(helper) MyHelper;

  private:
    InternalVariable<MyHelper> ivh;

    // ...

  public:
    MyHelper& get_helper() const { return helper; }

    // ...
};
```

Dans cette situation, changer l'implémentation de `SomethingElse::helper` changera automatiquement le typedef pour nous, en raison de `decltype` . Cela minimise le nombre de modifications nécessaires lorsque l'on veut changer d' `helper` , ce qui minimise le risque d'erreur humaine.

Comme pour tout, cependant, cela peut être pris trop loin. Si le nom de fichier n'est utilisé qu'une ou deux fois en interne et zéro fois en externe, par exemple, il n'est pas nécessaire de lui fournir un alias. S'il est utilisé des centaines ou des milliers de fois au cours d'un projet ou s'il porte un nom suffisamment long, il peut être utile de le fournir en tant que typedef au lieu de toujours l'utiliser en termes absolus. Il faut équilibrer la compatibilité et la commodité avec la quantité de bruit inutile créée.

Cela peut également être utilisé avec les classes de modèle, pour fournir un accès aux paramètres du modèle depuis l'extérieur de la classe.

```
template<typename T>
class SomeClass {
    // ...

  public:
    typedef T MyParam;
    MyParam getParam() { return static_cast<T>(42); }
};

template<typename T>
typename T::MyParam some_func(T& t) {
    return t.getParam();
}

SomeClass<int> si;
int i = some_func(si);
```

Ceci est couramment utilisé avec les conteneurs, qui fournissent généralement leur type d'élément et d'autres types d'assistance, en tant qu'alias de type de membre. La plupart des conteneurs de la bibliothèque standard C ++, par exemple, fournissent les 12 types d'assistance suivants, ainsi que tout autre type spécial dont ils pourraient avoir besoin.

```
template<typename T>
class SomeContainer {
    // ...

  public:
    // Let's provide the same helper types as most standard containers.
    typedef T                                     value_type;
    typedef std::allocator<value_type>            allocator_type;
    typedef value_type&                           reference;
    typedef const value_type&                     const_reference;
    typedef value_type*                           pointer;
    typedef const value_type*                     const_pointer;
    typedef MyIterator<value_type>                iterator;
    typedef MyConstIterator<value_type>           const_iterator;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                size_type;
```

```
    typedef ptrdiff_t                             difference_type;
};
```

Avant C ++ 11, il était également couramment utilisé pour fournir un "template `typedef` " de toutes sortes, car la fonctionnalité n'était pas encore disponible; ceux-ci sont devenus un peu moins courants avec l'introduction de modèles d'alias, mais sont toujours utiles dans certaines situations (et sont combinés avec des modèles d'alias dans d'autres situations, ce qui peut être très utile pour obtenir des composants individuels complexes comme un pointeur de fonction)). Ils utilisent généralement le `type` nom pour leur alias de type.

```
template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.
```

Cela a souvent été utilisé avec des types avec plusieurs paramètres de modèle, pour fournir un alias qui définit un ou plusieurs paramètres.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

OneDArray<int, 3>::type     arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type   arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Membres de la classe statique

Une classe est également autorisée à avoir `static` membres `static` , qui peuvent être des variables ou des fonctions. Ceux-ci sont considérés comme étant dans la portée de la classe, mais ne sont pas traités comme des membres normaux; ils ont une durée de stockage statique (ils existent depuis le début du programme jusqu'à la fin), ne sont pas liés à une instance particulière de la classe et une seule copie existe pour la classe entière.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                         // Non-static member variable.
```

```
  public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();      // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
//  (&one.i != &two.i)
//  (&one.i != &three.i)
//  (&two.i != &three.i).
// All three Examples share "num_instances", such that:
//  (&one.num_instances == &two.num_instances)
//  (&one.num_instances == &three.num_instances)
//  (&two.num_instances == &three.num_instances)
```

Les variables de membre statiques ne sont pas considérées comme définies dans la classe, seulement déclarées, et ont donc leur définition en dehors de la définition de classe; le programmeur est autorisé, mais pas obligatoire, à initialiser les variables statiques dans leur définition. Lors de la définition des variables membres, le mot clé `static` est omis.

```
class Example {
    static int num_instances;               // Declaration.

  public:
    static std::string static_str;          // Declaration.

    // ...
};

int         Example::num_instances;         // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.
```

De ce fait, les variables statiques peuvent être des types incomplets (sauf le `void`), à condition qu'elles soient définies ultérieurement comme un type complet.

```
struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete    i_contain_myself;
    static int             an_array[];
};

struct ForwardDeclared {};

ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
```

```
int             ExIncomplete::an_array[5];
```

Les fonctions membres statiques peuvent être définies à l'intérieur ou à l'extérieur de la définition de classe, comme pour les fonctions membres normales. Comme pour les variables membres statiques, le mot clé `static` est omis lors de la définition de fonctions membres statiques en dehors de la définition de classe.

```
// For Example above, either...
class Example {
    // ...

  public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int  Example::static_func() { return num_instances; }
void Example::set_str(const std::string& str) { static_str = str; }
```

Si une variable membre statique est déclarée `const` mais pas `volatile` et est de type intégrale ou énumération, elle peut être initialisée à la déclaration, à l'intérieur de la définition de classe.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;              // Good.
    static const E ce = VAL;              // Good.
    const static double cd = 5;           // Error.
    static const volatile int cvi = 5;    // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;        // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

C ++ 11

A partir de C ++ 11, les variables membres statiques de types `LiteralType` (types pouvant être construits au moment de la compilation, selon les règles `constexpr`) peuvent également être déclarées en tant que `constexpr` ; Si tel est le cas, ils doivent être initialisés dans la définition de classe.

```
struct ExConstexpr {
    constexpr static int ci = 5;                        // Good.
    static constexpr double cd = 5;                     // Good.
```

```
    constexpr static int carr[] = { 1, 1, 2 };        // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;                      // Error.
};

constexpr int ExConstexpr::bad_ci = 5;                // Still an error.
```

Si une variable membre statique `const` ou `constexpr` est *odr-used* (de manière informelle, si son adresse a été prise ou est affectée à une référence), elle doit toujours avoir une définition distincte, en dehors de la définition de classe. Cette définition n'est pas autorisée à contenir un initialiseur.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

Comme les membres statiques ne sont pas liés à une instance donnée, ils peuvent être accédés en utilisant l'opérateur scope, `::` .

```
std::string str = Example::static_str;
```

Il est également possible d'y accéder comme s'il s'agissait de membres normaux et non statiques. Ceci a une importance historique, mais est utilisé moins souvent que l'opérateur de la portée pour éviter toute confusion quant à savoir si un membre est statique ou non.

```
Example ex;
std::string rts = ex.static_str;
```

Les membres de classe peuvent accéder aux membres statiques sans qualifier leur portée, comme avec les membres de classe non statiques.

```
class ExTwo {
    static int num_instances;
    int my_num;

  public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

Ils ne peuvent pas être `mutable` , et ils ne devraient pas l'être; comme ils ne sont liés à aucune instance donnée, le fait qu'une instance soit ou non constante n'affecte pas les membres statiques.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-
ness.
```

Les membres statiques respectent les modificateurs d'accès, tout comme les membres non statiques.

```
class ExAccess {
    static int prv_int;

  protected:
    static int pro_int;

  public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.
```

Comme ils ne sont pas liés à une instance donnée, les fonctions membres statiques n'ont pas `this` pointeur; De ce fait, ils ne peuvent pas accéder aux variables membres non statiques à moins de passer une instance.

```
class ExInstanceRequired {
    int i;

  public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }                         // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};
```

En raison de l'absence de `this` pointeur, leurs adresses ne peuvent pas être stockées dans des fonctions de pointeur à membre et sont stockées dans des pointeurs à fonctions normaux.

```
struct ExPointer {
           void nsfunc() {}
    static void  sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
    f_ptr p_sf = &ExPointer::sfunc; // Good.
```

En raison de l'absence de `this` pointeur, ils ne peuvent pas non plus être `const` ou `volatile` , ni avoir de qualificatifs ref. Ils ne peuvent pas non plus être virtuels.

```
struct ExCVQualifiersAndVirtual {
    static void   func()                {} // Good.
    static void  cfunc() const          {} // Error.
    static void  vfunc() volatile       {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void  rfunc() &              {} // Error.
    static void rvfunc() &&             {} // Error.

    virtual static void vsfunc()        {} // Error.
    static virtual void svfunc()        {} // Error.
};
```

Comme elles ne sont pas liées à une instance donnée, les variables de membre statiques sont traitées comme des variables globales spéciales. ils sont créés au démarrage du programme et détruits à la fermeture, que des instances de la classe existent ou non. Une seule copie de chaque variable membre statique existe (à moins que la variable soit déclarée `thread_local` (C ++ 11 ou version ultérieure), auquel cas il y a une copie par thread).

Les variables membres statiques ont le même lien que la classe, que la classe ait un lien externe ou interne. Les classes locales et les classes non nommées ne sont pas autorisées à avoir des membres statiques.

Fonctions de membre non statiques

Une classe peut avoir des fonctions membres non statiques qui opèrent sur des instances individuelles de la classe.

```
class CL {
  public:
    void member_function() {}
};
```

Ces fonctions sont appelées sur une instance de la classe, comme ceci:

```
CL instance;
instance.member_function();
```

Ils peuvent être définis à l'intérieur ou à l'extérieur de la définition de classe; Si elles sont définies à l'extérieur, elles sont spécifiées comme étant dans l'étendue de la classe.

```
struct ST {
    void  defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

Ils peuvent être qualifiés de CV et / ou ref-qualifiés , ce qui affecte la façon dont ils voient l'instance sur laquelle ils sont appelés; la fonction verra l'instance comme ayant le ou les qualificatifs cv spécifiés, le cas échéant. La version appelée sera basée sur les qualificatifs cv de l'instance. S'il n'y a pas de version avec les mêmes qualificatifs cv que l'instance, une version qualifiée plus-cv sera appelée si disponible.

```
struct CVQualifiers {
    void func()                   {} // 1: Instance is non-cv-qualified.
    void func() const             {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers       non_cv_instance;
const CVQualifiers      c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func();      // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only();      // Calls const volatile version.
```

C ++ 11

Les qualificateurs de référence de la fonction membre indiquent si la fonction est destinée à être appelée sur les instances rvalue et utilisent la même syntaxe que la fonction cv-qualifiers.

```
struct RefQualifiers {
    void func() &  {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func();              // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

Les qualificateurs de CV et les qualificatifs de référence peuvent également être combinés si nécessaire.

```
struct BothCVAndRef {
    void func() const& {} // Called on normal instances.  Sees instance as const.
    void func() &&     {} // Called on temporary instances.
```

```
};
```

Ils peuvent aussi être virtuels ; Ceci est fondamental pour le polymorphisme et permet à une ou plusieurs classes enfant de fournir la même interface que la classe parente, tout en fournissant leurs propres fonctionnalités.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp.func(); // Calls Base::func().
dp.func(); // Calls Derived::func().
```

Pour plus d'informations, voir ici .

Structure / classe sans nom

Un `struct` *sans nom* est autorisé (le type n'a pas de nom)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

ou

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

et ensuite

```
Circle circle;
circle.center.x = 42.f;
```

mais non *anonyme* `struct` (type sans nom et objet sans nom)

```
struct InvalidCircle
```

```
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Remarque: Certains compilateurs autorisent la `struct` *anonyme en* tant *qu'extension* .

C ++ 11

- *lamdba* peut être vu comme une `struct` spéciale *non nommée* .
- `decltype` permet de récupérer le type de `struct` *sans nom* :

  ```
  decltype(circle.point) otherPoint;
  ```

- *Une* instance de `struct` *sans nom* peut être un paramètre de la méthode template:

  ```
  void print_square_coordinates()
  {
      const struct {float x; float y;} points[] = {
          {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
      };

      // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
      for (const auto& point : points) {
          std::cout << "{" << point.x << ", " << point.y << "}\n";
      }

      decltype(points[0]) topRightCorner{1, 1};
      auto it = std::find(points, points + 4, topRightCorner);
      std::cout << "top right corner is the "
                << 1 + std::distance(points, it) << "th\n";
  }
  ```

Lire Classes / Structures en ligne: https://riptutorial.com/fr/cplusplus/topic/508/classes---structures

Chapitre 12: Compiler et construire

Introduction

Les programmes écrits en C ++ doivent être compilés avant de pouvoir être exécutés. Une grande variété de compilateurs est disponible en fonction de votre système d'exploitation.

Remarques

La plupart des systèmes d'exploitation sont livrés sans compilateur et doivent être installés ultérieurement. Certains choix de compilateurs courants sont les suivants:

- GCC, la collection de compilateurs GNU g ++
- clang: une interface de famille en langage C pour LLVM clang ++
- MSVC, Microsoft Visual C ++ (inclus dans Visual Studio) Visual-C ++
- C ++ Builder, Embarcadero C ++ Builder (inclus dans RAD Studio) Générateur c ++

Veuillez consulter le manuel du compilateur approprié pour savoir comment compiler un programme C ++.

Une autre option pour utiliser un compilateur spécifique avec son propre système de construction spécifique, il est possible de laisser les systèmes de construction génériques configurer le projet pour un compilateur spécifique ou pour celui installé par défaut.

Examples

Compiler avec GCC

En supposant un fichier source unique nommé `main.cpp` , la commande pour compiler et lier un exécutable non optimisé est la suivante (Compiler sans optimisation est utile pour le développement initial et le débogage, bien que `-Og` soit officiellement recommandé pour les nouvelles versions de GCC).

```
g++ -o app -Wall main.cpp -O0
```

Pour produire un exécutable optimisé à utiliser en production, utilisez l’une des options `-O` (voir: `-O1` , `-O2` , `-O3` , `-Os` , `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

Si l'option -O est omise, -O0, ce qui signifie aucune optimisation, est utilisé par défaut (en spécifiant -O sans nombre, le résultat est -O1).

Vous pouvez également utiliser directement les indicateurs d'optimisation des groupes `O` (ou des optimisations plus expérimentales). L'exemple suivant `-O2` avec l'optimisation `-O2` , plus un

indicateur du niveau d'optimisation `-O3` :

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

Pour produire un exécutable optimisé spécifique à la plateforme (à utiliser en production sur la machine avec la même architecture), utilisez:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

L'un ou l'autre des éléments ci-dessus produira un fichier binaire pouvant être exécuté avec `.\app.exe` sous Windows et `./app` sous Linux, Mac OS, etc.

L'indicateur `-o` peut également être ignoré. Dans ce cas, GCC créera le fichier exécutable par défaut `a.exe` sur Windows et `a.out` sur les systèmes de type Unix. Pour compiler un fichier sans le lier, utilisez l'option `-c` :

```
g++ -o file.o -Wall -c file.cpp
```

Cela produit un fichier objet nommé `file.o` qui peut ensuite être lié à d'autres fichiers pour produire un fichier binaire:

```
g++ -o app file.o otherfile.o
```

Vous trouverez plus d'informations sur les options d'optimisation sur gcc.gnu.org . On notera en particulier `-Og` (optimisation mettant l'accent sur l'expérience de débogage - recommandée pour le cycle standard de modification-compilation-débogage) et `-Ofast` (toutes les optimisations, y compris celles ne respectant pas la stricte conformité aux normes).

L'indicateur `-Wall` active des avertissements pour de nombreuses erreurs courantes et doit toujours être utilisé. Pour améliorer la qualité du code , il est souvent aussi encouragé à utiliser `-Wextra` et d' autres drapeaux d'avertissement qui ne sont pas activés automatiquement par `-Wall` et `-Wextra` .

Si le code attend un standard C ++ spécifique, spécifiez le standard à utiliser en incluant l' `-std=` . Les valeurs prises en charge correspondent à l'année de finalisation pour chaque version du standard ISO C ++. A partir de GCC 6.1.0, les valeurs valides pour le drapeau `std=` sont `c++98` / `c++03` , `c++11` , `c++14` et `c++17` / `c++1z` . Les valeurs séparées par une barre oblique sont équivalentes.

```
g++ -std=c++11 <file>
```

GCC inclut des extensions spécifiques au compilateur désactivées lorsqu'elles sont en conflit avec une norme spécifiée par l' `-std=` . Pour compiler avec toutes les extensions activées, la valeur `gnu++XX` peut être utilisée, où `XX` est l'une des années utilisées par les valeurs `c++` mentionnées ci-dessus.

Le standard par défaut sera utilisé si aucun n'est spécifié. Pour les versions de GCC antérieures à

6.1.0, la valeur par défaut est `-std=gnu++03` ; dans GCC 6.1.0 et supérieur, la valeur par défaut est `-std=gnu++14` .

Notez qu'en raison de bogues dans GCC, l'indicateur `-pthread` doit être présent lors de la compilation et de la liaison pour que GCC prenne en charge la fonctionnalité de thread standard C ++ introduite avec C ++ 11, telle que `std::thread` et `std::wait_for` . L'omettre lors de l'utilisation des fonctions de threading peut entraîner aucun avertissement mais des résultats non valides sur certaines plates-formes.

Liaison avec les bibliothèques:

Utilisez l'option `-l` pour passer le nom de la bibliothèque:

```
g++ main.cpp -lpcre2-8
#pcre2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

Si la bibliothèque ne se trouve pas dans le chemin de la bibliothèque standard, ajoutez le chemin avec l'option `-L` :

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Plusieurs bibliothèques peuvent être liées entre elles:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

Si une bibliothèque dépend d'une autre, placez la bibliothèque dépendante **avant** la bibliothèque indépendante:

```
g++ main.cpp -lchild-lib -lbase-lib
```

Ou laissez l'éditeur de liens déterminer lui `--start-group` même l'ordre via `--start-group` et `--end-group` (remarque: cela a un coût de performance significatif):

```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Compilation avec Visual C ++ (ligne de commande)

Pour les programmeurs venant de GCC ou Clang vers Visual Studio ou les programmeurs plus à l'aise avec la ligne de commande en général, vous pouvez utiliser le compilateur Visual C ++ à partir de la ligne de commande ainsi que l'EDI.

Si vous souhaitez compiler votre code à partir de la ligne de commande dans Visual Studio, vous devez d'abord configurer l'environnement de ligne de commande. Cela peut être effectué en ouvrant l' `Visual Studio Command Prompt` / l' `Visual Studio Command Prompt Developer Command Prompt` / l' `Visual Studio Command Prompt Developer Command Prompt x86 Native Tools Command Prompt` / l' `Developer Command Prompt x86 Native Tools Command Prompt x64 Native Tools Command Prompt` ou similaire

(fournie par votre version de Visual Studio) ou à l'invite de commande. le sous-répertoire `VC` répertoire d'installation du compilateur (généralement `\Program Files (x86)\Microsoft Visual Studio x\VC` , où `x` est le numéro de version (tel que `10.0` pour 2010 ou `14.0` pour 2015) et exécute le fichier de commandes `VCVARSALL` avec un paramètre de ligne de commande spécifié ici .

Notez que contrairement à GCC, Visual Studio ne fournit pas de front-end pour l'éditeur de liens (`link.exe`) via le compilateur (`cl.exe`), mais fournit l'éditeur de liens en tant que programme distinct, appelé par le compilateur à sa sortie. `cl.exe` et `link.exe` peuvent être utilisés séparément avec différents fichiers et options, ou `cl` peut transmettre des fichiers et des options à `link` si les deux tâches sont effectuées ensemble. Toutes les options de liaison spécifiées pour `cl` seront converties en options de `link` , et tous les fichiers non traités par `cl` seront transmis directement au `link` . Comme il s'agit principalement d'un guide simple de compilation avec la ligne de commande Visual Studio, les arguments pour le `link` ne seront pas décrits pour le moment; si vous avez besoin d'une liste, voir ici .

Notez que les arguments de `cl` sont sensibles à la casse, alors que les arguments à `link` ne le sont pas.

[Soyez averti que certains des exemples suivants utilisent la variable "Répertoire actuel" du shell Windows, `%cd%` , lors de la spécification des noms de chemin absolus. Pour toute personne peu familière avec cette variable, elle se développe dans le répertoire de travail en cours. À partir de la ligne de commande, ce sera le répertoire dans lequel vous vous trouviez lorsque vous avez exécuté `cl` , et est spécifié par défaut dans l'invite de commande (si votre invite de commande est `C:\src>` , par exemple, `%cd%` est `C:\src\`).]

En supposant qu'un fichier source unique nommé `main.cpp` dans le dossier en cours, la commande permettant de compiler et de lier un exécutable non optimisé (utile pour le développement initial et le débogage) est (utilisez l'une des méthodes suivantes):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".

cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

En supposant un fichier source supplémentaire "niam.cpp" dans le même répertoire, utilisez ce qui suit:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Vous pouvez également utiliser des caractères génériques, comme on peut s'y attendre:

```
cl main.cpp src\*.cpp
```

```
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
//  "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
// Generates executable "main.exe".
```

Pour renommer ou déplacer l'exécutable, utilisez l'une des options suivantes:

```
cl /o name main.cpp
// Generates executable named "name.exe".

cl /o folder\ main.cpp
// Generates executable named "main.exe", in folder "%cd%\folder".

cl /o folder\name main.cpp
// Generates executable named "name.exe", in folder "%cd%\folder".

cl /Fename main.cpp
// Same as "/o name".

cl /Fefolder\ main.cpp
// Same as "/o folder\".

cl /Fefolder\name main.cpp
// Same as "/o folder\name".
```

Les deux `/o` et `/Fe` passent leur paramètre (appelons - le `o-param`) pour `link` comme `/OUT:o-param` , annexant l'extension appropriée (généralement `.exe` ou `.dll`) au "nom" `o-param` s au besoin. Alors que `/o` et `/Fe` sont à ma connaissance identiques en termes de fonctionnalités, ce dernier est préféré pour Visual Studio. `/o` est marqué comme obsolète, et semble être principalement fourni pour les programmeurs plus familiers avec GCC ou Clang.

Notez que même si l'espace entre `/o` et le dossier et / ou le nom spécifié est facultatif, il *ne doit pas y* avoir d'espace entre `/Fe` et le dossier et / ou le nom spécifiés.

De même, pour produire un exécutable optimisé (à utiliser en production), utilisez:

```
cl /O1 main.cpp
// Optimise for executable size.  Produces small programs, at the possible expense of slower
//  execution.

cl /O2 main.cpp
// Optimise for execution speed.  Produces fast programs, at the possible expense of larger
//  file size.

cl /GL main.cpp other.cpp
// Generates special object files used for whole-program optimisation, which allows CL to
//  take every module (translation unit) into consideration during optimisation.
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during
//  the linking phase to perform additional optimisations.  If linking is not performed at
this
//  time, the generated object files should be linked with "/LTCG".
// Can be used with other CL optimisation options.
```

Enfin, pour créer un exécutable optimisé spécifique à la plate-forme (à utiliser en production sur la machine avec l'architecture spécifiée), choisissez l'invite de commande ou le paramètre `VCVARSALL` pour la plate-forme cible. `link` devrait détecter la plate-forme souhaitée à partir des fichiers objets; Sinon, utilisez l' option `/MACHINE` pour spécifier explicitement la plate-forme cible.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
cl main.cpp /link /machine:X64
```

Tout ce qui précède produira un exécutable avec le nom spécifié par `/o` ou `/Fe` , ou si aucun n'est fourni, avec un nom identique au premier fichier source ou objet spécifié pour le compilateur.

```
cl a.cpp b.cpp c.cpp
// Generates "a.exe".

cl d.obj a.cpp q.cpp
// Generates "d.exe".

cl y.lib n.cpp o.obj
// Generates "n.exe".

cl /o yo zp.obj pz.cpp
// Generates "yo.exe".
```

Pour compiler un fichier sans liaison, utilisez:

```
cl /c main.cpp
// Generates object file "main.obj".
```

Cela indique à `cl` de quitter sans appeler le `link` et produit un fichier objet, qui peut ensuite être lié à d'autres fichiers pour produire un fichier binaire.

```
cl main.obj niam.cpp
// Generates object file "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".

link main.obj niam.obj
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Il existe également d'autres paramètres de ligne de commande précieux, qu'il serait très utile de connaître pour les utilisateurs:

```
cl /EHsc main.cpp
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,
//  and `extern "C"` functions will not throw exceptions.
// This is recommended when writing portable, platform-independent code.

cl /clr main.cpp
// "/clr" specifies that the code should be compiled to use the common language runtime,
//  the .NET Framework's virtual machine.
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,
```

```
//  and creates an executable that requires .NET to run.

cl /Za main.cpp
// "/Za" specifies that Microsoft extensions should be disabled, and code should be
//  compiled strictly according to ISO C++ specifications.
// This is recommended for guaranteeing portability.

cl /Zi main.cpp
// "/Zi" generates a program database (PDB) file for use when debugging a program, without
//  affecting optimisation specifications, and passes the option "/DEBUG" to LINK.

cl /LD dll.cpp
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling
those
//  programs.

cl main.cpp /link /LINKER_OPTION
// "/link" passes everything following it directly to LINK, without parsing it in any way.
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

Pour ceux qui connaissent mieux les systèmes * nix et / ou GCC / Clang, les outils de ligne de commande `cl` , `link` et autres Visual Studio peuvent accepter les paramètres spécifiés avec un trait d'union (tel que `-c`) au lieu d'une barre oblique (tel que `/c`). En outre, Windows reconnaît une barre oblique ou une barre oblique inverse comme séparateur de chemin d'accès valide. Par conséquent, les chemins d'accès de style * nix peuvent également être utilisés. Cela facilite la conversion de lignes de commande simples du compilateur de `g++` ou `clang++` en `cl` , ou vice versa, avec des modifications minimales.

```
g++ -o app src/main.cpp
cl  -o app src/main.cpp
```

Bien entendu, lorsque vous portez des lignes de commande utilisant des options `g++` ou `clang++` plus complexes, vous devez rechercher des commandes équivalentes dans les documentations de compilateurs et / ou sur les sites de ressources, mais cela facilite le démarrage nouveaux compilateurs.

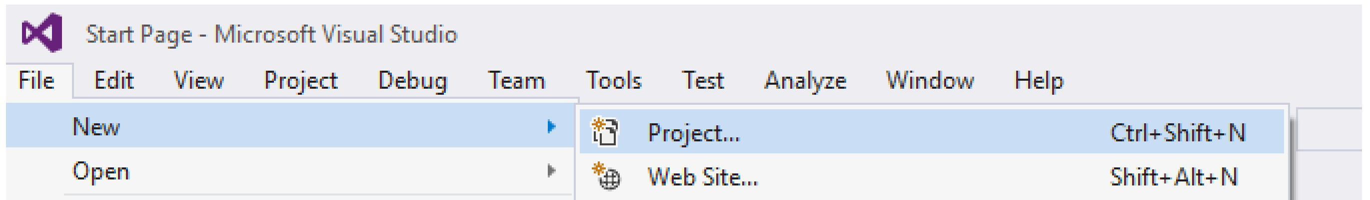
Si vous avez besoin de fonctionnalités linguistiques spécifiques pour votre code, une version spécifique de MSVC était requise. À partir de Visual C ++ 2015 Update 3 , il est possible de choisir la version du standard à compiler via l'indicateur `/std` . Les valeurs possibles sont `/std:c++14` et `/std:c++latest` (`/std:c++17` suivra bientôt).

Remarque: Dans les anciennes versions de ce compilateur, des indicateurs de fonctionnalités spécifiques étaient disponibles, mais cette option était principalement utilisée pour les aperçus de nouvelles fonctionnalités.

Compiler avec Visual Studio (interface graphique) - Hello World

1. Téléchargez et installez Visual Studio Community 2015
2. Open Visual Studio Community

3. Cliquez sur Fichier -> Nouveau -> Projet



4. Cliquez sur Modèles -> Visual C ++ -> Application Console Win32, puis nommez le projet **MyFirstProgram** .

New Project

Recent
Installed
Templates
Visual C#
Visual Basic
Visual F#
Visual C++
Windows
ATL
CLR
General
MFC
Test
Win32
Multiplateforme
Extensibility
XLL+
SQL Server
Intel(R) Visual Fortran
Python
AWS
JavaScript
HDInsight
Online

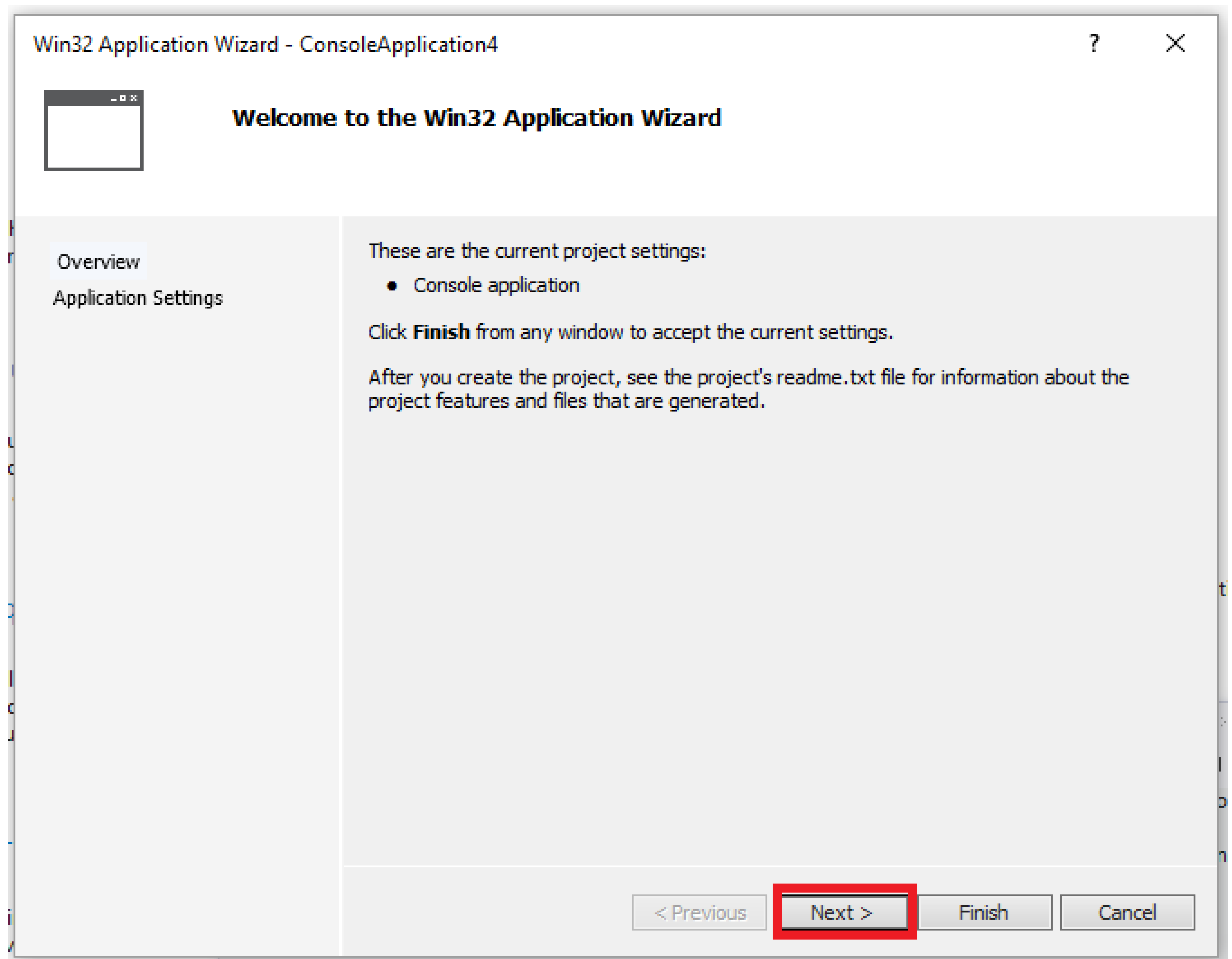
.NET Framework 4.5 Sort by: Default

Win32 Console Application Visual C++
Win32 Project Visual C++

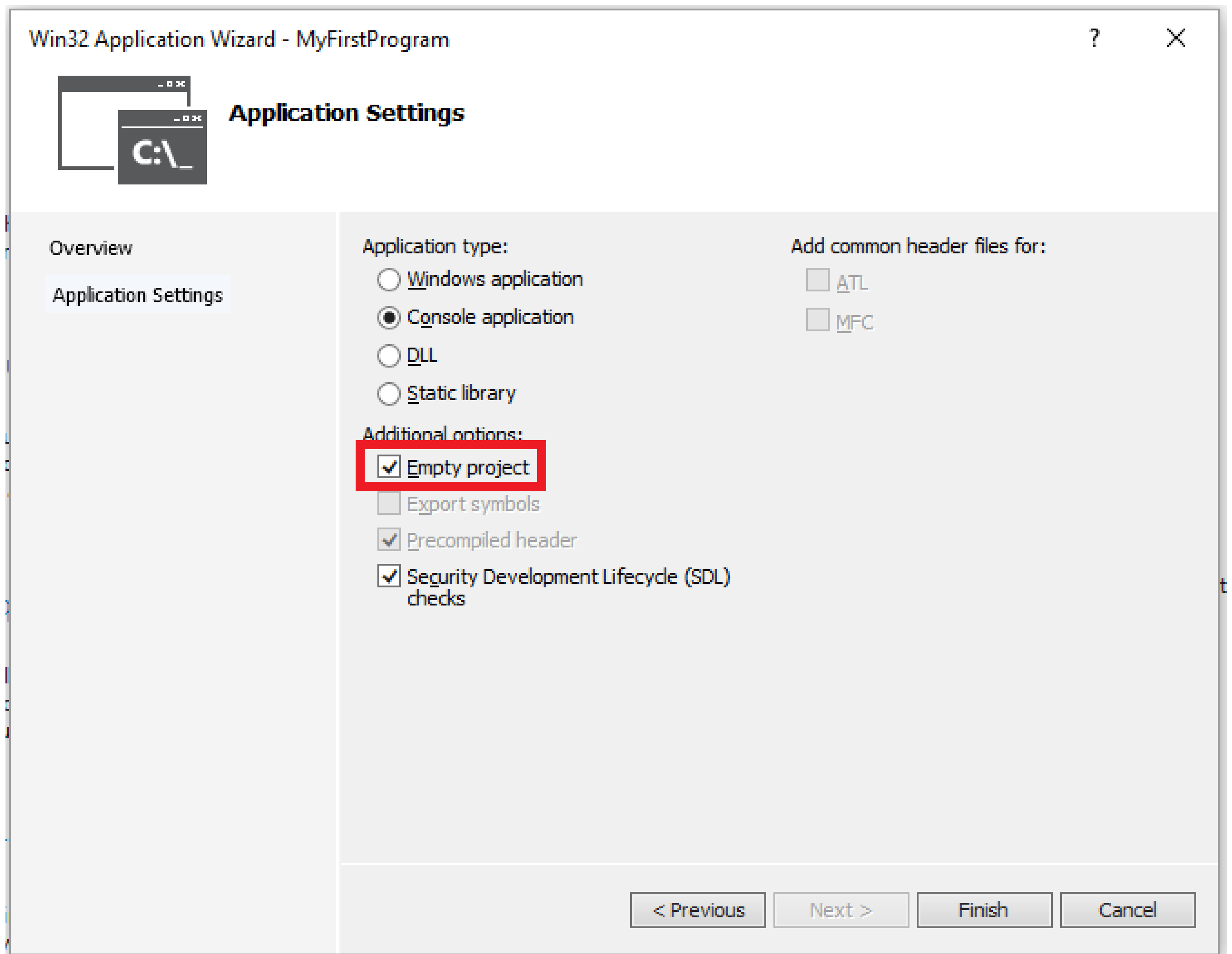
Click here to go online and find templates.

Name: MyFirstProgram
Location: C:\Users\
Solution name: MyFirstProgram

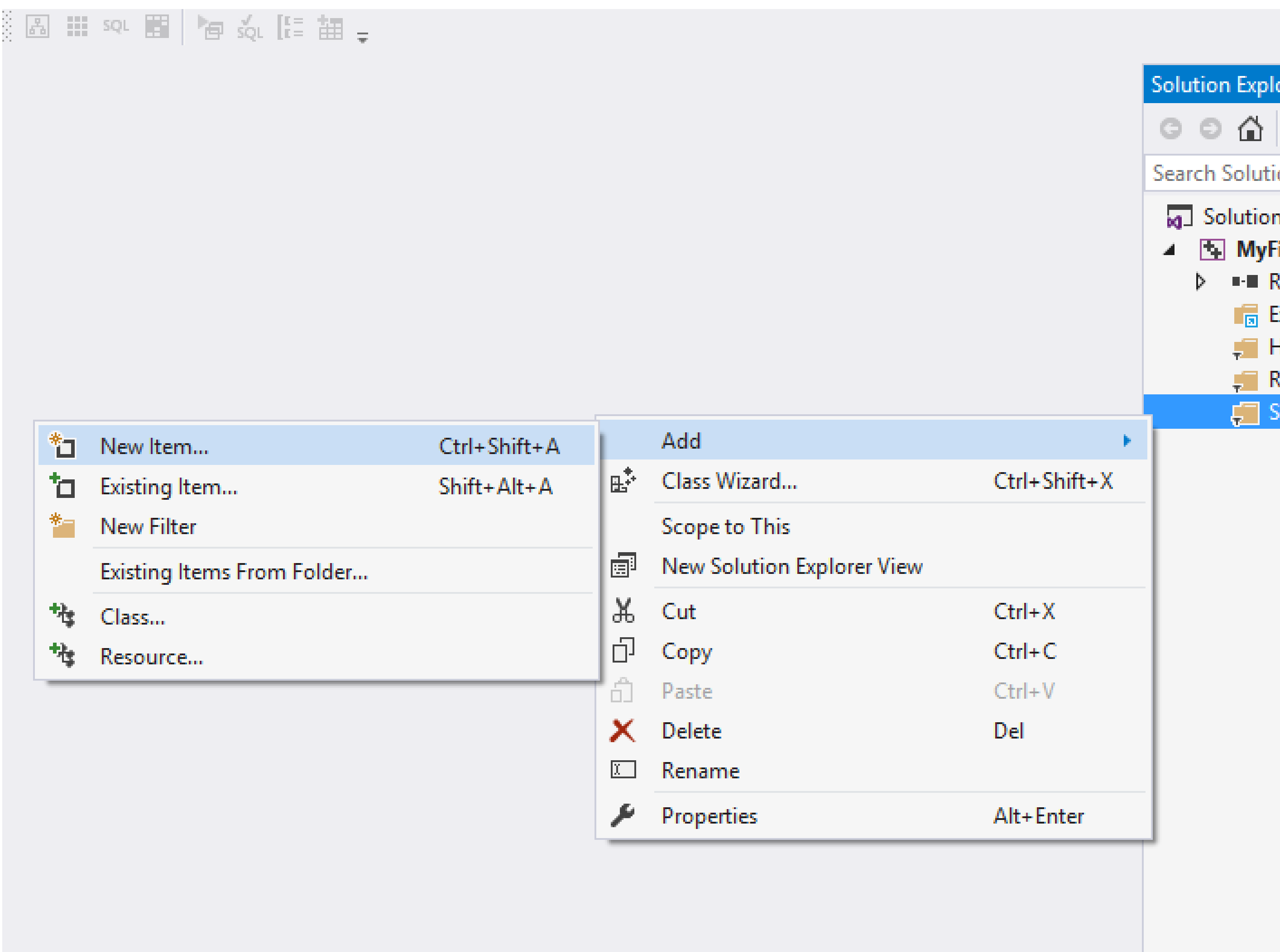
5. Cliquez sur OK
6. Cliquez sur Suivant dans la fenêtre suivante.



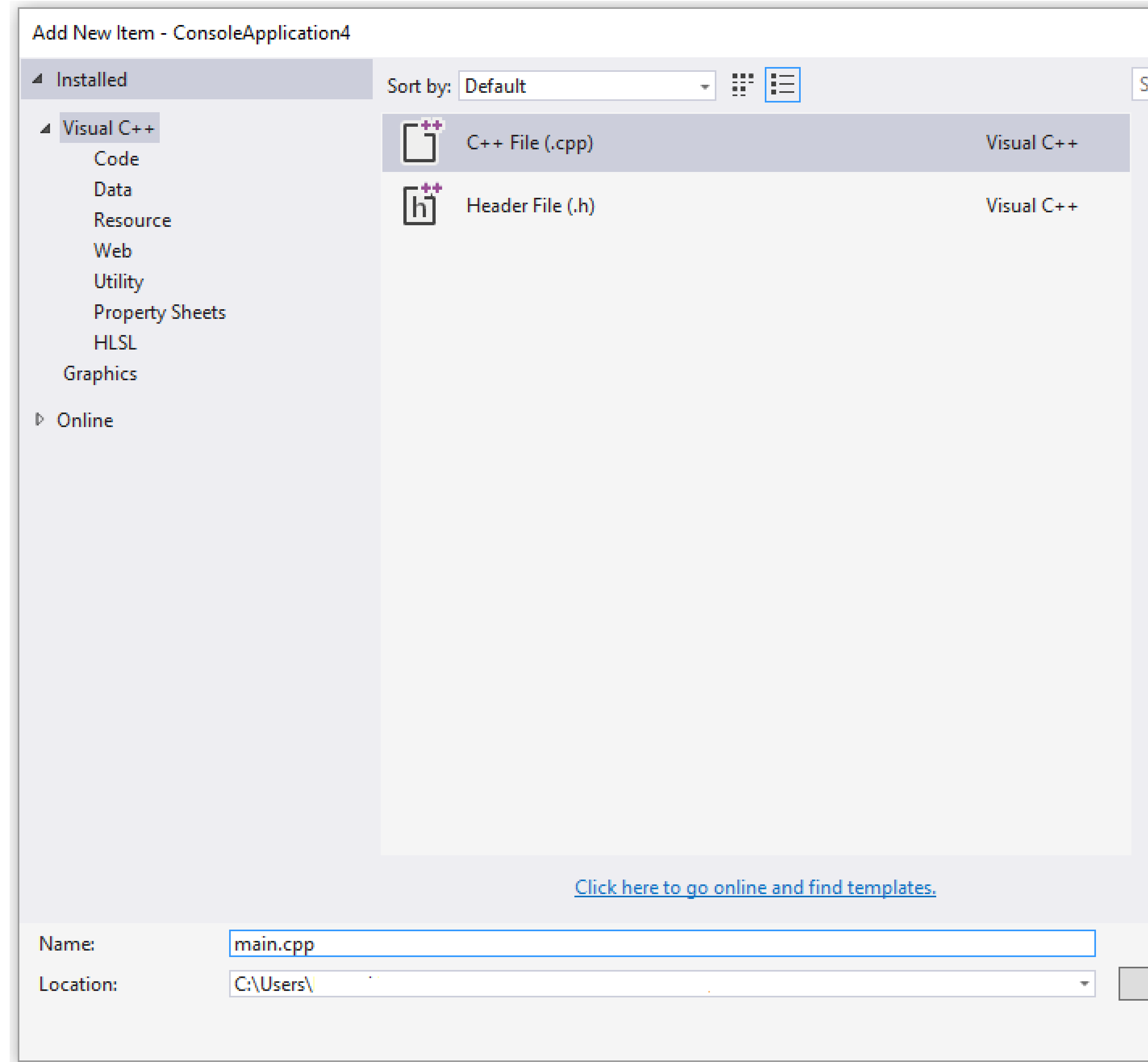
7. Cochez la case `Empty project` , puis cliquez sur Terminer:



8. Faites un clic droit sur le dossier Fichier source puis -> Ajouter -> Nouvel élément:



9. Sélectionnez Fichier C ++ et nommez le fichier main.cpp, puis cliquez sur Ajouter:



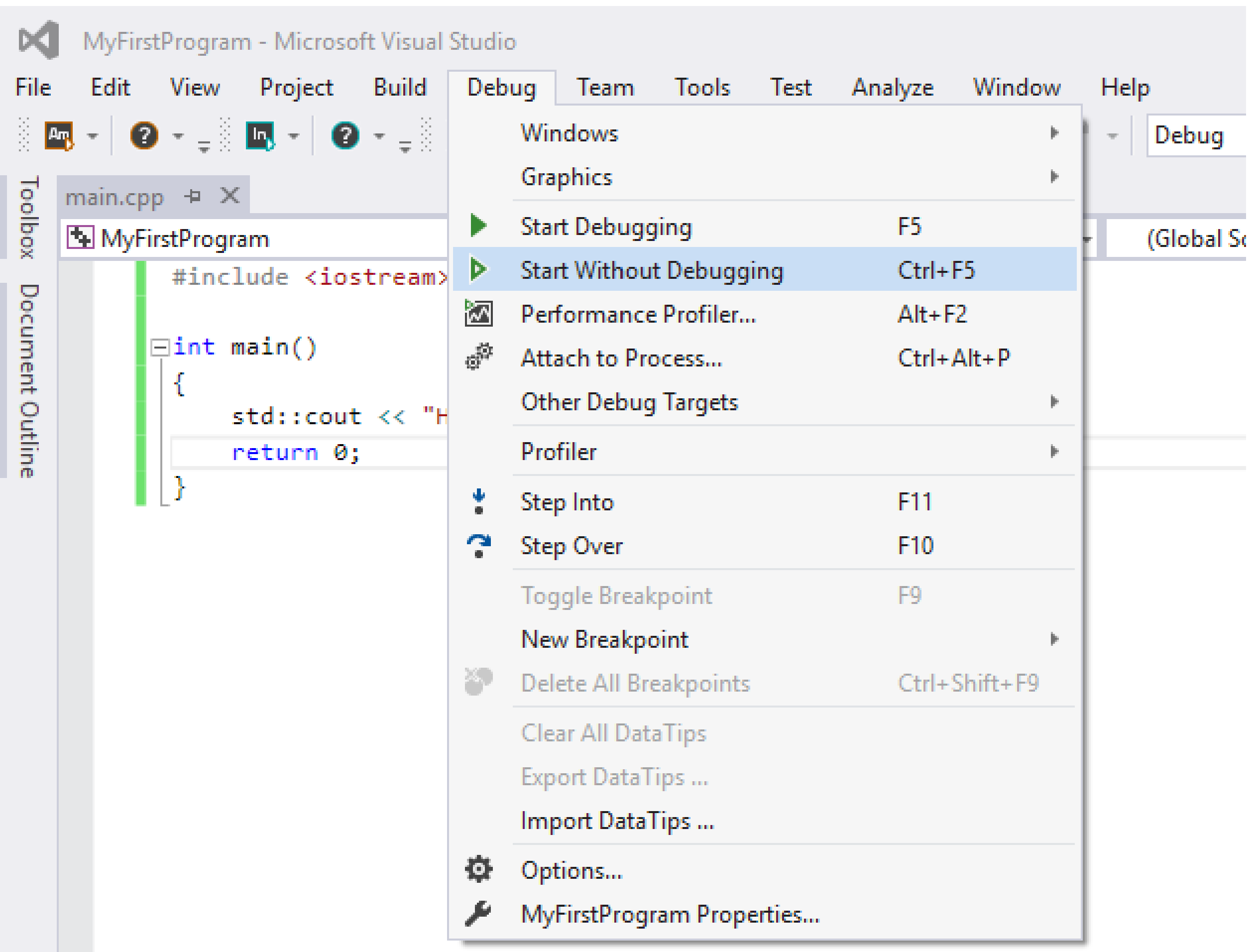
10: Copiez et collez le code suivant dans le nouveau fichier main.cpp:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

Votre environnement devrait ressembler à:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
    return 0;
}
```

11. Cliquez sur Debug -> Start **Without** Debugging (ou appuyez sur Ctrl + F5):



12. Terminé. Vous devriez obtenir la sortie de console suivante:

Compiler avec Clang

Le frontal de Clang étant conçu pour être compatible avec GCC, la plupart des programmes pouvant être compilés via GCC seront compilés lorsque vous `clang++` `g++` par `clang++` dans les scripts de construction. Si aucune `-std=version` n'est donnée, gnu11 sera utilisé.

Les utilisateurs Windows habitués à MSVC peuvent échanger `cl.exe` avec `clang-cl.exe` . Par défaut, clang essaie d'être compatible avec la version la plus élevée de MSVC installée.

Dans le cas d'une compilation avec Visual Studio, clang-cl peut être utilisé en modifiant le `Platform toolset` dans les propriétés du projet.

Dans les deux cas, clang est uniquement compatible via son front-end, bien qu'il tente également de générer des fichiers d'objets compatibles binaires. Les utilisateurs de clang-cl doivent noter que la compatibilité avec MSVC n'est pas encore complète .

Pour utiliser clang ou clang-cl, on pourrait utiliser l'installation par défaut sur certaines distributions Linux ou celles fournies avec des IDE (comme XCode sur Mac). Pour les autres versions de ce compilateur ou sur les plates-formes sur lesquelles ce n'est pas installé, vous pouvez le télécharger depuis la page de téléchargement officielle .

Si vous utilisez CMake pour créer votre code, vous pouvez généralement changer le compilateur en définissant les variables d'environnement `CC` et `CXX` comme ceci:

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

Voir aussi introduction à Cmake .

Compilateurs en ligne

Divers sites Web offrent un accès en ligne aux compilateurs C ++. Les fonctionnalités du compilateur en ligne varient considérablement d'un site à l'autre, mais elles permettent généralement d'effectuer les opérations suivantes:

- Collez votre code dans un formulaire Web dans le navigateur.
- Sélectionnez des options de compilation et compilez le code.
- Collecte les sorties du compilateur et / ou du programme.

Le comportement du site Web du compilateur en ligne est généralement assez restrictif car il permet à quiconque d'exécuter des compilateurs et d'exécuter du code arbitraire côté serveur, alors que l'exécution de code arbitraire à distance est généralement considérée comme une vulnérabilité.

Les compilateurs en ligne peuvent être utiles aux fins suivantes:

- Exécutez un petit extrait de code à partir d'une machine dépourvue du compilateur C ++ (smartphones, tablettes, etc.).
- Assurez-vous que le code compile correctement avec différents compilateurs et s'exécute de la même manière quel que soit le compilateur avec lequel il a été compilé.
- Apprendre ou enseigner les bases du C ++.
- Découvrez les fonctionnalités C ++ modernes (C ++ 14 et C ++ 17 dans un futur proche) lorsque le compilateur C ++ à jour n'est pas disponible sur la machine locale.
- Repérez un bogue dans votre compilateur en le comparant à un grand nombre d’autres compilateurs. Vérifiez si un bogue du compilateur a été corrigé dans les futures versions, qui ne sont pas disponibles sur votre ordinateur.
- Résoudre des problèmes de juge en ligne.

Quels compilateurs en ligne **ne** devraient **pas** être utilisés pour:

- Développez des applications complètes (même petites) en utilisant C ++. Généralement, les compilateurs en ligne ne permettent pas de créer des liens avec des bibliothèques tierces ou de télécharger des artefacts de génération.
- Effectuer des calculs intensifs. Les ressources informatiques côté serveur étant limitées, tout programme fourni par l'utilisateur sera tué après quelques secondes d'exécution. Le temps d'exécution autorisé est généralement suffisant pour les tests et l'apprentissage.
- Attaquez le serveur de compilateur lui-même ou tout hôte tiers sur le net.

Exemples:

> Déni de responsabilité: les auteurs de la documentation ne sont affiliés à aucune des ressources énumérées ci-dessous. Les sites Web sont classés par ordre alphabétique.

- http://codepad.org/ Compilateur en ligne avec partage de code. La modification du code après la compilation avec un avertissement ou une erreur de code source ne fonctionne pas si bien.
- http://coliru.stacked-crooked.com/ Compilateur en ligne pour lequel vous spécifiez la ligne de commande. Fournit des compilateurs GCC et Clang à utiliser.
- http://cpp.sh/ - Compilateur en ligne avec support C ++ 14. Ne vous permet pas d'éditer la

ligne de commande du compilateur, mais certaines options sont disponibles via les contrôles de l'interface graphique.

- https://gcc.godbolt.org/ - Fournit une liste étendue des versions du compilateur, des architectures et des résultats de désassemblage. Très utile lorsque vous devez inspecter ce que votre code compile en différents compilateurs. GCC, Clang, MSVC (`CL`), le compilateur Intel (`icc`), ELLCC et Zapcc sont présents, avec un ou plusieurs de ces compilateurs disponibles pour l'ARM, ARMv8 (comme ARM64), Atmel AVR, MIPS, MIPS64, MSP430, PowerPC. , architectures x86 et x64. Les arguments de la ligne de commande du compilateur peuvent être modifiés.
- https://ideone.com/ - Largement utilisé sur le Net pour illustrer le comportement des extraits de code. Fournit GCC et Clang pour une utilisation, mais ne vous permet pas de modifier la ligne de commande du compilateur.
- http://melpon.org/wandbox - Prend en charge de nombreuses versions du compilateur Clang et GNU / GCC.
- http://onlinegdb.com/ - Un IDE extrêmement minimaliste qui inclut un éditeur, un compilateur (gcc) et un débogueur (gdb).
- http://rextester.com/ - Fournit des compilateurs Clang, GCC et Visual Studio pour C et C ++ (ainsi que des compilateurs pour d'autres langages), avec la bibliothèque Boost disponible pour utilisation.
- http://tutorialspoint.com/compile_cpp11_online.php - Un shell UNIX complet avec GCC et un explorateur de projet convivial.
- http://webcompiler.cloudapp.net/ - Compilateur en ligne Visual Studio 2015, fourni par Microsoft dans le cadre de RiSE4fun.

Le processus de compilation C ++

Lorsque vous développez un programme C ++, l'étape suivante consiste à compiler le programme avant de l'exécuter. La compilation est le processus qui convertit le programme écrit dans un langage lisible par l'homme, comme C, C ++, etc., en un code machine, directement compris par l'unité centrale de traitement. Par exemple, si vous avez un fichier de code source C ++ nommé prog.cpp et que vous exécutez la commande de compilation,

```
g++ -Wall -ansi -o prog prog.cpp
```

La création d'un fichier exécutable à partir du fichier source comporte quatre étapes principales.

1. Le préprocesseur C ++ utilise un fichier de code source C ++ et gère les en-têtes (#include), les macros (#define) et les autres directives de préprocesseur.

2. Le fichier de code source C ++ développé par le préprocesseur C ++ est compilé dans le langage d'assemblage de la plateforme.

3. Le code assembleur généré par le compilateur est assemblé dans le code objet de la plate-forme.

4. Le fichier de code objet produit par l'assembleur est lié entre eux
avec les fichiers de code objet pour toutes les fonctions de bibliothèque utilisées pour

produire une bibliothèque ou un fichier exécutable.

Prétraitement

Le préprocesseur gère les directives de préprocesseur, telles que #include et #define. Il est agnostique de la syntaxe de C ++, c'est pourquoi il doit être utilisé avec précaution.

Il fonctionne sur un fichier source C ++ à la fois en remplaçant les directives #include par le contenu des fichiers respectifs (qui ne sont généralement que des déclarations), en remplaçant les macros (#define) et en sélectionnant différentes parties du texte en fonction de #if, Directives #ifdef et #ifndef.

Le préprocesseur fonctionne sur un flux de jetons de prétraitement. La substitution de macros est définie comme le remplacement de jetons par d'autres jetons (l'opérateur ## permet de fusionner deux jetons lorsque cela se justifie).

Après tout cela, le préprocesseur produit une seule sortie qui est un flux de jetons résultant des transformations décrites ci-dessus. Il ajoute également des marqueurs spéciaux indiquant au compilateur d'où provient chaque ligne, afin de pouvoir les utiliser pour générer des messages d'erreur sensibles.

Certaines erreurs peuvent être produites à ce stade avec une utilisation intelligente des directives #if et #error.

En utilisant l'indicateur de compilation ci-dessous, nous pouvons arrêter le processus à l'étape de prétraitement.

```
g++ -E prog.cpp
```

Compilation

L'étape de compilation est effectuée sur chaque sortie du préprocesseur. Le compilateur analyse le code source C ++ pur (maintenant sans directives du préprocesseur) et le convertit en code assembleur. Invoque ensuite le back-end sous-jacent (assembleur dans toolchain) qui assemble ce code en code machine produisant un fichier binaire réel dans un certain format (ELF, COFF, a.out, ...). Ce fichier objet contient le code compilé (sous forme binaire) des symboles définis dans l'entrée. Les symboles dans les fichiers objets sont désignés par leur nom.

Les fichiers d'objets peuvent faire référence à des symboles non définis. C'est le cas lorsque vous utilisez une déclaration et ne la définissez pas. Cela ne dérange pas le compilateur et produira avec plaisir le fichier objet tant que le code source est bien formé.

Les compilateurs vous laissent généralement arrêter la compilation à ce stade. Ceci est très utile car vous pouvez compiler chaque fichier de code source séparément. L'avantage que cela procure est que vous n'avez pas besoin de tout recompiler si vous ne modifiez qu'un seul fichier.

Les fichiers d'objet produits peuvent être placés dans des archives spéciales appelées bibliothèques statiques, pour une réutilisation plus facile par la suite.

C'est à ce stade que sont signalées les erreurs "régulières" du compilateur, telles que les erreurs de syntaxe ou les erreurs de résolution des surcharges.

Afin d'arrêter le processus après l'étape de compilation, nous pouvons utiliser l'option -S:

```
g++ -Wall -ansi -S prog.cpp
```

Assemblage

L'assembleur crée le code objet. Sur un système UNIX, vous pouvez voir des fichiers avec un suffixe .o (.OBJ sur MSDOS) pour indiquer les fichiers de code objet. Au cours de cette phase, l'assembleur convertit ces fichiers objets du code assembleur en instructions de niveau machine et le fichier créé est un code objet relogeable. Par conséquent, la phase de compilation génère le programme objet relogeable et ce programme peut être utilisé à différents endroits sans avoir à compiler à nouveau.

Pour arrêter le processus après l'étape d'assemblage, vous pouvez utiliser l'option -c:

```
g++ -Wall -ansi -c prog.cpp
```

Mise en relation

L'éditeur de liens est ce qui produit la sortie finale de la compilation à partir des fichiers objets produits par l'assembleur. Cette sortie peut être soit une bibliothèque partagée (ou dynamique) (et même si le nom est similaire, ils n’ont pas beaucoup de points communs avec les bibliothèques statiques mentionnées précédemment) ou un exécutable.

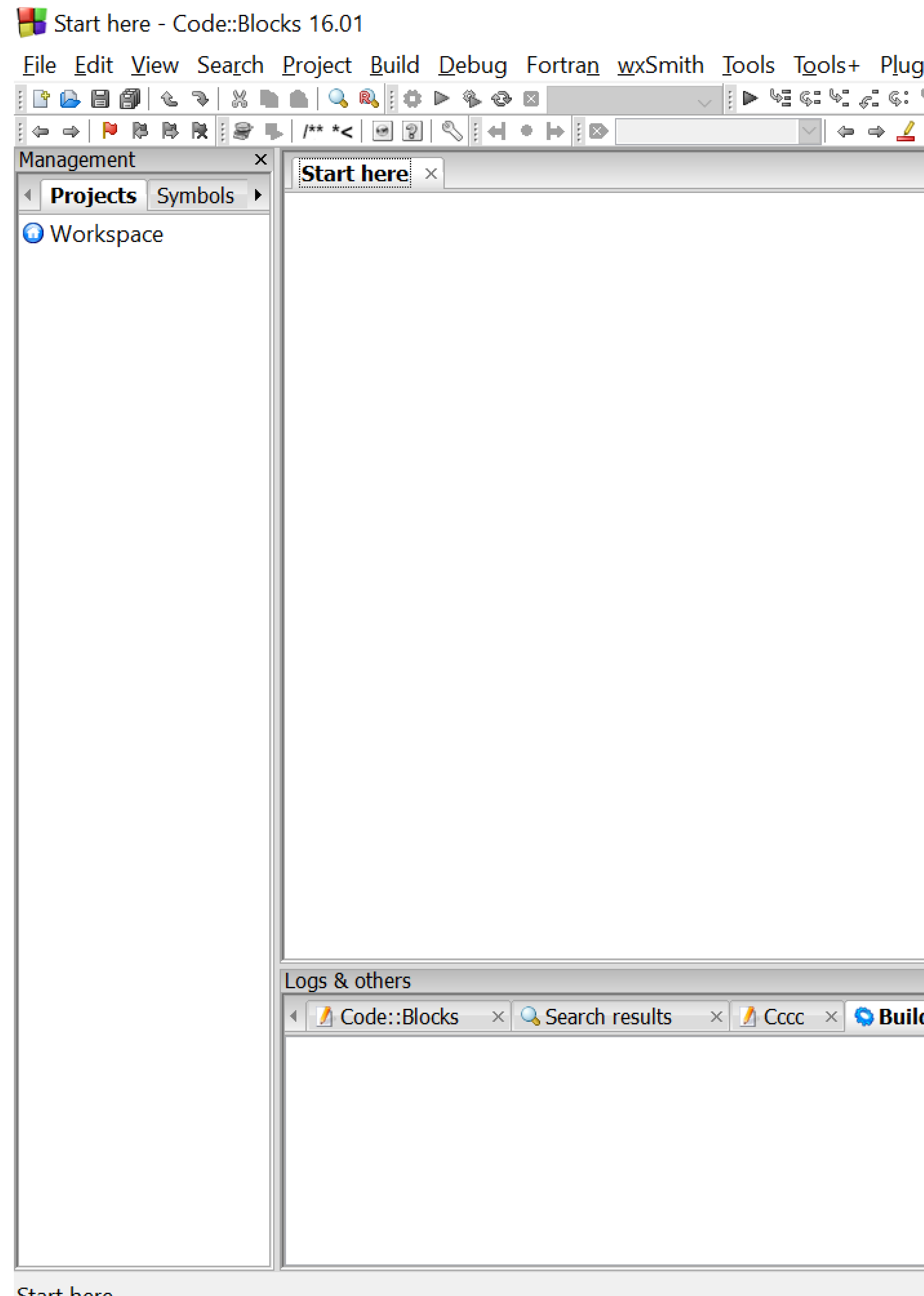
Il lie tous les fichiers objets en remplaçant les références à des symboles non définis par les adresses correctes. Chacun de ces symboles peut être défini dans d'autres fichiers objets ou dans des bibliothèques. S'ils sont définis dans des bibliothèques autres que la bibliothèque standard, vous devez en informer l'éditeur de liens.

À ce stade, les erreurs les plus courantes sont les définitions manquantes ou les définitions en double. Le premier signifie que les définitions n'existent pas (c'est-à-dire qu'elles ne sont pas écrites), ou que les fichiers d'objets ou les bibliothèques où ils résident ne sont pas fournis à l'éditeur de liens. Ce dernier est évident: le même symbole a été défini dans deux fichiers ou bibliothèques d'objets différents.

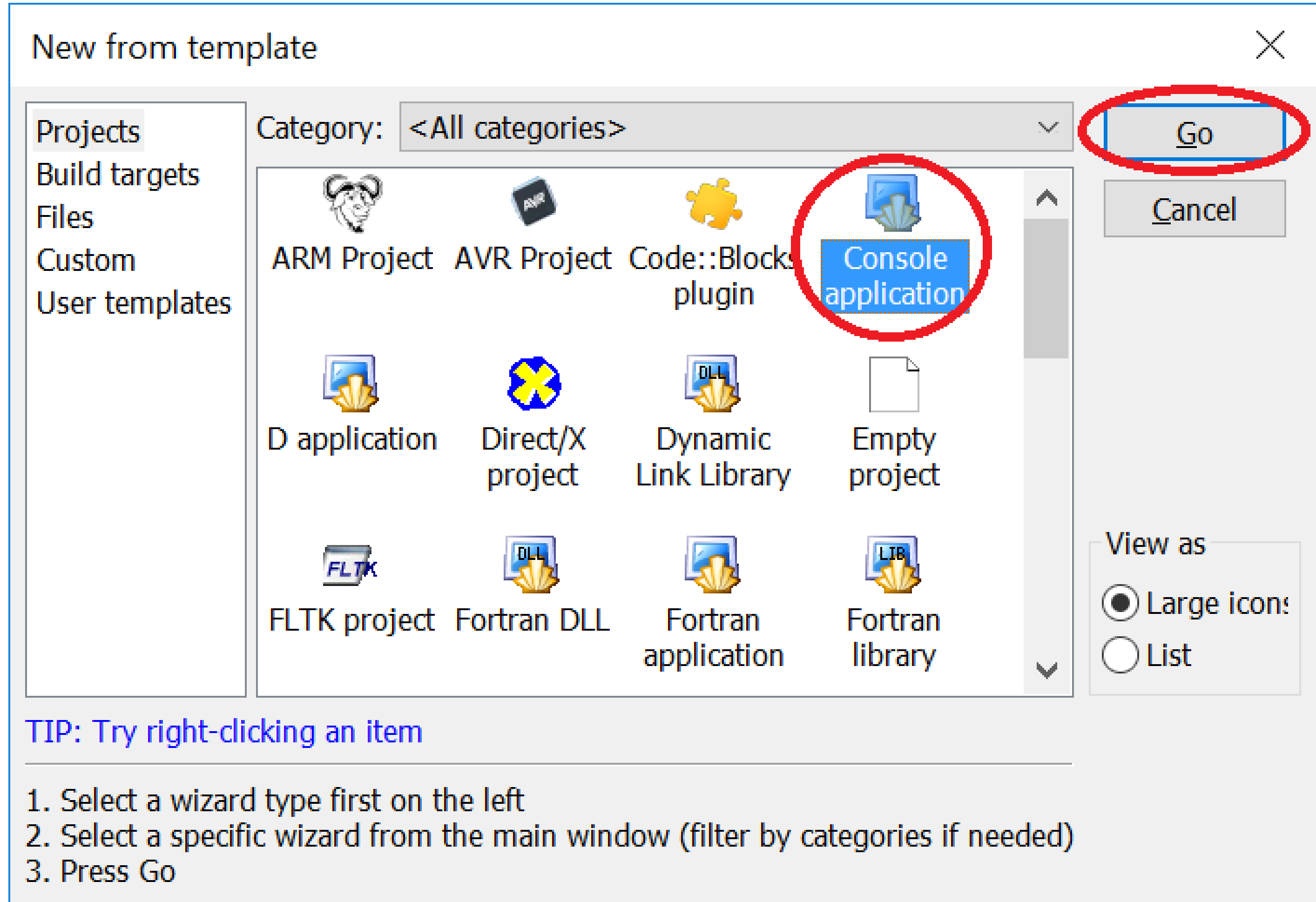
Compiler avec Code :: Blocks (interface graphique)

1. Téléchargez et installez Code :: Blocks ici . Si vous êtes sous Windows, veillez à sélectionner un fichier pour lequel le nom contient `mingw` , les autres fichiers n’installent aucun compilateur.

2. Ouvrez Code :: Blocks et cliquez sur "Créer un nouveau projet":

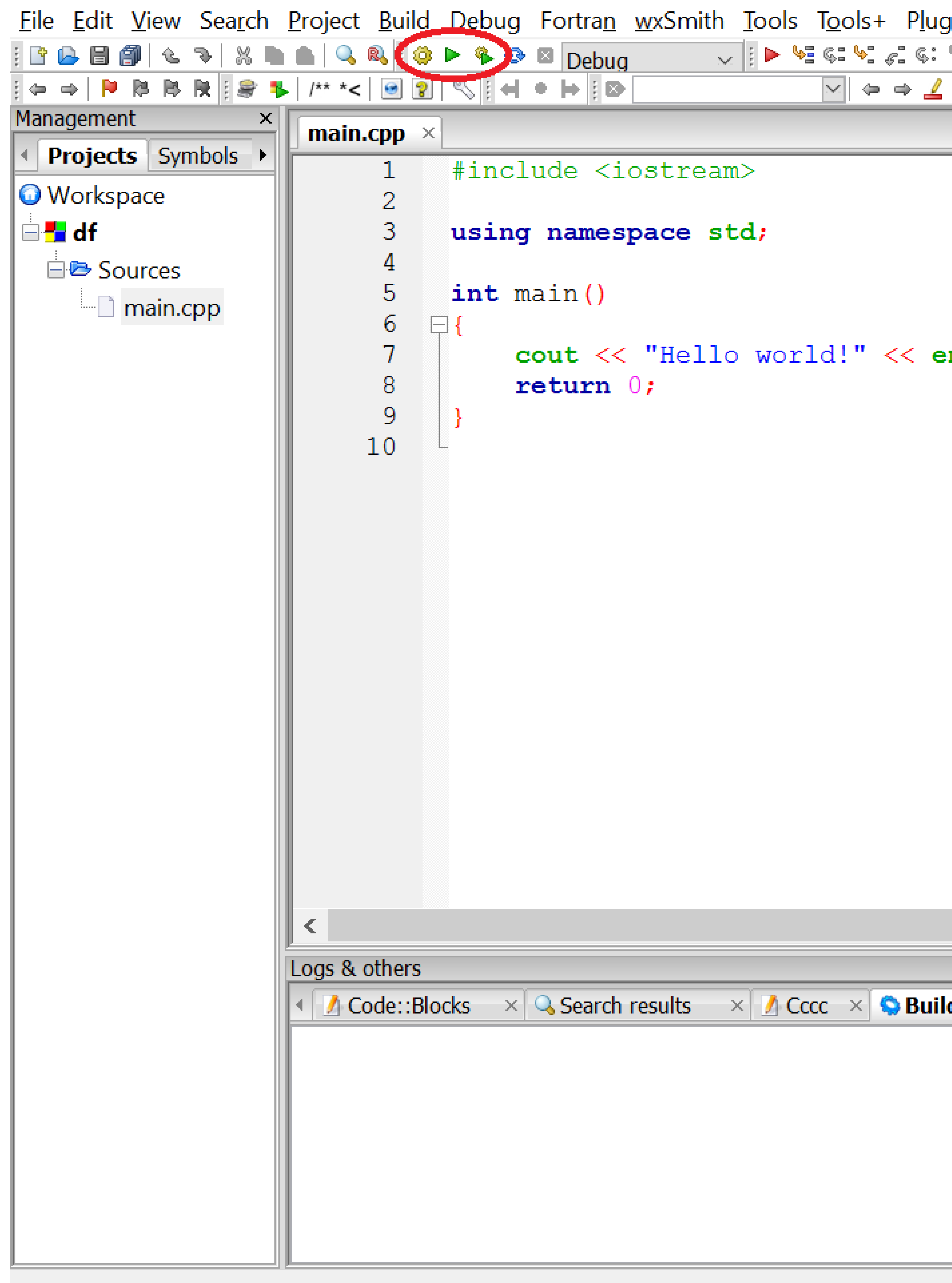

Start here - Code::Blocks 16.01
File
Edit
View
Search
Project
Build
Debug
Fortran
wxSmith
Tools
Tools+
Management
Projects
Symbols
Workspace
Start here
Logs & others
Code::Blocks
Search results
Cccc
Start here

3. Sélectionnez "Application console" et cliquez sur "Go":



4. Cliquez sur "Suivant", sélectionnez "C ++", cliquez sur "Suivant", sélectionnez un nom pour votre projet et choisissez un dossier pour l'enregistrer, cliquez sur "Suivant" puis cliquez sur "Terminer".

5. Maintenant, vous pouvez éditer et compiler votre code. Un code par défaut qui affiche "Hello world!" dans la console est déjà là. Pour compiler et / ou exécuter votre programme, appuyez sur l’un des trois boutons de compilation / exécution de la barre d’outils:

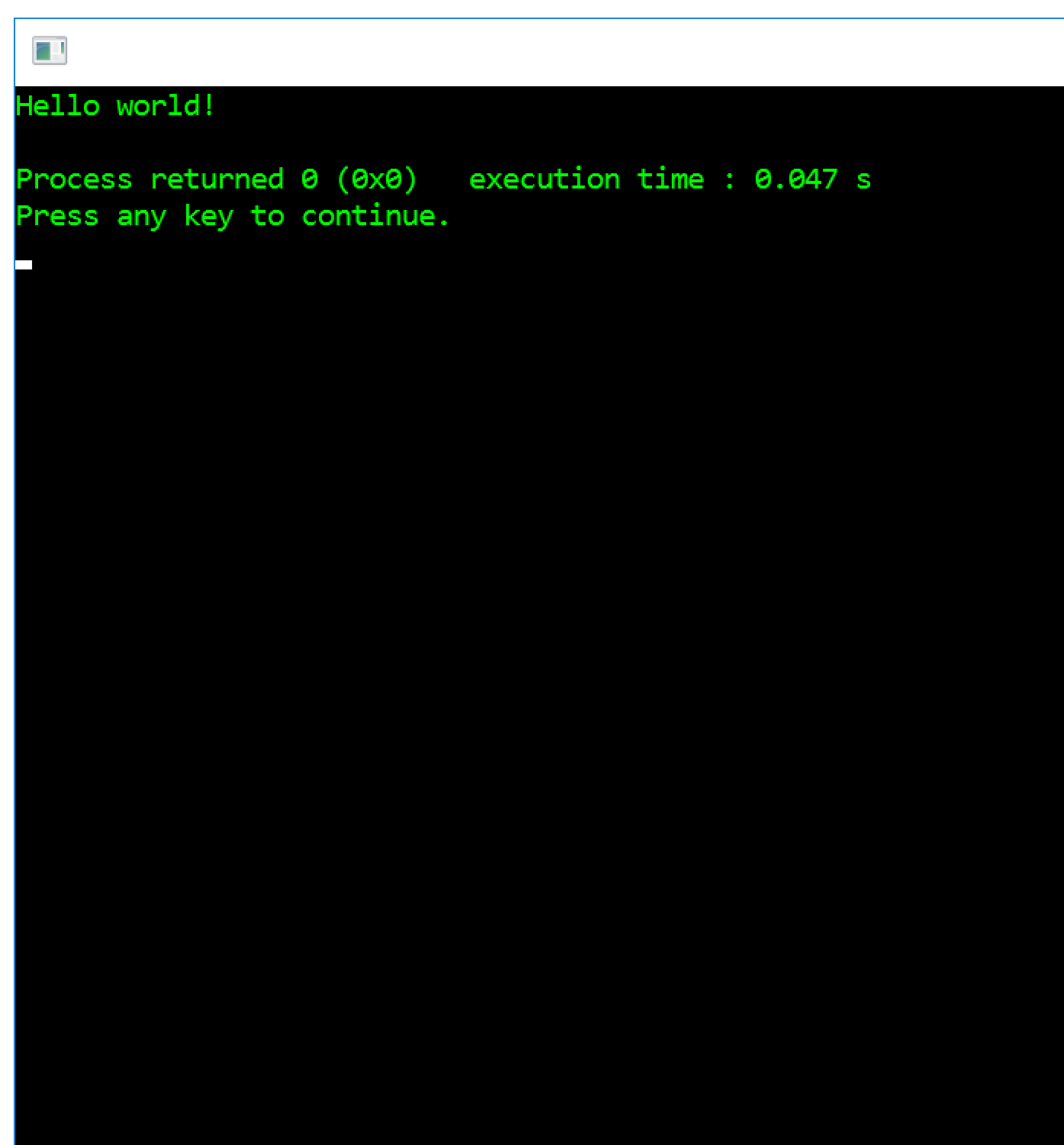

main.cpp [df] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Debug
Management
Projects Symbols
Workspace
df
Sources
main.cpp
main.cpp
#include <iostream>
using namespace std;
int main()
{
cout << "Hello world!" << en
return 0;
}
Logs & others
Code::Blocks
Search results
Cccc
C:\Users\glind\Desktop\df\main.cpp

Pour compiler sans exécuter, appuyez sur , pour exécuter à nouveau sans compilation, appuyez sur et pour compiler puis exécuter, appuyez sur .

, pour exécuter à nouveau sans compilation, appuyez sur et pour compiler puis exécuter, appuyez sur .

, pour exécuter à nouveau sans compilation, appuyez sur et pour compiler puis exécuter, appuyez sur .

Compiler et exécuter la valeur par défaut "Hello world!" le code donne le résultat suivant:



Lire Compiler et construire en ligne: https://riptutorial.com/fr/cplusplus/topic/4708/compiler-et-construire

Chapitre 13: Comportement défini par la mise en œuvre

Examples

Char peut être non signé ou signé

La norme ne spécifie pas si `char` doit être signé ou non signé. Différents compilateurs l'implémentent différemment ou peuvent le modifier à l'aide d'un commutateur de ligne de commande.

Taille des types intégraux

Les types suivants sont définis comme *types intégraux* :

- `char`
- Types d'entiers signés
- Types entiers non signés
- `char16_t` et `char32_t`
- `bool`
- `wchar_t`

À l'exception de `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)` , qui est divisé entre le § 3.9.1.1 [basic.fundamental / 1] et le § 5.3.3.1 [expr.sizeof], et `sizeof(bool)` , qui est entièrement défini par la mise en œuvre et n'a pas de taille minimale, les exigences de taille minimale de ces types sont données dans la section § 3.9.1 [élément fondamental] de la norme et doivent être détaillées ci-dessous.

Taille de `char`

Toutes les versions du C de la norme spécifient, au § 5.3.3.1, que `sizeof` rendements `1` pour `unsigned char` , `signed char` , et l' `char` (il est défini par l' implémentation si le `char` type est `signed` ou `unsigned`).

C ++ 14

`char` est assez grand pour représenter 256 valeurs différentes, pour pouvoir stocker des unités de code UTF-8.

Taille des types d'entiers signés et non signés

La norme spécifie, au § 3.9.1.2, que dans la liste des *types entiers signés standard* , consistant en un caractère `signed char` , `short int` , un `int` , un `long int` et un `long long int` , chaque type fournira au moins autant de stockage que ceux précédents dans la liste. De plus, comme spécifié au § 3.9.1.3, chacun de ces types a un *type d'entier non signé standard* correspondant, *un caractère non* `unsigned char` , un `unsigned short int` , un `unsigned int` , un `unsigned long int` et `unsigned long long int` son type signé correspondant. En outre, comme spécifié au § 3.9.1.1, `char` a la même taille et les exigences d' alignement à la fois comme `signed char` et `unsigned char` .

C ++ 11

Avant C ++ 11, `long long` et `unsigned long long` ne faisaient pas officiellement partie du standard C ++. Cependant, après leur introduction à C, en C99, de nombreux compilateurs prenaient en charge `long long` *entier signé étendu et ne signaient* `unsigned long long` tant que *type entier non signé étendu* , avec les mêmes règles que les types C.

La norme garantit ainsi que:

```
1 == sizeof(char)  == sizeof(signed char) == sizeof(unsigned char)
  <= sizeof(short) == sizeof(unsigned short)
  <= sizeof(int)   == sizeof(unsigned int)
  <= sizeof(long)  == sizeof(unsigned long)
```

C ++ 11

```
 <= sizeof(long long) == sizeof(unsigned long long)
```

Les tailles minimales spécifiques pour chaque type ne sont pas données par la norme. Au lieu de cela, chaque type a une plage minimale de valeurs qu'il peut prendre en charge, à savoir, comme spécifié au § 3.9.1.3, hérité du standard C, au § 5.2.4.2.1. La taille minimale de chaque type peut être approximativement déduite de cette plage, en déterminant le nombre minimal de bits requis; notez que pour toute plate-forme donnée, la plage prise en charge réelle de tout type peut être supérieure au minimum. Notez que pour les types signés, les plages correspondent à son complément, pas au complément à deux plus communément utilisé; Cela permet à un plus large éventail de plates-formes de se conformer à la norme.

Type	Portée minimale	Bits minimum requis
`signed char`	-127 à 127 (- (2^{7} - 1) à (2^{7} - 1))	8
`unsigned char`	0 à 255 (0 à 2^{8} - 1)	8
`signed short`	-32767 à 32767 (- (février 15 à 1) à (15 à 1 février))	16
`unsigned short`	0 à 65 535 (0 à 2^{16} - 1)	16
`signed int`	-32 767 à 32 767 (- (2^{15} - 1) à (2^{15} - 1))	16
`unsigned int`	0 à 65 535 (0 à 2^{16} - 1)	16

Type	Portée minimale	Bits minimum requis
`signed long`	-2 147 483 647 à 2 147 483 647 (- (2^{31} - 1) à (2^{31} - 1))	32
`unsigned long`	0 à 4 294 967 295 (0 à 2^{32} - 1)	32

C ++ 11

Type	Portée minimale	Bits minimum requis
`signed long long`	-9 223 372 036 854 775 807 à 9 223 372 036 854 775 807 (- (2^{63} - 1) à (2^{63} - 1))	64
`unsigned long long`	0 à 18,446,744,073,709,551,615 (0 à 2^{64-1})	64

Chaque type pouvant être supérieur à la taille minimale requise, les types peuvent varier en taille entre les implémentations. L'exemple le plus notable de c'est avec les modèles de données 64 bits LP64 et LLP64, où les systèmes de LLP64 (tels que Windows 64 bits) ont 32 bits `ints` et `long` s et des systèmes LP64 (tels que Linux 64 bits) ont 32 bits `int` et 64 bits `long` s. De ce fait, les types d'entiers ne peuvent pas être supposés avoir une largeur fixe sur toutes les plates-formes.

C ++ 11

Si les entiers avec largeur fixe sont nécessaires, les types d'utilisation de la `<cstdint>` en- tête, mais notez que la norme rend facultative pour les implémentations de soutenir les types exacts de largeur `int8_t` , `int16_t` , `int32_t` , `int64_t` , `intptr_t` , `uint8_t` , `uint16_t` , `uint32_t` , `uint64_t` et `uintptr_t` .

C ++ 11

Taille de `char16_t` et `char32_t`

Les tailles de `char16_t` et `char32_t` sont définies par l'implémentation, comme spécifié au § 5.3.3.1, avec les stipulations du § 3.9.1.5:

- `char16_t` est assez grand pour représenter n'importe quelle unité de code UTF-16 et a la même taille, la même signature et le même alignement que `uint_least16_t` ; il faut donc au moins 16 bits.
- `char32_t` est assez grand pour représenter n'importe quelle unité de code UTF-32 et a la même taille, la même signature et le même alignement que `uint_least32_t` ; il faut donc au moins 32 bits.

Taille de `bool`

La taille de `bool` est définie par l'implémentation et peut ou non être `1` .

Taille de wchar_t

`wchar_t` , comme spécifié au § 3.9.1.5, est un type distinct, dont la plage de valeurs peut représenter chaque unité de code distincte du plus grand jeu de caractères étendu parmi les paramètres régionaux pris en charge. Il a la même taille, la même signature et le même alignement que l'un des autres types intégraux, connu sous le nom de son *type sous-jacent* . La taille de ce type est définie par la mise en œuvre, comme spécifié au § 5.3.3.1, et peut être, par exemple, d'au moins 8, 16 ou 32 bits; Si un système prend en charge Unicode, par exemple, `wchar_t` doit comporter au moins 32 bits (une exception à cette règle est Windows, où `wchar_t` correspond à 16 bits à des fins de compatibilité). Il est hérité de la norme C90, ISO 9899: 1990 § 4.1.5, avec seulement une reformulation mineure.

Selon l'implémentation, la taille de `wchar_t` est souvent, mais pas toujours, 8, 16 ou 32 bits. Les exemples les plus courants sont les suivants:

- Dans les systèmes Unix et Unix, `wchar_t` est 32 bits et est généralement utilisé pour UTF-32.
- Dans Windows, `wchar_t` est 16 bits et est utilisé pour UTF-16.
- Sur un système qui ne prend en charge que 8 bits, `wchar_t` 8 bits.

C ++ 11

Si le support Unicode est souhaité, il est recommandé d'utiliser `char` pour UTF-8, `char16_t` pour UTF-16 ou `char32_t` pour UTF-32, au lieu d'utiliser `wchar_t` .

Modèles de données

Comme mentionné ci-dessus, les largeurs des types entiers peuvent différer entre les plates-formes. Les modèles les plus courants sont les suivants, avec des tailles spécifiées en bits:

Modèle	`int`	`long`	aiguille
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

Sur ces modèles:

- Windows 16 bits utilisé LP32.
- Les systèmes nix 32 bits * (Unix, Linux, Mac OSX et autres systèmes d'exploitation de type

Unix) et Windows utilisent ILP32.
- Windows 64 bits utilise LLP64.
- Les systèmes 64 bits * nix utilisent LP64.

Notez cependant que ces modèles ne sont pas spécifiquement mentionnés dans la norme elle-même.

Nombre de bits dans un octet

En C ++, un *octet* est l'espace occupé par un objet `char` . Le nombre de bits dans un octet est donné par `CHAR_BIT` , qui est défini dans les `climits` et doit être d'au moins 8. Alors que la plupart des systèmes modernes ont des octets de 8 bits, et POSIX exige que `CHAR_BIT` soit exactement 8, il existe certains systèmes où `CHAR_BIT` est supérieur à 8, c'est-à-dire qu'un seul octet peut être composé de 8, 16, 32 ou 64 bits.

Valeur numérique d'un pointeur

Le résultat du lancement d'un pointeur sur un entier à l'aide de `reinterpret_cast` est défini par l'implémentation, mais "... est destiné à être sans surprise pour ceux qui connaissent la structure d'adressage de la machine sous-jacente."

```
int x = 42;
int* p = &x;
long addr = reinterpret_cast<long>(p);
std::cout << addr << "\n"; // prints some numeric address,
                           // probably in the architecture's native address format
```

De même, le pointeur obtenu par conversion à partir d'un entier est également défini par l'implémentation.

La manière correcte de stocker un pointeur en tant `uintptr_t` consiste à utiliser les types `uintptr_t` ou `intptr_t` :

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type
#include <stdint.h>

uintptr_t uip;
```

C ++ 11

```
// There is an optional `std::uintptr_t` in C++11
#include <cstdint>

std::uintptr_t uip;
```

C ++ 11 fait référence à C99 pour la définition de `uintptr_t` (norme C99, 6.3.2.3):

> un type d'entier non signé avec la propriété que tout pointeur valide à `void` peut être converti en ce type, puis reconverti en pointeur sur `void` , et le résultat sera égal au pointeur d'origine.

Alors que, pour la majorité des plates - formes modernes, vous pouvez supposer un espace d'adressage plat et que l' arithmétique sur `uintptr_t` équivaut à l' arithmétique sur `char *` , il est tout à fait possible pour une mise en œuvre pour réaliser une transformation lors de la coulée `void *` à `uintptr_t` tant la transformation peut être inversé lors du retour de `uintptr_t` pour `void *` .

Techniques

- Sur les systèmes `intptr_t` XSI (X / Open System Interfaces), les types `intptr_t` et `uintptr_t` sont obligatoires, sinon ils sont **facultatifs** .

- Au sens de la norme C, les fonctions ne sont pas des objets; le standard C ne garantit pas que `uintptr_t` puisse contenir un pointeur de fonction. Quoi qu'il en soit, la conformité POSIX (2.12.3) nécessite que:

  > Tous les types de pointeurs de fonction doivent avoir la même représentation que le pointeur de type à annuler. La conversion d'un pointeur de fonction en void * ne modifie pas la représentation. Une valeur vide * résultant d'une telle conversion peut être reconvertie dans le type de pointeur de fonction d'origine, en utilisant une conversion explicite, sans perte d'informations.

- C99 §7.18.1:

  > Lorsque des noms typedef ne différant que par l'absence ou la présence de l'u initial sont définis, ils doivent indiquer les types signés et non signés correspondants, comme décrit au 6.2.5; une implémentation fournissant l'un de ces types correspondants fournira également l'autre.

  `uintptr_t` peut avoir un sens si vous voulez faire des choses sur les bits du pointeur que vous ne pouvez pas faire aussi intelligemment avec un entier signé.

Plages de types numériques

Les plages des types entiers sont définies par l'implémentation. L'en-tête `<limits>` fournit le modèle `std::numeric_limits<T>` qui fournit les valeurs minimales et maximales de tous les types fondamentaux. Les valeurs satisfont aux garanties fournies par le standard C via les en- `<climits>` et (> = C ++ 11) `<cinttypes>` .

- `std::numeric_limits<signed char>::min()` est égal à `SCHAR_MIN` , qui est inférieur ou égal à -127.
- `std::numeric_limits<signed char>::max()` est égal à `SCHAR_MAX` , qui est supérieur ou égal à 127.
- `std::numeric_limits<unsigned char>::max()` est égal à `UCHAR_MAX` , qui est supérieur ou égal à 255.
- `std::numeric_limits<short>::min()` est égal à `SHRT_MIN` , qui est inférieur ou égal à -32767.
- `std::numeric_limits<short>::max()` est égal à `SHRT_MAX` , qui est supérieur ou égal à 32767.
- `std::numeric_limits<unsigned short>::max()` est égal à `USHRT_MAX` , qui est supérieur ou égal à 65535.
- `std::numeric_limits<int>::min()` est égal à `INT_MIN` , qui est inférieur ou égal à -32767.
- `std::numeric_limits<int>::max()`

est égal à `INT_MAX` , qui est supérieur ou égal à 32767.
- `std::numeric_limits<unsigned int>::max()` est égal à `UINT_MAX` , qui est supérieur ou égal à 65535.
- `std::numeric_limits<long>::min()` est égal à `LONG_MIN` , qui est inférieur ou égal à -2147483647.
- `std::numeric_limits<long>::max()` est égal à `LONG_MAX` , qui est supérieur ou égal à 2147483647.
- `std::numeric_limits<unsigned long>::max()` est égal à `ULONG_MAX` , qui est supérieur ou égal à 4294967295.

C ++ 11

- `std::numeric_limits<long long>::min()` est égal à `LLONG_MIN` , qui est inférieur ou égal à -9223372036854775807.
- `std::numeric_limits<long long>::max()` est égal à `LLONG_MAX` , qui est supérieur ou égal à 9223372036854775807.
- `std::numeric_limits<unsigned long long>::max()` est égal à `ULLONG_MAX` , qui est supérieur ou égal à 18446744073709551615.

Pour les types à virgule flottante `T` , `max()` est la valeur finie maximale tandis que `min()` est la valeur normalisée positive minimale. Des membres supplémentaires sont fournis pour les types à virgule flottante, qui sont également définis par l'implémentation mais satisfont à certaines garanties fournies par le standard C via l'en-tête `<cfloat>` .

- Le membre `digits10` donne le nombre de chiffres décimaux de précision.
  - `std::numeric_limits<float>::digits10` est égal à `FLT_DIG` , qui est au moins 6.
  - `std::numeric_limits<double>::digits10` est égal à `DBL_DIG` , soit au moins 10.
  - `std::numeric_limits<long double>::digits10` est égal à `LDBL_DIG` , soit au moins 10.
- Le membre `min_exponent10` est le minimum négatif E tel que 10 à la puissance E est normal.
  - `std::numeric_limits<float>::min_exponent10` est égal à `FLT_MIN_10_EXP` , soit au maximum -37.
  - `std::numeric_limits<double>::min_exponent10` est égal à `DBL_MIN_10_EXP` , soit au maximum -37. `std::numeric_limits<long double>::min_exponent10` est égal à `LDBL_MIN_10_EXP` , soit au maximum -37.
- Le membre `max_exponent10` est le maximum E tel que 10 à la puissance E est fini.
  - `std::numeric_limits<float>::max_exponent10` est égal à `FLT_MIN_10_EXP` , soit au moins 37.
  - `std::numeric_limits<double>::max_exponent10` est égal à `DBL_MIN_10_EXP` , soit au moins 37.
  - `std::numeric_limits<long double>::max_exponent10` est égal à `LDBL_MIN_10_EXP` , soit au moins 37.
- Si le membre `is_iec559` est vrai, le type est conforme à la norme IEC 559 / IEEE 754 et sa plage est donc déterminée par cette norme.

Représentation de la valeur des types à virgule flottante

Le standard exige que le `long double` fournisse au moins autant de précision que le `double` , ce qui fournit au moins autant de précision que le `float` ; et qu'un `long double` peut représenter n'importe

quelle valeur qu'un `double` peut représenter, tandis qu'un `double` peut représenter n'importe quelle valeur qu'un `float` peut représenter. Les détails de la représentation sont cependant définis par la mise en œuvre.

Pour un type à virgule flottante `T` , `std::numeric_limits<T>::radix` spécifie la base utilisée par la représentation de `T`

Si `std::numeric_limits<T>::is_iec559` est vrai, alors la représentation de `T` correspond à l'un des formats définis par la norme IEC 559 / IEEE 754.

Débordement lors de la conversion d'un entier en entier signé

Lorsqu'un entier signé ou non signé est converti en un type entier signé et que sa valeur n'est pas représentable dans le type de destination, la valeur produite est définie par l'implémentation. Exemple:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and
// the range of unsigned char is 0 to 255
int x = 12345;
signed char sc = x;   // sc has an implementation-defined value
unsigned char uc = x; // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Type sous-jacent (et donc taille) d'un enum

Si le type sous-jacent n'est pas explicitement spécifié pour un type d'énumération non segmenté, il est déterminé d'une manière définie par l'implémentation.

```
enum E {
    RED,
    GREEN,
    BLUE,
};
using T = std::underlying_type<E>::type; // implementation-defined
```

Toutefois, la norme exige que le type sous-jacent d’une énumération ne soit pas supérieur à `int` moins que `int` et `unsigned int` ne puissent pas représenter toutes les valeurs de l’énumération. Par conséquent, dans le code ci-dessus, `T` pourrait être `int` , `unsigned int` ou `short` , mais pas `long long` , pour donner quelques exemples.

Notez qu'une énumération a la même taille (renvoyée par `sizeof`) que son type sous-jacent.

Lire Comportement défini par la mise en œuvre en ligne:
https://riptutorial.com/fr/cplusplus/topic/1363/comportement-defini-par-la-mise-en-ouvre

Chapitre 14: Comportement non défini

Introduction

Qu'est-ce qu'un comportement indéfini (UB)? Selon la norme ISO C ++ (§1.3.24, N4296), c'est "le comportement pour lequel la présente Norme internationale n'impose aucune exigence".

Cela signifie que lorsqu'un programme rencontre UB, il est autorisé à faire ce qu'il veut. Cela signifie souvent un accident, mais il peut simplement ne rien faire , faire en sorte que les démons ne vous touchent pas ou même *sembler* fonctionner correctement!

Inutile de dire que vous devriez éviter d'écrire du code qui appelle UB.

Remarques

Si un programme contient un comportement indéfini, le standard C ++ ne place aucune contrainte sur son comportement.

- Il peut sembler fonctionner comme prévu par le développeur, mais il peut également tomber en panne ou produire des résultats étranges.
- Le comportement peut varier entre les exécutions du même programme.
- Toute partie du programme peut ne pas fonctionner correctement, y compris les lignes précédant la ligne contenant un comportement indéfini.
- L'implémentation n'est pas requise pour documenter le résultat d'un comportement non défini.

Une implémentation *peut* documenter le résultat d'une opération qui produit un comportement indéfini conformément à la norme, mais un programme qui dépend de ce comportement documenté n'est pas portable.

Pourquoi un comportement indéfini existe

Intuitivement, le comportement non défini est considéré comme une mauvaise chose, car de telles erreurs ne peuvent être traitées gracieusement, par exemple, par des gestionnaires d'exceptions.

Mais laisser un comportement indéfini fait en réalité partie intégrante de la promesse de C ++: "vous ne payez pas pour ce que vous n'utilisez pas". Un comportement indéfini permet à un compilateur de supposer que le développeur sait ce qu'il fait et n'introduit pas de code pour vérifier les erreurs mises en évidence dans les exemples ci-dessus.

Trouver et éviter un comportement indéfini

Certains outils peuvent être utilisés pour découvrir un comportement indéfini pendant le développement:

- La plupart des compilateurs ont des drapeaux d'avertissement pour avertir de certains cas de comportement indéfini au moment de la compilation.

- Les versions plus récentes de gcc et de clang incluent un indicateur appelé "Désinfectant de comportement `-fsanitize=undefined` " (`-fsanitize=undefined`) qui vérifie le comportement indéfini à l'exécution, à un coût de performance.
- `lint` outils ressemblant à des `lint` peuvent effectuer une analyse comportementale non définie plus approfondie.

Comportement non défini , non spécifié et défini par l'implémentation

De la norme C ++ 14 (ISO / IEC 14882: 2014), section 1.9 (Exécution du programme):

1. Les descriptions sémantiques de la présente Norme internationale définissent une machine abstraite non déterministe paramétrée. [COUPER]

2. Certains aspects et opérations de la machine abstraite sont décrits dans la présente Norme internationale comme **définis** par la **mise** en **œuvre** (par exemple, `sizeof(int)`). Celles-ci constituent *les paramètres de la machine abstraite* . Chaque mise en œuvre doit inclure une documentation décrivant ses caractéristiques et son comportement à ces égards. [COUPER]

3. Certains autres aspects et opérations de la machine abstraite sont décrits dans la présente Norme internationale comme **non spécifiés** (par exemple, évaluation des expressions dans un *nouvel initialiseur* si la fonction d'allocation ne parvient pas à allouer de la mémoire). Dans la mesure du possible, la présente Norme internationale définit un ensemble de comportements admissibles. Celles-ci définissent les aspects non déterministes de la machine abstraite. Une instance de la machine abstraite peut donc avoir plus d'une exécution possible pour un programme donné et une entrée donnée.

4. Certaines autres opérations sont décrites dans la présente Norme internationale comme **non définies** (ou exemple, l'effet de la tentative de modification d'un objet `const`). [*Note* : la présente Norme internationale n'impose aucune exigence concernant le comportement des programmes qui contiennent un comportement indéfini. - *note finale*]

Examples

Lecture ou écriture à travers un pointeur nul

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

Il s'agit d' **un comportement indéfini** , car un pointeur null ne pointe sur aucun objet valide, il n'y a donc aucun objet à écrire dans `*ptr` .

Bien que cela provoque le plus souvent une erreur de segmentation, elle est indéfinie et tout peut arriver.

Aucune déclaration de retour pour une fonction avec un type de retour non vide

L'omission de l'instruction `return` dans une fonction dont le type de retour est non `void` est **un comportement indéfini** .

```
int function() {
    // Missing return statement
}

int main() {
    function(); //Undefined Behavior
}
```

La plupart des compilateurs modernes émettent un avertissement au moment de la compilation pour ce type de comportement indéfini.

Note: `main` est la seule exception à la règle. Si `main` n'a pas d'instruction de `return` , le compilateur insère automatiquement `return 0;` pour vous, il peut donc être laissé de côté.

Modifier un littéral de chaîne

C ++ 11

```
char *str = "hello world";
str[0] = 'H';
```

`"hello world"` est un littéral de chaîne, donc le modifier donne un comportement indéfini.

L'initialisation de `str` dans l'exemple ci-dessus était officiellement obsolète (il est prévu de la supprimer d'une future version du standard) en C ++ 03. Un certain nombre de compilateurs avant 2003 pourraient émettre un avertissement à ce sujet (par exemple, une conversion suspecte). Après 2003, les compilateurs avertissent généralement d'une conversion obsolète.

C ++ 11

L'exemple ci-dessus est illégal et entraîne un diagnostic du compilateur, en C ++ 11 et versions ultérieures. Un exemple similaire peut être construit pour présenter un comportement indéfini en permettant explicitement la conversion de type, telle que:

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

Accéder à un index hors limites

C'est **un comportement indéfini** pour accéder à un index qui est hors limites pour un tableau (ou un conteneur de bibliothèque standard, car ils sont tous implémentés en utilisant un tableau *brut*):

```
int array[] = {1, 2, 3, 4, 5};
array[5] = 0;  // Undefined behavior
```

Il est *permis* d'avoir un pointeur pointant vers la fin du tableau (dans ce cas, `array + 5`), vous ne pouvez pas le déréférencer, car ce n'est pas un élément valide.

```
const int *end = array + 5;  // Pointer to one past the last index
for (int *p = array; p != end; ++p)
  // Do something with `p`
```

En général, vous n'êtes pas autorisé à créer un pointeur hors limites. Un pointeur doit pointer sur un élément du tableau ou sur un élément passé.

Division entière par zéro

```
int x = 5 / 0;    // Undefined behavior
```

La division par `0` est mathématiquement indéfinie et, en tant que telle, il est logique qu'il s'agisse d'un comportement indéfini.

Toutefois:

```
float x = 5.0f / 0.0f;   // x is +infinity
```

La plupart des implémentations implémentent IEEE-754, qui définit la division en virgule flottante par zéro pour renvoyer `NaN` (si le numérateur est `0.0f`), l' `infinity` (si le numérateur est positif) ou `-infinity` (si le numérateur est négatif).

Dépassement d'entier signé

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

> Si, lors de l'évaluation d'une expression, le résultat n'est pas défini mathématiquement ou ne figure pas dans la plage des valeurs représentables pour son type, le comportement n'est pas défini.

(Norme C ++ 11, paragraphe 5/4)

C’est l’un des plus désagréables, car il produit généralement un comportement reproductible et non écrasant, de sorte que les développeurs peuvent être tentés de s’appuyer fortement sur le comportement observé.

D'autre part:

```
unsigned int x = UINT_MAX + 1;
```

```
// x is 0
```

est bien défini depuis:

> Les entiers non signés, déclarés non signés, doivent obéir aux lois de l'arithmétique modulo `2^n` où `n` est le nombre de bits dans la représentation des valeurs de la taille entière entière.

(Norme C ++ 11, paragraphe 3.9.1 / 4)

Parfois, les compilateurs peuvent exploiter un comportement indéfini et optimiser

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Dans la mesure où un dépassement d'entier signé n'est pas défini, le compilateur est libre de supposer qu'il peut ne jamais se produire et peut donc optimiser le bloc "if"

Utilisation d'une variable locale non initialisée

```
int a;
std::cout << a; // Undefined behavior!
```

Cela se traduit par **un comportement indéfini** , car `a` est non initialisé.

Il est souvent, à tort, prétendu que c'est parce que la valeur est "indéterminée", ou "quelle que soit la valeur présente dans cet emplacement de mémoire avant". Cependant, c'est l'acte d'accéder à la valeur d' `a` dans l'exemple ci-dessus qui donne un comportement indéfini. En pratique, imprimer un "garbage value" est un symptôme courant dans ce cas, mais ce n’est qu’une forme possible de comportement non défini.

Bien que très improbable dans la pratique (puisqu'il dépend d'un support matériel spécifique), le compilateur pourrait également électrocuter le programmeur lors de la compilation de l'exemple de code ci-dessus. Avec un tel support de compilateur et de matériel, une telle réponse à un comportement indéfini augmenterait nettement la compréhension moyenne (vivante) du programmeur de la véritable signification du comportement indéfini - à savoir que la norme n'impose aucune contrainte au comportement résultant.

C ++ 14

L'utilisation d'une valeur indéterminée de type `unsigned char` ne produit pas un comportement indéfini si la valeur est utilisée comme:

- le deuxième ou le troisième opérande de l'opérateur conditionnel ternaire;
- le bon opérande de l'opérateur de virgule intégré;
- l'opérande d'une conversion en caractère `unsigned char` ;

- l'opérande droit de l'opérateur d'affectation, si l'opérande gauche est également du type `unsigned char` ;
- l'initialiseur d'un objet `unsigned char` ;

ou si la valeur est ignorée. Dans de tels cas, la valeur indéterminée se propage simplement au résultat de l'expression, le cas échéant.

Notez qu'une variable `static` est **toujours** initialisée à zéro (si possible):

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

Plusieurs définitions non identiques (la règle de définition unique)

Si une classe, une énumération, une fonction en ligne, un modèle ou un membre d'un modèle possède un lien externe et est défini dans plusieurs unités de traduction, toutes les définitions doivent être identiques ou non définies conformément à la règle ODR .

`foo.h` :

```
class Foo {
  public:
    double x;
  private:
    int y;
};

Foo get_foo();
```

`foo.cpp` :

```
#include "foo.h"
Foo get_foo() { /* implementation */ }
```

`main.cpp` :

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
  public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

Le programme ci-dessus présente un comportement indéfini car il contient deux définitions de la classe `::Foo` , qui a un lien externe, dans différentes unités de traduction, mais les deux définitions ne sont pas identiques. Contrairement à la redéfinition d'une classe dans la *même* unité de traduction, le compilateur n'a pas besoin de diagnostiquer ce problème.

Appariement incorrect de l'allocation de mémoire et de la désallocation

Un objet ne peut être désalloué que par `delete` s'il a été alloué par `new` et n'est pas un tableau. Si l'argument à `delete` n'a pas été renvoyé par `new` ou s'il s'agit d'un tableau, le comportement n'est pas défini.

Un objet ne peut être désalloué que par `delete[]` s'il a été alloué par `new` et est un tableau. Si l'argument de `delete[]` n'a pas été renvoyé par `new` ou n'est pas un tableau, le comportement est indéfini.

Si l'argument à `free` n'a pas été renvoyé par `malloc` , le comportement est indéfini.

```
int* p1 = new int;
delete p1;      // correct
// delete[] p1; // undefined
// free(p1);    // undefined

int* p2 = new int[10];
delete[] p2;    // correct
// delete p2;   // undefined
// free(p2);    // undefined

int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3);       // correct
// delete p3;   // undefined
// delete[] p3; // undefined
```

Ces problèmes peuvent être évités en évitant complètement les programmes `malloc` et `free` dans C ++, préférant les pointeurs intelligents de la bibliothèque standard sur raw `new` et `delete` , et préférant `std::vector` et `std::string` sur raw `new` et `delete[]` .

Accéder à un objet avec le mauvais type

Dans la plupart des cas, il est illégal d'accéder à un objet d'un type comme s'il s'agissait d'un type différent (sans tenir compte des qualificatifs cv). Exemple:

```
float x = 42;
int y = reinterpret_cast<int&>(x);
```

Le résultat est un comportement indéfini.

Il existe quelques exceptions à cette règle d' *aliasing stricte* :

- Un objet de type classe est accessible comme s'il s'agissait d'un type qui est une classe de base du type de classe réel.
- Vous pouvez accéder à n'importe quel type en tant que caractère `char` ou `unsigned char` , mais l'inverse n'est pas vrai: il est impossible d'accéder à un tableau de caractères comme s'il s'agissait d'un type arbitraire.
- Un type entier signé peut être accédé en tant que type non signé correspondant et *vice versa* .

Une règle connexe est que si une fonction membre non statique est appelée sur un objet qui n'a pas réellement le même type que la classe de définition de la fonction ou une classe dérivée, le comportement non défini se produit. Cela est vrai même si la fonction n'accède pas à l'objet.

```
struct Base {
};
struct Derived : Base {
    void f() {}
};
struct Unrelated {};
Unrelated u;
Derived& r1 = reinterpret_cast<Derived&>(u); // ok
r1.f();                                      // UB
Base b;
Derived& r2 = reinterpret_cast<Derived&>(b); // ok
r2.f();                                      // UB
```

Débordement en virgule flottante

Si une opération arithmétique qui produit un type à virgule flottante produit une valeur qui ne se situe pas dans la plage des valeurs représentables du type de résultat, le comportement n'est pas défini conformément à la norme C ++, mais peut être défini par d'autres normes comme IEEE 754.

```
float x = 1.0;
for (int i = 0; i < 10000; i++) {
    x *= 10.0; // will probably overflow eventually; undefined behavior
}
```

Appel de membres virtuels (purs) à partir d'un constructeur ou d'un destructeur

La norme (10.4) stipule:

> Les fonctions membres peuvent être appelées à partir d'un constructeur (ou d'un destructeur) d'une classe abstraite; L'effet de faire un appel virtuel (10.3) à une fonction virtuelle pure directement ou indirectement pour l'objet créé (ou détruit) à partir d'un tel constructeur (ou destructeur) n'est pas défini.

Plus généralement, certaines autorités C ++, par exemple Scott Meyers, suggèrent de ne jamais appeler de fonctions virtuelles (même non pures) à partir de constructeurs et de constructeurs.

Prenons l'exemple suivant, modifié à partir du lien ci-dessus:

```
class transaction
{
public:
    transaction(){ log_it(); }
    virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
```

```
public:
    virtual void log_it() const { /* Do something */ }
};
```

Supposons que nous créons un objet `sell_transaction` :

```
sell_transaction s;
```

Cela appelle implicitement le constructeur de `sell_transaction` , qui appelle d'abord le constructeur de `transaction` . Cependant, lorsque le constructeur de `transaction` est appelé, l'objet n'est pas encore du type `sell_transaction` , mais uniquement de la `transaction` type.

Par conséquent, l'appel de `transaction::transaction()` à `log_it` ne fera pas ce qui peut sembler être intuitif, à savoir appeler `sell_transaction::log_it` .

- Si `log_it` est purement virtuel, comme dans cet exemple, le comportement est indéfini.
- Si `log_it` est virtuel non pur, `transaction::log_it` sera appelée.

Suppression d'un objet dérivé via un pointeur sur une classe de base sans destructeur virtuel.

```
class base { };
class derived: public base { };

int main() {
    base* p = new derived();
    delete p; // The is undefined behavior!
}
```

Dans la section [expr.delete] §5.3.5 / 3, la norme indique que si la `delete` est appelée sur un objet dont le type statique ne comporte pas `virtual` destructeur `virtual` :

> si le type statique de l'objet à supprimer est différent de son type dynamique, le type statique doit être une classe de base du type dynamique de l'objet à supprimer et le type statique doit avoir un destructeur virtuel ou le comportement n'est pas défini.

C'est le cas quelle que soit la question de savoir si la classe dérivée a ajouté des membres de données à la classe de base.

Accéder à une référence en suspens

Il est illégal d'accéder à une référence à un objet hors de portée ou détruit d'une autre manière. Une telle référence est dite *pendante* car elle ne fait plus référence à un objet valide.

```
#include <iostream>
int& getX() {
    int x = 42;
    return x;
}
```

```
int main() {
    int& r = getX();
    std::cout << r << "\n";
}
```

Dans cet exemple, la variable locale `x` sort de la portée lorsque `getX` retourne. (Notez que l' *extension de durée de vie* ne peut pas prolonger la durée de vie d'une variable locale au-delà de la portée du bloc dans lequel elle est définie). Par conséquent, `r` est une référence en suspens. Ce programme a un comportement indéfini, bien qu'il puisse sembler fonctionner et imprimer `42` dans certains cas.

Extension de l'espace de noms `std` ou` posix`

La norme (17.6.4.2.1 / 1) interdit généralement l'extension de l'espace de noms `std` :

> Le comportement d'un programme C ++ n'est pas défini s'il ajoute des déclarations ou des définitions à l'espace de noms std ou à un espace de noms dans un espace de noms std, sauf indication contraire.

Il en va de même pour `posix` (17.6.4.2.2 / 1):

> Le comportement d'un programme C ++ n'est pas défini s'il ajoute des déclarations ou des définitions à posix d'espace de noms ou à un espace de noms dans posix d'espace de noms, sauf indication contraire.

Considérer ce qui suit:

```
#include <algorithm>

namespace std
{
    int foo(){}
}
```

Rien dans la norme n'interdit à l' `algorithm` (ou à l'un des en-têtes qu'il inclut) de définir la même définition, et ce code violerait donc la règle de définition unique .

Donc, en général, ceci est interdit. Il existe toutefois des exceptions spécifiques . Peut-être plus utile, il est permis d'ajouter des spécialisations pour les types définis par l'utilisateur. Par exemple, supposons que votre code a

```
class foo
{
    // Stuff
};
```

Alors ce qui suit va bien

```
namespace std
{
    template<>
```

```
    struct hash<foo>
    {
    public:
        size_t operator()(const foo &f) const;
    };
}
```

Débordement lors de la conversion vers ou à partir du type à virgule flottante

Si, lors de la conversion de:

- un type entier à un type à virgule flottante,
- un type à virgule flottante pour un type entier, ou
- un type à virgule flottante pour un type à virgule flottante plus court,

la valeur source est en dehors de la plage de valeurs pouvant être représentée dans le type de destination, le résultat est un comportement non défini. Exemple:

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Jet statique de base à dérivé invalide

Si `static_cast` est utilisé pour convertir un pointeur (resp. Référence) en une classe de base en un pointeur (resp. Référence) en une classe dérivée, mais l'opérande ne pointe pas (resp. Se référer) à un objet du type de classe dérivé, le comportement est indéfini Voir Base à la conversion dérivée .

Appel de fonction via le type de pointeur de fonction incompatible

Pour appeler une fonction via un pointeur de fonction, le type du pointeur de fonction doit correspondre exactement au type de la fonction. Sinon, le comportement est indéfini. Exemple:

```
int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // undefined
```

Modifier un objet const

Toute tentative de modification d'un objet `const` entraîne un comportement indéfini. Cela s'applique aux variables `const` , aux membres des objets `const` et aux membres de classe déclarés `const` . (Cependant, un membre `mutable` d'un objet `const` n'est pas `const` .)

Une telle tentative peut être faite via `const_cast` :

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

Un compilateur inclura généralement la valeur d'un objet `const int` , il est donc possible que ce code compile et imprime `123` . Les compilateurs peuvent également placer les valeurs des objets `const` dans une mémoire en lecture seule, ce qui peut entraîner une erreur de segmentation. Dans tous les cas, le comportement n'est pas défini et le programme peut faire n'importe quoi.

Le programme suivant cache une erreur beaucoup plus subtile:

```
#include <iostream>

class Foo* instance;

class Foo {
  public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
  private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}
```

Dans ce code, `getFoo` crée un singleton de type `const Foo` et son membre `m_x` est initialisé à `123` . Ensuite, `do_evil` est appelé et la valeur de `foo.m_x` est apparemment passée à 456. Qu'est-ce qui n'a pas fonctionné?

En dépit de son nom, `do_evil` ne fait rien de particulièrement mal; tout ce qu'il fait est d'appeler un passeur à travers un `Foo*` . Mais ce pointeur pointe vers un objet `const Foo` même si `const_cast` n'a pas été utilisé. Ce pointeur a été obtenu via le constructeur de `Foo` . Un `const` objet ne devient pas `const` jusqu'à ce que son initialisation est terminée, donc `this` a le type `Foo*` , non `const Foo*` , dans le constructeur.

Par conséquent, un comportement indéfini se produit même s'il n'y a pas de constructions manifestement dangereuses dans ce programme.

Accès à un membre inexistant via un pointeur sur un membre

Lorsque vous accédez à un membre non statique d'un objet via un pointeur sur membre, si l'objet

ne contient pas réellement le membre indiqué par le pointeur, le comportement est indéfini. (Un tel pointeur vers membre peut être obtenu via `static_cast` .)

```
struct Base { int x; };
struct Derived : Base { int y; };
int Derived::*pdy = &Derived::y;
int Base::*pby = static_cast<int Base::*>(pdy);

Base* b1 = new Derived;
b1->*pby = 42; // ok; sets y in Derived object to 42
Base* b2 = new Base;
b2->*pby = 42; // undefined; there is no y member in Base
```

Conversion de base en base invalide pour les pointeurs vers les membres

Lorsque `static_cast` est utilisé pour convertir `TD::*` en `TB::*` , le membre désigné doit appartenir à une classe qui est une classe de base ou une classe dérivée de `B` Sinon, le comportement n'est pas défini. Voir Dérivé à la conversion de base pour les pointeurs vers les membres

Arithmétique de pointeur invalide

Les utilisations suivantes de l'arithmétique de pointeur provoquent un comportement indéfini:

- Ajout ou soustraction d'un entier, si le résultat n'appartient pas au même objet tableau que l'opérande du pointeur. (Ici, l'élément un passé est considéré comme appartenant toujours au tableau.)

```
int a[10];
int* p1 = &a[5];
int* p2 = p1 + 4; // ok; p2 points to a[9]
int* p3 = p1 + 5; // ok; p2 points to one past the end of a
int* p4 = p1 + 6; // UB
int* p5 = p1 - 5; // ok; p2 points to a[0]
int* p6 = p1 - 6; // UB
int* p7 = p3 - 5; // ok; p7 points to a[5]
```

- Soustraction de deux pointeurs s'ils n'appartiennent pas tous deux au même objet tableau. (Encore une fois, l'élément un passé après la fin est considéré comme appartenant au tableau.) L'exception est que deux pointeurs nuls peuvent être soustraits, ce qui donne 0.

```
int a[10];
int b[10];
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // yields 5
int *p3 = p1 + 2; // ok; p3 points to one past the end of a
int d2 = p3 - p2; // yields 7
int *p4 = &b[0];
int d3 = p4 - p1; // UB
```

- Soustraction de deux pointeurs si le résultat déborde `std::ptrdiff_t` .

- Toute arithmétique de pointeur où le type de pointe de l'un des opérandes ne correspond

pas au type dynamique de l'objet pointé (en ignorant la qualification cv). Selon la norme, "[en] particulier, un pointeur vers une classe de base ne peut pas être utilisé pour l'arithmétique du pointeur lorsque le tableau contient des objets d'un type de classe dérivé."

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
auto p5 = p4 - p1;          // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter
```

Déplacement par un nombre de postes invalide

Pour l'opérateur de quart intégré, l'opérande droit doit être non négatif et strictement inférieur à la largeur de bit de l'opérande gauche promu. Sinon, le comportement est indéfini.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0;  // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                       // int must be at least 16 bits
```

Retourner d'une fonction [[noreturn]]

C ++ 11

Exemple du standard, [dcl.attr.noreturn]:

```
[[ noreturn ]] void f() {
  throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
}
```

Détruire un objet qui a déjà été détruit

Dans cet exemple, un destructeur est explicitement appelé pour un objet qui sera ultérieurement détruit automatiquement.

```
struct S {
    ~S() { std::cout << "destroying S\n"; }
};
int main() {
```

```
    S s;
    s.~S();
} // UB: s destroyed a second time here
```

Un problème similaire se produit lorsqu'un `std::unique_ptr<T>` est dirigé vers un `T` avec une durée de stockage automatique ou statique.

```
void f(std::unique_ptr<S> p);
int main() {
    S s;
    std::unique_ptr<S> p(&s);
    f(std::move(p)); // s destroyed upon return from f
}                    // UB: s destroyed
```

Une autre façon de détruire un objet à deux reprises consiste à faire en sorte que deux `shared_ptr` gèrent l'objet sans en partager la propriété.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);
int main() {
    S* p = new S;
    // I want to pass the same object twice...
    std::shared_ptr<S> sp1(p);
    std::shared_ptr<S> sp2(p);
    f(sp1, sp2);
} // UB: both sp1 and sp2 will destroy s separately
// NB: this is correct:
// std::shared_ptr<S> sp(p);
// f(sp, sp);
```

Récursion du modèle infini

Exemple de la norme, [temp.inst] / 17:

```
template<class T> class X {
    X<T>* p; // OK
    X<T*> a; // implicit generation of X<T> requires
             // the implicit instantiation of X<T*> which requires
             // the implicit instantiation of X<T**> which ...
};
```

Lire Comportement non défini en ligne:
https://riptutorial.com/fr/cplusplus/topic/1812/comportement-non-defini

Chapitre 15: Comportement non spécifié

Remarques

Si le comportement d'une construction n'est pas spécifié, la norme impose des contraintes au comportement, mais laisse une certaine liberté à l'implémentation, ce qui n'est *pas* nécessaire pour documenter ce qui se passe dans une situation donnée. Cela contraste avec le comportement défini par l' implémentation , dans lequel l'implémentation *est* nécessaire pour documenter ce qui se passe, et un comportement non défini, dans lequel tout peut arriver.

Examples

Ordre d'initialisation des globales à travers TU

Alors qu'au sein d'une unité de traduction, l'ordre d'initialisation des variables globales est spécifié, l'ordre d'initialisation entre les unités de traduction n'est pas spécifié.

Donc programme avec les fichiers suivants

- foo.cpp

  ```
  #include <iostream>

  int dummyFoo = ((std::cout << "foo"), 0);
  ```

- bar.cpp

  ```
  #include <iostream>

  int dummyBar = ((std::cout << "bar"), 0);
  ```

- main.cpp

  ```
  int main() {}
  ```

pourrait produire comme sortie:

```
foobar
```

ou

```
barfoo
```

Cela peut conduire à un *Fiasco d'ordre d'initialisation statique* .

Valeur d'un enum hors gamme

Si un enum de portée est converti en un type intégral trop petit pour contenir sa valeur, la valeur résultante n'est pas spécifiée. Exemple:

```
enum class E {
    X = 1,
    Y = 1000,
};
// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

De plus, si un entier est converti en enum et que la valeur de l'entier est en dehors de la plage des valeurs de l'énumération, la valeur résultante n'est pas spécifiée. Exemple:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};
Color c = static_cast<Color>(4);
```

Cependant, dans l'exemple suivant, le comportement n'est *pas* spécifié, car la valeur source est *comprise* dans la *plage* de l'énumération, bien qu'elle soit inégale pour tous les énumérateurs:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};
Scale s = static_cast<Scale>(3);
```

`s` aura la valeur 3 et sera inégale à `ONE` , `TWO` et `FOUR` .

Cast statique à partir de la valeur nulle et bidon

Si une valeur `void*` est convertie en un pointeur vers le type d'objet, `T*` , mais n'est pas correctement alignée pour `T` , la valeur du pointeur obtenue n'est pas spécifiée. Exemple:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

La valeur de `p3` n'est pas spécifiée car `p2` ne peut pas pointer vers un objet de type `int` ; sa valeur n'est pas une adresse correctement alignée.

Résultat de certaines conversions réinterprétées

Le résultat d'un `reinterpret_cast` d'un type de pointeur de fonction à un autre, ou d'un type de référence de fonction à un autre, n'est pas spécifié. Exemple:

```
int f();
auto fp = reinterpret_cast<int(*)(int)>(&f); // fp has unspecified value
```

C ++ 03

Le résultat d'un `reinterpret_cast` d'un type de pointeur d'objet à un autre, ou d'un type de référence d'objet à un autre, n'est pas spécifié. Exemple:

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

Cependant, avec la plupart des compilateurs, cela équivalait à `static_cast<char*>(static_cast<void*>(&x))` donc le pointeur résultant `p` indiquait le premier octet de `x` . Cela a été fait le comportement standard en C ++ 11. Voir la conversion de types pour plus de détails.

Résultat de certaines comparaisons de pointeurs

Si deux pointeurs sont comparés en utilisant `<` , `>` , `<=` ou `>=` , le résultat n'est pas spécifié dans les cas suivants:

- Les pointeurs pointent vers différents tableaux. (Un objet non-tableau est considéré comme un tableau de taille 1.)

```
int x;
int y;
const bool b1 = &x < &y;            // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];      // true
const bool b3 = &a[0] < &x;         // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                    // note: a+10 points past the end of the array
```

- Les pointeurs pointent vers le même objet, mais vers des membres avec un contrôle d'accès différent.

```
class A {
  public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
    bool f2() { return &x < &z; } // unspecified
  private:
    int z;
};
```

Espace occupé par une référence

Une référence n'est pas un objet et contrairement à un objet, il n'est pas garanti qu'elle occupe certains octets de mémoire contigus. La norme ne précise pas si une référence nécessite un stockage. Un certain nombre de caractéristiques du langage se compliquent pour qu'il soit impossible d'examiner de manière portable tout stockage que la référence pourrait occuper:

- Si `sizeof` est appliqué à une référence, il retourne la taille du type référencé, ne donnant ainsi aucune information sur le fait de savoir si la référence occupe un stockage.
- Les tableaux de références sont illégaux, il n'est donc pas possible d'examiner les adresses de deux éléments consécutifs d'une référence hypothétique de tableaux afin de déterminer la taille d'une référence.
- Si l'adresse d'une référence est prise, le résultat est l'adresse du référent, nous ne pouvons donc pas obtenir un pointeur sur la référence elle-même.
- Si une classe possède un membre de référence, la tentative d'extraction de l'adresse de ce membre à l'aide de `offsetof` un comportement indéfini, car cette classe n'est pas une classe d'agencement standard.
- Si une classe a un membre de référence, la classe n'est plus une présentation standard. Par conséquent, les tentatives d'accès aux données utilisées pour stocker la référence entraînent un comportement indéfini ou non spécifié.

En pratique, dans certains cas, une variable de référence peut être implémentée de manière similaire à une variable de pointeur et occuper ainsi la même quantité de stockage qu'un pointeur, alors que dans d'autres cas, une référence peut ne pas occuper d'espace car elle peut être optimisée. Par exemple, dans:

```
void f() {
    int x;
    int& r = x;
    // do something with r
}
```

le compilateur est libre de traiter simplement `r` comme un alias pour `x` et remplacer toutes les occurrences de `r` dans le reste de la fonction `f` à `x` , et d' allouer un stockage pour contenir `r` .

Ordre d'évaluation des arguments de fonction

Si une fonction a plusieurs arguments, l'ordre dans lequel ils sont évalués n'est pas spécifié. Le code suivant pourrait imprimer `x = 1, y = 2` ou `x = 2, y = 1` mais il n'est pas spécifié lequel.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}
int get_val() {
    static int x = 0;
    return ++x;
}
int main() {
    f(get_val(), get_val());
}
```

C ++ 17

En C ++ 17, l'ordre d'évaluation des arguments de fonction reste indéterminé.

Cependant, chaque argument de fonction est complètement évalué et l'objet appelant est garanti évalué avant tout argument de fonction.

```
struct from_int {
  from_int(int x) { std::cout << "from_int (" << x << ")\n"; }
};
int make_int(int x){ std::cout << "make_int (" << x << ")\n"; return x; }


void foo(from_int a, from_int b) {
}
void bar(from_int a, from_int b) {
}

auto which_func(bool b){
  std::cout << b?"foo":"bar" << "\n";
  return b?foo:bar;
}

int main(int argc, char const*const* argv) {
  which_func( true )( make_int(1), make_int(2) );
}
```

cela doit imprimer:

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

ou

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

il se peut qu’elle n’imprime *pas de* `bar` après une `make` ou `from`

```
bar
make_int(2)
make_int(1)
from_int(2)
from_int(1)
```

ou similaire. Avant C ++ 17 `bar` impression après `make_int` s était légale, comme le faisaient les deux `make_int` s avant de faire des `from_int` s.

Déplacé de l'état de la plupart des classes de bibliothèque standard

C ++ 11

Tous les conteneurs de bibliothèque standard sont laissés dans un état *valide mais non spécifié* après avoir été déplacés. Par exemple, dans le code suivant, `v2` contiendra `{1, 2, 3, 4}` après le déplacement, mais il n'est pas garanti que `v1` soit vide.

```
int main() {
    std::vector<int> v1{1, 2, 3, 4};
    std::vector<int> v2 = std::move(v1);
}
```

Certaines classes ont un état de déplacement défini avec précision. Le cas le plus important est celui de `std::unique_ptr<T>` , qui est garanti nul après avoir été déplacé.

Lire Comportement non spécifié en ligne:
https://riptutorial.com/fr/cplusplus/topic/4939/comportement-non-specifie

Chapitre 16: Comportements plus indéfinis en C ++

Introduction

Plus d'exemples sur la manière dont C ++ peut mal tourner.

Suite du comportement indéfini

Examples

Se référant à des membres non statiques dans les listes d'initialisation

Se référer à des membres non statiques dans les listes d'initialisation avant que le constructeur ne commence à s'exécuter peut entraîner un comportement indéfini. Cela résulte du fait que tous les membres ne sont pas construits à ce moment. A partir du brouillon standard:

> §12.7.1: Pour un objet avec un constructeur non trivial, le fait de se référer à un membre ou à une classe de base non statique de l'objet avant que le constructeur ne commence l'exécution entraîne un comportement indéfini.

Exemple

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
   int *p;
   X x;
   Y() : p(&x.j) { // undefined, x is not yet constructed
   }
};
```

Lire Comportements plus indéfinis en C ++ en ligne:
https://riptutorial.com/fr/cplusplus/topic/9885/comportements-plus-indefinis-en-c-plusplus

Chapitre 17: Concurrence avec OpenMP

Introduction

Cette rubrique couvre les bases de la concurrence en C ++ en utilisant OpenMP. OpenMP est documenté plus en détail dans la balise OpenMP .

Le parallélisme ou la simultanéité implique l'exécution de code en même temps.

Remarques

OpenMP ne nécessite pas d'en-tête ou de bibliothèque particulière, car il s'agit d'une fonctionnalité de compilateur intégrée. Toutefois, si vous utilisez des fonctions API OpenMP telles que `omp_get_thread_num()` , vous devrez inclure `omp.h` et sa bibliothèque.

Les instructions OpenMP `pragma` sont ignorées lorsque l'option OpenMP n'est pas activée lors de la compilation. Vous voudrez peut-être faire référence à l'option du compilateur dans le manuel de votre compilateur.

- GCC utilise `-fopenmp`
- Clang utilise `-fopenmp`
- MSVC utilise `/openmp`

Examples

OpenMP: Sections parallèles

Cet exemple illustre les bases de l'exécution de sections de code en parallèle.

Comme OpenMP est une fonctionnalité de compilateur intégrée, il fonctionne sur tous les compilateurs pris en charge sans inclure de bibliothèque. Vous souhaiterez peut-être inclure `omp.h` si vous souhaitez utiliser l'une des fonctionnalités de l'API openMP.

Exemple de code

```
std::cout << "begin ";
//    This pragma statement hints the compiler that the
//    contents within the { } are to be executed in as
//    parallel sections using openMP, the compiler will
//    generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    //    This pragma statement hints the compiler that
    //    this is a section that can be executed in parallel
    //    with other section, a single section will be executed
    //    by a single thread.
    //    Note that it is "section" as opposed to "sections" above
    #pragma omp section
```

```
    {
        std::cout << "hello " << std::endl;
        /** Do something **/
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /** Do something **/
    }
}
//    This line will not be executed until all the
//    sections defined above terminates
std::cout << "end" << std::endl;
```

Les sorties

Cet exemple produit 2 sorties possibles et dépend du système d'exploitation et du matériel. La sortie illustre également un problème de **condition de** concurrence qui se produirait à partir d'une telle implémentation.

SORTIE A	SORTIE B
commence bonjour la fin du monde	commence le monde bonjour fin

OpenMP: Sections parallèles

Cet exemple montre comment exécuter des morceaux de code en parallèle

```
std::cout << "begin ";
//    Start of parallel sections
#pragma omp parallel sections
{
    //    Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
//    end of parallel sections
std::cout << "end";
```

Sortie

- commence bonjour monde pour toujours fin

- commence le monde bonjour pour toujours fin
- commence bonjour fin du monde
- commencer à jamais bonjour fin du monde

Comme l'ordre d'exécution n'est pas garanti, vous pouvez observer l'une des sorties ci-dessus.

OpenMP: Parallel For Loop

Cet exemple montre comment diviser une boucle en parties égales et les exécuter en parallèle.

```
//    Splits element vector into element.size() / Thread Qty
//    and allocate that range for each thread.
#pragma omp parallel for
for    (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

//    Example Allocation (100 element per thread)
//    Thread 1 : 0 ~ 99
//    Thread 2 : 100 ~ 199
//    Thread 2 : 200 ~ 299
//    ...

//    Continue process
//    Only when all threads completed their allocated
//    loop job
...
```

* Veuillez prendre des précautions supplémentaires pour ne pas modifier la taille du vecteur utilisé en parallèle pour les boucles car les **index de plage alloués ne sont pas mis à jour automatiquement** .

OpenMP: collecte / réduction en parallèle

Cet exemple illustre un concept permettant d'effectuer une réduction ou une collecte à l'aide de `std::vector` et OpenMP.

Supposons que nous ayons un scénario où nous voulons que plusieurs threads nous aident à générer un tas de choses, `int` est utilisé ici pour simplifier et peut être remplacé par d'autres types de données.

Ceci est particulièrement utile lorsque vous devez fusionner des résultats provenant d'esclaves pour éviter les erreurs de segmentation ou les violations d'accès à la mémoire et que vous ne souhaitez pas utiliser de bibliothèques ou de bibliothèques de conteneurs de synchronisation personnalisées.

```
//    The Master vector
//    We want a vector of results gathered from slave threads
std::vector<int> Master;

//    Hint the compiler to parallelize this { } of code
//    with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
```

```
    //    In this area, you can write any code you want for each
    //    slave thread, in this case a vector to hold each of their results
    //    We don't have to worry about how many threads were spawn or if we need
    //    to repeat this declaration or not.
    std::vector<int> Slave;

    //    Tell the compiler to use all threads allocated for this parallel region
    //    to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    //    The nowait keyword tells the compiler that the slave threads don't
    //    have to wait for all other slaves to finish this for loop job
    #pragma omp for nowait
    for (size_t i = 0; i < 1000000; ++i
    {
        /* Do something */
        ....
        Slave.push_back(...);
    }

    //    Slaves that finished their part of the job
    //    will perform this thread by thread one at a time
    //    critical section ensures that only 0 or 1 thread performs
    //    the { } at any time
    #pragma omp critical
    {
        //    Merge slave into master
        //    use move iterators instead, avoid copy unless
        //    you want to use it for something else after this section
        Master.insert(Master.end(),
                      std::make_move_iterator(Slave.begin()),
                      std::make_move_iterator(Slave.end()));
    }
}

//    Have fun with Master vector
...
```

Lire Concurrence avec OpenMP en ligne: https://riptutorial.com/fr/cplusplus/topic/8222/concurrence-avec-openmp

Chapitre 18: constexpr

Introduction

`constexpr` est un mot - clé qui peut être utilisé pour marquer la valeur d'une variable en tant qu'expression constante, une fonction potentiellement utilisable dans les expressions constantes ou (depuis C ++ 17) une instruction if n'ayant qu'une seule de ses branches sélectionnée pour être compilée.

Remarques

Le mot-clé `constexpr` été ajouté en C ++ 11 mais depuis quelques années depuis la publication du standard C ++ 11, tous les compilateurs principaux ne l'ont pas supporté. au moment de la publication du standard C ++ 11. Au moment de la publication de C ++ 14, tous les principaux compilateurs prennent en charge `constexpr` .

Examples

variables constexpr

Une variable déclarée `constexpr` est implicitement `const` et sa valeur peut être utilisée comme une expression constante.

Comparaison avec `#define`

Un `constexpr` est un remplacement de type sécurisé pour les expressions de compilation basées sur `#define` . Avec `constexpr` l'expression évaluée à la compilation est remplacée par le résultat. Par exemple:

C ++ 11

```
int main()
{
   constexpr int N = 10 + 2;
   cout << N;
}
```

produira le code suivant:

```
cout << 12;
```

Une macro de compilation basée sur un pré-processeur serait différente. Considérer:

```
#define N 10 + 2

int main()
```

```
{
    cout << N;
}
```

produira:

```
cout << 10 + 2;
```

qui sera évidemment converti en `cout << 10 + 2;` . Cependant, le compilateur devrait faire plus de travail. En outre, cela crée un problème s'il n'est pas utilisé correctement.

Par exemple (avec `#define`):

```
cout << N * 2;
```

formes:

```
cout << 10 + 2 * 2; // 14
```

Mais un `constexpr` pré-évalué donnerait correctement `24` .

Comparaison avec `const`

Une variable `const` est une **variable** qui a besoin de mémoire pour son stockage. Un `constexpr` ne le fait pas. Un `constexpr` produit une constante de temps de compilation, qui ne peut pas être modifiée. Vous pouvez soutenir que `const` peut également ne pas être modifié. Mais considérez:

```
int main()
{
   const int size1 = 10;
   const int size2 = abs(10);

   int arr_one[size1];
   int arr_two[size2];
}
```

Avec la plupart des compilateurs, la deuxième instruction échouera (peut fonctionner avec GCC, par exemple). La taille de tout tableau, comme vous le savez peut-être, doit être une expression constante (c’est-à-dire qu’il en résulte une valeur à la compilation). La deuxième variable `size2` se voit attribuer une valeur qui est décidée à l'exécution (même si vous savez qu'il s'agit de `10` , pour le compilateur, ce n'est pas la compilation).

Cela signifie qu'un `const` peut ou peut ne pas être une constante constante de compilation. Vous ne pouvez pas garantir ou imposer qu'une valeur de `const` particulière est absolument à la compilation. Vous pouvez utiliser `#define` mais il a ses propres pièges.

Par conséquent, utilisez simplement:

C ++ 11

```
int main()
{
    constexpr int size = 10;

    int arr[size];
}
```

Une expression `constexpr` doit donner une valeur à la compilation. Ainsi, vous ne pouvez pas utiliser:

C ++ 11

```
constexpr int size = abs(10);
```

Sauf si la fonction (`abs`) renvoie elle-même un `constexpr` .

Tous les types de base peuvent être initialisés avec `constexpr` .

C ++ 11

```
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

Fait intéressant, et commodément, vous pouvez également utiliser `auto` :

C ++ 11

```
constexpr auto domain = ".COM";  // const char * const domain = ".COM"
constexpr auto PI = 3.14;        // constexpr double
```

fonctions constexpr

Une fonction déclarée `constexpr` est implicitement `constexpr` et les appels à une telle fonction peuvent générer des expressions constantes. Par exemple, la fonction suivante, si elle est appelée avec des arguments d'expression constante, produit également une expression constante:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Ainsi, le résultat de l'appel de fonction peut être utilisé en tant que tableau lié ou argument de modèle, ou pour initialiser une variable `constexpr` :

C ++ 11

```
int main()
```

```
{
    constexpr int S = Sum(10,20);

    int Array[S];
    int Array2[Sum(20,30)]; // 50 array size, compile time
}
```

Notez que si vous supprimez `constexpr` de la spécification de type de retour de la fonction, l'affectation à `S` ne fonctionnera pas, car `S` est une variable `constexpr` et doit être `constexpr` à un const de compilation. De même, la taille du tableau ne sera pas non plus une expression constante, si la fonction `Sum` n'est pas `constexpr` .

Ce qui est intéressant `constexpr` fonctions `constexpr` , c'est que vous pouvez aussi l'utiliser comme des fonctions ordinaires:

C ++ 11

```
int a = 20;
auto sum = Sum(a, abs(-20));
```

`Sum` ne sera plus une fonction `constexpr` , elle sera compilée en tant que fonction ordinaire, en prenant des arguments variables (non constants) et en renvoyant une valeur non constante. Vous n'avez pas besoin d'écrire deux fonctions.

Cela signifie également que si vous essayez d'attribuer un tel appel à une variable non-const, il ne sera pas compilé:

C ++ 11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

La raison en est simple: une `constexpr` doit être attribuée à `constexpr` . Cependant, l'appel de fonction ci-dessus fait de `Sum` un non- `constexpr` (la valeur R est non-const, mais la valeur L se déclare `constexpr`).

La fonction `constexpr` **doit** également renvoyer une constante à la compilation. Ce qui suit ne compilera pas:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;     // ERROR
    return a + b;
}
```

Parce que `a1` est une *variable* non-constexpr et interdit à la fonction d'être une véritable fonction `constexpr` . Le rendre `constexpr` et lui assigner `a` va également ne pas fonctionner - puisque la valeur d' `a` paramètre (paramètre entrant) n'est toujours pas connue:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   constexpr int a1 = a;     // ERROR
   ..
```

De plus, la suite ne compilera pas non plus:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   return abs(a) + b; // or abs(a) + abs(b)
}
```

Puisque `abs(a)` n'est pas une expression constante (même `abs(10)` ne fonctionnera pas, puisque `abs` ne renvoie pas de `constexpr int` !

Et ça?

C ++ 11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

Nous avons conçu notre propre fonction `Abs` qui est un `constexpr` , et le corps d' `Abs` ne `constexpr` pas non plus la règle. En outre, sur le site d'appel (à l'intérieur de la `Sum`), l'expression donne lieu à une `constexpr` . Par conséquent, l'appel à la `Sum(-10, 20)` sera une expression constante à la compilation qui donnera `30` .

Static si déclaration

C ++ 17

L' `if constexpr` peut être utilisée pour compiler de manière conditionnelle du code. La condition doit être une expression constante. La branche non sélectionnée est *supprimée.* Une instruction ignorée à l'intérieur d'un modèle n'est pas instanciée. Par exemple:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
  // ... handle p
  if constexpr (sizeof...(rs) > 0)
    g(rs...);  // never instantiated with an empty argument list
}
```

De plus, les variables et les fonctions qui ne sont utilisées que dans des instructions ignorées ne sont pas obligatoirement définies et les instructions de `return` ignorées ne sont pas utilisées pour la déduction de type retour de fonction.

`if constexpr` est distinct de `#ifdef` . `#ifdef` compile conditionnellement le code, mais uniquement sur la base de conditions pouvant être évaluées au moment du prétraitement. Par exemple, `#ifdef` n'a pas pu être utilisé pour compiler conditionnellement du code en fonction de la valeur d'un paramètre de modèle. Par contre, `if constexpr` ne peut pas être utilisé pour écarter un code syntaxiquement invalide, alors `#ifdef` peut.

```
if constexpr(false) {
    foobar;  // error; foobar has not been declared
    std::vector<int> v("hello, world");  // error; no matching constructor
}
```

Lire constexpr en ligne: https://riptutorial.com/fr/cplusplus/topic/3899/constexpr

Chapitre 19: Conteneurs C ++

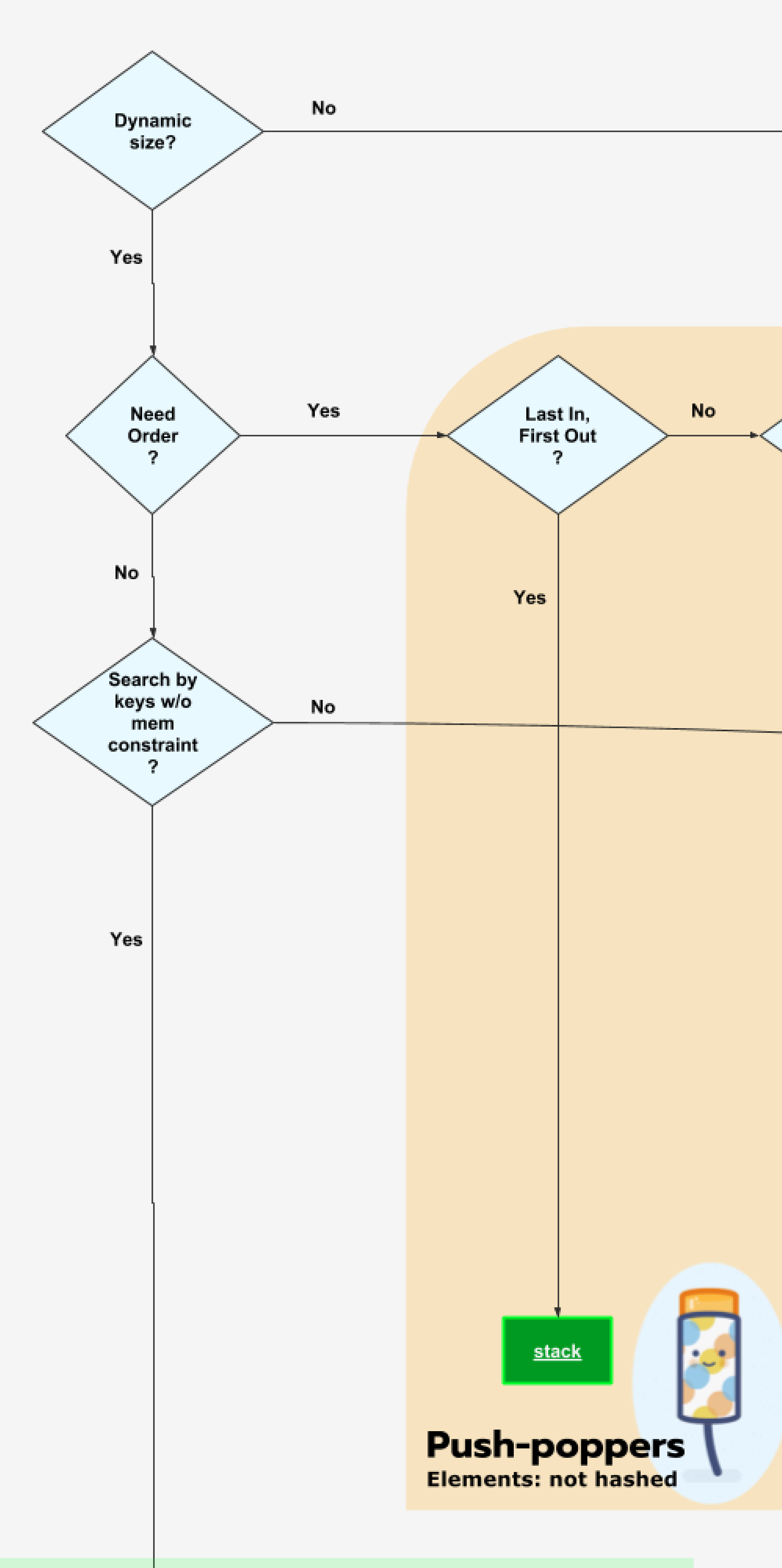
Introduction

Les conteneurs C ++ stockent une collection d'éléments. Les conteneurs incluent des vecteurs, des listes, des cartes, etc. En utilisant des modèles, les conteneurs C ++ contiennent des collections de primitives (par exemple, des ints) ou des classes personnalisées (par exemple, MyClass).

Examples

Organigramme des conteneurs C ++

Choisir le conteneur C ++ à utiliser peut être difficile, alors voici un organigramme simple pour vous aider à déterminer quel conteneur est le mieux adapté à votre tâche.


Dynamic size?
No
Yes
Need Order ?
Yes
Last In, First Out ?
No
No
Yes
Search by keys w/o mem constraint ?
No
Yes
stack
Push-poppers
Elements: not hashed
Unordered
Elements: hash code

. Ce petit graphique dans l'organigramme est de Megan Hopkins

Lire Conteneurs C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/10848/conteneurs-c-plusplus

Chapitre 20: Contrôle de flux

Remarques

Consultez le sujet des boucles pour les différents types de boucles.

Examples

Cas

Introduit une étiquette de cas d'une instruction switch. L'opérande doit être une expression constante et correspondre à la condition du commutateur dans le type. Lorsque l'instruction switch est exécutée, elle saute à l'étiquette de cas avec un opérande égal à la condition, le cas échéant.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

commutateur

Selon le standard C ++,

> L'instruction `switch` entraîne le transfert du contrôle sur l'une des instructions en fonction de la valeur d'une condition.

Le mot clé `switch` est suivi d'une condition entre parenthèses et d'un bloc, pouvant contenir des étiquettes de `case` et une étiquette `default` facultative. Lorsque l'instruction switch est exécutée, le contrôle est transféré vers une étiquette de `case` avec une valeur correspondant à celle de la condition, le cas échéant, ou à l'étiquette `default` , le cas échéant.

La condition doit être une expression ou une déclaration, de type entier ou énumération, ou un type de classe avec une fonction de conversion en type entier ou énumération.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
```

```
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

capture

Le mot clé `catch` introduit un gestionnaire d'exceptions, c'est-à-dire un bloc dans lequel le contrôle sera transféré lorsqu'une exception de type compatible est levée. Le mot-clé `catch` est suivi d'une *déclaration d'exception* entre parenthèses, dont la forme est similaire à celle d'une déclaration de paramètre de fonction: le nom du paramètre peut être omis et les points de suspension `...` sont autorisés, ce qui correspond à tout type. Le gestionnaire d'exceptions ne traitera l'exception que si sa déclaration est compatible avec le type de l'exception. Pour plus de détails, voir les exceptions en attente .

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

défaut

Dans une instruction switch, introduit une étiquette sur laquelle on sautera si la valeur de la condition n'est pas égale à l'une des valeurs des étiquettes de cas.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

C ++ 11

Définit un constructeur par défaut, un constructeur de copie, un constructeur de déplacement, un destructeur, un opérateur d'affectation de copie ou un opérateur d'affectation de déplacement pour

avoir son comportement par défaut.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

si

Introduit une instruction if. Le mot `if` clé `if` doit être suivi d'une condition entre parenthèses, qui peut être une expression ou une déclaration. Si la condition est vraie, le sous-élément après la condition sera exécuté.

```
int x;
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

autre

Le premier sous-composant d'une instruction if peut être suivi du mot-clé `else` . Le sous-composant après le mot-clé `else` sera exécuté lorsque la condition est Falsey (c'est-à-dire lorsque le premier sous-composant n'est pas exécuté).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

aller à

Passe à une instruction étiquetée, qui doit être située dans la fonction en cours.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
    result = true;
  end:
    release_widget(widget);
    return result;
```

```
}
```

revenir

Renvoie le contrôle d'une fonction à son appelant.

Si `return` a un opérande, l'opérande est converti en type de retour de la fonction et la valeur convertie est renvoyée à l'appelant.

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

Si `return` n'a pas d'opérande, la fonction doit avoir un type de retour `void` . En tant que cas particulier, une fonction de retour `void` peut également renvoyer une expression si l'expression est de type `void` .

```
void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}
```

Lorsque `main` retourne, `std::exit` est implicitement appelé avec la valeur de retour et la valeur est donc renvoyée dans l'environnement d'exécution. (Cependant, le retour de `main` détruit les variables locales automatiques, alors que l'appel de `std::exit` ne le fait pas directement.)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

jeter

1. Lorsque `throw` se produit dans une expression avec un opérande, son effet est de lancer une exception , qui est une copie de l'opérande.

   ```
   void print_asterisks(int count) {
       if (count < 0) {
           throw std::invalid_argument("count cannot be negative!");
       }
   ```

```
    while (count--) { putchar('*'); }
}
```

2. Lorsque `throw` se produit dans une expression sans opérande, son effet est de renvoyer l'exception en cours . S'il n'y a pas d'exception actuelle, `std::terminate` est appelée.

   ```
   try {
       // something risky
   } catch (const std::bad_alloc&) {
       std::cerr << "out of memory" << std::endl;
   } catch (...) {
       std::cerr << "unexpected exception" << std::endl;
       // hope the caller knows how to handle this exception
       throw;
   }
   ```

3. Lorsque `throw` se produit dans un déclarateur de fonction, il introduit une spécification d'exception dynamique, qui répertorie les types d'exceptions que la fonction est autorisée à propager.

   ```
   // this function might propagate a std::runtime_error,
   // but not, say, a std::logic_error
   void risky() throw(std::runtime_error);
   // this function can't propagate any exceptions
   void safe() throw();
   ```

   Les spécifications d'exception dynamiques sont obsolètes à partir de C ++ 11.

Notez que les deux premières utilisations de `throw` listées ci-dessus constituent des expressions plutôt que des déclarations. (Le type d'une expression est `void` .) Cela permet de les imbriquer dans des expressions, comme ceci:

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

essayer

Le mot-clé `try` est suivi d'un bloc ou d'une liste d'initialisation du constructeur, puis d'un bloc (voir ici). Le bloc try est suivi d'un ou plusieurs blocs catch . Si une exception se propage hors du bloc try, chacun des blocs catch correspondants après le bloc try a la possibilité de gérer l'exception, si les types correspondent.

```
std::vector<int> v(N);     // if an exception is thrown here,
                           // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
                           // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
}
```

Structures conditionnelles: si, si..se

si et sinon:

il permet de vérifier si l'expression donnée renvoie true ou false et agit comme tel:

```
if (condition) statement
```

la condition peut être toute expression C ++ valide qui renvoie quelque chose qui soit vérifié par rapport à la vérité / au mensonge, par exemple:

```
if (true) { /* code here */ }  // evaluate that true is true and execute the code in the
brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

la condition peut être n'importe quoi, une fonction, une variable ou une comparaison par exemple

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evalute the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any
non zero value will be true,
```

Si nous voulons vérifier plusieurs expressions, nous pouvons le faire de deux manières:

en utilisant des opérateurs binaires :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside
the scope here
if (a || b ) { } //true if a or b is true
```

en utilisant if / ifelse / else :

pour un simple commutateur si ou sinon

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

pour des choix multiples:

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
```

```
}
```

Cependant, il faut noter que vous devez utiliser ' **switch** ' à la place si votre code vérifie la valeur de la même variable

Instructions de saut: pause, continuer, aller, sortir.

L'instruction de pause:

En utilisant break, nous pouvons laisser une boucle même si la condition de sa fin n'est pas remplie. Il peut être utilisé pour terminer une boucle infinie ou pour le forcer à se terminer avant sa fin naturelle

La syntaxe est

```
break;
```

Exemple: nous utilisons souvent `break` dans l' `switch` cas, par exemple une fois par cas i commutateur est satisfait alors le bloc de code de cette condition est exécutée.

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

dans ce cas, si le cas 1 est satisfait, alors le bloc 1 est exécuté, ce que nous voulons vraiment, c'est que le bloc1 soit traité, mais les blocs restants, block2, block3 et blockdefault sont traités même si seul le cas 1 est satisfait .Pour éviter cela, nous utilisons break à la fin de chaque bloc comme:

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

Ainsi, un seul bloc est traité et le contrôle quitte la boucle de commutation.

break peut également être utilisé dans d'autres boucles conditionnelles et non conditionnelles comme `if` , `while` , `for` etc.

Exemple:

```
if(condition1){
```

```
    ....
    if(condition2){
     .......
     break;
     }
  ...
}
```

L'instruction continue:

L'instruction continue oblige le programme à ignorer le reste de la boucle dans l'itération actuelle, comme si la fin du bloc d'instruction aurait été atteinte, ce qui l'amènerait à passer à l'itération suivante.

La syntaxe est

```
continue;
```

Exemple considérer les points suivants:

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

qui produit la sortie:

```
 @1
 @3
 @5
 @7
 @9
```

i ce code à chaque fois que la condition `i%2==0` est satisfaite `continue` est traitée, ce qui provoque le compilateur pour sauter tout le code restant (impression @ i) et l' état incrément / décrément de la boucle est exécuté.

```
for (initialisation;condition;increment/decrement)
    {
     ...
      if (True Condition)
          continue;
      ...
    }
```

Continues Loop with the Next Value

L'instruction goto:

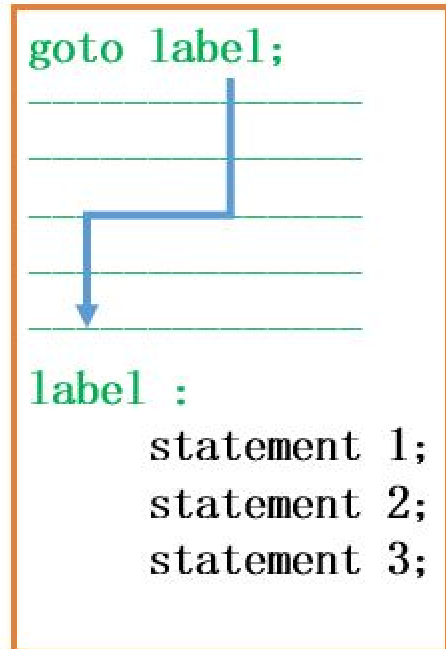
Cela permet de sauter à un autre point du programme. Vous devez utiliser cette fonctionnalité avec précaution car son exécution ignore tout type de limitation d'imbrication. Le point de

destination est identifié par une étiquette, qui est ensuite utilisée comme argument pour l'instruction goto. Une étiquette est faite d'un identifiant valide suivi d'un deux-points (:)

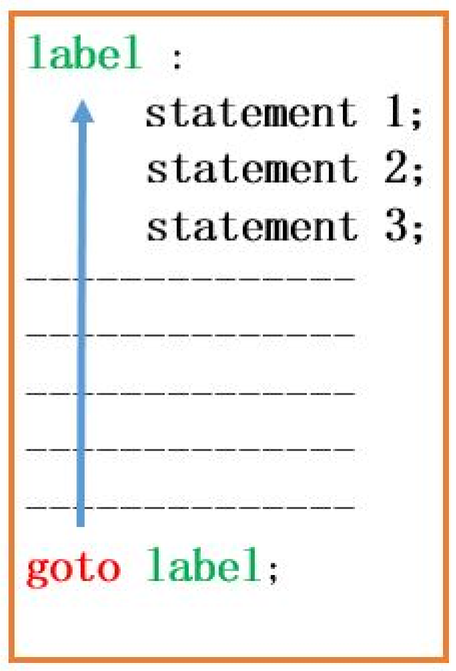
La syntaxe est

```
goto label;
..
.
label: statement;
```

Remarque: L' *utilisation de l'instruction goto est fortement déconseillée car elle rend difficile le suivi du flux de contrôle d'un programme, rendant le programme difficile à comprendre et à modifier.*



Forward Reference



Backward Reference

Exemple :

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
        goto STEP;
     }

   cout << "value of num : " << num << endl;
   num = num + 1;
 }while( num < 10 );
```

sortie:

```
value of num : 1
```

```
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

chaque fois que la condition `num%2==0` est satisfaite, le goto envoie le contrôle d'exécution au début de la boucle `do-while` while.

La fonction de sortie:

`exit` est une fonction définie dans `cstdlib` . Le but de `exit` est de terminer le programme en cours avec un code de sortie spécifique. Son prototype est:

```
void exit (int exit code);
```

`cstdlib` définit les codes de sortie standard `EXIT_SUCCESS` et `EXIT_FAILURE` .

Lire Contrôle de flux en ligne: https://riptutorial.com/fr/cplusplus/topic/7837/controle-de-flux

Chapitre 21: Conversions de type explicites

Introduction

Une expression peut être *explicitement converti* ou *casté* en type `T` en utilisant `dynamic_cast<T>` , `static_cast<T>` , `reinterpret_cast<T>` ou `const_cast<T>` , en fonction de ce type de fonte est destiné.

C ++ prend également en charge la notation de type cast, `T(expr)` et la notation cast de type C, `(T)expr` .

Syntaxe

- *spécificateur de type simple* `( )`
- *simple-type-specifier* `(` *liste-expression* `)`
- *simple-type-specifier braced-init-list*
- *typename-specifier* `( )`
- *typename-specifier* `(` *liste d'expressions* `)`
- *typename-specifier braced- init-list*
- `dynamic_cast` `<` *id-type* `>` `(` *expression* `)`
- `static_cast` `<` *id-type* `>` `(` *expression* `)`
- `reinterpret_cast` `<` *id-type* `>` `(` *expression* `)`
- `const_cast` `<` *id-type* `>` `(` *expression* `)`
- *expression- type* `(` *type-id* `)`

Remarques

Les six notations de la distribution ont une chose en commun:

- La conversion en un type de référence lvalue, comme dans `dynamic_cast<Derived&>(base)` , produit une lvalue. Par conséquent, si vous voulez faire quelque chose avec le même objet, mais le traiter comme un type différent, vous devez le convertir en un type de référence lvalue.
- Le passage à un type de référence rvalue, comme dans `static_cast<string&&>(s)` , génère une valeur.
- Le passage à un type sans référence, comme dans `(int)x` , donne une valeur qui peut être considérée comme une *copie* de la valeur en cours de projection, mais d'un type différent de l'original.

Le mot-clé `reinterpret_cast` est responsable de l'exécution de deux types de conversions "dangereuses":

- Les conversions "type punning" , qui peuvent être utilisées pour accéder à la mémoire d'un type comme s'il s'agissait d'un type différent.
- Conversions entre types entiers et types de pointeurs , dans les deux sens.

Le mot clé `static_cast` peut effectuer différentes conversions:

- Base aux conversions dérivées
- Toute conversion pouvant être effectuée par une initialisation directe, y compris les conversions implicites et les conversions appelant un constructeur explicite ou une fonction de conversion. Voir ici et ici pour plus de détails.
- `void` , ce qui élimine la valeur de l'expression.

```
// on some compilers, suppresses warning about x being unused
static_cast<void>(x);
```

- Entre les types arithmétiques et énumération, et entre différents types d'énumération. Voir les conversions enum
- De pointeur vers membre de classe dérivée, pointeur sur membre de classe de base. Les types indiqués doivent correspondre. Voir la conversion pour baser la conversion des pointeurs en membres
- `void*` à `T*` .

C ++ 11

- D'une lvalue à une xvalue, comme dans `std::move` . Voir sémantique de déplacement .

Examples

Base à la conversion dérivée

Un pointeur sur la classe de base peut être converti en un pointeur sur la classe dérivée à l'aide de `static_cast` . `static_cast` n'effectue aucune vérification au moment de l'exécution et peut entraîner un comportement indéfini lorsque le pointeur ne pointe pas vers le type souhaité.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1;                        // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
                                         // point to a Derived object
```

De même, une référence à une classe de base peut être convertie en une référence à une classe dérivée à l'aide de `static_cast` .

```
struct Base {};
struct Derived : Base {};
Derived d;
```

```
Base& r1 = d;
Derived& r2 = r1;                        // error; cast required
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object
```

Si le type de source est polymorphe, `dynamic_cast` peut être utilisé pour effectuer une conversion de base en dérivée. Il effectue un contrôle d'exécution et l'échec est récupérable au lieu de générer un comportement indéfini. Dans le cas du pointeur, un pointeur nul est renvoyé en cas d'échec. Dans le cas de référence, une exception est levée en cas d'échec du type `std::bad_cast` (ou d'une classe dérivée de `std::bad_cast`).

```
struct Base { virtual ~Base(); }; // Base is polymorphic
struct Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer
```

Jetant la constance

Un pointeur sur un objet const peut être converti en un pointeur sur un objet non-const à l'aide du mot-clé `const_cast` . Ici, nous utilisons `const_cast` pour appeler une fonction qui n'est pas constante. Il n'accepte qu'un argument non-const `char*` même s'il n'écrit jamais via le pointeur:

```
void bad_strlen(char*);
const char* s = "hello, world!";
bad_strlen(s);                       // compile error
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` à référence type peut être utilisé pour convertir une lvalue qualifiée en const en une valeur non-const-qualifiée.

`const_cast` est dangereux car il empêche le système de type C ++ de vous empêcher de modifier un objet const. Cela entraîne un comportement indéfini.

```
const int x = 123;
int& mutable_x = const_cast<int&>(x);
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Type de conversion

Un pointeur (resp. Référence) vers un type d'objet peut être converti en un pointeur (resp. Référence) vers tout autre type d'objet à l'aide de `reinterpret_cast` . Cela n'appelle aucun constructeur ni aucune fonction de conversion.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                              // maybe this modifies x (see below)
```

C ++ 11

Le résultat de `reinterpret_cast` représente la même adresse que l'opérande, à condition que l'adresse soit correctement alignée pour le type de destination. Sinon, le résultat n'est pas spécifié.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

C ++ 11

Le résultat de `reinterpret_cast` n'est pas spécifié, sauf qu'un pointeur (resp. Référence) survivra à un aller-retour du type source au type de destination et inversement, tant que les exigences d'alignement du type de destination ne sont pas plus strictes que celles du type source.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

Sur la plupart des implémentations, `reinterpret_cast` ne modifie pas l'adresse, mais cette exigence n'a pas été normalisée avant C ++ 11.

`reinterpret_cast` peut également être utilisé pour convertir un type de pointeur en membre de données en un autre, ou un type de pointeur vers une fonction membre vers un autre.

L'utilisation de `reinterpret_cast` est considérée comme dangereuse car la lecture ou l'écriture via un pointeur ou une référence obtenue à l'aide de `reinterpret_cast` peut déclencher un comportement indéfini lorsque les types source et destination ne sont pas liés.

Conversion entre pointeur et entier

Un pointeur d'objet (y compris `void*`) ou un pointeur de fonction peut être converti en un type entier en utilisant `reinterpret_cast` . Cela ne compilera que si le type de destination est suffisamment long. Le résultat est défini par l'implémentation et donne généralement l'adresse numérique de l'octet en mémoire sur lequel pointe le pointeur.

En général, `long` ou `unsigned long` est suffisamment long pour contenir une valeur de pointeur, mais cela n'est pas garanti par la norme.

C ++ 11

Si les types `std::intptr_t` et `std::uintptr_t` existent, ils sont garantis suffisamment longs pour contenir un `void*` (et donc tout pointeur sur le type d'objet). Cependant, ils ne sont pas garantis pour contenir un pointeur de fonction.

De même, `reinterpret_cast` peut être utilisé pour convertir un type entier en un type de pointeur. Là encore, le résultat est défini par la mise en œuvre, mais une valeur de pointeur est garantie inchangée par un aller-retour à travers un type entier. La norme ne garantit pas que la valeur zéro

est convertie en un pointeur nul.

```
void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;
register_callback(my_callback,
                  reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Conversion par constructeur explicite ou fonction de conversion explicite

Une conversion impliquant l'appel d'un constructeur explicite ou d'une fonction de conversion ne peut pas être effectuée implicitement. Nous pouvons demander que la conversion soit faite explicitement en utilisant `static_cast` . La signification est la même que celle d'une initialisation directe, sauf que le résultat est temporaire.

```
class C {
    std::unique_ptr<int> p;
  public:
    explicit C(int* p) : p(p) {}
};
void f(C c);
void g(int* p) {
    f(p);                 // error: C::C(int*) is explicit
    f(static_cast<C>(p)); // ok
    f(C(p));              // equivalent to previous line
    C c(p); f(c);         // error: C is not copyable
}
```

Conversion implicite

`static_cast` peut effectuer toute conversion implicite. Cette utilisation de `static_cast` peut parfois être utile, comme dans les exemples suivants:

- Lors du passage d'arguments à une ellipse, le type d'argument "attendu" n'est pas connu statiquement, donc aucune conversion implicite ne se produira.

  ```
  const double x = 3.14;
  printf("%d\n", static_cast<int>(x)); // prints 3
  // printf("%d\n", x); // undefined behaviour; printf is expecting an int here
  // alternative:
  // const int y = x; printf("%d\n", y);
  ```

  Sans la conversion de type explicite, un objet `double` serait transmis aux points de suspension et un comportement indéfini se produirait.

- Un opérateur d'affectation de classe dérivé peut appeler un opérateur d'affectation de classe de base comme suit:

  ```
  struct Base { /* ... */ };
  ```

```
struct Derived : Base {
    Derived& operator=(const Derived& other) {
        static_cast<Base&>(*this) = other;
        // alternative:
        // Base& this_base_ref = *this; this_base_ref = other;
    }
};
```

Conversions Enum

`static_cast` peut convertir un type entier ou à virgule flottante en un type d'énumération (de portée ou non), et *vice versa.* Il peut également convertir entre les types d'énumération.

- La conversion d'un type d'énumération non découpé en un type arithmétique est une conversion implicite. il est possible, mais pas nécessaire d'utiliser `static_cast` .

C ++ 11

- Lorsqu'un type d'énumération de portée est converti en un type arithmétique:
  - Si la valeur de l'énumération peut être représentée exactement dans le type de destination, le résultat est cette valeur.
  - Sinon, si le type de destination est un type entier, le résultat n'est pas spécifié.
  - Sinon, si le type de destination est un type à virgule flottante, le résultat est identique à celui de la conversion au type sous-jacent, puis au type à virgule flottante.

  Exemple:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};
Format f = Format::PDF;
int a = f;                         // error
int b = static_cast<int>(f);       // ok; b is 1000
char c = static_cast<char>(f);     // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- Lorsqu'un type d'entier ou d'énumération est converti en un type d'énumération:
  - Si la valeur d'origine est comprise dans la plage de l'énum de destination, le résultat est cette valeur. Notez que cette valeur peut être inégale pour tous les énumérateurs.
  - Sinon, le résultat est non spécifié (<= C ++ 14) ou indéfini (> = C ++ 17).

  Exemple:

```
enum Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
```

```
Scale s1 = 1;                     // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

C ++ 11

- Lorsqu'un type à virgule flottante est converti en un type d'énumération, le résultat est identique à la conversion au type sous-jacent de l'énumération, puis au type enum.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

Dérivé de la conversion de base pour les pointeurs en membres

Un pointeur sur le membre de la classe dérivée peut être converti en un pointeur sur le membre de la classe de base à l'aide de `static_cast` . Les types indiqués doivent correspondre.

Si l'opérande est un pointeur nul sur la valeur du membre, le résultat est également un pointeur nul sur la valeur du membre.

Sinon, la conversion n'est valide que si le membre désigné par l'opérande existe réellement dans la classe de destination ou si la classe de destination est une classe de base ou dérivée de la classe contenant le membre désigné par l'opérande. `static_cast` ne vérifie pas la validité. Si la conversion n'est pas valide, le comportement est indéfini.

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1;                              // ok; implicit conversion
int B::*p3 = p2;                              // error
int B::*p4 = static_cast<int B::*>(p2);       // ok; p4 is equal to p1
int A::*p5 = static_cast<int A::*>(p2);       // undefined; p2 points to x, which is a member
                                              // of the unrelated class B
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z
int A::*p8 = static_cast<int A::*>(p6);       // error: types don't match
```

annule * à T *

En C ++, `void*` ne peut pas être implicitement converti en `T*` où `T` est un type d'objet. À la place, `static_cast` doit être utilisé pour effectuer la conversion de manière explicite. Si l'opérande pointe réellement sur un objet `T` , le résultat pointe sur cet objet. Sinon, le résultat n'est pas spécifié.

C ++ 11

Même si l'opérande ne pointe pas sur un objet `T` , tant que l'opérande pointe sur un octet dont l'adresse est correctement alignée pour le type `T` , le résultat des points de conversion sur le même octet.

```
// allocating an array of 100 ints, the hard way
int* a = malloc(100*sizeof(*a));                    // error; malloc returns void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok
// int* a = new int[100];                           // no cast needed
// std::vector<int> a(100);                         // better

const char c = '!';
const void* p1 = &c;
const char* p2 = p1;                           // error
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c
const int* p4 = static_cast<const int*>(p1);   // unspecified in C++03;
                                               // possibly unspecified in C++11 if
                                               // alignof(int) > alignof(char)
char* p5 = static_cast<char*>(p1);             // error: casting away constness
```

Coulée de style C

Le casting C-Style peut être considéré comme étant le «meilleur effort» et est nommé ainsi car il est le seul qui puisse être utilisé en C. La syntaxe de cette distribution est la `(NewType)variable` .

Chaque fois que cette conversion est utilisée, elle utilise l'un des modèles c ++ suivants (dans l'ordre):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

Le casting fonctionnel est très similaire, bien que quelques restrictions résultent de sa syntaxe: `NewType(expression)` . Par conséquent, seuls les types sans espace peuvent être convertis.

Il est préférable d'utiliser le nouveau cast c ++, car plus lisible et facilement repérable n'importe où dans un code source C ++ et que les erreurs seront détectées au moment de la compilation, plutôt qu'au moment de l'exécution.

Comme cette distribution peut entraîner un `reinterpret_cast` involontaire, elle est souvent considérée comme dangereuse.

Lire Conversions de type explicites en ligne: https://riptutorial.com/fr/cplusplus/topic/3090/conversions-de-type-explicites

Chapitre 22: Copier Elision

Examples

But de l'élision de la copie

Il y a des endroits dans le standard où un objet est copié ou déplacé pour initialiser un objet. L'élision de la copie (parfois appelée optimisation de la valeur de retour) est une optimisation par laquelle, dans certaines circonstances spécifiques, un compilateur est autorisé à éviter la copie ou le déplacement même si la norme indique que cela doit se produire.

Considérons la fonction suivante:

```
std::string get_string()
{
  return std::string("I am a string.");
}
```

Selon la formulation stricte du standard, cette fonction initialisera un `std::string` temporaire, puis copiera / déplacera celui-ci dans l'objet de valeur de retour, puis détruira le temporaire. La norme est très claire: c'est ainsi que le code est interprété.

Copier elision est une règle permettant à un compilateur C ++ d' *ignorer* la création du fichier temporaire et de sa copie / destruction ultérieure. En d'autres termes, le compilateur peut prendre l'expression d'initialisation pour le temporaire et initialiser directement la valeur de retour de la fonction. Cela évite évidemment les performances.

Cependant, il a deux effets visibles sur l'utilisateur:

1. Le type doit avoir le constructeur copy / move qui aurait été appelé. Même si le compilateur élimine la copie / le déplacement, le type doit toujours pouvoir être copié / déplacé.

2. Les effets secondaires des constructeurs de copier / déplacer ne sont pas garantis dans les cas où une élision peut se produire. Considérer ce qui suit:

C ++ 11

```
struct my_type
{
  my_type() = default;
  my_type(const my_type &) {std::cout <<"Copying\n";}
  my_type(my_type &&) {std::cout <<"Moving\n";}
};

my_type func()
{
  return my_type();
}
```

Que va faire l'appel `func` ? Eh bien, il n'imprimera jamais "Copie", car le temporaire est une valeur et `my_type` est un type mobile. Alors, va-t-il imprimer "Moving"?

Sans la règle d'élision de la copie, il faudrait toujours imprimer "Moving". Mais comme la règle de copie existe, le constructeur de déplacement peut ou non être appelé; il dépend de la mise en œuvre.

Et par conséquent, vous ne pouvez pas compter sur l'appel de constructeurs copier / déplacer dans des contextes où l'élision est possible.

Elision étant une optimisation, votre compilateur peut ne pas prendre en charge l'élision dans tous les cas. Et que le compilateur élimine ou non un cas particulier, le type doit toujours prendre en charge l'opération en cours. Ainsi, si une construction de copie est supprimée, le type doit toujours avoir un constructeur de copie, même s'il ne sera pas appelé.

Elision de copie garantie

C ++ 17

Normalement, l'élision est une optimisation. Bien que quasiment tous les compilateurs prennent en charge l'élision de copie dans les cas les plus simples, l'élision impose toujours un fardeau particulier aux utilisateurs. A savoir, le type qui est copié / déplacé *doit* toujours avoir l'opération de copie / déplacement qui a été éluée.

Par exemple:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  return std::lock_guard<std::mutex>(a_mutex);
}
```

Cela peut être utile dans les cas où `a_mutex` est un mutex détenu par un système privé, mais un utilisateur externe peut vouloir y placer un verrou.

Ce n'est pas légal non plus, car `std::lock_guard` ne peut pas être copié ou déplacé. Même si pratiquement tous les compilateurs C ++ élimine la copie / le déplacement, la norme *nécessite* toujours *que* le type dispose de cette opération.

Jusqu'à C ++ 17.

C ++ 17 impose l'élision en redéfinissant efficacement la signification même de certaines expressions afin qu'aucune copie / déplacement n'ait lieu. Considérez le code ci-dessus.

Sous un libellé pré-C ++ 17, ce code dit de créer un temporaire, puis d'utiliser le temporaire pour copier / déplacer dans la valeur de retour, mais la copie temporaire peut être éluée. Sous C ++ 17, cela ne crée pas du tout un temporaire.

En C ++ 17, toute expression de valeur utilisée pour initialiser un objet du même type que l'expression ne génère pas de valeur temporaire. L'expression initialise directement cet objet. Si

vous retournez une valeur du même type que la valeur de retour, le type n'a pas besoin d'avoir un constructeur de copier / déplacer. Et par conséquent, sous les règles C ++ 17, le code ci-dessus peut fonctionner.

Le langage C ++ 17 fonctionne dans les cas où le type de la valeur correspond au type en cours d'initialisation. Donc, étant donné `get_lock` ci-dessus, cela ne nécessitera pas non plus de copier / déplacer:

```
std::lock_guard the_lock = get_lock();
```

Comme le résultat de `get_lock` est une expression de valeur utilisée pour initialiser un objet du même type, aucune copie ou déplacement ne se produira. Cette expression ne crée jamais de temporaire; il est utilisé pour initialiser directement `the_lock` . Il n'y a pas d'élision car il n'y a pas de copie / mouvement à élider élide.

Le terme "élision de copie garantie" est donc quelque peu trompeur, mais c'est le nom de la fonctionnalité telle qu'elle est proposée pour la normalisation C ++ 17 . Cela ne garantit pas du tout l'élision; cela *élimine* complètement le copier / déplacer, redéfinissant le C ++ de sorte qu'il n'y ait jamais eu de copier / déplacer.

Cette fonctionnalité ne fonctionne que dans les cas impliquant une expression de valeur. En tant que tel, cela utilise les règles d'élision habituelles:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  std::lock_guard<std::mutex> my_lock(a_mutex);
  //Do stuff
  return my_lock;
}
```

Bien qu'il s'agisse d'un cas valable pour l'élision de la copie, les règles C ++ 17 *n'éliminent* pas la copie / le déplacement dans ce cas. En tant que tel, le type doit toujours avoir un constructeur copy / move à utiliser pour initialiser la valeur de retour. Et comme `lock_guard` ne le fait pas, c'est toujours une erreur de compilation. Les implémentations sont autorisées à refuser les copies lors de la transmission ou du renvoi d'un objet de type trivialement copiable. Cela permet de déplacer de tels objets dans des registres, ce que certains ABI peuvent exiger dans leurs conventions d'appel.

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

Valeur de retour elision

Si vous retournez une expression de valeur à partir d'une fonction et que l'expression de valeur a

le même type que le type de retour de la fonction, la copie de la valeur temporaire peut être supprimée:

```
std::string func()
{
  return std::string("foo");
}
```

Presque tous les compilateurs éluderont la construction temporaire dans ce cas.

Élision des paramètres

Lorsque vous transmettez un argument à une fonction, et que l'argument est une expression de valeur du type de paramètre de la fonction, et que ce type n'est pas une référence, la construction de la valeur peut être élidée.

```
void func(std::string str) { ... }

func(std::string("foo"));
```

Cela dit pour créer une `string` temporaire, puis déplacez-le dans le paramètre de fonction `str` . L'option Copier permet à cette expression de créer directement l'objet dans `str` , plutôt que d'utiliser un déplacement temporaire.

Ceci est une optimisation utile pour les cas où un constructeur est déclaré `explicit` . Par exemple, nous aurions pu écrire ce qui précède sous la forme `func("foo")` , mais uniquement parce que `string` a un constructeur implicite qui convertit un caractère `const char*` en une `string` . Si ce constructeur était `explicit` , nous serions obligés d'utiliser un temporaire pour appeler le constructeur `explicit` . Copier elision nous évite d'avoir à faire une copie / un déplacement inutile.

Elision de valeur de retour nommée

Si vous retournez une expression lvalue d'une fonction, et cette lvalue:

- représente une variable automatique locale à cette fonction, qui sera détruite après le `return`
- la variable automatique n'est pas un paramètre de fonction
- et le type de la variable est du même type que le type de retour de la fonction

Si tout cela est le cas, alors la copie / déplacement de la lvalue peut être éluée:

```
std::string func()
{
  std::string str("foo");
  //Do stuff
  return str;
}
```

Les cas plus complexes sont éligibles, mais plus le cas est complexe, moins le compilateur aura tendance à l'éliminer:

```
std::string func()
{
  std::string ret("foo");
  if(some_condition)
  {
    return "bar";
  }
  return ret;
}
```

Le compilateur pourrait encore échapper à `ret` , mais les chances qu’elles le fassent disparaissent.

Comme indiqué précédemment, l'élision n'est pas autorisée pour les *paramètres de* valeur.

```
std::string func(std::string str)
{
  str.assign("foo");
  //Do stuff
  return str; //No elision possible
}
```

Copie de l'initialisation

Si vous utilisez une expression de valeur pour copier l'initialisation d'une variable et que cette variable a le même type que l'expression de la valeur, la copie peut être supprimée.

```
std::string str = std::string("foo");
```

L'initialisation de la copie transforme effectivement cela en `std::string str("foo");` (il y a des différences mineures).

Cela fonctionne également avec les valeurs de retour:

```
std::string func()
{
  return std::string("foo");
}

std::string str = func();
```

Sans élision de copie, cela provoquerait 2 appels au constructeur de déplacement de `std::string` . L'option Copier permet d'appeler le constructeur de déplacement 1 fois ou zéro, et la plupart des compilateurs opteront pour ce dernier.

Lire Copier Elision en ligne: https://riptutorial.com/fr/cplusplus/topic/2489/copier-elision

Chapitre 23: Copier vs assignation

Syntaxe

- **Constructeur de copie**
- MyClass (const MyClass & other);
- MyClass (MyClass & other);
- MyClass (volatil const MyClass et autres);
- MyClass (volatile MyClass et autres);
- **Constructeur d'affectation**
- MyClass & operator = (const MyClass & rhs);
- MyClass & operator = (MyClass & rhs);
- MyClass & operator = (MyClass rhs);
- const MyClass & operator = (const MyClass & rhs);
- const MyClass & operator = (MyClass & rhs);
- const MyClass & operator = (MyClass rhs);
- Opérateur MyClass = (const MyClass & rhs);
- Opérateur MyClass = (MyClass & rhs);
- MyClass operator = (MyClass rhs);

Paramètres

rhs	Côté droit de l'égalité pour les constructeurs de copie et d'affectation. Par exemple le constructeur d'affectation: Opérateur MyClass = (MyClass & rhs);
Placeholder	Placeholder

Remarques

Autres bonnes ressources pour des recherches ultérieures:

Quelle est la différence entre l'opérateur d'affectation et le constructeur de copie?

opérateur d'affectation vs constructeur de copie C ++

GeeksForGeeks

Articles C ++

Examples

Opérateur d'assignation

L'opérateur d'affectation est lorsque vous remplacez les données par un objet déjà existant (précédemment initialisé) par d'autres données d'objet. Prenons ceci comme exemple:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
            data = rhs.data;
            return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2(42);
   foo = foo2; // Assignment Operator Called
   cout << foo.data << endl; //Prints 42
}
```

Vous pouvez voir ici que j'appelle l'opérateur d'assignation lorsque j'ai déjà initialisé l'objet `foo` . Ensuite, `foo2` à `foo` . Toutes les modifications à afficher lorsque vous appelez cet opérateur de signe égal sont définies dans votre fonction `operator=` . Vous pouvez voir une sortie exécutable ici: http://cpp.sh/3qtbm

Constructeur de copie

Le constructeur de copie, par contre, est l'opposé complet du constructeur d'affectation. Cette fois, il est utilisé pour initialiser un objet déjà inexistant (ou non précédemment initialisé). Cela signifie qu'il copie toutes les données de l'objet auquel vous les affectez, sans pour autant initialiser l'objet sur lequel il est copié. Maintenant, regardons le même code que précédemment, mais modifions le constructeur d'affectation pour qu'il devienne un constructeur de copie:

```
// Copy Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;
```

```
class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2 = foo; // Copy Constructor called
   cout << foo2.data << endl;
}
```

Vous pouvez voir ici `Foo foo2 = foo;` dans la fonction principale, j'attribue immédiatement l'objet avant de l'initialiser, ce qui signifie que c'est un constructeur de copie. Et notez que je n'ai pas eu besoin de passer le paramètre int pour l'objet `foo2` car j'ai automatiquement tiré les données précédentes de l'objet foo. Voici un exemple de sortie: http://cpp.sh/5iu7

Constructeur de constructeur Vs Assignment Construct

Ok, nous avons brièvement regardé ce que le constructeur de copie et le constructeur d'affectation sont au-dessus et nous avons donné des exemples de chacun maintenant, voyons les deux dans le même code. Ce code sera similaire à ci-dessus deux. Prenons ceci:

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
```

```
        return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) / Normal Constructor called
   Foo foo2 = foo; //Copy Constructor Called
   cout << foo2.data << endl;

   Foo foo3(42);
   foo3=foo; //Assignment Constructor Called
   cout << foo3.data << endl;
}
```

Sortie:

```
2
2
```

Ici, vous pouvez voir que nous appelons d'abord le constructeur de copie en exécutant la ligne `Foo foo2 = foo;` . Puisque nous ne l'avons pas initialisé auparavant. Et ensuite nous appelons l'opérateur d'affectation sur foo3 car il était déjà initialisé `foo3=foo` ;

Lire Copier vs assignation en ligne: https://riptutorial.com/fr/cplusplus/topic/7158/copier-vs-assignation

Chapitre 24: Correct Correct

Syntaxe

- classe ClassOne {public: bool non_modifying_member_function () const {/ * ... * /}};
- int ClassTwo :: non_modifying_member_function () const {/ * ... * /}
- annuler ClassTwo :: modifying_member_function () {/ * ... * /}
- char non_param_modding_func (const ClassOne & one, const ClassTwo * deux) {/ * ... * /}
- float parameter_modifying_function (ClassTwo & one, ClassOne * two) {/ * ... * /}
- short ClassThree :: non_modding_non_param_modding_f (const ClassOne &) const {/ * ... * /}

Remarques

`const` correction de `const` est un outil de dépannage très utile, car elle permet au programmeur de déterminer rapidement quelles fonctions peuvent modifier le code par inadvertance. Il empêche également les erreurs involontaires, telles que celles présentées dans les `Const Correct Function Parameters` , de se compiler correctement et de passer inaperçues.

Il est beaucoup plus facile de concevoir une classe pour `const` exactitude, que d'ajouter plus tard `const` correct à une classe préexistante. Si possible, concevoir une classe qui *peut* être `const` correcte pour qu'il *soit* `const` correct, pour vous sauver et d' autres les tracas de modifier plus tard.

Notez que cela peut aussi être appliqué à la correction de la `volatile` si nécessaire, avec les mêmes règles que pour la correction des `const` , mais cela est beaucoup moins utilisé.

Réfractions:

ISO_CPP

Vendez-moi sur const correct

Tutoriel C ++

Examples

Les bases

`const` *exactitude* est la pratique de la conception du code de sorte que seul code qui a *besoin* de modifier une instance est en *mesure* de modifier une instance (c. -à- accès en écriture), et inversement, que tout code qui n'a pas besoin de modifier une instance est incapable de le faire so (c.-à-d. seulement accès en lecture). Cela empêche la modification involontaire de l'instance, ce qui réduit le risque d'erreur du code et indique si le code est destiné à modifier l'état de l'instance ou non. Cela permet également de traiter les instances en tant que `const` quand elles n'ont pas besoin d'être modifiées ou définies comme `const` si elles n'ont pas besoin d'être

modifiées après l'initialisation, sans perdre aucune fonctionnalité.

Ceci est fait en donnant aux fonctions membres des qualificateurs de CV `const` , et en faisant des paramètres de pointeur / référence `const` , sauf dans le cas où ils ont besoin d'un accès en écriture.

```
class ConstCorrectClass {
    int x;

  public:
    int getX() const { return x; } // Function is const: Doesn't modify instance.
    void setX(int i) { x = i; }    // Not const: Modifies instance.
};

// Parameter is const: Doesn't modify parameter.
int const_correct_reader(const ConstCorrectClass& c) {
    return c.getX();
}

// Parameter isn't const: Modifies parameter.
void const_correct_writer(ConstCorrectClass& c) {
    c.setX(42);
}

const ConstCorrectClass invariant; // Instance is const: Can't be modified.
ConstCorrectClass         variant; // Instance isn't const: Can be modified.

// ...

const_correct_reader(invariant); // Good.   Calling non-modifying function on const instance.
const_correct_reader(variant);   // Good.   Calling non-modifying function on modifiable
instance.

const_correct_writer(variant);   // Good.   Calling modifying function on modifiable instance.
const_correct_writer(invariant); // Error.  Calling modifying function on const instance.
```

En raison de la nature de la correction de const, cela commence par les fonctions de membre de la classe et fonctionne vers l'extérieur; Si vous essayez d'appeler une fonction membre non `const` partir d'une instance `const` ou d'une instance non `const` traitée comme `const` , le compilateur vous donnera une erreur à propos de la perte des qualificatifs cv.

Const Correct Design de classe

Dans une classe `const` -correct, toutes les fonctions membres qui ne changent pas d'état ont `this` cv qualifiée en `const` , indiquant qu'elles ne modifient pas l'objet (à l'exception `mutable` champs `mutable` , qui peuvent être librement modifiés même dans les instances `const`); Si une fonction `const` cv-qualifications renvoie une référence, cette référence doit également être `const` . Cela leur permet d'être appelés à la fois sur des instances constantes et non-cv, car un `const T*` est capable de se lier à un `T*` ou à un `const T*` . Ceci, à son tour, permet aux fonctions de déclarer leurs paramètres passés par référence en tant que `const` quand ils n'ont pas besoin d'être modifiés, sans perdre aucune fonctionnalité.

De plus, dans une classe `const` correcte, tous les paramètres de fonction passés par référence seront `const` , comme indiqué dans les `Const Correct Function Parameters` , afin qu'ils ne puissent

être modifiés que lorsque la fonction *doit* explicitement les modifier.

Tout d'abord, regardons `this` qualificatifs cv:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField();        // Might modify.  Also exposes member as non-const reference,
                              //  allowing indirect modification.
    void setField(Field& f);  // Modifies.

    void doSomething(int i);  // Might modify.
    void doNothing();         // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; }    // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {            // Modifies.
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {}                  // Doesn't modify.


class ConstCorrectCVQ {
    Field fld;

  public:
    ConstCorrectCVQ(Field& f);     // Modifies.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(Field& f);       // Modifies.

    void doSomething(int i);       // Modifies.
    void doNothing() const;        // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const  {}

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
// getField() and doNothing() can be called on const ConstCorrectCVQ instances.
```

```
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}
```

Nous pouvons alors combiner ceci avec les `Const Correct Function Parameters` `const` , provoquant la correction complète de la classe.

```
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f);  // Modifies instance.  Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(const Field& f); // Modifies instance.  Doesn't modify parameter.

    void doSomething(int i);       // Modifies.  Doesn't modify parameter (passed by value).
    void doNothing() const;        // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

Cela peut aussi être combiné avec une surcharge basée sur la `const` , dans le cas où nous voulons un comportement si l'instance est `const` et un comportement différent si ce n'est pas le cas; une utilisation courante pour cela est les constainsers fournissant des accesseurs qui n'autorisent la modification que si le conteneur lui-même est non `const` .

```
class ConstCorrectContainer {
    int arr[5];

  public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
          int& operator[](size_t index)       { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

Ceci est couramment utilisé dans la bibliothèque standard, avec la plupart des conteneurs , à condition de prendre les surcharges `const` ness en compte.

Paramètres de fonction const constants

Dans une fonction `const` -correct, tous les paramètres passés par référence sont marqués comme `const` sauf si la fonction les modifie directement ou indirectement, empêchant le programmeur de

modifier par inadvertance quelque chose qu'ils ne voulaient pas modifier. Cela permet à la fonction de prendre les deux `const` et des instances non-cv-qualifié, et à son tour, provoque l'instance est `this` être de type `const T*` lorsqu'une fonction membre est appelée, où `T` est le type de classe.

```
struct Example {
    void func()       { std::cout << 3 << std::endl; }
    void func() const { std::cout << 5 << std::endl; }
};

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5
```

Alors que les effets de cette situation sont moins immédiatement visibles que celles de `const` conception de classe correcte (dans ce `const` fonctions et `const` classes de provoquera des erreurs de compilation, alors que `const` classes -correct et `const` fonctions compileront correctement), `const` correcte les fonctions attraperont beaucoup d'erreurs que des fonctions `const` fausses laisseraient passer, comme celle ci-dessous. [Notez, cependant, que `const` la fonction provoque des erreurs de compilation si elle est adoptée une `const` par exemple quand il attend un non - `const` un.]

```
// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};

    int v_ind = h.get_index();               // Current working index for v.
    int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];

    temp -= h.poll_device();
```

```
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Oops.  Should've been accessing vals.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

// Const correct version.  Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}
```

Const correctité comme documentation

L'une des choses les plus utiles à propos de la correction de `const` est qu'elle sert de moyen de documentation du code, fournissant certaines garanties au programmeur et aux autres utilisateurs. Ces garanties sont imposées par le compilateur en raison de la `const` , avec un manque de `const` à son tour, indiquant que le code ne les fournit pas.

`const` CV-Qualified Member Fonctions:

- Toute fonction membre qui est `const` peut être supposée avoir l'intention de lire l'instance et:
    - Ne modifiera pas l'état logique de l'instance sur laquelle ils sont appelés. Par conséquent, ils ne doivent modifier aucune variable membre de l'instance sur laquelle ils sont appelés, à l'exception `mutable` variables `mutable` .
    - Ne doit appeler aucune *autre* fonction susceptible de modifier des variables membres de l'instance, à l'exception `mutable` variables `mutable` .
- À l'inverse, toute fonction membre qui n'est pas `const` peut être supposée avoir l'intention de modifier l'instance et:
    - Peut ou non modifier l'état logique.
    - Peut ou non appeler d'autres fonctions qui modifient l'état logique.

Cela peut être utilisé pour faire des suppositions sur l'état de l'objet après l'appel d'une fonction

membre donnée, même sans voir la définition de cette fonction:

```
// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

  public:
    // Constructor clearly changes logical state.  No assumptions necessary.
    ConstMemberFunctions(int v = 0);

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call squared_calc() or bad_func().
    int calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or bad_func().
    int squared_calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or squared_calc().
    void bad_func() const;

    // We can assume this function changes logical state, and may or may not call
    //  calc(), squared_calc(), or bad_func().
    void set_val(int v);
};
```

En raison des règles `const` , ces hypothèses seront en fait appliquées par le compilateur.

```
// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0*/)
  : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}

// Our assumption was correct.
```

```
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}
```

`const` Paramètres de fonction:

- Toute fonction avec un ou plusieurs paramètres qui sont `const` peut être supposée avoir l'intention de lire ces paramètres et:
    - Ne pas modifier ces paramètres, ni appeler les fonctions membres qui les modifieraient.
    - Ne doit pas transmettre ces paramètres à une *autre* fonction qui les modifierait et / ou appelerait les fonctions membres qui les modifieraient.
- Inversement, toute fonction avec un ou plusieurs paramètres qui ne sont pas `const` peut être supposée avoir l'intention de modifier ces paramètres et:
    - Peut ou non modifier ces paramètres, ou appeler des fonctions membres qui les modifieraient.
    - Peut ou non transmettre ces paramètres à d'autres fonctions qui les modifieraient et / ou appeleraient les fonctions membres qui les modifieraient.

Cela peut être utilisé pour faire des hypothèses sur l'état des paramètres après avoir été transmis à une fonction donnée, même sans voir la définition de cette fonction.

```
// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
//  to any of these functions.  If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

// We can assume that:
  // l is not modified, and l.set_val() won't be called.
  // l may or may not be passed to const_function_parameter().
  // r is modified, and/or r.set_val() may be called.
  // r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void bad_parameter(const ConstMemberFunctions& c);
```

En raison des règles `const` , ces hypothèses seront en fait appliquées par le compilateur.

```
// function_parameter.cpp

// Our assumption was correct.
```

```
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
//  it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers in c.set_val().
void bad_parameter(const ConstMemberFunctions& c) {
    c.set_val(18);
}
```

Bien qu'il *soit* possible de contourner la `const` constants et, par extension, d'interrompre ces garanties, cela doit être fait intentionnellement par le programmeur (tout comme rompre l'encapsulation avec `Machiavelli` , ci-dessus) et entraîner un comportement indéfini.

```
class DealBreaker : public ConstMemberFunctions {
  public:
    DealBreaker(int v = 0);

    // A foreboding name, but it's const...
    void no_guarantees() const;
}

DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}

// Our assumption was incorrect.
// const_cast removes const-ness, making the compiler think we know what we're doing.
void DealBreaker::no_guarantees() const {
    const_cast<DealBreaker*>(this)->set_val(823);
}

// ...
```

```
const DealBreaker d(50);
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

Toutefois, en raison de ce qui nécessite le programmeur de *dire* très précisément le compilateur qu'ils ont l' intention d'ignorer `const` ness, et d' être incompatibles entre compilateurs, il est généralement prudent de supposer que `const` s'abstenir de le faire , sauf indication contraire code correct.

Lire Correct Correct en ligne: https://riptutorial.com/fr/cplusplus/topic/7217/correct-correct

Chapitre 25: Date et heure en utilisant entête

Examples

Temps de mesure en utilisant

`system_clock` peut être utilisé pour mesurer le temps écoulé pendant une partie de l'exécution d'un programme.

c ++ 11

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

Dans cet exemple, `sleep_for` été utilisé pour rendre le thread actif `sleep_for` pendant une période mesurée dans `std::chrono::seconds` , mais le code entre accolades peut être tout appel de fonction qui prend un certain temps à exécuter.

Trouver le nombre de jours entre deux dates

Cet exemple montre comment trouver le nombre de jours entre deux dates. Une date est spécifiée par année / mois / jour du mois et heure / minute / seconde.

Le programme calcule le nombre de jours en années depuis 2000.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/***
 * Creates a std::tm structure from raw date.
 *
 * \param year (must be 1900 or greater)
 * \param month months since January - [1, 12]
 * \param day day of the month - [1, 31]
 * \param minutes minutes after the hour - [0, 59]
 * \param seconds seconds after the minute - [0, 61](until C++11) / [0, 60] (since C++11)
 *
```

```
 * Based on http://en.cppreference.com/w/cpp/chrono/c/tm
 */
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
    tm_ret.tm_mon = month - 1;
    tm_ret.tm_year = year - 1900;

    return tm_ret;
}

int get_days_in_year(int year) {

    using namespace std;
    using namespace std::chrono;

    // We want results to be in days
    typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

    // Create start time span
    std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
    auto tms = system_clock::from_time_t(std::mktime(&tm_start));

    // Create end time span
        std::tm tm_end =   CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
    auto tme = system_clock::from_time_t(std::mktime(&tm_end));

    // Calculate time duration between those two dates
    auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

    return diff_in_days.count();
}

int main()
{
    for ( int year = 2000; year <= 2016; ++year )
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}
```

Lire Date et heure en utilisant entête en ligne: https://riptutorial.com/fr/cplusplus/topic/3936/date-et-heure-en-utilisant--chrono--entete

Chapitre 26: decltype

Introduction

Le mot clé `decltype` peut être utilisé pour obtenir le type d'une variable, d'une fonction ou d'une expression.

Examples

Exemple de base

Cet exemple illustre simplement comment ce mot clé peut être utilisé.

```
int a = 10;

// Assume that type of variable 'a' is not known here, or it may
// be changed by programmer (from int to long long, for example).
// Hence we declare another variable, 'b' of the same type using
// decltype keyword.
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

Si, par exemple, quelqu'un change, tapez «a» pour:

```
float a=99.0f;
```

Le type de variable `b` devient alors automatiquement `float` .

Un autre exemple

Disons que nous avons un vecteur:

```
std::vector<int> intVector;
```

Et nous voulons déclarer un itérateur pour ce vecteur. Une idée évidente est d'utiliser l' `auto` . Cependant, il peut être nécessaire de simplement déclarer une variable d'itérateur (et de ne rien lui attribuer). Nous ferions:

```
vector<int>::iterator iter;
```

Cependant, avec `decltype` cela devient facile et moins sujet aux erreurs (si le type de changements `intVector`).

```
decltype(intVector)::iterator iter;
```

Alternativement:

```
decltype(intVector.begin()) iter;
```

Dans le deuxième exemple, le type de retour de `begin` est utilisé pour déterminer le type réel, qui est le `vector<int>::iterator` .

Si nous avons besoin d'un const_iterator, il suffit d'utiliser `cbegin` :

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

Lire decltype en ligne: https://riptutorial.com/fr/cplusplus/topic/9930/decltype

Chapitre 27: déduction de type

Remarques

En novembre 2014, le Comité de normalisation C ++ a adopté la proposition N3922, qui élimine la règle de déduction de type spécial pour les initialiseurs automatiques et couplés à l'aide de la syntaxe d'initialisation directe. Cela ne fait pas partie du standard C ++ mais a été implémenté par certains compilateurs.

Examples

Déduction du paramètre de modèle pour les constructeurs

Avant C ++ 17, la déduction de modèle ne peut pas déduire le type de classe pour vous dans un constructeur. Il doit être explicitement spécifié. Parfois, cependant, ces types peuvent être très encombrants ou (dans le cas de lambdas) impossibles à nommer, nous avons donc eu une prolifération de types de fabriques (comme `make_pair()` , `make_tuple()` , `back_inserter()` , etc.).

C ++ 17

Ce n'est plus necessaire:

```
std::pair p(2, 4.5);     // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
    std::back_insert_iterator(vi2)); // constructs a back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Les constructeurs sont supposés déduire les paramètres du modèle de classe, mais dans certains cas, cela est insuffisant et nous pouvons fournir des guides de déduction explicites:

```
template <class Iter>
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Déduction de type de modèle

Syntaxe générique du modèle

```
template<typename T>
void f(ParamType param);

f(expr);
```

Cas 1: `ParamType` est une référence ou un pointeur, mais pas une référence universelle ou directe.

Dans ce cas, la déduction de type fonctionne de cette façon. Le compilateur ignore la partie de référence si elle existe dans `expr` . Le compilateur alors Patronniers matchs `expr` de type s » contre `ParamType` à DETERMINATION `T` .

```
template<typename T>
void f(T& param);       //param is a reference

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // T is int, param's type is int&
f(cx);                  // T is const int, param's type is const int&
f(rx);                  // T is const int, param's type is const int&
```

Cas 2: `ParamType` est une référence universelle ou une référence directe. Dans ce cas, la déduction de type est la même que dans le cas 1 si `expr` est une valeur. Si `expr` est une lvalue, `T` et `ParamType` sont tous deux des références lvalue.

```
template<typename T>
void f(T&& param);      // param is a universal reference

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // x is lvalue, so T is int&, param's type is also int&
f(cx);                  // cx is lvalue, so T is const int&, param's type is also const int&
f(rx);                  // rx is lvalue, so T is const int&, param's type is also const int&
f(27);                  // 27 is rvalue, so T is int, param's type is therefore int&&
```

Cas 3: `ParamType` est ni un pointeur ni une référence. Si `expr` est une référence, la partie de référence est ignorée. Si `expr` est const, elle est également ignorée. S'il est volatile, il est également ignoré lors de la déduction du type de T.

```
template<typename T>
void f(T param);        // param is now passed by value

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // T's and param's types are both int
f(cx);                  // T's and param's types are again both int
f(rx);                  // T's and param's types are still both int
```

Déduction automatique du type

C ++ 11

La déduction de type à l'aide du mot `auto` clé `auto` presque identique à la déduction de type de modèle. Voici quelques exemples:

```
auto x = 27;           // (x is neither a pointer nor a reference), x's type is int
const auto cx = x;     // (cx is neither a pointer nor a reference), cs's type is const int
const auto& rx = x;    // (rx is a non-universal reference), rx's type is a reference to a
const int

auto&& uref1 = x;      // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx;     // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27;     // 27 is an int and rvalue, so uref3's type is int&&
```

Les différences sont décrites ci-dessous:

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // type is int, value is 27
auto x3 = { 27 };      // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 };         // type is std::initializer_list<int>, value is { 27 }
                       // in some compilers type may be deduced as an int with a
                       // value of 27. See remarks for more information.
auto x5 = { 1, 2.0 }   // error! can't deduce T for std::initializer_list<t>
```

Comme vous pouvez le voir si vous utilisez des initialiseurs à contreventement, auto est forcé de créer une variable de type `std::initializer_list<T>` . S'il ne peut pas déduire le de `T` , le code est rejeté.

Lorsque `auto` est utilisé comme type de retour d'une fonction, il spécifie que la fonction a un type de retour de fin .

```
auto f() -> int {
    return 42;
}
```

C ++ 14

C ++ 14 permet, en plus des utilisations d'auto autorisées dans C ++ 11, les suivantes:

1. Lorsqu'il est utilisé comme type de retour d'une fonction sans type de retour final, spécifie que le type de retour de la fonction doit être déduit des instructions de retour dans le corps de la fonction, le cas échéant.

   ```
   // f returns int:
   auto f() { return 42; }
   // g returns void:
   auto g() { std::cout << "hello, world!\n"; }
   ```

2. Lorsqu'il est utilisé dans le type de paramètre d'un lambda, définit le lambda comme étant un lambda générique .

   ```
   auto triple = [](auto x) { return 3*x; };
   const auto x = triple(42); // x is a const int with value 126
   ```

La forme spéciale `decltype(auto)` déduit un type en utilisant les règles de déduction de type de `decltype` plutôt que celles de `auto` .

```
int* p = new int(42);
auto x = *p;           // x has type int
decltype(auto) y = *p; // y is a reference to *p
```

En C ++ 03 et versions antérieures, le mot `auto` clé `auto` avait une signification complètement différente en tant que spécificateur de classe de stockage hérité de C.

Lire déduction de type en ligne: https://riptutorial.com/fr/cplusplus/topic/7863/deduction-de-type

Chapitre 28: Déplacer la sémantique

Examples

Déplacer la sémantique

La sémantique de déplacement est un moyen de déplacer un objet vers un autre en C ++. Pour cela, nous viderons l'ancien objet et placerons tout ce qu'il contenait dans le nouvel objet.

Pour cela, nous devons comprendre ce qu'est une référence de valeur. Une référence de valeur (`T&&` où T est le type d'objet) n'est pas très différente d'une référence normale (`T&` , maintenant appelée références lvalue). Mais ils agissent comme 2 types différents, et nous pouvons donc créer des constructeurs ou des fonctions qui prennent un type ou un autre, ce qui sera nécessaire pour traiter la sémantique du mouvement.

La raison pour laquelle nous avons besoin de deux types différents est de spécifier deux comportements différents. Les constructeurs de référence Lvalue sont liés à la copie, tandis que les constructeurs de référence Rvalue sont liés au déplacement.

Pour déplacer un objet, nous utiliserons `std::move(obj)` . Cette fonction renvoie une référence de valeur à l'objet, afin que nous puissions voler les données de cet objet dans une nouvelle. Il y a plusieurs façons de faire cela, qui sont discutées ci-dessous.

Il est important de noter que l'utilisation de `std::move` crée uniquement une référence de valeur. En d'autres termes, l'instruction `std::move(obj)` ne modifie pas le contenu de obj, alors que `auto obj2 = std::move(obj)` (éventuellement) le fait.

Déplacer constructeur

Disons que nous avons cet extrait de code.

```
class A {
public:
    int a;
    int b;

    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

Pour créer un constructeur de copie, c'est-à-dire créer une fonction qui copie un objet et en créer un nouveau, nous choisirions normalement la syntaxe ci-dessus, nous aurions un constructeur pour A qui prend une référence à un autre objet de type A, et nous copierions l'objet manuellement dans la méthode.

Alternativement, nous aurions pu écrire `A(const A &) = default;` qui copie automatiquement tous

les membres, en utilisant son constructeur de copie.

Pour créer un constructeur de déplacement, nous allons prendre une référence de valeur plutôt qu'une référence de lvalue, comme ici.

```
class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};
```

Veuillez noter que nous avons mis les anciennes valeurs à `zero` . Le constructeur de déplacement par défaut (`Wallet(Wallet&&) = default;`) copie la valeur de `nrOfDollars` , car il s'agit d'un POD.

Comme la sémantique de mouvement est conçue pour permettre un état de «vol» à partir de l'instance d'origine, il est important de considérer comment l'instance d'origine devrait ressembler après ce vol. Dans ce cas, si nous ne modifions pas la valeur à zéro, nous aurions doublé le montant en dollars.

```
Wallet a;
a.nrOfDollars = 1;
Wallet b (std::move(a)); //calling B(B&& other);
std::cout << a.nrOfDollars << std::endl; //0
std::cout << b.nrOfDollars << std::endl; //1
```

Nous avons donc construit un objet à partir d'un ancien.

Bien que ce qui précède soit un exemple simple, il montre ce que le constructeur de déplacement est censé faire. Cela devient plus utile dans les cas plus complexes, par exemple lorsque la gestion des ressources est impliquée.

```
    // Manages operations involving a specified type.
    // Owns a helper on the heap, and one in its memory (presumably on the stack).
    // Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.
    template<typename T,
             template<typename> typename HeapHelper,
             template<typename> typename StackHelper>
    class OperationsManager {
        using MyType = OperationsManager<T, HeapHelper, StackHelper>;

        HeapHelper<T>* h_helper;
        StackHelper<T> s_helper;
        // ...

      public:
        // Default constructor & Rule of Five.
        OperationsManager() : h_helper(new HeapHelper<T>) {}
        OperationsManager(const MyType& other)
```

```
          : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}
        MyType& operator=(MyType copy) {
            swap(*this, copy);
            return *this;
        }
        ~OperationsManager() {
            if (h_helper) { delete h_helper; }
        }

        // Move constructor (without swap()).
        // Takes other's HeapHelper<T>*.
        // Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move
constructor.
        // Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
        //  new helper when it's destroyed.
        OperationsManager(MyType&& other) noexcept
          : h_helper(other.h_helper),
            s_helper(std::move(other.s_helper)) {
            other.h_helper = nullptr;
        }

        // Move constructor (with swap()).
        // Places our members in the condition we want other's to be in, then switches members
        //  with other.
        // OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
        //     swap(*this, other);
        // }

        // Copy/move helper.
        friend void swap(MyType& left, MyType& right) noexcept {
            std::swap(left.h_helper, right.h_helper);
            std::swap(left.s_helper, right.s_helper);
        }
    };
```

Déplacer la cession

De la même façon que nous pouvons assigner une valeur à un objet avec une référence lvalue, en la copiant, nous pouvons également déplacer les valeurs d'un objet vers un autre sans en construire un nouveau. Nous appelons cette affectation de déplacement. Nous déplaçons les valeurs d'un objet vers un autre objet existant.

Pour cela, nous devrons surcharger `operator =` , pas pour qu'il prenne une référence lvalue, comme dans une affectation de copie, mais pour qu'il prenne une référence de valeur.

```
class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};
```

C'est la syntaxe typique pour définir l'affectation de déplacement. Nous surchargeons l' `operator =` afin que nous puissions lui fournir une référence de valeur et qu'il puisse l'assigner à un autre

objet.

```
A a;
a.a = 1;
A b;
b = std::move(a); //calling A& operator= (A&& other)
std::cout << a.a << std::endl; //0
std::cout << b.a << std::endl; //1
```

Ainsi, nous pouvons déplacer affecter un objet à un autre.

Utiliser std :: move pour réduire la complexité de O (n²) à O (n)

C ++ 11 a introduit le support du langage de base et de la bibliothèque standard pour **déplacer** un objet. L'idée est que lorsqu'un objet *o* est temporaire et veut une copie logique, sa sécurité à seulement chaparder *o* ressources de, comme un tampon alloué dynamiquement, laissant *o* logiquement vide mais toujours destructible et copiable.

Le support linguistique de base est principalement

- le type de **référence rvalue** builder `&&` , par exemple `std::string&&` est une référence rvalue à un `std::string` , indiquant que l'objet référencé est un temporaire dont les ressources peuvent simplement être volées (c'est-à-dire déplacées)
- support spécial pour un **constructeur de déplacement** `T( T&& )` , qui est supposé déplacer efficacement les ressources de l'objet spécifié, au lieu de copier réellement ces ressources, et
- support spécial pour un **opérateur d'** `auto operator=(T&&) -> T&` **attribution de mouvement** `auto operator=(T&&) -> T&` , qui est également supposé se déplacer de la source.

Le support standard de la bibliothèque est principalement le modèle de fonction `std::move` de l'en-tête `<utility>` . Cette fonction produit une référence de valeur à l'objet spécifié, indiquant qu'elle peut être déplacée, comme s'il s'agissait d'un objet temporaire.

Pour un conteneur, la copie réelle est généralement de complexité O (*n*), où *n* est le nombre d'éléments dans le conteneur, tandis que le déplacement est O (1), temps constant. Et pour un algorithme qui logiquement copies contenant *n* fois, cela peut réduire la complexité de l'O habituellement peu pratique *(n* ²) à seulement O linéaire *(n).*

Dans son article «Containers That Never Change» du Dr. Dobbs Journal du 19 septembre 2013 , Andrew Koenig a présenté un exemple intéressant d'inefficacité algorithmique lors de l'utilisation d'un style de programmation où les variables sont immuables après l'initialisation. Avec ce style, les boucles sont généralement exprimées en utilisant la récursivité. Et pour certains algorithmes tels que la génération d'une séquence Collatz, la récursivité nécessite de copier logiquement un conteneur:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
// "Containers That Never Change" September 19, 2013, available at
```

```
// <url: http://www.drdobbs.com/cpp/containters-that-never-change/240161543>

// Includes here, e.g. <vector>

namespace my {
    template< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        return result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            return result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            return collatz_aux( n/2, new_result );
        }
        else
        {
            return collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        return collatz_aux( n, Vector_<int>() );
    }
}  // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}
```

Sortie:

```
42 21 64 32 16 8 4 2
```

Le nombre d'opérations de copie d'éléments dues à la copie des vecteurs est ici approximativement O (n^2), puisque c'est la somme 1 + 2 + 3 + ... *n* .

En chiffres concrets, avec les compilateurs g ++ et Visual C ++, l'invocation de `collatz(42)` ci-dessus a entraîné une séquence Collatz de 8 éléments et 36 opérations de copie d'éléments (8 * `collatz(42)` = 28, plus quelques).

Toutes ces opérations de copie d'éléments peuvent être supprimées en déplaçant simplement des vecteurs dont les valeurs ne sont plus nécessaires. Pour ce faire, il est nécessaire de supprimer `const` et reference pour les arguments de type vectoriel, en passant les vecteurs *par valeur* . Les retours de fonctions sont déjà automatiquement optimisés. Pour les appels où les vecteurs sont passés et ne sont plus utilisés dans la fonction, appliquez simplement `std::move` pour *déplacer* ces tampons plutôt que de les copier:

```
using std::move;

auto concat( Vector_<int> v, int const x )
    -> Vector_<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-
Wpessimizing-move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector_<int> result )
    -> Vector_<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result;      // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector_<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector_<int>() );
}
```

Ici, avec les compilateurs g ++ et Visual C ++, le nombre d'opérations de copie d'éléments dues aux invocations du constructeur de la copie vectorielle était exactement 0.

L'algorithme est nécessairement encore O *(n)* de la longueur de la séquence Collatz produit, mais cela est une amélioration tout à fait spectaculaire: O (n^2) → O *(n).*

Avec certains langages, on peut peut-être utiliser le mouvement et exprimer et imposer l'immuabilité d'une variable *entre son initialisation et son déplacement final* , après quoi toute utilisation de cette variable devrait être une erreur. Hélas, à partir de C ++ 14 C ++ ne le supporte pas. Pour le code sans boucle, la non utilisation après le déplacement peut être imposée via une nouvelle déclaration du nom correspondant en tant que `struct` incomplète, comme avec le `struct result;` ci-dessus, mais c'est moche et peu susceptible d'être compris par d'autres programmeurs; les diagnostics peuvent aussi être trompeurs.

En résumé, le support du langage C ++ et de la bibliothèque pour le déplacement permet des améliorations drastiques de la complexité des algorithmes, mais en raison de l'incomplétude du support, au détriment des garanties d'exactitude du code et de clarté que `const` peut fournir.

Pour être complet, la classe de vecteur instrumentée utilisée pour mesurer le nombre d'opérations de copie d'éléments dues à des invocations de constructeur de copie:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

    vector<Item>    items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

    Copy_tracking_vector(){}

    Copy_tracking_vector( Copy_tracking_vector const& other )
        : items_( other.items_ )
    { n_copy_ops() += items_.size(); }

    Copy_tracking_vector( Copy_tracking_vector&& other )
        : items_( move( other.items_ ) )
    {}
};
```

Utilisation de la sémantique de déplacement sur les conteneurs

Vous pouvez déplacer un conteneur au lieu de le copier:

```
void print(const std::vector<int>& vec) {
    for (auto&& val : vec) {
        std::cout << val << ", ";
    }
    std::cout << std::endl;
```

```
}

int main() {
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector
    std::vector<int> vec1{1, 2, 3, 4};
    std::vector<int> vec2;

    // The following line will print 1, 2, 3, 4
    print(vec1);

    // The following line will print a new line
    print(vec2);

    // The vector vec2 is assigned with move assignment.
    // This will "steal" the value of vec1 without copying it.
    vec2 = std::move(vec1);

    // Here the vec1 object is in an indeterminate state, but still valid.
    // The object vec1 is not destroyed,
    // but there's is no guarantees about what it contains.

    // The following line will print 1, 2, 3, 4
    print(vec2);
}
```

Réutiliser un objet déplacé

Vous pouvez réutiliser un objet déplacé:

```
void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector as gained a determinate value, we can use it normally.
    vec.push_back(42);

    // Send the vector by move again.
    consumingFunction(std::move(vec));
}
```

Lire Déplacer la sémantique en ligne: https://riptutorial.com/fr/cplusplus/topic/2129/deplacer-la-semantique

Chapitre 29: Des exceptions

Examples

Catching exceptions

Un bloc `try/catch` est utilisé pour intercepter des exceptions. Le code dans la section `try` est le code qui peut générer une exception et le code dans la ou `catch` clauses `catch` gère l'exception.

```
#include <iostream>
#include <string>
#include <stdexcept>

int main() {
  std::string str("foo");

  try {
      str.at(10); // access element, may throw std::out_of_range
  } catch (const std::out_of_range& e) {
      // what() is inherited from std::exception and contains an explanatory message
      std::cout << e.what();
  }
}
```

Plusieurs clauses `catch` peuvent être utilisées pour gérer plusieurs types d'exceptions. Si plusieurs clauses `catch` sont présentes, le mécanisme de gestion des exceptions tente de les faire correspondre **dans l'ordre** d'apparition dans le code:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Les classes d'exception dérivées d'une classe de base commune peuvent être interceptées avec une seule clause `catch` pour la classe de base commune. L'exemple ci-dessus peut remplacer les deux clauses `catch` pour `std::length_error` et `std::out_of_range` avec une clause unique pour `std:exception` :

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Comme les clauses `catch` sont essayées dans l'ordre, veillez à écrire plus de clauses catch spécifiques, sinon votre code de gestion des exceptions risque de ne jamais être appelé:

```
try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
    /* This block of code will never execute, because std::runtime_error inherits
       from std::exception, and all exceptions of type std::exception were already
       caught by the previous catch clause. */
}
```

Une autre possibilité est le gestionnaire fourre-tout, qui attrape tout objet jeté:

```
try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}
```

Renvoyer (propager) une exception

Parfois, vous voulez faire quelque chose avec l'exception que vous avez capturée (comme l'écriture pour vous connecter ou imprimer un avertissement) et la laisser remonter jusqu'au périmètre supérieur à traiter. Pour ce faire, vous pouvez relancer toute exception que vous avez détectée:

```
try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}
```

En utilisant `throw;` sans arguments rejettera l'exception actuellement interceptée.

C ++ 11

Pour relancer un fichier `std::exception_ptr` géré, la bibliothèque standard C ++ a la fonction `rethrow_exception` qui peut être utilisée en incluant l'en-tête `<exception>` dans votre programme.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
```

```
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

Fonction Try Blocks In constructeur

La seule façon d'attraper une exception dans la liste d'initialisation:

```
struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};
```

Fonction Essayez Bloquer pour une fonction régulière

```
void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}
```

Ce qui équivaut à

```
void function_with_try_block()
{
    try
    {
```

```
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}
```

Notez que pour les constructeurs et les destructeurs, le comportement est différent car le bloc catch renvoie une exception de toute façon (celui qui est attrapé s'il n'y a pas d'autre jet dans le corps du bloc catch).

La fonction `main` est autorisée à avoir un bloc try de fonction comme toute autre fonction, mais le bloc try de la fonction `main` n'acceptera pas les exceptions qui se produisent pendant la construction d'une variable statique non locale ou la destruction de toute variable statique. Au lieu de cela, `std::terminate` est appelé.

Fonction Try Blocks In destructor

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Notez que, bien que cela soit possible, il faut faire très attention au lancement du destructeur, comme si un destructeur appelé lors du déroulement de la pile lançait une exception, `std::terminate` est appelé.

Meilleure pratique: lancer par valeur, référence par const

En général, il est conseillé de lancer par valeur (plutôt que par pointeur), mais attraper par référence (const).

```
try {
    // throw new std::runtime_error("Error!");   // Don't do this!
    // This creates an exception object
    // on the heap and would require you to catch the
    // pointer and manage the memory yourself. This can
    // cause memory leaks!

    throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
```

Une des raisons pour lesquelles la capture par référence est une bonne pratique est qu'elle élimine le besoin de reconstruire l'objet lorsqu'elle est transmise au bloc catch (ou lorsqu'elle se propage à d'autres blocs catch). La saisie par référence permet également de gérer les exceptions de manière polymorphe et d'éviter le découpage d'objets. Cependant, si vous renvoyez une exception (comme `throw e;` voir exemple ci-dessous), vous pouvez toujours obtenir un découpage d'objet car le `throw e;` L'énoncé fait une copie de l'exception, quel que soit le type déclaré:

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                     // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}
```

Si vous êtes certain que vous ne ferez rien pour modifier l'exception (comme ajouter des informations ou modifier le message), attraper par référence de référence permet au compilateur d'effectuer des optimisations et d'améliorer les performances. Mais cela peut toujours provoquer un épissage d'objet (comme dans l'exemple ci-dessus).

Avertissement: Faites attention aux exceptions non prévues dans `catch` blocs `catch` , en particulier en ce qui concerne l'allocation de mémoire ou de ressources supplémentaires. Par exemple, la construction de `logic_error` , `runtime_error` ou de leurs sous-classes peut `bad_alloc` raison de la mémoire `bad_alloc` lors de la copie de la chaîne d'exception, les flux d'E / S pouvant se produire lors de la journalisation

Exception imbriquée

C ++ 11

Lors de la gestion des exceptions, il existe un cas d'utilisation courant lorsque vous capturez une exception générique à partir d'une fonction de bas niveau (erreur de système de fichiers ou erreur

de transfert de données, par exemple) et que ne pas être effectué (par exemple, ne pas pouvoir publier une photo sur le Web). Cela permet à la gestion des exceptions de réagir à des problèmes spécifiques avec des opérations de haut niveau et permet également, en ayant uniquement un message d'erreur, au programmeur de trouver une place dans l'application où une exception s'est produite. L'inconvénient de cette solution est que le bloc d'appel des exceptions est tronqué et que l'exception d'origine est perdue. Cela oblige les développeurs à inclure manuellement le texte de l'exception d'origine dans celui qui vient d'être créé.

Les exceptions imbriquées visent à résoudre le problème en attachant une exception de bas niveau, qui décrit la cause, à une exception de haut niveau, qui décrit ce que cela signifie dans ce cas particulier.

`std::nested_exception` permet d'imbriquer des exceptions grâce à `std::throw_with_nested` :

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unkown exception\n";
    }
}

void print_current_exception_with_nested(int level =  0)
{
    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
        //Empty // End recursion
    }
```

```
}

// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch(...) {
        print_current_exception_with_nested();
    }
}
```

Sortie possible:

```
exception: run() failed
 MyException: Couldn't open nonexistent.file
  exception: basic_ios::clear
```

Si vous travaillez uniquement avec des exceptions héritées de `std::exception` , le code peut même être simplifié.

std :: uncaught_exceptions

c ++ 17

C ++ 17 introduit `int std::uncaught_exceptions()` (pour remplacer la `bool std::uncaught_exception()` limitée) pour savoir combien d'exceptions sont actuellement non capturées. Cela permet à une classe de déterminer si elle est détruite lors du déroulement ou non d’une pile.

```
#include <exception>
#include <string>
#include <iostream>

// Apply change on destruction:
// Rollback in case of exception (failure)
```

```
// Else Commit (success)
class Transaction
{
public:
    Transaction(const std::string& s) : message(s) {}
    Transaction(const Transaction&) = delete;
    Transaction& operator =(const Transaction&) = delete;
    void Commit() { std::cout << message << ": Commit\n"; }
    void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

private:
    std::string message;
    int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
public:
    ~Foo() {
        try {
            Transaction transaction("In ~Foo"); // Commit,
                                                // even if there is an uncaught exception
            //...
        } catch (const std::exception& e) {
            std::cerr << "exception/~Foo:" << e.what() << std::endl;
        }
    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
        Foo foo; // ~Foo commit its transaction.
        //...
        throw std::runtime_error("Error");
    } catch (const std::exception& e) {
        std::cerr << "exception/main:" << e.what() << std::endl;
    }
}
```

Sortie:

```
In ~Foo: Commit
In main: Rollback
exception/main:Error
```

Exception personnalisée

Vous ne devriez pas lancer de valeurs brutes en tant qu'exceptions, utilisez plutôt l'une des classes d'exception standard ou créez la vôtre.

Avoir votre propre classe d'exception héritée de `std::exception` est un bon moyen de s'y prendre. Voici une classe d'exception personnalisée qui hérite directement de `std::exception` :

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset
    std::string error_message;      ///< Error message

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
        {}

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     *  @return A pointer to a const char*. The underlying memory
     *  is in possession of the Except object. Callers must
     *  not attempt to free the memory.
     */
    virtual const char* what() const throw () {
       return error_message.c_str();
    }

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Un exemple jet catch:

```
try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
             <<"\nError number: "<<e.getErrorNumber()
             <<"\nError offset: "<<e.getErrorOffset();
}
```

Comme vous ne faites pas que lancer un message d'erreur idiot, mais aussi d'autres valeurs représentant exactement l'erreur, votre gestion des erreurs devient beaucoup plus efficace et significative.

Il y a une classe d'exception qui vous permet de gérer `std::runtime_error` messages d'erreur: `std::runtime_error`

Vous pouvez également hériter de cette classe:

```
#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
        {
            error_number = err_num;
            error_offset = err_off;

        }

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
```

```
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Notez que je n'ai pas remplacé la fonction `what()` de la classe de base (`std::runtime_error`), c'est-à-dire que nous utiliserons la version de `what()` la classe de base. Vous pouvez le remplacer si vous avez d'autres ordres du jour.

Lire Des exceptions en ligne: https://riptutorial.com/fr/cplusplus/topic/1354/des-exceptions

Chapitre 30: Disposition des types d'objet

Remarques

Voir aussi Taille des types intégraux .

Examples

Types de classes

Par «classe», nous entendons un type défini avec le mot `struct` clé `class` ou `struct` (mais pas `enum struct` ou `enum class`).

- Même une classe vide occupe au moins un octet de stockage; il s'agira donc uniquement de rembourrage. Cela garantit que si `p` pointe sur un objet d'une classe vide, alors `p + 1` est une adresse distincte et pointe vers un objet distinct. Cependant, il est possible qu'une classe vide ait une taille de 0 lorsqu'elle est utilisée comme classe de base. Voir optimisation de base vide .

```
class Empty_1 {};                               // sizeof(Empty_1)       == 1
class Empty_2 {};                               // sizeof(Empty_2)       == 1
class Derived : Empty_1 {};                     // sizeof(Derived)       == 1
class DoubleDerived : Empty_1, Empty_2 {};      // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; };                    // sizeof(Holder)        == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder)  == 2
class DerivedHolder : Empty_1 { Empty_1 e; };   // sizeof(DerivedHolder) == 2
```

- La représentation d'objet d'un type de classe contient les représentations d'objet de la classe de base et des types de membres non statiques. Par exemple, dans la classe suivante:

```
struct S {
    int x;
    char* y;
};
```

il existe une séquence consécutive de `sizeof(int)` octets `sizeof(int)` dans un objet `S` , appelée *sous* - *objet,* qui contient la valeur de `x` , et un autre sous-objet de `sizeof(char*)` octets `sizeof(char*)` contenant la valeur de `y` . Les deux ne peuvent pas être entrelacés.

- Si un type de classe a des membres et / ou des classes de base avec les types `t1, t2,...tN` , la taille doit être au minimum `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` compte tenu des points précédents . Toutefois, en fonction des exigences d' alignement des membres et des classes de base, le compilateur peut être contraint d'insérer un remplissage entre les sous-objets ou au début ou à la fin de l'objet complet.

```
struct AnInt      { int i; };
  // sizeof(AnInt)        == sizeof(int)
```

```
  // Assuming a typical 32- or 64-bit system, sizeof(AnInt)        == 4 (4).
struct TwoInts    { int i, j; };
  // sizeof(TwoInts)      >= 2 * sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(TwoInts)      == 8 (4 + 4).
struct IntAndChar { int i; char c; };
  // sizeof(IntAndChar)   >= sizeof(int) + sizeof(char)
  // Assuming a typical 32- or 64-bit system, sizeof(IntAndChar)   == 8 (4 + 1 +
padding).
struct AnIntDerived : AnInt { long long l; };
  // sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
  // Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding +
8).
```

- Si le remplissage est inséré dans un objet en raison des exigences d'alignement, la taille sera supérieure à la somme des tailles des membres et des classes de base. Avec l'alignement sur `n` octets, la taille sera généralement le plus petit multiple de `n` supérieur à la taille de tous les membres et classes de base. Chaque membre `memN` sera généralement placé à une adresse multiple de `alignof(memN)` , et `n` sera généralement le plus grand `alignof` de tous les `alignof` des membres. De ce fait, si un membre avec un `alignof` plus `alignof` est suivi par un membre avec un `alignof` plus `alignof` , il est possible que ce dernier membre ne soit pas correctement aligné s'il est placé immédiatement après le premier. Dans ce cas, un rembourrage (également appelé *élément d'alignement*) sera placé entre les deux éléments, de sorte que ce dernier membre puisse avoir son alignement souhaité. Inversement, si un membre avec un `alignof` plus `alignof` est suivi par un membre avec un `alignof` plus `alignof` , aucun remplissage ne sera généralement nécessaire. Ce processus est également appelé "emballage".
  Étant donné que les classes partageant généralement l' `alignof` de leur membre avec l' `alignof` le plus `alignof` , les classes seront généralement alignées sur l' `alignof` du plus grand type intégré qu'elles contiennent directement ou indirectement.

```
// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
  // sizeof(Char)                == 1 (sizeof(char))
struct Int  { int i; };
  // sizeof(Int)                 == 4 (sizeof(int))
struct CharInt { char c; int i; };
  // sizeof(CharInt)             == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
  // sizeof(ShortIntCharInt)     == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                   3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
  // sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                   1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
  // sizeof(ShortCharShortInt)   == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
  //                                   2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
  // sizeof(IntLLInt)            == 16 (4 (int) + 8 (long long) + 4 (int))
  // If packing isn't explicitly specified, most compilers will pack this as
  //   8-byte alignment, such that:
  // sizeof(IntLLInt)            == 24 (4 (int) + 4 (padding) + 8 (long long) +
  //                                   4 (int) + 4 (padding))
```

```
// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};
  // ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
  //                                             alignof(int) >= alignof(short)
  // sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
  //                                      2 (short) + 4 (int))
  // Note that t is placed at alignment of 2, not 4.  alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};
  // Large_1 has 4-byte alignment.
    // alignof(ShortIntCharInt) == alignof(int) == 4
    // alignof(b) == 1
    // Therefore, alignof(Large_1) == 4.
  // sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
  //                        16 (ShortIntCharInt))
struct Large_2 {
    IntLLInt illi;
    float f;
    IntLLInt jmmj;
};
  // Large_2 has 8-byte alignment.
    // alignof(IntLLInt) == alignof(long long) == 8
    // alignof(float) == 4
    // Therefore, alignof(Large_2) == 8.
  // sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))
```

C ++ 11

- Si l'alignement strict est forcé avec `alignas` , le remplissage sera utilisé pour forcer le type à respecter l'alignement spécifié, même s'il serait autrement plus petit. Par exemple, avec la définition ci-dessous, les caractères `Chars<5>` auront trois (ou peut-être plus) octets de remplissage insérés à la fin de sorte que leur taille totale soit 8. Il n'est pas possible pour une classe avec un alignement de 4 d'avoir une taille de 5 car il serait impossible de faire un tableau de cette classe, donc la taille doit être "arrondie" à un multiple de 4 en insérant des octets de remplissage.

```
// This type shall always be aligned to a multiple of 4.  Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");
```

- Si deux membres non statiques d'une classe ont le même spécificateur d'accès , celui qui apparaîtra plus tard dans l'ordre de déclaration sera garanti plus tard dans la représentation

de l'objet. Mais si deux membres non statiques ont des spécificateurs d'accès différents, leur ordre relatif dans l'objet n'est pas spécifié.

- Il n'est pas spécifié dans quel ordre les sous-objets de classe de base apparaissent dans un objet, qu'ils soient consécutifs ou non, qu'ils apparaissent avant, après ou entre les sous-objets membres.

Types arithmétiques

Types de caractères étroits

Le type de caractère `unsigned char` utilise tous les bits pour représenter un nombre binaire. Par conséquent, par exemple, si le caractère `unsigned char` longueur de 8 bits, alors les 256 modèles de bits possibles d'un objet `char` représentent les 256 valeurs différentes {0, 1, ..., 255}. Le nombre 42 est garanti pour être représenté par le modèle de bit `00101010` .

Le type de caractère `signed char` n'a pas de bits de remplissage, *c'est* -à- *dire que* si le caractère `signed char` longueur de 8 bits, il a une capacité de 8 bits pour représenter un nombre.

Notez que ces garanties ne s'appliquent pas aux types autres que les types de caractères étroits.

Types entiers

Les types entiers non signés utilisent un système binaire pur, mais peuvent contenir des bits de remplissage. Par exemple, il est possible (mais improbable) que `unsigned int` ait une longueur de 64 bits, mais soit uniquement capable de stocker des entiers compris entre 0 et $2^{32} - 1$ inclus. Les 32 autres bits seraient des bits de remplissage, auxquels il ne faut pas écrire directement.

Les types entiers signés utilisent un système binaire avec un bit de signe et éventuellement des bits de remplissage. Les valeurs qui appartiennent à la plage commune d'un type d'entier signé et au type d'entier non signé correspondant ont la même représentation. Par exemple, si le modèle de bit `0001010010101011` d'un objet `unsigned short` représente la valeur `5291` , il représente également la valeur `5291` lorsqu'il est interprété comme un objet `short` .

Il est défini par la mise en œuvre si une représentation à deux, un complément ou une magnitude de signe est utilisée, car les trois systèmes satisfont aux exigences du paragraphe précédent.

Virgule flottante

La représentation de la valeur des types à virgule flottante est définie par l'implémentation. Le plus souvent, les types `float` et `double` sont conformes à IEEE 754 et ont une longueur de 32 et 64 bits (par exemple, `float` aurait 23 bits de précision qui suivraient 8 bits d’exposant et 1 bit de signe). Cependant, la norme ne garantit rien. Les types à virgule flottante ont souvent des "représentations d'interruptions", qui provoquent des erreurs lorsqu'elles sont utilisées dans les calculs.

Tableaux

Un type de tableau n'a pas de remplissage entre ses éléments. Par conséquent, un tableau avec le type d'élément `T` n'est qu'une séquence d'objets `T` disposés en mémoire, dans l'ordre.

Un tableau multidimensionnel est un tableau de tableaux, et ce qui précède s'applique de manière récursive. Par exemple, si nous avons la déclaration

```
int a[5][3];
```

alors `a` est un tableau de 5 tableaux de 3 `int` s. Par conséquent, `a[0]` , qui consiste en trois éléments `a[0][0]` , `a[0][1]` , `a[0][2]` , est mis en mémoire avant `a[1]` , qui consiste à de `a[1][0]` , `a[1][1]` et `a[1][2]` . Ceci s'appelle ordre *majeur de rangée* .

Lire Disposition des types d'objet en ligne:
https://riptutorial.com/fr/cplusplus/topic/9329/disposition-des-types-d-objet

Chapitre 31: Entrée / sortie basique en c ++

Remarques

La bibliothèque standard `<iostream>` définit peu de flux pour l'entrée et la sortie:

```
|stream | description                      |
|-------|----------------------------------|
|cin    | standard input stream            |
|cout   | standard output stream           |
|cerr   | standard error (output) stream   |
|clog   | standard logging (output) stream |
```

Sur les quatre flux mentionnés ci-dessus, `cin` est essentiellement utilisé pour la saisie par l'utilisateur et trois autres pour la sortie des données. En général ou dans la plupart des environnements de codage, `cin` (*entrée de console* ou *entrée* standard) est clavier et `cout` (*sortie de console* ou sortie standard) est moniteur.

```
cin >> value

cin   - input stream
'>>'  - extraction operator
value - variable (destination)
```

`cin` extrait ici l'entrée saisie par l'utilisateur et alimente la valeur variable. La valeur est extraite uniquement après que l'utilisateur a appuyé sur la touche ENTER.

```
cout << "Enter a value: "

cout               - output stream
'<<'               - insertion operator
"Enter a value: " - string to be displayed
```

`cout` prend ici la chaîne à afficher et l'insère à la sortie standard ou au moniteur

Les quatre flux sont situés dans l'espace de noms standard `std` . Nous devons donc imprimer `std::stream` pour que le `stream` puisse l'utiliser.

Il y a aussi un manipulateur `std::endl` dans le code. Il ne peut être utilisé qu'avec des flux de sortie. Il insère le caractère `'\n'` de fin de ligne dans le flux et le vide. Cela provoque une production immédiate.

Examples

entrée utilisateur et sortie standard

```
#include <iostream>
```

```
int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Lire Entrée / sortie basique en c ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/10683/entree---sortie-basique-en-c-plusplus

Chapitre 32: Énumération

Examples

Déclaration de dénombrement de base

Les énumérations standard permettent aux utilisateurs de déclarer un nom utile pour un ensemble d'entiers. Les noms sont collectivement appelés énumérateurs. Une énumération et ses énumérateurs associés sont définis comme suit:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

Une énumération est un *type* , qui est distinct de tous les autres types. Dans ce cas, le nom de ce type est `myEnum` . Les objets de ce type sont censés prendre la valeur d'un énumérateur dans l'énumération.

Les énumérateurs déclarés dans l'énumération sont des valeurs constantes du type de l'énumération. Bien que les énumérateurs soient déclarés dans le type, l'opérateur d'étendue `::` n'est pas nécessaire pour accéder au nom. Le nom du premier énumérateur est donc `enumName1` .

C ++ 11

L'opérateur d'étendue peut éventuellement être utilisé pour accéder à un énumérateur dans une énumération. Donc, `enumName1` peut également être orthographié `myEnum::enumName1` .

Les énumérateurs se voient attribuer des valeurs entières commençant à 0 et augmentant de 1 pour chaque énumérateur dans une énumération. Donc, dans le cas ci-dessus, `enumName1` a la valeur 0, alors que `enumName2` a la valeur 1.

Les énumérateurs peuvent également se voir attribuer une valeur spécifique par l'utilisateur. cette valeur doit être une expression constante intégrale. Les énumérateurs dont les valeurs ne sont pas explicitement fournies auront leur valeur définie sur la valeur de l'énumérateur précédent + 1.

```
enum myEnum
{
    enumName1 = 1, // value will be 1
    enumName2 = 2, // value will be 2
    enumName3,     // value will be 3, previous value + 1
    enumName4 = 7, // value will be 7
    enumName5,     // value will be 8
    enumName6 = 5, // value will be 5, legal to go backwards
    enumName7 = 3, // value will be 3, legal to reuse numbers
    enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Enumération dans les instructions de commutateur

Les énumérateurs sont couramment utilisés pour les instructions de commutation et apparaissent donc généralement dans les machines à états. En fait, une caractéristique utile des instructions de commutation avec énumérations est que si aucune instruction par défaut n'est incluse pour le commutateur et que toutes les valeurs de l'énumération n'ont pas été utilisées, le compilateur émettra un avertissement.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
       ...
    case middle:
       ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

Itération sur un enum

Il n'y a pas d'itéré pour itérer l'énumération.

Mais il y a plusieurs façons

- pour `enum` avec uniquement des valeurs consécutives:

```
enum E {
    Begin,
    E1 = Begin,
    E2,
    // ..
    En,
    End
};

for (E e = E::Begin; e != E::End; ++e) {
    // Do job with e
}
```

C ++ 11

avec la `enum class` , `operator ++` doit être implémenté:

```
E& operator ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
```

```
    return e;
}
```

- en utilisant un conteneur en tant que `std::vector`

  ```
  enum E {
      E1 = 4,
      E2 = 8,
      // ..
      En
  };

  std::vector<E> build_all_E()
  {
      const E all[] = {E1, E2, /*..*/ En};

      return std::vector<E>(all, all + sizeof(all) / sizeof(E));
  }

  std::vector<E> all_E = build_all_E();
  ```

  et alors

  ```
  for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
      E e = *it;
      // Do job with e;
  }
  ```

C ++ 11

- ou `std::initializer_list` et une syntaxe plus simple:

  ```
  enum E {
      E1 = 4,
      E2 = 8,
      // ..
      En
  };

  constexpr std::initializer_list<E> all_E = {E1, E2, /*..*/ En};
  ```

  et alors

  ```
  for (auto e : all_E) {
      // Do job with e
  }
  ```

Énumérés

C ++ 11 introduit ce que l'on appelle les *énumérations* de *portée* . Ce sont des énumérations dont les membres doivent être qualifiés avec `enumname::membername` . Les énumérations de portée sont déclarées en utilisant la syntaxe de la `enum class` . Par exemple, pour stocker les couleurs dans un arc-en-ciel:

```
enum class rainbow {
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

Pour accéder à une couleur spécifique:

```
rainbow r = rainbow::INDIGO;
```

`enum class` es ne peuvent pas être implicitement converties en `int` s sans conversion. Donc `int x = rainbow::RED` n'est pas valide.

Les énumérations de portée vous permettent également de spécifier le *type sous* - *jacent* , qui est le type utilisé pour représenter un membre. Par défaut, il est `int` . Dans un jeu Tic-Tac-Toe, vous pouvez stocker la pièce comme

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
};
```

Comme vous pouvez le constater, `enum` s peut avoir une virgule après le dernier membre.

Énumérer la déclaration en avant en C ++ 11

Énumérations de portée:

```
...
enum class Status; // Forward declaration
Status doWork(); // Use the forward declaration
...
enum class Status { Invalid, Success, Fail };
Status doWork() // Full declaration required for implementation
{
    return Status::Success;
}
```

Énumérations non découpées:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

Un exemple détaillé de fichiers multiples peut être trouvé ici: Exemple de marchand de fruits à

l'aveugle

Lire Énumération en ligne: https://riptutorial.com/fr/cplusplus/topic/2796/enumeration

Chapitre 33: Erreurs courantes de compilation / édition de liens (GCC)

Examples

erreur: '***' n'a pas été déclaré dans cette portée

Cette erreur se produit si un objet inconnu est utilisé.

Les variables

Ne pas compiler:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Réparer:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }

    return 0;
}
```

Les fonctions

> La plupart du temps, cette erreur se produit si l'en-tête nécessaire n'est pas inclus (par exemple, en utilisant `std::cout` sans `#include <iostream>`)

Ne pas compiler:

```
#include <iostream>
```

```
int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Réparer:

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Ou:

```
#include <iostream>

void doCompile() // define the function before using it
{
    std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}
```

Note: Le compilateur interprète le code de haut en bas (simplification). Tout doit être au moins déclaré (ou défini) avant utilisation.

référence indéfinie à «***»

Cette erreur de l'éditeur de liens se produit si l'éditeur de liens ne parvient pas à trouver un symbole utilisé. La plupart du temps, cela se produit si une bibliothèque utilisée n'est pas liée.

qmake:

```
LIBS += nameOfLib
```

cmake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

appel g ++:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

On pourrait également oublier de compiler et de lier tous les fichiers `.cpp` utilisés (functionsModule.cpp définit la fonction nécessaire):

```
g++  -o binName main.o functionsModule.o
```

erreur fatale: ***: aucun fichier ou répertoire de ce type

Le compilateur ne peut pas trouver de fichier (un fichier source utilise `#include "someFile.hpp"`).

qmake:

```
INCLUDEPATH += dir/Of/File
```

cmake:

```
include_directories(dir/Of/File)
```

appel g ++:

```
g++ -o main main.cpp -Idir/Of/File
```

Lire Erreurs courantes de compilation / édition de liens (GCC) en ligne: https://riptutorial.com/fr/cplusplus/topic/4256/erreurs-courantes-de-compilation---edition-de-liens--gcc-

Chapitre 34: Espaces de noms

Introduction

Utilisé pour empêcher les collisions de noms lors de l'utilisation de plusieurs bibliothèques, un espace de noms est un préfixe déclaratif pour les fonctions, les classes, les types, etc.

Syntaxe

- *identifiant d'* espace de noms (*opt*) { *declaration-seq* }
- *identificateur d'* espace de nommage en ligne (*opt*) { *declaration-seq* } / * depuis C ++ 11 * /
- inline (*opt*) namespace *attribut-spécificateur-seq identifiant* (*opt*) { *declaration-seq* } / * depuis C ++ 17 * /
- espace de noms *enclosing-namespace-specifier* :: *identifier* { *declaration-seq* } / * depuis C ++ 17 * /
- *identifiant d'* espace de nommage = *spécificateur-espace-nom qualifié* ;
- using *namespace-name-spécificateur imbriqué (opt) namespace-name;*
- *attribute-specifier-seq* utilisant un espace de nommage *nom-nom-spécificateur* (*opt*) *nom-espace* ; / * depuis C ++ 11 * /

Remarques

L' `namespace` mot clé a trois significations différentes selon le contexte:

1. Lorsqu'il est suivi d'un nom facultatif et d'une séquence de déclarations entourée d'accolades, il définit un nouvel espace de noms ou étend un espace de noms existant avec ces déclarations. Si le nom est omis, l'espace de noms est un espace de noms sans nom .

2. Lorsqu'il est suivi d'un nom et d'un signe égal, il déclare un alias d'espace de noms .

3. Lorsqu'elle est précédée par l' `using` et le suivi d'un nom d'espace de noms, elle forme une *directive using* , qui permet aux noms dans l'espace de noms donné d'être trouvés par une recherche de noms non qualifiée (mais ne les redéclarent pas dans la portée actuelle). Une *directive using* ne peut pas exister à la portée de la classe.

`using namespace std;` est découragé. Pourquoi? Parce que l' `namespace std` est énorme! Cela signifie qu'il y a de fortes chances que les noms entrent en collision:

```
//Really bad!
using namespace std;

//Calculates p^e and outputs it to std::cout
void pow(double p, double e) { /*...*/ }

//Calls pow
pow(5.0, 2.0); //Error! There is already a pow function in namespace std with the same
signature,
```

```
            //so the call is ambiguous
```

Examples

Que sont les espaces de noms?

Un espace de noms C ++ est un ensemble d'entités C ++ (fonctions, classes, variables), dont les noms sont précédés du nom de l'espace de noms. Lors de l'écriture de code dans un espace de noms, les entités nommées appartenant à cet espace de noms n'ont pas besoin d'être précédées du nom de l'espace de noms, mais les entités extérieures doivent utiliser le nom complet. Le nom qualifié complet a le format `<namespace>::<entity>` . Exemple:

```
namespace Example
{
  const int test = 5;

  const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

Les espaces de noms sont utiles pour regrouper les définitions associées. Prenez l'analogie d'un centre commercial. Généralement, un centre commercial est divisé en plusieurs magasins, chaque magasin vendant des articles d'une catégorie spécifique. Un magasin peut vendre des produits électroniques, tandis qu'un autre magasin peut vendre des chaussures. Ces séparations logiques dans les types de magasin aident les acheteurs à trouver les éléments qu'ils recherchent. Les espaces de noms aident les programmeurs c ++, comme les acheteurs, à trouver les fonctions, les classes et les variables qu'ils recherchent en les organisant de manière logique. Exemple:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
```

```
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

Il y a un seul espace de noms prédéfini, qui est l'espace de noms global qui n'a pas de nom, mais qui peut être noté `::` . Exemple:

```
void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar();   // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}
```

Faire des espaces de noms

Créer un espace de noms est très simple:

```
//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}
```

Pour appeler la `bar` , vous devez d'abord spécifier l'espace de noms, suivi de l'opérateur de résolution de portée `::`

```
Foo::bar();
```

Il est permis de créer un espace de noms dans un autre, par exemple:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
}
```

C ++ 17

Le code ci-dessus pourrait être simplifié comme suit:

```
namespace A::B::C
{
    void bar() {}
}
```

Extension des espaces de noms

Une fonctionnalité utile de l' `namespace` de `namespace` est que vous pouvez les développer (ajouter des membres).

```
namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}
```

Utiliser la directive

Le mot clé "using" a trois saveurs. Combiné avec le mot clé "namespace", vous écrivez une "directive using":

Si vous ne voulez pas écrire `Foo::` devant chaque élément de l'espace de noms `Foo` , vous pouvez utiliser `using namespace Foo;` importer chaque chose de `Foo` .

```
namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

Il est également possible d'importer des entités sélectionnées dans un espace de noms plutôt que dans tout l'espace de noms:

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

Un mot d'avertissement: l' `using namespace` dans les fichiers d'en-tête est considérée comme un mauvais style dans la plupart des cas. Si cela est fait, l'espace de noms est importé dans *chaque*

fichier qui inclut l'en-tête. Comme il n'y a aucun moyen de « ONU- en `using` » un espace de noms, cela peut conduire à la pollution de l' espace de noms (symboles plus ou inattendus dans l'espace de noms global) ou, pire, les conflits. Voir cet exemple pour une illustration du problème:

```
/***** foo.h *****/
namespace Foo
{
    class C;
}

/***** bar.h *****/
namespace Bar
{
    class C;
}

/***** baz.h *****/
#include "foo.h"
using namespace Foo;

/***** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

Une *directive using* ne peut pas exister à la portée de la classe.

Recherche dépendante de l'argument

Lors de l'appel d'une fonction sans qualificateur d'espace de nommage explicite, le compilateur peut choisir d'appeler une fonction dans un espace de noms si l'un des types de paramètre de cette fonction se trouve également dans cet espace de noms. Cela s'appelle "Argument Dependent Lookup" ou ADL:

```
namespace Test
{
  int call(int i);

  class SomeClass {...};

  int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); //Succeeds
```

`call` échoue car aucun de ses types de paramètres ne provient de l'espace de noms `Test` . `call_too` fonctionne car `SomeClass` est membre de `Test` et par conséquent, il est qualifié pour les règles ADL.

Quand l'ADL ne se produit pas

ADL ne se produit pas si la recherche normale non qualifiée trouve un membre de classe, une fonction qui a été déclarée à portée de bloc ou quelque chose qui n'est pas de type fonction. Par exemple:

```
void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});       // error: ADL is disabled and ::foo() doesn't take any arguments
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}
```

Espace de noms en ligne

C ++ 11

`inline namespace` inclut le contenu de l’espace de noms inséré dans l’espace de nom

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

est principalement équivalent à

```
namespace Outer
{

    namespace Inner
    {
        void foo();
    }

```

```
    using Inner::foo;
}
```

mais l'élément de `Outer::Inner::` et ceux associés à `Outer::` sont identiques.

Donc, suivre est équivalent

```
Outer::foo();
Outer::Inner::foo();
```

L'alternative `using namespace Inner;` ne serait pas équivalent pour certaines parties délicates en tant que spécialisation de modèle:

Pour

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- L'espace de noms en ligne permet la spécialisation de `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      inline namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
  }
  ```

- Considérant que l' `using namespace` ne permet pas la spécialisation de `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
      using namespace Inner;
      // Specialization of `Outer::foo` is not possible
      // it should be `Outer::Inner::foo`.
  }
  ```

L'espace de nommage en ligne est un moyen de permettre à plusieurs versions de cohabiter et de passer à la version en `inline`

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }

}
```

Et avec l'usage

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Espace de noms anonyme / anonyme

Un espace de nom sans nom peut être utilisé pour garantir que les noms ont un lien interne (qui ne peut être référencé que par l'unité de traduction actuelle). Un tel espace de noms est défini de la même manière que tout autre espace de noms, mais sans le nom:

```
namespace {
    int foo = 42;
}
```

`foo` n'est visible que dans l'unité de traduction dans laquelle il apparaît.

Il est recommandé de ne jamais utiliser d'espaces de noms non nommés dans les fichiers d'en-tête, car cela donne une version du contenu de chaque unité de traduction dans laquelle elle est incluse. Ceci est particulièrement important si vous définissez des globales non const.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...
```

```
std::cout << globalString; //< Will always print the empty string
```

Espaces de noms imbriqués compacts

C ++ 17

```
namespace a {
  namespace b {
    template<class T>
    struct qualifies : std::false_type {};
  }
}

namespace other {
  struct bob {};
}

namespace a::b {
  template<>
  struct qualifies<::other::bob> : std::true_type {};
}
```

Vous pouvez entrer à la fois l' `a` et `b` namespaces en une seule étape avec `namespace a::b` à partir de 17 C ++.

Aliasing d'un long espace de noms

Ceci est généralement utilisé pour renommer ou raccourcir de longues références d'espace de noms, faisant référence aux composants d'une bibliothèque.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;


//    Both Type declarations are equivalent
boost::multiprecision::Number X   //    Writing the full namespace path, longer
Name1::Number Y                   //    using the name alias, shorter
```

Portée de déclaration d'alias

La déclaration d'alias est affectée par l' *utilisation des* instructions précédentes

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
```

```
    }
}


using namespace boost;

//   Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

Cependant, il est plus facile de se perdre dans l’espace de noms que vous utilisez lorsque vous avez quelque chose comme ça:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;

//    Not recommended as
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Alias d'espace de noms

Un espace de noms peut recevoir un alias (*c.-à-d. Un* autre nom pour le même espace de noms) en utilisant l' *identificateur d'* `namespace =` syntaxe. Les membres de l'espace de noms aliasé sont accessibles en les qualifiant avec le nom de l'alias. Dans l'exemple suivant, l'espace de noms imbriqué `AReallyLongName::AnotherReallyLongName` n'est pas pratique à taper, donc la fonction `qux` déclare localement un alias `N` Les membres de cet espace de noms peuvent alors être accédés simplement en utilisant `N::` .

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
```

```
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

Lire Espaces de noms en ligne: https://riptutorial.com/fr/cplusplus/topic/495/espaces-de-noms

Chapitre 35: Exemples de serveur client

Examples

Bonjour TCP Server

Permettez-moi de commencer en disant que vous devriez d'abord visiter le Guide de programmation réseau de Beej et le lire rapidement, ce qui explique la plupart de ces choses un peu plus verbalement. Nous allons créer un simple serveur TCP ici qui dira "Hello World" à toutes les connexions entrantes et les fermera ensuite. Une autre chose à noter est que le serveur communiquera avec les clients de manière itérative, ce qui signifie un client à la fois. Assurez-vous de consulter les pages de manuel pertinentes car elles peuvent contenir des informations précieuses sur chaque structure d'appel et de socket.

Nous allons exécuter le serveur avec un port, nous allons donc également prendre un argument pour le numéro de port. Commençons avec le code -

```
#include <cstring>    // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h>    // close()


int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
        std::cerr << "Run program as 'program <port>'\n";
        return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8;  // number of connections allowed on the incoming queue


    addrinfo hints, *res, *p;    // we need 2 pointers, res to hold and p to iterate over
    memset(&hints, 0, sizeof(hints));

    // for more explanation, man socket
    hints.ai_family   = AF_UNSPEC;    // don't specify which IP version to use yet
    hints.ai_socktype = SOCK_STREAM;  // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
    hints.ai_flags    = AI_PASSIVE;


    // man getaddrinfo
    int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
    if (gAddRes != 0) {
```

```
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }

    std::cout << "Detecting addresses" << std::endl;

    unsigned int numOfAddr = 0;
    char ipStr[INET6_ADDRSTRLEN];    // ipv6 length makes sure both ipv4/6 addresses can be
stored in this variable


    // Now since getaddrinfo() has given us a list of addresses
    // we're going to iterate over them and ask user to choose one
    // address for program to bind to
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        std::string ipVer;

        // if address is ipv4 address
        if (p->ai_family == AF_INET) {
            ipVer             = "IPv4";
            sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
            addr              = &(ipv4->sin_addr);
            ++numOfAddr;
        }

        // if address is ipv6 address
        else {
            ipVer              = "IPv6";
            sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
            addr               = &(ipv6->sin6_addr);
            ++numOfAddr;
        }

        // convert IPv4 and IPv6 addresses from binary to text form
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " << ipStr
                  << std::endl;
    }

    // if no addresses found :(
    if (!numOfAddr) {
        std::cerr << "Found no host address to use\n";
        return -3;
    }

    // ask user to choose an address
    std::cout << "Enter the number of host address to bind with: ";
    unsigned int choice = 0;
    bool madeChoice     = false;
    do {
        std::cin >> choice;
        if (choice > (numOfAddr + 1) || choice < 1) {
            madeChoice = false;
            std::cout << "Wrong choice, try again!" << std::endl;
        } else
            madeChoice = true;
    } while (!madeChoice);
```

```
    p = res;

    // let's create a new socket, socketFD is returned as descriptor
    // man socket for more information
    // these calls usually return -1 as result of some error
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        freeaddrinfo(res);
        return -4;
    }


    // Let's bind address to our socket we've just created
    int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
    if (bindR == -1) {
        std::cerr << "Error while binding socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -5;
    }


    // finally start listening for connections on our socket
    int listenR = listen(sockFD, backLog);
    if (listenR == -1) {
        std::cerr << "Error while Listening on socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -6;
    }


    // structure large enough to hold client's address
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);


    const std::string response = "Hello World";


    // a fresh infinite loop to communicate with incoming connections
    // this will take client connections one at a time
    // in further examples, we're going to use fork() call for each client connection
    while (1) {

        // accept call will give us a new socket descriptor
        int newFD
          = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
        if (newFD == -1) {
            std::cerr << "Error while Accepting on socket\n";
            continue;
        }

        // send call sends the data you specify as second param and it's length as 3rd param,
also returns how many bytes were actually sent
        auto bytes_sent = send(newFD, response.data(), response.length(), 0);
```

```
        close(newFD);
    }

    close(sockFD);
    freeaddrinfo(res);

    return 0;
}
```

Le programme suivant s'exécute comme -

```
Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1
```

Bonjour client TCP

Ce programme est complémentaire au programme Hello TCP Server, vous pouvez les exécuter pour vérifier leur validité. Le flux de programme est assez courant avec le serveur Hello TCP, alors assurez-vous de jeter un oeil à cela.

Voici le code -

```
#include <cstring>
#include <iostream>
#include <string>

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>


int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }


    auto &ipAddress = argv[1];
    auto &portNum   = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
```

```
    }

    if (p == NULL) {
        std::cerr << "No addresses found\n";
        return -3;
    }

    // socket() call creates a new socket and returns it's descriptor
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        return -4;
    }

    // Note: there is no bind() call as there was in Hello TCP Server
    // why? well you could call it though it's not necessary
    // because client doesn't necessarily has to have a fixed port number
    // so next call will bind it to a random available port number

    // connect() call tries to establish a TCP connection to the specified server
    int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
    if (connectR == -1) {
        close(sockFD);
        std::cerr << "Error while connecting socket\n";
        return -5;
    }

    std::string reply(15, ' ');

    // recv() call tries to get the response from server
    // BUT there's a catch here, the response might take multiple calls
    // to recv() before it is completely received
    // will be demonstrated in another example to keep this minimal
    auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
    if (bytes_recv == -1) {
        std::cerr << "Error while receiving bytes\n";
        return -6;
    }

    std::cout << "\nClient recieved: " << reply << std::endl;
    close(sockFD);
    freeaddrinfo(p);

    return 0;
}
```

Lire Exemples de serveur client en ligne: https://riptutorial.com/fr/cplusplus/topic/7177/exemples-de-serveur-client

Chapitre 36: Expressions régulières

Introduction

Les expressions régulières (parfois appelées regexs ou regexps) sont une syntaxe textuelle qui représente les modèles pouvant être mis en correspondance dans les chaînes utilisées.

Les expressions régulières, introduites dans c ++ 11 , peuvent éventuellement prendre en charge un tableau de retour de chaînes correspondantes ou une autre syntaxe textuelle définissant la manière de remplacer les modèles correspondants dans les chaînes sur lesquelles elles sont exécutées.

Syntaxe

- regex_match // Indique si l'intégralité de la séquence de caractères a été appariée par l'expression régulière, éventuellement capturée dans un objet de correspondance
- regex_search // Indique si une partie de la séquence de caractères a été appariée par l'expression régulière, éventuellement capturée dans un objet de correspondance
- regex_replace // Renvoie la séquence de caractères d'entrée modifiée par une regex via une chaîne de format de remplacement
- regex_token_iterator // Initialisé avec une séquence de caractères définie par les itérateurs, une liste des index de capture à parcourir et une expression régulière. Dereferencing renvoie la correspondance actuellement indexée du regex. L'incrémentation passe à l'index de capture suivant ou, si elle se trouve au dernier index, réinitialise l'index et empêche la prochaine occurrence d'une correspondance regex dans la séquence de caractères.
- regex_iterator // Initialisé avec une séquence de caractères définie par des itérateurs et une regex. Le déréférencement renvoie la partie de la séquence de caractères à laquelle correspond la regex entière. L'incrémentation trouve l'occurrence suivante d'une correspondance d'expression régulière dans la séquence de caractères

Paramètres

Signature	La description
`bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)`	**`BidirectionalIterator`** est tout iterator de caractères qui fournit incrémentation et les opérateurs décrémentation **`smatch`** peuvent être `cmatch` ou tout autre autre variante de `match_results` qui accepte le type de `BidirectionalIterator` le `smatch` argument peut être ignoré si les résultats de l'expression régulière ne sont pas nécessaires Vérifie si `re` correspond au caractère entier séquence définie par le `first` et le `last`

Signature	La description
`bool regex_match(const string& str, smatch& sm, const regex re&, regex_constraints::match_flag_type flags)`	**`string`** peut être un `const char*` ou une `string` L-Value, *les fonctions acceptant une `string` R-Value sont explicitement supprimées* **`smatch`** peut être `cmatch` ou toute autre variante de `match_results` qui accepte le type de `str` l'argument `smatch` peut être omis si les résultats du regex ne sont pas nécessaires **Renvoie** si `re` correspond à toute la séquence de caractères définie par `str`

Examples

Exemples de base de regex_match et regex_search

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use
regular expressions.\""s;
smatch sm;

cout << input << endl;

// If input ends in a quotation that contains a word that begins with "reg" and another word
begining with "ex" then capture the preceeding portion of input
if (regex_match(input, sm, regex("(.*)\".*\\breg.*\\bex.*\"\\s*$"))) {
    const auto capture = sm[1].str();

    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a
problem, think\n"

    // Search our capture for "a problem" or "# problems"
    if(regex_search(capture, sm, regex("(a|d+)\\s+problems?"))) {
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);

        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1
problem\n"
        cout << "Now they have " << count + 1 << " problems.\n"; // Ouputs: "Now they have 2
problems\n"
    }
}
```

Exemple Live

Exemple de regex_replace

Ce code prend différents styles d'accolades et les convertit en un seul style d'accolade:

```
const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n    foo();\n}\nif
(allman)\n{\n\tfoo();\n}\nif (horstmann)\n{\tfoo();\n}\nif (pico)\n{\tfoo(); }\nif
(whitesmiths)\n\t{\n\tfoo();\n\t}\n"s;

cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+?;)\\s*\\}?\\s*"), "$1
{\n\t$2\n}\n") << endl;
```

Exemple Live

Exemple avec regex_token_iterator

Un `std::regex_token_iterator` fournit un formidable outil pour extraire des éléments d'un fichier de valeurs séparées par des virgules . Outre les avantages de l'itération, cet itérateur est également capable de capturer des virgules échappées lorsque d'autres méthodes sont en difficulté:

```
const auto input = "please split,this,csv, ,line,\\,\n"s;
const regex re{ "((?:[^\\\\,]|\\\\.)+)(?:,|$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\n"));
```

Exemple Live

Un piège notable avec les itérateurs de regex est que l'argument `regex` doit être une valeur L. Une valeur R ne fonctionnera pas .

Exemple de regex_iterator

Lorsque le traitement des captures doit être effectué itérativement, un `regex_iterator` est un bon choix. `regex_iterator` un `regex_iterator` renvoie un `match_result` . Ceci est idéal pour les captures conditionnelles ou les captures qui ont une interdépendance. Disons que nous voulons tokenize du code C ++. Donné:

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

On peut tokenize cette chaîne: `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s` avec un `regex_iterator` comme ceci:

```
vector<TOKENS> tokens;
const regex re{ "\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\)?)\\s*(?:(\\+)|(-)|(\\*)|(/)|(=))" };

for_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto&
i) {
    if(i[1].length() > 0) {
        tokens.push_back(OPEN_PARENTHESIS);
    }

    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);
```

```
    if(i[3].length() > 0) {
        tokens.push_back(CLOSE_PARENTHESIS);
    }

    auto it = next(cbegin(i), 4);

    for(int result = ADDITION; it != cend(i); ++result, ++it) {
        if (it->length() > 0U) {
            tokens.push_back(static_cast<TOKENS>(result));
            break;
        }
    }
});

match_results<string::const_reverse_iterator> sm;

if(regex_search(crbegin(input), crend(input), sm, regex{ tokens.back() == SUBTRACTION ?
"^\\s*\\d+\\s*-\\s*(-?)" : "^\\s*\\d+\\s*(-?)" })) {
    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);
}
```

Exemple Live

Un argument notable avec les itérateurs de regex est que l'argument `regex` doit être une valeur L, une valeur R ne fonctionnera pas: Visual Studio regex_iterator Bug?

Fractionner une chaîne

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}
```

```
split("Some  string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

Quantificateurs

Disons que nous avons reçu `const string input` tant que numéro de téléphone à valider. Nous pourrions commencer par demander une entrée numérique avec un **quantificateur nul ou plus** : `regex_match(input, regex("\\d*"))` ou un **ou plusieurs quantificateurs** : `regex_match(input, regex("\\d+"))` Les deux ne sont vraiment pas corrects si l' `input` contient une chaîne numérique invalide telle que: "123" Utilisons un **quantificateur n ou plus** pour nous assurer que nous obtenons au moins 7 chiffres:

```
regex_match(input, regex("\\d{7,}"))
```

Cela garantira que nous aurons au moins un numéro de téléphone de chiffres, mais l' `input` pourrait également contenir une chaîne numérique trop longue comme: "123456789012". Laissons donc **entre** un **quantificateur compris entre n et m, de** sorte que l' `input` comporte au

moins 7 chiffres mais pas plus de 11:

```
regex_match(input, regex("\\d{7,11}"));
```

Cela nous rapproche, mais les chaînes numériques illégales comprises entre [7, 11] sont toujours acceptées, comme: "123456789" Rendons donc le code du pays facultatif avec un **quantificateur différé** :

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

Il est important de noter que le **quantificateur paresseux** correspond *au moins de caractères possible* , de sorte que la seule façon dont ce caractère sera comparé est de savoir si 10 caractères ont déjà été identifiés par `\d{7,10}` . (Pour faire correspondre le premier caractère avec avidité, nous aurions dû faire: `\d{0,1}` .) Le **quantificateur paresseux** peut être ajouté à tout autre quantificateur.

Maintenant, comment pourrions-nous rendre l'indicatif régional facultatif *et* accepter uniquement un code de pays si l'indicatif régional était présent?

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

Dans cette dernière regex, le `\d{7}` *nécessite* 7 chiffres. Ces 7 chiffres sont éventuellement précédés de 3 ou 4 chiffres.

Notez que nous n'avons pas ajouté le **quantificateur paresseux** : ~~`\d{3,4}?\d{7}`~~ , le `\d{3,4}?` aurait correspondu soit 3 ou 4 caractères, préférant 3. Au lieu de cela, nous faisons la correspondance de groupe non capturant au plus une fois, préférant ne pas correspondre. Causant une non-concordance si l' `input` ne comprenait pas l'indicatif régional tel que: "1234567".

En conclusion du sujet du quantificateur, j'aimerais mentionner l'autre quantificateur que vous pouvez utiliser, le **quantificateur possessif** . *Le* **quantificateur différé** ou le **quantificateur possessif** peuvent être ajoutés à tout quantificateur. La seule fonction du **quantificateur possessif** est d 'aider le moteur regex en lui disant, de prendre ces caractères avec avidité *et de ne jamais les abandonner même si cela provoque l' échec de l 'expression rationnelle* . Cela n'a pas beaucoup de sens, par exemple: `regex_match(input, regex("\\d{3,4}+\\d{7}))` car une `input` comme" 1234567890 "ne correspondrait pas à `\d{3,4}+` correspondra toujours à 4 caractères même si la correspondance 3 aurait permis à l'expression rationnelle de réussir.
Le **quantificateur possessif** est mieux utilisé *lorsque le jeton quantifié limite le nombre de caractères pouvant être associés* . Par exemple:

```
regex_match(input, regex("(?:.*\\d{3,4}+){3}"))
```

Peut être utilisé pour faire correspondre si l’ `input` contient l’un des éléments suivants:

> 123 456 7890
> 123-456-7890

(123)456-7890
(123) 456 - 7890

Mais quand cette regex brille vraiment, c'est quand l' `input` contient une entrée *illégale* :

12345 - 67890

Sans le **quantificateur possessif,** le moteur d'expressions rationnelles doit revenir en arrière et tester *toutes les combinaisons de `.*` Et de 3 ou 4 caractères* pour voir s'il peut trouver une combinaison compatible. Avec le **quantificateur possessif,** l'expression rationnelle commence là où le deuxième **quantificateur possessif s'est** arrêté, le caractère '0' et le moteur d'expression régulière essaie d'ajuster le `.*` Pour permettre à `\d{3,4}` de correspondre; Si le regex ne réussit pas, il n'y a pas de retour en arrière pour voir si un ajustement a été effectué auparavant `.*` .

Ancres

C ++ ne fournit que 4 ancres:

- `^` qui affirme le début de la chaîne
- `$` qui affirme la fin de la chaîne
- `\b` qui affirme un caractère `\W` ou le début ou la fin de la chaîne
- `\B` qui affirme un caractère `\w`

Disons par exemple que nous voulons capturer un numéro *avec* son signe:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {

    do {
        cout << sm[1] << endl;
        input = sm.suffix().str();
    } while(regex_search(input, sm, regex{ "(?:^\\W|\\b\\W)([+-]?\\d+)" }));
}
```

Exemple Live

Une note importante ici est que l'ancre ne consomme aucun caractère.

Lire Expressions régulières en ligne: https://riptutorial.com/fr/cplusplus/topic/1681/expressions-regulieres

Chapitre 37: Fichier I / O

Introduction

Les E / S de fichiers C ++ sont effectuées via des *flux* . Les principales abstractions sont:

`std::istream` pour lire du texte.

`std::ostream` pour écrire du texte.

`std::streambuf` pour lire ou écrire des caractères.

Entrée formatée utilise un `operator>>` .

La sortie formatée utilise l' `operator<<` .

Les flux utilisent `std::locale` , par exemple pour plus de détails sur le formatage et la traduction entre les codages externes et le codage interne.

Plus sur les flux: <iostream> Library

Examples

Ouvrir un fichier

L'ouverture d'un fichier se fait de la même manière pour les 3 flux de fichiers (`ifstream` , `ofstream` et `fstream`).

Vous pouvez ouvrir le fichier directement dans le constructeur:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.

std::fstream iofs("foo.txt");  // fstream:  Opens file "foo.txt" for reading and writing.
```

Vous pouvez également utiliser la fonction membre `open()` du fichier de flux de données:

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream:  Opens file "bar.txt" for reading and writing.
```

Vous devez **toujours** vérifier si un fichier a été ouvert avec succès (même lors de l'écriture). Les échecs peuvent inclure: le fichier n'existe pas, le fichier n'a pas les droits d'accès appropriés, le

fichier est déjà utilisé, les erreurs de disque se sont produites, le disque a été déconnecté ... La vérification peut être effectuée comme suit:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt");  // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

Lorsque le chemin du fichier contient des barres obliques inverses (par exemple, sur le système Windows), vous devez leur échapper correctement:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:\\folder\\foo.txt"); // using escaped backslashes
```

C ++ 11

ou utilisez le littéral brut:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs(R"(c:\folder\foo.txt)"); // using raw literal
```

ou utilisez plutôt des barres obliques:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

C ++ 11

Si vous souhaitez ouvrir un fichier avec des caractères non-ASCII dans le chemin sur Windows, vous pouvez utiliser l'argument de chemin de caractère large **non standard** :

```
// Open the file 'пример\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\foo.txt)"); // using wide characters with raw literal
```

Lecture d'un fichier

Il existe plusieurs manières de lire les données d’un fichier.

Si vous savez comment les données sont formatées, vous pouvez utiliser l'opérateur d'extraction de flux (`>>`). Supposons que vous avez un fichier nommé *foo.txt* qui contient les données suivantes:

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

Ensuite, vous pouvez utiliser le code suivant pour lire ces données à partir du fichier:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

L'opérateur d'extraction de flux `>>` extrait chaque caractère et s'arrête s'il trouve un caractère qui ne peut pas être stocké ou s'il s'agit d'un caractère spécial:

- Pour les types de chaîne, l'opérateur s'arrête à un espace blanc ( ) ou à une nouvelle ligne (`\n`).
- Pour les nombres, l'opérateur s'arrête à un caractère sans numéro.

Cela signifie que la version suivante du fichier *foo.txt* sera également lue avec succès par le code précédent:

```
John
Doe 25
4 6 1987


Jane
Doe
15 5
24
1976
```

L'opérateur d'extraction de flux `>>` renvoie toujours le flux qui lui est donné. Par conséquent, plusieurs opérateurs peuvent être enchaînés afin de lire les données consécutivement. Cependant, un courant peut également être utilisé comme une expression booléenne (comme le montre le `while` en boucle dans le code précédent). C'est parce que les classes de flux ont un opérateur de conversion pour le type `bool` . Cet opérateur `bool()` retournera `true` tant que le flux ne contient aucune erreur. Si un flux entre dans un état d'erreur (par exemple, car plus aucune donnée ne peut être extraite), l'opérateur `bool()` renverra `false` . Par conséquent, le `while` en boucle dans le code précédent est sorti après que le fichier d'entrée a été lu à sa fin.

Si vous souhaitez lire un fichier entier en tant que chaîne, vous pouvez utiliser le code suivant:

```
// Opens 'foo.txt'.
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());
```

```
// Sets position to the start of the file.
is.seekg(0, std::ios::beg);

// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
  std::istreambuf_iterator<char>());
```

Ce code réserve un espace pour la `string` afin de réduire les allocations de mémoire inutiles.

Si vous voulez lire un fichier ligne par ligne, vous pouvez utiliser la fonction `getline()` :

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

Si vous voulez lire un nombre fixe de caractères, vous pouvez utiliser la fonction membre `read()` du flux `read()` :

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

Après l' exécution d' une commande de lecture, vous devez toujours vérifier si l'état d'erreur drapeau `failbit` a été défini, car il indique si l'opération a échoué ou non. Cela peut être fait en appelant la fonction membre du flux de fichiers `fail()` :

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

Ecrire dans un fichier

Il existe plusieurs manières d'écrire dans un fichier. Le plus simple est d'utiliser un flux de fichiers de sortie (`ofstream`) avec l'opérateur d'insertion de flux (`<<`):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Au lieu de `<<` , vous pouvez également utiliser la fonction membre du flux du fichier de sortie `write()` :

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";
```

```
    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

Après avoir écrit à un flux, vous devriez toujours vérifier si l' état d'erreur drapeau `badbit` a été défini, car il indique si l'opération a échoué ou non. Cela peut être fait en appelant la fonction membre du flux du fichier de sortie `bad()` :

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Modes d'ouverture

Lors de la création d'un flux de fichiers, vous pouvez spécifier un mode d'ouverture. Un mode d'ouverture est essentiellement un paramètre permettant de contrôler la manière dont le flux ouvre le fichier.

(Tous les modes peuvent être trouvés dans l'espace de noms `std::ios` .)

Un mode d'ouverture peut être fourni comme second paramètre au constructeur d'un flux de fichiers ou à sa fonction membre `open()` :

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

Il convient de noter que vous devez définir `ios::in` ou `ios::out` si vous souhaitez définir d'autres indicateurs, car ils ne sont pas implicitement définis par les membres iostream, bien qu'ils aient une valeur par défaut correcte.

Si vous ne spécifiez pas de mode d'ouverture, les modes par défaut suivants sont utilisés:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` et `out`

Les modes d'ouverture de fichier que vous pouvez spécifier par conception sont les suivants:

Mode	Sens	Pour	La description
`app`	ajouter	Sortie	Ajoute des données à la fin du fichier.
`binary`	binaire	Entrée sortie	L'entrée et la sortie se font en binaire.
`in`	contribution	Contribution	Ouvre le fichier pour la lecture.

Mode	Sens	Pour	La description
`out`	sortie	Sortie	Ouvre le fichier pour l'écriture.
`trunc`	tronquer	Entrée sortie	Supprime le contenu du fichier lors de son ouverture.
`ate`	à la fin	Contribution	Va à la fin du fichier lors de l'ouverture.

Remarque: la définition du mode `binary` permet de lire / écrire les données exactement telles quelles; Si vous ne la définissez pas, la nouvelle ligne `'\n'` convertie en une séquence de fin de ligne spécifique à la plate-forme.

Par exemple, sous Windows, la séquence de fin de ligne est CRLF (`"\r\n"`).
Ecrire: `"\n"` => `"\r\n"`
Lire: `"\r\n"` => `"\n"`

Fermer un fichier

La fermeture explicite d'un fichier est rarement nécessaire en C ++, car un flux de fichiers ferme automatiquement son fichier associé dans son destructeur. Toutefois, vous devez essayer de limiter la durée de vie d'un objet de flux de fichiers, afin qu'il ne conserve pas le descripteur de fichiers ouvert plus longtemps que nécessaire. Par exemple, cela peut être fait en plaçant toutes les opérations sur les fichiers dans une portée propre (`{}`):

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
}  // The ofstream will go out of scope here.
   // Its destructor will take care of closing the file properly.
```

L'appel de `close()` explicitement n'est nécessaire que si vous souhaitez réutiliser le même objet `fstream` ultérieurement, mais que vous ne souhaitez pas conserver le fichier ouvert entre:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();

// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();
```

```
// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();
```

Flushing un flux

Les flux de fichiers sont mis en mémoire tampon par défaut, de même que de nombreux autres types de flux. Cela signifie que les écritures dans le flux ne peuvent pas entraîner la modification immédiate du fichier sous-jacent. Pour forcer toutes les écritures mises en mémoire tampon à se produire immédiatement, vous pouvez *vider* le flux. Vous pouvez le faire directement en appelant la méthode `flush()` ou via le manipulateur `std::flush` stream:

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

Il y a un flux manipulateur `std::endl` qui combine l'écriture d'une nouvelle ligne avec le vidage du flux:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

La mise en mémoire tampon peut améliorer les performances d'écriture sur un flux. Par conséquent, les applications qui écrivent beaucoup doivent éviter de vider inutilement. À l'inverse, si les E / S sont effectuées rarement, les applications devraient envisager de les vider fréquemment afin d'éviter que les données ne restent bloquées dans l'objet flux.

Lire un fichier ASCII dans une chaîne std ::

```
std::ifstream f("file.txt");

if (f)
{
  std::stringstream buffer;
  buffer << f.rdbuf();
  f.close();

  // The content of "file.txt" is available in the string `buffer.str()`
}
```

La méthode `rdbuf()` renvoie un pointeur sur un `streambuf` qui peut être poussé dans le `buffer` via la fonction membre `stringstream::operator<<` .

Une autre possibilité (popularisée dans Effective STL par Scott Meyers) est:

```
std::ifstream f("file.txt");

if (f)
{
  std::string str((std::istreambuf_iterator<char>(f)),
                  std::istreambuf_iterator<char>());

  // Operations on `str`...
}
```

Ceci est intéressant car nécessite peu de code (et permet de lire un fichier directement dans un conteneur STL, pas seulement les chaînes), mais peut être lent pour les gros fichiers.

Remarque : les parenthèses supplémentaires autour du premier argument du constructeur de chaîne sont essentielles pour éviter le problème d' *analyse* le *plus frustrant* .

Enfin et surtout:

```
std::ifstream f("file.txt");

if (f)
{
  f.seekg(0, std::ios::end);
  const auto size = f.tellg();

  std::string str(size, ' ');
  f.seekg(0);
  f.read(&str[0], size);
  f.close();

  // Operations on `str`...
}
```

qui est probablement l'option la plus rapide (parmi les trois proposées).

Lecture d'un fichier dans un conteneur

Dans l'exemple ci-dessous, nous utilisons `std::string` et `operator>>` pour lire les éléments du fichier.

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v;

    std::string s;
    while(file >> s) // keep reading until we run out
    {
        v.push_back(s);
    }
```

Dans l'exemple ci-dessus, nous parcourons simplement le fichier en lisant un "élément" à la fois

en utilisant l' `operator>>` . Ce même effet peut être obtenu en utilisant le `std::istream_iterator` qui est un itérateur en entrée qui lit un "élément" à la fois dans le flux. De plus, la plupart des conteneurs peuvent être construits à l'aide de deux itérateurs afin de simplifier le code ci-dessus pour:

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v(std::istream_iterator<std::string>{file},
                                std::istream_iterator<std::string>{});
```

Nous pouvons étendre cela pour lire tous les types d'objets que nous aimons en spécifiant simplement l'objet que nous voulons lire en tant que paramètre de modèle pour `std::istream_iterator` . Ainsi, nous pouvons simplement étendre ce qui précède pour lire des lignes (plutôt que des mots) comme ceci:

```
// Unfortunately there is  no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};


    std::ifstream file("file3.txt");

    // Read the lines of a file into a container.
    std::vector<std::string>  v(std::istream_iterator<Line>{file},
                                std::istream_iterator<Line>{});
```

Lecture d'un `struct` à partir d'un fichier texte formaté.

C ++ 11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
```

```
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "   age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

fichier4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Sortie:

```
name: Wogger Wabbit
 age: 2 years
height: 6.2lbs

name: Bilbo Baggins
 age: 111 years
height: 81.3lbs

name: Mary Poppins
 age: 29 years
height: 154.8lbs
```

Copier un fichier

```
std::ifstream  src("source_filename", std::ios::binary);
std::ofstream  dst("dest_filename",   std::ios::binary);
```

```
dst << src.rdbuf();
```

C ++ 17

Avec C ++ 17, la méthode standard pour copier un fichier consiste à inclure l'en-tête `<filesystem>` et à utiliser `copy_file` :

```
std::fileystem::copy_file("source_filename", "dest_filename");
```

La bibliothèque de systèmes de fichiers a été initialement développée en tant que `boost.filesystem` et finalement fusionnée en ISO C ++ à partir de C ++ 17.

Vérification de la fin du fichier dans une condition de boucle, mauvaise pratique?

`eof` retourne `true` qu'après **avoir** lu la fin du fichier. Il n'indique PAS que la prochaine lecture sera la fin du flux.

```
while (!f.eof())
{
  // Everything is OK

  f >> buffer;

  // What if *only* now the eof / fail bit is set?

  /* Use `buffer` */
}
```

Vous pouvez écrire correctement:

```
while (!f.eof())
{
  f >> buffer >> std::ws;

  if (f.fail())
    break;

  /* Use `buffer` */
}
```

mais

```
while (f >> buffer)
{
  /* Use `buffer` */
}
```

est plus simple et moins sujet aux erreurs.

Autres références:

- `std::ws` : supprime les espaces blancs d'un flux d'entrée
- `std::basic_ios::fail` : renvoie `true` si une erreur s'est produite sur le flux associé

Ecriture de fichiers avec des paramètres régionaux non standard

Si vous avez besoin d'écrire un fichier en utilisant des paramètres régionaux différents à la valeur par défaut, vous pouvez utiliser `std::locale` et `std::basic_ios::imbue()` pour le faire pour un flux de fichiers spécifique:

Guide d'utilisation:

- Vous devez toujours appliquer un local à un flux avant d'ouvrir le fichier.
- Une fois que le flux a été imprégné, vous ne devez pas modifier les paramètres régionaux.

Raisons des restrictions: Si vous modifiez un flux de fichiers avec des paramètres régionaux, le comportement n'est pas défini si les paramètres régionaux actuels ne sont pas indépendants de l'état ou ne pointent au début du fichier.

Les flux UTF-8 (et autres) ne sont pas indépendants de l'état. De plus, un flux de fichiers avec un environnement local UTF-8 peut essayer de lire le marqueur de nomenclature à partir du fichier lorsqu'il est ouvert. il suffit donc d'ouvrir le fichier pour lire les caractères du fichier et ce ne sera pas au début.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
  std::cout << "User-preferred locale setting is "
            << std::locale("").name().c_str() << std::endl;

  // Write a floating-point value using the user's preferred locale.
  std::ofstream ofs1;
  ofs1.imbue(std::locale(""));
  ofs1.open("file1.txt");
  ofs1 << 78123.456 << std::endl;

  // Use a specific locale (names are system-dependent)
  std::ofstream ofs2;
  ofs2.imbue(std::locale("en_US.UTF-8"));
  ofs2.open("file2.txt");
  ofs2 << 78123.456 << std::endl;

  // Switch to the classic "C" locale
  std::ofstream ofs3;
  ofs3.imbue(std::locale::classic());
  ofs3.open("file3.txt");
  ofs3 << 78123.456 << std::endl;
}
```

Le passage explicite aux paramètres régionaux classiques "C" est utile si votre programme utilise des paramètres régionaux par défaut différents et que vous souhaitez garantir un standard fixe pour la lecture et l'écriture de fichiers. Avec un environnement local préféré "C", l'exemple écrit

```
78,123.456
78,123.456
78123.456
```

Si, par exemple, les paramètres régionaux préférés sont l'allemand et utilisent par conséquent un format numérique différent, l'exemple écrit

```
78 123,456
78,123.456
78123.456
```

(notez la virgule décimale dans la première ligne).

Lire Fichier I / O en ligne: https://riptutorial.com/fr/cplusplus/topic/496/fichier-i---o

Chapitre 38: Fichiers d'en-tête

Remarques

En C ++, comme en C, le compilateur et le processus de compilation C ++ utilisent le préprocesseur C. Comme spécifié dans le manuel GNU C Preprocessor, un fichier d'en-tête est défini comme suit:

> Un fichier d'en-tête est un fichier contenant des déclarations C et des définitions de macro (voir Macros) à partager entre plusieurs fichiers sources. Vous demandez l'utilisation d'un fichier d'en-tête dans votre programme en l'incluant, avec la directive de prétraitement C '#include'.
>
> Les fichiers d'en-tête ont deux objectifs.
>
> - Les fichiers d'en-tête système déclarent les interfaces à des parties du système d'exploitation. Vous les incluez dans votre programme pour fournir les définitions et les déclarations dont vous avez besoin pour appeler les appels système et les bibliothèques.
> - Vos propres fichiers d'en-tête contiennent des déclarations pour les interfaces entre les fichiers source de votre programme. Chaque fois que vous avez un groupe de déclarations associées et de définitions de macros dont la plupart ou la plupart sont nécessaires dans plusieurs fichiers sources différents, il est judicieux de créer un fichier d'en-tête pour ces fichiers.

Cependant, pour le préprocesseur C lui-même, un fichier d'en-tête n'est pas différent d'un fichier source.

Le schéma d'organisation des fichiers en-tête / source est simplement une convention standard et tenue en main établie par différents projets logiciels afin de fournir une séparation entre interface et implémentation.

Bien qu'il ne soit pas formellement appliqué par le standard C ++ lui-même, il est fortement recommandé de suivre la convention des fichiers en-tête / source et, dans la pratique, il est déjà presque omniprésent.

Notez que les fichiers d'en-tête peuvent être remplacés en tant que convention de structure de fichier de projet par la fonctionnalité à venir des modules, qui doit encore être prise en compte lors de l'écriture (par exemple C ++ 20).

Examples

Exemple de base

L'exemple suivant contiendra un bloc de code à scinder en plusieurs fichiers sources, comme indiqué par `// filename` comment.

Fichiers source

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
  return global_value; // return 42;
}
```

Les fichiers d'en-tête sont ensuite inclus par d'autres fichiers sources qui souhaitent utiliser la fonctionnalité définie par l'interface d'en-tête, mais ne nécessitent pas de connaissances sur son implémentation (réduisant ainsi le couplage de code). Le programme suivant utilise l'en-tête `my_function.h` tel que défini ci-dessus:

```
// main.cpp

#include <iostream>       // A C++ Standard Library header.
#include "my_function.h"  // A personal header

int main(int argc, char** argv) {
  std::cout << my_function() << std::endl;
  return 0;
```

```
}
```

Le processus de compilation

Les fichiers d'en-tête faisant souvent partie d'un workflow de processus de compilation, un processus de compilation classique utilisant la convention d'en-tête / fichier source effectue généralement les opérations suivantes.

En supposant que le fichier d'en-tête et le fichier de code source se trouvent déjà dans le même répertoire, un programmeur exécutera les commandes suivantes:

```
g++ -c my_function.cpp       # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o   # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

Sinon, si vous souhaitez compiler d'abord `main.cpp` dans un fichier objet, puis ne lier que les fichiers objets comme étape finale:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Modèles dans les fichiers d'en-tête

Les modèles requièrent la génération de code à la compilation: une fonction basée sur un modèle, par exemple, sera effectivement transformée en plusieurs fonctions distinctes une fois qu'une fonction basée sur un modèle est paramétrée par utilisation dans le code source.

Cela signifie que la fonction de modèle, la fonction membre et les définitions de classe ne peuvent pas être déléguées à un fichier de code source distinct, car tout code qui utilisera une construction basée sur un modèle nécessite la connaissance de sa définition pour générer un code dérivé.

Ainsi, le code modélisé, s'il est placé en en-tête, doit également contenir sa définition. Un exemple de ceci est ci-dessous:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
  T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                        // for C++11 or later
  return type_point;
}
```

Lire Fichiers d'en-tête en ligne: https://riptutorial.com/fr/cplusplus/topic/7211/fichiers-d-en-tete

Chapitre 39: Filetage

Syntaxe

- fil()
- thread (thread && autre)
- thread explicite (Fonction && func, Args && ... args)

Paramètres

Paramètre	Détails
`other`	Prend possession de l' `other` , l' `other` ne possède plus le fil
`func`	Fonction d'appeler dans un fil séparé
`args`	Arguments pour le `func`

Remarques

Quelques notes:

- Deux objets `std::thread` **ne** peuvent **jamais** représenter le même thread.
- Un objet `std::thread` peut être dans un état où il ne représente **aucun** thread (après un déplacement, après avoir appelé `join` , etc.).

Examples

Opérations de filetage

Lorsque vous démarrez un thread, il s'exécute jusqu'à ce qu'il soit terminé.

Souvent, à un moment donné, vous devez (peut-être - le thread peut-être déjà fait) attendre que le thread se termine, car vous souhaitez par exemple utiliser le résultat.

```
int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();

//Now 'n' has the result of the calculation done in the seperate thread
std::cout << n << '\n';
```

Vous pouvez également `detach` le thread en le laissant s'exécuter librement:

```
std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns
```

Passer une référence à un fil

Vous ne pouvez pas transmettre une référence (ou une référence `const`) directement à un thread car `std::thread` les copiera / déplacera. Au lieu de cela, utilisez `std::reference_wrapper` :

```
void foo(int& b)
{
    b = 10;
}

int a = 1;
std::thread thread{ foo, std::ref(a) }; //'a' is now really passed as reference

thread.join();
std::cout << a << '\n'; //Outputs 10
```

```
void bar(const ComplexObject& co)
{
    co.doCalculations();
}

ComplexObject object;
std::thread thread{ bar, std::cref(object) }; //'object' is passed as const&

thread.join();
std::cout << object.getResult() << '\n'; //Outputs the result
```

Créer un thread std ::

En C ++, les threads sont créés à l'aide de la classe std :: thread. Un thread est un flux séparé d'exécution. c'est comme si une aide effectuait une tâche pendant que vous exécutiez une autre tâche simultanément. Lorsque tout le code du thread est exécuté, il se *termine* . Lors de la création d'un thread, vous devez transmettre quelque chose à exécuter. Quelques points à transmettre à un sujet:

- Fonctions gratuites
- Fonctions membres
- Objets foncteur
- Expressions lambda

Exemple de fonction libre - exécute une fonction sur un thread séparé (Exemple Live):

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                 // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Exemple de fonction membre - exécute une fonction membre sur un thread séparé (Exemple Live):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Exemple d'objet Functor (en direct Exemple):

```
#include <iostream>
#include <thread>
```

```
class Bar
{
public:
    void operator()(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Exemple d'expression Lambda (Exemple Live):

```
#include <iostream>
#include <thread>

int main()
{
    auto lambda = [](int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Opérations sur le thread en cours

`std::this_thread` est un `namespace` qui a des fonctions pour faire des choses intéressantes sur le thread en cours depuis la fonction à partir de laquelle il est appelé.

Fonction	La description
`get_id`	Renvoie l'id du thread
`sleep_for`	Dort pour une durée déterminée
`sleep_until`	Dort jusqu'à une heure précise

Fonction	La description
`yield`	Replanifier les threads en cours d'exécution, en donnant la priorité aux autres threads

Obtenir l'ID de thread actuel en utilisant `std::this_thread::get_id` :

```
void foo()
{
    //Print this threads id
    std::cout << std::this_thread::get_id() << '\n';
}

std::thread thread{ foo };
thread.join(); //'threads' id has now been printed, should be something like 12556

foo(); //The id of the main thread is printed, should be something like 2420
```

Dormir pendant 3 secondes en utilisant `std::this_thread::sleep_for` :

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Dormir jusqu'à 3 heures dans le futur en utilisant `std::this_thread::sleep_until` :

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Laisser les autres threads prendre la priorité en utilisant `std::this_thread::yield` :

```
void foo(int a)
{
    for (int i = 0; i < al ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                   //isn't doing anything important

    std::cout << "Hello World!\n";
}
```

```
std::thread thread{ foo, 10 };
thread.join();
```

Utiliser std :: async au lieu de std :: thread

`std::async` est également capable de créer des threads. Comparé à `std::thread` il est considéré comme moins puissant mais plus facile à utiliser lorsque vous souhaitez simplement exécuter une fonction de manière asynchrone.

Appel asynchrone d'une fonction

```
#include <future>
#include <iostream>

unsigned int square(unsigned int i){
    return i*i;
}

int main() {
    auto f = std::async(std::launch::async, square, 8);
    std::cout << "square currently running\n"; //do something while square is running
    std::cout << "result is " << f.get() << '\n'; //getting the result from square
}
```

Pièges courants

- `std::async` renvoie un `std::future` contenant la valeur de retour qui sera calculée par la fonction. Lorsque ce `future` est détruit, il attend que le thread se termine, ce qui rend votre code efficacement unique. Ceci est facilement négligé lorsque vous n'avez pas besoin de la valeur de retour:

  ```
  std::async(std::launch::async, square, 5);
  //thread already completed at this point, because the returning future got destroyed
  ```

- `std::async` fonctionne sans politique de lancement, donc `std::async(square, 5);` compile. Lorsque vous faites cela, le système peut décider s'il veut créer un thread ou non. L'idée était que le système choisisse de créer un thread à moins qu'il n'exécute déjà plus de threads qu'il ne peut exécuter efficacement. Malheureusement, les implémentations choisissent généralement de ne pas créer de thread dans cette situation, donc vous devez remplacer ce comportement par `std::launch::async` ce qui force le système à créer un thread.

- Attention aux conditions de course.

Plus sur async sur les contrats à terme et les promesses

S'assurer qu'un fil est toujours joint

Lorsque le destructeur de `std::thread` est appelé, un appel à `join()` ou `detach()` **doit** avoir été effectué. Si un thread n'a pas été joint ou détaché, alors `std::terminate` sera appelé par défaut. En utilisant RAII , ceci est généralement assez simple à réaliser:

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }

private:

    std::thread t_;
}
```

Ceci est alors utilisé comme ça:

```
void perform_work()
{
    // Perform some work
}

void t()
{
    thread_joiner j{std::thread(perform_work)};
    // Do some other calculations while thread is running
} // Thread is automatically joined here
```

Cela fournit également une sécurité d'exception; Si nous avions créé notre thread normalement et que le travail effectué dans `t()` effectuant d'autres calculs avait généré une exception, `join()` n'aurait jamais été appelée sur notre thread et notre processus aurait été terminé.

Réaffectation des objets thread

Nous pouvons créer des objets de threads vides et leur assigner du travail ultérieurement.

Si nous affectons un objet thread à un autre thread actif, `joinable` , `std::terminate` sera automatiquement appelé avant le remplacement du thread.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
//create 100 thread objects that do nothing
```

```
std::thread executors[100];

// Some code

// I want to create some threads now

for (int i = 0;i < 100;i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
         executors[i] = std::thread(foo);
}
```

Synchronisation de base

La synchronisation des threads peut être effectuée à l'aide de mutex, parmi d'autres primitives de synchronisation. Il existe plusieurs types de mutex fournis par la bibliothèque standard, mais le plus simple est `std::mutex` . Pour verrouiller un mutex, vous construisez un verrou. Le type de verrou le plus simple est `std::lock_guard` :

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

Avec `std::lock_guard` le mutex est verrouillé pendant toute la durée de vie de l'objet verrou. Dans les cas où vous devez contrôler manuellement les régions pour le verrouillage, utilisez plutôt `std::unique_lock` :

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

Plus de structures de synchronisation de threads

Utilisation de variables de condition

Une variable de condition est une primitive utilisée avec un mutex pour orchestrer la communication entre les threads. Bien que ce ne soit ni le moyen exclusif ou le plus efficace pour y parvenir, il peut être parmi les plus simples pour ceux qui connaissent le modèle.

On attend sur une `std::condition_variable` avec un `std::unique_lock<std::mutex>` . Cela permet au

code d'examiner en toute sécurité l'état partagé avant de décider de procéder ou non à l'acquisition.

Vous trouverez ci-dessous un croquis producteur-consommateur utilisant `std::thread` , `std::condition_variable` , `std::mutex` et quelques autres pour rendre les choses intéressantes.

```
#include <condition_variable>
#include <cstddef>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>


int main()
{
    std::condition_variable cond;
    std::mutex mtx;
    std::queue<int> intq;
    bool stopped = false;

    std::thread producer{[&]()
    {
        // Prepare a random number generator.
        // Our producer will simply push random numbers to intq.
        //
        std::default_random_engine gen{};
        std::uniform_int_distribution<int> dist{};

        std::size_t count = 4006;
        while(count--)
        {
            // Always lock before changing
            // state guarded by a mutex and
            // condition_variable (a.k.a. "condvar").
            std::lock_guard<std::mutex> L{mtx};

            // Push a random int into the queue
            intq.push(dist(gen));

            // Tell the consumer it has an int
            cond.notify_one();
        }

        // All done.
        // Acquire the lock, set the stopped flag,
        // then inform the consumer.
        std::lock_guard<std::mutex> L{mtx};

        std::cout << "Producer is done!" << std::endl;

        stopped = true;
        cond.notify_one();
    }};

    std::thread consumer{[&]()
    {
        do{
            std::unique_lock<std::mutex> L{mtx};
```

```
            cond.wait(L,[&]()
            {
                // Acquire the lock only if
                // we've stopped or the queue
                // isn't empty
                return stopped || ! intq.empty();
            });

            // We own the mutex here; pop the queue
            // until it empties out.

            while( ! intq.empty())
            {
                const auto val = intq.front();
                intq.pop();

                std::cout << "Consumer popped: " << val << std::endl;
            }

            if(stopped){
                // producer has signaled a stop
                std::cout << "Consumer is done!" << std::endl;
                break;
            }

        }while(true);
    }};

    consumer.join();
    producer.join();

    std::cout << "Example Completed!" << std::endl;

    return 0;
}
```

Créez un pool de threads simple

Les primitives de threading C ++ 11 sont encore relativement peu nombreuses. Ils peuvent être utilisés pour écrire une construction de niveau supérieur, comme un pool de threads:

C ++ 14

```
struct tasks {
  // the mutex, condition variable and deque form a single
  // thread-safe triggered queue of tasks:
  std::mutex m;
  std::condition_variable v;
  // note that a packaged_task<void> can store a packaged_task<R>:
  std::deque<std::packaged_task<void()>> work;

  // this holds futures representing the worker threads being done:
  std::vector<std::future<void>> finished;

  // queue( lambda ) will enqueue the lambda into the tasks for the threads
  // to use.  A future of the type the lambda returns is given to let you get
  // the result out.
  template<class F, class R=std::result_of_t<F&()>>
  std::future<R> queue(F&& f) {
```

```
    // wrap the function object into a packaged task, splitting
    // execution from the return value:
    std::packaged_task<R()> p(std::forward<F>(f));

    auto r=p.get_future(); // get the return value before we hand off the task
    {
      std::unique_lock<std::mutex> l(m);
      work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
    }
    v.notify_one(); // wake a thread to work on the task

    return r; // return the future result of the task
  }

  // start N threads in the thread pool.
  void start(std::size_t N=1){
    for (std::size_t i = 0; i < N; ++i)
    {
      // each thread is a std::async running this->thread_task():
      finished.push_back(
        std::async(
          std::launch::async,
          [this]{ thread_task(); }
        )
      );
    }
  }
  // abort() cancels all non-started tasks, and tells every working thread
  // stop running, and waits for them to finish up.
  void abort() {
    cancel_pending();
    finish();
  }
  // cancel_pending() merely cancels all non-started tasks:
  void cancel_pending() {
    std::unique_lock<std::mutex> l(m);
    work.clear();
  }
  // finish enques a "stop the thread" message for every thread, then waits for them:
  void finish() {
    {
      std::unique_lock<std::mutex> l(m);
      for(auto&&unused:finished){
        work.push_back({});
      }
    }
    v.notify_all();
    finished.clear();
  }
  ~tasks() {
    finish();
  }
private:
  // the work that a worker thread does:
  void thread_task() {
    while(true){
      // pop a task off the queue:
      std::packaged_task<void()> f;
      {
        // usual thread-safe queue code:
        std::unique_lock<std::mutex> l(m);
```

```
        if (work.empty()){
          v.wait(l,[&]{return !work.empty();});
        }
        f = std::move(work.front());
        work.pop_front();
      }
      // if the task is invalid, it means we are asked to abort:
      if (!f.valid()) return;
      // otherwise, run the task:
      f();
    }
  }
};
```

`tasks.queue( []{ return "hello world"s; } )` renvoie un `std::future<std::string>` qui, lorsque l'objet tâche est exécuté, est rempli avec `hello world` .

Vous créez des threads en exécutant `tasks.start(10)` (qui démarre 10 threads).

L'utilisation de `packaged_task<void()>` est simplement due au fait qu'il n'y a pas d'équivalent `std::function` effacé par type qui stocke les types de déplacement uniquement. Écrire un fichier personnalisé serait probablement plus rapide que d'utiliser `packaged_task<void()>` .

Exemple en direct .

C ++ 11

En C ++ 11, remplacez `result_of_t<blah>` par `typename result_of<blah>::type` .

Plus sur Mutexes .

Stockage local

Le stockage local de thread peut être créé à l'aide du mot clé `thread_local` . Une variable déclarée avec le spécificateur `thread_local` est dite avoir **une durée de stockage de thread.**

- Chaque thread dans un programme a sa propre copie de chaque variable locale de thread.
- Une variable locale au thread avec une portée de fonction (locale) sera initialisée lors du premier passage du contrôle dans sa définition. Une telle variable est implicitement statique, sauf si déclarée `extern` .
- Une variable locale au thread avec un espace de nommage ou une étendue de classe (non locale) sera initialisée dans le cadre du démarrage du thread.
- Les variables thread-locales sont détruites à la fin du thread.
- Un membre d'une classe ne peut être local que s'il est statique. Il y aura donc une copie de cette variable par thread, plutôt qu'une copie par paire (thread, instance).

Exemple:

```
void debug_counter() {
    thread_local int count = 0;
    Logger::log("This function has been called %d times by this thread", ++count);
}
```

Lire Filetage en ligne: https://riptutorial.com/fr/cplusplus/topic/699/filetage

Chapitre 40: Flux C ++

Remarques

Le constructeur par défaut de `std::istream_iterator` construit un itérateur qui représente la fin du flux. Ainsi, `std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(), ....` signifie copier de la position actuelle dans `ifs` à la fin.

Examples

Flux de chaînes

`std::ostringstream` est une classe dont les objets ressemblent à un flux de sortie (vous pouvez leur écrire via un `operator<<`), mais stocke réellement les résultats de l'écriture et les fournit sous la forme d'un flux.

Considérez le code court suivant:

```
#include <sstream>
#include <string>


using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

La ligne

```
ostringstream ss;
```

crée un tel objet. Cet objet est d'abord manipulé comme un flux régulier:

```
ss << "the answer to everything is " << 42;
```

Après cela, le flux résultant peut être obtenu comme ceci:

```
const string result = ss.str();
```

(le `result` la chaîne sera égal à `"the answer to everything is 42"`).

Ceci est surtout utile lorsque nous avons une classe pour laquelle la sérialisation du flux a été définie et pour laquelle nous voulons une forme de chaîne. Par exemple, supposons que nous

ayons une classe

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

Pour obtenir la représentation sous forme de chaîne d'un objet `foo` ,

```
foo f;
```

nous pourrions utiliser

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Le `result` contient alors la représentation sous forme de chaîne de l'objet `foo` .

Lire un fichier jusqu'à la fin

Lecture d'un fichier texte ligne par ligne

Une manière appropriée de lire un fichier texte ligne par ligne jusqu'à la fin n'est généralement pas claire dans la documentation `ifstream` . Considérons quelques erreurs courantes commises par des programmeurs C ++ débutants, et une manière appropriée de lire le fichier.

Lignes sans caractères d'espacement

Par souci de simplicité, supposons que chaque ligne du fichier ne contient aucun symbole d'espacement.

`ifstream` a l' `operator bool()` , qui renvoie true lorsqu'un flux ne contient aucune erreur et est prêt à être lu. De plus, `ifstream::operator >>` renvoie une référence au flux lui-même, nous pouvons donc lire et vérifier EOF (ainsi que les erreurs) sur une seule ligne avec une syntaxe très élégante:

```
std::ifstream ifs("1.txt");
std::string s;
while(ifs >> s) {
    std::cout << s << std::endl;
}
```

Lignes avec des caractères d'espacement

`ifstream::operator >>` lit le flux jusqu'à ce qu'un caractère d' `ifstream::operator >>` apparaisse, le

code ci-dessus imprimera les mots d'une ligne sur des lignes séparées. Pour tout lire jusqu'à la fin de la ligne, utilisez `std::getline` au lieu de `ifstream::operator >>`. `getline` renvoie une référence au thread avec lequel il a travaillé, donc la même syntaxe est disponible:

```
while(std::getline(ifs, s)) {
    std::cout << s << std::endl;
}
```

De toute évidence, `std::getline` devrait également être utilisé pour lire un fichier d'une seule ligne jusqu'à la fin.

Lecture d'un fichier dans un tampon à la fois

Enfin, lisons le fichier du début à la fin sans arrêter aucun caractère, y compris les espaces blancs et les nouvelles lignes. Si nous savons que la taille exacte du fichier ou la limite supérieure de la longueur est acceptable, nous pouvons redimensionner la chaîne et lire ensuite:

```
s.resize(100);
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    s.begin());
```

Sinon, il faut insérer chaque caractère à la fin de la chaîne, donc `std::back_inserter` est ce dont nous avons besoin:

```
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::back_inserter(s));
```

Alternativement, il est possible d'initialiser une collection avec des données de flux, en utilisant un constructeur avec des arguments de plage d'itérateurs:

```
std::vector v(std::istreambuf_iterator<char>(ifs),
    std::istreambuf_iterator<char>());
```

Notez que ces exemples sont également applicables si `ifs` est ouvert en tant que fichier binaire:

```
std::ifstream ifs("1.txt", std::ios::binary);
```

Copier des flux

Un fichier peut être copié dans un autre fichier avec des flux et des itérateurs:

```
std::ofstream ofs("out.file");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(ofs));
ofs.close();
```

ou redirigé vers un autre type de flux avec une interface compatible. Par exemple flux de réseau Boost.Asio:

```
boost::asio::ip::tcp::iostream stream;
stream.connect("example.com", "http");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(stream));
stream.close();
```

Tableaux

Comme les itérateurs peuvent être considérés comme une généralisation de pointeurs, les conteneurs STL dans les exemples ci-dessus peuvent être remplacés par des tableaux natifs. Voici comment analyser les nombres dans un tableau:

```
int arr[100];
std::copy(std::istream_iterator<char>(ifs), std::istream_iterator<char>(), arr);
```

Méfiez-vous du débordement de la mémoire tampon, car les tableaux ne peuvent pas être redimensionnés à la volée après leur attribution. Par exemple, si le code ci-dessus est alimenté par un fichier contenant plus de 100 nombres entiers, il essaiera d'écrire en dehors du tableau et aura un comportement indéfini.

Impression de collections avec iostream

Impression de base

`std::ostream_iterator` permet d'imprimer le contenu d'un conteneur STL sur n'importe quel flux de sortie sans boucles explicites. Le second argument du constructeur `std::ostream_iterator` définit le délimiteur. Par exemple, le code suivant:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

imprimera

```
1 ! 2 ! 3 ! 4 !
```

Type de distribution implicite

`std::ostream_iterator` permet de transtyper implicitement le type de contenu du conteneur. Par exemple, accordons `std::cout` pour imprimer des valeurs à virgule flottante avec 3 chiffres après le point décimal:

```
std::cout << std::setprecision(3);
std::fixed(std::cout);
```

et instancier `std::ostream_iterator` avec `float` , alors que les valeurs contenues restent `int` :

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

donc le code ci-dessus donne

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

malgré `std::vector` tient `int` s.

Génération et transformation

`std::generate` , `std::generate_n` et `std::transform` fournissent un outil très puissant pour la manipulation des données à la volée. Par exemple, avoir un vecteur:

```
std::vector<int> v = {1,2,3,4,8,16};
```

nous pouvons facilement imprimer la valeur booléenne de l'instruction "x is even" pour chaque élément:

```
std::boolalpha(std::cout); // print booleans alphabetically
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),
[](int val) {
    return (val % 2) == 0;
});
```

ou imprimer l'élément carré:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Impression de N nombres séparés par des espaces:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Tableaux

Comme dans la section sur la lecture des fichiers texte, presque toutes ces considérations peuvent être appliquées aux tableaux natifs. Par exemple, imprimons les valeurs au carré d'un

tableau natif:

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Fichiers d'analyse

Analyse des fichiers dans des conteneurs STL

`istream_iterator` s sont très utiles pour lire des séquences de nombres ou d'autres données analysables dans des conteneurs STL sans boucles explicites dans le code.

En utilisant une taille de conteneur explicite:

```
std::vector<int> v(100);
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    v.begin());
```

ou en insérant un itérateur:

```
std::vector<int> v;
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    std::back_inserter(v));
```

Notez que les nombres dans le fichier d'entrée peuvent être divisés par un nombre quelconque de caractères d'espacement et de nouvelles lignes.

Analyse de tables de texte hétérogènes

Au `istream::operator>>` mesure que `istream::operator>>` lit le texte jusqu'à un symbole d' `istream::operator>>` , il peut être utilisé dans une condition `while` pour analyser des tables de données complexes. Par exemple, si nous avons un fichier avec deux nombres réels suivis d'une chaîne (sans espaces) sur chaque ligne:

```
1.12 3.14 foo
2.1 2.2 barr
```

il peut être analysé comme ceci:

```
std::string s;
double a, b;
while(ifs >> a >> b >> s) {
```

```
    std::cout << a << " " << b << " " << s << std::endl;
}
```

Transformation

Toute fonction de manipulation de plage peut être utilisée avec les plages `std::istream_iterator`. L'un d'eux est `std::transform`, qui permet de traiter des données à la volée. Par exemple, lisons les valeurs entières, multiplions-les par 3.14 et stockez le résultat dans un conteneur à virgule flottante:

```
std::vector<double> v(100);
std::transform(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
v.begin(),
[](int val) {
    return val * 3.14;
});
```

Lire Flux C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/7660/flux-c-plusplus

Chapitre 41: Fonction C ++ "appel par valeur" vs. "appel par référence"

Introduction

Le but de cette section est d'expliquer les différences de théorie et de mise en œuvre pour ce qui se passe avec les paramètres d'une fonction lors de l'appel.

En détail, les paramètres peuvent être considérés comme des variables avant l'appel de la fonction et à l'intérieur de la fonction, où le comportement visible et l'accessibilité à ces variables diffèrent de la méthode utilisée pour les transmettre.

En outre, cette rubrique explique également la réutilisation des variables et leurs valeurs respectives après l'appel de la fonction.

Examples

Appel par valeur

Lors de l'appel d'une fonction, de nouveaux éléments sont créés sur la pile du programme. Celles-ci incluent des informations sur la fonction et l'espace (emplacements de mémoire) pour les paramètres et la valeur de retour.

Lors de la transmission d'un paramètre à une fonction, la valeur de la variable utilisée (ou littérale) est copiée dans l'emplacement mémoire du paramètre de fonction. Cela implique que maintenant il y a deux emplacements de mémoire avec la même valeur. À l'intérieur de la fonction, nous travaillons uniquement sur l'emplacement de la mémoire de paramètres.

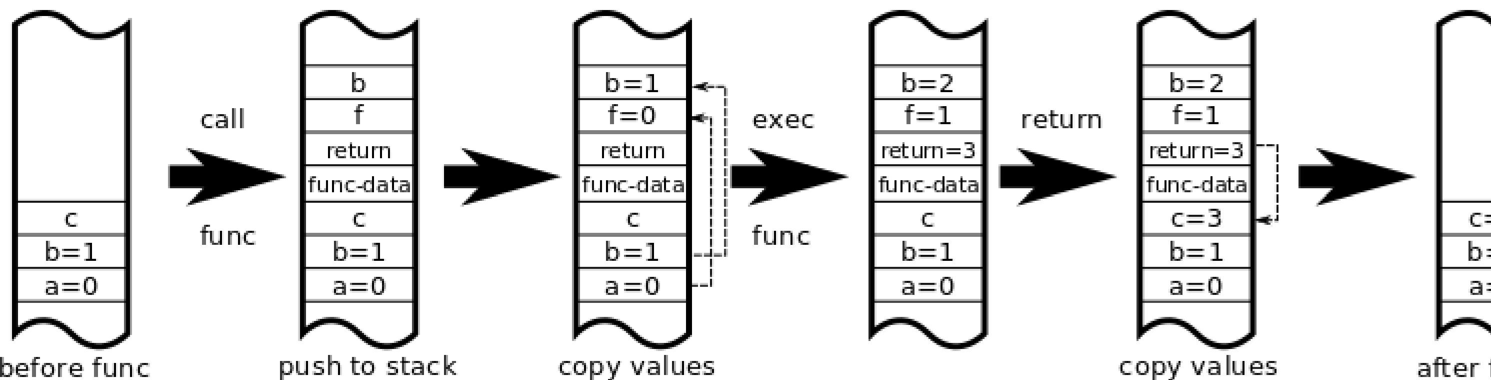
Après avoir quitté la fonction, la mémoire de la pile de programmes est supprimée (supprimée), ce qui efface toutes les données de l'appel de fonction, y compris l'emplacement mémoire des paramètres utilisés à l'intérieur. Ainsi, les valeurs modifiées dans la fonction n’affectent pas les valeurs des variables externes.

```
int func(int f, int b) {
  //new variables are created and values from the outside copied
  //f has a value of 0
  //inner_b has a value of 1
  f = 1;
  //f has a value of 1
  b = 2;
  //inner_b has a value of 2
  return f+b;
}

int main(void) {
  int a = 0;
  int b = 1; //outer_b
```

```
    int c;

    c = func(a,b);
    //the return value is copied to c

    //a has a value of 0
    //outer_b has a value of 1   <--- outer_b and inner_b are different variables
    //c has a value of 3
}
```

Dans ce code, nous créons des variables dans la fonction principale. Ceux-ci reçoivent des valeurs assignées. Lors de l'appel des fonctions, deux nouvelles variables ont été créées: `f` et `inner_b` où `b` partage le nom avec la variable externe dont il ne partage pas l'emplacement mémoire. Le comportement d' `a<->f` et `b<->b` est identique.

Le graphique suivant symbolise ce qui se passe sur la pile et pourquoi il n'y a pas de changement dans varibale `b` . Le graphique n'est pas tout à fait exact mais souligne l'exemple.



On l'appelle "call by value" car on ne remet pas les variables mais seulement les valeurs de ces variables.

Lire Fonction C ++ "appel par valeur" vs. "appel par référence" en ligne: https://riptutorial.com/fr/cplusplus/topic/10669/fonction-c-plusplus--appel-par-valeur--vs---appel-par-reference-

Chapitre 42: Fonction de surcharge

Introduction

Voir aussi la rubrique séparée sur la résolution de la surcharge

Remarques

Des ambiguïtés peuvent se produire lorsqu'un type peut être implicitement converti en plusieurs types et qu'il n'y a pas de fonction correspondante pour ce type spécifique.

Par exemple:

```
void foo(double, double);
void foo(long, long);

//Call foo with 2 ints
foo(1, 2); //Function call is ambiguous - int can be converted into a double/long at the same
time
```

Examples

Qu'est-ce que la surcharge de fonctions?

La surcharge de fonctions fait en sorte que plusieurs fonctions déclarées dans la même étendue avec exactement le même nom existent au même endroit (appelé *scope*) et ne diffèrent que par leur *signature* , ce qui signifie les arguments qu'elles acceptent.

Supposons que vous écrivez une série de fonctions pour des capacités d’impression généralisées, en commençant par `std::string` :

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

Cela fonctionne bien, mais supposons que vous vouliez une fonction qui accepte également un `int` et l'imprime également. Vous pourriez écrire:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

Mais comme les deux fonctions acceptent des paramètres différents, vous pouvez simplement écrire:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Maintenant, vous avez 2 fonctions, toutes deux nommées `print` , mais avec des signatures différentes. L'un accepte `std::string` , l'autre un `int` . Maintenant, vous pouvez les appeler sans vous soucier des noms différents:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Au lieu de:

```
print("Hello world!");
print_int(1337);
```

Lorsque vous avez des fonctions surchargées, le compilateur déduit quelles fonctions appeler à partir des paramètres que vous lui avez fournis. Des précautions doivent être prises lors de l'écriture de surcharges de fonctions. Par exemple, avec les conversions de types implicites:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Maintenant, il n'est pas immédiatement clair quelle surcharge d' `print` est appelée lorsque vous écrivez:

```
print(5);
```

Et vous pourriez avoir besoin de donner à votre compilateur des indices, tels que:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

Des précautions doivent également être prises lors de l'écriture de surcharges acceptant des paramètres facultatifs:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
```

```
    std::cout << "This is an int: " << num << std::endl;
}
```

Comme il n'y a aucun moyen pour le compilateur de dire si un appel comme `print(17)` est destiné à la première ou à la deuxième fonction à cause du deuxième paramètre facultatif, la compilation échouera.

Type de retour en surcharge de fonction

Notez que vous ne pouvez pas surcharger une fonction en fonction de son type de retour. Par exemple:

```
// WRONG CODE
std::string getValue()
{
  return "hello";
}

int getValue()
{
  return 0;
}

int x = getValue();
```

Cela provoquera une erreur de compilation car le compilateur ne pourra pas déterminer quelle version de `getValue` appeler, même si le type de retour est assigné à un `int` .

Membre Fonction cv-qualifier Surcharge

Les fonctions d'une classe peuvent être surchargées lorsqu'elles sont accessibles via une référence qualifiée CV à cette classe; Ceci est le plus souvent utilisé pour surcharger `const` , mais peut également être utilisé pour surcharger des `volatile` et `const volatile` . En effet, toutes les fonctions membres non statiques prennent `this` comme paramètre caché, auquel les qualificateurs cv sont appliqués. Ceci est le plus couramment utilisé pour surcharger `const` , mais peut également être utilisé pour `volatile` et `const volatile` .

Cela est nécessaire car une fonction membre ne peut être appelée que si elle est au moins qualifiée cv par l'instance à laquelle elle est appelée. Alors qu'une instance non- `const` peut appeler des membres `const` et non- `const` , une instance `const` ne peut appeler que des membres `const` . Cela permet à une fonction d'avoir un comportement différent en fonction des qualificateurs cv de l'instance appelante et permet au programmeur d'interdire les fonctions pour un ou plusieurs qualificatifs cv indésirables en ne fournissant pas de version avec ce ou ces qualificatifs.

Une classe avec une méthode d' `print` base pourrait être surchargée en `const` comme suit:

```
#include <iostream>

class Integer
{
    public:
```

```
        Integer(int i_): i{i_}{}

        void print()
        {
            std::cout << "int: " << i << std::endl;
        }

        void print() const
        {
            std::cout << "const int: " << i << std::endl;
        }

    protected:
        int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

C'est un principe clé de la correction des `const` : En marquant les fonctions membres comme `const` , elles peuvent être appelées sur des instances `const` , ce qui permet aux fonctions de prendre des instances en tant que pointeurs / références `const` si elles n'ont pas besoin de les modifier. Cela permet au code de spécifier s'il modifie l'état en prenant des paramètres non modifiés comme `const` et des paramètres modifiés sans qualificatifs cv, rendant le code à la fois plus sûr et plus lisible.

```
class ConstCorrect
{
  public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." <<
std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Error.  Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Good.  Can only be called from non-const instance.
```

```
}
```

Une utilisation courante de ceci est la déclaration des accesseurs en tant que `const` et les mutateurs en tant que non `const` .

Aucun membre de classe ne peut être modifié dans une fonction membre `const` . S'il y a un membre à modifier, comme le verrouillage d'un `std::mutex` , vous pouvez le déclarer `mutable` :

```
class Integer
{
    public:
        Integer(int i_): i{i_}{}

        int get() const
        {
            std::lock_guard<std::mutex> lock{mut};
            return i;
        }

        void set(int i_)
        {
            std::lock_guard<std::mutex> lock{mut};
            i = i_;
        }

    protected:
        int i;
        mutable std::mutex mut;
};
```

Lire Fonction de surcharge en ligne: https://riptutorial.com/fr/cplusplus/topic/510/fonction-de-surcharge

Chapitre 43: Fonctions en ligne

Introduction

Une fonction définie avec le spécificateur en `inline` est une fonction en ligne. Une fonction en ligne peut être définie de manière multiple sans violer la règle de définition unique et peut donc être définie dans un en-tête avec un lien externe. Déclarer une fonction inline indique au compilateur que la fonction doit être intégrée lors de la génération du code, mais ne fournit aucune garantie.

Syntaxe

- *fonction_déclaration* **`inline`**
- **`inline`** *function_definition*
- *classe {définition_fonction};*

Remarques

Habituellement, si le code généré pour une fonction est *suffisamment* petit, c'est un bon candidat pour être intégré. Pourquoi donc Si une fonction est grande et intégrée dans une boucle, pour tous les appels effectués, le code de la grande fonction serait dupliqué, ce qui conduirait à la taille binaire générée. Mais quelle est la taille suffisante?

Bien que les fonctions en ligne semblent être un excellent moyen d'éviter les appels de fonction, il convient de noter que toutes les fonctions marquées en `inline` sont pas toutes intégrées. En d'autres termes, lorsque vous dites en `inline` , ce n'est qu'un indice pour le compilateur, pas un ordre: le compilateur n'est pas obligé d'inclure la fonction, il est libre de l'ignorer - la plupart le font. Les compilateurs modernes sont mieux à même de faire de telles optimisations que ce mot-clé est maintenant un vestige du passé, lorsque les compilateurs ont pris cette suggestion au sérieux. Même les fonctions qui ne sont pas marquées en `inline` sont insérées par le compilateur lorsque cela s'avère avantageux.

Inline comme directive de liaison

L'utilisation plus pratique de l' `inline` dans le C ++ moderne provient de son utilisation comme directive de liaison. Lors de la *définition* , et non de la déclaration, d'une fonction dans un en-tête qui sera incluse dans plusieurs sources, chaque unité de traduction aura sa propre copie de cette fonction entraînant une violation ODR (One Definition Rule); Cette règle dit grossièrement qu'il ne peut y avoir qu'une seule définition d'une fonction, d'une variable, etc. Pour contourner cette violation, le marquage de la définition de fonction en `inline` fait implicitement le lien de fonction interne.

FAQ

Quand dois-je écrire le mot-clé 'inline' pour une fonction / méthode en C ++?

Seulement lorsque vous souhaitez que la fonction soit définie dans un en-tête. Plus exactement lorsque la définition de la fonction peut apparaître dans plusieurs unités de compilation. C'est une bonne idée de définir des fonctions de petite taille (comme dans une ligne) dans le fichier d'en-tête, car cela donne au compilateur plus d'informations pour travailler tout en optimisant votre code. Cela augmente également le temps de compilation.

Quand ne devrais-je pas écrire le mot-clé 'inline' pour une fonction / méthode en C ++?

N'ajoutez pas de `inline` lorsque vous pensez que votre code s'exécutera plus rapidement si le compilateur l'a intégré.

Quand le compilateur ne saura-t-il pas quand une fonction / méthode doit être intégrée?

Généralement, le compilateur sera en mesure de le faire mieux que vous. Cependant, le compilateur n'a pas la possibilité d'insérer du code s'il ne possède pas la définition de la fonction. Dans le code optimisé au maximum, toutes les méthodes privées sont généralement intégrées, que vous le demandiez ou non.

Voir également

- Quand dois-je écrire le mot-clé 'inline' pour une fonction / méthode?
- Y a-t-il encore une utilisation pour inline?

Examples

Déclaration de fonction en ligne non membre

```
inline int add(int x, int y);
```

Définition de fonction inline non membre

```
inline int add(int x, int y)
{
    return x + y;
}
```

Fonctions inline membres

```
// header (.hpp)
struct A
```

```
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

Qu'est-ce que la fonction inline?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

Dans le code ci-dessus, lorsque l' `add` est en ligne, le code résultant deviendrait quelque chose comme ça

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

La fonction inline est introuvable, son corps est *inséré* dans le corps de l'appelant. Si `add` n'avait pas été intégré, une fonction serait appelée. La surcharge liée à l'appel d'une fonction, telle que la création d'un nouveau frame de pile , la copie d'arguments, la création de variables locales, le saut (perte de localisation de référence et absence de cache), doit être engagée.

Lire Fonctions en ligne en ligne: https://riptutorial.com/fr/cplusplus/topic/7150/fonctions-en-ligne

Chapitre 44: Fonctions membres de classe constante

Remarques

Que signifie vraiment les fonctions de membre const d'une classe? La définition simple semble être que, une fonction membre const ne peut pas changer l'objet. Mais ce qui ne peut pas changer signifie vraiment ici. Cela signifie simplement que vous ne pouvez pas faire d'affectation pour les membres de données de classe.

Cependant, vous pouvez effectuer d'autres opérations indirectes, telles que l'insertion d'une entrée dans une carte, comme indiqué dans l'exemple. Permettre que cela ressemble à cette fonction const modifie l'objet (oui, dans un sens), mais c'est permis.

Donc, la véritable signification est qu'une fonction membre const ne peut pas faire une affectation pour les variables de données de classe. Mais il peut faire d'autres choses comme expliqué dans l'exemple.

Examples

fonction membre constante

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
         (*mapOfStrings)[key] = value;             // This works? Yes it does.
        delete mapOfStrings;                       // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
```

```
};

int main(int argc, char* argv[]) {

    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Lire Fonctions membres de classe constante en ligne:
https://riptutorial.com/fr/cplusplus/topic/7120/fonctions-membres-de-classe-constante

Chapitre 45: Fonctions membres non statiques

Syntaxe

- // appelant:
  - variable.member_function ();
  - variable_pointer-> member_function ();
- // Définition:
  - ret_type class_name :: member_function () cv-qualifiers {
    - corps;
  - }
- // Prototype:
  - class class_name {
    - spécificateur virtuel ret_type member_function () cv-qualifiers virt-specifier-seq;
    - // virt-specifier: "virtuel", le cas échéant.
    - // qualificatifs-cv: "const" et / ou "volatile", le cas échéant.
    - // virt-specifier-seq: "override" et / ou "final", le cas échéant.
  - }

Remarques

Une fonction membre non `static` est une fonction membre `class` / `struct` / `union` , appelée sur une instance particulière et opérant sur cette instance. Contrairement `static` fonctions membres `static` , il ne peut pas être appelé sans spécifier d'instance.

Pour plus d'informations sur les classes, les structures et les unions, consultez le sujet parent .

Examples

Fonctions de membres non statiques

Une `class` ou une `struct` peut avoir des fonctions membres ainsi que des variables membres. Ces fonctions ont une syntaxe similaire à celle des fonctions autonomes et peuvent être définies à l'intérieur ou à l'extérieur de la définition de la classe. S'il est défini en dehors de la définition de la classe, le nom de la fonction est précédé du nom de la classe et de l'opérateur scope (`::` :).

```
class CL {
  public:
    void  definedInside() {}
```

```
    void definedOutside();
};
void CL::definedOutside() {}
```

Ces fonctions sont appelées sur une instance (ou référence à une instance) de la classe avec l'opérateur point (`.`), Ou un pointeur sur une instance avec l'opérateur arrow (`->`), et chaque appel est lié à l'instance de la fonction a été appelé; Lorsqu'une fonction membre est appelée sur une instance, elle a accès à tous les champs de cette instance (via `this` pointeur), mais ne peut accéder qu'aux champs des autres instances si ces instances sont fournies en tant que paramètres.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

  private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int  i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

Ces fonctions sont autorisées à accéder aux variables membres et / ou à d'autres fonctions membres, indépendamment des modificateurs d'accès de la variable ou de la fonction. Ils peuvent également être écrits dans le désordre, accéder aux variables membres et / ou aux fonctions membres appelantes déclarées avant eux, car la définition de classe entière doit être analysée avant que le compilateur puisse commencer à compiler une classe.

```
class Access {
  public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
  protected:
    int j;
    int get_i() const { return i; }
  private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

Encapsulation

Une utilisation courante des fonctions membres est l'encapsulation, utilisant un *accesseur* (communément appelé getter) et un *mutateur* (communément appelé setter) au lieu d'accéder

directement aux champs.

```
class Encapsulator {
    int encapsulated;

  public:
    int  get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e)  { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

À l'intérieur de la classe, n'importe quelle fonction membre non statique peut accéder librement à l' `encapsulated` . en dehors de la classe, l'accès à celle-ci est régulé par les fonctions membres, en utilisant `get_encapsulated()` pour le lire et `set_encapsulated()` pour le modifier. Cela évite les modifications involontaires de la variable, car des fonctions distinctes sont utilisées pour la lire et l'écrire. [Il y a beaucoup de discussions pour savoir si les getters et les installateurs fournissent ou cassent l'encapsulation, avec de bons arguments pour les deux revendications; un tel débat est hors de portée de cet exemple.]

Nom Cacher et importer

Lorsqu'une classe de base fournit un ensemble de fonctions surchargées et qu'une classe dérivée ajoute une autre surcharge à l'ensemble, cela masque toutes les surcharges fournies par la classe de base.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1);    // Output:  int
hb.f(true); // Output:  bool
hb.f(s);    // Output:  std::string;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  float
hd.f(true); // Output:  float
hd.f(s);    // Error: Can't convert from std::string to float.
```

Ceci est dû aux règles de résolution des noms: Lors de la recherche de nom, une fois le nom correct trouvé, nous ne cherchons plus, même si nous n'avons clairement pas trouvé la *version*

correcte de l'entité avec ce nom (avec `hd.f(s)` par `hd.f(s)`); De ce fait, la surcharge de la fonction dans la classe dérivée empêche la recherche de nom de découvrir les surcharges dans la classe de base. Pour éviter cela, une déclaration d'utilisation peut être utilisée pour "importer" des noms de la classe de base dans la classe dérivée, afin qu'ils soient disponibles lors de la recherche de nom.

```
struct HidingDerived : HiddenBase {
     // All members named HiddenBase::f shall be considered members of HidingDerived for
lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  int
hd.f(true); // Output:  bool
hd.f(s);    // Output:  std::string
```

Si une classe dérivée importe des noms avec une déclaration using, mais déclare également des fonctions avec la même signature que les fonctions de la classe de base, les fonctions de classe de base seront ignorées ou masquées.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()    {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

Une déclaration using peut également être utilisée pour modifier les modificateurs d'accès, à condition que l'entité importée soit `public` ou `protected` dans la classe de base.

```
struct ProMem {
  protected:
    void func() {}
};

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
```

```
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.
```

De même, si nous voulons explicitement appeler une fonction membre à partir d'une classe spécifique dans la hiérarchie d'héritage, nous pouvons qualifier le nom de la fonction lors de l'appel de la fonction, en spécifiant cette classe par son nom.

```
struct One {
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.
```

Fonctions membres virtuelles

Les fonctions membres peuvent également être déclarées `virtual` . Dans ce cas, si elles sont appelées sur un pointeur ou une référence à une instance, elles ne seront pas directement accessibles. ils rechercheront plutôt la fonction dans la table des fonctions virtuelles (une liste de fonctions pointeur vers membre pour les fonctions virtuelles, plus communément appelée `vtable` ou `vftable`), et l'utiliseront pour appeler la version adaptée à la dynamique de l'instance. type (réel) Si la fonction est appelée directement, à partir d'une variable d'une classe, aucune recherche n'est effectuée.

```
struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }
```

```
// ...

Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.

b.func();   // Output:  In Base.
d.func();   // Output:  In Derived.

pb->func(); // Output:  In Base.
pd->func(); // Output:  In Derived.

rb.func();  // Output:  In Base.
rd.func();  // Output:  In Derived.

slicer(b);  // Output:  In Base.
slicer(d);  // Output:  In Base.
```

Notez que si `pd` est `Base*` et que `rd` est une `Base&` , appeler `func()` sur l'un des deux appels `Derived::func()` au lieu de `Base::func()` ; C'est parce que la `vtable` pour `Derived` met à jour l'entrée `Base::func()` pour indiquer plutôt `Derived::func()` . À l'inverse, notez que le fait de passer une instance à `slicer()` entraîne toujours l'appel de `Base::func()` , même si l'instance transmise est un `Derived` . Cela est dû à quelque chose appelé *découpage de données* , où le fait de passer une instance `Derived` dans un paramètre `Base` par valeur rend la partie de l'instance `Derived` qui n'est pas une instance de `Base` inaccessible.

Lorsqu'une fonction membre est définie comme étant virtuelle, toutes les fonctions membres de classe dérivées ayant la même signature la remplacent, que la fonction de substitution soit spécifiée comme `virtual` ou non. Cela peut rendre les classes dérivées plus difficiles à analyser pour les programmeurs, car il n'y a aucune indication sur la ou les fonctions `virtual` .

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                //  You'd have to check B to know that, though.
};
```

Notez cependant qu'une fonction dérivée ne remplace une fonction de base que si leurs signatures correspondent; même si une fonction dérivée est explicitement déclarée `virtual` , elle créera à la place une nouvelle fonction virtuelle si les signatures ne correspondent pas.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

C ++ 11

A partir de C ++ 11, l'intention de remplacer peut être explicite avec le `override` mot clé contextuel. Cela indique au compilateur que le programmeur s'attend à ce qu'il remplace une fonction de classe de base, ce qui oblige le compilateur à omettre une erreur si elle *ne* remplace rien.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

Cela a également l'avantage de dire aux programmeurs que la fonction est à la fois virtuelle et déclarée dans au moins une classe de base, ce qui peut faciliter l'analyse des classes complexes.

Lorsqu'une fonction est déclarée `virtual` et définie en dehors de la définition de la classe, le spécificateur `virtual` doit être inclus dans la déclaration de la fonction et non répété dans la définition.

C ++ 11

Cela est également vrai pour le `override` .

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

Si une classe de base surcharge une fonction `virtual` , seules les surcharges explicitement spécifiées comme `virtual` seront virtuelles.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

Pour plus d'informations, voir le sujet pertinent .

Correct Correct

L'une des principales utilisations de `this` qualificateurs cv est la *correction de* `const` . C'est la pratique de garantir que seuls les accès ayant *besoin* de modifier un objet *peuvent* modifier l'objet, et que toute fonction (membre ou non-membre) n'ayant pas besoin de modifier un objet n'a pas accès en écriture à cet objet. objet (directement ou indirectement). Cela empêche les modifications involontaires, rendant le code moins erreurprone. Il permet également à toute fonction qui n'a pas besoin de modifier l'état de prendre un objet `const` ou non `const` , sans avoir à réécrire ou à surcharger la fonction.

`const` correction de `const` , de par sa nature, commence par le bas: toute fonction de classe qui n'a pas besoin de changer d'état est déclarée comme `const` , de sorte qu'elle puisse être appelée sur des instances `const` . Ceci, à son tour, permet aux paramètres passés par référence d'être déclarés `const` quand ils n'ont pas besoin d'être modifiés, ce qui permet aux fonctions de prendre des objets `const` ou non `const` sans se plaindre, et `const` -ness peut se propager vers l'extérieur manière. De ce fait, les getters sont souvent `const` , de même que toute autre fonction qui n'a pas besoin de modifier l'état logique.

```
class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(const Field& f) : fld(f) {}     // Modifies.

    const Field& get_field()       { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; }    // Modifies.

    void do_something(int i) {                     // Modifies.
        fld.insert_value(i);
    }
    void do_nothing()        { }                   // Doesn't modify; should be const.
};

class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f) : fld(f) {}       // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }    // Not const: Modifies.

    void do_something(int i) {                     // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const  { }                   // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
  // Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
  // Error: Same as above.
// Oops.
```

```
const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();          // Good.
```

Comme illustré par les commentaires sur `ConstIncorrect` et `ConstCorrect` , les fonctions qualifiant cv servent également de documentation. Si une classe est `const` correcte, une fonction qui ne `const` peut supposer sans risque de changer d' état, et toute fonction qui est `const` peut supposer sans risque de ne pas changer d' état.

Lire Fonctions membres non statiques en ligne:
https://riptutorial.com/fr/cplusplus/topic/5661/fonctions-membres-non-statiques

Chapitre 46: Fonctions membres spéciales

Examples

Destructeurs virtuels et protégés

Une classe conçue pour être héritée de s'appelle une classe de base. Des précautions doivent être prises avec les fonctions membres spéciales d'une telle classe.

Une classe conçue pour être utilisée de manière polymorphe à l'exécution (via un pointeur sur la classe de base) doit déclarer le destructeur `virtual` . Cela permet aux parties dérivées de l'objet d'être correctement détruites, même lorsque l'objet est détruit via un pointeur vers la classe de base.

```
class Base {
public:
    virtual ~Base() = default;

private:
    //    data members etc.
};

class Derived : public Base { //  models Is-A relationship
public:
    //    some methods

private:
    //    more data members
};

//    virtual destructor in Base ensures that derived destructors
//    are also called when the object is destroyed
std::unique_ptr<Base> base = std::make_unique<Derived>();
base = nullptr;  //    safe, doesn't leak Derived's members
```

Si la classe n'a pas besoin d'être polymorphe, mais doit toujours permettre à son interface d'être héritée, utilisez un destructeur `protected` non virtuel.

```
class NonPolymorphicBase {
public:
    //    some methods

protected:
    ~NonPolymorphicBase() = default; //    note: non-virtual

private:
    //    etc.
};
```

Une telle classe ne peut jamais être détruite par un pointeur, évitant les fuites silencieuses dues au découpage.

Cette technique s'applique particulièrement aux classes conçues pour être `private` classes de base `private` . Une telle classe pourrait être utilisée pour encapsuler certains détails d'implémentation courants, tout en fournissant `virtual` méthodes `virtual` tant que points de personnalisation. Ce type de classe ne devrait jamais être utilisé de manière polymorphe, et un destructeur `protected` aide à documenter cette exigence directement dans le code.

Enfin, certaines classes peuvent exiger qu'elles *ne* soient *jamais* utilisées comme classe de base. Dans ce cas, la classe peut être marquée comme `final` . Un destructeur public non virtuel normal est correct dans ce cas.

```
class FinalClass final {  //    marked final here
public:
    ~FinalClass() = default;

private:
    //    etc.
};
```

Déplacement et copie implicites

Gardez à l'esprit que la déclaration d'un destructeur empêche le compilateur de générer des constructeurs de déplacements implicites et de déplacer des opérateurs d'affectation. Si vous déclarez un destructeur, n'oubliez pas d'ajouter également les définitions appropriées pour les opérations de déplacement.

De plus, la déclaration des opérations de déplacement supprime la génération des opérations de copie, elles doivent donc être ajoutées (si les objets de cette classe doivent avoir une sémantique de copie).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Copier et échanger

Si vous écrivez une classe qui gère des ressources, vous devez implémenter toutes les fonctions membres spéciales (voir Règle de trois / cinq / zéro). L'approche la plus directe pour écrire le constructeur de copie et l'opérateur d'assignation serait:

```
person(const person &other)
```

```
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

Mais cette approche a quelques problèmes. Il ne la garantie forte d'exception - si `new[]` lancers francs, nous avons déjà éclairci les ressources appartenant à `this` et ne peut pas récupérer. Nous dupliquons une grande partie de la logique de la construction de copies dans l’affectation de copies. Et nous devons nous souvenir de la vérification de l’auto-assignation, qui ne fait généralement qu’ajouter de la charge à l’opération de copie, mais qui reste essentielle.

Pour satisfaire la garantie d'exception forte et éviter la duplication de code (doubler avec l'opérateur d'affectation de mouvement suivant), nous pouvons utiliser l'idiome de copie et d'échange:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

Pourquoi ça marche? Considérez ce qui se passe quand nous avons

```
person p1 = ...;
person p2 = ...;
p1 = p2;
```

Tout d'abord, nous copions- `rhs` partir de `p2` (que nous n'avons pas eu à dupliquer ici). Si cette opération est lancée, nous ne faisons rien dans `operator=` et `p1` reste intact. Ensuite, nous échangeons les membres entre `*this` et `rhs` , puis `rhs` sort de la portée. Lorsque l' `operator=` , qui

nettoie implicitement les ressources d' origine de `this` (via le destructor, que nous ne devions pas faire double emploi). L'auto-assignation fonctionne aussi - elle est moins efficace avec les opérations de copie-échange (qui impliquent une allocation et une désallocation supplémentaires), mais si c'est le cas peu probable, nous ne ralentissons pas le cas d'utilisation type pour le prendre en compte.

C ++ 11

La formulation ci-dessus fonctionne comme si elle était déjà utilisée pour une affectation de déplacement.

```
p1 = std::move(p2);
```

Ici, on déplace-construit `rhs` partir de `p2` , et tout le reste est tout aussi valide. Si une classe est déplaçable mais non copiable, il n'est pas nécessaire de supprimer l'assignation de copie, car cet opérateur d'affectation sera simplement mal formé en raison du constructeur de copie supprimé.

Constructeur par défaut

Un *constructeur par défaut* est un type de constructeur qui ne nécessite aucun paramètre lorsqu'il est appelé. Il est nommé d'après le type qu'il construit et en est une fonction membre (comme tous les constructeurs).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Une autre façon de satisfaire à l'exigence "pas de paramètres" est que le développeur fournisse des valeurs par défaut pour tous les paramètres:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};
```

```
D d; // calls constructor of D with the provided default values for the parameters
```

Dans certaines circonstances (c'est-à-dire que le développeur ne fournit aucun constructeur et qu'il n'y a pas d'autres conditions de disqualification), le compilateur fournit implicitement un constructeur par défaut vide:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Avoir un autre type de constructeur est l’une des conditions de disqualification mentionnées précédemment:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};

C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

c ++ 11

Pour empêcher la création implicite de constructeur par défaut, une technique courante consiste à la déclarer `private` (sans définition). L'intention est de provoquer une erreur de compilation lorsque quelqu'un essaie d'utiliser le constructeur (cela entraîne un *accès à une* erreur *privée* ou une erreur de l'éditeur de liens, selon le compilateur).

Pour être sûr qu'un constructeur par défaut (fonctionnellement similaire à l'implicite) est défini, un développeur pourrait en écrire un explicitement vide.

c ++ 11

En C ++ 11, un développeur peut également utiliser le mot-clé `delete` pour empêcher le compilateur de fournir un constructeur par défaut.

```
class C{
    int i;
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

En outre, un développeur peut également être explicite quant à la volonté du compilateur de fournir un constructeur par défaut.

```
class C{
    int i;
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;

    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

c ++ 14

Vous pouvez déterminer si un type a un constructeur par défaut (ou est un type primitif) en utilisant `std::is_default_constructible` partir de `<type_traits>` :

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

c ++ 11

En C ++ 11, il est toujours possible d'utiliser la version non-foncteur de `std::is_default_constructible` :

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Destructeur

Un *destructeur* est une fonction sans arguments appelée lorsqu'un objet défini par l'utilisateur est sur le point d'être détruit. Il est nommé d'après le type qu'il détruit avec un préfixe `~` .

```
class C{
    int* is;
    string s;
public:
    C()
    : is( new int[10] ){
    }

    ~C(){  // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
```

```
public:
    C_child(){}
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch;  // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

Dans la plupart des cas (c.-à-d. Qu'un utilisateur ne fournit aucun destructeur et qu'il n'y a pas d'autres conditions disqualifiantes), le compilateur fournit implicitement un destructeur par défaut:

```
class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
```

c ++ 11

En C ++ 11, un développeur peut remplacer ce comportement en empêchant le compilateur de fournir un destructeur par défaut.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
```

```
    C c1;
} // Compile ERROR: C has no destructor
```

En outre, un développeur peut également être explicite quant à la volonté du compilateur de fournir un destructeur par défaut.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

c ++ 11

Vous pouvez déterminer si un type a un destructeur (ou est un type primitif) en utilisant `std::is_destructible` de `<type_traits>` :

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Lire Fonctions membres spéciales en ligne: https://riptutorial.com/fr/cplusplus/topic/1476/fonctions-membres-speciales

Chapitre 47: Fonctions membres virtuelles

Syntaxe

- vide virtuel f ();
- void virtuel g () = 0;
- // C ++ 11 ou version ultérieure:
  - virtual void h () remplace;
  - annuler i () outrepasser;
  - vide virtuel j () final;
  - void k () final;

Remarques

- Seules les fonctions membres non statiques, autres que les modèles, peuvent être `virtual` .
- Si vous utilisez C ++ 11 ou une version ultérieure, il est recommandé d'utiliser le `override` lors du `override` une fonction membre virtuelle par une classe de base.
- Les classes de base polymorphes ont souvent des destructeurs virtuels pour permettre la suppression d'un objet dérivé via un pointeur vers la classe de base . Si le destructeur n'était pas virtuel, une telle opération entraîne un comportement indéfini [expr.delete] §5.3.5 / 3 .

Examples

Utilisation de la substitution avec virtual en C ++ 11 et versions ultérieures

Le `override` spécificateur a une signification particulière à partir de C ++ 11, s'il est ajouté à la signature de fin de fonction. Cela signifie qu'une fonction est

- Remplacement de la fonction présente dans la classe de base &
- La fonction de classe de base est `virtual`

Il n'y a pas de signification à l' `run time` de ce spécificateur car il est principalement utilisé comme une indication pour les compilateurs

L'exemple ci-dessous montrera le changement de comportement avec notre sans utiliser de remplacement.

Sans `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
```

```
};

struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

Avec `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Notez que le `override` n'est pas un mot-clé, mais un identifiant spécial qui peut uniquement apparaître dans les signatures de fonction. Dans tous les autres contextes, la `override` peut toujours être utilisée comme identifiant:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2;  // Compilation error: keywords can't be used as identifiers.
}
```

Fonctions membres virtuelles et non virtuelles

Avec des fonctions membres virtuelles:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
```

```
}
```

Sans fonctions membres virtuelles:

```
#include <iostream>

struct X {
   void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
   void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Fonctions virtuelles finales

C ++ 11 introduit un spécificateur `final` qui interdit la méthode si elle apparaît dans la signature de la méthode:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

Le spécificateur `final` ne peut être utilisé qu'avec la fonction membre «virtual» et ne peut pas être

appliqué aux fonctions membres non virtuelles.

Comme `final` , il existe également un appel de spécificateur «override» qui empêche la substitution des fonctions `virtual` dans la classe dérivée.

Les spécificateurs `override` et `final` peuvent être combinés pour avoir l'effet désiré:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Comportement des fonctions virtuelles dans les constructeurs et les destructeurs

Le comportement des fonctions virtuelles dans les constructeurs et les destructeurs est souvent déroutant au premier contact.

```
#include <iostream>
using namespace std;

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", "  << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }

};

int main() {
     derived d;
}
```

Sortie:

> Appelé par le constructeur de base, base :: v () est appelé.
> Lorsqu'elle est appelée à partir d'un constructeur dérivé, :: v () dérivé est appelé.
> Lorsqu'elle est appelée depuis un destructeur dérivé, :: v () dérivé est appelé.
> Appelé à partir du destructeur de base, base :: v () est appelé.

La raison en est que la classe dérivée peut définir des membres supplémentaires qui ne sont pas encore initialisés (dans le cas du constructeur) ou déjà détruits (dans le cas du destructeur), et appeler ses fonctions membres serait dangereux. Par conséquent, lors de la construction et de la destruction d'objets C ++, le type *dynamique* de `*this` est considéré comme étant la classe du constructeur ou du destructeur et non une classe plus dérivée.

Exemple:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override   { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Fonctions virtuelles pures

Nous pouvons également spécifier qu'une fonction `virtual` est *pure virtuelle* (abstraite), en ajoutant `= 0` à la déclaration. Les classes avec une ou plusieurs fonctions virtuelles pures sont considérées comme abstraites et ne peuvent pas être instanciées; seules les classes dérivées qui définissent ou héritent des définitions pour toutes les fonctions virtuelles pures peuvent être instanciées.

```
struct Abstract {
    virtual void f() = 0;
};

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Même si une fonction est spécifiée comme pure virtuelle, une implémentation par défaut peut lui être attribuée. Malgré cela, la fonction sera toujours considérée comme abstraite et les classes

dérivées devront la définir avant de pouvoir être instanciées. Dans ce cas, la version dérivée de la fonction est même autorisée à appeler la version de la classe de base.

```
struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

Il y a plusieurs raisons pour lesquelles nous pourrions vouloir faire ceci:

- Si nous voulons créer une classe qui ne peut pas être elle-même instanciée, mais n'empêche pas ses classes dérivées d'être instanciées, nous pouvons déclarer le destructeur comme virtuel pur. En tant que destructeur, il doit être défini de toute façon si nous voulons pouvoir désallouer l'instance. Et comme le destructeur est probablement déjà virtuel pour empêcher les fuites de mémoire lors d'une utilisation polymorphe , nous ne rencontrerons pas de problèmes de performances inutiles en déclarant une autre fonction `virtual` . Cela peut être utile lors de la création d'interfaces.

  ```
  struct Interface {
      virtual ~Interface() = 0;
  };
  Interface::~Interface() = default;

  struct Implementation : Interface {};
  // ~Implementation() is automatically defined by the compiler if not explicitly
  //  specified, meeting the "must be defined before instantiation" requirement.
  ```

- Si la plupart ou la totalité des implémentations de la fonction virtuelle pure contiennent du code en double, ce code peut plutôt être déplacé vers la version de classe de base, ce qui facilite la gestion du code.

  ```
  class SharedBase {
      State my_state;
      std::unique_ptr<Helper> my_helper;
      // ...

    public:
      virtual void config(const Context& cont) = 0;
      // ...
  };
  /* virtual */ void SharedBase::config(const Context& cont) {
      my_helper = new Helper(my_state, cont.relevant_field);
      do_this();
      and_that();
  }

  class OneImplementation : public SharedBase {
      int i;
      // ...
  ```

```
  public:
    void config(const Context& cont) override;
    // ...
};
void OneImplementation::config(const Context& cont) /* override */ {
    my_state = { cont.some_field, cont.another_field, i };
    SharedBase::config(cont);
    my_unique_setup();
};

// And so on, for other classes derived from SharedBase.
```

Lire Fonctions membres virtuelles en ligne: https://riptutorial.com/fr/cplusplus/topic/1752/fonctions-membres-virtuelles

Chapitre 48: Futures et promesses

Introduction

Promises and Futures sont utilisés pour transporter un objet d'un fil à un autre.

Un objet `std::promise` est défini par le thread qui génère le résultat.

Un objet `std::future` peut être utilisé pour récupérer une valeur, vérifier si une valeur est disponible ou interrompre l'exécution jusqu'à ce que la valeur soit disponible.

Examples

std :: future et std :: promis

L'exemple suivant définit une promesse à utiliser par un autre thread:

```
    {
        auto promise = std::promise<std::string>();

        auto producer = std::thread([&]
        {
            promise.set_value("Hello World");
        });

        auto future = promise.get_future();

        auto consumer = std::thread([&]
        {
            std::cout << future.get();
        });

        producer.join();
        consumer.join();
}
```

Exemple asynchrone différé

Ce code implémente une version de `std::async` , mais il se comporte comme si `async` était toujours appelé avec la politique de lancement `deferred` . Cette fonction n'a pas non plus de comportement `future` particulier pour `async` ; le `future` retourné peut être détruit sans jamais acquérir sa valeur.

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    using result_type = decltype(func());

    auto promise = std::promise<result_type>();
    auto future  = promise.get_future();

    std::thread(std::bind([=](std::promise<result_type>& promise)
```

```
    {
        try
        {
            promise.set_value(func());
            // Note: Will not work with std::promise<void>. Needs some meta-template
programming which is out of scope for this example.
        }
        catch(...)
        {
            promise.set_exception(std::current_exception());
        }
    }, std::move(promise))).detach();

    return future;
}
```

std :: packaged_task et std :: future

`std::packaged_task` regroupe une fonction et la promesse associée pour son type de retour:

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task   = std::packaged_task<decltype(func())()>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}
```

Le thread commence à s'exécuter immédiatement. Nous pouvons soit le détacher, soit le joindre à la fin du périmètre. Lorsque la fonction appelle std :: thread se termine, le résultat est prêt.

Notez que ceci est légèrement différent de `std::async` où le `std::future` retourné lors de la destruction **bloquera** réellement jusqu'à ce que le thread soit terminé.

std :: future_error et std :: future_errc

Si les contraintes pour std :: promise et std :: future ne sont pas remplies, une exception de type std :: future_error est levée.

Le membre de code d'erreur dans l'exception est de type std :: future_errc et les valeurs sont comme ci-dessous, avec quelques cas de test:

```
enum class future_errc {
    broken_promise             = /* the task is no longer shared */,
    future_already_retrieved   = /* the answer was already retrieved */,
    promise_already_satisfied  = /* the answer was stored already */,
    no_state                   = /* access to a promise in non-shared state */
};
```

Promesse inactive:

```
int test()
{
    std::promise<int> pr;
    return 0; // returns ok
}
```

Promesse active, non utilisée:

```
  int test()
    {
        std::promise<int> pr;
        auto fut = pr.get_future(); //blocks indefinitely!
        return 0;
    }
```

Double récupération:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

    try{
        auto fut2 = pr.get_future();    //   second attempt to get future
        return 0;
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The future has already been retrieved
from the promise or packaged_task."
        return -1;
    }
    return fut2.get();
}
```

Définir la valeur std :: promise deux fois:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10);  // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The state of the promise has already been
set."
        return -1;
    }
    return fut.get();
}
```

std :: future et std :: async

Dans l'exemple de tri de fusion parallèle naïf suivant, `std::async` est utilisé pour lancer plusieurs tâches parallèles de fusion. `std::future` est utilisé pour attendre les résultats et les synchroniser:

```
#include <iostream>
using namespace std;


void merge(int low,int mid,int high, vector<int>&num)
{
    vector<int> copy(num.size());
    int h,i,j,k;
    h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
            copy[i]=num[h];
            h++;
        }
        else
        {
            copy[i]=num[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    else
    {
        for(k=h;k<=mid;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}


void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low<high)
    {
        mid = low + (high-low)/2;
        auto future1    =  std::async(std::launch::deferred,[&]()
                                      {
                                        merge_sort(low,mid,num);
                                      });
```

```
        auto future2     =  std::async(std::launch::deferred, [&]()
                                {
                                   merge_sort(mid+1,high,num) ;
                                });

        future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}
```

Remarque: Dans l'exemple `std::async` est lancé avec la stratégie `std::launch_deferred` . Ceci pour éviter qu'un nouveau thread ne soit créé dans chaque appel. Dans le cas de notre exemple, les appels à `std::async` sont `std::async` , ils se synchronisent lors des appels à `std::future::get()` .

`std::launch_async` force la `std::launch_async` un nouveau thread à chaque appel.

La politique par défaut est `std::launch::deferred| std::launch::async` , ce qui signifie que l'implémentation détermine la stratégie de création de nouveaux threads.

Classes d'opération asynchrones

- std :: async: effectue une opération asynchrone.
- std :: future: donne accès au résultat d'une opération asynchrone.
- std :: promise: package le résultat d'une opération asynchrone.
- std :: packaged_task: regroupe une fonction et la promesse associée pour son type de retour.

Lire Futures et promesses en ligne: https://riptutorial.com/fr/cplusplus/topic/9840/futures-et-promesses

Chapitre 49: Génération de nombres aléatoires

Remarques

La génération de nombres aléatoires en C ++ est fournie par l'en-tête `<random>` . Cet en-tête définit des périphériques aléatoires, des générateurs pseudo-aléatoires et des distributions.

Les périphériques aléatoires renvoient des nombres aléatoires fournis par le système d'exploitation. Ils doivent être utilisés soit pour l'initialisation de générateurs pseudo-aléatoires, soit directement pour des besoins cryptographiques.

Les générateurs pseudo-aléatoires renvoient des nombres pseudo-aléatoires entiers basés sur leur graine initiale. La plage de nombres pseudo-aléatoires couvre généralement toutes les valeurs d'un type non signé. Tous les générateurs pseudo-aléatoires de la bibliothèque standard renverront les mêmes numéros pour la même graine initiale pour toutes les plates-formes.

Les distributions consomment des nombres aléatoires de générateurs pseudo-aléatoires ou de dispositifs aléatoires et produisent des nombres aléatoires avec la distribution nécessaire. Les distributions ne sont pas indépendantes de la plate-forme et peuvent produire des nombres différents pour les mêmes générateurs avec les mêmes semences initiales sur différentes plates-formes.

Examples

Véritable générateur de valeur aléatoire

Pour générer de vraies valeurs aléatoires pouvant être utilisées pour la cryptographie, `std::random_device` doit être utilisé comme générateur.

```
#include <iostream>
#include <random>

int main()
{
   std::random_device crypto_random_generator;
   std::uniform_int_distribution<int> int_distribution(0,9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(crypto_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i < 10; i++) {
       std::cout << actual_distribution[i] << " ";
   }
```

```
    return 0;
}
```

`std::random_device` est utilisé de la même manière qu'un générateur de valeur pseudo-aléatoire est utilisé.

Cependant, `std::random_device` **peut être implémenté en termes de moteur de nombres pseudo-aléatoires défini par l'implémentation** si une source non déterministe (par exemple un périphérique matériel) n'est pas disponible pour l'implémentation.

La détection de telles implémentations devrait être possible via la fonction membre `entropy` (qui retourne zéro lorsque le générateur est totalement déterministe), mais de nombreuses bibliothèques populaires (libstdc ++ et libc ++ de LLVM) renvoient toujours zéro, même lorsqu'elles utilisent un caractère aléatoire externe de haute qualité. .

Générer un nombre pseudo-aléatoire

Un générateur de nombres pseudo-aléatoires génère des valeurs qui peuvent être devinées en fonction des valeurs précédemment générées. En d'autres termes: c'est déterministe. N'utilisez pas de générateur de nombres pseudo-aléatoires dans les cas où un nombre aléatoire réel est requis.

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);

    int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

    for(int i = 0; i < 10000; i++) {
        int result = int_distribution(pseudo_random_generator);
        actual_distribution[result]++;
    }

    for(int i = 0; i <= 9; i++) {
        std::cout << actual_distribution[i] << " ";
    }

    return 0;
}
```

Ce code crée un générateur de nombres aléatoires et une distribution qui génère des nombres entiers dans la plage [0,9] avec une probabilité égale. Il compte ensuite combien de fois chaque résultat a été généré.

Le paramètre template de `std::uniform_int_distribution<T>` spécifie le type d'entier à générer. Utilisez `std::uniform_real_distribution<T>` pour générer des flottants ou des doubles.

Utilisation du générateur pour plusieurs distributions

Le générateur de nombres aléatoires peut (et devrait) être utilisé pour plusieurs distributions.

```
#include <iostream>
#include <random>

int main()
{
   std::default_random_engine pseudo_random_generator;
   std::uniform_int_distribution<int> int_distribution(0, 9);
   std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
   std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

   std::cout << int_distribution(pseudo_random_generator) << std::endl;
   std::cout << float_distribution(pseudo_random_generator) << std::endl;
   std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

   return 0;
}
```

Dans cet exemple, un seul générateur est défini. Il est ensuite utilisé pour générer une valeur aléatoire dans trois distributions différentes. La distribution `rigged_dice` va générer une valeur comprise entre 0 et 5, mais génère presque toujours un `5` , car la chance de générer un `5` est de `100 / 105` .

Lire Génération de nombres aléatoires en ligne: https://riptutorial.com/fr/cplusplus/topic/1541/generation-de-nombres-aleatoires

Chapitre 50: Gestion de la mémoire

Syntaxe

- : :(*opt*) new (*liste-expression*) (*opt*) *new-type-id new-initializer* (*opt*)
- : :(*opt*) new (*liste-expression*) (*opt*) (*type-id*) *nouvel-initialiseur* (*opt*)
- : :(*opt*) supprimer l' *expression-cast*
- : :(*opt*) delete [] *cast-expression*
- std :: unique_ptr < *type-id* > nom_variable (nouvel *identificateur de type* (*opt*)); // C ++ 11
- std :: shared_ptr < *type-id* > nom_variable (nouveau *type-id* (*opt*)); // C ++ 11
- std :: shared_ptr < *id-type* > nom_var = std :: make_shared < *id-type* > (*opt*); // C ++ 11
- std :: unique_ptr < *type-id* > nom_var = std :: make_unique < *type-id* > (*opt*); // C ++ 14

Remarques

Un leader `::` force l'opérateur new ou delete à rechercher dans une portée globale, en remplaçant tous les opérateurs surchargés ou nouveaux spécifiques à la classe.

Les arguments facultatifs qui suivent le `new` mot-clé sont généralement utilisés pour appeler le placement new , mais peuvent également être utilisés pour transmettre des informations supplémentaires à l'allocateur, par exemple une balise demandant que la mémoire soit allouée depuis un pool choisi.

Le type alloué est généralement explicitement spécifié, *par exemple* `new Foo` , mais peut aussi être écrit comme `auto` (depuis C ++ 11) ou `decltype(auto)` (depuis C ++ 14) pour le déduire de l'initialiseur.

L'initialisation de l'objet alloué se produit selon les mêmes règles que l'initialisation des variables locales. En particulier, l'objet sera initialisé par défaut si l'initialiseur est omis, et lors de l'allocation dynamique d'un type scalaire ou d'un tableau de type scalaire, rien ne garantit que la mémoire sera mise à zéro.

Un objet tableau créé par une *nouvelle expression* doit être détruit à l'aide de `delete[]` , que la *nouvelle expression ait* été écrite avec `[]` ou non. Par exemple:

```
using IA = int[4];
int* pIA = new IA;
delete[] pIA;  // right
// delete pIA;  // wrong
```

Examples

Empiler

La pile est une petite région de mémoire dans laquelle des valeurs temporaires sont placées

pendant l'exécution. L'allocation de données dans la pile est très rapide par rapport à l'allocation de tas car toute la mémoire a déjà été affectée à cette fin.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

La pile est nommée parce que les chaînes d'appels de fonctions auront leur mémoire temporaire «empilée» l'une sur l'autre, chacune utilisant une petite section de mémoire distincte.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

Les données stockées sur la pile ne sont valides que tant que la portée à laquelle la variable est allouée est toujours active.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

Stockage gratuit (tas, allocation dynamique ...)

Le terme **«tas»** est un terme informatique général désignant une zone de mémoire à partir de laquelle des parties peuvent être allouées et désallouées indépendamment de la mémoire fournie par la **pile** .

En `C++` la *norme* fait référence à cette zone en tant que **magasin gratuit,** ce qui est considéré comme un terme plus précis.

Les zones de mémoire allouées à partir du **magasin gratuit** peuvent vivre plus longtemps que la portée d'origine dans laquelle elles ont été allouées. Les données trop volumineuses pour être stockées sur la pile peuvent également être attribuées à partir du **magasin gratuit** .

La mémoire brute peut être allouée et libérée par le *nouveau* et *supprimer les* mots-clés.

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;              // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;          // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;       // Setting the pointer to nullptr after delete is often considered good
practice
```

Il est également possible d'allouer des tableaux de taille fixe avec *new* et *delete* , avec une syntaxe légèrement différente. L'attribution des tableaux n'est pas compatible avec l'allocation sans tableau, et le mélange des deux entraînera une corruption du tas. L'allocation d'un tableau alloue également de la mémoire pour suivre la taille du tableau en vue d'une suppression ultérieure, de manière définie par l'implémentation.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

Si vous utilisez *new* et *delete* plutôt que *malloc* et *free* , le constructeur et le destructeur seront exécutés (similaire aux objets basés sur la pile). C'est pourquoi les *nouveaux* et les *supprimer* sont préférables à *malloc* et *gratuits* .

```
struct ComplexType {
    int a = 0;

    ComplexType() { std::cout << "Ctor" << std::endl; }
    ~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
//Calls the destructor for ComplexType() and deletes memory for a Complextype at pC
delete foo;
```

C ++ 11

A partir de C ++ 11, l'utilisation de pointeurs intelligents est recommandée pour indiquer la propriété.

C ++ 14

C ++ 14 a ajouté `std::make_unique` à la STL, en modifiant la recommandation pour privilégier `std::make_unique` ou `std::make_shared` au lieu d'utiliser nue *new* et *delete* .

Nouveau placement

Il y a des situations où nous ne voulons pas compter sur Free Store pour allouer de la mémoire et nous voulons utiliser des allocations de mémoire personnalisées en utilisant `new` .

Pour ces situations, nous pouvons utiliser `Placement New` , où nous pouvons demander à un nouvel opérateur d'allouer de la mémoire à partir d'un emplacement de mémoire pré-alloué.

Par exemple

```
int a4byteInteger;

char *a4byteChar = new (&a4byteInteger) char[4];
```

Dans cet exemple, la mémoire pointée par `a4byteChar` est de 4 octets alloués à 'stack' via la variable entière `a4byteInteger` .

L'avantage de ce type d'allocation de mémoire est le fait que les programmeurs contrôlent l'allocation. Dans l'exemple ci-dessus, comme `a4byteInteger` est alloué sur la pile, nous n'avons pas besoin d'appeler explicitement 'delete a4byteChar`.

Le même comportement peut être obtenu pour la mémoire allouée dynamique également. Par exemple

```
int *a8byteDynamicInteger = new int[2];

char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

Dans ce cas, le pointeur de mémoire par `a8byteChar` fera référence à la mémoire dynamique allouée par `a8byteDynamicInteger` . Dans ce cas, cependant, nous devons appeler explicitement `delete a8byteDynamicInteger` pour libérer la mémoire

Un autre exemple pour la classe C ++

```
struct ComplexType {
    int a;

    ComplexType() : a(0) {}
    ~ComplexType() {}
};

int main() {
    char* dynArray = new char[256];

    //Calls ComplexType's constructor to initialize memory as a ComplexType
    new((void*)dynArray) ComplexType();

    //Clean up memory once we're done
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();
    delete[] dynArray;

    //Stack memory can also be used with placement new
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11
```

```
    new((void*)localArray) ComplexType();

    //Only need to call the destructor for stack memory
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();

    return 0;
}
```

Lire Gestion de la mémoire en ligne: https://riptutorial.com/fr/cplusplus/topic/2873/gestion-de-la-memoire

Chapitre 51: Implémentation du modèle de conception en C ++

Introduction

Sur cette page, vous trouverez des exemples d'implémentation de modèles de conception en C ++. Pour plus de détails sur ces modèles, vous pouvez consulter la documentation sur les modèles de conception .

Remarques

Un modèle de conception est une solution réutilisable générale à un problème courant dans un contexte donné lors de la conception de logiciels.

Examples

Motif d'observateur

L'intention de Observer Pattern est de définir une dépendance un à plusieurs entre les objets, de sorte que lorsqu'un objet change d'état, tous ses dépendants sont notifiés et mis à jour automatiquement.

Le sujet et les observateurs définissent la relation un-à-plusieurs. Les observateurs dépendent du sujet de telle sorte que lorsque l'état du sujet change, les observateurs sont notifiés. Selon la notification, les observateurs peuvent également être mis à jour avec de nouvelles valeurs.

Voici l'exemple du livre "Design Patterns" de Gamma.

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
     virtual ~Subject() = default;
     void Attach(Observer& o) { observers.push_back(&o); }
     void Detach(Observer& o)
     {
         observers.erase(std::remove(observers.begin(), observers.end(), &o));
```

```
    }
    void Notify()
    {
        for (auto* o : observers) {
            o->Update(*this);
        }
    }
private:
    std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:

    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
        this->minute = minute;
        this->second = second;

        Notify();
    }

    int GetHour() const { return hour; }
    int GetMinute() const { return minute; }
    int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
    explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~DigitalClock() { subject.Detach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Digital time is " << hour << ":"
                  << minute << ":"
                  << second << std::endl;
    }

private:
    ClockTimer& subject;
};
```

```
class AnalogClock: public Observer
{
public:
     explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~AnalogClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }
     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Analog time is " << hour << ":"
                   << minute << ":"
                   << second << std::endl;
     }
private:
     ClockTimer& subject;
};

int main()
{
    ClockTimer timer;

    DigitalClock digitalClock(timer);
    AnalogClock analogClock(timer);

    timer.SetTime(14, 41, 36);
}
```

Sortie:

```
Digital time is 14:41:36
Analog time is 14:41:36
```

Voici le résumé du motif:

1. Les objets (objet `DigitalClock` ou `AnalogClock`) utilisent les interfaces Subject (`Attach()` ou `Detach()`) pour s'inscrire (s'inscrire) en tant qu'observateur ou se désabonner (supprimer) de la fonction d'observateur (`subject.Attach(*this); subject.Detach(*this);`

2. Chaque sujet peut avoir plusieurs observateurs (observateurs `vector<Observer*> observers;`).

3. Tous les observateurs doivent implémenter l'interface Observer. Cette interface a juste une méthode, `Update()` , qui est appelée lorsque l'état du sujet change (`Update(Subject &)`)

4. Outre les méthodes `Attach()` et `Detach()` , le sujet concret implémente une méthode `Notify()` qui permet de mettre à jour tous les observateurs actuels chaque fois que l'état change. Mais dans ce cas, tous sont faits dans la classe parente, `Subject` (`Subject::Attach (Observer&)` , `void Subject::Detach(Observer&)` et `void Subject::Notify()` .

5. L'objet Concrete peut également avoir des méthodes pour définir et obtenir son état.

6. Les observateurs concrets peuvent être n'importe quelle classe qui implémente l'interface Observer. Chaque observateur s'abonne (s'inscrire) à un sujet concret pour recevoir la mise à jour (`subject.Attach(*this);`).

7. Les deux objets de Observer Pattern sont **faiblement couplés** , ils peuvent interagir mais avec peu de connaissance les uns des autres.

Variation:

Signal et Slots

Les signaux et les slots sont une construction de langage introduite dans Qt, ce qui facilite l'implémentation du pattern Observer tout en évitant le code passe-partout. Le concept est que les contrôles (également appelés widgets) peuvent envoyer des signaux contenant des informations d'événement pouvant être reçues par d'autres contrôles utilisant des fonctions spéciales appelées slots. L'emplacement dans Qt doit être un membre de classe déclaré comme tel. Le système de signal / logement correspond bien à la conception des interfaces utilisateur graphiques. De même, le système de signal / logement peut être utilisé pour la notification d'événement des E / S asynchrones (y compris les sockets, les tubes, les périphériques série, etc.) ou pour associer des événements de temporisation aux instances et méthodes ou fonctions d'objet appropriées. Aucun code d'inscription / de désenregistrement / d'invocation n'a besoin d'être écrit, car le compilateur de méta-objets (MOC) de Qt génère automatiquement l'infrastructure nécessaire.

Le langage C # prend également en charge un concept similaire, bien qu'avec une terminologie et une syntaxe différentes: les événements jouent le rôle de signaux et les délégués sont les emplacements. De plus, un délégué peut être une variable locale, un peu comme un pointeur de fonction, alors qu'un emplacement dans Qt doit être un membre de classe déclaré comme tel.

Modèle d'adaptateur

Convertir l'interface d'une classe en une autre interface attendue par les clients. Adapter (ou Wrapper) permet aux classes de fonctionner ensemble, ce qui ne pourrait pas être dû à des interfaces incompatibles. La motivation du modèle d'adaptateur est que nous pouvons réutiliser un logiciel existant si nous pouvons modifier l'interface.

1. Le modèle d'adaptateur repose sur la composition de l'objet.

2. Opération d'appels clients sur l'objet adaptateur.

3. L'adaptateur appelle Adaptee pour effectuer l'opération.

4. En STL, pile adaptée du vecteur: Lorsque la pile exécute push (), le vecteur sous-jacent fait vector :: push_back ().

Exemple:

```
#include <iostream>
```

```
// Desired interface (Target)
class Rectangle
{
  public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
  public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
  private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
         std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};

int main()
{
  int x = 20, y = 50, w = 300, h = 200;
  Rectangle *r = new RectangleAdapter(x,y,w,h);
  r->draw();
}

//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)
```

Récapitulatif du code:

1. Le client pense qu'il parle à un `Rectangle`
2. La cible est la classe `Rectangle` . C'est ce que le client appelle la méthode.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
```

3. Notez que la classe d'adaptateur utilise plusieurs héritages.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}
```

4. Adapter `RectangleAdapter` permet à `LegacyRectangle` répondre aux requêtes (`draw()` sur un `Rectangle`) en héritant des deux classes.

5. La classe `LegacyRectangle` n'a pas les mêmes méthodes (`draw()`) que `Rectangle` , mais l' `Adapter(RectangleAdapter)` peut prendre les appels de méthode `Rectangle` et `LegacyRectangle` la méthode sur `LegacyRectangle` , `oldDraw()` .

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

Le modèle de conception de l' **adaptateur** traduit l'interface d'une classe en une interface compatible mais différente. Donc, ceci est similaire au modèle de **proxy** en ce qu’il s’agit d’un wrapper à un seul composant. Mais l'interface pour la classe d'adaptateur et la classe d'origine peuvent être différentes.

Comme nous l'avons vu dans l'exemple ci-dessus, ce modèle d' **adaptateur** est utile pour exposer une interface différente pour une API existante afin de lui permettre de fonctionner avec un autre code. De plus, en utilisant un modèle d'adaptateur, nous pouvons prendre des interfaces hétérogènes et les transformer pour fournir une API cohérente.

Le pont a une structure similaire à un adaptateur d'objet, mais Bridge a une intention différente: il est conçu pour **séparer** une interface de son implémentation afin de pouvoir la modifier facilement et indépendamment. Un **adaptateur** est destiné à **modifier l'interface** d'un objet **existant** .

Modèle d'usine

Le motif d'usine dissocie la création d'objet et permet la création par nom en utilisant une interface commune:

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
```

```
    virtual std::string  getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};


class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string  getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string&   name )
    {
      if ( name == "bear" )
        return std::make_shared<Bear>();
      if ( name == "cat" )
        return std::shared_ptr<Cat>();

    return nullptr;
    }


};
```

Modèle de générateur avec API Fluent

Le modèle de générateur dissocie la création de l'objet de l'objet lui-même. L'idée principale est qu'un **objet ne doit pas nécessairement être responsable de sa propre création** . L'assemblage correct et valide d'un objet complexe peut être une tâche compliquée en soi, cette tâche peut donc être déléguée à une autre classe.

Inspiré par Email Builder en C # , j'ai décidé de faire une version C ++ ici. Un objet Email n'est pas nécessairement un *objet très complexe* , mais il peut démontrer le motif.

```
#include <iostream>
#include <sstream>
#include <string>
```

```
using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
  public:
    friend class EmailBuilder;  // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
               << "\nto: " << m_to
               << "\nsubject: " << m_subject
               << "\nbody: " << m_body;
        return stream.str();
    }

  private:
    Email() = default; // restrict construction to builder

    string m_from;
    string m_to;
    string m_subject;
    string m_body;
};

class EmailBuilder
{
  public:
    EmailBuilder& from(const string &from) {
        m_email.m_from = from;
        return *this;
    }

    EmailBuilder& to(const string &to) {
        m_email.m_to = to;
        return *this;
    }

    EmailBuilder& subject(const string &subject) {
        m_email.m_subject = subject;
        return *this;
    }

    EmailBuilder& body(const string &body) {
        m_email.m_body = body;
        return *this;
    }

    operator Email&&() {
        return std::move(m_email); // notice the move
    }

  private:
    Email m_email;
};
```

```
EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}


int main()
{
    Email mail = Email::make().from("me@mail.com")
                              .to("you@mail.com")
                              .subject("C++ builders")
                              .body("I like this API, don't you?");

    cout << mail << endl;
}
```

Pour les anciennes versions de C ++, il suffit d'ignorer l'opération `std::move` et de supprimer le && de l'opérateur de conversion (bien que cela crée une copie temporaire).

Le générateur termine son travail lorsqu'il libère l'e- `operator Email&&()` par l' `operator Email&&()` . Dans cet exemple, le générateur est un objet temporaire et renvoie le courrier électronique avant sa destruction. Vous pouvez également utiliser une opération explicite comme `Email EmailBuilder::build() {...}` au lieu de l'opérateur de conversion.

Passer le constructeur autour

Une caractéristique intéressante du modèle **de création** est la possibilité d' **utiliser plusieurs acteurs pour construire un objet ensemble.** Cela se fait en passant le constructeur aux autres acteurs qui donneront chacun plus d'informations à l'objet construit. Ceci est particulièrement puissant lorsque vous créez une sorte de requête, en ajoutant des filtres et d'autres spécifications.

```
void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
           .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
           .body("And the body. Someone else knows the addresses.");
}

int main()
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);
```

```
    Email mail = builder;
    cout << mail << endl;
}
```

Variante de conception: objet Mutable

Vous pouvez modifier la conception de ce modèle pour répondre à vos besoins. Je vais donner une variante.

Dans l'exemple donné, l'objet Email est immuable, c'est-à-dire que ses propriétés ne peuvent pas être modifiées car il n'y a pas d'accès. C'était une fonctionnalité souhaitée. Si vous devez modifier l'objet après sa création, vous devez lui en fournir. Étant donné que ces paramètres seraient dupliqués dans le générateur, vous pouvez envisager de tout faire en une seule classe (aucune classe de générateur plus nécessaire). Néanmoins, je considérerais le besoin de rendre l'objet construit mutable en premier lieu.

Lire Implémentation du modèle de conception en C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/4335/implementation-du-modele-de-conception-en-c-plusplus

Chapitre 52: Internationalisation en C ++

Remarques

Le langage C ++ ne dicte pas de jeu de caractères, certains compilateurs peuvent prendre en ***charge*** l'utilisation de UTF-8, ou même UTF-16. Cependant, rien ne garantit que des éléments autres que de simples caractères ANSI / ASCII seront fournis.

Ainsi, toute prise en charge linguistique internationale est définie par l'implémentation, en fonction de la plate-forme, du système d'exploitation et du compilateur que vous utilisez.

Plusieurs bibliothèques tierces (telles que International Unicode Committee Library) pouvant être utilisées pour étendre le support international de la plate-forme.

Examples

Comprendre les caractéristiques de la chaîne C ++

```
#include <iostream>
#include <string>

int main()
{
    const char * C_String = "This is a line of text w";
    const char * C_Problem_String = "This is a line of text ῗ";
  std::string Std_String("This is a second line of text w");
    std::string Std_Problem_String("This is a second line of □ϵx□ ῗ");

  std::cout << "String Length: " << Std_String.length() << '\n';
    std::cout << "String Length: " << Std_Problem_String.length() << '\n';

    std::cout << "CString Length: " << strlen(C_String) << '\n';
    std::cout << "CString Length: " << strlen(C_Problem_String) << '\n';
    return 0;
}
```

Selon la plate-forme (Windows, OSX, etc.) et le compilateur (GCC, MSVC, etc.), ce programme **peut ne pas compiler, afficher des valeurs différentes ou afficher les mêmes valeurs** .

Exemple de sortie sous le compilateur Microsoft MSVC:

> Longueur de chaîne: 31
> Longueur de chaîne: 31
> Longueur CString: 24
> Longueur CString: 24

Cela montre que sous MSVC, chacun des caractères étendus utilisés est considéré comme un "caractère" unique et que cette plate-forme supporte entièrement les langues internationalisées. *Il convient de noter cependant que ce comportement est inhabituel, ces caractères internationaux*

sont stockés en interne sous la forme Unicode et ont donc en réalité plusieurs octets de long. ***Cela peut provoquer des erreurs inattendues***

Sous le compilateur GNC / GCC, la sortie du programme est la suivante:

> Longueur de chaîne: 31
> Longueur de chaîne: 36
> Longueur CString: 24
> Longueur CString: 26

Cet exemple montre que, bien que le compilateur GCC utilisé sur cette plate-forme (Linux) prenne en charge ces caractères étendus, il utilise également (*correctement)* plusieurs octets pour stocker un caractère individuel.
Dans ce cas, l'utilisation de caractères Unicode est possible, mais le programmeur doit faire très attention en se rappelant que la longueur d'une "chaîne" dans ce scénario est la ***longueur en octets*** , et non la ***longueur en caractères lisibles*** .

Ces différences sont dues à la façon dont des langues internationales sont gérées sur une base par plateforme - et plus important encore , que les chaînes C et C ++ utilisées dans cet exemple peut être considéré comme ***un tableau d'octets,*** de sorte que (pour cet usage) ***le langage C ++ considère un caractère (char) doit être un octet unique*** .

Lire Internationalisation en C ++ en ligne:
https://riptutorial.com/fr/cplusplus/topic/5270/internationalisation-en-c-plusplus

Chapitre 53: Itération

Examples

Pause

Saute hors de la boucle englobante ou de l'instruction `switch` la plus proche.

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

continuer

Saute à la fin de la plus petite boucle englobante.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

faire

Introduit une boucle do-while .

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

pour

Introduit une boucle for ou, en C ++ 11 et versions ultérieures, une boucle basée sur une plage .

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
```

```
    putchar('*');
}
```

tandis que

Introduit une boucle while .

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

basé sur la plage pour la boucle

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Lire Itération en ligne: https://riptutorial.com/fr/cplusplus/topic/7841/iteration

Chapitre 54: La gestion des ressources

Introduction

L'une des choses les plus difficiles à faire en C et C ++ est la gestion des ressources. Heureusement, en C ++, nous avons plusieurs façons de concevoir la gestion des ressources dans nos programmes. Cet article espère expliquer certains des idiomes et méthodes utilisés pour gérer les ressources allouées.

Examples

L'acquisition de ressources est une initialisation

L'acquisition de ressources est l'initialisation (RAII) est un idiome commun dans la gestion des ressources. Dans le cas de la mémoire dynamique, il utilise des pointeurs intelligents pour effectuer la gestion des ressources. Lors de l'utilisation de RAII, une ressource acquise est immédiatement attribuée à un pointeur intelligent ou à un gestionnaire de ressources équivalent. La ressource est uniquement accessible via ce gestionnaire, de sorte que le gestionnaire peut suivre les différentes opérations. Par exemple, `std::auto_ptr` libère automatiquement la ressource correspondante lorsqu'elle est hors de portée ou est supprimée d'une autre manière.

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    {
        auto_ptr ap(new int(5)); // dynamic memory is the resource
        cout << *ap << endl; // prints 5
    } // auto_ptr is destroyed, its resource is automatically freed
}
```

C ++ 11

Le problème principal de `std::auto_ptr` est qu'il ne peut pas être copié sans transférer la propriété:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
    cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

À cause de cette sémantique de copie bizarre, `std::auto_ptr` ne peut pas être utilisé, entre autres,

dans des conteneurs. La raison en est que cela empêche la suppression de la mémoire deux fois: s'il y a deux `auto_ptrs` avec la même ressource, ils essaient tous deux de le libérer lorsqu'ils sont détruits. Libérer une ressource déjà libérée peut généralement poser problème, il est donc important de la prévenir. Cependant, `std::shared_ptr` a une méthode pour éviter cela tout en ne transférant pas la propriété lors de la copie:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr sp2;
    {
        shared_ptr sp1(new int(5)); // give ownership to sp1
        cout << *sp1 << endl; // prints 5
        sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
        cout << *sp1 << endl; // prints 5
        cout << *sp2 << endl; // prints 5
    } // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
    cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed
```

Mutexes et sécurité des fils

Des problèmes peuvent survenir lorsque plusieurs threads tentent d'accéder à une ressource. Pour un exemple simple, supposons que nous ayons un thread qui en ajoute un à une variable. Pour cela, il faut d'abord lire la variable, en ajouter une, puis la stocker. Supposons que nous initialisons cette variable à 1, puis créons deux instances de ce thread. Une fois les deux threads terminés, l'intuition suggère que cette variable ait une valeur de 3. Cependant, le tableau ci-dessous illustre ce qui pourrait mal tourner:

	Fil 1	Fil 2
Étape 1	Lire 1 de la variable	
Étape 2		Lire 1 de la variable
Étape 3	Ajouter 1 plus 1 pour obtenir 2	
Étape 4		Ajouter 1 plus 1 pour obtenir 2
Étape 5	Stocker 2 en variable	
Étape 6		Stocker 2 en variable

Comme vous pouvez le voir, à la fin de l'opération, 2 est dans la variable, au lieu de 3. La raison en est que Thread 2 lit la variable avant que Thread 1 ne soit terminé. La solution? Mutexes

Un mutex (portemanteau **mut ex** uel clusion) est un objet de gestion des ressources conçu pour résoudre ce type de problème. Lorsqu'un thread veut accéder à une ressource, il "acquiert" le mutex de la ressource. Une fois l'accès à la ressource terminé, le thread "libère" le mutex. Tant

que le mutex est acquis, tous les appels pour acquérir le mutex ne reviendront pas tant que le mutex n'est pas libéré. Pour mieux comprendre cela, pensez à un mutex comme une file d'attente au supermarché: les threads s'alignent en essayant d'acquérir le mutex, puis attendent que les threads qui les précèdent finissent, puis utilisent la ressource, puis sortent de ligne en libérant le mutex. Il y aurait un pandémonium complet si tout le monde essayait d'accéder à la ressource immédiatement.

C ++ 11

`std::mutex` est l'implémentation d'un mutex par C ++ 11.

```
#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish

    cout << var << endl; // prints 3
}
```

Lire La gestion des ressources en ligne: https://riptutorial.com/fr/cplusplus/topic/8336/la-gestion-des-ressources

Chapitre 55: La norme ISO C ++

Introduction

En 1998, il y a eu une première publication du standard rendant C ++ un langage standardisé en interne. A partir de ce moment, C ++ a évolué, donnant lieu à différents dialectes du C ++. Sur cette page, vous trouverez un aperçu de toutes les différentes normes et de leurs modifications par rapport à la version précédente. Les détails sur l'utilisation de ces fonctionnalités sont décrits sur des pages plus spécialisées.

Remarques

Lorsque C ++ est mentionné, il est souvent fait référence à "la norme". Mais quelle est exactement cette norme?

C ++ a une longue histoire. Lancé en tant que petit projet par Bjarne Stroustrup au sein des Bell Labs, au début des années 90, il était devenu très populaire. Plusieurs sociétés créaient des compilateurs C ++ afin que les utilisateurs puissent exécuter leurs compilateurs C ++ sur un large éventail d’ordinateurs. Mais pour faciliter cela, tous les compilateurs concurrents devraient partager une définition unique de la langue.

À ce stade, le langage C avait été normalisé avec succès. Cela signifie qu'une description formelle de la langue a été écrite. Celle-ci a été soumise à l’American National Standards Institute (ANSI), qui a ouvert le document pour examen et l’a ensuite publié en 1989. Un an plus tard, l’Organisation internationale de normalisation , ISO, dérivé du grec isos, qui signifie égal.), A adopté la norme américaine en tant que norme internationale.

Pour C ++, il était clair dès le début qu'il y avait un intérêt international. Un groupe de travail au sein de l'ISO a été créé (connu sous le nom de GT21, au sein du sous-comité 22). Ce groupe de travail a rédigé une première norme vers 1995. Mais comme nous le savons les programmeurs, il n’ya rien de plus dangereux pour une diffusion planifiée que les fonctionnalités de dernière minute, et c’est aussi le cas pour C ++. En 1995, une nouvelle bibliothèque géniale nommée STL a fait surface et les personnes travaillant dans WG21 ont décidé d'ajouter une version allégée au brouillon C ++. Naturellement, les délais ont été dépassés et, trois ans plus tard, le document est devenu définitif. ISO est une organisation très formelle, de sorte que le standard C ++ a été baptisé avec le nom peu commercialisable de ISO / IEC 14882. Comme les normes peuvent être mises à jour, cette version exacte est connue sous le nom de 14882: 1998.

Et en effet, il y avait une demande de mise à jour de la norme. Le standard est un document très épais, qui vise à décrire exactement le fonctionnement des compilateurs C ++. Même une légère ambiguïté mérite d’être corrigée. En 2003, une mise à jour a été publiée en 14882: 2003. Cependant, cela n'a ajouté aucune fonctionnalité à C ++; les nouvelles fonctionnalités ont été programmées pour la deuxième mise à jour.

De manière informelle, cette deuxième mise à jour était connue sous le nom de C ++ 0x, car on ne

savait pas si cela prendrait avant 2008 ou 2009. Eh bien, cette version a également été légèrement retardée, ce qui explique pourquoi elle est devenue 14882: 2011.

Heureusement, le GT21 a décidé de ne pas laisser cela se reproduire. C ++ 11 a été bien accueilli et a suscité un intérêt renouvelé pour C ++. Pour conserver cet élan, la troisième mise à jour est passée de la planification à la publication en 3 ans, pour devenir 14882: 2014.

Le travail ne s'est pas arrêté là non plus. La norme C ++ 17 a été proposée et le travail pour C ++ 20 a été démarré.

Examples

Projets de travail actuels

Toutes les normes ISO publiées sont disponibles à la vente auprès de la boutique ISO (http://www.iso.org). Les versions de travail des normes C ++ sont disponibles gratuitement pour le public.

Les différentes versions de la norme:

- À venir (parfois appelé C ++ 20 ou C ++ 2a): version de travail actuelle (version HTML)
- Proposé (parfois appelé C ++ 17 ou C ++ 1z): ébauche de travail de mars 2017 N4659 .
- C ++ 14 (parfois appelé C ++ 1y): version de travail de novembre 2014 N4296
- C ++ 11 (parfois appelé C ++ 0x): brouillon de travail de février 2011 N3242
- C ++ 03
- C ++ 98

C ++ 11

Le standard C ++ 11 est une extension majeure du standard C ++. Vous trouverez ci-dessous un aperçu des modifications telles qu'elles ont été regroupées dans la FAQ isocpp avec des liens vers une documentation plus détaillée.

Extensions de langue

Caractéristiques générales

- auto
- decltype
- Déclaration de portée
- Listes d'initialisation
- Syntaxe d'initialisation uniforme et sémantique
- Références Rvalue et sémantique de déplacement
- Lambdas
- sauf pour empêcher la propagation des exceptions

- constexpr
- nullptr - un littéral de pointeur nul
- Copier et redéfinir les exceptions
- Espaces de noms en ligne
- Littéraux définis par l'utilisateur

Des classes

- = par défaut et = supprimer
- Contrôle du déplacement et de la copie par défaut
- Déléguer des constructeurs
- Initialisateurs membres en classe
- Constructeurs hérités
- Contrôles de remplacement: remplacer
- Contrôles de remplacement: final
- Opérateurs de conversion explicite

Autres types

- classe enum
- long long - un entier plus long
- Types entiers étendus
- Unions généralisées
- POD généralisé

Modèles

- Modèles externes
- Alias de modèle
- Modèles variadiques
- Types locaux comme arguments de modèle

Concurrence

- Modèle de mémoire de concurrence
- Initialisation dynamique et destruction avec accès simultané
- Stockage local

Fonctions linguistiques diverses

- Quelle est la valeur de __cplusplus pour C ++ 11?
- Suffixe type de syntaxe de retour
- Prévenir le rétrécissement
- Crochets à angle droit

- static_assert assertions à la compilation
- Littéraux de chaîne bruts
- Les attributs
- Alignement
- Fonctions C99

Extensions de bibliothèque

Général

- unique_ptr
- shared_ptr
- faiblesse_ptr
- Collecte des ordures ABI
- tuple
- Caractères de type
- fonction et lier
- Expressions régulières
- Utilitaires de temps
- Génération de nombres aléatoires
- Scalled allocators

Conteneurs et Algorithmes

- Améliorations des algorithmes
- Amélioration des conteneurs
- unordered_ * conteneurs
- std :: array
- forward_list

Concurrence

- Des filets
- Exclusion mutuelle
- Serrures
- Les variables de condition
- Atomique
- Futures et promesses
- async
- Abandonner un processus

C ++ 14

La norme C ++ 14 est souvent appelée correction de bogue pour C ++ 11. Il ne contient qu'une

liste limitée de modifications dont la plupart sont des extensions des nouvelles fonctionnalités de C ++ 11. Vous trouverez ci-dessous un aperçu des modifications telles qu'elles ont été regroupées dans la FAQ isocpp avec des liens vers une documentation plus détaillée.

Extensions de langue

- Littéraux binaires
- Déduction de type retour généralisée
- decltype (auto)
- Captures de lambda généralisées
- Lambdas génériques
- Modèles variables
- `constexpr` étendu
- L'attribut `[[deprecated]]`
- Séparateurs de chiffres

Extensions de bibliothèque

- Verrouillage partagé
- Littéraux définis par l'utilisateur pour `std::` types
- `std::make_unique`
- Transformation de type `_t` alias
- Adressage des tuples par type (ex: `get<string>(t)`)
- Opérateurs transparents (par exemple, `greater<>(x)`)
- `std::quoted`

Déconseillé / Supprimé

- `std::gets` été déprécié en C ++ 11 et retiré de C ++ 14
- `std::random_shuffle` est obsolète

C ++ 17

Le standard C ++ 17 est complet et a été proposé pour la normalisation. Dans les compilateurs avec support expérimental pour ces fonctionnalités, il est généralement appelé C ++ 1z.

Extensions de langue

- Plier les expressions
- déclarant des arguments de modèle non-type avec `auto`
- Elision de copie garantie
- Déduction du paramètre de modèle pour les constructeurs

- Liaisons structurées
- Espaces de noms imbriqués compacts
- Nouveaux attributs: `[[fallthrough]]` , `[[nodiscard]]` , `[[maybe_unused]]`
- Message par défaut pour `static_assert`
- Initialiseurs dans `if` et `switch`
- Variables en ligne
- `if constexpr`
- Ordre d'évaluation de l'expression garantit
- Allocation de mémoire dynamique pour les données sur-alignées

Extensions de bibliothèque

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` et `extract()` pour les conteneurs associatifs
- Une bibliothèque de système de fichiers avec l'en-tête `<filesystem>` .
- Versions parallèles de la plupart des algorithmes standard (dans l' en -tête `<algorithm>`).
- Ajout de fonctions mathématiques spéciales dans l' en -tête `<cmath>` .
- Déplacement de nœuds entre map <>, unordered_map <>, set <> et unordered_set <>

C ++ 03

Le standard C ++ 03 traite principalement des rapports de défauts du standard C ++ 98. En dehors de ces défauts, il ne fait qu'ajouter une nouvelle fonctionnalité.

Extensions de langue

- Initalisation de la valeur

C ++ 98

C ++ 98 est la première version standardisée de C ++. Comme il a été développé en tant qu'extension de C, de nombreuses fonctionnalités qui séparent C ++ de C sont ajoutées.

Extensions linguistiques (en ce qui concerne C89 / C90)

- Classes, classes dérivées, fonctions de membre virtuel, fonctions de membre const
- Fonction surcharge, surcharge de l'opérateur
- Commentaires sur une seule ligne (a été introduit dans la langue C avec la norme C99)
- Les références
- nouveau et supprimer

- type booléen (a été introduit dans la langue C avec la norme C99)
- des modèles
- espaces de noms
- des exceptions
- moulages spécifiques

Extensions de bibliothèque

- La bibliothèque de modèles standard

C ++ 20

C ++ 20 est la nouvelle norme de C ++, actuellement en développement, basée sur la norme C ++ 17. Ses progrès peuvent être suivis sur le site officiel ISO cpp .

Les fonctionnalités suivantes sont simplement ce qui a été accepté pour la prochaine version du standard C ++, ciblé pour 2020.

Extensions de langue

Aucune extension de langue n'a été acceptée pour le moment.

Extensions de bibliothèque

Aucune extension de bibliothèque n'a été acceptée pour le moment.

Lire La norme ISO C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/2742/la-norme-iso-c-plusplus

Chapitre 56: La règle des trois, cinq et zéro

Examples

Règle de cinq

C ++ 11

C ++ 11 introduit deux nouvelles fonctions membres: le constructeur de déplacement et l'opérateur d'affectation de mouvement. Pour toutes les mêmes raisons pour lesquelles vous souhaitez suivre la règle des trois en C ++ 03, vous devez généralement suivre la règle des cinq en C ++ 11: si une classe requiert UNE des cinq fonctions membres spéciales et si la sémantique est déplacée sont désirés, alors il est très probable qu'ils nécessitent tous les cinq d'entre eux.

Notez, cependant, que ne pas suivre la règle de cinq n'est généralement pas considéré comme une erreur, mais comme une opportunité d'optimisation manquée, tant que la règle de trois est toujours suivie. Si aucun constructeur de déplacement ou opérateur d'attribution de déplacement n'est disponible lorsque le compilateur en utilisera normalement un, il utilisera plutôt la sémantique de la copie si possible, ce qui entraînera une opération moins efficace en raison d'opérations de copie inutiles. Si la sémantique de déplacement n'est pas souhaitée pour une classe, elle n'a pas besoin de déclarer un constructeur de déplacement ou un opérateur d'affectation.

Même exemple que pour la règle de trois:

```
class Person
{
    char* name;
    int age;

public:
    // Destructor
    ~Person() { delete [] name; }

    // Implement Copy Semantics
    Person(Person const& other)
        : name(new char[std::strlen(other.name) + 1])
        , age(other.age)
    {
        std::strcpy(name, other.name);
    }

    Person &operator=(Person const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Person copy(other);
        swap(*this, copy);
        return *this;
    }

    // Implement Move Semantics
    // Note: It is usually best to mark move operators as noexcept
    //       This allows certain optimizations in the standard library
```

```
    //       when the class is used in a container.

    Person(Person&& that) noexcept
        : name(nullptr)               // Set the state so we know it is undefined
        , age(0)
    {
        swap(*this, that);
    }

    Person& operator=(Person&& that) noexcept
    {
        swap(*this, that);
        return *this;
    }

    friend void swap(Person& lhs, Person& rhs) noexcept
    {
        std::swap(lhs.name, rhs.name);
        std::swap(lhs.age, rhs.age);
    }
};
```

L'opérateur d'affectation de copie et de déplacement peut également être remplacé par un seul opérateur d'affectation, qui prend une instance par valeur plutôt qu'une référence ou une référence de valeur pour faciliter l'utilisation de l'idiome de copie et d'échange.

```
Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}
```

Étendre de la règle des trois à la règle des cinq est important pour des raisons de performance, mais n'est pas strictement nécessaire dans la plupart des cas. L'ajout du constructeur de copie et de l'opérateur d'affectation garantit que le déplacement du type n'entraînera pas de fuite de mémoire (la construction par déplacement reviendra simplement à la copie dans ce cas), mais effectuera des copies imprévues.

Règle de zéro

C ++ 11

Nous pouvons combiner les principes de la règle des cinq et de la RAII pour obtenir une interface beaucoup plus simple: la règle de zéro: toute ressource devant être gérée doit être propre. Ce type devrait suivre la règle des cinq, mais tous les utilisateurs de cette ressource ne pas besoin d'écrire l' *une* des cinq fonctions membres spéciales et peut tout simplement `default` tous.

En utilisant la classe `Person` introduite dans l' exemple Rule of Three , nous pouvons créer un objet de gestion de ressources pour les `cstrings` de `cstrings` :

```
class cstring {
private:
    char* p;
```

```
public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

Et une fois que ceci est séparé, notre classe de `Person` devient beaucoup plus simple:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

Les membres spéciaux en `Person` n'ont même pas besoin d'être déclarés explicitement; le compilateur va par défaut ou les supprimer de manière appropriée, en fonction du contenu de `Person` . Par conséquent, ce qui suit est également un exemple de la règle de zéro.

```
struct Person {
    cstring name;
    int arg;
};
```

Si `cstring` devait être un type à déplacement uniquement, avec un opérateur de `delete` / copie constructeur / affectation, alors `Person` ne serait automatiquement déplacé.

Le terme règle de zéro a été introduit par R. Martinho Fernandes

Règle de trois

c ++ 03

La règle de trois stipule que si un type doit avoir un constructeur de copie défini par l'utilisateur, un opérateur d'attribution de copie ou un destructeur, il doit avoir *les trois* .

La raison de cette règle est qu'une classe nécessitant l'un des trois gère une ressource (descripteurs de fichiers, mémoire allouée dynamiquement, etc.) et que les trois sont nécessaires pour gérer cette ressource de manière cohérente. Les fonctions de copie traitent de la manière dont la ressource est copiée entre les objets, et le destructeur détruirait la ressource, conformément aux principes de RAII .

Considérons un type qui gère une ressource de chaîne:

```
class Person
{
    char* name;
    int age;

public:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
       std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};
```

Étant donné que `name` était alloué dans le constructeur, le destructeur le désalloue pour éviter toute fuite de mémoire. Mais que se passe-t-il si un tel objet est copié?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

Tout d'abord, `p1` sera construit. Ensuite, `p2` sera copié à partir de `p1` . Cependant, le constructeur de copie généré par C ++ copiera chaque composant du type tel quel. Ce qui signifie que `p1.name` et `p2.name` pointent tous deux vers la **même** chaîne.

Lorsque `main` fin `main` , les destructeurs seront appelés. Le destructeur de `p2` sera appelé; il va supprimer la chaîne. Ensuite, le destructeur de `p1` sera appelé. Cependant, la chaîne est *déjà supprimée* . L'appel de la `delete` sur la mémoire déjà supprimée entraîne un comportement indéfini.

Pour éviter cela, il est nécessaire de fournir un constructeur de copie approprié. Une approche consiste à implémenter un système compté de référence, dans lequel différentes instances `Person` partagent les mêmes données de chaîne. Chaque fois qu'une copie est effectuée, le nombre de références partagées est incrémenté. Le destructeur décrémente ensuite le compte de référence, ne libérant la mémoire que si le compte est à zéro.

Ou nous pourrions implémenter la sémantique des valeurs et le comportement de copie en profondeur :

```
Person(Person const& other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}
```

```
Person &operator=(Person const& other)
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);            //  assume swap() exchanges contents of *this and copy
    return *this;
}
```

La mise en œuvre de l'opérateur d'assignation de copie est compliquée par la nécessité de libérer un tampon existant. La technique de copie et d'échange crée un objet temporaire contenant un nouveau tampon. Changer le contenu de `*this` et `copy` donne la propriété à la `copy` du tampon d'origine. La destruction de la `copy` , à mesure que la fonction retourne, libère le tampon précédemment détenu par `*this` .

Protection d'auto-assignation

Lors de l'écriture d'un opérateur d'affectation de copie, il est *très* important qu'il soit capable de travailler en cas d'auto-affectation. Autrement dit, il doit permettre cela:

```
SomeType t = ...;
t = t;
```

L'auto-assignation ne se produit généralement pas de manière aussi évidente. Il se produit généralement par une voie détournée par divers systèmes de code, où l'emplacement de l'affectation a simplement deux `Person` pointeurs ou références et n'a aucune idée qu'ils sont le même objet.

Tout opérateur d'affectation de copie que vous écrivez doit pouvoir en tenir compte.

La manière typique de le faire est d’emballer toute la logique d’affectation dans un état comme celui-ci:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Remarque: Il est important de penser à l'auto-affectation et de vous assurer que votre code se comporte correctement lorsqu'il se produit. Cependant, l'auto-assignation est une occurrence très rare et l'optimisation pour l'empêcher peut en fait pessimiser le cas normal. Étant donné que le cas normal est beaucoup plus courant, pessimiser l’auto-assignation peut réduire l’efficacité de votre code (soyez donc prudent en l’utilisant).

À titre d'exemple, la technique normale pour implémenter l'opérateur d'affectation est l' `copy and swap idiom` . La mise en œuvre normale de cette technique ne prend pas la peine de tester pour l'auto-assignation (même si l'auto-attribution est coûteuse car une copie est faite). La raison en est

que la pessimisation du cas normal s’est révélée beaucoup plus coûteuse (comme cela arrive plus souvent).

c ++ 11

Les opérateurs d'attribution de mouvement doivent également être protégés contre l'auto-assignation. Cependant, la logique de nombreux opérateurs de ce type est basée sur `std::swap` , qui peut gérer le transfert de / vers la même mémoire. Donc, si votre logique d'affectation de mouvement n'est rien d'autre qu'une série d'opérations d'échange, vous n'avez pas besoin de protection d'auto-assignation.

Si ce n'est pas le cas, vous *devez* prendre des mesures similaires à celles ci-dessus.

Lire La règle des trois, cinq et zéro en ligne: https://riptutorial.com/fr/cplusplus/topic/1206/la-regle-des-trois--cinq-et-zero

Chapitre 57: Lambdas

Syntaxe

- [*default-capture* , *capture-list*] (*liste des arguments*) *attributs de spécification de jetable* mutable -> *type de retour* { *lambda-body* } // Ordre des spécificateurs et attributs lambda.
- [*capture-list*] (*liste d' arguments*) { *lambda-body* } // Définition lambda commune.
- [=] (*liste d'arguments*) { *lambda-body* } // Capture toutes les variables locales nécessaires par valeur.
- [&] (*liste d'arguments*) { *lambda-body* } // Capture toutes les variables locales nécessaires par référence.
- [*capture-list*] { *lambda-body* } // La liste des arguments et les spécificateurs peuvent être omis.

Paramètres

Paramètre	Détails
capture par défaut	Spécifie comment toutes les variables non répertoriées sont capturées. Peut être `=` (capture par valeur) ou `&` (capture par référence). En cas d'omission, les variables non répertoriées sont inaccessibles dans le *corps lambda* . La *capture par défaut* doit précéder la *liste de capture* .
liste de capture	Spécifie comment les variables locales sont rendues accessibles dans le *corps lambda* . Les variables sans préfixe sont capturées par valeur. Les variables précédées de `&` sont capturées par référence. Dans une méthode de classe, `this` peut être utilisé pour rendre tous ses membres accessibles par référence. Les variables non répertoriées sont inaccessibles, sauf si la liste est précédée d'une *capture* par *défaut* .
liste d'arguments	Spécifie les arguments de la fonction lambda.
mutable	*(facultatif)* Normalement, les variables capturées par valeur sont `const` . Spécifier le `mutable` rend non const. Les modifications apportées à ces variables sont conservées entre les appels.
spécification jet	*(facultatif)* Spécifie le comportement de lancement d'exception de la fonction lambda. Par exemple: `noexcept` ou `throw(std::exception)` .
les attributs	*(facultatif)* Tout attribut pour la fonction lambda. Par exemple, si le *corps lambda* génère toujours une exception, alors `[[noreturn]]` peut être utilisé.
-> *type de retour*	*(facultatif)* Spécifie le type de retour de la fonction lambda. Obligatoire lorsque le type de retour ne peut pas être déterminé par le compilateur.

Paramètre	Détails
corps lambda	Un bloc de code contenant l'implémentation de la fonction lambda.

Remarques

C ++ 17 (le brouillon actuel) introduit `constexpr` lambdas, essentiellement des lambdas qui peuvent être évalués au moment de la compilation. Un lambda est automatiquement `constexpr` s'il satisfait `constexpr` exigences de `constexpr` , mais vous pouvez également le spécifier à l'aide du mot clé `constexpr` :

```
//Explicitly define this lambdas as constexpr
[]() constexpr {
    //Do stuff
}
```

Examples

Qu'est-ce qu'une expression lambda?

Une **expression lambda** fournit un moyen concis de créer des objets de fonction simples. Une expression lambda est une valeur dont l'objet de résultat est appelé objet de fermeture , qui se comporte comme un objet fonction.

Le nom «expression lambda» provient du lambda calculus , un formalisme mathématique inventé dans les années 1930 par Alonzo Church pour étudier les questions de logique et de calculabilité. Le Lambda calcul est à la base du langage LISP , un langage de programmation fonctionnel. Par rapport au lambda calcul et au LISP, les expressions lambda en C ++ partagent les propriétés d'être non nommé et capturent les variables du contexte environnant, mais elles n'ont pas la capacité d'opérer et de renvoyer des fonctions.

Une expression lambda est souvent utilisée comme argument pour les fonctions qui prennent un objet appelable. Cela peut être plus simple que de créer une fonction nommée, qui ne serait utilisée qu'en tant qu'argument. Dans de tels cas, les expressions lambda sont généralement préférées car elles permettent de définir les objets de fonction en ligne.

Un lambda se compose généralement de trois parties: une liste de capture `[]` , une liste de paramètres facultative `()` et un corps `{}` , tous pouvant être vides:

```
[](){}                // An empty lambda, which does and returns nothing
```

Liste de capture

`[]` est la **liste de capture** . Par défaut, les variables de la portée englobante ne sont pas accessibles par un lambda. *La capture d'* une variable la rend accessible à l'intérieur du lambda,

soit comme copie, soit comme référence . Les variables capturées font partie du lambda; contrairement aux arguments de fonction, ils ne doivent pas être transmis lors de l'appel du lambda.

```
int a = 0;                       // Define an integer variable
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                 //      Note: It is the responsibility of the programmer
                                 //      to ensure that a is not destroyed before the
                                 //      lambda is called.
auto b = f();                    // Call the lambda function. a is taken from the capture list
and not passed here.
```

Liste de paramètres

`()` est la **liste de paramètres** , qui est presque la même que dans les fonctions normales. Si le lambda ne prend aucun argument, ces parenthèses peuvent être omises (sauf si vous devez déclarer le `mutable` lambda). Ces deux lambda sont équivalentes:

```
auto call_foo  = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C ++ 14

La liste de paramètres peut utiliser le type d'espace réservé `auto` au lieu des types réels. Ce faisant, cet argument se comporte comme un paramètre de modèle d'un modèle de fonction. Les lambda suivants sont équivalents lorsque vous voulez trier un vecteur en code générique:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
{ return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Corps de fonction

`{}` est le **corps** , qui est le même que dans les fonctions régulières.

Appeler un lambda

L'objet résultat d'une expression lambda est une fermeture , qui peut être appelée à l'aide de l' `operator()` (comme pour les autres objets de fonction):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

Type de retour

Par défaut, le type de retour d'une expression lambda est déduit.

```
[](){ return true; };
```

Dans ce cas, le type de retour est `bool` .

Vous pouvez également spécifier manuellement le type de retour en utilisant la syntaxe suivante:

```
[]() -> bool { return true; };
```

Lambda Mutable

Les objets capturés par valeur dans le lambda sont par défaut immuables. C'est parce que l' `operator()` de l'objet de fermeture généré est `const` par défaut.

```
auto func = [c = 0](){++c; std::cout << c;};  // fails to compile because ++c
                                              // tries to mutate the state of
                                              // the lambda.
```

La modification peut être autorisée en utilisant le mot-clé `mutable` , qui rend l' `operator()` non- `const` l'objet plus proche:

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

Si utilisé avec le type de retour, le `mutable` vient avant lui.

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

Un exemple pour illustrer l'utilité de lambdas

Avant C ++ 11:

C ++ 11

```
// Generic functor used for comparison
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);
```

```
// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

Depuis C ++ 11:

C ++ 11

```
// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value <
threshold; });
```

Spécification du type de retour

Pour les lambdas avec une seule instruction de retour ou plusieurs instructions de retour dont les expressions sont du même type, le compilateur peut en déduire le type de retour:

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}
```

Pour les lambdas avec plusieurs instructions de retour de *différents* types, le compilateur ne peut pas déduire le type de retour:

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

Dans ce cas, vous devez spécifier le type de retour explicitement:

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

Les règles pour cela correspondent aux règles de déduction `auto` . Les Lambdas sans types de retour explicitement spécifiés ne renvoient jamais de références. Par conséquent, si un type de référence est souhaité, il doit également être spécifié explicitement:

```
auto copy = [](X& x) { return x; };       // 'copy' returns an X, so copies its input
auto ref  = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Capturer par valeur

Si vous spécifiez le nom de la variable dans la liste de capture, le lambda le capturera par valeur. Cela signifie que le type de fermeture généré pour le lambda stocke une copie de la variable. Cela nécessite également que le type de la variable soit *constructible par copie* :

```
int a = 0;

[a]() {
    return a;   // Ok, 'a' is captured by value
};
```

C ++ 14

```
auto p = std::unique_ptr<T>(...);

[p]() {         // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

A partir de C ++ 14 on peut initialiser des variables sur place. Cela permet aux types de déplacement uniquement d'être capturés dans le lambda.

C ++ 14

```
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Même si un lambda capture des variables par valeur quand elles sont données par leur nom, ces variables ne peuvent pas être modifiées par défaut dans le corps lambda. En effet, le type de fermeture place le corps lambda dans une déclaration d' `operator() const` .

Le `const` s'applique aux accès aux variables membres du type de fermeture et aux variables capturées qui sont membres de la fermeture (toutes les apparences contraires):

```
int a = 0;

[a]() {
    a = 2;      // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

Pour supprimer le `const` , vous devez spécifier le mot-clé `mutable` sur le lambda:

```
int a = 0;

[a]() mutable {
    a = 2;      // OK, 'a' can be modified
    return a;
};
```

Comme `a` été capturé par valeur, toute modification effectuée en appelant le lambda n'affectera pas `a` . La valeur de `a` été copié dans le lambda quand il a été construit, de sorte que la copie de la lambda `a` est séparée de l'extérieur d' `a` variable.

```
int a = 5 ;
auto plus5Val = [a] (void) { return a + 5 ; } ;
auto plus5Ref = [&a] (void) {return a + 5 ; } ;

a = 7 ;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref() ;
// The result will be "7, value 10, reference 12"
```

Capture généralisée

C ++ 14

Les Lambdas peuvent capturer des expressions, plutôt que de simples variables. Cela permet aux lambdas de stocker les types de déplacement uniquement:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
  p->SomeFunc();
};
```

Cela déplace la variable externe `p` dans la variable de capture lambda, également appelée `p` . `lamb` possède maintenant la mémoire allouée par `make_unique` . Parce que la fermeture contient un type non copiable, cela signifie que l' `lamb` est lui-même non copiable. Mais il peut être déplacé:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

Maintenant, `lamb_move` possède la mémoire.

Notez que `std::function<>` nécessite que les valeurs stockées soient copiables. Vous pouvez écrire votre propre `std::function` nécessitant un déplacement , ou vous pouvez simplement insérer le lambda dans un wrapper `shared_ptr` :

```
auto shared_lambda = [](auto&& f){
  return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
  (auto&&...args)->decltype(auto) {
    return (*spf)(decltype(args)(args)...);
  };
```

```
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

prend notre mouvement seul lambda et place son état dans un pointeur partagé puis retourne un lambda qui *peut* être copié, puis stocké dans une `std::function` ou similaire.

La capture généralisée utilise `auto` déduction `auto` pour le type de la variable. Il déclarera ces captures comme valeurs par défaut, mais elles peuvent également être des références:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
  v += add; //Modifies `a`
};

lamb(20); //`a` becomes 20.
```

La capture de généralisation n'a pas besoin de capturer une variable externe. Il peut capturer une expression arbitraire:

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

Ceci est utile pour donner aux valeurs lambdas arbitraires qu'elles peuvent contenir et éventuellement modifier, sans avoir à les déclarer en externe à la lambda. Bien entendu, cela n'est utile que si vous n'avez pas l'intention d'accéder à ces variables une fois que le lambda a terminé son travail.

Capturer par référence

Si vous faites précéder le nom d'une variable locale par un `&` , la variable sera capturée par référence. Conceptuellement, cela signifie que le type de fermeture de lambda aura une variable de référence, initialisée en référence à la variable correspondante en dehors de la portée de lambda. Toute utilisation de la variable dans le corps lambda fera référence à la variable d'origine:

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
assert(a == 1);
```

Le mot clé `mutable` n'est pas nécessaire car `a` lui-même n'est pas `const` .

Bien sûr, capturer par référence signifie que le lambda **ne doit pas** échapper à la portée des variables qu'il capture. Vous pouvez donc appeler des fonctions qui prennent une fonction, mais vous ne devez pas appeler une fonction qui *stockera* le lambda au-delà de la portée de vos références. Et vous ne devez pas retourner le lambda.

Capture par défaut

Par défaut, les variables locales qui ne sont pas explicitement spécifiées dans la liste de capture ne sont pas accessibles depuis le corps lambda. Cependant, il est possible de capturer implicitement les variables nommées par le corps lambda:

```
int a = 1;
int b = 2;

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

La capture explicite peut toujours être effectuée parallèlement à la capture implicite par défaut. La définition de capture explicite remplacera la capture par défaut:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Lambdas génériques

c ++ 14

Les fonctions Lambda peuvent prendre des arguments de types arbitraires. Cela permet à un lambda d'être plus générique:

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

Ceci est implémenté en C ++ en faisant en sorte que l' `operator()` du type de fermeture surcharge une fonction de modèle. Le type suivant a un comportement équivalent à la fermeture lambda ci-dessus:

```
struct _unique_lambda_type
{
  template<typename T>
  auto operator() (T x) const {return x + x;}
```

```
};
```

Tous les paramètres d'un lambda générique n'ont pas besoin d'être génériques:

```
[](auto x, int y) {return x + y;}
```

Ici, `x` est déduit en fonction du premier argument de la fonction, alors que `y` sera toujours `int` .

Les lambdas génériques peuvent également prendre des arguments en utilisant les règles habituelles pour `auto` et `&` . Si un paramètre générique est considéré comme `auto&&` , c'est une référence *de transfert* au passé dans l' argumentation et non une référence *rvalue* :

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Les fonctions Lambda peuvent être variadiques et transmettre parfaitement leurs arguments:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

ou:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

qui ne fonctionne "correctement" qu'avec des variables de type `auto&&` .

Une bonne raison d'utiliser des lambdas génériques est la syntaxe de visite.

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
  std::cout << e;
});
```

Ici, nous visitons de manière polymorphe; mais dans d'autres contextes, les noms du type que nous passons ne sont pas intéressants:

```
mutex_wrapped<std::ostream&> os = std::cout;
os.write([&](auto&& os){
  os << "hello world\n";
});
```

Répéter le type de `std::ostream&` est ici du bruit; ce serait comme devoir mentionner le type d'une variable chaque fois que vous l'utilisez. Ici, nous créons un visiteur, mais pas un visiteur polymorphe; `auto` est utilisé pour la même raison que vous pourriez utiliser `auto` dans une boucle `for(:)` .

Conversion en pointeur de fonction

Si la liste de capture d'un lambda est vide, le lambda a une conversion implicite en un pointeur de fonction qui prend les mêmes arguments et retourne le même type de retour:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};

using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

Une telle conversion peut également être effectuée en utilisant un opérateur unaire plus:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

L'appel de ce pointeur de fonction se comporte exactement comme l'appel de l' `operator()` sur le lambda. Ce pointeur de fonction ne dépend en aucun cas de l'existence de la fermeture lambda source. Il peut donc survivre à la fermeture de lambda.

Cette fonctionnalité est principalement utile pour utiliser des lambdas avec des API qui traitent des pointeurs de fonction, plutôt que des objets de fonction C ++.

C ++ 14

La conversion en un pointeur de fonction est également possible pour les lambda génériques avec une liste de capture vide. Si nécessaire, la déduction des arguments du modèle sera utilisée pour sélectionner la spécialisation appropriée.

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };
using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter;  // deduces int, int
// note however that the following is ambiguous
// func_ptr sorter_func2 = +sorter;
```

Classe lambda et capture de cette

Une expression lambda évaluée dans une fonction membre d'une classe est implicitement un ami de cette classe:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };
```

```
        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
};
```

Un tel lambda n'est pas seulement un ami de cette classe, il a le même accès que la classe dans laquelle il est déclaré.

Lambdas peut capturer le pointeur `this` qui représente l'instance d'objet sur laquelle la fonction externe a été appelée. Cela se fait en ajoutant `this` à la liste de capture:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

Lorsque `this` est capturé, le lambda peut utiliser les noms de membre de sa classe contenant comme s'il était dans sa classe contenant. Donc, un implicite `this->` est appliqué à ces membres.

Sachez que `this` est capturé par valeur, mais pas la valeur du type. Il est capturé par la valeur de `this` , qui est un *pointeur* . En tant que tel, le lambda ne *possède* pas `this` . Si le lambda out a la durée de vie de l'objet qui l'a créé, le lambda peut devenir invalide.

Cela signifie également que le lambda peut modifier `this` sans être déclaré `mutable` . C'est le pointeur qui est `const` , pas l'objet pointé. C'est-à-dire, à moins que la fonction membre externe ne soit elle-même une fonction `const` .

Sachez également que les clauses de capture par défaut, à la fois `[=]` et `[&]` , captureront *également* `this` implicitement. Et ils le capturent tous deux par la valeur du pointeur. En effet, il est erroné de spécifier `this` dans la liste de capture quand une valeur par défaut est donnée.

C ++ 17

Les Lambdas peuvent capturer une copie de `this` objet, créé au moment où le lambda est créé. Cela se fait en ajoutant `*this` à la liste de capture:

```
class Foo
```

```
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};
```

Portage des fonctions lambda en C ++ 03 à l'aide de foncteurs

Les fonctions Lambda en C ++ sont des sucres syntaxiques qui fournissent une syntaxe très concise pour l'écriture des fonceurs . En tant que tel, une fonctionnalité équivalente peut être obtenue en C ++ 03 (bien que beaucoup plus verbeux) en convertissant la fonction lambda en un foncteur:

```
// Some dummy types:
struct T1 {int dummy;};
struct T2 {int dummy;};
struct R {int dummy;};

// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
  // Use auto because the type of the lambda is unknown.
  auto lambda = [val, &ref](int arg1, int arg2) -> R {
    /* lambda-body */
    return R();
  };
  return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
  // Capture list.
  T1 val;
  T2& ref;

public:
  // Constructor
  inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

  // Functor body
  R operator()(int arg1, int arg2) const {
    /* lambda-body */
    return R();
  }
};
```

```
// Equivalent to use_lambda, but uses a functor (valid C++03).
R use_functor(T1 val, T2 ref) {
  Functor functor(val, ref);
  return functor(12, 27);
}

// Make this a self-contained example.
int main() {
  T1 t1;
  T2 t2;
  use_functor(t1,t2);
  use_lambda(t1,t2);
  return 0;
}
```

Si la fonction lambda est `mutable` faites en sorte que l'opérateur de l'appel soit non-const, c'est-à-dire:

```
R operator()(int arg1, int arg2) /*non-const*/ {
  /* lambda-body */
  return R();
}
```

Lambda récursive

Disons que nous souhaitons écrire `gcd()` Euclid en tant que lambda. En tant que fonction, c'est:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Mais un lambda ne peut pas être récursif, il n'a aucun moyen d'invoquer lui-même. A lambda n'a pas de nom et d' utiliser `this` dans le corps d'un lambda se réfère à un capturé `this` (en supposant que le lambda est créé dans le corps d'une fonction membre, sinon il est une erreur). Alors, comment pouvons-nous résoudre ce problème?

Utilisez `std::function`

Nous pouvons avoir une capture lambda une référence à une `std::function` non encore construite:

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

Cela fonctionne, mais devrait être utilisé avec parcimonie. C'est lent (nous utilisons maintenant l'effacement de type au lieu d'un appel de fonction direct), c'est fragile (copier `gcd` ou renvoyer `gcd` sera cassé car le lambda fait référence à l'objet d'origine), et cela ne fonctionnera pas avec les lambdas génériques.

En utilisant deux pointeurs intelligents:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
*gcd_self = std::make_unique<std::function<int(int, int)>>(
  [gcd_self](int a, int b){
    return b == 0 ? a : (**gcd_self)(b, a%b);
  };
};
```

Cela ajoute beaucoup d'indirection (qui est une surcharge), mais il peut être copié / retourné, et toutes les copies partagent l'état. Il vous permet de retourner le lambda et est par ailleurs moins fragile que la solution ci-dessus.

Utiliser un combinateur Y

Avec l'aide d'une structure utilitaire courte, nous pouvons résoudre tous ces problèmes:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // we pass ourselves to f, then the arguments.
        // the lambda should take the first argument as `auto&& recurse` or similar.
        return f(*this, std::forward<Args>(args)...);
    }
};
// helper function that deduces the type of the lambda:
template <class F>
y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
    return {std::forward<F>(f)};
}
// (Be aware that in C++17 we can do better than a `make_` function)
```

nous pouvons implémenter notre `gcd` comme:

```
auto gcd = make_y_combinator(
  [](auto&& gcd, int a, int b){
    return b == 0 ? a : gcd(b, a%b);
  }
);
```

Le `y_combinator` est un concept du calcul lambda qui vous permet d'avoir de la récursivité sans pouvoir vous nommer tant que vous n'êtes pas défini. C'est exactement le problème des lambda.

Vous créez un lambda qui prend "recurse" comme premier argument. Lorsque vous voulez vous remettre en forme, vous transmettez les arguments à récurer.

Le `y_combinator` retourne alors un objet fonction qui appelle cette fonction avec ses arguments, mais avec un objet "recurse" approprié (à savoir le `y_combinator` lui-même) comme premier argument. Il transmet également le reste des arguments que vous appelez le `y_combinator` au lambda.

En bref:

```
auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
  // write body that processes some arguments
  // when you want to recurse, call recurse(some other arguments)
});
```

et vous avez une récursivité dans un lambda sans restriction sérieuse ni surcharge significative.

Utilisation de lambdas pour le déballage du pack de paramètres en ligne

C ++ 14

Le déballage du paquet de paramètres nécessite traditionnellement d’écrire une fonction d’aide à chaque fois que vous le souhaitez.

Dans cet exemple de jouet:

```
template<std::size_t...Is>
void print_indexes( std::index_sequence<Is...> ) {
  using discard=int[];
  (void)discard{0,((void)(
    std::cout << Is << '\n' // here Is is a compile-time constant.
  ),0)...};
}
template<std::size_t I>
void print_indexes_upto() {
  return print_indexes( std::make_index_sequence<I>{} );
}
```

Le `print_indexes_upto` veut créer et décompresser un paquet de paramètres d'index. Pour ce faire, il doit appeler une fonction d'assistance. Chaque fois que vous voulez décompresser un pack de paramètres que vous avez créé, vous devez créer une fonction d'assistance personnalisée pour le faire.

Cela peut être évité avec les lambda.

Vous pouvez décompresser les paquets de paramètres dans un ensemble d'appels de lambda, comme ceci:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    using discard=int[];
    (void)discard{0,(void(
      f( index<Is> )
    ),0)...};
  };
}
```

```
template<std::size_t N>
auto index_over(index_t<N> = {}) {
  return index_over( std::make_index_sequence<N>{} );
}
```

C ++ 17

Avec les expressions de pliage, `index_over()` peut être simplifié pour:

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    ((void)(f(index<Is>)), ...);
  };
}
```

Une fois que vous avez fait cela, vous pouvez l'utiliser pour remplacer la décompression manuelle des paquets de paramètres par une seconde surcharge dans un autre code, ce qui vous permet de décompresser les paquets de paramètres "inline":

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
  using T = std::remove_reference_t<Tup>;
  using std::tuple_size;
  auto from_zero_to_N = index_over< tuple_size<T>{} >();

  from_zero_to_N(
    [&](auto i){
      using std::get;
      f( get<i>( std::forward<Tup>(tup) ) );
    }
  );
}
```

L' `auto i` passée au lambda par l' `index_over` est un `std::integral_constant<std::size_t, ???>` . Cela a une conversion `constexpr` en `std::size_t` qui ne dépend pas de l'état de `this` , donc nous pouvons l'utiliser comme une constante de compilation, comme quand nous la passons à `std::get<i>` ci-dessus.

Pour revenir à l'exemple de jouet en haut, réécrivez-le comme suit:

```
template<std::size_t I>
void print_indexes_upto() {
  index_over(index<I>)([](auto i){
    std::cout << i << '\n'; // here i is a compile-time constant
  });
}
```

ce qui est beaucoup plus court, et conserve la logique dans le code qui l’utilise.

Exemple en direct avec lequel jouer.

Lire Lambdas en ligne: https://riptutorial.com/fr/cplusplus/topic/572/lambdas

Chapitre 58: Le pointeur

Remarques

Le pointeur `this` est un mot-clé pour C ++, il n'y a donc pas de bibliothèque nécessaire pour l'implémenter. Et ne pas oublier c'est un pointeur! `this` Donc, vous ne pouvez pas faire:

```
this.someMember();
```

Lorsque vous accédez à des fonctions membres ou à des variables membres à partir de pointeurs à l'aide du symbole flèche `->` :

```
this->someMember();
```

Autres liens utiles pour mieux comprendre le pointeur `this` :

Quel est le pointeur 'this'?

http://www.geeksforgeeks.org/this-pointer-in-c/

https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm

Examples

ce pointeur

Toutes les fonctions membres non statiques ont un paramètre caché, un pointeur sur une instance de la classe, nommé `this` ; Ce paramètre est inséré en silence au début de la liste de paramètres et entièrement géré par le compilateur. Lorsqu'un membre de la classe est accédé à l'intérieur d'une fonction membre, il est accessible silencieusement à travers `this` ; Cela permet au compilateur d'utiliser une fonction membre non statique unique pour toutes les instances et permet à une fonction membre d'appeler d'autres fonctions membres de manière polymorphe.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
// As the constructor is responsible for creating the object, 'this' will not be "fully"
```

```
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int  ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int  ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

Dans un constructeur, `this` peut être utilisé en toute sécurité (implicitement ou explicitement) pour accéder à tout champ déjà initialisé ou à tout champ d'une classe parente; inversement, (implicitement ou explicitement) l'accès à tous les champs qui n'ont pas encore été initialisés, ou à tout champ d'une classe dérivée, est dangereux (la classe dérivée n'étant pas encore construite et ses champs n'étant ni initialisés ni existants). Il est également dangereux d'appeler des fonctions membres virtuelles via `this` dans le constructeur, comme toutes les fonctions de classe dérivée ne seront pas pris en compte (en raison de la classe dérivée pas encore en cours de construction, et donc son constructeur ne mettre à jour encore le vtable).

Notez également que, dans un constructeur, le type de l'objet est le type que ce constructeur construit. Cela est vrai même si l'objet est déclaré comme un type dérivé. Par exemple, dans l'exemple ci-dessous, `ctd_good` et `ctd_bad` sont de type `CtorThisBase` dans `CtorThisBase()` , et tapez `CtorThis` dans `CtorThis()` , même si leur type canonique est `CtorThisDerived` . Au fur et à mesure que les classes dérivées sont construites autour de la classe de base, l'instance passe progressivement par la hiérarchie des classes jusqu'à ce qu'elle devienne une instance entièrement construite du type prévu.

```
class CtorThisBase {
    short s;

  public:
    CtorThisBase() : s(516) {}
};

class CtorThis : public CtorThisBase {
    int i, j, k;

  public:
```

```
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

  public:
    CtorThisDerived()       : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

Avec ces classes et fonctions membres:

- Dans le bon constructeur, pour `ctd_good` :
  - `CtorThisBase` est entièrement construit au moment où le constructeur `CtorThis` est entré. Par conséquent, `s` est dans un état valide lors de l'initialisation `i` , et peut donc être consulté.
  - `i` est initialisé avant que `j(this->i)` soit atteint. Par conséquent, `i` dans un état valide lors de l'initialisation `j` , et peut donc être consulté.
  - `j` est initialisé avant que `k(j)` soit atteint. Par conséquent, `j` est dans un état valide lors de l'initialisation de `k` , et peut donc être consulté.
- Dans le mauvais constructeur, pour `ctd_bad` :
  - `k` est initialisé après que `j(this->k)` est atteint. Par conséquent, `k` est dans un état invalide lors de l'initialisation `j` , et y accéder entraîne un comportement indéfini.
  - `CtorThisDerived` n'est construit qu'après la construction de `CtorThis` . Par conséquent, `b` est dans un état invalide lors de l'initialisation de `k` , et l'accès à celui-ci entraîne un comportement indéfini.
  - L'objet `ctd_bad` est toujours un `CtorThis` jusqu'à ce qu'il quitte `CtorThis()` , et ne sera pas mis à jour pour utiliser la `CtorThisDerived` de CtorThisDerived jusqu'à `CtorThisDerived()` que `CtorThisDerived()` . Par conséquent, `virt_func()` appellera `CtorThis::virt_func()` , qu'il soit destiné à appeler cela ou `CtorThisDerived::virt_func()` .

Utiliser le pointeur this pour accéder aux données des membres

Dans ce contexte, l'utilisation du pointeur `this` n'est pas absolument nécessaire, mais cela rendra votre code plus clair pour le lecteur, en indiquant qu'une fonction ou une variable donnée est un membre de la classe. Un exemple dans cette situation:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
  public:
    Class();
    ~Class();
    int getPrivateNumber () const;
  private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

Voyez-le en action ici .

Utiliser le pointeur this pour différencier les données de membre et les paramètres

Ceci est une stratégie très utile pour différencier les données des membres des paramètres ... Prenons l'exemple suivant:

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
* @class Dog
*   @member name
*       Dog's name
*   @function bark
*       Dog Barks!
*   @function getName
*       To Get Private
*       Name Variable
*/
class Dog
{
```

```
 public:
    Dog(std::string name);
    ~Dog();
    void  bark() const;
    std::string  getName() const;
 private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
    *  this->name is the
    *  name variable from
    *  the class dog . and
    *  name is from the
    *  parameter of the function
    */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
  cout << "BARK" << endl;
}

std::string  Dog::getName() const
{
    return this->name;
}


int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}
```

Vous pouvez voir ici dans le constructeur que nous exécutons ce qui suit:

```
this->name = name;
```

Ici, vous pouvez voir que nous associons le nom du paramètre au nom de la variable privée de la classe Dog (this-> name).

Pour voir la sortie du code ci-dessus: http://cpp.sh/75r7

ce Pointer CV-Qualifiers

`this` peut aussi être qualifié CV, comme n'importe quel autre pointeur. Cependant, étant donné que `this` paramètre n'est pas répertorié dans la liste de paramètres, une syntaxe spéciale est requise pour cela; les qualificateurs cv sont listés après la liste des paramètres, mais avant le corps de la fonction.

```
struct ThisCVQ {
    void no_qualifier()                {} // "this" is: ThisCVQ*
    void  c_qualifier() const          {} // "this" is: const ThisCVQ*
    void  v_qualifier() volatile       {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

Comme `this` s'agit d'un paramètre, une fonction peut être surchargée en fonction de `this` ou de ces qualificateurs cv .

```
struct CVOverload {
    int func()                { return    3; }
    int func() const          { return   33; }
    int func() volatile       { return  333; }
    int func() const volatile { return 3333; }
};
```

Lorsque c'est `this const` (y compris `const volatile`), la fonction est incapable d'écrire à des variables membres à travers elle, que ce soit explicitement ou implicitement. La seule exception à cette règle concerne les variables de membre `mutable` , qui peuvent être écrites indépendamment de la constance. De ce fait, `const` est utilisé pour indiquer que la fonction membre ne modifie pas l'état logique de l'objet (la façon dont l'objet apparaît dans le monde extérieur), même s'il modifie l'état physique (la façon dont l'objet se présente sous le capot)).

> L'état logique est la manière dont l'objet apparaît aux observateurs extérieurs. Il n'est pas directement lié à l'état physique et peut même ne pas être stocké en tant qu'état physique. Tant que les observateurs extérieurs ne peuvent voir aucun changement, l'état logique est constant, même si vous retournez chaque bit de l'objet.
>
> L'état physique, également appelé état binaire, est la manière dont l'objet est stocké en mémoire. Ceci est le nitty-graffy de l'objet, les 1 et 0 bruts qui composent ses données. Un objet n'est physiquement constant que si sa représentation en mémoire ne change jamais.

Notez que C ++ base la `const` sur l'état logique et non sur l'état physique.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somehow(const Param& p);

    // ...

  public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somehow(p);
    }

    void change_state(Param p) {
        modify_somehow(p);
        state_changed = true;
```

```
    }

    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};
ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    //  by caching the result, so it doesn't need to be recalculated every time the function
    //  is called.  This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Notez que si vous *pouvez* utiliser techniquement `const_cast` sur `this` pour le rendre non-cv-qualifié, vous avez vraiment, ne devrait ***vraiment*** pas, et devrait utiliser `mutable` à la place. Un `const_cast` est susceptible d'appeler un comportement indéfini lorsqu'il est utilisé sur un objet qui *est* réellement `const` , alors que `mutable` est conçu pour être sûr à utiliser. Il est cependant possible que vous rencontriez ceci dans un code extrêmement ancien.

Une exception à cette règle consiste à définir des accesseurs non qualifiés en cv en termes d'accesseurs `const` ; comme il est garanti que l'objet ne sera pas `const` si la version non qualifiée de cv est appelée, il n'y a aucun risque d'UB.

```
class CVAccessor {
    int arr[5];

  public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};
```

Cela évite la duplication inutile du code.

Comme avec les pointeurs réguliers, si `this` est `volatile` (y compris `const volatile`), il est chargé depuis la mémoire à chaque fois qu’il est accédé, au lieu d’être mis en cache. Cela a les mêmes effets sur l'optimisation que de déclarer tout autre pointeur `volatile` , il faut donc faire attention.

Notez que si une instance est cv-qualifié, les seules fonctions de membres , il est autorisé à accéder sont des fonctions membres dont `this` pointeur est au moins aussi cv-qualifié l'instance elle - même:

- Les instances non-cv peuvent accéder à toutes les fonctions membres.
- `const`

instances `const` peuvent accéder aux fonctions `const` et `const volatile` .

- `volatile` instances `volatile` peuvent accéder aux fonctions `volatile` et `const volatile` .
- `const volatile` instances `const volatile` peuvent accéder à `const volatile` fonctions `const volatile` .

C'est l'un des principes clés de la correction des `const` .

```
struct CVAccess {
    void    func()                {}
    void  func_c() const          {}
    void  func_v() volatile       {}
    void func_cv() const volatile {}
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();    // Error.
c_cva.func_c();  // Good.
c_cva.func_v();  // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();    // Error.
v_cva.func_c();  // Error.
v_cva.func_v();  // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();    // Error.
cv_cva.func_c();  // Error.
cv_cva.func_v();  // Error.
cv_cva.func_cv(); // Good.
```

ce Pointer Ref-Qualifiers

C ++ 11

De même pour `this` cv-qualificatifs, nous pouvons également appliquer *ref-qualificatifs* à `*this` . Ref-qualificatifs sont utilisés pour choisir entre la sémantique de référence normale et rvalue, ce qui permet au compilateur d'utiliser soit copier ou déplacer la sémantique en fonction qui sont plus appropriés et sont appliqués à `*this` lieu de `this` .

Notez que malgré les qualificateurs de référence utilisant la syntaxe de référence, `this` reste un pointeur. Notez également que les qualificateurs de ref ne modifient pas réellement le type de `*this` ; il est juste plus facile de décrire et de comprendre leurs effets en les regardant comme s'ils le faisaient.

```
struct RefQualifiers {
    std::string s;
```

```
    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
    void func() &  { std::cout << "Accessed on normal instance "    << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() &  { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func();              // Output:  Accessed on normal instance Fred
RefQualifiers{}.func(); // Output:  Accessed on temporary instance The nameless one
```

Une fonction membre ne peut pas avoir de surcharge avec et sans qualificatif ref; le programmeur doit choisir entre l'un ou l'autre. Heureusement, les qualificatifs cv peuvent être utilisés en conjonction avec les qualificatifs ref, ce qui permet de respecter les règles d' exactitude des `const` .

```
struct RefCV {
    void func() &                {}
    void func() &&               {}
    void func() const&           {}
    void func() const&&          {}
    void func() volatile&        {}
    void func() volatile&&       {}
    void func() const volatile&  {}
    void func() const volatile&& {}
};
```

Lire Le pointeur en ligne: https://riptutorial.com/fr/cplusplus/topic/7146/le-pointeur

Chapitre 59: Les attributs

Syntaxe

- [[détails]]: attribut simple sans argument
- [[détails (arguments)]]: attribut avec arguments
- __attribute (détails): spécifique GCC / Clang / IBM non standard
- __declspec (détails): Spécifique à MSVC non standard

Examples

[[non-retour]]

C ++ 11

C ++ 11 a introduit l'attribut `[[noreturn]]` . Il peut être utilisé pour une fonction pour indiquer que la fonction ne retourne pas à l'appelant soit en exécutant une instruction *return* , soit en atteignant la fin si c'est un corps (il est important de noter que cela ne s'applique pas aux fonctions `void` , car retourne à l'appelant, ils ne renvoient juste aucune valeur). Une telle fonction peut se terminer en appelant `std::terminate` ou `std::exit` ou en lançant une exception. Il convient également de noter qu'une telle fonction peut être `longjmp` exécutant `longjmp` .

Par exemple, la fonction ci-dessous jettera toujours une exception ou appellera `std::terminate` , ce qui en fait un bon candidat pour `[[noreturn]]` :

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

Ce type de fonctionnalité permet au compilateur de mettre fin à une fonction sans instruction de retour s'il sait que le code ne sera jamais exécuté. Ici, comme l'appel à `ownAssertFailureHandler` (défini ci-dessus) dans le code ci-dessous ne reviendra jamais, le compilateur n'a pas besoin d'ajouter de code en dessous de cet appel:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
    }
```

```
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}
```

Il s'agit d'un comportement indéfini si la fonction est effectivement renvoyée. Par conséquent, les éléments suivants ne sont pas autorisés:

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
        ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}
```

Notez que le `[[noreturn]]` est principalement utilisé dans les fonctions vides. Cependant, ce n'est pas une exigence, permettant aux fonctions d'être utilisées dans la programmation générique:

```
template<class InconsistencyHandler>
double fortyTwoDivideBy(int i) {
    if (i == 0)
         i = InconsistencyHandler::correct(i);
    return 42. / i;
}

struct InconsistencyThrower {
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown
inconsistency"s); }
}

struct InconsistencyChangeToOne {
    static int correct(int i) { return 1; }
}

double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);
```

Les fonctions de bibliothèque standard suivantes ont cet attribut:

- std :: abort
- std :: exit
- std :: quick_exit
- std :: unknown
- std :: terminate
- std :: rethrow_exception
- std :: throw_with_nested
- std :: nested_exception :: rethrow_nested

[[tomber dans]]

C ++ 17

Chaque fois qu'un `case` se termine dans un `switch` , le code du cas suivant sera exécuté. Ce dernier peut être empêché en utilisant l'instruction ´break`. Comme ce soi-disant comportement

trompeur peut introduire des bogues lorsqu'il n'est pas prévu, plusieurs compilateurs et analyseurs statiques émettent un avertissement à ce sujet.

A partir de C ++ 17, un attribut standard a été introduit pour indiquer que l'avertissement n'est pas nécessaire lorsque le code est censé tomber. Les compilateurs peuvent donner des avertissements en toute sécurité lorsqu'un dossier est terminé sans `break` ou `[[fallthrough]]` et possède au moins une instruction.

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

Voir la proposition pour des exemples plus détaillés sur la manière dont `[[fallthrough]]` peut être utilisé.

[[déconseillé]] et [[déconseillé ("raison")]]

C ++ 14

C ++ 14 a introduit un moyen standard de déconseiller les fonctions via des attributs. `[[deprecated]]` peut être utilisé pour indiquer qu'une fonction est obsolète. `[[deprecated("reason")]]` permet d'ajouter une raison spécifique pouvant être affichée par le compilateur.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the
next release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

Cet attribut peut être appliqué à:

- la déclaration d'une classe
- un nom de typedef
- une variable
- un membre de données non statique
- une fonction
- une énumération
- une spécialisation de template

(réf. c ++ 14 brouillon standard : 7.6.5 Attribut obsolète)

[[nodiscard]]

C ++ 17

L'attribut `[[nodiscard]]` peut être utilisé pour indiquer que la valeur de retour d'une fonction ne doit pas être ignorée lors d'un appel de fonction. Si la valeur de retour est ignorée, le compilateur doit en avertir. L'attribut peut être ajouté à:

- Une définition de fonction
- Un type

L'ajout de l'attribut à un type a le même comportement que l'ajout de l'attribut à chaque fonction unique qui renvoie ce type.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0);                    // Just to make comments clear!
    ++i;                               // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i;                               // i == 2
    onExit([&i]{ --i; });              // BUG: Reducing by 1 directly
                                       //      Compiler warning expected
    std::cout << i << std::end;        // Expected: 2, Real: 1
}
```

Voir la proposition pour des exemples plus détaillés sur la façon dont `[[nodiscard]]` peut être utilisé.

Remarque: Les détails d'implémentation de `Finally` / `onExit` sont omis dans l'exemple, voir Finally / ScopeExit .

[[peut-être_unused]]

L'attribut `[[maybe_unused]]` est créé pour indiquer dans le code qu'une certaine logique pourrait ne pas être utilisée. Ceci est souvent lié aux conditions du préprocesseur où cela pourrait être utilisé ou ne pas être utilisé. Comme les compilateurs peuvent donner des avertissements sur les variables inutilisées, cela permet de les supprimer en indiquant leur intention.

Un exemple typique de variables nécessaires dans les versions de débogage, même si elles sont inutiles en production, sont les valeurs de retour indiquant le succès. Dans les versions de débogage, la condition doit être affirmée, bien que, en production, ces assertions aient été supprimées.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the
map
```

Un exemple plus complexe est celui des différentes fonctions d'aide qui se trouvent dans un

espace de noms sans nom. Si ces fonctions ne sont pas utilisées lors de la compilation, un compilateur peut les avertir. Idéalement, vous voudriez les garder avec les mêmes balises de préprocesseur que l'appelant, bien que cela puisse devenir complexe, l'attribut `[[maybe_unused]]` est une alternative plus `[[maybe_unused]]` maintenir.

```
namespace {
    [[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
    // TODO: Reuse this on BSD, MAC ...
    [[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#if OS == "WINDOWS"
      return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
      return createLinuxConfigFilePath(relativePath);
#else
#error "OS is not yet supported"
#endif
}
```

Voir la proposition pour des exemples plus détaillés sur la façon dont `[[maybe_unused]]` peut être utilisé.

Lire Les attributs en ligne: https://riptutorial.com/fr/cplusplus/topic/5251/les-attributs

Chapitre 60: Les itérateurs

Examples

C itérateurs (pointeurs)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

Ce code afficherait les nombres 1 à 5, un sur chaque ligne comme ceci:

> 1
> 2
> 3
> 4
> 5

Le briser

```
const int array[] = { 1, 2, 3, 4, 5 };
```

Cette ligne crée un nouveau tableau d'entiers avec 5 valeurs. Les tableaux C ne sont que des pointeurs vers la mémoire où chaque valeur est stockée ensemble dans un bloc contigu.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

Ces lignes créent deux pointeurs. Le premier pointeur reçoit la valeur du pointeur de tableau, qui est l'adresse du premier élément du tableau. L'opérateur `sizeof` utilisé sur un tableau C renvoie la

taille du tableau en octets. Divisé par la taille d'un élément, cela donne le nombre d'éléments dans le tableau. Nous pouvons l'utiliser pour trouver l'adresse du bloc *après* le tableau.

```
for (const int* i = first; i < afterLast; ++i) {
```

Nous créons ici un pointeur que nous utiliserons comme itérateur. Il est initialisé avec l'adresse du premier élément que nous voulons parcourir, et nous allons continuer à itérer tant que `i` est inférieur à `afterLast` , ce qui signifie aussi longtemps que `i` pointe vers une adresse dans `array` .

```
    std::cout << *i << std::endl;
```

Enfin, dans la boucle , nous pouvons accéder à la valeur de notre itérateur `i` pointe par déréférencement elle. Ici, l'opérateur de déréférence `*` renvoie la valeur à l'adresse dans `i` .

Vue d'ensemble

Les itérateurs sont des positions

Les itérateurs sont un moyen de naviguer et d'opérer sur une séquence d'éléments et constituent une extension généralisée des pointeurs. Conceptuellement, il est important de se rappeler que les itérateurs sont des positions et non des éléments. Par exemple, prenez la séquence suivante:

```
A B C
```

La séquence contient trois éléments et quatre positions

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Les éléments sont des choses dans une séquence. Les positions sont des endroits où des opérations significatives peuvent avoir lieu dans la séquence. Par exemple, on insère dans une position, *avant* ou *après l'* élément A, pas dans un élément. Même la suppression d'un élément (`erase(A)`) se fait en trouvant d'abord sa position, puis en la supprimant.

Des itérateurs aux valeurs

Pour convertir une position en une valeur, un itérateur est *déréférencé* :

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

On peut penser à un itérateur en tant que déréférencement à la valeur à laquelle il fait référence dans la séquence. Ceci est particulièrement utile pour comprendre pourquoi vous ne devriez jamais déréférencer l'itérateur `end()` dans une séquence:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           +-- An iterator here has no value. Do not dereference it!
  +-------------- An iterator here dereferences to the value A.
```

Dans toutes les séquences et tous les conteneurs trouvés dans la bibliothèque standard C ++, `begin()` renvoie un itérateur à la première position et `end()` renvoie un itérateur à un autre après la dernière position (*pas* la dernière!). Par conséquent, les noms de ces itérateurs dans les algorithmes sont souvent étiquetés en `first` et en `last` :

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           |
  +- first    +- last
```

Il est également possible d'obtenir un itérateur à *n'importe quelle séquence* , car même une séquence vide contient au moins une position:

```
+---+
|   |
+---+
```

Dans une séquence vide, `begin()` et `end()` auront la même position et *aucune des deux ne* pourra être déréférencée:

```
+---+
|   |
+---+
  ↑
  |
  +- empty_sequence.begin()
  |
  +- empty_sequence.end()
```

La visualisation alternative des itérateurs est qu'ils marquent les positions *entre les* éléments:

```
+---+---+---+
| A | B | C |
+---+---+---+
↑   ^   ^   ↑
|           |
+- first    +- last
```

et déréférencer un itérateur renvoie une référence à l'élément venant après l'itérateur. Certaines situations où cette vue est particulièrement utile sont les suivantes:

- `insert` opérations d'insertion insèrent des éléments dans la position indiquée par l'itérateur,
- `erase` opérations d' `erase` renverront un itérateur correspondant à la même position que celle

passée,
- un itérateur et son itérateur inverse correspondant sont situés dans la même position entre les éléments

Itérateurs non valides

Un itérateur devient *invalidé* si (par exemple, au cours d'une opération) sa position ne fait plus partie d'une séquence. Un itérateur invalidé ne peut pas être déréférencé tant qu'il n'a pas été réaffecté à une position valide. Par exemple:

```
std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid
```

Les nombreux algorithmes et fonctions de membre de la séquence dans la bibliothèque standard C ++ ont des règles régissant le moment où les itérateurs sont invalidés. Chaque algorithme est différent dans la manière dont il traite (et invalide) les itérateurs.

Naviguer avec les itérateurs

Comme nous le savons, les itérateurs servent à naviguer dans les séquences. Pour ce faire, un itérateur doit migrer sa position tout au long de la séquence. Les itérateurs peuvent avancer dans la séquence et certains peuvent reculer:

```
auto first = my_vector.begin();
++first;                                             // advance the iterator 1 position
std::advance(first, 1);                              // advance the iterator 1 position
first = std::next(first);                            // returns iterator to the next element
std::advance(first, -1);                             // advance the iterator 1 position
backwards
first = std::next(first, 20);                        // returns iterator to the element 20
position forward
first = std::prev(first, 5);                         // returns iterator to the element 5
position backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two
iterators.
```

Notez que le second argument de std :: distance devrait être accessible depuis le premier (ou, en d’autres termes, `first` devrait être inférieur ou égal à la `second`).

Même si vous pouvez exécuter des opérateurs arithmétiques avec des itérateurs, toutes les opérations ne sont pas définies pour tous les types d'itérateurs. `a = b + 3;` fonctionnerait pour les itérateurs à accès aléatoire, mais ne fonctionnerait pas pour les itérateurs directs ou bidirectionnels, qui peuvent toujours être avancés de 3 positions avec quelque chose comme `b = a; ++b; ++b; ++b;` . Il est donc recommandé d'utiliser des fonctions spéciales si vous n'êtes pas sûr

du type d'itérateur (par exemple, dans une fonction de modèle acceptant un itérateur).

Concepts d'itérateur

Le standard C ++ décrit plusieurs concepts d'itérateurs différents. Ceux-ci sont regroupés en fonction de leur comportement dans les séquences auxquelles ils se réfèrent. Si vous connaissez le concept d'un *modèle d'* itérateur (se comporte comme), vous pouvez être assuré du comportement de cet itérateur *indépendamment de la séquence à laquelle il appartient* . Ils sont souvent décrits dans l'ordre, du plus restrictif au moins restrictif (car le concept d'itérateur suivant est un pas meilleur que son prédécesseur):

- Les itérateurs d'entrée: peuvent être déréférencés *seulement une fois* par position. Ne peut avancer que d'une seule position à la fois.
- Itérateurs directs: un itérateur d'entrée pouvant être déréférencé un nombre illimité de fois.
- Itérateurs: Un Bidirectionnel iterator avant qui peut également avancer en *arrière* une position à la fois.
- Randator Access Iterators: Un itérateur bidirectionnel qui peut avancer ou reculer d'un nombre quelconque de positions à la fois.
- Itérateurs contigus (depuis C ++ 17): itérateur à accès aléatoire garantissant que les données sous-jacentes sont contiguës en mémoire.

Les algorithmes peuvent varier en fonction du concept modélisé par les itérateurs. Par exemple, bien que `random_shuffle` puisse être implémenté pour les itérateurs avant, une variante plus efficace nécessitant des itérateurs à accès aléatoire pourrait être fournie.

Traits d'itérateur

Les traits d'itérateur fournissent une interface uniforme aux propriétés des itérateurs. Ils permettent de récupérer la valeur, la différence, le pointeur, les types de référence et également la catégorie d'itérateur:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val)  {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

La catégorie d'itérateur peut être utilisée pour spécialiser des algorithmes:

```
template<class BidirIt>
void test(BidirIt a, std::bidirectional_iterator_tag)  {
    std::cout << "Bidirectional iterator is used" << std::endl;
}
```

```
template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag)  {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a)  {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Les catégories d'itérateurs sont essentiellement des concepts d'itérateurs, sauf que les itérateurs contigus ne disposent pas de leur propre balise, car il a été trouvé que le code est cassé.

Itérateurs inverses

Si nous voulons parcourir en arrière une liste ou un vecteur, nous pouvons utiliser un `reverse_iterator` . Un itérateur inversé est créé à partir d'un itérateur bidirectionnel ou à accès aléatoire qu'il conserve en tant que membre et accessible via `base()` .

Pour itérer en arrière, utilisez `rbegin()` et `rend()` comme itérateurs pour la fin de la collection et le début de la collection, respectivement.

Par exemple, pour itérer l'utilisation en arrière:

```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

Un itérateur inversé peut être converti en un itérateur avant via la fonction membre `base()` . La relation est que l'itérateur inversé référence un élément après l'itérateur `base()` :

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators

 +---+---+---+---+---+---+---+
 |   | 1 | 2 | 3 | 4 | 5 |   |
 +---+---+---+---+---+---+---+
   ↑   ↑               ↑   ↑
   |   |               |   |
rend() |         rbegin()  end()
       |                   rbegin().base()
     begin()
     rend().base()
```

Dans la visualisation où les itérateurs marquent les positions entre les éléments, la relation est plus simple:

```
  +---+---+---+---+---+
  | 1 | 2 | 3 | 4 | 5 |
```

```
  +---+---+---+---+---+
  ↑                   ↑
  |                   |
  |                 end()
  |                 rbegin()
begin()             rbegin().base()
rend()
rend().base()
```

Itérateur de vecteur

`begin` renvoie un `iterator` au premier élément du conteneur de séquence.

`end` renvoie un `iterator` au premier élément après la fin.

Si l'objet vectoriel est `const` , les deux `begin` et `end` renvoient un `const_iterator` . Si vous souhaitez qu'un `const_iterator` soit renvoyé même si votre vecteur n'est pas `const` , vous pouvez utiliser `cbegin` et `cend` .

Exemple:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };  //intialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Sortie:

> 1 2 3 4 5

Itérateur de carte

Un itérateur sur le premier élément du conteneur.

Si un objet map est qualifié en const, la fonction renvoie un `const_iterator` . Sinon, il renvoie un `iterator` .

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;

// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Sortie:

> a => 200
> b => 100
> c => 300

Itérateurs de flux

Les itérateurs de flux sont utiles lorsque vous devez lire une séquence ou imprimer des données formatées à partir d'un conteneur:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2     3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));

// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimeted by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

Le programme d'exemple imprimera `1 -- 2 -- 3 -- 4 --` sur la sortie standard.

Ecrivez votre propre itérateur avec générateur

Un modèle courant dans d'autres langages consiste à avoir une fonction qui produit un "flux" d'objets et à pouvoir utiliser le code de boucle pour la parcourir.

Nous pouvons modéliser ceci en C ++ comme

```
template<class T>
struct generator_iterator {
  using difference_type=std::ptrdiff_t;
  using value_type=T;
  using pointer=T*;
  using reference=T;
  using iterator_category=std::input_iterator_tag;
  std::optional<T> state;
  std::function< std::optional<T>() > operation;
  // we store the current element in "state" if we have one:
  T operator*() const {
    return *state;
  }
  // to advance, we invoke our operation.  If it returns a nullopt
  // we have reached the end:
  generator_iterator& operator++() {
    state = operation();
    return *this;
```

```
  }
  generator_iterator operator++(int) {
    auto r = *this;
    ++(*this);
    return r;
  }
  // generator iterators are only equal if they are both in the "end" state:
  friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
    if (!lhs.state && !rhs.state) return true;
    return false;
  }
  friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
    return !(lhs==rhs);
  }
  // We implicitly construct from a std::function with the right signature:
  generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
  {
    if (operation)
      state = operation();
  }
  // default all special member functions:
  generator_iterator( generator_iterator && ) =default;
  generator_iterator( generator_iterator const& ) =default;
  generator_iterator& operator=( generator_iterator && ) =default;
  generator_iterator& operator=( generator_iterator const& ) =default;
  generator_iterator() =default;
};
```

exemple vivant .

Nous stockons l'élément généré tôt afin de pouvoir détecter plus facilement si nous sommes déjà à la fin.

Comme la fonction d'un itérateur de générateur d'extrémité n'est jamais utilisée, nous pouvons créer une gamme d'itérateurs de générateur en ne copiant qu'une seule fois la `std::function` . Un générateur de génération construit par défaut se compare à lui-même et à tous les autres générateurs-finisseurs.

Lire Les itérateurs en ligne: https://riptutorial.com/fr/cplusplus/topic/473/les-iterateurs

Chapitre 61: Les portées

Examples

Étendue de bloc simple

La portée d'une variable dans un bloc `{ ... }` commence après la déclaration et se termine à la fin du bloc. S'il existe un bloc imbriqué, le bloc interne peut masquer l'étendue d'une variable déclarée dans le bloc externe.

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
}   // <- Scope of `x` ends here
```

Si un bloc imbriqué commence dans un bloc externe, une nouvelle variable déclarée portant le même nom que dans la classe externe cache la première.

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x;  // <- Output is 200
    }

    std::cout << x;  // <- Output is 100
}
```

Variables globales

Pour déclarer une seule instance d'une variable accessible dans différents fichiers sources, il est possible de la définir dans le cadre global avec le mot-clé `extern` . Ce mot-clé indique que le compilateur contient une définition de cette variable quelque part dans le code, de sorte qu'il peut être utilisé partout et que toute écriture / lecture sera effectuée dans un seul endroit de la mémoire.

```
// File my_globals.h:

#ifndef __MY_GLOBALS_H__
#define __MY_GLOBALS_H__

extern int circle_radius; // Promise to the compiler that circle_radius
                          // will be defined somewhere

#endif
```

```
// File foo1.cpp:

#include "my_globals.h"

int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:

#include "my_globals.h"
#include <iostream>

int main()
{
    std::cout << "The radius is: " << circle_radius << "\n";'
    return 0;
}
```

Sortie:

```
The radius is: 123
```

Lire Les portées en ligne: https://riptutorial.com/fr/cplusplus/topic/3453/les-portees

Chapitre 62: Les références

Examples

Définir une référence

Les références se comportent de la même manière, mais pas complètement comme les pointeurs const. Une référence est définie en suffixant une esperluette `&` à un nom de type.

```
int i = 10;
int &refi = i;
```

Ici, `refi` est une référence liée à `i` .
Les références abstraite la sémantique des pointeurs, agissant comme un alias à l'objet sous-jacent:

```
refi = 20; // i = 20;
```

Vous pouvez également définir plusieurs références dans une même définition:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

Les références **doivent** être initialisées correctement au moment de la définition et ne peuvent plus être modifiées par la suite. Le morceau de code suivant provoque une erreur de compilation:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

Vous ne pouvez pas non plus lier directement une référence à `nullptr` , contrairement aux pointeurs:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a
temporary of type 'nullptr_t'
```

Les références C ++ sont des alias de variables existantes

Une référence en C ++ n'est qu'un `Alias` ou un autre nom d'une variable. Tout comme la plupart d'entre nous peuvent être référés en utilisant notre nom de passeport et notre surnom.

Les références n'existent pas littéralement et elles n'occupent aucune mémoire. Si nous imprimons l'adresse de la variable de référence, elle affichera la même adresse que celle de la

variable à laquelle elle fait référence.

```
int main() {
    int i = 10;
    int &j = i;

    cout<<&i<<endl;
    cout<<&b<<endl;
    return 0;
}
```

Dans l'exemple ci-dessus, les deux `cout` impriment la même adresse. La situation sera la même si on prend une variable comme référence dans une fonction

```
void func (int &fParam ) {
   cout<<"Address inside function => "<<fParam<<endl;
}

int main() {
    int i = 10;
    cout<<"Address inside Main => "<<&i<<endl;

    func(i);

    return 0;
}
```

Dans cet exemple également, les deux `cout` imprimeront la même adresse.

Comme nous le savons maintenant, les `C++ References` sont que des alias, et pour qu'un alias soit créé, nous avons besoin de quelque chose auquel l'alias peut faire référence.

C'est la raison précise pour laquelle cette déclaration lance une erreur de compilation

```
int &i;
```

Parce que l'alias ne fait référence à rien.

Lire Les références en ligne: https://riptutorial.com/fr/cplusplus/topic/1548/les-references

Chapitre 63: Les syndicats

Remarques

Les syndicats sont des outils très utiles, mais apportent quelques précautions importantes:

- Il s'agit d'un comportement indéfini, conformément à la norme C ++, pour accéder à un élément d'une union qui n'était pas le dernier membre modifié. Bien que de nombreux compilateurs C ++ autorisent cet accès de manière bien définie, il s’agit d’extensions et ne peuvent être garanties sur tous les compilateurs.

  Une `std::variant` (depuis C ++ 17) est comme une union, mais elle vous indique seulement ce qu’elle contient actuellement (une partie de son état visible est le type de la valeur qu’elle contient à un moment donné: elle applique uniquement l’accès à la valeur). à ce type).

- Les implémentations n'alignent pas nécessairement les membres de tailles différentes sur la même adresse.

Examples

Fonctions de base de l'Union

Les unions sont une structure spécialisée dans laquelle tous les membres occupent une mémoire qui se chevauchent.

```
union U {
    int a;
    short b;
    float c;
};
U u;

//Address of a and b will be equal
(void*)&u.a == (void*)&u.b;
(void*)&u.a == (void*)&u.c;

//Assigning to any union member changes the shared memory of all members
u.c = 4.f;
u.a = 5;
u.c != 4.f;
```

Utilisation typique

Les unions sont utiles pour minimiser l'utilisation de la mémoire pour les données exclusives, par exemple lors de l'implémentation de types de données mixtes.

```
struct AnyType {
    enum {
        IS_INT,
```

```
        IS_FLOAT
    } type;

    union Data {
        int as_int;
        float as_float;
    } value;

    AnyType(int i) : type(IS_INT) { value.as_int = i; }
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }

    int get_int() const {
        if(type == IS_INT)
            return value.as_int;
        else
            return (int)value.as_float;
    }

    float get_float() const {
        if(type == IS_FLOAT)
            return value.as_float;
        else
            return (float)value.as_int;
    }
};
```

Comportement non défini

```
union U {
    int a;
    short b;
    float c;
};
U u;

u.a = 10;
if (u.b == 10) {
   // this is undefined behavior since 'a' was the last member to be
   // written to. A lot of compilers will allow this and might issue a
   // warning, but the result will be "as expected"; this is a compiler
   // extension and cannot be guaranteed across compilers (i.e. this is
   // not compliant/portable code).
}
```

Lire Les syndicats en ligne: https://riptutorial.com/fr/cplusplus/topic/2678/les-syndicats

Chapitre 64: Littéraux

Introduction

Traditionnellement, un littéral est une expression désignant une constante dont le type et la valeur sont évidents. Par exemple, `42` est un littéral, tandis que `x` n'est pas puisque l'on doit voir sa déclaration pour connaître son type et lire les lignes de code précédentes pour connaître sa valeur.

Cependant, C ++ 11 a également ajouté des littéraux définis par l'utilisateur , qui ne sont pas des littéraux au sens traditionnel mais peuvent être utilisés comme raccourci pour les appels de fonctions.

Examples

vrai

Un mot-clé désignant l'une des deux valeurs possibles du type `bool` .

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

faux

Un mot-clé désignant l'une des deux valeurs possibles du type `bool` .

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

nullptr

C ++ 11

Un mot clé indiquant une constante de pointeur nul. Il peut être converti en n'importe quel type de pointeur ou de pointeur sur membre, produisant un pointeur nul du type résultant.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Notez que `nullptr` n'est pas lui-même un pointeur. Le type de `nullptr` est un type fondamental

appelé `std::nullptr_t` .

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

ce

Dans une fonction membre d'une classe, le mot `this` clé `this` est un pointeur sur l'instance de la classe sur laquelle la fonction a été appelée. `this` ne peut pas être utilisé dans une fonction membre statique.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

Le type de `this` dépend de la qualification cv de la fonction membre: si `X::f` est `const` , alors le type de `this` dans `f` est `const X*` , donc `this` ne peut pas être utilisé pour modifier des données non statiques Fonction membre `const` . De même, `this` hérite de la qualification `volatile` de la fonction dans laquelle elle apparaît.

C ++ 11

`this` peut également être utilisé dans un *initialiseur* d' *accolade ou d'égale* pour un membre de données non statique.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` une valeur, donc il ne peut pas être assigné à.

Littéral entier

Un littéral entier est une expression primaire de la forme

- littéral décimal

C'est un chiffre décimal non nul (1, 2, 3, 4, 5, 6, 7, 8, 9), suivi de zéro ou plusieurs chiffres décimaux (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- littéral octal

C'est le chiffre zéro (0) suivi de zéro ou plusieurs chiffres octaux (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- hex-littéral

C'est la séquence de caractères 0x ou la séquence de caractères 0X suivie d'un ou plusieurs chiffres hexadécimaux (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C , d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- littéral binaire (depuis C ++ 14)

C'est la séquence de caractères 0b ou la séquence de caractères 0B suivie d'un ou plusieurs chiffres binaires (0, 1)

```
int b = 0b101010; // C++14
```

Le suffixe d'entier, s'il est fourni, peut contenir l'un des deux ou les deux suivants (si les deux sont fournis, ils peuvent apparaître dans n'importe quel ordre:

- unsigned-suffix (le caractère u ou le caractère U)

```
unsigned int u_1 = 42u;
```

- suffixe long (le caractère l ou le caractère L) ou le long suffixe long (la séquence de caractères ll ou la séquence de caractères LL) (depuis C ++ 11)

Les variables suivantes sont également initialisées à la même valeur:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Remarques

Les lettres dans les littéraux entiers sont insensibles à la casse: 0xDeAdBaBeU et 0XdeadBABEu représentent le même nombre (une exception est le suffixe long-long, qui est soit ll ou LL, jamais lL ou Ll)

Il n'y a pas de littéral entier négatif. Des expressions telles que -1 appliquent l'opérateur unaire

moins à la valeur représentée par le littéral, ce qui peut impliquer des conversions de types implicites.

Dans C avant C99 (mais pas dans C ++), les valeurs décimales non mappées qui ne rentrent pas dans long int peuvent avoir le type unsigned long int.

Lorsqu'elles sont utilisées dans une expression de contrôle de #if ou #elif, toutes les constantes entières signées agissent comme si elles avaient le type std :: intmax_t et toutes les constantes entières non signées agissent comme si elles avaient le type std :: uintmax_t.

Lire Littéraux en ligne: https://riptutorial.com/fr/cplusplus/topic/7836/litteraux

Chapitre 65: Littéraux définis par l'utilisateur

Examples

Littéraux définis par l'utilisateur avec de longues valeurs doubles

```
#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}
```

La sortie de ce programme est la suivante:

```
3 km = 3000 m
3 mi = 4828.03 m
```

Littéraux standard définis par l'utilisateur pour la durée

C ++ 14

Les littéraux d'utilisateur de durée suivants sont déclarés dans l' `namespace std::literals::chrono_literals` , où les `literals` et les `chrono_literals` sont des espaces de noms `chrono_literals` . L'accès à ces opérateurs peut être obtenu en `using namespace std::literals` `using namespace std::chrono_literals` , en `using namespace std::literals::chrono_literals` `using namespace std::chrono_literals` et en `using namespace std::literals::chrono_literals` .

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
```

```
    std::chrono::minutes t5 = 88min;
    auto t6 = 2 * 0.5h;

    auto total = t1 + t2 + t3 + t4 + t5 + t6;

    std::cout.precision(13);
    std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
              << " hours" << std::endl; // 2 hours
}
```

Littéraux standard définis par l'utilisateur pour les chaînes

C ++ 14

Les littéraux d'utilisateur de chaîne suivants sont déclarés dans l' `namespace std::literals::string_literals` , où `literals` et `string_literals` sont des espaces de noms `string_literals` . L'accès à ces opérateurs peut être obtenu en `using namespace std::literals` `using namespace std::string_literals` , en `using namespace std::string_literals` et en `using namespace std::literals::string_literals` .

```
#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;

    std::cout << s << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
    std::cout << utf16conv.to_bytes(s16) << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
    std::cout << utf32conv.to_bytes(s32) << std::endl;

    std::wcout << ws << std::endl;
}
```

Remarque:

Chaîne littérale peut contenir `\0`

```
std::string s1 = "foo\0\0bar";  // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle
```

Littéraux standard définis par l'utilisateur pour complexes

C ++ 14

Les littéraux utilisateur complexes suivants sont déclarés dans l' `namespace std::literals::complex_literals` , où `literals` et `complex_literals` sont des espaces de noms `complex_literals` . L'accès à ces opérateurs peut être obtenu en `using namespace std::literals` `using namespace std::complex_literals` , en `using namespace std::literals::complex_literals` `using namespace std::complex_literals` et en `using namespace std::literals::complex_literals` .

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;         // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;       // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl;   // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}
```

Littéral auto-créé défini par l'utilisateur pour binaire

Malgré cela, vous pouvez écrire un nombre binaire en C ++ 14 comme:

```
int number =0b0001'0101; // ==21
```

voici un exemple célèbre avec une implémentation auto-faite pour les nombres binaires:

Remarque: L'ensemble du programme d'extension de modèle est en cours d'exécution au moment de la compilation.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" ) ;
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value  } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };


// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value ; }

// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value ; }

#include <iostream>

int main()
{
```

```
    std::cout  << 10101_B << ", " << 011011000111_b  << '\n' ; // prints 21, 1735
}
```

Lire Littéraux définis par l'utilisateur en ligne: https://riptutorial.com/fr/cplusplus/topic/2745/litteraux-definis-par-l-utilisateur

Chapitre 66: Manipulateurs de flux

Introduction

Les manipulateurs sont des fonctions d'assistance spéciales qui aident à contrôler les flux d'entrée et de sortie en utilisant l' `operator >>` ou l' `operator <<` .

Ils peuvent tous être inclus par `#include <iomanip>` .

Remarques

Les manipulateurs peuvent être utilisés autrement. Par exemple:

1. `os.width(n);` est égal à `os << std::setw(n);`
   `is.width(n);` est égal à `is >> std::setw(n);`

2. `os.precision(n);` est égal à `os << std::setprecision(n);`
   `is.precision(n);` est égal à `is >> std::setprecision(n);`

3. `os.setfill(c);` est égal à `os << std::setfill(c);`

4. `str >> std::setbase(base);` ou `str << std::setbase(base);` est égal à

```
str.setf(base ==  8 ? std::ios_base::oct :
            base == 10 ? std::ios_base::dec :
                base == 16 ? std::ios_base::hex :
                     std::ios_base::fmtflags(0),
         std::ios_base::basefield);
```

5. `os.setf(std::ios_base::flag);` est égal à `os << std::flag;`
   `is.setf(std::ios_base::flag);` est égal à `is >> std::flag;`
   `os.unsetf(std::ios_base::flag);` est égal à `os << std::no ## flag;`
   `is.unsetf(std::ios_base::flag);` est égal à `is >> std::no ## flag;`
   (où **##** - est l' *opérateur de concaténation*)
   pour les prochains `flag` : **`boolalpha`** , **`showbase`** , **`showpoint`** , **`showpos`** , **`skipws`** , **`uppercase`** .

6. **`std::ios_base::basefield`** .
   Pour les `flag` : **`dec`** , **`hex`** et **`oct`** :

- `os.setf(std::ios_base::flag, std::ios_base::basefield);` est égal à `os << std::flag;` `is.setf(std::ios_base::flag, std::ios_base::basefield);` est égal à `is >> std::flag;` **(1)**
- `str.unsetf(std::ios_base::flag, std::ios_base::basefield);` est égal à `str.setf(std::ios_base::fmtflags(0), std::ios_base::basefield);` **(2)**

7. **`std::ios_base::adjustfield`** .
   Pour les `flag` : **`left`** , **`right`** et **`internal`** :

- `os.setf(std::ios_base::flag, std::ios_base::adjustfield);` est égal à `os << std::flag;` `is.setf(std::ios_base::flag, std::ios_base::adjustfield);` est égal à `is >> std::flag;` **(1)**
- `str.unsetf(std::ios_base::flag, std::ios_base::adjustfield);` est égal à `str.setf(std::ios_base::fmtflags(0), std::ios_base::adjustfield);` **(2)**

(1) Si le drapeau du champ correspondant précédemment défini est déjà `unsetf` par `unsetf` .
(2) Si le `flag` est défini.

8. **`std::ios_base::floatfield`** .

- `os.setf(std::ios_base::flag, std::ios_base::floatfield);` est égal à `os << std::flag;` `is.setf(std::ios_base::flag, std::ios_base::floatfield);` est égal à `is >> std::flag;` pour les `flag` : **`fixed`** et **`scientific`** .
- `os.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` est égal à `os << std::defaultfloat;` `is.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` est égal à `is >> std::defaultfloat;`

9. `str.setf(std::ios_base::fmtflags(0), std::ios_base::flag);` est égal à `str.unsetf(std::ios_base::flag)`
   pour les `flag` : **`basefield`** , **`adjustfield`** , **`floatfield`** .

10. `os.setf(mask)` est égal à `os << setiosflags(mask);`
    `is.setf(mask)` est égal à `is >> setiosflags(mask);`
    `os.unsetf(mask)` est égal à `os << resetiosflags(mask);`
    `is.unsetf(mask)` est égal à `is >> resetiosflags(mask);`
    Pour presque tous les `mask` de type `std::ios_base::fmtflags` .

Examples

Manipulateurs de flux

`std::boolalpha` et `std::noboolalpha` - bascule entre la représentation textuelle et numérique des booléens.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` et `std::noshowbase` - contrôle si le préfixe indiquant la base numérique est utilisé.

`std::dec` (decimal), `std::hex` (hexadécimal) et `std::oct` (octal) - sont utilisés pour changer la base des entiers.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29  '\n';
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Les valeurs par défaut sont `std::ios_base::noshowbase` et `std::ios_base::dec` .

Si vous voulez en savoir plus sur `std::istringstream` consultez l'en-tête < sstream >.

`std::uppercase` `std::nouppercase` et `std::nouppercase` - déterminent si les caractères majuscules sont utilisés dans la sortie entière hexadécimale et à virgule flottante. N'a aucun effet sur les flux d'entrée.

```
std::cout << std::hex << std::showbase
              << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
              << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

La valeur par défaut est `std::nouppercase` .

`std::setw(n)` - change la largeur du prochain champ d'entrée / sortie en exactement `n` .

La propriété de largeur `n` est réinitialisée à `0` lorsque certaines fonctions sont appelées (la liste complète est ici).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

La valeur par défaut est `std::setw(0)` .

`std::left` , `std::right` et `std::internal` - modifie la position par défaut des caractères de remplissage en définissant `std::ios_base::adjustfield` sur `std::ios_base::left` , `std::ios_base::right` et `std::ios_base::internal` correspondly. `std::left` et `std::right` s'appliquent à toutes les sorties, `std::internal` - pour les sorties entières, flottantes et monétaires. N'a aucun effet sur les flux d'entrée.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87**********
// hex: 41*************
//   $: $3.67**********
// usd: USD *3.67******
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
```

```
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: -**********9.87
// hex: *************41
//   $: $3.67**********
// usd: USD *******3.67
// usd: USD        3.67

std::cout << std::right << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: **********-9.87
// hex: *************41
//   $: **********$3.67
// usd: ******USD *3.67
// usd:       USD  3.67
```

La valeur par défaut est `std::left` .

`std::fixed` , `std::scientific` , `std::hexfloat` [C ++ 11] et `std::defaultfloat` [C ++ 11] - modifient le formatage pour les entrées / sorties en virgule flottante.

`std::fixed` définit `std::ios_base::floatfield` sur `std::ios_base::fixed` ,
`std::scientific` - to `std::ios_base::scientific` ,
`std::hexfloat` - to `std::ios_base::fixed | std::ios_base::scientific` et
`std::defaultfloat` - to `std::ios_base::fmtflags(0)` .

`fmtflags`

```
#include <sstream>
...

std::cout << '\n'
          << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
          << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
          << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'
          << "The number 0.07 in default:    " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
```

```
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308
```

La valeur par défaut est `std::ios_base::fmtflags(0)` .

Il y a un **bug** sur certains compilateurs qui provoque

```
double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0
```

`std::showpoint` et `std::noshowpoint` - contrôle si le point décimal est toujours inclus dans la représentation en virgule flottante. N'a aucun effet sur les flux d'entrée.

```
std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
          << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7
```

La valeur par défaut est `std::showpoint` .

`std::showpos` et `std::noshowpos` - contrôle l'affichage du signe `+` dans la sortie *non négative* . N'a aucun effet sur les flux d'entrée.

```
std::cout << "With showpos: " << std::showpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
          << "Without showpos: " << std::noshowpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17
```

Par défaut si `std::noshowpos` .

`std::unitbuf` , `std::nounitbuf` - contrôle le flux de sortie après chaque opération. N'a aucun effet sur le flux d'entrée. `std::unitbuf` provoque le vidage.

`std::setbase(base)` - définit la base numérique du flux.

`std::setbase(8)` est égal à la définition de `std::ios_base::basefield` à `std::ios_base::oct` ,
`std::setbase(16)` - to `std::ios_base::hex` ,
`std::setbase(10)` - to `std::ios_base::dec` .

Si `base` est autre que `8` , `10` ou `16` alors `std::ios_base::basefield` est en `std::ios_base::fmtflags(0)` . Cela signifie une sortie décimale et une entrée dépendant du préfixe.

Par défaut `std::ios_base::basefield` est `std::ios_base::dec` alors par défaut `std::setbase(10)` .

`std::setprecision(n)` - modifie la précision en virgule flottante.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
const long double pi = std::acos(-1.L);

std::cout << '\n'
          << "default precision (6):   pi: " << pi << '\n'
          << "                       10pi: " << 10 * pi << '\n'
          << "std::setprecision(4):  10pi: " << std::setprecision(4) << 10 * pi << '\n'
          << "                    10000pi: " << 10000 * pi << '\n'
          << "std::fixed:         10000pi: " << std::fixed << 10000 * pi << std::defaultfloat
<< '\n'
          << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
          << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi <<
'\n'
          << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi <<
'\n'
          << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                        10pi: 31.4159
// std::setprecision(4):  10pi: 31.42
//                     10000pi: 3.142e+04
// std::fixed:         10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi:
3.14159265358979323851280895940618620443274267017841339111328125
// max+1 radix precision:   pi:
3.14159265358979323851280895940618620443274267017841339111328125
// significant digits prec: pi: 3.14159265358979324
```

La valeur par défaut est `std::setprecision(6)` .

`std::setiosflags(mask)` et `std::resetiosflags(mask)` - définissent et `std::resetiosflags(mask)` les indicateurs spécifiés dans le `mask` du type `std::ios_base::fmtflags` .

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
```

```
std::cout << "Parsing \"10 010\" with std::oct gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                              std::ios_base::uppercase |
                              std::ios_base::showbase) << 42 << '\n';
// Output: OX2A
```

`std::skipws` et `std::noskipws` - contrôlent le saut des espaces blancs en tête par les fonctions d'entrée formatées. N'a aucun effet sur les flux de sortie.

```
#include <sstream>
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default  behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';
// Output: Default  behavior:  c1 = a  c2 = b  c3 = c
// noskipws behavior:  c1 = a  c2 =    c3 = b
```

La valeur par défaut est `std::ios_base::skipws` .

`std::quoted(s[, delim[, escape]])` [C ++ 14] - insère ou extrait des chaînes entre guillemets avec des espaces incorporés.

`s` - la chaîne à insérer ou à extraire.
`delim` - le caractère à utiliser comme délimiteur, `"` par défaut.
`escape` - le caractère à utiliser comme caractère d'échappement, `\` par défaut.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in     [" << in << "]\n"
          << "stored as   [" << ss.str() << "]\n";

ss >> std::quoted(out);
```

```
std::cout << "written out [" << out << "]\n";
// Output:
// read in     [String with spaces, and embedded "quotes" too]
// stored as   ["String with spaces, and embedded \"quotes\" too"]
// written out [String with spaces, and embedded "quotes" too]
```

Pour plus d'informations, voir le lien ci-dessus.

Manipulateurs de flux de sortie

`std::ends` - insère un caractère nul `'\0'` dans le flux de sortie. Plus formellement, la déclaration de ce manipulateur ressemble à

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

et ce manipulateur place un caractère en appelant `os.put(charT())` lorsqu'il est utilisé dans une expression
`os << std::ends;`

`std::endl` et `std::flush` deux flux de sortie affleurant `out` en appelant `out.flush()` . Cela provoque une production immédiate. Mais `std::endl` insère `'\n'` symbole de fin de ligne `'\n'` avant le vidage.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
// Output: First line.
// Second line. Still second line.
```

`std::setfill(c)` - change le caractère de remplissage en `c` . Souvent utilisé avec `std::setw` .

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:         79
// setfill('#'): ########79
```

`std::put_money(mon[, intl])` [C ++ 11]. Dans une expression `out << std::put_money(mon, intl)` , convertit la valeur monétaire `mon` (de type `long double` ou `std::basic_string`) en sa représentation en caractères comme spécifié par la facette `std::money_put` de la localisation actuellement imprégnée `out` . Utilisez des chaînes de devises internationales si `intl` est `true` , utilisez les symboles de devise autrement.

```
long double money = 123.45;
// or std::string money = "123.45";
```

```
std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD  1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ▯123 or JPY  123
```

`std::put_time(tmb, fmt)` [C ++ 11] - formate et affiche une valeur date / heure dans `std::tm` selon le format spécifié `fmt` .

`tmb` - pointeur sur la structure de temps du calendrier `const std::tm*` obtenue à partir de `localtime()` ou de `gmtime()` .
`fmt` - pointeur sur une chaîne terminée par un caractère nul `const CharT*` spécifiant le format de conversion.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

Pour plus d'informations, voir le lien ci-dessus.

Manipulateurs de flux d'entrée

`std::ws` - consomme les principaux espaces blancs dans le flux d'entrée. C'est différent de `std::skipws` .

```
#include <sstream>
...

std::string str;
std::istringstream("  \v\n\r\t    Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C ++ 11]. Dans une expression `in >> std::get_money(mon, intl)` parse

l'entrée de caractères en tant que valeur monétaire, comme spécifié par le `std::money_get` facette de l'environnement local actuellement imprégné de `in` , et stocke la valeur dans `mon` (de `long double` ou `std::basic_string`). Le manipulateur attend des chaînes de devises internationales *obligatoires* si `intl` est `true` , attend *des* symboles de devise *facultatifs* , sinon.

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C ++ 11] - analyse une valeur date / heure stockée dans `tmb` format spécifié `fmt` .

`tmb` - pointeur valide vers l'objet `const std::tm*` où le résultat sera stocké.
`fmt` - pointeur vers une chaîne terminée par un caractère nul `const CharT*` spécifiant le format de conversion.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

Pour plus d'informations, voir le lien ci-dessus.

Lire Manipulateurs de flux en ligne: https://riptutorial.com/fr/cplusplus/topic/10699/manipulateurs-de-flux

Chapitre 67: Métaprogrammation

Introduction

En C ++, la métaprogrammation fait référence à l'utilisation de macros ou de modèles pour générer du code au moment de la compilation.

En général, les macros sont mal vues dans ce rôle et les modèles sont préférés, bien qu'ils ne soient pas aussi génériques.

La métaprogrammation des modèles utilise souvent des calculs à la compilation, que ce soit via des modèles ou des fonctions `constexpr` , pour atteindre ses objectifs de génération de code. Cependant, les calculs à la compilation ne sont pas des métaprogrammes en soi.

Remarques

La métaprogrammation (ou plus spécifiquement la métaprogrammation des modèles) consiste à utiliser des modèles pour créer des constantes, des fonctions ou des structures de données au moment de la compilation. Cela permet d'effectuer des calculs une fois au moment de la compilation plutôt qu'à chaque exécution.

Examples

Calcul des factoriels

Les factoriels peuvent être calculés au moment de la compilation en utilisant des techniques de métaprogrammation de modèles.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` est une structure, mais dans la métaprogrammation des modèles, elle est traitée comme une métafonction de modèle. Par convention, les métafonctions de gabarits sont évaluées en vérifiant un membre particulier, soit `::type` pour les métafonctions qui génèrent des types, ou `::value` pour les métafonctions qui génèrent des valeurs.

Dans le code ci-dessus, nous évaluons la métafonction `factorial` en instanciant le modèle avec les paramètres que nous voulons passer, et en utilisant `::value` pour obtenir le résultat de l'évaluation.

La métafonction elle-même repose sur l'instanciation récursive de la même métafonction avec des valeurs plus petites. La spécialisation `factorial<0>` représente la condition de terminaison. La métaprogrammation de gabarit a la plupart des restrictions d'un langage de programmation fonctionnel , donc la récursivité est la construction principale de "bouclage".

Comme les métafonctions de modèles s'exécutent au moment de la compilation, leurs résultats peuvent être utilisés dans des contextes nécessitant des valeurs de compilation. Par exemple:

```
int my_array[factorial<5>::value];
```

Les tableaux automatiques doivent avoir une taille définie à la compilation. Et le résultat d'une métafonction est une constante à la compilation, il peut donc être utilisé ici.

Limitation : La plupart des compilateurs ne permettent pas une profondeur de récursivité supérieure à une limite. Par exemple, le compilateur `g++` limite par défaut la récursion à 256 niveaux. Dans le cas de `g++` , le programmeur peut définir la profondeur de récursion en utilisant l' `-ftemplate-depth-X` .

C ++ 11

Depuis C ++ 11, le modèle `std::integral_constant` peut être utilisé pour ce type de calcul de modèle:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
  std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
  std::integral_constant<long long, 1> {};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

De plus, les fonctions `constexpr` deviennent une alternative plus propre.

```
#include <iostream>
```

```
constexpr long long factorial(long long n)
{
  return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
  char test[factorial(3)];
  std::cout << factorial(7) << '\n';
}
```

Le corps de `factorial()` est écrit en tant `constexpr` unique car en C ++ 11, les fonctions `constexpr` ne peuvent utiliser qu'un sous-ensemble assez limité du langage.

C ++ 14

Depuis C ++ 14, de nombreuses restrictions pour les fonctions `constexpr` ont été supprimées et elles peuvent maintenant être écrites beaucoup plus facilement:

```
constexpr long long factorial(long long n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Ou même:

```
constexpr long long factorial(int n)
{
  long long result = 1;
  for (int i = 1; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

C ++ 17

Depuis c ++ 17, on peut utiliser l'expression des plis pour calculer la factorielle:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
   std::cout << factorial<int, 5>::value << std::endl;
```

```
}
```

Itération sur un paquet de paramètres

Souvent, nous devons effectuer une opération sur chaque élément d'un pack de paramètres de modèle variadic. Il existe plusieurs façons de procéder, et les solutions sont plus faciles à lire et à écrire avec C ++ 17. Supposons que nous voulions simplement imprimer chaque élément d'un pack. La solution la plus simple consiste à faire appel à:

C ++ 11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

Nous pourrions utiliser l'astuce d'expansion pour effectuer tout le streaming en une seule fonction. Cela présente l'avantage de ne pas nécessiter une seconde surcharge, mais présente l'inconvénient d'une lisibilité inférieure aux étoiles:

C ++ 11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Pour une explication de son fonctionnement, voir l'excellente réponse de TC .

C ++ 17

Avec C ++ 17, nous avons deux nouveaux outils puissants dans notre arsenal pour résoudre ce problème. Le premier est une expression de pli:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

Et la seconde est `if constexpr` , ce qui nous permet d’écrire notre solution récursive originale en une seule fonction:

```
template <class T, class... Ts>
```

```
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Itération avec std :: integer_sequence

Depuis C ++ 14, le standard fournit le modèle de classe

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

et une métafonction génératrice pour cela:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

Bien que cela soit standard dans C ++ 14, cela peut être implémenté à l'aide des outils C ++ 11.

Nous pouvons utiliser cet outil pour appeler une fonction avec un `std::tuple` d'arguments (normalisé en C ++ 17 comme `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
        std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}


// this will print 3
int f(int, char, double);

auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Expédition de tag

Une manière simple de sélectionner des fonctions au moment de la compilation consiste à envoyer une fonction à une paire de fonctions surchargées qui prennent une balise comme un argument (généralement le dernier). Par exemple, pour implémenter `std::advance()` , nous pouvons envoyer la catégorie itérateur:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }
}

template <class Iter, class Distance>
void advance(Iter& it, Distance n) {
    details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
}
```

Les arguments `std::XY_iterator_tag` des fonctions `details::advance` sont des paramètres de fonction non utilisés. L'implémentation réelle n'a pas d'importance (en fait, elle est complètement vide). Leur seul but est de permettre au compilateur de sélectionner une surcharge en fonction des `details::advance` classe de balise `details::advance` est appelée avec.

Dans cet exemple, `advance` utilise la métafonction `iterator_traits<T>::iterator_category` qui renvoie une des classes `iterator_tag` , en fonction du type réel d' `Iter` . Un objet construit par défaut de la catégorie `iterator_category<Iter>::type` permet alors au compilateur de sélectionner l'une des différentes surcharges de `details::advance` . (Ce paramètre de fonction est susceptible d'être complètement optimisé, car il s'agit d'un objet construit par défaut d'une `struct` vide et jamais utilisé.)

La distribution de balises peut vous donner un code beaucoup plus facile à lire que les équivalents utilisant SFINAE et `enable_if` .

Remarque: alors que C ++ 17 `if constexpr` peut simplifier l'implémentation de l' `advance` en particulier, il ne convient pas pour les implémentations ouvertes contrairement à la distribution de

balises.

Détecter si l'expression est valide

Il est possible de détecter si un opérateur ou une fonction peut être appelé sur un type. Pour tester si une classe a une surcharge de `std::hash` , on peut le faire:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};
```

C ++ 17

Depuis C ++ 17, `std::void_t` peut être utilisé pour simplifier ce type de construction

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) > >
    : std::true_type
{};
```

où `std::void_t` est défini comme suit:

```
template< class... > using void_t = void;
```

Pour détecter si un opérateur, tel que `operator<` est défini, la syntaxe est presque la même:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

Ceux-ci peuvent être utilisés pour utiliser un `std::unordered_map<T>` si `T` a une surcharge pour `std::hash` , mais tentez sinon d'utiliser un `std::map<T>` :

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K,V>>;
```

Calcul de la puissance avec C ++ 11 (et supérieur)

Avec C ++ 11 et des calculs plus élevés au moment de la compilation peuvent être beaucoup plus faciles. Par exemple, le calcul de la puissance d'un nombre donné au moment de la compilation sera le suivant:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Le mot-clé `constexpr` est responsable du calcul de la fonction au moment de la compilation, et alors seulement, lorsque toutes les conditions requises seront satisfaites (voir plus loin le mot-clé constexpr), tous les arguments doivent être connus au moment de la compilation.

Remarque: En C ++ 11, la fonction `constexpr` ne doit composer qu'à partir d'une seule déclaration de retour.

Avantages: En comparant cela à la méthode standard de calcul de la compilation, cette méthode est également utile pour les calculs d’exécution. Cela signifie que si les arguments de la fonction ne sont pas connus au moment de la compilation (par exemple la valeur et la puissance sont données en entrée via l'utilisateur), alors la fonction est exécutée dans un temps de compilation, il n'est donc pas nécessaire de dupliquer un code serait forcé dans les anciennes normes de C ++).

Par exemple

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                               // as both arguments are known at compilation time
                               // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3);  // runtime calculated,
                                     // because value is known only at runtime.
}
```

C ++ 17

Une autre façon de calculer la puissance au moment de la compilation peut utiliser l'expression de repli comme suit:

```
#include <iostream>
```

```
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};

int main() {
   std::cout << power<int, 4, 2>::value << std::endl;
}
```

Distinction manuelle des types avec n'importe quel type T

Lors de l'implémentation de SFINAE à l' aide de `std::enable_if` , il est souvent utile d'avoir accès à des modèles d'aide qui déterminent si un type donné `T` correspond à un ensemble de critères.

Pour nous aider, le standard fournit déjà deux types analogiques à `true` et `false` qui sont `std::true_type` et `std::false_type` .

L'exemple suivant montre comment détecter si un type `T` est un pointeur ou non, le modèle `is_pointer` imite le comportement de l'assistant `std::is_pointer` standard:

```
template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }
```

Le code ci-dessus comporte trois étapes (il suffit parfois de deux):

1. La première déclaration de `is_pointer_` est le *cas par défaut* et hérite de `std::false_type` . Le cas *par défaut* devrait toujours hériter de `std::false_type` car il est analogue à une " `false` condition".

2. La seconde déclaration spécialise le modèle `is_pointer_` pour le pointeur `T*` sans se soucier de ce qu'est réellement `T` Cette version hérite de `std::true_type` .

3. La troisième déclaration (la vraie) supprime simplement toutes les informations inutiles de `T` (dans ce cas, nous `const volatile` qualificateurs `const` et `volatile`), puis nous nous basons sur l'une des deux déclarations précédentes.

Puisque `is_pointer<T>` est une classe, pour accéder à sa valeur, vous devez soit:

- Use `::value` , par exemple `is_pointer<int>::value` - `value` est un membre de classe statique de type `bool` hérité de `std::true_type` ou `std::false_type` ;
- Construire un objet de ce type, par exemple `is_pointer<int>{}` - Cela fonctionne car

`std::is_pointer` hérite de son constructeur par défaut de `std::true_type` ou `std::false_type` (qui ont des constructeurs `constexpr`) et `std::true_type` et `std::false_type` a des opérateurs de conversion `constexpr` à `bool` .

C'est une bonne habitude de fournir des "modèles d'assistance" qui vous permettent d'accéder directement à la valeur:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

C ++ 17

En C ++ 17 et versions `_v` , la plupart des modèles d'aide fournissent déjà une version `_v` , par exemple:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

If-then-else

C ++ 11

Le type `std::conditional` dans l'en-tête de bibliothèque standard `<type_traits>` peut sélectionner un type ou l'autre, en fonction d'une valeur booléenne à la compilation:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

Cette structure contient un pointeur sur `T` si `T` est plus grand que la taille d'un pointeur ou `T` si elle est plus petite ou égale à la taille d'un pointeur. Par conséquent, `sizeof(ValueOrPointer)` sera toujours <= `sizeof(void*)` .

Min / Max générique avec nombre d'arguments variable

C ++ 11

Il est possible d'écrire une fonction générique (par exemple `min`) qui accepte différents types numériques et un nombre d'arguments arbitraire par méta-programmation de modèles. Cette fonction déclare un `min` pour deux arguments et récursivement pour plus.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
```

```
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Lire Métaprogrammation en ligne: https://riptutorial.com/fr/cplusplus/topic/462/metaprogrammation

Chapitre 68: Métaprogrammation arithmitique

Introduction

Ce sont des exemples d'utilisation de métaprogrammation de modèles C ++ dans le traitement d'opérations arithmétiques au moment de la compilation.

Examples

Calcul de la puissance en O (log n)

Cet exemple montre un moyen efficace de calculer la puissance en utilisant la métaprogrammation de modèles.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
   static const int value = 1;
   static_assert(base != 0, "power<0, 0> is not allowed");
};


template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Exemple d'utilisation:

```
std::cout << power<2, 9>::value;
```

C ++ 14

Celui-ci gère également les exposants négatifs:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs
% 2>::intermediateValue;
```

```
    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) :
intermediateValue;

};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};


template <int base>
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};


int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

Lire Métaprogrammation arithmitique en ligne:
https://riptutorial.com/fr/cplusplus/topic/10907/metaprogrammation-arithmitique

Chapitre 69: Modèle de mémoire C ++ 11

Remarques

Différents threads essayant d'accéder au même emplacement de mémoire participent à une *course de données* si au moins une des opérations est une modification (également appelée *opération de stockage*). Ces *courses de données* provoquent *un comportement indéfini* . Pour les éviter, il est nécessaire d'empêcher ces threads d'exécuter simultanément de telles opérations conflictuelles.

Les primitives de synchronisation (mutex, section critique, etc.) peuvent protéger ces accès. Le modèle de mémoire introduit en C ++ 11 définit deux nouvelles méthodes portables pour synchroniser l'accès à la mémoire dans un environnement multithread: les opérations atomiques et les clôtures.

Opérations atomiques

Il est maintenant possible de lire et d'écrire dans un emplacement de mémoire donné en utilisant les opérations de *chargement atomique* et *de stockage atomique* . Pour plus de commodité, elles sont regroupées dans la classe de modèle `std::atomic<t>` . Cette classe enveloppe une valeur de type `t` mais cette fois-ci, les *charges* et les *stockages* dans l'objet sont atomiques.

Le modèle n'est pas disponible pour tous les types. Les types disponibles sont spécifiques à l'implémentation, mais ils incluent généralement la plupart des types intégraux (ou la totalité) ainsi que des types de pointeurs. Donc, `std::atomic<unsigned>` et `std::atomic<std::vector<foo> *>` devraient être disponibles, alors que `std::atomic<std::pair<bool,char>>` ne le sera probablement pas.

Les opérations atomiques ont les propriétés suivantes:

- Toutes les opérations atomiques peuvent être effectuées simultanément à partir de plusieurs threads sans provoquer un comportement indéfini.
- Une *charge atomique* verra soit la valeur initiale avec laquelle l'objet atomique a été construit, soit la valeur écrite via une opération de *stockage atomique* .
- *Les magasins atomiques* du même objet atomique sont ordonnés de la même manière dans tous les threads. Si un thread a déjà vu la valeur d'une opération de *stockage atomique* , les opérations de *chargement atomique* suivantes verront soit la même valeur, soit la valeur stockée par une opération de *stockage atomique* ultérieure.
- *Les opérations de lecture-modification-écriture atomiques* permettent à *la charge atomique* et à *la mémoire atomique* de se produire sans autre *entrepôt atomique* . Par exemple, on peut incrémenter de manière atomique un compteur à partir de plusieurs threads, et aucun incrément ne sera perdu indépendamment du conflit entre les threads.
- Les opérations atomiques reçoivent un paramètre optionnel `std::memory_order` qui définit les propriétés supplémentaires de l'opération par rapport aux autres emplacements de mémoire.

std :: memory_order	Sens
`std::memory_order_relaxed`	pas de restrictions supplémentaires
`std::memory_order_release` → `std::memory_order_acquire`	si `load-acquire` voit la valeur stockée par `store-release` alors les magasins sont *séquencés avant que* la `store-release` se produise avant que les charges ne soient séquencées après l' *acquisition* du *chargement*
`std::memory_order_consume`	comme `memory_order_acquire` mais seulement pour les charges dépendantes
`std::memory_order_acq_rel`	combine `load-acquire` et `store-release`
`std::memory_order_seq_cst`	cohérence séquentielle

Ces balises d'ordre mémoire permettent trois disciplines différentes de la mémoire: la *cohérence séquentielle* , *relaxée* et la *libération-acquisition* avec sa *version-consume* frère.

Consistance séquentielle

Si aucun ordre de mémoire n'est spécifié pour une opération atomique, l'ordre par défaut est la *cohérence séquentielle* . Ce mode peut également être explicitement sélectionné en balisant l'opération avec `std::memory_order_seq_cst` .

Avec cet ordre, aucune opération de mémoire ne peut traverser l'opération atomique. Toutes les opérations de mémoire séquencées avant l'opération atomique se produisent avant l'opération atomique et l'opération atomique se produit avant toutes les opérations de mémoire qui sont séquencées après celle-ci. Ce mode est probablement le plus facile à raisonner, mais il entraîne également la plus grande pénalité en matière de performance. Il empêche également toutes les optimisations du compilateur qui pourraient autrement tenter de réorganiser les opérations après l'opération atomique.

Commande détendue

Le contraire de la *cohérence séquentielle* est le *classement de la* mémoire *détendue* . Il est sélectionné avec la balise `std::memory_order_relaxed` . Une opération atomique détendue n'imposera aucune restriction aux autres opérations de mémoire. Le seul effet qui reste, c'est que l'opération est elle-même encore atomique.

Validation des commandes

Une opération de *stockage atomique* peut être balisée avec `std::memory_order_release` et une opération de *chargement atomique* peut être balisée avec `std::memory_order_acquire` . La première opération est appelée *(atomique) release-release* tandis que la seconde est appelée *(atomic) load -acquise* .

Lorsque *load-ACED* voit la valeur écrite par une *libération de magasin,* il se produit ce qui suit:

toutes les opérations de magasin séquencées avant la *libération de magasin* deviennent visibles pour (*se produire avant*) les opérations de chargement après l' *acquisition de charge* .

Les opérations de lecture-modification-écriture atomiques peuvent également recevoir la balise cumulative `std::memory_order_acq_rel` . Cela rend la partie *de* la *charge atomique* de l'opération une *charge atomique acquisition* tandis que la partie *du magasin atomique* devient *magasin à libération atomique.*

Le compilateur n'est pas autorisé à déplacer les opérations de stockage après une opération de *libération en mémoire atomique* . Il est également interdit de déplacer des opérations de chargement avant *l'acquisition atomique* (ou la *charge-consommation*).

Notez également qu'il n'y a pas de *libération de charge atomique* ou d' *acquisition de mémoire atomique* . Tenter de créer de telles opérations en fait des opérations *détendues* .

Commande de sortie-consommation

Cette combinaison est similaire à *release-acquisition* , mais cette fois la *charge atomique* est balisée avec `std::memory_order_consume` et devient une opération de *chargement-consommation (atomique)* . Ce mode est identique à celui de *release-acquisition* avec la seule différence que parmi les opérations de chargement séquencées après le *load-consume,* seules celles *-ci* sont ordonnées en fonction de la valeur chargée par *load-consume* .

Clôtures

Les clôtures permettent également de classer les opérations de mémoire entre les threads. Une clôture est soit une clôture de libération ou acquérir une clôture.

Si une clôture de relâchement se produit avant une clôture d'acquisition, alors les magasins séquencés avant la clôture de libération sont visibles des charges séquencées après la clôture d'acquisition. Pour garantir que la clôture de validation se produit avant que la barrière d'acquisition ne soit utilisée, il est possible d'utiliser d'autres primitives de synchronisation, y compris des opérations atomiques assouplies.

Examples

Besoin d'un modèle de mémoire

```
int x, y;
bool ready = false;

void init()
{
  x = 2;
  y = 3;
  ready = true;
}
void use()
```

```
{
  if (ready)
    std::cout << x + y;
}
```

Un thread appelle la fonction `init()` tandis qu'un autre thread (ou gestionnaire de signal) appelle la fonction `use()` . On pourrait s'attendre à ce que la fonction `use()` imprime `5` ou ne fasse rien. Cela peut ne pas toujours être le cas pour plusieurs raisons:

- Le processeur peut réorganiser les écritures qui se produisent dans `init()` afin que le code qui s'exécute réellement puisse ressembler à ceci:

  ```
  void init()
  {
    ready = true;
    x = 2;
    y = 3;
  }
  ```

- Le CPU peut réorganiser les lectures qui se produisent dans `use()` pour que le code réellement exécuté devienne:

  ```
  void use()
  {
    int local_x = x;
    int local_y = y;
    if (ready)
      std::cout << local_x + local_y;
  }
  ```

- Un compilateur C ++ optimisé peut décider de réorganiser le programme de la même manière.

Un tel réordonnancement ne peut pas modifier le comportement d'un programme s'exécutant dans un seul thread car un thread ne peut pas entrelacer les appels à `init()` et `use()` . D'un autre côté, dans un paramètre multithread, un thread peut voir une partie des écritures effectuées par l'autre thread où il peut arriver `use()` voit `ready==true` et que garbage in `x` ou `y` ou les deux.

Le modèle de mémoire C ++ permet au programmeur de spécifier quelles opérations de réordonnancement sont autorisées et lesquelles ne le sont pas, de sorte qu’un programme multithread puisse également se comporter comme prévu. L'exemple ci-dessus peut être réécrit de manière sécurisée comme ceci:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  ready.store(true, std::memory_order_release);
}
void use()
```

```
{
  if (ready.load(std::memory_order_acquire))
    std::cout << x + y;
}
```

Ici, `init()` effectue une opération de *libération de mémoire atomique* . Cela non seulement stocke la valeur `true` dans `ready` , mais indique également au compilateur qu'il ne peut pas déplacer cette opération avant les opérations d'écriture qui sont *séquencées avant* elle.

La fonction `use()` effectue une opération d' *acquisition de charge atomique* . Il lit la valeur actuelle de `ready` et interdit également au compilateur de placer les opérations de lecture qui sont *séquencées après que* cela se soit *produit avant que* la *charge atomique ne soit acquise* .

Ces opérations atomiques obligent également le compilateur à mettre en place les instructions matérielles nécessaires pour informer le CPU de l'absence de réorganisations indésirables.

Étant donné que la version *de mémoire atomique se* trouve dans le même emplacement mémoire que le *chargement-acquisition atomique* , le modèle de mémoire stipule que si l'opération *load-tâche* voit la valeur écrite par l'opération *store-release* , toutes les écritures `init()` ' s thread avant cette *release-store* sera visible pour les charges que le thread `use()` exécute après son *load-acquisition* . C'est-à-dire que si `use()` `ready==true` , alors il est garanti que `x==2` et `y==3` .

Notez que le compilateur et le processeur sont toujours autorisés à écrire dans `y` avant d'écrire dans `x` , et de même les lectures de ces variables dans `use()` peuvent se produire dans n'importe quel ordre.

Exemple de clôture

L'exemple ci-dessus peut également être implémenté avec des clôtures et des opérations atomiques assouplies:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  atomic_thread_fence(std::memory_order_release);
  ready.store(true, std::memory_order_relaxed);
}
void use()
{
  if (ready.load(std::memory_order_relaxed))
  {
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
  }
}
```

Si l'opération de charge atomique voit la valeur écrite par le magasin atomique alors le magasin se produit avant la charge, et ainsi faire les clôtures: la clôture de libération se produit avant la

clôture acquire faire les écritures à `x` et `y` qui précèdent la clôture de libération pour devenir visible à l'instruction `std::cout` qui suit la barrière d'acquisition.

Une clôture peut être utile si elle permet de réduire le nombre total d'acquisitions, de versions ou d'autres opérations de synchronisation. Par exemple:

```
void block_and_use()
{
  while (!ready.load(std::memory_order_relaxed))
    ;
  atomic_thread_fence(std::memory_order_acquire);
  std::cout << x + y;
}
```

La fonction `block_and_use()` tourne jusqu'à ce que l'indicateur `ready` soit défini à l'aide d'une charge atomique relâchée. Ensuite, une seule barrière d'acquisition est utilisée pour fournir la commande de mémoire nécessaire.

Lire Modèle de mémoire C ++ 11 en ligne: https://riptutorial.com/fr/cplusplus/topic/7975/modele-de-memoire-c-plusplus-11

Chapitre 70: Modèle de modèle curieusement récurrent (CRTP)

Introduction

Un modèle dans lequel une classe hérite d'un modèle de classe avec lui-même comme l'un de ses paramètres de modèle. CRTP est généralement utilisé pour fournir *un polymorphisme statique* en C ++.

Examples

Le modèle de modèle curieusement récurrent (CRTP)

CRTP est une alternative puissante et statique aux fonctions virtuelles et à l'héritage traditionnel qui peut être utilisée pour donner des propriétés de type à la compilation. Cela fonctionne en ayant un modèle de classe de base qui prend, comme l'un de ses paramètres de modèle, la classe dérivée. Cela lui permet d'effectuer légalement un `static_cast` de `this` pointeur vers la classe dérivée.

Bien sûr, cela signifie également qu'une classe CRTP doit *toujours* être utilisée comme classe de base d'une autre classe. Et la classe dérivée doit se transmettre à la classe de base.

C ++ 14

Disons que vous avez un ensemble de conteneurs qui supportent tous les fonctions `begin()` et `end()` . Les exigences de la bibliothèque standard pour les conteneurs nécessitent davantage de fonctionnalités. Nous pouvons concevoir une classe de base CRTP qui fournit cette fonctionnalité, basée uniquement sur `begin()` et `end()` :

```
#include <iterator>
template <typename Sub>
class Container {
  private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

  public:
    decltype(auto) front() {
      return *self().begin();
    }

    decltype(auto) back() {
      return *std::prev(self().end());
    }

    decltype(auto) size() const {
      return std::distance(self().begin(), self().end());
```

```
    }

    decltype(auto) operator[](std::size_t i) {
      return *std::next(self().begin(), i);
    }
};
```

La classe ci-dessus fournit les fonctions `front()` , `back()` , `size()` et `operator[]` pour toute sous-classe fournissant `begin()` et `end()` . Un exemple de sous-classe est un tableau simple alloué dynamiquement:

```
#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
  public:
    using Base = Container<DynArray<T>>;

    DynArray(std::size_t size)
      : size_{size},
      data_{std::make_unique<T[]>(size_)}
    { }

    T* begin() { return data_.get(); }
    const T* begin() const { return data_.get(); }
    T* end() { return data_.get() + size_; }
    const T* end() const { return data_.get() + size_; }

  private:
    std::size_t size_;
    std::unique_ptr<T[]> data_;
};
```

Les utilisateurs de la classe `DynArray` peuvent utiliser les interfaces fournies par la classe de base CRTP facilement comme suit:

```
DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);
```

Utilité: ce modèle évite en particulier les appels de fonction virtuels à l'exécution qui se produisent dans la hiérarchie d'héritage et repose simplement sur des conversions statiques:

```
DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls
```

La seule distribution statique à l'intérieur de la fonction `begin()` dans la classe de base `Container<DynArray<int>>` permet au compilateur d'optimiser considérablement le code et aucune consultation de table virtuelle ne se produit à l'exécution.

Limitations: Comme la classe de base est basée sur des modèles et différente pour deux `DynArray` différents, il est impossible de stocker des pointeurs dans leurs classes de base dans un

tableau homogène, comme on peut généralement le faire avec l'héritage normal. type:

```
class A {};
class B: public A{};

A* a = new B;
```

CRTP pour éviter la duplication de code

L'exemple dans Visitor Pattern fournit un cas d'utilisation convaincant pour CRTP:

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

struct Square : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};
```

Chaque type d'enfant d' `IShape` doit implémenter la même fonction de la même manière. C'est beaucoup de frappe supplémentaire. Au lieu de cela, nous pouvons introduire un nouveau type dans la hiérarchie qui le fait pour nous:

```
template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // visit with our exact type
        visitor.visit(*static_cast<Derived const*>(this));
    }
};
```

Et maintenant, chaque forme doit simplement hériter de l'accepteur:

```
struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double radius;
};
```

```
struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength)
{}
    Point topLeft;
    double sideLength;
};
```

Aucun code en double nécessaire.

Lire Modèle de modèle curieusement récurrent (CRTP) en ligne:
https://riptutorial.com/fr/cplusplus/topic/9269/modele-de-modele-curieusement-recurrent--crtp-

Chapitre 71: Modèles

Introduction

Les classes, les fonctions et les variables (depuis C ++ 14) peuvent être modélisées. Un template est un morceau de code avec des paramètres libres qui deviendront une classe concrète, une fonction ou une variable lorsque tous les paramètres sont spécifiés. Les paramètres peuvent être des types, des valeurs ou eux-mêmes des modèles. Un modèle bien connu est `std::vector` , qui devient un type de conteneur concret lorsque le type d'élément est spécifié, *par exemple* std::vector<int> .

Syntaxe

- modèle < *template-parameter-list* > *déclaration*
- exporter le modèle < *template-parameter-list* > *declaration* / * jusqu'à C ++ 11 * /
- modèle <> *déclaration*
- *déclaration de* modèle
- *déclaration de* modèle extern / * depuis C ++ 11 * /
- template < *template-liste-paramètre* > classe ... (*opt*) *identifiant* (*opt*)
- template < *template-liste-paramètre* > *identifiant de* classe (*opt*) = *id-expression*
- template < *template-liste-paramètre* > typename ... (*opt*) *identifiant* (*opt*) / * depuis C ++ 17 * /
- template < *template-liste-paramètre* > *identifiant* typename (*opt*) = *id-expression* / * depuis C ++ 17 * /
- *expression postfixe* . *expression-* modèle
- *Postfix expression* -> *id-expression* de modèle
- *imbriqué-name-spécificateur* `template` *template-simple-id* `::`

Remarques

Le mot `template` est un mot clé avec cinq significations différentes dans le langage C ++, selon le contexte.

1. Lorsqu'elle est suivie d'une liste de paramètres de modèle inclus dans `<>` , elle déclare un modèle tel qu'un modèle de classe , un modèle de fonction ou une spécialisation partielle d'un modèle existant.

```
template <class T>
void increment(T& x) { ++x; }
```

2. Lorsqu'il est suivi d'un *vide* `<>` , il déclare une spécialisation explicite (complète) .

```
template <class T>
void print(T x);
```

```
template <> // <-- keyword used in this sense here
void print(const char* s) {
    // output the content of the string
    printf("%s\n", s);
}
```

3. Lorsqu'elle est suivie d'une déclaration sans `<>` , elle forme une déclaration ou une définition d' instanciation explicite .

```
template <class T>
std::set<T> make_singleton(T x) { return std::set<T>(x); }

template std::set<int> make_singleton(int x); // <-- keyword used in this sense here
```

4. Dans une liste de paramètres de modèle, il introduit un paramètre de modèle de modèle .

```
template <class T, template <class U> class Alloc>
//                 ^^^^^^^^ keyword used in this sense here
class List {
    struct Node {
        T value;
        Node* next;
    };
    Alloc<Node> allocator;
    Node* allocate_node() {
        return allocator.allocate(sizeof(T));
    }
    // ...
};
```

5. Après l'opérateur de résolution de portée `::` et les opérateurs d'accès de membre de classe `.` et `->` , il spécifie que le nom suivant est un modèle.

```
struct Allocator {
    template <class T>
    T* allocate();
};

template <class T, class Alloc>
class List {
    struct Node {
        T value;
        Node* next;
    }
    Alloc allocator;
    Node* allocate_node() {
        // return allocator.allocate<Node>();       // error: < and > are interpreted as
                                                    // comparison operators
        return allocator.template allocate<Node>(); // ok; allocate is a template
        //               ^^^^^^^^ keyword used in this sense here
    }
};
```

Avant C ++ 11, un modèle pouvait être déclaré avec le mot - clé `export` , ce qui en faisait un

modèle *exporté* . La définition d'un modèle exporté n'a pas besoin d'être présente dans chaque unité de traduction dans laquelle le modèle est instancié. Par exemple, ce qui suit devait fonctionner:

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
export template <class T> T identity(T x);
#endif
```

`foo.cpp` :

```
#include "foo.h"
template <class T> T identity(T x) { return x; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    const int x = identity(42); // x is 42
}
```

En raison de la difficulté de l'implémentation, le mot clé d' `export` n'était pas pris en charge par la plupart des compilateurs principaux. Il a été supprimé en C ++ 11; maintenant, il est illégal d'utiliser le mot clé d' `export` . Au lieu de cela, il est généralement nécessaire de définir des modèles dans les en-têtes (contrairement aux fonctions non-modèles, qui ne sont généralement *pas* définies dans les en-têtes). Voir Pourquoi les modèles ne peuvent-ils être implémentés que dans le fichier d'en-tête?

Examples

Modèles de fonction

Le template peut également être appliqué aux fonctions (ainsi qu'aux structures plus traditionnelles) avec le même effet.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

Cela peut ensuite être utilisé de la même manière que les modèles de structure.

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

Dans ces deux cas, l'argument template est utilisé pour remplacer les types de paramètres; le résultat fonctionne exactement comme une fonction C ++ normale (si les paramètres ne correspondent pas au type de modèle, le compilateur applique les conversions standard).

Une propriété supplémentaire des fonctions de modèle (contrairement aux classes de modèle) est que le compilateur peut déduire les paramètres du modèle en fonction des paramètres transmis à la fonction.

```
printSum(4, 5);     // Both parameters are int.
                    // This allows the compiler deduce that the type
                    // T is also int.

printSum(5.0, 4);   // In this case the parameters are two different types.
                    // The compiler is unable to deduce the type of T
                    // because there are contradictions. As a result
                    // this is a compile time error.
```

Cette fonctionnalité nous permet de simplifier le code lorsque nous combinons des structures et des fonctions de modèle. Il existe un modèle commun dans la bibliothèque standard qui nous permet de créer une `template structure X` utilisant une fonction d'aide `make_X()` .

```
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1      first;
    T2      second;
};
// 2) A make function that has a parameter type for
//    each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

Comment ça aide?

```
auto val1 = MyPair<int, float>{5, 8.7};     // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7);            // Create object using the types of the paramters.
                                            // In this code both val1 and val2 are the same
                                            // type.
```

Note: Ceci n'est pas conçu pour raccourcir le code. Ceci est conçu pour rendre le code plus robuste. Il permet de modifier les types en modifiant le code dans un seul endroit plutôt que dans plusieurs emplacements.

Transmission d'argument

Le modèle peut accepter les références lvalue et rvalue en utilisant la *référence de transfert* :

```
template <typename T>
void f(T &&t);
```

Dans ce cas, le type réel de `t` sera déduit en fonction du contexte:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

Dans le premier cas, le type `T` est déduit comme *référence à* `X` (`X&`) et le type de `t` est la *référence de lvalue à* `X` , tandis que dans le second cas, le type de `T` est déduit comme `X` et le type de `t` comme *référence de valeur à* `X` (`X&&`).

Remarque: il convient de noter que dans le premier cas, `decltype(t)` est identique à `T` , mais pas dans le second.

Pour transmettre parfaitement `t` à une autre fonction, que ce soit une référence lvalue ou rvalue, il faut utiliser `std::forward` :

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

Les références de transfert peuvent être utilisées avec des modèles variadiques:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Remarque: Les références de transfert ne peuvent être utilisées que pour les paramètres de modèle, par exemple, dans le code suivant, `v` est une référence de valeur, pas une référence de transfert:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Modèle de classe de base

L'idée de base d'un modèle de classe est que le paramètre template est remplacé par un type au moment de la compilation. Le résultat est que la même classe peut être réutilisée pour plusieurs types. L'utilisateur spécifie quel type sera utilisé lorsqu'une variable de la classe est déclarée. Trois exemples en sont illustrés dans `main()` :

```
#include <iostream>
```

```
using std::cout;

template <typename T>         // A simple class to hold one number of any type
class Number {
public:
    void setNum(T n);         // Sets the class field to the given number
    T plus1() const;          // returns class field's "follower"
private:
    T num;                    // Class field
};

template <typename T>         // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T>         // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt;        // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n";     // Prints 2

    Number<double> aDouble;   // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n";    // Prints 4.14159

    Number<float> aFloat;     // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n";      // Prints 2.4

    return 0;  // Successful completion
}
```

Spécialisation de template

Vous pouvez définir une implémentation pour des instanciations spécifiques d'une classe / méthode de modèle.

Par exemple si vous avez:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

Vous pouvez alors écrire:

```
template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Ensuite, un utilisateur qui écrit `sqrt(4.0)` recevra l'implémentation générique alors que `sqrt(4)` obtiendra l'implémentation spécialisée.

Spécialisation du modèle partiel

Au contraire d'un modèle complet, la spécialisation du modèle partiel permet d'introduire un modèle avec certains des arguments du modèle existant. La spécialisation partielle des modèles est uniquement disponible pour les classes / structures de modèles:

```
// Common case:
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) {// Do something}
};
```

Comme indiqué ci-dessus, les spécialisations de modèles partiels peuvent introduire des ensembles de données et de membres de fonction complètement différents.

Lorsqu'un modèle partiellement spécialisé est instancié, la spécialisation la plus appropriée est sélectionnée. Par exemple, définissons un modèle et deux spécialisations partielles:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Maintenant, les appels suivants:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

imprimera

```
General case
T = int
T = int, U = double
```

Les modèles de fonction ne peuvent être que entièrement spécialisés:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1,2);    // Prints "Two ints: 1 2"
}

// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Valeur du paramètre de modèle par défaut

Tout comme dans le cas des arguments de fonction, les paramètres du modèle peuvent avoir leurs valeurs par défaut. Tous les paramètres de modèle avec une valeur par défaut doivent être déclarés à la fin de la liste des paramètres du modèle. L'idée de base est que les paramètres du modèle avec la valeur par défaut peuvent être omis lors de l'instanciation du modèle.

Exemple simple d'utilisation du paramètre template default:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;

    /* Print the length of a.arr: 5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* Last parameter is omitted, N = 10 */
    my_array<int> b;

    /* Print the length of a.arr: 10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

Modèle d'alias

C ++ 11

Exemple de base:

```
template<typename T> using pointer = T*;
```

Cette définition fait du `pointer<T>` un alias de `T*` . Par exemple:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

Les modèles d'alias ne peuvent pas être spécialisés. Cependant, cette fonctionnalité peut être obtenue indirectement en les faisant référence à un type imbriqué dans une structure:

```
template<typename T>
 struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
 struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

Paramètres du modèle de modèle

Parfois, nous aimerions passer dans le modèle un type de modèle sans en fixer les valeurs. C'est pour cela que sont créés les paramètres du modèle de modèle. Exemples de paramètres de modèle de modèle très simples:

```
template <class T>
struct Tag1 { };

template <class T>
struct Tag2 { };

template <template <class> class Tag>
struct IntTag {
   typedef Tag<int> type;
};

int main() {
   IntTag<Tag1>::type t;
}
```

C ++ 11

```
#include <vector>
#include <iostream>

template <class T, template <class...> class C, class U>
C<T> cast_all(const C<U> &c) {
   C<T> result(c.begin(), c.end());
   return result;
```

```
}

int main() {
   std::vector<float> vf = {1.2, 2.6, 3.7};
   auto vi = cast_all<int>(vf);
   for(auto &&i: vi) {
      std::cout << i << std::endl;
   }
}
```

Déclaration des arguments de modèle non-type avec auto

Avant C ++ 17, lorsque vous écriviez un paramètre de type non-modèle, vous deviez d'abord spécifier son type. Donc, un modèle commun est devenu écrit quelque chose comme:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

Mais pour les expressions compliquées, utiliser quelque chose comme cela implique d'avoir à écrire `decltype(expr), expr` lors de l'instanciation des modèles. La solution consiste à simplifier cet idiome et à autoriser simplement l’ `auto` :

C ++ 17

```
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};

using five = integral_constant<5>;
```

Vide deleter personnalisé pour unique_ptr

Un bon exemple de motivation peut être d’essayer de combiner l’optimisation de la base vide avec un paramètre personnalisé pour `unique_ptr` . Différents déléteurs de l'API C ont des types de retour différents, mais peu importe - nous voulons simplement que quelque chose fonctionne pour n'importe quelle fonction:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) const {
        DeleteFn(ptr);
    }
};

template <T, auto DeleteFn>
```

```
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Et maintenant, vous pouvez simplement utiliser n'importe quel pointeur de fonction qui peut prendre un argument de type `T` tant que paramètre non-type de modèle, quel que soit le type de retour, et en extraire un en-tête `unique_ptr` :

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Paramètre de type non-type

Outre les types en tant que paramètre de modèle, nous sommes autorisés à déclarer des valeurs d'expressions constantes répondant à l'un des critères suivants:

- type intégral ou énumération,
- pointeur vers un objet ou un pointeur vers une fonction,
- référence lvalue à une référence à une fonction objet ou lvalue,
- pointeur vers membre,
- `std::nullptr_t` .

Comme tous les paramètres de modèle, les paramètres de modèle non typés peuvent être explicitement spécifiés, définis par défaut ou déduits implicitement via la déduction de l'argument de modèle.

Exemple d'utilisation du paramètre template non-type:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size])  // Pass array by reference. Requires.
{                                        // an exact size. We allow all sizes
    return size;                         // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int  anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

Exemple de spécification explicite des paramètres de type type et non-type:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

Les paramètres de modèle non typés sont l'un des moyens d'obtenir une récurrence du modèle et

permettent de réaliser une métaprogrammation .

Structures de données du modèle Variadic

C ++ 14

Il est souvent utile de définir des classes ou des structures ayant un nombre et un type variables de membres de données définis au moment de la compilation. L'exemple canonique est `std::tuple` , mais il est parfois nécessaire de définir vos propres structures personnalisées. Voici un exemple qui définit la structure en utilisant la composition (plutôt que l'héritage comme avec `std::tuple` . Commencez par la définition générale (vide), qui sert également de base à la fin de la recrusion dans la spécialisation ultérieure:

```
template<typename ... T>
struct DataStructure {};
```

Cela nous permet déjà de définir une structure vide, `DataStructure<> data` , même si cela n’est pas encore très utile.

Vient ensuite la spécialisation récursive:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

Cela nous suffit maintenant pour créer des structures de données arbitraires, telles que `DataStructure<int, float, std::string> data(1, 2.1, "hello")` .

Alors que se passe-t-il? Tout d'abord, notez qu'il s'agit d'une spécialisation nécessitant qu'au moins un paramètre de modèle variadic (à savoir `T` ci-dessus) existe, sans se soucier de la composition spécifique du pack `Rest` . Savoir que `T` existe permet de définir son membre de données en `first` . Le reste des données est empaqueté récursivement en tant que `DataStructure<Rest ... > rest` . Le constructeur initie ces deux membres, y compris un appel de constructeur récursif au membre `rest` .

Pour mieux comprendre cela, nous pouvons travailler sur un exemple: supposons que vous ayez une déclaration `DataStructure<int, float> data` . La déclaration commence par correspondre à la spécialisation, produisant une structure avec les membres de données `int first` et `DataStructure<float> rest` . La définition de `rest` correspond à nouveau à cette spécialisation, en créant ses propres membres `float first` et `DataStructure<> rest` . Enfin, ce dernier `rest` correspond à la définition de base, produisant une structure vide.

Vous pouvez visualiser ceci comme suit:

```
DataStructure<int, float>
   -> int first
   -> DataStructure<float> rest
         -> float first
         -> DataStructure<> rest
              -> (empty)
```

Maintenant nous avons la structure de données, mais ce n'est pas encore très utile car nous ne pouvons pas accéder facilement aux éléments de données individuels (par exemple, pour accéder au dernier membre des données `DataStructure<int, float, std::string> data` nous devrions utiliser des `data.rest.rest.first` , qui n'est pas exactement convivial, Nous ajoutons donc un `get` méthode (seulement nécessaire dans la spécialisation que la structure de base cas n'a aucune donnée à `get`):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
    ...
};
```

Comme vous pouvez le voir, cette fonction `get` member est elle-même basée sur un modèle - cette fois-ci sur l'index du membre requis (l'utilisation peut donc être `data.get<1>()` chose comme `data.get<1>()` , similaire à `std::tuple`). Le travail réel est effectué par une fonction statique dans une classe auxiliaire, `GetHelper` . La raison pour laquelle nous ne pouvons pas définir les fonctionnalités requises directement dans `DataStructure` de `get idx DataStructure` de `get` est parce que (comme nous le verrons bientôt voir) , nous aurions besoin de se spécialiser sur `idx` - mais il est impossible de se spécialiser en fonction de membre de modèle sans se spécialiser la classe contenant modèle. Notez également que l'utilisation d'une `auto` style C ++ 14 rend nos vies beaucoup plus simples car sinon nous aurions besoin d'une expression assez compliquée pour le type de retour.

Donc, à la classe d'assistance. Cette fois, nous aurons besoin d'une déclaration préalable vide et de deux spécialisations. D'abord la déclaration:

```
template<size_t idx, typename T>
struct GetHelper;
```

Maintenant, le cas de base (quand `idx==0`). Dans ce cas, nous renvoyons simplement le `first` membre:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
```

```
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

Dans le cas récursif, nous décrémentons `idx` et `GetHelper` le `GetHelper` pour le membre `rest` :

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

Pour travailler sur un exemple, supposons que nous ayons `DataStructure<int, float> data` et que nous avons besoin de `data.get<1>()` . Cela appelle `GetHelper<1, DataStructure<int, float>>::get(data)` (la 2ème spécialisation), qui à son tour appelle `GetHelper<0, DataStructure<float>>::get(data.rest)` , qui retourne finalement (par la 1ère spécialisation comme maintenant `idx` est 0) `data.rest.first` .

Alors c'est tout! Voici le code de fonctionnement complet, avec quelques exemples d'utilisation dans la fonction `main` :

```
#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
```

```
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}
```

Instanciation explicite

Une définition d'instanciation explicite crée et déclare une classe, une fonction ou une variable concrète à partir d'un modèle, sans l'utiliser pour l'instant. Une instanciation explicite peut être référencée à partir d'autres unités de traduction. Cela peut être utilisé pour éviter de définir un modèle dans un fichier d'en-tête, s'il n'est instancié qu'avec un ensemble fini d'arguments. Par exemple:

```
// print_string.h
template <class T>
void print_string(const T* str);

// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Comme `print_string<char>` et `print_string<wchar_t>` sont explicitement instanciés dans `print_string.cpp` , l'éditeur de liens peut les trouver même si le modèle `print_string` n'est pas défini dans l'en-tête. Si ces déclarations d'instanciation explicites n'étaient pas présentes, une erreur de l'éditeur de liens se produirait probablement. Voir Pourquoi les modèles ne peuvent-ils être implémentés que dans le fichier d'en-tête?

C ++ 11

Si une définition d’instanciation explicite est précédée du mot clé `extern` , elle devient une *déclaration d’* instanciation explicite à la place. La présence d'une déclaration d'instanciation explicite pour une spécialisation donnée empêche l'instanciation implicite de la spécialisation

donnée au sein de l'unité de traduction en cours. Au lieu de cela, une référence à cette spécialisation qui provoquerait une instanciation implicite peut faire référence à une définition d’instanciation explicite dans la même UT ou une autre.

`foo.h`

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
#endif
```

`foo.cpp`

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

`main.cpp`

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42);   // instantiates foo<int> here;
               // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
               // uses instantiation of foo<double> in foo.cpp instead
}
```

Lire Modèles en ligne: https://riptutorial.com/fr/cplusplus/topic/460/modeles

Chapitre 72: Modèles d'expression

Examples

Modèles d'expression de base sur des expressions algébriques élémentaires

---

Introduction et motivation

Les modèles d'expression (**appelés ET** dans la suite) sont une technique puissante de méta-programmation de modèles, utilisée pour accélérer les calculs d'expressions parfois très coûteuses. Il est largement utilisé dans différents domaines, par exemple dans la mise en œuvre de bibliothèques d'algèbre linéaire.

Pour cet exemple, considérons le contexte des calculs algébriques linéaires. Plus précisément, les calculs impliquant uniquement **des opérations par éléments** . Ce type de calcul est l'application la plus élémentaire des **ET** et constitue une bonne introduction à la manière dont les **ET** fonctionnent en interne.

Regardons un exemple motivant. Considérons le calcul de l'expression:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

Pour simplifier, je suppose que les classes `Vector` et operation + (vector plus: element-wise plus operation) et operation * (ici: produit interne vectoriel: opération par élément) sont toutes deux correctement implémentées, comment ils devraient être, mathématiquement.

Dans une implémentation conventionnelle sans utiliser d' **ET** (ou d'autres techniques similaires), **au moins cinq** constructions d'instances `Vector` ont lieu afin d'obtenir le `result` final:

1. Trois instances correspondant à `vec_1` , `vec_2` et `vec_3` .
2. Une instance de `Vector` temporaire `_tmp` , représentant le résultat de `_tmp = vec_2*vec_3;` .
3. Enfin, avec une utilisation correcte de l' **optimisation de la valeur** de **retour** , la construction du `result` final dans `result = vec_1 + _tmp;` .

L'implémentation à l'aide d' **ET** peut **éliminer la création d'un** `Vector _tmp` **temporaire** `Vector _tmp` en 2, ne laissant ainsi que **quatre** constructions d'instances `Vector` . Plus intéressant encore, considérons l'expression suivante qui est plus complexe:

```
Vector result = vec_1 + (vec_2*vec3 + vec_1)*(vec_2 + vec_3*vec_1);
```

Il y aura également quatre constructions d'instances `Vector` au total: `vec_1, vec_2, vec_3` et `result` .

En d'autres termes, dans cet exemple, **où seules les opérations élémentaires sont impliquées** , il est garanti **qu'aucun objet temporaire ne sera créé à partir de calculs intermédiaires** .

Comment fonctionnent les ET

Fondamentalement, les **ET pour tous les calculs algébriques** sont constitués de deux blocs de construction:

1. **Expressions algébriques pures** (**PAE**): ce sont des proxies / abstractions d'expressions algébriques. Une algébrique pure ne fait pas de calculs réels, ce ne sont que des abstractions / modélisation du flux de travail de calcul. Un PAE peut être un modèle de **l'entrée ou de la sortie de toutes les expressions algébriques** . Les instances de **PAE** sont souvent considérées comme peu coûteuses à copier.
2. **Évaluations paresseuses** : qui sont des implémentations de calculs réels. Dans l'exemple suivant, nous verrons que pour les expressions n'impliquant que des opérations élémentaires, les évaluations paresseuses peuvent implémenter des calculs réels à l'intérieur de l'opération indexée sur le résultat final, créant ainsi un schéma d'évaluation à la demande: seulement si le résultat final est accédé / demandé.

Donc, spécifiquement, comment implémentons-nous les **ET** dans cet exemple? Passons à travers maintenant.

Considérez toujours l'extrait de code suivant:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

L'expression pour calculer le résultat peut être décomposée en deux sous-expressions:

1. Un vecteur plus une expression (notée **plus_expr**)
2. Une expression de produit interne vectoriel (dénommée **innerprod_expr**).

Ce que font les **ET** est le suivant:

- Au lieu de calculer tout de suite chaque sous-expression, les **ET** modélisent d'abord toute l'expression à l'aide d'une structure graphique. Chaque nœud du graphique représente un **PAE** . La connexion de périphérie des nœuds représente le flux de calcul réel. Donc, pour l'expression ci-dessus, nous obtenons le graphique suivant:

```
       result = plus_expr( vec_1, innerprod_expr(vec_2, vec_3) )
          /   \
         /     \
        /       \
       /   innerprod_expr( vec_2, vec_3 )
      /         /  \
     /         /    \
```

```
    /         /       \
vec_1     vec_2    vec_3
```

- Le calcul final est réalisé en **parcourant la hiérarchie graphique** : puisque nous ne traitons ici que d'opérations **élémentaires** , le calcul de chaque valeur indexée dans le `result` **peut être effectué de manière indépendante** : l'évaluation finale du `result` peut être différée à un **élément. évaluation sage** de chaque élément de `result` . En d'autres termes, puisque le calcul d'un élément de `result` , `elem_res` , peut être exprimé en utilisant les éléments correspondants dans `vec_1` (`elem_1`), `vec_2` (`elem_2`) et `vec_3` (`elem_3`) comme:

```
elem_res = elem_1 + elem_2*elem_3;
```

il n'est donc pas nécessaire de créer un `Vector` temporaire pour stocker le résultat du produit interne intermédiaire: **tout le calcul pour un élément peut être fait entièrement et être codé dans l'opération d'accès indexé** .

Voici les exemples de codes en action.

Fichier vec.hh: wrapper pour std :: vector, utilisé pour afficher le journal lorsqu'une construction est appelée.

```
#ifndef EXPR_VEC
# define EXPR_VEC

# include <vector>
# include <cassert>
# include <utility>
# include <iostream>
# include <algorithm>
# include <functional>

///
/// This is a wrapper for std::vector. It's only purpose is to print out a log when a
/// vector constructions in called.
/// It wraps the indexed access operator [] and the size() method, which are
/// important for later ETs implementation.
///

// std::vector wrapper.
template<typename ScalarType> class Vector
{
public:
  explicit Vector() { std::cout << "ctor called.\n"; };
  explicit Vector(int size): _vec(size) { std::cout << "ctor called.\n"; };
  explicit Vector(const std::vector<ScalarType> &vec): _vec(vec)
  { std::cout << "ctor called.\n"; };

  Vector(const Vector<ScalarType> & vec): _vec{vec()}
  { std::cout << "copy ctor called.\n"; };
  Vector(Vector<ScalarType> && vec): _vec(std::move(vec()))
```

```
  { std::cout << "move ctor called.\n"; };

  Vector<ScalarType> & operator=(const Vector<ScalarType> &) = default;
  Vector<ScalarType> & operator=(Vector<ScalarType> &&) = default;

  decltype(auto) operator[](int indx) { return _vec[indx]; }
  decltype(auto) operator[](int indx) const { return _vec[indx]; }

  decltype(auto) operator()() & { return (_vec); };
  decltype(auto) operator()() const & { return (_vec); };
  Vector<ScalarType> && operator()() && { return std::move(*this); }

  int size() const { return _vec.size(); }

private:
  std::vector<ScalarType> _vec;
};

///
/// These are conventional overloads of operator + (the vector plus operation)
/// and operator * (the vector inner product operation) without using the expression
/// templates. They are later used for bench-marking purpose.
///

// + (vector plus) operator.
template<typename ScalarType>
auto operator+(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops plus -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::plus<>());
  return Vector<ScalarType>(std::move(_vec));
}

// * (vector inner product) operator.
template<typename ScalarType>
auto operator*(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops multiplies -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::multiplies<>());
  return Vector<ScalarType>(std::move(_vec));
}

#endif //!EXPR_VEC
```

Fichier expr.hh: implémentation de modèles d'expression pour les opérations élémentaires (vecteur plus et produit

interne vectoriel)

Divisons le en sections.

1. La section 1 implémente une classe de base pour toutes les expressions. Il utilise le **modèle de modèle curieusement récurrent** (CRTP).
2. La section 2 implémente le premier **PAE** : un **terminal** , qui n'est qu'un wrapper (référence const) d'une structure de données d'entrée contenant une valeur d'entrée réelle pour le calcul.
3. La section 3 implémente le second **PAE** : **binary_operation** , qui est un modèle de classe utilisé par la suite pour vector_plus et vector_innerprod. Il est paramétré par le **type d'opération** , **le PAE côté gauche** et **le PAE côté droit** . Le calcul réel est codé dans l'opérateur d'accès indexé.
4. La section 4 définit les opérations vector_plus et vector_innerprod comme des opérations **élémentaires** . Il surcharge également l'opérateur + et * pour les **PAE** : ces deux opérations renvoient également **PAE** .

```
#ifndef EXPR_EXPR
# define EXPR_EXPR


/// Fwd declaration.
template<typename> class Vector;

namespace expr
{


/// -----------------------------------------
///
/// Section 1.
///
/// The first section is a base class template for all kinds of expression. It
/// employs the Curiously Recurring Template Pattern, which enables its instantiation
/// to any kind of expression structure inheriting from it.
///
/// -----------------------------------------


  /// Base class for all expressions.
  template<typename Expr> class expr_base
  {
  public:
    const Expr& self() const { return static_cast<const Expr&>(*this); }
    Expr& self() { return static_cast<Expr&>(*this); }

  protected:
    explicit expr_base() {};
    int size() const { return self().size_impl(); }
    auto operator[](int indx) const { return self().at_impl(indx); }
    auto operator()() const { return self()(); };
  };
```

```
/// -----------------------------------------
///
/// The following section 2 & 3 are abstractions of pure algebraic expressions (PAE).
/// Any PAE can be converted to a real object instance using operator(): it is in
/// this conversion process, where the real computations are done.

///
/// Section 2. Terminal
///
/// A terminal is an abstraction wrapping a const reference to the Vector data
/// structure. It inherits from expr_base, therefore providing a unified interface
/// wrapping a Vector into a PAE.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator.
///
/// It might no be necessary for user defined data structures to have a terminal
/// wrapper, since user defined structure can inherit expr_base, therefore eliminates
/// the need to provide such terminal wrapper.
///
/// -----------------------------------------


  /// Generic wrapper for underlying data structure.
  template<typename DataType> class terminal: expr_base<terminal<DataType>>
  {
  public:
    using base_type = expr_base<terminal<DataType>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit terminal(const DataType &val): _val(val) {}
    int size_impl() const { return _val.size(); };
    auto at_impl(int indx) const { return _val[indx]; };
    decltype(auto) operator()() const { return (_val); }

  private:
    const DataType &_val;
  };


/// -----------------------------------------
///
/// Section 3. Binary operation expression.
///
/// This is a PAE abstraction of any binary expression. Similarly it inherits from
/// expr_base.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator. Each call to the at_impl() method is
/// a element wise computation.
///
/// -----------------------------------------


  /// Generic wrapper for binary operations (that are element-wise).
  template<typename Ops, typename lExpr, typename rExpr>
  class binary_ops: public expr_base<binary_ops<Ops,lExpr,rExpr>>
  {
  public:
```

```
    using base_type = expr_base<binary_ops<Ops,lExpr,rExpr>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit binary_ops(const Ops &ops, const lExpr &lxpr, const rExpr &rxpr)
      : _ops(ops), _lxpr(lxpr), _rxpr(rxpr) {};
    int size_impl() const { return _lxpr.size(); };

    /// This does the element-wise computation for index indx.
    auto at_impl(int indx) const { return _ops(_lxpr[indx], _rxpr[indx]); };

    /// Conversion from arbitrary expr to concrete data type. It evaluates
    /// element-wise computations for all indices.
    template<typename DataType> operator DataType()
    {
      DataType _vec(size());
      for(int _ind = 0; _ind < _vec.size(); ++_ind)
        _vec[_ind] = (*this)[_ind];
      return _vec;
    }

  private: /// Ops and expr are assumed cheap to copy.
    Ops   _ops;
    lExpr _lxpr;
    rExpr _rxpr;
  };


/// -----------------------------------------
/// Section 4.
///
/// The following two structs defines algebraic operations on PAEs: here only vector
/// plus and vector inner product are implemented.
///
/// First, some element-wise operations are defined : in other words, vec_plus and
/// vec_prod acts on elements in Vectors, but not whole Vectors.
///
/// Then, operator + & * are overloaded on PAEs, such that: + & * operations on PAEs
/// also return PAEs.
///
/// -----------------------------------------


  /// Element-wise plus operation.
  struct vec_plus_t
  {
    constexpr explicit vec_plus_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs+rhs; }
  };

  /// Element-wise inner product operation.
  struct vec_prod_t
  {
    constexpr explicit vec_prod_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs*rhs; }
  };
```

```
  /// Constant plus and inner product operator objects.
  constexpr vec_plus_t vec_plus{};
  constexpr vec_prod_t vec_prod{};

  /// Plus operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator+(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_plus_t,lExpr,rExpr>(vec_plus,lhs,rhs); }

  /// Inner prod operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator*(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_prod_t,lExpr,rExpr>(vec_prod,lhs,rhs); }

} //!expr


#endif //!EXPR_EXPR
```

Fichier main.cc: test du fichier src

```
# include <chrono>
# include <iomanip>
# include <iostream>
# include "vec.hh"
# include "expr.hh"
# include "boost/core/demangle.hpp"


int main()
{
  using dtype = float;
  constexpr int size = 5e7;

  std::vector<dtype> _vec1(size);
  std::vector<dtype> _vec2(size);
  std::vector<dtype> _vec3(size);

  // ... Initialize vectors' contents.

  Vector<dtype> vec1(std::move(_vec1));
  Vector<dtype> vec2(std::move(_vec2));
  Vector<dtype> vec3(std::move(_vec3));

  unsigned long start_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "\nNo-ETs evaluation starts.\n";

  Vector<dtype> result_no_ets = vec1 + (vec2*vec3);

  unsigned long stop_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << std::setprecision(6) << std::fixed
            << "No-ETs. Time eclapses: " << (stop_ms_no_ets-start_ms_no_ets)/1000.0
```

```
          << " s.\n" << std::endl;

  unsigned long start_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "Evaluation using ETs starts.\n";

  expr::terminal<Vector<dtype>> vec4(vec1);
  expr::terminal<Vector<dtype>> vec5(vec2);
  expr::terminal<Vector<dtype>> vec6(vec3);

  Vector<dtype> result_ets = (vec4 + vec5*vec6);

  unsigned long stop_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << std::setprecision(6) << std::fixed
          << "With ETs. Time eclapses: " << (stop_ms_ets-start_ms_ets)/1000.0
          << " s.\n" << std::endl;

  auto ets_ret_type = (vec4 + vec5*vec6);
  std::cout << "\nETs result's type:\n";
  std::cout << boost::core::demangle( typeid(decltype(ets_ret_type)).name() ) << '\n';

  return 0;
}
```

Voici une sortie possible lors de la compilation avec `-O3 -std=c++14` utilisant GCC 5.3:

```
ctor called.
ctor called.
ctor called.

No-ETs evaluation starts.
ctor called.
ctor called.
No-ETs. Time eclapses: 0.571000 s.

Evaluation using ETs starts.
ctor called.
With ETs. Time eclapses: 0.164000 s.


ETs result's type:
expr::binary_ops<expr::vec_plus_t, expr::terminal<Vector<float> >,
expr::binary_ops<expr::vec_prod_t, expr::terminal<Vector<float> >,
expr::terminal<Vector<float> > > >
```

Les observations sont les suivantes:

- Dans **ce cas, l'** utilisation des **ET** permet une amélioration significative des performances (> 3x).
- La création d'un objet vectoriel temporaire est éliminée. Comme dans le cas **de ETs,** cteur est appelée une seule fois.
- Boost :: demangle a été utilisé pour visualiser le type de retour d'ET avant la conversion: il a clairement construit exactement le même graphique d'expression que celui présenté ci-dessus.

Reculs et mises en garde

- Un inconvénient évident des **ET** est la courbe d'apprentissage, la complexité de la mise en œuvre et la difficulté de maintenance du code. Dans l'exemple ci-dessus où seules les opérations par éléments sont prises en compte, l'implémentation contient déjà énormément de points d'exclusion, sans parler du monde réel, où des expressions algébriques plus complexes apparaissent.), la difficulté sera exponentielle.

- Un autre inconvénient de l'utilisation des **ET** est qu'ils jouent bien avec le mot `auto` clé `auto` . Comme mentionné ci-dessus, les **PAE** sont essentiellement des proxies: et les proxys ne jouent pas bien avec l' `auto` . Prenons l'exemple suivant:

```
auto result = ...;                // Some expensive expression:
                                  // auto returns the expr graph,
                                  // NOT the computed value.
for(auto i = 0; i < 100; ++i)
    ScalrType value = result* ... // Some other expensive computations using result.
```

Dans **chaque itération de la boucle for, le résultat sera réévalué** , car le graphique de l'expression au lieu de la valeur calculée est transmis à la boucle for.

Bibliothèques existantes implémentant des **ET**

- **boost :: proto** est une puissante bibliothèque qui vous permet de définir vos propres règles et grammaires pour vos propres expressions et de les exécuter à l'aide d' **ET** .
- **Eigen** est une bibliothèque d'algèbre linéaire qui implémente différents calculs algébriques avec des **ET** .

Un exemple de base illustrant des modèles d'expression

Un modèle d'expression est une technique d'optimisation à la compilation utilisée principalement dans le calcul scientifique. Son objectif principal est d'éviter les tâches temporaires inutiles et d'optimiser les calculs de boucle en un seul passage (généralement lors d'opérations sur des agrégats numériques). Les modèles d'expression ont été initialement conçus pour contourner l'inefficacité de la surcharge des opérateurs naïfs lors de l' implémentation numérique `Array` ou `Matrix` types. Une terminologie équivalente pour les modèles d’expression a été introduite par Bjarne Stroustrup, qui les appelle "opérations fusionnées" dans la dernière version de son livre, "Le langage de programmation C ++".

Avant de plonger dans des modèles d’expression, vous devez comprendre pourquoi vous en avez besoin au départ. Pour illustrer ceci, considérons la classe de matrice très simple donnée ci-dessous:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;
```

```
    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW>
operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

Étant donné la définition de classe précédente, vous pouvez maintenant écrire des expressions Matrix telles que:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
        c(x, y) = 3.0;
    }
}

Matrix<double, cols, rows> d = a + b + c;  // d(x, y) = 6
```

Comme illustré ci-dessus, le fait de pouvoir surcharger `operator+()` vous fournit une notation qui imite la notation mathématique naturelle des matrices.

Malheureusement, l'implémentation précédente est également très inefficace par rapport à une version «artisanale» équivalente.

Pour comprendre pourquoi, vous devez considérer ce qui se passe lorsque vous écrivez une expression telle que `Matrix d = a + b + c` . Cela se développe en fait en `((a + b) + c)` ou en `operator+(operator+(a, b), c)` . En d'autres termes, la boucle à l'intérieur de l' `operator+()` est exécutée deux fois, alors qu'elle aurait pu être facilement exécutée en un seul passage. Cela se

traduit également par la création de 2 versions temporaires, ce qui dégrade davantage les performances. Essentiellement, en ajoutant la flexibilité d'utiliser une notation proche de son équivalent mathématique, vous avez également rendu la classe `Matrix` très inefficace.

Par exemple, sans surcharge de l'opérateur, vous pouvez implémenter une somme de matrice beaucoup plus efficace en utilisant un seul passage:

```
template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}
```

L'exemple précédent a cependant ses propres inconvénients, car il crée une interface beaucoup plus compliquée pour la classe Matrix (vous devrez envisager des méthodes telles que `Matrix::add2()` , `Matrix::AddMultiply()` , etc.).

Au lieu de cela, prenons un peu de recul et voyons comment nous pouvons adapter la surcharge de l'opérateur pour une performance plus efficace.

Le problème provient du fait que l'expression `Matrix d = a + b + c` est évaluée trop "avidement" avant que vous ayez eu l'occasion de construire l'arborescence complète de l'expression. En d'autres termes, ce que vous voulez vraiment réaliser, c'est d'évaluer `a + b + c` en une seule fois et seulement une fois que vous avez réellement besoin d'affecter l'expression résultante à `d` .

C'est l'idée de base des modèles d'expression: au lieu de l' `operator+()` évalue immédiatement le résultat de l'ajout de deux instances Matrix, il renverra un *"modèle d'expression"* pour une évaluation ultérieure une fois que toute l'arborescence aura été construite.

Par exemple, voici une implémentation possible pour un modèle d'expression correspondant à la somme de deux types:

```
template <typename LHS, typename RHS>
class MatrixSum
{
public:
    using value_type = typename LHS::value_type;

    MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

    value_type operator() (int x, int y) const  {
        return lhs(x, y) + rhs(x, y);
    }
private:
    const LHS& lhs;
    const RHS& rhs;
```

```
};
```

Et voici la version mise à jour de l' `operator+()`

```
template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}
```

Comme vous pouvez le voir, `operator+()` ne renvoie plus une "évaluation enthousiaste" du résultat de l'ajout de 2 instances Matrix (ce qui serait une autre instance Matrix), mais plutôt un modèle d'expression représentant l'opération d'addition. Le point le plus important à garder à l'esprit est que l'expression n'a pas encore été évaluée. Il ne contient que des références à ses opérandes.

En fait, rien ne vous empêche d'instancier le modèle d'expression `MatrixSum<>` comme suit:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

Vous pouvez cependant à un stade ultérieur, lorsque vous avez réellement besoin du résultat de la sommation, évaluer l'expression `d = a + b` comme suit:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}
```

Comme vous pouvez le voir, un autre avantage de l'utilisation d'un modèle d'expression est que vous avez essentiellement réussi à évaluer la somme de `a` et `b` et à l'assigner à `d` en un seul passage.

De plus, rien ne vous empêche de combiner plusieurs modèles d'expression. Par exemple, `a + b + c` entraînerait le modèle d'expression suivant:

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

Et là encore, vous pouvez évaluer le résultat final en un seul passage:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}
```

Enfin, la dernière pièce du puzzle consiste à brancher votre modèle d'expression dans la classe `Matrix` . Ceci est essentiellement réalisé en fournissant une implémentation pour `Matrix::operator=()` , qui prend le modèle d'expression comme argument et l'évalue en une seule fois, comme vous l'avez fait "manuellement" avant:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

    template <typename E>
    Matrix<T, COL, ROW>& operator=(const E& expression) {
        for (std::size_t y = 0; y != rows(); ++y) {
            for (std::size_t x = 0; x != cols(); ++x) {
                values[y * COL + x] = expression(x, y);
            }
        }
        return *this;
    }

private:
    std::vector<T> values;
};
```

Lire Modèles d'expression en ligne: https://riptutorial.com/fr/cplusplus/topic/3404/modeles-d-expression

Chapitre 73: Mot clé ami

Introduction

Des classes bien conçues encapsulent leurs fonctionnalités, cachant leur implémentation tout en fournissant une interface propre et documentée. Cela permet de redessiner ou de modifier tant que l'interface est inchangée.

Dans un scénario plus complexe, plusieurs classes reposant sur les détails d'implémentation des autres peuvent être nécessaires. Les classes et les fonctions d'ami permettent à ces pairs d'accéder aux informations des autres, sans compromettre l'encapsulation et le masquage des informations de l'interface documentée.

Examples

Fonction ami

Une classe ou une structure peut déclarer n'importe quelle fonction comme ami. Si une fonction est un ami d'une classe, elle peut accéder à tous ses membres protégés et privés:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Les modificateurs d'accès ne modifient pas la sémantique des amis. Les déclarations publiques, protégées et privées d'un ami sont équivalentes.

Les déclarations d'amis ne sont pas héritées. Par exemple, si nous sous- `PrivateHolder` :

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

et essayez d'accéder à ses membres, nous obtiendrons ce qui suit:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
    std::cout << pd.derived_private_value << std::endl;
}
```

Notez que la fonction membre `PrivateHolderDerived` ne peut pas accéder à `PrivateHolder::private_value` , alors que la fonction friend peut le faire.

Méthode d'ami

Les méthodes peuvent être déclarées comme amis et fonctions:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Classe d'ami

Une classe entière peut être déclarée comme amie. La déclaration de classe d'amis signifie que tout membre de l'ami peut accéder aux membres privés et protégés de la classe déclarante:

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};
```

```
class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

La déclaration de classe d'amis n'est pas réfléchie. Si les classes ont besoin d'un accès privé dans les deux sens, les deux ont besoin de déclarations d'amis.

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};
```

Lire Mot clé ami en ligne: https://riptutorial.com/fr/cplusplus/topic/3275/mot-cle-ami

Chapitre 74: mot clé const

Syntaxe

- const Type myVariable = initial; // déclare une variable const; ne peut pas être changé
- const Type & myReference = myVariable; // déclare une référence à une variable const
- const Type * myPointer = & myVariable; // Déclare un pointeur sur const. Le pointeur peut changer, mais le membre de données sous-jacent ne peut pas être modifié via le pointeur
- Tapez * const myPointer = & myVariable; // Déclare un pointeur const. Le pointeur ne peut pas être réaffecté pour pointer sur autre chose, mais le membre de données sous-jacent peut être modifié
- const Type * const myPointer = & myVariable; // déclare un const pointeur sur const.

Remarques

Une variable marquée comme `const` ne peut pas être changé [1]. Si vous tentez d'appeler des opérations autres que des const sur cela entraînera une erreur de compilation.

1: Eh bien, il peut être changé par `const_cast` , mais vous ne devriez presque jamais l'utiliser

Examples

Variables locales const

Déclaration et usage

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Reliure de références et de pointeurs

```
int &b = a;       // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;      // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;           // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1            // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;           // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Pointeurs Const

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;

//Error: Cannot assign to const pointer
pC = &b;
```

Fonctions de membre Const

Les fonctions membres d'une classe peuvent être déclarées `const` , ce qui indique au compilateur et aux futurs lecteurs que cette fonction ne modifiera pas l'objet:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

Dans une fonction membre `const` , le pointeur `this` est en fait un `const MyClass *` au lieu d'un `MyClass *` . Cela signifie que vous ne pouvez modifier aucune variable membre dans la fonction; le compilateur émettra un avertissement. Donc, `setMyInt` n'a pas pu être déclaré `const` .

Vous devriez presque toujours marquer les fonctions des membres comme `const` lorsque cela est possible. Seules les fonctions de membre `const` peuvent être appelées sur un `const MyClass` .

`static` méthodes `static` ne peuvent pas être déclarées comme `const` . En effet, une méthode statique appartient à une classe et n’est pas appelée sur l’objet; par conséquent, il ne peut jamais modifier les variables internes de l'objet. Donc, déclarer `static` méthodes `static` comme `const` serait redondant.

Éviter la duplication du code dans les méthodes get const et non const.

En C ++, les méthodes qui ne diffèrent que par un qualificateur `const` peuvent être surchargées.

Parfois, deux versions de getter peuvent être nécessaires pour renvoyer une référence à un membre.

Soit `Foo` une classe, qui a deux méthodes qui effectuent des opérations identiques et renvoie une référence à un objet de type `Bar` :

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

La seule différence entre eux est qu'une méthode est non-const et renvoie une référence non-const (qui peut être utilisée pour modifier un objet) et la seconde est const et renvoie une référence const.

Pour éviter la duplication de code, il est tentant d'appeler une méthode d'une autre. Cependant, nous ne pouvons pas appeler la méthode non-const du const. Mais on peut appeler la méthode const de non-const. Cela nécessitera d'utiliser 'const_cast' pour supprimer le qualificatif const.

La solution est la suivante:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
        return foo;
    }
};
```

Dans le code ci-dessus, nous appelons la version const de `GetBar` partir du non-const `GetBar` en le `GetBar` en const: `const_cast<const Foo*>(this)` . Puisque nous appelons la méthode const de non-const, l'objet lui-même est non-const, et le rejet de const est autorisé.

Examinez l'exemple plus complet suivant:

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Lire mot clé const en ligne: https://riptutorial.com/fr/cplusplus/topic/2386/mot-cle-const

Chapitre 75: mot-clé mutable

Examples

modificateur de membre de classe non statique

`mutable` modificateur `mutable` dans ce contexte est utilisé pour indiquer qu'un champ de données d'un objet const peut être modifié sans affecter l'état visible de l'objet de manière externe.

Si vous envisagez de mettre en cache un résultat de calcul coûteux, vous devriez probablement utiliser ce mot-clé.

Si vous avez un champ de données de verrouillage (par exemple, `std::unique_lock`) qui est verrouillé et déverrouillé dans une méthode const, ce mot clé est également ce que vous pourriez utiliser.

Vous ne devez pas utiliser ce mot-clé pour casser la constance logique d'un objet.

Exemple avec mise en cache:

```
class pi_calculator {
 public:
     double get_pi() const {
         if (pi_calculated) {
             return pi;
         } else {
             double new_pi = 0;
             for (int i = 0; i < 1000000000; ++i) {
                 // some calculation to refine new_pi
             }
             // note: if pi and pi_calculated were not mutable, we would get an error from a
compiler
             // because in a const method we can not change a non-mutable field
             pi = new_pi;
             pi_calculated = true;
             return pi;
         }
     }
 private:
     mutable bool pi_calculated = false;
     mutable double pi = 0;
};
```

lambda mutable

Par défaut, l' `operator()` implicite `operator()` d'un lambda est `const` . Cela interdit d'effectuer des opérations non `const` sur le lambda. Pour permettre la modification des membres, un lambda peut être marqué comme `mutable` , ce qui rend l' `operator()` implicite `operator()` non- `const` :

```
int a = 0;
```

```
auto bad_counter = [a] {
    return a++;    // error: operator() is const
                   // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;  // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Lire mot-clé mutable en ligne: https://riptutorial.com/fr/cplusplus/topic/2705/mot-cle-mutable

Chapitre 76: Mots clés

Introduction

Les mots-clés ont une signification fixe définie par le standard C ++ et ne peuvent pas être utilisés comme identificateurs. Il est illégal de redéfinir les mots clés en utilisant le préprocesseur dans une unité de traduction incluant un en-tête de bibliothèque standard. Cependant, les mots-clés perdent leur signification particulière dans les attributs.

Syntaxe

- asm (*string-literal*);
- noexcept (*expression*) // sens 1
- noexcept (*expression constante*) // sens 2
- noexcept // sens 2
- sizeof *expression unaire*
- sizeof (*id-type*)
- sizeof ... (*identifiant*) // depuis C ++ 11
- typename identifiant *-nom-spécificateur identificateur* // signification 1
- *modèle de* spécificateur *de nom imbriqué-* typename (*opt*) *simple-template-id* // sens 1
- *identifiant* typename (*opt*) // sens 2
- typename ... *identifiant* (*opt*) // signifiant 2; depuis C ++ 11
- *identifiant* typename (*opt*) = *id-type* // signifiant 2
- template < *template-liste-paramètres* > typename ... (*opt*) *identifiant* (*opt*) // signification 3
- template < *template-liste-paramètres* > *identifiant* typename (*opt*) = *id-expression* // signifiant 3

Remarques

La liste complète des mots-clés est la suivante:

- `alignas` (depuis C ++ 11)
- `alignof` (depuis C ++ 11)
- `asm`
- `auto` : depuis C ++ 11 , avant C ++ 11
- `bool`
- `break`
- `case`
- `catch`
- `char`
- `char16_t` (depuis C ++ 11)
- `char32_t` (depuis C ++ 11)
- `class`
- `const`
- `constexpr` (depuis C ++ 11)
- `const_cast`

- continue
- decltype (depuis C ++ 11)
- default
- delete pour la gestion de la mémoire , pour les fonctions (depuis C ++ 11)
- do
- double
- dynamic_cast
- else
- enum
- explicit
- export
- extern tant que spécificateur de déclaration , dans la spécification de liaison , pour les modèles
- false
- float
- for
- friend
- goto
- if
- inline pour les fonctions , pour les espaces de noms (depuis C ++ 11), pour les variables (depuis C ++ 17)
- int
- long
- mutable
- namespace
- new
- noexcept (depuis C ++ 11)
- nullptr (depuis C ++ 11)
- operator
- private
- protected
- public
- register
- reinterpret_cast
- return
- short
- signed
- sizeof
- static
- static_assert (depuis C ++ 11)
- static_cast
- struct
- switch
- template
- this
- thread_local (depuis C ++ 11)
- throw
- true
- try
- typedef
- typeid
- typename
- union
- unsigned

- `using` pour redéclarer un nom , alias un espace de noms , alias un type
- `virtual` pour les fonctions , pour les classes de base
- `void`
- `volatile`
- `wchar_t`
- `while`

Les jetons `final` et le `override` ne sont pas des mots clés. Ils peuvent être utilisés comme identifiants et n'ont une signification particulière que dans certains contextes.

Les jetons `and` , `and_eq` , `bitand` , `bitor` , `compl` , `not` , `not_eq` `or` `or_eq` , `xor` et `xor_eq` sont des orthographes alternatives de `&&` , `&=` , `&` , `|` , `~` `!` , `!=` , `||` , respectivement `|=` , `^` et `^=` . La norme ne les traite pas comme des mots-clés, mais ils sont des mots-clés à toutes fins utiles, car il est impossible de les redéfinir ou de les utiliser autrement que par les opérateurs qu'ils représentent.

Les rubriques suivantes contiennent des explications détaillées sur de nombreux mots-clés en C ++, qui servent des objectifs fondamentaux tels que le nom des types de base ou le contrôle du flux d'exécution.

- Mots-clés de type de base
- Contrôle de flux
- Itération
- Mots-clés littéraux
- Mots-clés de type
- Mots-clés de déclaration de variable
- Classes / Structures
- Spécificateurs de classe de stockage

Examples

asm

Le mot clé `asm` prend un seul opérande, qui doit être un littéral de chaîne. Il a une signification définie par l'implémentation, mais est généralement transmise à l'assembleur de l'implémentation, la sortie de l'assembleur étant incorporée dans l'unité de traduction.

L'instruction `asm` est une *définition* et non une *expression* . Elle peut donc apparaître soit dans la portée du bloc, soit dans la portée de l'espace de noms (y compris la portée globale). Cependant, comme l'assembly inline ne peut pas être contraint par les règles du langage C ++, `asm` peut ne pas apparaître dans une fonction `constexpr` .

Exemple:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

explicite

1. Lorsqu'elle est appliquée à un constructeur à argument unique, empêche ce constructeur d'être utilisé pour effectuer des conversions implicites.

   ```
   class MyVector {
     public:
       explicit MyVector(uint64_t size);
   };
   MyVector v1(100);  // ok
   uint64_t len1 = 100;
   MyVector v2{len1}; // ok, len1 is uint64_t
   int len2 = 100;
   MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
   ```

   Étant donné que C ++ 11 a introduit des listes d'initialisation, en C ++ 11 et versions ultérieures, l' `explicit` peut être appliqué à un constructeur avec un nombre quelconque d'arguments, avec la même signification que dans le cas à argument unique.

   ```
   struct S {
       explicit S(int x, int y);
   };
   S f() {
       return {12, 34};  // ill-formed
       return S{12, 34}; // ok
   }
   ```

C ++ 11

2. Appliqué à une fonction de conversion, cette fonction de conversion ne peut pas être utilisée pour effectuer des conversions implicites.

   ```
   class C {
       const int x;
     public:
       C(int x) : x(x) {}
       explicit operator int() { return x; }
   };
   C c(42);
   int x = c;                     // ill-formed
   int y = static_cast<int>(c); // ok; explicit conversion
   ```

pas d'exception

C ++ 11

1. Opérateur unaire qui détermine si l'évaluation de son opérande peut propager une exception. Notez que les corps des fonctions appelées ne sont pas examinés, donc `noexcept`

peut donner des faux négatifs. L'opérande n'est pas évalué.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n';   // prints 1
}
```

Dans cet exemple, même si `bar()` ne peut jamais lancer une exception, `noexcept(bar())` est toujours faux car le fait que `bar()` ne puisse pas propager une exception n'a pas été explicitement spécifié.

2. Lors de la déclaration d'une fonction, spécifie si la fonction peut ou non propager une exception. Seul, il déclare que la fonction ne peut pas propager une exception. Avec un argument entre parenthèses, il déclare que la fonction peut ou ne peut pas propager une exception en fonction de la valeur de vérité de l'argument.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

Dans cet exemple, nous avons déclaré que `f4` , `f5` et `f6` ne peuvent pas propager des exceptions. (Bien qu'une exception puisse être levée lors de l'exécution de `f6` , elle est interceptée et non autorisée à se propager hors de la fonction.) Nous avons déclaré que `f2` peut propager une exception. Lorsque le spécificateur `noexcept` est omis, il est équivalent à `noexcept(false)` . Nous avons donc implicitement déclaré que `f1` et `f3` peuvent propager des exceptions, même si des exceptions ne peuvent pas être générées lors de l'exécution de `f3` .

C ++ 17

Le fait que la fonction soit ou non ne fait `noexcept` partie du type de la fonction: dans l'exemple ci-dessus, `f1` , `f2` et `f3` ont des types différents de `f4` , `f5` et `f6` . Par conséquent, `noexcept` est également significatif dans les pointeurs de fonction, les arguments de modèle, etc.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;          // ok; types match
void (*p4)() = &g2;          // ok; implicit conversion
```

nom_type

1. Lorsque suivi d'un nom qualifié, `typename` spécifie qu'il s'agit du nom d'un type. Cela est souvent requis dans les modèles, en particulier lorsque le spécificateur de nom imbriqué est un type dépendant autre que l'instanciation en cours. Dans cet exemple, `std::decay<T>` dépend du paramètre de modèle `T` , donc pour nommer le type de `type` imbriqué, il faut préfixer l'intégralité du nom qualifié avec `typename` . Pour plus de détails, voir Où et pourquoi dois-je mettre les mots-clés "template" et "typename"?

```
template <class T>
auto decay_copy(T&& r) -> typename std::decay<T>::type;
```

2. Introduit un paramètre de type dans la déclaration d'un modèle . Dans ce contexte, il est interchangeable avec la `class` .

```
template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

C ++ 17

3. `typename` peut également être utilisé lors de la déclaration d'un paramètre de modèle de modèle , précédant le nom du paramètre, tout comme `class` .

```
template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

taille de

Opérateur unaire qui fournit la taille en octets de son opérande, qui peut être une expression ou un type. Si l'opérande est une expression, elle n'est pas évaluée. La taille est une expression constante de type `std::size_t` .

Si l'opérande est un type, il doit être entre parenthèses.

- Il est illégal d'appliquer `sizeof` à un type de fonction.
- Il est illégal d'appliquer `sizeof` à un type incomplet, y compris le `void` .
- Si sizeof est appliqué à un type de référence `T&` ou `T&&` , il est équivalent à `sizeof(T)` .
- Lorsque `sizeof` est appliqué à un type de classe, il renvoie le nombre d'octets dans un objet complet de ce type, y compris les octets de remplissage au milieu ou à la fin. Par conséquent, une expression `sizeof` ne peut jamais avoir une valeur de 0. Voir la présentation des types d'objet pour plus de détails.
- Le `char` , `signed char` et `unsigned char` types ont une taille de 1. A l' inverse, un octet est défini comme étant la quantité de mémoire nécessaire pour stocker une `char` objet. Cela ne signifie pas nécessairement 8 bits, car certains systèmes ont des objets `char` plus longs que 8 bits.

Si *expr* est une expression, `sizeof(` *expr* `)` est équivalent à `sizeof(T)` où `T` est le type de *expr.*

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

C ++ 11

L'opérateur `sizeof...` renvoie le nombre d'éléments d'un pack de paramètres.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Mots-clés différents

annuler C ++

1. Utilisé comme type de retour de fonction, le mot-clé void spécifie que la fonction ne renvoie pas de valeur. Utilisé pour la liste de paramètres d'une fonction, void spécifie que la fonction ne prend aucun paramètre. Lorsqu'il est utilisé dans la déclaration d'un pointeur, void spécifie que le pointeur est "universel".

2. Si le type d'un pointeur est void *, le pointeur peut pointer sur une variable qui n'est pas déclarée avec le mot clé const ou volatile. Un pointeur vide ne peut être déréférencé que s'il est converti en un autre type. Un pointeur vide peut être converti en n'importe quel autre type de pointeur de données.

3. Un pointeur vide peut pointer sur une fonction, mais pas sur un membre de classe en C ++.

```
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
pv = &i;
   // Cast optional in C required in C++
pint = (int *)pv;
```

Volatile C ++

1. Un qualificateur de type que vous pouvez utiliser pour déclarer qu'un objet peut être modifié dans le programme par le matériel.

```
volatile declarator ;
```

C ++ virtuel

1. Le mot-clé virtual déclare une fonction virtuelle ou une classe de base virtuelle.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Paramètres

1. **spécificateurs de type** Spécifie le type de retour de la fonction de membre virtuel.
2. **member-function-declarator** Déclare une fonction membre.
3. **spécificateur d'accès** Définit le niveau d'accès à la classe de base, publique, protégée ou privée. Peut apparaître avant ou après le mot-clé virtuel.
4. **base-class-name** Identifie un type de classe précédemment déclaré

   ce pointeur

1. Le pointeur this est un pointeur accessible uniquement dans les fonctions membres non statiques d'un type class, struct ou union. Il pointe vers l'objet pour lequel la fonction membre est appelée. Les fonctions membres statiques n'ont pas ce pointeur.

   ```
   this->member-identifier
   ```

Le pointeur d'un objet ne fait pas partie de l'objet lui-même; cela ne se reflète pas dans le résultat d'une instruction sizeof sur l'objet. Au lieu de cela, lorsqu'une fonction membre non statique est appelée pour un objet, l'adresse du objet est transmise par le compilateur en tant qu'argument masqué à la fonction. Par exemple, l'appel de fonction suivant:

```
myDate.setMonth( 3 );

can be interpreted this way:


setMonth( &myDate, 3 );

The object's address is available from within the member function as the this pointer. Most
uses of this are implicit. It is legal, though unnecessary, to explicitly use this when
referring to members of the class. For example:


void Date::setMonth( int mn )
{
   month = mn;            // These three statements
   this->month = mn;      // are equivalent
   (*this).month = mn;
}

The expression *this is commonly used to return the current object from a member function:


return *this;

The this pointer is also used to guard against self-reference:
```

```
if (&Object != this) {
// do not execute in cases of self-reference
```

essayer, lancer et attraper des instructions (C ++)

1. Pour implémenter la gestion des exceptions dans C ++, utilisez les expressions try, throw et catch.
2. Tout d'abord, utilisez un bloc try pour inclure une ou plusieurs instructions pouvant générer une exception.
3. Une expression de lancement signale qu'une condition exceptionnelle - souvent une erreur - s'est produite dans un bloc try. Vous pouvez utiliser un objet de n'importe quel type en tant qu'opérande d'une expression de projection. Cet objet est généralement utilisé pour communiquer des informations sur l'erreur. Dans la plupart des cas, nous vous recommandons d'utiliser la classe std :: exception ou l'une des classes dérivées définies dans la bibliothèque standard. Si l'un d'entre eux n'est pas approprié, nous vous recommandons de dériver votre propre classe d'exception à partir de std :: exception.
4. Pour gérer les exceptions pouvant être levées, implémentez un ou plusieurs blocs catch immédiatement après un bloc try. Chaque bloc catch spécifie le type d'exception qu'il peut gérer.

```
    MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
      throw myDataFormatException("Format error");
   // ...
}
```

Le code après la clause try est la section protégée du code. L'expression jeter jette, c'est-à-dire déclenche une exception. Le bloc de code après la clause catch est le gestionnaire d'exceptions. C'est le gestionnaire qui capture l'exception qui est lancée si les types dans les expressions de lancement et de capture sont compatibles.

```
    try {
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions - dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

ami (C ++)

1. Dans certaines circonstances, il est plus pratique d'accorder un accès au niveau des membres aux fonctions qui ne sont pas membres d'une classe ou à tous les membres d'une classe distincte. Seul l'implémenteur de classe peut déclarer qui sont ses amis. Une fonction ou une classe ne peut pas se déclarer amie d'une classe. Dans une définition de classe, utilisez le mot-clé friend et le nom d'une fonction non membre ou d'une autre classe pour lui accorder l'accès aux membres privés et protégés de votre classe. Dans une définition de modèle, un paramètre de type peut être déclaré comme ami.

2. Si vous déclarez une fonction ami qui n'a pas été déclarée précédemment, cette fonction est exportée vers l'étendue non-classe englobante.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
   friend int ForwardDeclared::IsAFriend();// C2039 error expected
};
```

fonctions ami

1. Une fonction amie est une fonction qui n'est pas membre d'une classe mais a accès aux membres privés et protégés de la classe. Les fonctions amis ne sont pas considérées comme membres de la classe. Ce sont des fonctions externes normales qui bénéficient de privilèges d'accès spéciaux.

2. Les amis ne sont pas dans la portée de la classe et ils ne sont pas appelés à l'aide des opérateurs de sélection de membres (. Et ->), sauf s'ils sont membres d'une autre classe.

3. Une fonction amie est déclarée par la classe qui accorde l'accès. La déclaration d'ami peut être placée n'importe où dans la déclaration de classe. Il n'est pas affecté par les mots-clés de contrôle d'accès.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
```

```
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
    // Output: 0
           1
}
```

Membres de la classe en tant qu'amis

```
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Lire Mots clés en ligne: https://riptutorial.com/fr/cplusplus/topic/4891/mots-cles

Chapitre 77: Mots-clés de déclaration de variable

Examples

const

Un spécificateur de type; Appliqué à un type, génère la version qualifiée du type. Voir le mot-clé const pour plus de détails sur la signification de `const` .

```
const int x = 123;
x = 456;     // error
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

decltype

C ++ 11

Donne le type de son opérande, qui n'est pas évalué.

- Si l'opérande `e` est un nom sans parenthèses supplémentaires, `decltype(e)` est le *type déclaré* de `e` .

  ```
  int x = 42;
  std::vector<decltype(x)> v(100, x); // v is a vector<int>
  ```

- Si l'opérande `e` est un accès de membre de classe sans parenthèses supplémentaires, `decltype(e)` est le *type déclaré* du membre accédé.

  ```
  struct S {
      int x = 42;
  };
  const S s;
  decltype(s.x) y; // y has type int, even though s.x is const
  ```

- Dans tous les autres cas, `decltype(e)` fournit à la fois le type et la catégorie de valeur de l'expression `e` , comme suit:
  - Si `e` est une lvalue de type `T` , alors `decltype(e)` est `T&` .
  - Si `e` est une valeur x de type `T` , alors `decltype(e)` est `T&&` .

- Si `e` est une valeur de type `T` , alors `decltype(e)` est `T`

  Cela inclut le cas avec des parenthèses étrangères.

  ```
  int f() { return 42; }
  int& g() { static int x = 42; return x; }
  int x = 42;
  decltype(f()) a = f(); // a has type int
  decltype(g()) b = g(); // b has type int&
  decltype((x)) c = x;   // c has type int&, since x is an lvalue
  ```

C ++ 14

La forme spéciale `decltype(auto)` déduit le type d'une variable de son initialiseur ou le type de retour d'une fonction des instructions de `return` dans sa définition, en utilisant les règles de déduction de type de `decltype` plutôt que celles de `auto` .

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

signé

Un mot-clé qui fait partie de certains noms de type entier.

- Utilisé seul, `int` est implicite, de sorte que `signed` , `signed int` et `int` sont du même type.
- Lorsqu'il est combiné avec l' `char` , donne le type `signed char` , qui est un type différent de `char` , même si `char` est également signé. `signed char` a une portée qui comprend au moins -127 à +127 inclus.
- Lorsqu'il est associé à un format `short` , `long` ou `long long` , il est redondant, car ces types sont déjà signés.
- `signed` ne peut pas être combiné avec `bool` , `wchar_t` , `char16_t` ou `char32_t` .

Exemple:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

non signé

Un spécificateur de type qui demande la version non signée d'un type entier.

- Utilisé seul, `int` est implicite, donc `unsigned` est du même type que `unsigned int` .
- Le type `unsigned char` est différent du type `char` , même si `char` n'est pas signé. Il peut contenir des nombres entiers jusqu’à 255.
- `unsigned` peut également être combiné avec `short` , `long` ou `long long` . Il ne peut pas être combiné avec `bool` , `wchar_t` , `char16_t` ou `char32_t` .

Exemple:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

volatil

Un qualificatif de type; appliqué à un type, produit la version qualifiée volatile du type. La qualification volatile joue le même rôle que la qualification `const` dans le système de type, mais `volatile` n'empêche pas la modification des objets; au contraire, il oblige le compilateur à traiter tous les accès à de tels objets comme des effets secondaires.

Dans l'exemple ci-dessous, si `memory_mapped_port` n'était pas volatile, le compilateur pourrait optimiser la fonction afin qu'elle n'effectue que l'écriture finale, ce qui serait incorrect si `sizeof(int)` est supérieur à 1. La qualification `volatile` oblige à traiter toutes les `sizeof(int)` écrit en tant qu'effets secondaires différents et les exécute donc (dans l'ordre).

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Lire Mots-clés de déclaration de variable en ligne: https://riptutorial.com/fr/cplusplus/topic/7840/mots-cles-de-declaration-de-variable

Chapitre 78: Mots-clés de type

Examples

classe

1. Introduit la définition d'un type de classe .

```
class foo {
    int x;
  public:
    int get_x();
    void set_x(int new_x);
};
```

2. Introduit un *spécificateur de type élaboré,* qui spécifie que le nom suivant est le nom d'un type de classe. Si le nom de la classe a déjà été déclaré, il peut être trouvé même s'il est masqué par un autre nom. Si le nom de la classe n'a pas déjà été déclaré, il est déclaré en avant.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
class baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
class foo {
    bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
  public:
    foo();
};
```

3. Introduit un paramètre de type dans la déclaration d'un modèle .

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. Dans la déclaration d'un paramètre de modèle de modèle , la `class` mot clé précède le nom du paramètre. L'argument pour un paramètre de modèle de modèle ne pouvant être qu'un modèle de classe, l'utilisation de la `class` ici est redondante. Cependant, la grammaire de C ++ l'exige.

```
template <template <class T> class U>
//                           ^^^^^ "class" used in this sense here;
```

```
//                                  U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Notez que le sens 2 et le sens 3 peuvent être combinés dans la même déclaration. Par exemple:

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

C ++ 11

6. Dans la déclaration ou la définition d'une énumération, déclare l'énumération comme une énumération de portée .

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

struct

Interchangeable avec la `class` , sauf pour les différences suivantes:

- Si un type de classe est défini à l'aide du mot `struct` clé `struct` , l'accessibilité par défaut des bases et des membres est `public` plutôt que `private` .
- `struct` ne peut pas être utilisé pour déclarer un paramètre de type de modèle ou un paramètre de modèle de modèle; seule `class` peut.

enum

1. Introduit la définition d'un type d'énumération .

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

C ++ 11

En C ++ 11, `enum` peut éventuellement être suivi par `class` ou `struct` pour définir un enum de portée

. De plus, le type sous-jacent peut être explicitement spécifié par `: T` suivant le nom enum, où `T` désigne un type entier.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Les énumérateurs dans les `enum` normales peuvent également être précédés de l'opérateur scope, bien qu'ils soient toujours considérés comme faisant partie de la portée dans laquelle l' `enum` été définie.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Introduit un *spécificateur de type élaboré,* qui spécifie que le nom suivant est le nom d'un type enum déclaré précédemment. (Un spécificateur de type élaboré ne peut pas être utilisé pour déclarer en avant un type enum.) Une énumération peut être nommée de cette manière même si elle est masquée par un autre nom.

```
enum Foo { FOO };
void Foo() {}
Foo foo = FOO;      // ill-formed; Foo refers to the function
enum Foo foo = FOO; // ok; Foo refers to the enum type
```

C ++ 11

3. Introduit une *déclaration enum opaque,* qui déclare une énumération sans la définir. Il peut soit redéclarer une énumération précédemment déclarée, soit déclarer en avant une énumération qui n'a pas encore été déclarée.

   Une énumération déclarée comme première portée ne peut plus être déclarée ultérieurement comme étant non tronquée, ou *inversement.* Toutes les déclarations d'une énumération doivent correspondre à un type sous-jacent.

   Lors de la déclaration en avant d'un enum non tronqué, le type sous-jacent doit être explicitement spécifié, car il ne peut être inféré tant que les valeurs des énumérateurs ne sont pas connues.

```
enum class Format; // underlying type is implicitly int
void f(Format f);
```

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

syndicat

1. Introduit la définition d'un type d' union .

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

2. Introduit un *spécificateur de type élaboré,* qui spécifie que le nom suivant est le nom d'un type d'union. Si le nom de l'union a déjà été déclaré, il peut être trouvé même s'il est masqué par un autre nom. Si le nom du syndicat n'a pas encore été déclaré, il est déclaré en avant.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Lire Mots-clés de type en ligne: https://riptutorial.com/fr/cplusplus/topic/7838/mots-cles-de-type

Chapitre 79: Mots-clés de type de base

Examples

int

Indique un type entier signé avec "la taille naturelle suggérée par l'architecture de l'environnement d'exécution", dont la plage inclut au moins -32767 à +32767 inclus.

```
int x = 2;
int y = 3;
int z = x + y;
```

Peut être combiné avec `unsigned` , `short` , `long` et `long long` (qv) pour donner d'autres types d'entiers.

bool

Un type entier dont la valeur peut être `true` ou `false` .

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

carboniser

Un type d'entier qui est "assez grand pour stocker n'importe quel membre du jeu de caractères de base de l'implémentation". Elle est mise en oeuvre définie si `char` est signé (et a une plage d'au moins -127 à 127, inclusivement) ou non signé (et a une plage d'au moins 0 à 255, inclusivement).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

char16_t

C ++ 11

Un type d'entier non signé de même taille et de même alignement que `uint_least16_t` , qui est donc assez grand pour contenir une unité de code UTF-16.

```
const char16_t message[] = u"□□□□□\n";            // Chinese for "hello, world\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\n"; // prints 7
```

char32_t

C ++ 11

Un type entier non signé de la même taille et du même alignement que `uint_least32_t` , donc suffisamment grand pour contenir une unité de code UTF-32.

```
const char32_t full_house[] = U"□□□□□";               // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\n"; // prints 6
```

flotte

Un type à virgule flottante. Possède la plage la plus étroite parmi les trois types de virgule flottante en C ++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

double

Un type à virgule flottante. Sa portée comprend celle du `float` . Lorsqu'il est combiné avec `long` , désigne le `long double` virgule flottante, dont l'intervalle inclut celui du `double` .

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

longue

Indique un type entier signé qui est au moins aussi long que `int` et dont l'intervalle inclut au moins -2147483647 à +2147483647 inclus (c'est-à-dire - (2 ^ 31 - 1) à + (2 ^ 31 - 1)). Ce type peut également être écrit en tant que `long int` .

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

La combinaison `long double` dénote un type à virgule flottante, qui a la plus grande portée parmi les trois types à virgule flottante.

```
long double area(long double radius) {
    const long double pi = 3.1415926535897932385L;
    return pi*radius*radius;
}
```

C ++ 11

Lorsque la `long` spécificateur se produit deux fois, comme au `long long` , il désigne un type entier

signé qui est au moins aussi longue que `long` , et dont la portée comprend au moins -9223372036854775807 à 9223372036854775807 inclusivement (qui est, - (2 ^ 63 - 1) à + (2 ^ 63-1)).

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

court

Indique un type d'entier signé qui est au moins aussi long que `char` et dont l'intervalle inclut au moins -32767 à +32767 inclus. Ce type peut également être écrit en tant que `short int` .

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

vide

Un type incomplet; Il n'est pas possible qu'un objet soit de type `void` , ni de tableau `void` ou de références à `void` . Il est utilisé comme type de retour des fonctions qui ne renvoient rien.

De plus, une fonction peut être redondée avec un seul paramètre de type `void` ; Cela équivaut à déclarer une fonction sans paramètre (par exemple, `int main()` et `int main(void)` déclarent la même fonction). Cette syntaxe est autorisée pour la compatibilité avec C (où les déclarations de fonctions ont un sens différent de celui de C ++).

Le type `void*` ("pointer to `void` ") a la propriété que tout pointeur d'objet peut être converti en lui et en retour et aboutir au même pointeur. Cette fonctionnalité rend le type `void*` adapté à certains types d’interfaces d’effacement de type (de type non sécurisé), par exemple pour des contextes génériques dans les API de style C (par exemple, `qsort` , `pthread_create`).

Toute expression peut être convertie en une expression de type `void` ; cela s'appelle une *expression à valeur rejetée* :

```
static_cast<void>(std::printf("Hello, %s!\n", name));  // discard return value
```

Cela peut être utile pour signaler explicitement que la valeur d'une expression n'est pas intéressante et que l'expression doit être évaluée pour ses seuls effets secondaires.

wchar_t

Un type entier suffisamment grand pour représenter tous les caractères du plus grand jeu de caractères étendu pris en charge, également appelé jeu de caractères larges. (Il n'est pas portable de supposer que `wchar_t` utilise un encodage particulier, tel que UTF-16.)

Il est normalement utilisé lorsque vous devez stocker des caractères sur ASCII 255, car sa taille est supérieure à celle du caractère de type `char` .

```
const wchar_t message_ahmaric[] = L"ሰላም ልዑል\n"; //Ahmaric for "hello, world\n"
const wchar_t message_chinese[] = L"□□□□□□\n";// Chinese for "hello, world\n"
const wchar_t message_hebrew[]  = L"שלום עולם\n"; //Hebrew for "hello, world\n"
const wchar_t message_russian[] = L"Привет мир\n";  //Russian for "hello, world\n"
const wchar_t message_tamil[]   = L"ஹலோ உலகம்\n"; //Tamil for "hello, world\n"
```

Lire Mots-clés de type de base en ligne: https://riptutorial.com/fr/cplusplus/topic/7839/mots-cles-de-type-de-base

Chapitre 80: Mutex récursif

Examples

std :: recursive_mutex

Le mutex récursif permet au même thread de verrouiller récursivement une ressource - jusqu'à une limite non spécifiée.

Il y a très peu de justifications réelles pour cela. Certaines implémentations complexes peuvent nécessiter d'appeler une copie surchargée d'une fonction sans libérer le verrou.

```
    std::atomic_int temp{0};
    std::recursive_mutex _mutex;

    //launch_deferred launches asynchronous tasks on the same thread id

    auto future1 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;

                    std::this_thread::sleep_for(std::chrono::seconds(3));
                    std::unique_lock<std::recursive_mutex> lock( _mutex);
                    temp=0;

                });

    auto future2 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;
                    while ( true )
                    {
                        std::this_thread::sleep_for(std::chrono::milliseconds(1));
                        std::unique_lock<std::recursive_mutex> lock( _mutex,
std::try_to_lock);
                        if ( temp < INT_MAX )
                             temp++;

                        cout << temp << endl;

                    }
                });
    future1.get();
    future2.get();
```

Lire Mutex récursif en ligne: https://riptutorial.com/fr/cplusplus/topic/9929/mutex-recursif

Chapitre 81: Mutexes

Remarques

Il est préférable d'utiliser *std :: shared_mutex* que *std :: shared_timed_mutex* .

La différence de performance est plus que doublée.

Si vous souhaitez utiliser RWLock, vous trouverez deux options.
C'est std :: shared_mutex et shared_timed_mutex.
Vous pouvez penser que std :: shared_timed_mutex est juste la version 'std :: shared_mutex + time method'.

Mais la mise en œuvre est totalement différente.

Le code ci-dessous est l'implémentation MSVC14.1 de std :: shared_mutex.

```
class shared_mutex
{
public:
typedef _Smtx_t * native_handle_type;

shared_mutex() _NOEXCEPT
    : _Myhandle(0)
    {    // default construct
    }

~shared_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock() _NOEXCEPT
    {    // lock exclusive
    _Smtx_lock_exclusive(&_Myhandle);
    }

bool try_lock() _NOEXCEPT
    {    // try to lock exclusive
    return (_Smtx_try_lock_exclusive(&_Myhandle) != 0);
    }

void unlock() _NOEXCEPT
    {    // unlock exclusive
    _Smtx_unlock_exclusive(&_Myhandle);
    }

void lock_shared() _NOEXCEPT
```

```
    {    // lock non-exclusive
    _Smtx_lock_shared(&_Myhandle);
    }

bool try_lock_shared() _NOEXCEPT
    {    // try to lock non-exclusive
    return (_Smtx_try_lock_shared(&_Myhandle) != 0);
    }

void unlock_shared() _NOEXCEPT
    {    // unlock non-exclusive
    _Smtx_unlock_shared(&_Myhandle);
    }

native_handle_type native_handle() _NOEXCEPT
    {    // get native handle
    return (&_Myhandle);
    }

shared_mutex(const shared_mutex&) = delete;
shared_mutex& operator=(const shared_mutex&) = delete;
private:
    _Smtx_t _Myhandle;
};

void __cdecl _Smtx_lock_exclusive(_Smtx_t * smtx)
{    /* lock shared mutex exclusively */
AcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_lock_shared(_Smtx_t * smtx)
{    /* lock shared mutex non-exclusively */
AcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}

int __cdecl _Smtx_try_lock_exclusive(_Smtx_t * smtx)
{    /* try to lock shared mutex exclusively */
return (TryAcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx)));
}

int __cdecl _Smtx_try_lock_shared(_Smtx_t * smtx)
{    /* try to lock shared mutex non-exclusively */
return (TryAcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx)));
}

void __cdecl _Smtx_unlock_exclusive(_Smtx_t * smtx)
{    /* unlock exclusive shared mutex */
ReleaseSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_unlock_shared(_Smtx_t * smtx)
{    /* unlock non-exclusive shared mutex */
ReleaseSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}
```

Vous pouvez voir que std :: shared_mutex est implémenté dans Windows Slim Reader / Write Locks (https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa904937(v=vs.85).aspx)

Maintenant, regardons l'implémentation de std :: shared_timed_mutex.

Le code ci-dessous est l'implémentation MSVC14.1 de std :: shared_timed_mutex.

```
class shared_timed_mutex
{
typedef unsigned int _Read_cnt_t;
static constexpr _Read_cnt_t _Max_readers = _Read_cnt_t(-1);
public:
shared_timed_mutex() _NOEXCEPT
    : _Mymtx(), _Read_queue(), _Write_queue(),
        _Readers(0), _Writing(false)
    {    // default construct
    }

~shared_timed_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock()
    {    // lock exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing)
        _Write_queue.wait(_Lock);
    _Writing = true;
    while (0 < _Readers)
        _Read_queue.wait(_Lock);    // wait for writing, no readers
    }

bool try_lock()
    {    // try to lock exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || 0 < _Readers)
        return (false);
    else
        {    // set writing, no readers
        _Writing = true;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock for duration
    return (try_lock_until(chrono::steady_clock::now() + _Rel_time));
    }

template<class _Clock,
    class _Duration>
    bool try_lock_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock until time point
    auto _Not_writing = [this] { return (!_Writing); };
    auto _Zero_readers = [this] { return (_Readers == 0); };
    unique_lock<mutex> _Lock(_Mymtx);

    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Not_writing))
        return (false);
```

```
    _Writing = true;

    if (!_Read_queue.wait_until(_Lock, _Abs_time, _Zero_readers))
        {    // timeout, leave writing state
        _Writing = false;
        _Lock.unlock();    // unlock before notifying, for efficiency
        _Write_queue.notify_all();
        return (false);
        }

    return (true);
    }

void unlock()
    {    // unlock exclusive
        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);

        _Writing = false;
        }

    _Write_queue.notify_all();
    }

void lock_shared()
    {    // lock non-exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing || _Readers == _Max_readers)
        _Write_queue.wait(_Lock);
    ++_Readers;
    }

bool try_lock_shared()
    {    // try to lock non-exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || _Readers == _Max_readers)
        return (false);
    else
        {    // count another reader
        ++_Readers;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_shared_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock non-exclusive for relative time
    return (try_lock_shared_until(_Rel_time
        + chrono::steady_clock::now()));
    }

template<class _Time>
    bool _Try_lock_shared_until(_Time _Abs_time)
    {    // try to lock non-exclusive until absolute time
    auto _Can_acquire = [this] {
        return (!_Writing && _Readers < _Max_readers); };

    unique_lock<mutex> _Lock(_Mymtx);
```

```
    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Can_acquire))
        return (false);

    ++_Readers;
    return (true);
    }

template<class _Clock,
    class _Duration>
    bool try_lock_shared_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

bool try_lock_shared_until(const xtime *_Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

void unlock_shared()
    {    // unlock non-exclusive
    _Read_cnt_t _Local_readers;
    bool _Local_writing;

        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);
        --_Readers;
        _Local_readers = _Readers;
        _Local_writing = _Writing;
        }

    if (_Local_writing && _Local_readers == 0)
        _Read_queue.notify_one();
    else if (!_Local_writing && _Local_readers == _Max_readers - 1)
        _Write_queue.notify_all();
    }

shared_timed_mutex(const shared_timed_mutex&) = delete;
shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
private:
mutex _Mymtx;
condition_variable _Read_queue, _Write_queue;
_Read_cnt_t _Readers;
bool _Writing;
};

class stl_condition_variable_win7 final : public stl_condition_variable_interface
{
public:
    stl_condition_variable_win7()
    {
        __crtInitializeConditionVariable(&m_condition_variable);
    }

    ~stl_condition_variable_win7() = delete;
    stl_condition_variable_win7(const stl_condition_variable_win7&) = delete;
    stl_condition_variable_win7& operator=(const stl_condition_variable_win7&) = delete;

    virtual void destroy() override {}
```

```
    virtual void wait(stl_critical_section_interface *lock) override
    {
        if (!stl_condition_variable_win7::wait_for(lock, INFINITE))
            std::terminate();
    }

    virtual bool wait_for(stl_critical_section_interface *lock, unsigned int timeout) override
    {
        return __crtSleepConditionVariableSRW(&m_condition_variable,
static_cast<stl_critical_section_win7 *>(lock)->native_handle(), timeout, 0) != 0;
    }

    virtual void notify_one() override
    {
        __crtWakeConditionVariable(&m_condition_variable);
    }

    virtual void notify_all() override
    {
        __crtWakeAllConditionVariable(&m_condition_variable);
    }

private:
    CONDITION_VARIABLE m_condition_variable;
};
```

Vous pouvez voir que std :: shared_timed_mutex est implémenté dans std :: condition_value.

C'est une grande différence.

Alors vérifions les performances de deux d'entre eux.

```
STLSharedMutex READ :            486647
STLSharedMutex WRITE :           205986
TOTAL READ&WRITE :               692633

STLSharedTimedMutex READ :       140291
STLSharedTimedMutex WRITE :      178849
TOTAL READ&WRITE :               319140
```

Ceci est le résultat d'un test de lecture / écriture de 1000 millisecondes.

std :: shared_mutex traité en lecture / écriture plus de 2 fois plus que std :: shared_timed_mutex.

Dans cet exemple, le taux de lecture / écriture est le même, mais le taux de lecture est plus fréquent que le taux d’écriture réel.
Par conséquent, la différence de performance peut être plus grande.

le code ci-dessous est le code dans cet exemple.

```
void useSTLSharedMutex()
{
    std::shared_mutex shared_mtx_lock;
```

```
    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;

                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;
```

```
    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedMutex READ :            " << readProcessedCnt << std::endl;
    std::cout << "STLSharedMutex WRITE :           " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :               " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}

void useSTLSharedTimedMutex()
{
    std::shared_timed_mutex shared_mtx_lock;

    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;
```

```
                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;

    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedTimedMutex READ :      " << readProcessedCnt << std::endl;
    std::cout << "STLSharedTimedMutex WRITE :     " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}
```

Examples

std :: unique_lock, std :: shared_lock, std :: lock_guard

Utilisé pour l'acquisition de style RAII de verrous d'essai, de verrous d'essais temporisés et de verrous récursifs.

`std::unique_lock` permet la propriété exclusive de mutex.

`std::shared_lock` permet la propriété partagée des mutex. Plusieurs threads peuvent contenir `std::shared_locks` sur un `std::shared_mutex` . Disponible à partir de C ++ 14.

`std::lock_guard` est une alternative légère à `std::unique_lock` et `std::shared_lock` .

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it =  _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
```

```
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string,std::string>  _phonebook;
};
```

Stratégies pour les classes de verrouillage: std :: try_to_lock, std :: adopt_lock, std :: defer_lock

Lors de la création d'un std :: unique_lock, vous avez le choix entre trois stratégies de verrouillage: `std::try_to_lock` , `std::defer_lock` et `std::adopt_lock`

1. `std::try_to_lock` permet d'essayer un verrou sans bloquer:

```
{
    std::atomic_int temp {0};
    std::mutex _mutex;

    std::thread t( [&](){

        while( temp!= -1){
            std::this_thread::sleep_for(std::chrono::seconds(5));
            std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

            if(lock.owns_lock()){
                //do something
                temp=0;
            }
        }
    });

    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if(lock.owns_lock()){
            if (temp < INT_MAX){
                ++temp;
            }
            std::cout << temp << std::endl;
        }
    }
}
```

2. `std::defer_lock` permet de créer une structure de verrouillage sans acquérir le verrou. Lorsque vous verrouillez plusieurs mutex, il y a une possibilité de blocage si deux appelants de fonctions tentent d'acquérir les verrous en même temps:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
```

```
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    lock1.lock()
    lock2.lock(); // deadlock here
    std::cout << "Locked! << std::endl;
    //...
}
```

Avec le code suivant, quoi qu'il arrive dans la fonction, les verrous sont acquis et libérés dans l'ordre approprié:

```
   {
       std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
       std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
       std::lock(lock1,lock2); // no deadlock possible
       std::cout << "Locked! << std::endl;
       //...

   }
```

3. `std::adopt_lock` ne tente pas de verrouiller une seconde fois si le thread appelant possède actuellement le verrou.

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
    std::cout << "Locked! << std::endl;
    //...
}
```

Il faut garder à l'esprit que std :: adopt_lock ne remplace pas l'utilisation de mutex récursive. Lorsque le verrou est hors de portée, le mutex est **libéré** .

std :: mutex

std :: mutex est une structure de synchronisation simple et non récursive utilisée pour protéger les données auxquelles accèdent plusieurs threads.

```
    std::atomic_int temp{0};
    std::mutex _mutex;

    std::thread t( [&](){

                  while( temp!= -1){
                      std::this_thread::sleep_for(std::chrono::seconds(5));
                      std::unique_lock<std::mutex> lock( _mutex);

                          temp=0;
                  }
              });


    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
```

```
        if ( temp < INT_MAX )
            temp++;
        cout << temp << endl;

    }
```

std :: scoped_lock (C ++ 17)

`std::scoped_lock` fournit une sémantique de style RAII pour posséder un autre mutex, combinée aux algorithmes d'évitement de verrou utilisés par `std::lock` . Lorsque `std::scoped_lock` est détruit, les mutex sont libérés dans l'ordre inverse duquel ils ont été acquis.

```
{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}
```

Types de mutex

C ++ 1x propose une sélection de classes de mutex:

- std :: mutex - offre une fonctionnalité de verrouillage simple.
- std :: timed_mutex - offre la fonctionnalité try_to_lock
- std :: recursive_mutex - permet le verrouillage récursif par le même thread.
- std :: shared_mutex, std :: shared_timed_mutex - propose une fonctionnalité de verrouillage partagée et unique.

std :: lock

`std::lock` utilise des algorithmes d'évitement de blocage pour verrouiller un ou plusieurs mutex. Si une exception est levée pendant un appel pour verrouiller plusieurs objets, `std::lock` déverrouille les objets verrouillés avec succès avant de relancer l'exception.

```
std::lock(_mutex1, _mutex2);
```

Lire Mutexes en ligne: https://riptutorial.com/fr/cplusplus/topic/9895/mutexes

Chapitre 82: Objets appelables

Introduction

Les objets appelables sont la collection de toutes les structures C ++ pouvant être utilisées comme une fonction. En pratique, ce sont toutes les choses que vous pouvez transmettre à la fonction C ++ 17 STL invoke () ou qui peuvent être utilisées dans le constructeur de std :: function, y compris: pointeurs de fonction, classes avec opérateur (), classes avec implicite conversions, références à des fonctions, pointeurs à des fonctions membres, pointeurs à des données de membre, lambdas. Les objets appelables sont utilisés dans de nombreux algorithmes STL en tant que prédicats.

Remarques

Un exposé très utile de Stephan T. Lavavej (<Fonctionnalité>: Quoi de neuf et bon usage) (Diapositives) mène à la base de cette documentation.

Examples

Pointeurs de fonction

Les pointeurs de fonctions sont le moyen le plus élémentaire de passer des fonctions, qui peuvent également être utilisés dans C. (Voir la documentation C pour plus de détails).

Pour les objets appelables, un pointeur de fonction peut être défini comme suit:

```
typedef returnType(*name)(arguments);                       // All
using name = returnType(*)(arguments);                      // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>;     // <= C++14
```

Si nous utilisions un pointeur de fonction pour écrire notre propre tri vectoriel, cela ressemblerait à ceci:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function
```

```
struct GreaterThanInt {
   static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member
function
```

Sinon, nous aurions pu invoquer le pointeur de fonction de l'une des manières suivantes:

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

Classes avec opérateur () (Functors)

Chaque classe qui surcharge l' `operator()` peut être utilisée comme objet de fonction. Ces classes peuvent être écrites à la main (souvent appelées foncteurs) ou générées automatiquement par le compilateur en écrivant Lambdas à partir de C ++ 11.

```
struct Person {
    std::string name;
    unsigned int age;
};

// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
         return person.name == _name;
    }
private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...
```

Comme les foncteurs ont leur propre identité, ils ne peuvent pas être placés dans un typedef et ceux-ci doivent être acceptés via un argument de modèle. La définition de `std::find_if` peut ressembler à `std::find_if` :

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
     for (Iterator i = begin, i != end, ++i)
         if (predicate(*i))
             return i;
     return end;
}
```

A partir de C ++ 17, l'appel du prédicat peut être fait avec invoke: `std::invoke(predicate, *i)` .

Lire Objets appelables en ligne: https://riptutorial.com/fr/cplusplus/topic/6073/objets-appelables

Chapitre 83: Opérateurs de bits

Remarques

Les opérations de décalage de bits ne sont pas portables pour toutes les architectures de processeur, différents processeurs peuvent avoir des largeurs de bits différentes. En d'autres termes, si vous écriviez

```
int a = ~0;
int b = a << 1;
```

Cette valeur serait différente sur une machine 64 bits par rapport à une machine 32 bits, ou entre un processeur x86 et un processeur PIC.

Endian-ness n'a pas besoin d'être pris en compte pour les opérations sur les bits elles-mêmes, c'est-à-dire que le décalage vers la droite (`>>`) décale les bits vers le bit le moins significatif et qu'un XOR exécute un bit exclusif ou sur les bits. Endian-ness n'a besoin d'être pris en compte que par les données elles-mêmes, c'est-à-dire que si l'endian-ness est une préoccupation pour votre application, il s'agit d'une préoccupation indépendamment des opérations sur les bits.

Examples

& - bitwise AND

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Sortie

```
a = 6, b = 10, c = 2
```

Pourquoi

Un peu sage `AND` fonctionne sur le niveau du bit et utilise la table de vérité booléenne suivante:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

Lorsque la valeur binaire pour `a` (`0110`) et la valeur binaire pour `b` (`1010`) sont `AND` ensemble, nous obtenons la valeur binaire de `0010` :

```
int a = 0 1 1 0
int b = 1 0 1 0 &
```

```
        ---------
int c = 0 0 1 0
```

Le bit sages ET ne modifie pas la valeur des valeurs d'origine, à moins que cela ne soit spécifiquement attribué à l'utilisation de l'opérateur de mappage par bits `&=` :

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

| - bit à bit OU

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Sortie

```
a = 5, b = 12, c = 13
```

Pourquoi

Un peu sage `OR` fonctionne sur le niveau du bit et utilise la table de vérité booléenne suivante:

```
true OR true = true
true OR false = true
false OR false = false
```

Lorsque la valeur binaire pour `a` (`0101`) et la valeur binaire pour `b` (`1100`) sont `OR` ensemble, nous obtenons la valeur binaire de `1101` :

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

Le bit sages OU ne modifie pas la valeur des valeurs d'origine à moins que ce soit spécifiquement affecté à l'aide de l'opérateur composé d'affectation par bit `|=` :

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

^ - bit à bit XOR (OU exclusif)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Sortie

```
a = 5, b = 9, c = 12
```

Pourquoi

Un peu `XOR` (exclusif ou) opère au niveau du bit et utilise la table de vérité booléenne suivante:

```
true OR true = false
true OR false = true
false OR false = false
```

Notez qu'avec une opération XOR `true OR true = false` où avec des opérations `true AND/OR true = true` , d'où la nature exclusive de l'opération XOR.

En utilisant ceci, lorsque la valeur binaire pour `a` (`0101`) et la valeur binaire pour `b` (`1001`) sont `XOR` ensemble, nous obtenons la valeur binaire de `1100` :

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

Le bit XOR ne modifie pas la valeur des valeurs d'origine, à moins que cela ne soit spécifiquement attribué à l'utilisation de l'opérateur composé d'affectation par bit `^=` :

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

Le bit XOR peut être utilisé de nombreuses manières et est souvent utilisé dans les opérations de masque binaire pour le chiffrement et la compression.

Remarque: L'exemple suivant est souvent présenté comme un exemple d'astuce. Mais ne devrait pas être utilisé dans le code de production (il existe de meilleures méthodes `std::swap()` pour obtenir le même résultat).

Vous pouvez également utiliser une opération XOR pour échanger deux variables sans temporaire:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

Pour produire cela, vous devez ajouter une vérification pour vous assurer qu'il peut être utilisé.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

Donc, bien que cela ressemble à une astuce en vase clos, ce n'est pas utile en code réel. xor n'est pas une opération logique de base, mais une combinaison d'autres opérations: a ^ c = ~ (a & c) & (a | c)

également en 2015+ les variables de compilateurs peuvent être assignées comme binaires:

```
int cn=0b0111;
```

~ - bitwise NOT (complément unaire)

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)

std::cout << "a = " << static_cast<int>(a) <<
             ", b = " << static_cast<int>(b) << std::endl;
```

Sortie

```
a = 234, b = 21
```

Pourquoi

Un peu sage `NOT` (complément unaire) fonctionne sur le niveau du bit et chaque bit est réfléchi. Si c'est un `1` , il est changé en `0` , si c'est un `0` , il est changé en `1` . Le bit not not NOT a le même effet que XOR sur une valeur par rapport à la valeur max pour un type spécifique:

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)
unsigned char c = a ^ ~0;
```

Le bit not NON peut aussi être un moyen pratique de vérifier la valeur maximale d'un type entier spécifique:

```
unsigned int i = ~0;
unsigned char c = ~0;

std::cout << "max uint = " << i << std::endl <<
             "max uchar = " << static_cast<short>(c) << std::endl;
```

Le bit not not NOT ne change pas la valeur de la valeur d'origine et n'a pas d'opérateur d'affectation composé, vous ne pouvez donc pas faire `a ~= 10` par exemple.

Le *bit* not not NOT (`~`) ne doit pas être confondu avec le NOT *logique* (`!`); où un peu sage ne retournera pas chaque bit, un NOT logique utilisera la valeur entière pour effectuer son opération, en d'autres termes `(!1) != (~1)`

<< - décalage gauche

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Sortie

```
a = 1, b = 2
```

Pourquoi

Le décalage gauche sage décalera les bits de la valeur de gauche (`a`) le nombre spécifié à droite (`1`), remplissant essentiellement les bits les moins significatifs avec les 0, déplaçant ainsi la valeur de `5` (`0000 0101` binaire) à gauche 4 fois (par exemple `5 << 4`) donneront la valeur de `80` (binaire `0101 0000`). Vous remarquerez peut-être que le fait de déplacer une valeur vers la gauche 1 fois est également identique à la multiplication de la valeur 2, par exemple:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

Mais il convient de noter que l’opération de décalage vers la gauche décale *tous les* bits vers la gauche, y compris le bit de signe, par exemple:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Sortie possible: `a = 2147483647, b = -2`

Bien que certains compilateurs donneront des résultats qui semblent attendus, il convient de noter que si vous laissez un décalage sur un numéro signé pour que le bit de signe soit affecté, le résultat est **indéfini** . Il est également **indéfini** si le nombre de bits que vous souhaitez décaler est un nombre négatif ou est supérieur au nombre de bits que le type à gauche peut contenir,

exemple:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

Le décalage gauche au niveau du bit ne modifie pas la valeur des valeurs d'origine, sauf si l'opérateur spécifique d'attribution de bits `<<=` :

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

>> - décalage vers la droite

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Sortie

```
a = 2, b = 1
```

Pourquoi

Le décalage vers la droite permet de décaler les bits de la valeur de gauche (`a`) du nombre spécifié à droite (`1`); il convient de noter que si l'opération d'un décalage à droite est standard, ce qui arrive aux bits d'un décalage à droite sur un nombre *négatif signé* est *défini par l'implémentation* et ne peut donc pas être garanti portable, par exemple:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

Il est également indéfini si le nombre de bits que vous souhaitez décaler est un nombre négatif, par exemple:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

Le décalage vers la droite au niveau du bit ne modifie pas la valeur des valeurs d'origine, sauf si elles sont spécifiquement associées à l'utilisation de l'opérateur composé d'affectation par bit `>>=` :

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Lire Opérateurs de bits en ligne: https://riptutorial.com/fr/cplusplus/topic/2572/operateurs-de-bits

Chapitre 84: Optimisation

Introduction

Lors de la compilation, le compilateur modifiera souvent le programme pour augmenter les performances. Cela est autorisé par la règle as-if , qui autorise toutes les transformations qui ne modifient pas le comportement observable.

Examples

Expansion Inline / Inlining

L'expansion en ligne (également appelée inlining) est une optimisation du compilateur qui remplace un appel à une fonction par le corps de cette fonction. Cela économise l'appel de la fonction, mais au détriment de l'espace, car la fonction peut être dupliquée plusieurs fois.

```
// source:

int process(int value)
{
    return 2 * value;
}

int foo(int a)
{
    return process(a);
}

// program, after inlining:

int foo(int a)
{
    return 2 * a; // the body of process() is copied into foo()
}
```

L'inlining est le plus souvent fait pour les petites fonctions, où la surcharge de l'appel de la fonction est significative par rapport à la taille du corps de la fonction.

Optimisation de la base vide

La taille de tout objet ou sous-objet membre doit être au moins égale à 1, même si le type est un type de `class` vide (c'est-à-dire une `class` ou une `struct` sans membres de données non statiques), afin de pouvoir garantir que les adresses d'objets distincts du même type sont toujours distinctes.

Cependant, `class` sous-objets de `class` base ne sont pas trop limités et peuvent être complètement optimisés à partir de la disposition de l'objet:

```
#include <cassert>
```

```
struct Base {}; // empty class

struct Derived1 : Base {
    int i;
};

int main() {
    // the size of any object of empty class type is at least 1
    assert(sizeof(Base) == 1);

    // empty base optimization applies
    assert(sizeof(Derived1) == sizeof(int));
}
```

L'optimisation de base vide est couramment utilisée par les classes de bibliothèque standard compatibles avec les allocateurs (`std::vector` , `std::function` , `std::shared_ptr` , etc.) pour éviter d'occuper un stockage supplémentaire pour son membre allocateur si l'allocateur est sans état. Ceci est réalisé en stockant l'un des membres de données requis (par exemple, pointeur de `begin` , de `end` ou de `capacity` pour le `vector`).

Référence: cppreference

Lire Optimisation en ligne: https://riptutorial.com/fr/cplusplus/topic/9767/optimisation

Chapitre 85: Optimisation en C ++

Examples

Optimisation de la classe de base vide

Un objet ne peut pas occuper moins de 1 octet, car les membres d'un tableau de ce type auraient la même adresse. Ainsi `sizeof(T)>=1` tient toujours. Il est également vrai qu'une classe dérivée ne peut être inférieure à *aucune de* ses classes de base. Cependant, lorsque la classe de base est vide, sa taille n'est pas nécessairement ajoutée à la classe dérivée:

```
class Base {};

class Derived : public Base
{
public:
    int i;
};
```

Dans ce cas, il n'est pas nécessaire d'allouer un octet à `Base` dans `Derived` pour avoir une adresse distincte par type et par objet. Si l'optimisation de la classe de base vide est effectuée (et qu'aucun remplissage n'est requis), alors `sizeof(Derived) == sizeof(int)` , c'est-à-dire qu'aucune allocation supplémentaire n'est effectuée pour la base vide. Cela est également possible avec plusieurs classes de base (en C ++, plusieurs bases ne peuvent pas avoir le même type, donc aucun problème ne se pose).

Notez que cela ne peut être effectué que si le premier membre de `Derived` diffère de type de l'une des classes de base. Cela inclut toute base commune directe ou indirecte. Si c'est le même type que l'une des bases (ou s'il y a une base commune), il faut au moins attribuer un seul octet pour s'assurer que deux objets distincts du même type n'ont pas la même adresse.

Introduction à la performance

C et C ++ sont bien connus en tant que langages haute performance - en grande partie en raison de la grande quantité de personnalisation du code, permettant à un utilisateur de spécifier les performances par choix de la structure.

Lors de l'optimisation, il est important d'évaluer le code pertinent et de comprendre complètement comment le code sera utilisé.

Les erreurs d'optimisation courantes incluent:

- **Optimisation prématurée:** le code complexe peut être *moins* performant après optimisation, perte de temps et d'effort. La première priorité devrait être d'écrire du code *correct* et *maintenable* , plutôt que du code optimisé.
- **Optimisation pour le cas d'utilisation incorrect: l'** ajout de frais généraux pour le 1% ne vaut peut-être pas le ralentissement pour les autres 99%

- **Micro-optimisation: les** compilateurs le font très efficacement et la micro-optimisation peut même nuire à la capacité des compilateurs à optimiser davantage le code

Les objectifs d'optimisation typiques sont:

- Faire moins de travail
- Utiliser des algorithmes / structures plus efficaces
- Pour mieux utiliser le matériel

Le code optimisé peut avoir des effets secondaires négatifs, notamment:

- Utilisation de mémoire supérieure
- Code complexe - difficile à lire ou à maintenir
- API et conception de code compromises

Optimiser en exécutant moins de code

L'approche la plus simple de l'optimisation consiste à exécuter moins de code. Cette approche donne généralement une accélération fixe sans changer la complexité temporelle du code.

Même si cette approche vous donne une accélération claire, cela ne vous apportera que des améliorations notables lorsque le code est appelé beaucoup.

Supprimer le code inutile

```
void func(const A *a); // Some random function

// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());

// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

C ++ 14

A partir de C ++ 14, les compilateurs sont autorisés à optimiser ce code pour supprimer l'allocation et la désallocation correspondante.

Faire du code une seule fois

```
std::map<std::string, std::unique_ptr<A>> lookup;
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a thirth time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}
```

```
// Within this function, we will have the same noticeable effect as the slow variant while
going at double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

Une approche similaire à cette optimisation peut être utilisé pour mettre en œuvre une version stable de `unique` ,

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
already in or not
        // This prevents an insertion, which will traverse through the map for every unique
element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Prévenir les réaffectations inutiles et la copie / déplacement

Dans l'exemple précédent, nous avons déjà empêché les recherches dans le std :: set, cependant le `std::vector` contient toujours un algorithme croissant, dans lequel il devra réallouer son stockage. Cela peut être évité en réservant d'abord pour la bonne taille.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the
vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
generated code
    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See example above
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Utiliser des conteneurs efficaces

Optimiser en utilisant les bonnes structures de données au bon moment peut modifier la complexité temporelle du code.

```
// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

En utilisant un conteneur qui utilise une implémentation différente pour stocker ses éléments (conteneur de hachage au lieu de l'arbre), nous pouvons transformer notre implémentation en complexité N. En tant qu'effet secondaire, nous appellerons l'opérateur de comparaison pour std :: string less, car ne doit être appelé que lorsque la chaîne insérée doit se retrouver dans le même compartiment.

```
// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Optimisation des petits objets

L'optimisation des petits objets est une technique utilisée dans les structures de données de bas niveau, par exemple la `std::string` (parfois appelée optimisation de chaîne courte / petite). Il est destiné à utiliser l'espace de pile comme tampon au lieu de la mémoire allouée au cas où le contenu serait suffisamment petit pour tenir dans l'espace réservé.

En ajoutant des surcharges de mémoire et des calculs supplémentaires, il essaie d'empêcher une allocation de segment de mémoire coûteuse. Les avantages de cette technique dépendent de l'utilisation et peuvent même nuire à la performance s'ils sont utilisés de manière incorrecte.

Exemple

Une manière très naïve d'implémenter une chaîne avec cette optimisation serait la suivante:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};                       ///< Remember if we allocated memory
    char *_buffer{nullptr};                         ///< Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE]= {'\0'};   ///< Stack space used for SMALL OBJECT
OPTIMIZATION

public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
       : _isAllocated(rhs._isAllocated)
       , _buffer(rhs._buffer)
       , _smallBuffer(rhs._smallBuffer) //< Not needed if allocated
    {
        if (_isAllocated)
        {
           // Prevent double deletion of the memory
           rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
    }
    // Other methods, including other constructors, copy constructor,
    // assignment operators have been omitted for readability
};
```

Comme vous pouvez le voir dans le code ci-dessus, une complexité supplémentaire a été ajoutée afin d'empêcher certaines opérations `new` et de `delete` . En plus de cela, la classe a une empreinte mémoire plus grande qui pourrait ne pas être utilisée sauf dans quelques cas.

On essaie souvent de coder la valeur bool `_isAllocated` , dans le pointeur `_buffer` avec manipulation des bits pour réduire la taille d'une seule instance (intel 64 bit: peut réduire la taille de 8 octets). Une optimisation qui n'est possible que lorsque ses règles d'alignement sont connues.

Quand utiliser?

Comme cette optimisation ajoute beaucoup de complexité, il n'est pas recommandé d'utiliser cette optimisation sur chaque classe. Cela se rencontrera souvent dans les structures de données de bas niveau couramment utilisées. Dans les implémentations de `standard library` C ++ 11 courantes, on peut trouver des utilisations dans `std::basic_string<>` et `std::function<>` .

Comme cette optimisation n'empêche que les allocations de mémoire lorsque les données stockées sont plus petites que le tampon, cela ne donnera des avantages que si la classe est souvent utilisée avec de petites données.

Un dernier inconvénient de cette optimisation est que des efforts supplémentaires sont nécessaires lors du déplacement du tampon, ce qui rend l'opération de déplacement plus coûteuse que lorsque le tampon ne serait pas utilisé. Cela est particulièrement vrai lorsque le tampon contient un type non-POD.

Lire Optimisation en C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/4474/optimisation-en-c-plusplus

Chapitre 86: Outils et techniques de débogage et de prévention du débogage C ++

Introduction

Les développeurs C ++ consacrent beaucoup de temps au débogage. Ce sujet est destiné à faciliter cette tâche et à inspirer des techniques. Ne vous attendez pas à une liste exhaustive des problèmes et solutions résolus par les outils ou un manuel sur les outils mentionnés.

Remarques

Ce sujet n'est pas encore complet, des exemples sur les techniques / outils suivants seraient utiles:

- Mentionnez plus d'outils d'analyse statique
- Outils d'instrumentation binaire (comme UBSan, TSan, MSan, ESan ...)
- Durcissement (CFI ...)
- Fuzzing

Examples

Mon programme C ++ se termine par segfault - valgrind

Ayons un programme de base défaillant:

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p3 << std::endl;
    }
}

int main() {
    fail();
}
```

Construisez-le (add -g pour inclure les informations de débogage):

```
g++ -g -o main main.cpp
```

Courir:

```
$ ./main
Segmentation fault (core dumped)
$
```

Déboguons-le avec valgrind:

```
$ valgrind ./main
==8515== Memcheck, a memory error detector
==8515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8515== Command: ./main
==8515==
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==8515==
==8515==
==8515== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==8515==  Access not within mapped region at address 0x0
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  If you believe this happened as a result of a stack
==8515==  overflow in your program's main thread (unlikely but
==8515==  possible), you can try to increase the size of the
==8515==  main thread stack using the --main-stacksize= flag.
==8515==  The main thread stack size used in this run was 8388608.
==8515==
==8515== HEAP SUMMARY:
==8515==     in use at exit: 72,704 bytes in 1 blocks
==8515==   total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated
==8515==
==8515== LEAK SUMMARY:
==8515==    definitely lost: 0 bytes in 0 blocks
==8515==    indirectly lost: 0 bytes in 0 blocks
==8515==      possibly lost: 0 bytes in 0 blocks
==8515==    still reachable: 72,704 bytes in 1 blocks
==8515==         suppressed: 0 bytes in 0 blocks
==8515== Rerun with --leak-check=full to see details of leaked memory
==8515==
==8515== For counts of detected and suppressed errors, rerun with: -v
==8515== Use --track-origins=yes to see where uninitialised values come from
==8515== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
$
```

Tout d'abord, nous nous concentrons sur ce bloc:

```
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

La première ligne nous indique que segfault est provoqué par la lecture de 4 octets. Les deuxième et troisième lignes sont des piles d'appels. Cela signifie que la lecture invalide est effectuée à la

fonction `fail()` , ligne 8 de main.cpp, appelée par main, ligne 13 de main.cpp.

En regardant la ligne 8 de main.cpp, nous voyons

```
std::cout << *p3 << std::endl;
```

Mais on vérifie d'abord le pointeur, alors qu'est-ce qui ne va pas? Permet de vérifier l'autre bloc:

```
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
```

Il nous dit qu'il y a une variable unitialisée à la ligne 7 et nous la lisons:

```
if (p3) {
```

Ce qui nous indique la ligne où nous vérifions p3 au lieu de p2. Mais comment est-il possible que p3 ne soit pas initialisé? Nous l'initialisons par:

```
int *p3 = p1;
```

Valgrind nous conseille de réexécuter avec `--track-origins=yes` , faisons-le:

```
valgrind --track-origins=yes ./main
```

L'argument pour valgrind est juste après valgrind. Si nous le mettons après notre programme, il serait transmis à notre programme.

La sortie est presque la même, il n'y a qu'une seule différence:

```
==8517== Conditional jump or move depends on uninitialised value(s)
==8517==    at 0x400813: fail() (main.cpp:7)
==8517==    by 0x40083F: main (main.cpp:13)
==8517==  Uninitialised value was created by a stack allocation
==8517==    at 0x4007F6: fail() (main.cpp:3)
```

Ce qui nous indique que la valeur non initialisée que nous avons utilisée à la ligne 7 a été créée à la ligne 3:

```
int *p1;
```

qui nous guide vers notre pointeur non initialisé.

Analyse Segfault avec GDB

Utilisons le même code que ci-dessus pour cet exemple.

```
#include <iostream>
```

```
void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

Commençons par le compiler

```
g++ -g -o main main.cpp
```

Permet de l'exécuter avec gdb

```
gdb ./main
```

Maintenant, nous serons dans le shell gdb. Tapez run.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400850 in fail () at debugging_with_gdb.cc:11
11            std::cout << *p2 << std::endl;
```

Nous voyons que la faute de segmentation se produit à la ligne 11. La seule variable utilisée à cette ligne est le pointeur p2. Permet d'examiner son impression de saisie de contenu.

```
(gdb) print p2
$1 = (int *) 0x0
```

Nous voyons maintenant que p2 a été initialisé à 0x0, ce qui signifie NULL. Sur cette ligne, nous savons que nous essayons de déréférencer un pointeur NULL. Nous allons donc le réparer.

Code propre

Le débogage commence par la compréhension du code que vous essayez de déboguer.

Mauvais code:

```
int main() {
    int value;
    std::vector<int> vectorToSort;
    vectorToSort.push_back(42); vectorToSort.push_back(13);
```

```
    for (int i = 52; i; i = i - 1)
    {
    vectorToSort.push_back(i *2);
    }
    /// Optimized for sorting small vectors
    if (vectorToSort.size() == 1);
    else
        {
        if (vectorToSort.size() <= 2)
            std::sort(vectorToSort.begin(), std::end(vectorToSort));
        }
    for (value : vectorToSort) std::cout << value << ' ';
return 0; }
```

Meilleur code:

```
std::vector<int> createSemiRandomData() {
    std::vector<int> data;
    data.push_back(42);
    data.push_back(13);
    for (int i = 52; i; --i)
        vectorToSort.push_back(i *2);
    return data;
}

/// Optimized for sorting small vectors
void sortVector(std::vector &v) {
    if (vectorToSort.size() == 1)
        return;
    if (vectorToSort.size() > 2)
        return;

    std::sort(vectorToSort.begin(), vectorToSort.end());
}

void printVector(const std::vector<int> &v) {
    for (auto i : v)
        std::cout << i << ' ';
}

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
 }
```

Quels que soient les styles de codage que vous préférez et que vous utilisez, le fait d’avoir un style de codage (et de mise en forme) cohérent vous aidera à comprendre le code.

En regardant le code ci-dessus, on peut identifier quelques améliorations pour améliorer la lisibilité et le débogage:

L'utilisation de fonctions séparées pour des actions séparées

L'utilisation de fonctions séparées vous permet d'ignorer certaines fonctions du débogueur si les détails ne vous intéressent pas. Dans ce cas précis, la création ou l'impression des données pourrait ne pas vous intéresser et vous ne souhaiterez intervenir que dans le tri.

Un autre avantage est que vous devez lire moins de code (et le mémoriser) tout en parcourant le code. Il ne vous reste plus qu'à lire 3 lignes de code dans `main()` pour le comprendre, au lieu de l'intégralité de la fonction.

Le troisième avantage est que vous avez simplement moins de code à regarder, ce qui aide un œil averti à repérer ce bug en quelques secondes.

Utiliser des mises en forme / constructions cohérentes

L'utilisation de mises en forme et de constructions cohérentes permet de supprimer l'encombrement du code, ce qui facilite la concentration sur le code plutôt que sur le texte. Beaucoup de discussions ont été nourries sur le bon style de mise en forme. Indépendamment de ce style, avoir un style unique et cohérent dans le code améliorera la familiarité et facilitera la mise au point du code.

Comme le code de formatage prend beaucoup de temps, il est recommandé d'utiliser un outil dédié à cette fin. La plupart des IDE ont au moins une sorte de support pour cela et peuvent rendre le formatage plus cohérent que les humains.

Vous remarquerez peut-être que le style ne se limite pas aux espaces et aux nouvelles lignes car nous ne mélangeons plus les fonctions de style libre et les fonctions de membre pour obtenir le début / la fin du conteneur. (`v.begin()` vs `std::end(v)`).

Faites attention aux parties importantes de votre code.

Quel que soit le style que vous choisissez de choisir, le code ci-dessus contient quelques marqueurs qui pourraient vous donner une idée de ce qui pourrait être important:

- Un commentaire affirmant `optimized` , cela indique des techniques de fantaisie
- Certains retours précoces dans `sortVector()` indiquent que nous faisons quelque chose de spécial
- Le `std::ref()` indique que quelque chose se passe avec le `sortVector()`

Conclusion

Avoir du code propre vous aidera à comprendre le code et réduira le temps nécessaire pour le déboguer. Dans le deuxième exemple, un réviseur de code pourrait même détecter le bogue à première vue, tandis que le bogue pourrait être caché dans les détails du premier. (PS: Le bogue est en comparaison avec `2`)

Analyse statique

L'analyse statique est la technique par laquelle on vérifie le code à la recherche de motifs liés à des bogues connus. L'utilisation de cette technique prend moins de temps qu'une revue de code, mais ses vérifications ne sont limitées qu'à celles programmées dans l'outil.

Les vérifications peuvent inclure le point-virgule incorrect derrière l'instruction `if (var);` (`if (var);`) jusqu'à des algorithmes de graphe avancés qui déterminent si une variable n'est pas initialisée.

Avertissements du compilateur

Activer l'analyse statique est facile, la version la plus simpliste est déjà intégrée dans votre compilateur:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

Si vous activez ces options, vous remarquerez que chaque compilateur trouvera des bogues que les autres ne détectent pas et que vous obtiendrez des erreurs sur des techniques qui pourraient être valides ou valides dans un contexte spécifique. `while (staticAtomicBool);` pourrait être acceptable même si `while (localBool);` n'est pas.

Donc, contrairement à la révision de code, vous vous battez contre un outil qui comprend votre code, vous indique beaucoup de bogues utiles et est parfois en désaccord avec vous. Dans ce dernier cas, vous devrez peut-être supprimer l'avertissement localement.

Comme les options ci-dessus activent tous les avertissements, elles peuvent activer les avertissements que vous ne souhaitez pas. (Pourquoi votre code devrait-il être compatible C ++ 98?) Si oui, vous pouvez simplement désactiver cet avertissement spécifique:

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Lorsque les avertissements du compilateur vous aident pendant le développement, ils ralentissent un peu la compilation. C'est pourquoi vous ne souhaitez peut-être pas toujours les activer par défaut. Soit vous les exécutez par défaut, soit vous permettez une intégration continue avec les contrôles les plus coûteux (ou tous).

Outils externes

Si vous décidez d'avoir une intégration continue, l'utilisation d'autres outils n'est pas un tel effort. Un outil comme clang-tidy a une liste de vérifications qui couvre un large éventail de questions, quelques exemples:

- Bugs réels
  - Prévention du tranchage
  - Affirme avec des effets secondaires
- Contrôles de lisibilité
  - Indication trompeuse

    - Vérifier le nom de l'identificateur
- Contrôles de modernisation
    - Utilisez make_unique ()
    - Utilisez nullptr
- Contrôles de performance
    - Trouver des copies inutiles
    - Trouver des appels d'algorithme inefficaces

La liste pourrait ne pas être aussi grande, car Clang a déjà beaucoup d'avertissements sur le compilateur, mais cela vous rapprochera encore plus d'une base de code de haute qualité.

Autres outils

D'autres outils ayant un but similaire existent, comme:

- l'analyseur statique de studio visuel comme outil externe
- clazy , un plugin de compilateur Clang pour vérifier le code Qt

Conclusion

De nombreux outils d'analyse statique existent pour C ++, tous deux intégrés au compilateur en tant qu'outils externes. Les essayer ne prend pas beaucoup de temps pour les configurations faciles et ils trouveront des bogues que vous pourriez manquer dans la révision du code.

Safe-stack (corruptions de piles)

Les corruptions de piles sont des bogues ennuyeux à regarder. Comme la pile est corrompue, le débogueur ne peut souvent pas vous donner une bonne trace de votre emplacement et de la manière dont vous l'avez obtenu.

C'est là que la pile de sécurité entre en jeu. Au lieu d'utiliser une seule pile pour vos threads, il utilisera deux: Une pile sécurisée et une pile dangereuse. La pile sécurisée fonctionne exactement comme avant, sauf que certaines pièces sont déplacées vers la pile dangereuse.

Quelles parties de la pile sont déplacées?

Chaque partie susceptible de corrompre la pile sera retirée de la pile sécurisée. Dès qu'une variable de la pile est passée par référence ou que l'on prend l'adresse de cette variable, le compilateur décidera de l'allouer sur la deuxième pile au lieu de la sûre.

Par conséquent, toute opération effectuée avec ces pointeurs, toute modification apportée à la mémoire (basée sur ces pointeurs / références) ne peut affecter que la mémoire de la deuxième pile. Comme on n'obtient jamais un pointeur proche de la pile sécurisée, la pile ne peut pas corrompre la pile et le débogueur peut toujours lire toutes les fonctions de la pile pour donner une belle trace.

A quoi sert-il réellement?

La pile sécurisée n'a pas été inventée pour vous donner une meilleure expérience de débogage, cependant, c'est un effet secondaire intéressant pour les bugs méchants. Son objectif initial est de faire partie du projet d'intégrité du pointeur de code (CPI) , dans lequel il tente d'empêcher de remplacer les adresses de retour pour empêcher l'injection de code. En d'autres termes, ils essaient d'empêcher l'exécution d'un code de piratage.

Pour cette raison, la fonctionnalité a été activée sur chrome et a été signalée comme ayant une surcharge de processeur inférieure à 1%.

Comment l'activer?

Pour le moment, l'option n'est disponible que dans le compilateur Clang , où l'on peut passer `-fsanitize=safe-stack` au compilateur. Une proposition a été faite pour implémenter la même fonctionnalité dans GCC.

Conclusion

Les corruptions de pile peuvent devenir plus faciles à déboguer lorsque la pile sécurisée est activée. En raison de la surcharge de performances, vous pouvez même activer par défaut dans votre configuration de construction.

Lire Outils et techniques de débogage et de prévention du débogage C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/9814/outils-et-techniques-de-debogage-et-de-prevention-du-debogage-c-plusplus

Chapitre 87: Pack de paramètres

Examples

Un modèle avec un pack de paramètres

```
template<class ... Types> struct Tuple {};
```

Un paquet de paramètres est un paramètre de modèle acceptant zéro ou plusieurs arguments de modèle. Si un gabarit a au moins un paquet de paramètres est un *gabarit variadic* .

Extension d'un pack de paramètres

Le pattern `parameter_pack ...` est développé en une liste de substitutions de `parameter_pack` séparées par des virgules avec chacun de ses paramètres

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
  std::cout << first_argument << "\n";
  variadic_printer(other_arguments...); // Parameter pack expansion
}
```

Le code ci-dessus invoqué avec `variadic_printer(1, 2, 3, "hello");` estampes

```
1
2
3
hello
```

Lire Pack de paramètres en ligne: https://riptutorial.com/fr/cplusplus/topic/7668/pack-de-parametres

Chapitre 88: Pimpl Idiom

Remarques

L'idiome de Pimpl (p ointer à **impl** ementation, parfois appelé *pointeur* ou *technique chat cheshire opaque),* réduit les temps de compilation d'une classe en déplaçant tous ses membres de données privées dans un struct défini dans le fichier .cpp.

La classe possède un pointeur sur l'implémentation. De cette façon, il peut être déclaré de manière à ce que le fichier d'en-tête n'ait pas besoin d' `#include` classes utilisées dans les variables de membre privé.

Lors de l'utilisation de l'idiome pimpl, la modification d'un membre de données privé ne nécessite pas de recompiler les classes qui en dépendent.

Examples

Idiome de base de Pimpl

C ++ 11

Dans le fichier d'en-tête:

```
// widget.h

#include <memory>  // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    public:
        Widget();
        ~Widget();
        void DoSomething();

    private:
        // the pImpl idiom is named after the typical variable name used
        // ie, pImpl:
        struct Impl;                    // forward declaration
        std::experimental::propagate_const<std::unique_ptr< Impl >> pImpl;  // ptr to actual
implementation
};
```

Dans le fichier d'implémentation:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // no need to include this header inside widget.h
```

```
struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
    pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}
```

Le `pImpl` contient l'état du `Widget` (ou une partie / la majeure partie). Au lieu que la description de l'état du `Widget` soit exposée dans le fichier d'en-tête, elle ne peut être exposée que dans l'implémentation.

`pImpl` signifie "pointer to implementation". La "vraie" implémentation de `Widget` trouve dans `pImpl` .

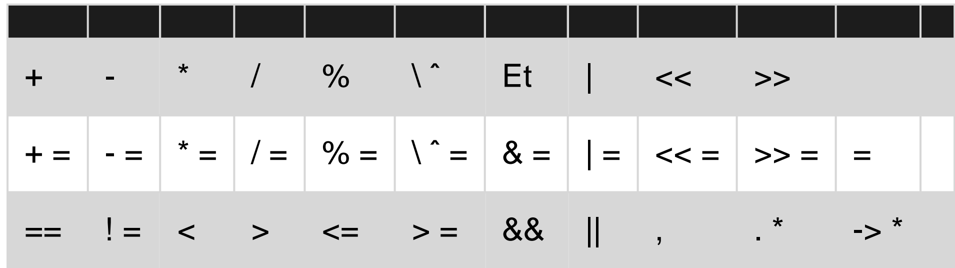
Danger: Notez que pour que cela fonctionne avec `unique_ptr` , `~Widget()` doit être implémenté en un point dans un fichier où l' `Impl` est entièrement visible. Vous pouvez `=default` , mais si `=default` où `Impl` est indéfini, le programme peut facilement devenir mal formé, aucun diagnostic requis.

Lire Pimpl Idiom en ligne: https://riptutorial.com/fr/cplusplus/topic/2143/pimpl-idiom

Chapitre 89: Plier les expressions

Remarques

Les expressions de pliage sont prises en charge pour les opérateurs suivants



+	-	*	/	%	\ ˆ	Et	|	<<	>>		
+ =	- =	* =	/ =	% =	\ ˆ =	& =	| =	<< =	>> =	=	
==	! =	<	>	<=	> =	&&	||	,	. *	-> *	

Lors du pliage d'une séquence vide, une expression de pli est mal formée, à l'exception des trois opérateurs suivants:

Opérateur	Valeur lorsque le pack de paramètres est vide
&&	vrai
||	faux
,	vide()

Examples

Unary Folds

Les plis unaires sont utilisés pour *plier les* paquets de paramètres sur un opérateur spécifique. Il existe 2 types de plis unaires:

- Unary **Left** Fold `(... op pack)` qui se développe comme suit:

  ```
  ((Pack1 op Pack2) op ...) op PackN
  ```

- Unary **Right** Fold `(pack op ...)` qui se développe comme suit:

  ```
  Pack1 op (... (Pack(N-1) op PackN))
  ```

Voici un exemple

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold
```

```
    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); //  6
```

Plis binaires

Les plis binaires sont essentiellement des plis unaires , avec un argument supplémentaire.

Il existe 2 types de plis binaires:

- Binary **Left** Fold - `(value op ... op pack)` - Développe comme suit:

  ```
  (((Value op Pack1) op Pack2) op ...) op PackN
  ```

- Binary **Right** Fold `(pack op ... op value)` - Se développe comme suit:

  ```
  Pack1 op (... op (Pack(N-1) op (PackN op Value)))
  ```

Voici un exemple:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}

int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Plier sur une virgule

C'est une opération courante de devoir exécuter une fonction particulière sur chaque élément d'un pack de paramètres. Avec C ++ 11, le mieux que nous puissions faire est de:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Mais avec une expression de pliage, ce qui précède simplifie bien:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
```

```
}
```

Pas de passe-partout cryptique requis.

Lire Plier les expressions en ligne: https://riptutorial.com/fr/cplusplus/topic/2676/plier-les-expressions

Chapitre 90: Pointeurs

Introduction

Un pointeur est une adresse faisant référence à un emplacement en mémoire. Ils sont couramment utilisés pour permettre aux fonctions ou aux structures de données de connaître et de modifier la mémoire sans avoir à copier la mémoire mentionnée. Les pointeurs sont utilisables avec les types primitifs (intégrés) ou définis par l'utilisateur.

Les pointeurs utilisent les opérateurs "dereference" `*` , "address of" `&` , "arrow" `->` . Les opérateurs '*' et '->' sont utilisés pour accéder à la mémoire pointée, et l'opérateur `&` est utilisé pour obtenir une adresse en mémoire.

Syntaxe

- <Type de données> * <Nom de la variable>;
- <Type de données> * <Nom de la variable> = & <Nom de la variable du même type de données>;
- <Type de données> * <Nom de la variable> = <Valeur du même type de données>;
- int * foo; // Un pointeur qui pointe vers un nombre entier
- int * bar = & myIntVar;
- long * bar [2];
- long * bar [] = {& myLongVar1, & myLongVar2}; // Égal à: long * bar [2]

Remarques

Soyez conscient des problèmes lors de la déclaration de plusieurs pointeurs sur la même ligne.

```
int* a, b, c; //Only a is a pointer, the others are regular ints.

int* a, *b, *c; //These are three pointers!

int *foo[2]; //Both *foo[0] and *foo[1] are pointers.
```

Examples

Notions de base sur les pointeurs

C ++ 11

Note: dans tout ce qui suit, l'existence de la constante C ++ 11 `nullptr` est supposée. Pour les versions antérieures, remplacez `nullptr` par `NULL` , la constante utilisée pour jouer un rôle similaire.

Créer une variable de pointeur

Une variable de pointeur peut être créée en utilisant la syntaxe spécifique `*` , par exemple `int *pointer_to_int;` .
Lorsqu'une variable est de *type pointeur* (`int *`), elle ne contient qu'une adresse mémoire. L'adresse mémoire est l'emplacement auquel sont stockées les données du *type sous* - *jacent* (`int`).

La différence est claire lorsque l'on compare la taille d'une variable avec la taille d'un pointeur au même type:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Prendre l'adresse d'une autre variable

Les pointeurs peuvent être assignés entre eux simplement comme des variables normales; dans ce cas, c'est l' **adresse mémoire** qui est copiée d'un pointeur à un autre, et **non les données réelles sur** lesquelles pointe un pointeur.
De plus, ils peuvent prendre la valeur `nullptr` qui représente un emplacement de mémoire nul. Un pointeur égal à `nullptr` contient un emplacement de mémoire non valide et ne fait donc pas référence à des données valides.

Vous pouvez obtenir l'adresse mémoire d'une variable d'un type donné en préfixant la variable avec l' *adresse de l'* opérateur `&` . La valeur renvoyée par `&` est un pointeur sur le type sous-jacent qui contient l'adresse mémoire de la variable (qui sont des données valides **tant que la variable ne sort pas du cadre**).

```
// Copy `p_bar0` into `p_bar_1`.
```

```
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

Contrairement aux références:

- l'affectation de deux pointeurs n'écrase pas la mémoire à laquelle se réfère le pointeur assigné;
- les pointeurs peuvent être nuls.
- l' *adresse de l'* opérateur est requise explicitement.

Accéder au contenu d'un pointeur

Comme prendre une adresse nécessite `&` , aussi bien accéder au contenu nécessite l'utilisation de l' *opérateur de déréférencement* `*` , comme préfixe. Lorsqu'un pointeur est déréférencé, il devient une variable du type sous-jacent (en fait, une référence à celui-ci). Il peut alors être lu et modifié, sinon `const` .

```
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

La combinaison de `*` et de l'opérateur `.` est abrégé par `->` :

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " <<  p_bar0->foo1  << std::endl; // Prints 5
```

Déréférencer les pointeurs invalides

Lors du déréférencement d'un pointeur, vous devez vous assurer qu'il pointe vers des données valides. Le fait de déréférencer un pointeur non valide (ou un pointeur nul) peut entraîner une violation de l'accès à la mémoire ou lire ou écrire des données erronées.

```
big_struct *never_do_this() {
   // This is a local variable. Outside `never_do_this` it doesn't exist.
   big_struct retval;
   retval.foo1 = 11;
   // This returns the address of `retval`.
   return &retval;
   // `retval` is destroyed and any code using the value returned
   // by `never_do_this` has a pointer to a memory location that
   // contains garbage data (or is inaccessible).
}
```

Dans un tel scénario, `g++` et `clang++` émettent correctement les avertissements:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-
Wreturn-stack-address]
(Gcc)   warning: address of local variable ‘retval’ returned [-Wreturn-local-addr]
```

Par conséquent, il faut faire attention lorsque les pointeurs sont des arguments de fonctions, car ils peuvent être nuls:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Opérations de pointeur

Il existe deux opérateurs pour les pointeurs: Adresse-de l'opérateur (&): Retourne l'adresse mémoire de son opérande. Opérateur Contents-of (Dereference) (*): Renvoie la valeur de la variable située à l'adresse spécifiée par son opérateur.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr
```

L'astérisque (*) est utilisé pour déclarer un pointeur dans le seul but d'indiquer qu'il s'agit d'un pointeur. Ne confondez pas cela avec l'opérateur de **déréférence** , qui est utilisé pour obtenir la

valeur située à l'adresse spécifiée. Ce sont simplement deux choses différentes représentées avec le même signe.

Arithmétique du pointeur

Incrémenter / Décrémenter

Un pointeur peut être incrémenté ou décrémenté (préfixe et postfix). L'incrémentation d'un pointeur fait avancer la valeur du pointeur vers l'élément du tableau un élément au-delà de l'élément actuellement pointé. Décrémenter un pointeur le déplace vers l'élément précédent du tableau.

L'arithmétique du pointeur n'est pas autorisée si le type sur lequel pointe le pointeur n'est pas complet. `void` est toujours un type incomplet.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011  in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

Si un pointeur sur l'élément final est incrémenté, le pointeur pointe sur un élément situé après la fin du tableau. Un tel pointeur ne peut pas être déréférencé, mais il peut être décrémenté.

L'incrémentation d'un pointeur sur l'élément one-past-the-end du tableau ou la décrémentation d'un pointeur sur le premier élément d'un tableau entraîne un comportement indéfini.

Un pointeur vers un objet non-tableau peut être traité, à des fins d'arithmétique de pointeur, comme s'il s'agissait d'un tableau de taille 1.

Addition soustraction

Les valeurs entières peuvent être ajoutées aux pointeurs. ils agissent comme incrémentation, mais par un nombre spécifique plutôt que par 1. Les valeurs entières peuvent également être soustraites des pointeurs, agissant comme une décrémentation du pointeur. Comme pour l'incrémentation / décrémentation, le pointeur doit pointer vers un type complet.

```
char* str = new char[10];  // str = 0x010
str += 2;                  // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10];    // arr = 0x100
arr += 2;                  // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) ==
4.
```

Différence de pointeur

La différence entre deux pointeurs sur le même type peut être calculée. Les deux pointeurs doivent être dans le même objet tableau; Résultats de comportement autrement indéfinis.

Étant donné deux pointeurs `P` et `Q` dans le même tableau, si `P` est le `i` ème élément du tableau, et `Q` est le `j` e élément, alors `P - Q` est `i - j` . Le type du résultat est `std::ptrdiff_t` , de `<cstddef>` .

```
char* start = new char[10];  // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Lire Pointeurs en ligne: https://riptutorial.com/fr/cplusplus/topic/3056/pointeurs

Chapitre 91: Pointeurs aux membres

Syntaxe

- En supposant une classe nommée Class ...
  - tapez * ptr = & Class :: member; // pointe uniquement sur les membres statiques
  - type Class :: * ptr = & Class :: member; // pointe vers des membres de classe non statiques
- Pour les pointeurs vers des membres de classe non statiques, compte tenu des deux définitions suivantes:
  - Instance de classe;
  - Classe * p = & instance;
- Pointeurs vers les variables membres de classe
  - ptr = & Class :: i; // pointe sur la variable i dans chaque classe
  - instance. * ptr = 1; // L'instance d'accès i
  - p -> * ptr = 1; // Accéder à p's i
- Pointeurs vers les fonctions des membres de la classe
  - ptr = & Class :: F; // pointe sur la fonction 'F' dans chaque classe
  - (instance. * ptr) (5); // Appel de l'instance F
  - (p -> * ptr) (6); // Appelez p's F

Examples

Pointeurs vers des fonctions membres statiques

Une fonction membre `static` est comme une fonction C / C ++ ordinaire, sauf avec la portée:

- Il se trouve dans une `class` , son nom doit donc être décoré du nom de la classe;
- Il est accessible, `public` , `protected` ou `private` .

Donc, si vous avez accès à la fonction membre `static` et que vous la décorez correctement, vous pouvez pointer sur la fonction comme toute fonction normale en dehors d'une `class` :

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
```

```
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;          // Point to one function
    fn(3);               // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);               // Call it
 } // main()
```

Pointeurs vers les fonctions membres

Pour accéder à une fonction membre d'une classe, vous devez avoir un "handle" pour l'instance particulière, soit comme instance elle-même, soit comme un pointeur ou une référence à celle-ci. Étant donné une instance de classe, vous pouvez pointer vers plusieurs de ses membres avec un pointeur vers membre, SI vous obtenez la syntaxe correcte! Bien sûr, le pointeur doit être déclaré comme étant du même type que celui sur lequel vous pointez ...

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
    fn = &Class::B;   // fn now points to B within any Class
    (p->*fn)(6);      // Pass 6 to c's (via p) function B (via fn)
} // main()
```

Contrairement aux pointeurs vers les variables membres (dans l'exemple précédent), l'association entre l'instance de classe et le pointeur membre doit être étroitement liée à des parenthèses, ce qui semble un peu étrange (comme si les éléments `.*` Et `->*` ne sont pas étranges) assez!)

Pointeurs vers les variables membres

Pour accéder à un membre d'une `class` , vous devez avoir un "handle" pour l'instance particulière, soit comme instance elle-même, soit comme un pointeur ou une référence à celle-ci. Étant donné une instance de `class` , vous pouvez pointer vers plusieurs de ses membres avec un pointeur vers membre, SI vous obtenez la syntaxe correcte! Bien sûr, le pointeur doit être déclaré comme étant du même type que celui sur lequel vous pointez ...

```
class Class {
public:
    int x, y, z;
    char m, n, o;
}; // Class

int x;  // Global variable

int main() {
    Class c;         // Need a Class instance to play with
    Class *p = &c;   // Need a Class pointer to play with

    int *p_i;        // Pointer to an int

    p_i = &x;        // Now pointing to x
    p_i = &c.x;      // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_c_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_c_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

La syntaxe du pointeur sur membre nécessite des éléments syntaxiques supplémentaires:

- Pour définir le type du pointeur, vous devez mentionner le type de base, ainsi que le fait qu'il se trouve dans une classe: `int Class::*ptr;` .
- Si vous avez une classe ou une référence et que vous voulez l' utiliser avec un pointeur à membre, vous devez utiliser le `.*` Opérateur (semblable au `.` Opérateur).
- Si vous avez un pointeur sur une classe et que vous souhaitez l'utiliser avec un pointeur sur membre, vous devez utiliser l'opérateur `->*` (similaire à l'opérateur `->`).

Pointeurs vers des variables membres statiques

Une variable membre `static` est juste comme une variable C / C ++ ordinaire, sauf avec la portée:

- Il se trouve dans une `class` , son nom doit donc être décoré du nom de la classe;
- Il est accessible, `public` , `protected` ou `private` .

Donc, si vous avez accès à la variable membre `static` et que vous la décorez correctement, vous pouvez pointer vers la variable comme toute variable normale en dehors d'une `class` :

```
class Class {
public:
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)
```

```
int j = 2;   // Just another global variable

int main() {
    int k = 3; // Local variable

    int *p;

    p = &k;   // Point to k
    *p = 2;   // Modify it
    p = &j;   // Point to j
    *p = 3;   // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;   // Modify it
} // main()
```

Lire Pointeurs aux membres en ligne: https://riptutorial.com/fr/cplusplus/topic/2130/pointeurs-aux-membres

Chapitre 92: Pointeurs intelligents

Syntaxe

- `std::shared_ptr<ClassType> variableName = std::make_shared<ClassType>(arg1, arg2, ...);`
- `std::shared_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`
- `std::unique_ptr<ClassType> variableName = std::make_unique<ClassType>(arg1, arg2, ...);` // C ++ 14
- `std::unique_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`

Remarques

C ++ n'est pas un langage géré par la mémoire. La mémoire allouée dynamiquement (c.-à-d. Les objets créés avec `new`) sera "fuite" si elle n'est pas explicitement libérée (avec `delete`). Il est de la responsabilité du programmeur de s'assurer que la mémoire allouée dynamiquement est libérée avant de supprimer le dernier pointeur sur cet objet.

Les pointeurs intelligents peuvent être utilisés pour gérer automatiquement la portée de la mémoire allouée dynamiquement (c.-à-d. Que lorsque la dernière référence du pointeur est hors de portée, elle est supprimée).

Les pointeurs intelligents sont préférés aux pointeurs "bruts" dans la plupart des cas. Ils expliquent la sémantique de propriété de la mémoire allouée dynamiquement en communiquant dans leurs noms si un objet doit être partagé ou possédé de manière unique.

Utilisez `#include <memory>` pour pouvoir utiliser des pointeurs intelligents.

Examples

Partage de propriété (std :: shared_ptr)

Le modèle de classe `std::shared_ptr` définit un pointeur partagé capable de partager la propriété d'un objet avec d'autres pointeurs partagés. Cela contraste avec `std::unique_ptr` qui représente la propriété exclusive.

Le comportement de partage est implémenté via une technique appelée comptage de référence, dans laquelle le nombre de pointeurs partagés qui pointent vers l'objet est stocké à côté. Lorsque ce nombre atteint zéro, soit par la destruction ou la réaffectation de la dernière instance `std::shared_ptr` , l'objet est automatiquement détruit.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Pour créer plusieurs pointeurs intelligents qui partagent le même objet, nous devons créer un autre `shared_ptr` qui utilise le premier pointeur partagé. Voici 2 façons de le faire:

```
std::shared_ptr<Foo> secondShared(firstShared);  // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared;                      // 2nd way: Assigning
```

L'une des manières ci-dessus fait de `secondShared` un pointeur partagé qui partage la propriété de notre instance de `Foo` avec `firstShared` .

Le pointeur intelligent fonctionne comme un pointeur brut. Cela signifie que vous pouvez utiliser `*` pour les déréférencer. L'opérateur normal `->` fonctionne également:

```
secondShared->test(); // Calls Foo::test()
```

Enfin, lorsque le dernier alias `shared_ptr` est hors de portée, le destructeur de notre instance `Foo` est appelé.

Avertissement: la construction d'un `shared_ptr` peut `bad_alloc` une exception `bad_alloc` lorsque des données supplémentaires pour la sémantique de propriété partagée doivent être allouées. Si le constructeur reçoit un pointeur normal, il suppose qu'il possède l'objet pointé et appelle le paramètre si une exception est levée. Cela signifie que `shared_ptr<T>(new T(args))` ne fuira pas un objet `T` si l'allocation de `shared_ptr<T>` échoue. Cependant, il est conseillé d'utiliser `make_shared<T>(args)` ou `make_shared<T>(args) allocate_shared<T>(alloc, args)` , ce qui permet à l'implémentation d'optimiser l'allocation de mémoire.

Allocation de tableaux ([]) à l'aide de shared_ptr

C ++ 11 C ++ 17

Malheureusement, il n'existe aucun moyen direct d'allouer des tableaux à l'aide de `make_shared<>` .

Il est possible de créer des tableaux pour `shared_ptr<>` utilisant `new` et `std::default_delete` .

Par exemple, pour allouer un tableau de 10 nombres entiers, nous pouvons écrire le code en tant que

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Spécifier `std::default_delete` est obligatoire ici pour vous assurer que la mémoire allouée est correctement nettoyée en utilisant `delete[]` .

Si nous connaissons la taille au moment de la compilation, nous pouvons le faire de la manière suivante:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
  std::shared_ptr<T> operator()const{
    auto r = std::make_shared<std::array<T,N>>();
    if (!r) return {};
```

```
    return {r.data(), r};
  }
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }
```

puis `make_shared_array<int[10]>` retourne un `shared_ptr<int>` pointant vers 10 ints tous construits par défaut.

C ++ 17

Avec C ++ 17, `shared_ptr` pris en charge les types de tableaux. Il n'est plus nécessaire de spécifier explicitement le tableau-deleter et le pointeur partagé peut être déréférencé à l'aide de l'opérateur d'index de tableau `[]` :

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Les pointeurs partagés peuvent pointer vers un sous-objet de l'objet qui lui appartient:

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

`p2` et `p1` possèdent tous deux l'objet de type `Foo` , mais `p2` pointe vers son membre `int x` . Cela signifie que si `p1` est hors de portée ou est réaffecté, l'objet `Foo` sous-jacent sera toujours en vie, garantissant que `p2` ne pendent pas.

Important: Un `shared_ptr` ne connaît que lui-même et tous les autres `shared_ptr` créés avec le constructeur alias. Il ne connaît aucun autre pointeur, y compris tous les autres `shared_ptr` créés avec une référence à la même instance de `Foo` :

```
Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                 // deleted already!!
```

Transfert de propriété de shared_ptr

Par défaut, `shared_ptr` incrémente le compte de référence et ne transfère pas la propriété. Cependant, il est possible de transférer la propriété en utilisant `std::move` :

```
shared_ptr<int> up = make_shared<int>();
```

```
// Transferring the ownership
shared_ptr<int> up2 = move(up);
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1
```

Partage avec propriété temporaire (std :: faiblesse_ptr)

Les instances de `std::weak_ptr` peuvent pointer vers des objets appartenant à des instances de `std::shared_ptr` tout en ne devenant que des propriétaires temporaires. Cela signifie que les pointeurs faibles ne modifient pas le compte de référence de l'objet et n'empêchent donc pas la suppression d'un objet si tous les pointeurs partagés de l'objet sont réaffectés ou détruits.

Dans l'exemple suivant, les instances de `std::weak_ptr` sont utilisées pour `std::weak_ptr` la destruction d'un objet d'arborescence:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

Au fur et à mesure que des nœuds enfants sont ajoutés aux enfants du nœud racine, leur `parent` membre `std::weak_ptr` est défini sur le nœud racine. Le `parent` membre est déclaré comme un pointeur faible par opposition à un pointeur partagé, de sorte que le compte de référence du nœud racine n'est pas incrémenté. Lorsque le nœud racine est réinitialisé à la fin de `main()` , la racine est détruite. Étant donné que les seules références `std::shared_ptr` restantes aux nœuds enfants étaient contenues dans les `children` collection root, tous les nœuds enfants sont également détruits par la suite.

En raison des détails de l'implémentation du bloc de contrôle, la mémoire allouée shared_ptr peut ne pas être libérée jusqu'à ce que le compteur de référence `shared_ptr` et le `weak_ptr` référence `weak_ptr` atteignent tous deux zéro.

```
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         {
             // std::make_shared is optimized by allocating only once
             // while std::shared_ptr<int>(new int(42)) allocates twice.
             // Drawback of std::make_shared is that control block is tied to our integer
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // sh memory should be released at this point...
         }
         // ... but wk is still alive and needs access to control block
     }
     // now memory is released (sh and wk)
}
```

Comme `std::weak_ptr` ne conserve pas son objet référencé en vie, l'accès direct aux données via `std::weak_ptr` n'est pas possible. Au lieu de cela, il fournit une fonction membre `lock()` qui tente de récupérer un `std::shared_ptr` vers l'objet référencé:

```
#include <cassert>
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         std::shared_ptr<int> sp;
         {
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // calling lock will create a shared_ptr to the object referenced by wk
             sp = wk.lock();
             // sh will be destroyed after this point, but sp is still alive
         }
         // sp still keeps the data alive.
         // At this point we could even call lock() again
         // to retrieve another shared_ptr to the same data from wk
         assert(*sp == 42);
         assert(!wk.expired());
         // resetting sp will delete the data,
         // as it is currently the last shared_ptr with ownership
         sp.reset();
         // attempting to lock wk now will return an empty shared_ptr,
         // as the data has already been deleted
         sp = wk.lock();
         assert(!sp);
         assert(wk.expired());
     }
}
```

Propriété unique (std :: unique_ptr)

C ++ 11

Un `std::unique_ptr` est un modèle de classe qui gère la durée de vie d'un objet stocké

dynamiquement. Contrairement à `std::shared_ptr` , l'objet dynamique `std::shared_ptr` qu'à *une* seule *instance* de `std::unique_ptr` ,

---

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Remarque: `std::unique_ptr` est disponible depuis C ++ 11 et `std::make_unique` depuis C ++ 14.)

Seule la variable `ptr` contient un pointeur sur un `int` alloué dynamiquement. Lorsqu'un pointeur unique possédant un objet est hors de portée, l'objet possédé est supprimé, c'est-à-dire que son destructeur est appelé si l'objet est de type classe et que la mémoire de cet objet est libérée.

Pour utiliser `std::unique_ptr` et `std::make_unique` avec des types de tableaux, utilisez leurs spécialisations de tableaux:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

Vous pouvez accéder à `std::unique_ptr` comme un pointeur brut, car il surcharge ces opérateurs.

Vous pouvez transférer la propriété du contenu d'un pointeur intelligent vers un autre pointeur en utilisant `std::move` , ce qui entraînera le pointeur intelligent d'origine à pointer sur `nullptr` .

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);

int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

Passer `unique_ptr` aux fonctions en paramètre:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

Renvoyer `unique_ptr` des fonctions. Il s’agit de la méthode C ++ 11 d’écriture des fonctions de

fabrique préférée, car elle transmet clairement la sémantique de propriété du retour: l'appelant est propriétaire du `unique_ptr` résultant et en est responsable.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Comparez ceci à:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

C ++ 14

Le modèle de classe `make_unique` est fourni depuis C ++ 14. Il est facile de l'ajouter manuellement au code C ++ 11:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

C ++ 11

Contrairement au pointeur intelligent *stupide* (`std::auto_ptr`), `unique_ptr` peut également être instancié avec l'allocation vectorielle (*pas* `std::vector`). Les exemples précédents concernaient les allocations *scalaires* . Par exemple, pour avoir un tableau entier alloué dynamiquement pour 10 éléments, vous devez spécifier `int[]` comme type de modèle (et pas seulement `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Qui peut être simplifié avec:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Maintenant, vous utilisez `arr_ptr` comme s'il s'agissait d'un tableau:

```
arr_ptr[2] =  10; // Modify third element
```

Vous ne devez pas vous soucier de la désaffectation. Cette version spécialisée du modèle appelle

les constructeurs et les destructeurs de manière appropriée. Utiliser une version vectorisée de `unique_ptr` ou un `vector` lui `vector` même - est un choix personnel.

Dans les versions antérieures à C ++ 11, `std::auto_ptr` était disponible. Contrairement à `unique_ptr` il est autorisé à copier `auto_ptr` s, sur lequel le `ptr` source perdra la propriété du pointeur contenu et la cible le recevra.

Utilisation de paramètres personnalisés pour créer un wrapper vers une interface C

De nombreuses interfaces C telles que SDL2 ont leurs propres fonctions de suppression. Cela signifie que vous ne pouvez pas utiliser directement les pointeurs intelligents:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Au lieu de cela, vous devez définir votre propre compteur. Les exemples utilisés ici utilisent la structure `SDL_Surface` qui devrait être libérée à l'aide de la fonction `SDL_FreeSurface()` , mais ils devraient pouvoir être adaptés à de nombreuses autres interfaces C.

Le deleter doit pouvoir être appelé avec un argument de pointeur et peut donc être, par exemple, un simple pointeur de fonction:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Tout autre objet appelable fonctionnera également, par exemple une classe avec un `operator()` :

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};

std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
                                                         // as the deleter is value-
initialized
```

Cela vous fournit non seulement une gestion de mémoire automatique sûre et sans surcharge (si vous utilisez `unique_ptr`), vous obtenez également une sécurité d'exception.

Notez que le paramètre deleter fait partie du type de `unique_ptr` , et que l'implémentation peut utiliser l' optimisation de la base vide pour éviter toute modification de la taille des paramètres personnalisés vides. Donc, bien que `std::unique_ptr<SDL_Surface, SurfaceDeleter>` et `std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)>` résolvent le même problème de la même manière, le premier type n'a toujours que la taille d'un pointeur ce dernier type doit contenir *deux* pointeurs: le `SDL_Surface*` et le pointeur de fonction! Lorsque vous avez des fonctions personnalisées, il est préférable d'emballer la fonction dans un type vide.

Dans les cas où le comptage de références est important, on pourrait utiliser un `shared_ptr` au lieu

d'un `unique_ptr` . Le `shared_ptr` stocke toujours un deleter, cela efface le type du deleter, ce qui peut être utile dans les API. Les inconvénients `shared_ptr` utilisation de `shared_ptr` sur `unique_ptr` comprennent un coût de mémoire plus élevé pour le stockage du compteur et un coût de performance pour le maintien du nombre de références.

```
// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

C ++ 17

Avec le `template auto` , nous pouvons encore plus facilement emballer nos filtres personnalisés:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Avec lequel l'exemple ci-dessus est simplement:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Ici, le but de l' `auto` est de gérer toutes les fonctions libres, qu'elles renvoient des `void` (par exemple `SDL_FreeSurface`) ou non (par exemple, `fclose`).

Propriété unique sans sémantique de déplacement (auto_ptr)

C ++ 11

REMARQUE: `std::auto_ptr` est obsolète dans C ++ 11 et sera supprimé dans C ++ 17. Vous ne devriez l'utiliser que si vous êtes obligé d'utiliser C ++ 03 ou une version antérieure et que vous êtes prêt à faire attention. Il est recommandé de passer à unique_ptr en combinaison avec `std::move` pour remplacer le comportement `std::auto_ptr` .

Avant d'avoir `std::unique_ptr` , avant que nous ayons la sémantique de déplacement, nous avions `std::auto_ptr` . `std::auto_ptr` fournit une propriété unique mais transfère la propriété sur la copie.

Comme avec tous les pointeurs intelligents, `std::auto_ptr` nettoie automatiquement les ressources (voir RAII):

```
{
    std::auto_ptr<int> p(new int(42));
    std::cout << *p;
```

```
} // p is deleted here, no memory leaked
```

mais autorise un seul propriétaire:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
  // px is now empty
```

Cela permet d'utiliser std :: auto_ptr pour garder la propriété explicite et unique au risque de perdre la propriété involontairement:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
       // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

Le transfert de propriété a eu lieu dans le constructeur "copy". Le constructeur de copie et l'opérateur d'assignation de copie `auto_ptr` prennent leurs opérandes par référence non `const` pour qu'ils puissent être modifiés. Un exemple de mise en œuvre pourrait être:

```
template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }

    /* other functions ... */
};
```

Cela rompt la sémantique de la copie, qui exige que la copie d'un objet vous laisse deux versions

équivalentes. Pour tout type copiable, `T` , je devrais pouvoir écrire:

```
T a = ...;
T b(a);
assert(b == a);
```

Mais pour `auto_ptr` , ce n'est pas le cas. Par conséquent, il n'est pas prudent de mettre `auto_ptr` s dans des conteneurs.

Obtenir un shared_ptr faisant référence à ceci

`enable_shared_from_this` vous permet d'obtenir un valide `shared_ptr` par exemple à `this` .

En dérivant votre classe du modèle de classe `enable_shared_from_this` , vous `shared_from_this` une méthode `shared_from_this` qui retourne une instance `shared_ptr` à `this` .

Notez que l'objet doit être créé en tant que `shared_ptr` à la première place:

```
#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Remarque (2) vous ne pouvez pas appeler `enable_shared_from_this` dans le constructeur.

```
#include <memory> // enable_shared_from_this

class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

Si vous utilisez `shared_from_this()` sur un objet n'appartenant pas à `shared_ptr` , tel qu'un objet automatique local ou un objet global, le comportement est indéfini. Depuis C ++ 17, il lance `std::bad_alloc` place.

Utiliser `shared_from_this()` partir d'un constructeur équivaut à l'utiliser sur un objet n'appartenant pas à `shared_ptr` , car les objets sont possédés par `shared_ptr` après le retour du constructeur.

Casting std :: shared_ptr pointeurs

Il est impossible d'utiliser directement `static_cast` , `const_cast` , `dynamic_cast` et `reinterpret_cast` sur `std::shared_ptr` pour récupérer une propriété de partage de pointeur avec le pointeur étant passé comme argument. Au lieu de cela, les fonctions `std::static_pointer_cast` , `std::const_pointer_cast` , `std::dynamic_pointer_cast` et `std::reinterpret_pointer_cast` doivent être utilisées:

```
struct Base { virtual ~Base() noexcept {}; };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Notez que `std::reinterpret_pointer_cast` n'est pas disponible en C ++ 11 et C ++ 14, car il était uniquement proposé par N3920 et adopté dans Library Fundamentals TS en février 2014 . Cependant, il peut être implémenté comme suit:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Ecrire un pointeur intelligent: value_ptr

Un `value_ptr` est un pointeur intelligent qui se comporte comme une valeur. Une fois copié, il copie son contenu. Une fois créé, il crée son contenu.

```
// Like std::default_delete:
template<class T>
struct default_copier {
  // a copier must handle a null T const* in and return null:
  T* operator()(T const* tin)const {
    if (!tin) return nullptr;
    return new T(*tin);
  }
  void operator()(void* dest, T const* tin)const {
    if (!tin) return;
    return new(dest) T(*tin);
  }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
// the value pointer type itself:
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
  class Base=std::unique_ptr<T, Deleter>
>
struct value_ptr:Base, private Copier {
  using copier_type=Copier;
  // also typedefs from unique_ptr
```

```
using Base::Base;

value_ptr( T const& t ):
  Base( std::make_unique<T>(t) ),
  Copier()
{}
value_ptr( T && t ):
  Base( std::make_unique<T>(std::move(t)) ),
  Copier()
{}
// almost-never-empty:
    value_ptr():
  Base( std::make_unique<T>() ),
  Copier()
{}
value_ptr( empty_ptr_t ) {}

value_ptr( Base b, Copier c={} ):
  Base(std::move(b)),
  Copier(std::move(c))
{}

Copier const& get_copier() const {
  return *this;
}

value_ptr clone() const {
  return {
    Base(
      get_copier()(this->get()),
      this->get_deleter()
    ),
    get_copier()
  };
}
value_ptr(value_ptr&&)=default;
value_ptr& operator=(value_ptr&&)=default;

value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
value_ptr& operator=(value_ptr const&o) {
  if (o && *this) {
    // if we are both non-null, assign contents:
    **this = *o;
  } else {
    // otherwise, assign a clone (which could itself be null):
    *this = o.clone();
  }
  return *this;
}
value_ptr& operator=( T const& t ) {
  if (*this) {
    **this = t;
  } else {
    *this = value_ptr(t);
  }
  return *this;
}
value_ptr& operator=( T && t ) {
  if (*this) {
    **this = std::move(t);
```

```
    } else {
      *this = value_ptr(std::move(t));
    }
    return *this;
  }
  T& get() { return **this; }
  T const& get() const { return **this; }
  T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  // operator-> from unique_ptr
};
template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
  return {std::make_unique<T>(std::forward<Args>(args)...)};
}
```

Ce value_ptr particulier n'est vide que si vous le construisez avec `empty_ptr_t` ou si vous en `empty_ptr_t` . Il expose le fait que c'est un `explicit operator bool() const` `unique_ptr` , donc `explicit operator bool() const` fonctionne dessus. `.get()` a été modifié pour renvoyer une référence (car elle n'est presque jamais vide) et `.get_pointer()` renvoie un pointeur à la place.

Ce pointeur intelligent peut être utile pour les cas `pImpl` , où nous voulons une sémantique de valeur, mais nous ne voulons pas non plus exposer le contenu de `pImpl` dehors du fichier d'implémentation.

Avec un `Copier` par défaut, il peut même gérer des classes de base virtuelles qui savent comment produire des instances de leurs dérivés et les transformer en types de valeur.

Lire Pointeurs intelligents en ligne: https://riptutorial.com/fr/cplusplus/topic/509/pointeurs-intelligents

Chapitre 93: Polymorphisme

Examples

Définir des classes polymorphes

L'exemple typique est une classe de forme abstraite, qui peut ensuite être dérivée en carrés, cercles et autres formes concrètes.

La classe parente:

Commençons par la classe polymorphe:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

Comment lire cette définition?

- Vous pouvez définir un comportement polymorphe par des fonctions membres introduites avec le mot-clé `virtual` . `get_surface()` et `describe_object()` seront évidemment implémentés différemment pour un carré que pour un cercle. Lorsque la fonction est appelée sur un objet, la fonction correspondant à la classe réelle de l'objet sera déterminée au moment de l'exécution.
- Cela n'a aucun sens de définir `get_surface()` pour une forme abstraite. C'est pourquoi la fonction est suivie de `= 0` . Cela signifie que la fonction est *une fonction virtuelle pure* .
- Une classe polymorphe doit toujours définir un destructeur virtuel.
- Vous pouvez définir des fonctions membres non virtuelles. Lorsque ces fonctions seront invoquées pour un objet, la fonction sera choisie en fonction de la classe utilisée au moment de la compilation. `get_double_surface()` est défini de cette manière.
- Une classe contenant au moins une fonction virtuelle pure est une classe abstraite. Les classes abstraites ne peuvent pas être instanciées. Vous ne pouvez avoir que des pointeurs ou des références d'un type de classe abstrait.

Classes dérivées

Une fois qu'une classe de base polymorphe est définie, vous pouvez la dériver. Par exemple:

```
class Square : public Shape {
    Point top_left;
```

```
    double side_length;
public:
    Square (const Point& top_left, double side)
       : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
                  << " with a length of " << side_length << std::endl;
    }
};
```

Quelques explications:

- Vous pouvez définir ou remplacer l'une des fonctions virtuelles de la classe parente. Le fait qu'une fonction soit virtuelle dans la classe parente le rend virtuel dans la classe dérivée. Pas besoin de redire le mot-clé `virtual` au compilateur. Mais il est recommandé d'ajouter le mot-clé `override` à la fin de la déclaration de la fonction, afin d'éviter les bogues subtils causés par des variations inaperçues de la signature de la fonction.
- Si toutes les fonctions virtuelles pures de la classe parente sont définies, vous pouvez instancier des objets pour cette classe, sinon elle deviendra également une classe abstraite.
- Vous n'êtes pas obligé de remplacer toutes les fonctions virtuelles. Vous pouvez conserver la version du parent si cela vous convient.

Exemple d'instanciation

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr;  // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Downcasting sûr

Supposons que vous ayez un pointeur sur un objet d'une classe polymorphe:

```
Shape *ps;                       // see example on defining a polymorphic class
ps =  get_a_new_random_shape();  // if you don't have such a function yet, you
                                 // could just write ps = new Square(0.0,0.0, 5);
```

un abaissement consisterait à convertir une `Shape` polymorphe générale en une `Shape` dérivée et plus spécifique, comme `Square` ou `Circle` .

Pourquoi baisser les bras?

La plupart du temps, vous n'avez pas besoin de savoir quel est le type réel de l'objet, car les fonctions virtuelles vous permettent de manipuler votre objet indépendamment de son type:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

Si vous n'avez pas besoin de baisses, votre conception serait parfaite.

Cependant, il se peut que vous deviez parfois baisser. Un exemple typique est lorsque vous souhaitez appeler une fonction non virtuelle qui n'existe que pour la classe enfant.

Prenons par exemple les cercles. Seuls les cercles ont un diamètre. Ainsi, la classe serait définie comme suit:

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
    Circle (const Point& center, double radius)
       : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

La fonction membre `get_diameter()` n'existe que pour les cercles. Il n'a pas été défini pour un objet `Shape` :

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

Comment baisser les bras?

Si vous savez avec certitude que `ps` pointe sur un cercle, vous pouvez opter pour un `static_cast` :

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

Ça fera l'affaire. Mais c'est très risqué: si `ps` apparaît par autre chose qu'un `Circle` le comportement de votre code sera indéfini.

Donc, plutôt que de jouer à la roulette russe, vous devez utiliser un `dynamic_cast` toute sécurité. Ceci est spécifiquement pour les classes polymorphes:

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();   // OUCH !!! Compilation error
```

```
    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Notez que `dynamic_cast` n'est pas possible sur une classe qui n'est pas polymorphe. Vous devez avoir au moins une fonction virtuelle dans la classe ou ses parents pour pouvoir l'utiliser.

Polymorphisme & Destructeurs

Si une classe est destinée à être utilisée de manière polymorphe, les instances dérivées étant stockées en tant que pointeurs / références de base, le destructeur de sa classe de base doit être `virtual` ou `protected` . Dans le premier cas, cela provoquera la destruction de l'objet par la `vtable` , en appelant automatiquement le destructeur correct en fonction du type dynamique. Dans ce dernier cas, la destruction de l'objet via un pointeur / référence de classe de base est désactivée et l'objet ne peut être supprimé que lorsqu'il est explicitement traité comme son type réel.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

struct ProtectedDestructor {
  protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Ces deux pratiques garantissent que le destructeur de la classe dérivée sera toujours appelé sur les instances de classe dérivées, empêchant les fuites de mémoire.

Lire Polymorphisme en ligne: https://riptutorial.com/fr/cplusplus/topic/1717/polymorphisme

Chapitre 94: Préprocesseur

Introduction

Le préprocesseur C est un analyseur / remplaçant de texte simple exécuté avant la compilation du code. Utilisé pour étendre et faciliter l'utilisation du langage C (et plus tard C ++), il peut être utilisé pour:

une. **Y compris d'autres fichiers en** utilisant `#include`

b. **Définir une macro de remplacement de texte à l'** aide de `#define`

c. **Compilation conditionnelle** utilisant `#if` `#ifdef`

ré. **Logique spécifique à la plate-forme / au compilateur** (en tant qu'extension de la compilation conditionnelle)

Remarques

Les instructions de préprocesseur sont exécutées avant que vos fichiers source ne soient remis au compilateur. Ils sont capables d'une logique conditionnelle de très bas niveau. Étant donné que les constructions de préprocesseur (par exemple, les macros de type objet) ne sont pas typées comme des fonctions normales (l'étape de prétraitement se produit avant la compilation), le compilateur ne peut pas appliquer les vérifications de type.

Examples

Inclure les gardes

Un fichier d'en-tête peut être inclus par d'autres fichiers d'en-tête. Un fichier source (unité de compilation) comprenant plusieurs en-têtes peut donc, indirectement, inclure plusieurs en-têtes plus d'une fois. Si un tel fichier d'en-tête contient plus d'une fois des définitions, le compilateur (après prétraitement) détecte une violation de la règle One Definition (par exemple, §3.2 du standard C ++ 2003) et émet un diagnostic et la compilation échoue.

L'inclusion multiple est empêchée en utilisant "include guards", qui sont parfois aussi appelés gardes d'en-tête ou macro-gardes. Celles-ci sont implémentées en utilisant les directives de préprocesseur `#define` , `#ifndef` , `#endif` .

par exemple

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo    //  a class definition
```

```
{
};

#endif
```

Le principal avantage de l'utilisation des gardes inclus est qu'ils fonctionneront avec tous les compilateurs et préprocesseurs conformes aux normes.

Cependant, les gardes d'inclusion posent également des problèmes aux développeurs, car il est nécessaire de s'assurer que les macros sont uniques dans tous les en-têtes utilisés dans un projet. Plus précisément, si deux en-têtes (ou plus) utilisent `FOO_H_INCLUDED` comme garde d'inclusion, le premier de ces en-têtes inclus dans une unité de compilation empêchera effectivement l'inclusion des autres. Des défis particuliers sont présentés si un projet utilise un certain nombre de bibliothèques tierces avec des fichiers d'en-tête qui utilisent des gardes en commun.

Il est également nécessaire de s’assurer que les macros utilisées dans les protections d’inclusion n’entrent pas en conflit avec d’autres macros définies dans les fichiers d’en-tête.

La plupart des implémentations C ++ prennent également en charge la directive `#pragma once` qui garantit que le fichier n'est inclus qu'une seule fois dans une seule compilation. C'est une directive standard *de facto* , mais elle ne fait partie d'aucune norme ISO C ++. Par exemple:

```
// Foo.h
#pragma once

class Foo
{
};
```

Alors que `#pragma once` évite certains problèmes associés aux gardes d'inclusion, un `#pragma` - par définition dans les standards - est intrinsèquement un hook spécifique au compilateur, et sera ignoré par les compilateurs qui ne le supportent pas. Les projets qui utilisent `#pragma once` sont plus difficiles à porter sur des compilateurs qui ne le prennent pas en charge.

Un certain nombre de directives de codage et de normes d'assurance pour C ++ découragent spécifiquement toute utilisation du préprocesseur autre que les fichiers d'en-tête `#include` ou pour les placer dans les en-têtes.

Logique conditionnelle et gestion multi-plateforme

En bref, la logique de pré-traitement conditionnel consiste à rendre la logique de code disponible ou non disponible pour la compilation à l'aide de définitions de macro.

Trois cas d'utilisation importants sont:

- différents **profils d'application** (par exemple, débogage, publication, test, optimisé) pouvant être candidats de la même application (par exemple, avec une journalisation supplémentaire).
- **compilations multi-plateformes** - base de code unique, plusieurs plates-formes de

compilation.
- utiliser une base de code commune pour plusieurs **versions d'application** (par exemple, **versions** Basic, Premium et Pro d'un logiciel) - avec des fonctionnalités légèrement différentes.

Exemple a: Une approche multi-plateforme pour la suppression de fichiers (illustrative):

```
#ifdef _WIN32
#include <windows.h> // and other windows system files
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
  return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
  return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
  //TODO: check if NSAPI has a more specific function with permission dialog
  return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Des macros comme `_WIN32` , `__APPLE__` ou `__unix__` sont normalement prédéfinies par les implémentations correspondantes.

Exemple b: Activation de la journalisation supplémentaire pour une génération de débogage:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "--------BEGIN-DUMP---------------\n"
              << AppState::Instance()->Settings().ToString() << "\n"
#if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
              << ListToString(AppState::UndoStack()->GetActionNames())
              << AppState::Instance()->CrntDocument().Name()
              << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
#endif
              << "--------END-DUMP---------------\n"
}
```

Exemple c: Activer une fonctionnalité premium dans une version de produit distincte (remarque: ceci est illustratif. Il est souvent préférable d'autoriser le déverrouillage d'une fonctionnalité sans avoir à réinstaller une application)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium
users. Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
```

```
}
```

Quelques astuces communes:

Définition des symboles au moment de l'invocation:

Le préprocesseur peut être appelé avec des symboles prédéfinis (avec une initialisation facultative). Par exemple cette commande (`gcc -E` exécute uniquement le préprocesseur)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

traite Sample.cpp de la même manière que si `#define OPTIMISE_FOR_OS_X` et `#define TESTING_MODE 1` étaient ajoutés au début de Sample.cpp.

S'assurer qu'une macro est définie:

Si une macro n'est pas définie et que sa valeur est comparée ou vérifiée, le préprocesseur assume presque toujours en silence la valeur `0` . Il y a plusieurs façons de travailler avec cela. Une approche consiste à supposer que les paramètres par défaut sont représentés par 0 et que toute modification (par exemple, le profil de génération de l'application) doit être explicitement effectuée (par exemple ENABLE_EXTRA_DEBUGGING = 0 par défaut, définissez -DENABLE_EXTRA_DEBUGGING = 1). Une autre approche consiste à rendre toutes les définitions et les valeurs par défaut explicites. Cela peut être réalisé en utilisant une combinaison de directives `#ifndef` et `#error` :

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#    error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#    if ( 1 == ENABLE_EXTRA_DEBUGGING )
  //code
#    endif
#endif
```

Macros

Les macros sont classées en deux groupes principaux: les macros de type objet et les macros de type fonction. Les macros sont traitées comme une substitution de jeton au début du processus de compilation. Cela signifie que des sections de code volumineuses (ou répétitives) peuvent être extraites dans une macro de préprocesseur.

```
// This is an object-like macro
#define    PI         3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesnt know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define    AREA(r)    (PI*(r)*(r))
```

```
// They can be used like this:
double pi_macro   = PI;
double area_macro = AREA(4.6);
```

La bibliothèque Qt utilise cette technique pour créer un système de méta-objets en demandant à l'utilisateur de déclarer la macro Q_OBJECT en tête de la classe définie par l'utilisateur qui étend QObject.

Les noms de macro sont généralement écrits en majuscules pour les différencier plus facilement du code normal. Ce n'est pas une exigence, mais est simplement considéré comme un bon style par de nombreux programmeurs.

Lorsqu'une macro de type objet est rencontrée, elle est développée sous la forme d'une simple opération de copier-coller, le nom de la macro étant remplacé par sa définition. Lorsqu'une macro de type fonction est rencontrée, son nom et ses paramètres sont développés.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
// Compiler sees:
double area = (3.14159265358979*(5)*(5))
```

De ce fait, les paramètres de macro de type fonction sont souvent placés entre parenthèses, comme dans `AREA()` ci-dessus. Cela permet d'éviter tout bogue pouvant survenir lors de l'expansion d'une macro, en particulier les bogues causés par un paramètre de macro unique composé de plusieurs valeurs réelles.

```
#define BAD_AREA(r) PI * r * r

double bad_area = BAD_AREA(5 + 1.6);
// Compiler sees:
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;

double good_area = AREA(5 + 1.6);
// Compiler sees:
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

Notez également qu'en raison de cette simple extension, les paramètres transmis aux macros doivent être pris en compte pour éviter les effets secondaires inattendus. Si le paramètre est modifié pendant l'évaluation, il sera modifié à chaque fois qu'il est utilisé dans la macro développée, ce qui n'est généralement pas ce que nous voulons. Cela est vrai même si la macro renferme les paramètres entre parenthèses pour empêcher tout développement de la part de l'extension.

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

En outre, les macros ne fournissent aucune sécurité de type, ce qui entraîne des erreurs difficiles à comprendre concernant la non-concordance de type.

Comme les programmeurs terminent normalement les lignes avec un point-virgule, les macros destinées à être utilisées comme lignes autonomes sont souvent conçues pour "avaler" un point-virgule; Cela évite que des bogues supplémentaires ne soient causés par un point-virgule supplémentaire.

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

Dans cet exemple, le double point-virgule involontaire rompt le bloc `if...else` , empêchant le compilateur de faire correspondre le `else` au `if` . Pour éviter cela, le point-virgule est omis de la définition de la macro, ce qui le fera "avaler" le point-virgule immédiatement après son utilisation.

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
    std::cout << "Hooray!  I work again!" << std::endl;
```

Laisser le point-virgule final permet également d'utiliser la macro sans mettre fin à l'instruction en cours, ce qui peut être bénéfique.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)

// ...

some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Normalement, une définition de macro se termine à la fin de la ligne. Si une macro doit couvrir plusieurs lignes, une barre oblique inverse peut être utilisée à la fin d'une ligne pour indiquer cela. Cette barre oblique inverse doit être le dernier caractère de la ligne, ce qui indique au préprocesseur que la ligne suivante doit être concaténée sur la ligne en cours, en les traitant comme une seule ligne. Cela peut être utilisé plusieurs fois de suite.

```
#define TEXT "I \
am \
many \
lines."

// ...

std::cout << TEXT << std::endl; // Output:   I am many lines.
```

Ceci est particulièrement utile dans les macros complexes de type fonction, qui peuvent nécessiter plusieurs lignes.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

Dans le cas de macros de type fonction plus complexes, il peut être utile de leur donner leur propre champ d'application pour empêcher les collisions de noms possibles ou pour provoquer la destruction d'objets à la fin de la macro, comme une fonction réelle. Un idiome commun pour cela est *do tout en 0* , où la macro est entourée d'un bloc *do-while* . Ce bloc n'est généralement *pas* suivi d'un point-virgule, ce qui lui permet d'avaler un point-virgule.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

Il existe également des macros variadiques; Tout comme les fonctions variadiques, elles prennent un nombre variable d'arguments, puis les développent toutes à la place d'un paramètre spécial "Varargs", `__VA_ARGS__` .

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(printf, "%d", 8);
// Compiler sees:
printf("%d", 8);
```

Notez que lors de l'expansion, `__VA_ARGS__` peut être placé n'importe où dans la définition et sera développé correctement.

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)

VARIADIC2(some_func, 3, 8, 6, 9);
// Compiler sees:
some_func(8, 6, 9, 3);
```

Dans le cas d'un paramètre variadic à argument nul, différents compilateurs gèrent différemment la virgule finale. Certains compilateurs, tels que Visual Studio, avalent silencieusement la virgule sans aucune syntaxe particulière. D'autres compilateurs, tels que GCC, exigent que vous `__VA_ARGS__ ##` immédiatement avant `__VA_ARGS__` . Pour cette raison, il est judicieux de définir de manière conditionnelle les macros variadiques lorsque la portabilité est un problème.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if       COMPILER == "VS"
    #define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif     COMPILER == "GCC"
    #define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Messages d'erreur de préprocesseur

Des erreurs de compilation peuvent être générées à l'aide du préprocesseur. Ceci est utile pour un certain nombre de raisons dont certaines incluent la notification à un utilisateur s'il se trouve sur une plate-forme non prise en charge ou un compilateur non pris en charge.

ex. Erreur de retour si la version de gcc est 3.0.0 ou antérieure.

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

par exemple, erreur de retour si compilation sur un ordinateur Apple.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

Macros prédéfinies

Les macros prédéfinies sont celles que le compilateur définit (contrairement à ce que l'utilisateur définit dans le fichier source). Ces macros ne doivent pas être redéfinies ou indéfinies par l'utilisateur.

Les macros suivantes sont prédéfinies par le standard C ++:

- `__LINE__` contient le numéro de ligne de la ligne sur laquelle cette macro est utilisée et peut être modifié par la directive `#line` .
- `__FILE__` contient le nom du fichier dans `__FILE__` cette macro est utilisée et peut être modifié par la directive `#line` .
- `__DATE__` contient la date (au format `"Mmm dd yyyy"`) de la compilation du fichier, où *Mmm* est formaté comme s'il était obtenu par un appel à `std::asctime()` .
- `__TIME__` contient l'heure (au format `"hh:mm:ss"`) de la compilation du fichier.
- `__cplusplus` est défini par des compilateurs C ++ (conformes) lors de la compilation de fichiers C ++. Sa valeur est la version standard avec laquelle le compilateur est **entièrement**

conforme, à savoir `199711L` pour C ++ 98 et C ++ 03, `201103L` pour C ++ 11 et `201402L` pour le standard C ++ 14.

c ++ 11

- `__STDC_HOSTED__` est défini sur `1` si l'implémentation est *hébergée* ou `0` si elle est *autonome* .

c ++ 17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` contient un littéral `size_t` , c'est-à-dire l'alignement utilisé pour un appel à un `operator new` alignement-ignorant.

De plus, les macros suivantes peuvent être prédéfinies par les implémentations et peuvent ou non être présentes:

- `__STDC__` a une signification dépendant de l'implémentation et est généralement définie uniquement lors de la compilation d'un fichier en C, pour indiquer la conformité totale au standard C. (Ou jamais, si le compilateur décide de ne pas supporter cette macro.)

c ++ 11

- `__STDC_VERSION__` a une signification dépendant de l'implémentation, et sa valeur est généralement la version C, de la même manière que `__cplusplus` est la version C ++. (Ou n'est même pas défini, si le compilateur décide de ne pas supporter cette macro.)
- `__STDC_MB_MIGHT_NEQ_WC__` est défini sur `1` , si les valeurs du codage étroit du jeu de caractères de base peuvent ne pas être égales aux valeurs de leurs homologues larges (par exemple if `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `__STDC_ISO_10646__` est défini si `wchar_t` est codé en Unicode et se développe en une constante entière sous la forme `yyyymmL` , indiquant la dernière révision Unicode prise en charge.
- `__STDCPP_STRICT_POINTER_SAFETY__` est défini sur `1` si l'implémentation a *une sécurité de pointeur stricte* (sinon, la *sécurité du pointeur* est *relâchée*)
- `__STDCPP_THREADS__` est défini sur `1` , si le programme peut avoir plusieurs threads d'exécution (applicable aux *implémentations autonomes* - les *implémentations hébergées* peuvent toujours avoir plusieurs threads)

Il convient également de mentionner `__func__` , qui n'est pas une macro, mais une variable fonction-locale prédéfinie. Il contient le nom de la fonction dans laquelle il est utilisé, en tant que tableau de caractères statiques dans un format défini par l'implémentation.

En plus de ces macros prédéfinies standard, les compilateurs peuvent avoir leur propre ensemble de macros prédéfinies. Il faut se référer à la documentation du compilateur pour les apprendre. Par exemple:

- gcc
- Microsoft Visual C ++
- bruit
- Compilateur Intel C ++

Certaines des macros sont juste pour interroger le support de certaines fonctionnalités:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
   // C library header declarations here
}
#endif
```

D'autres sont très utiles pour le débogage:

c ++ 11

```
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
              << " in function " << __func__ << "()"
              << " in file " << __FILE__
              << std::endl;
}
```

Et d'autres pour le contrôle de version trivial:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
                  << "v 1.1\n" // I have to remember to update this manually
                  << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
                  << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

X-macros

Une technique idiomatique pour générer des structures de code répétitives au moment de la compilation.

Une macro X se compose de deux parties: la liste et l'exécution de la liste.

Exemple:

```
#define LIST \
    X(dog)   \
    X(cat)   \
    X(racoon)

// class Animal {
//  public:
//    void say();
// };

#define X(name) Animal name;
LIST
```

```
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}
```

qui est développé par le préprocesseur dans les éléments suivants:

```
Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}
```

Comme les listes deviennent plus grandes (disons plus de 100 éléments), cette technique aide à éviter un copier-coller excessif.

Source: https://en.wikipedia.org/wiki/X_Macro

Voir aussi: Macros X

Si définir un `X` peu pertinent avant d'utiliser `LIST` n'est pas à votre goût, vous pouvez également passer un nom de macro en argument:

```
#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)
```

Maintenant, vous spécifiez explicitement quelle macro doit être utilisée lors du développement de la liste, par exemple

```
#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)
```

Si chaque appel de la `MACRO` doit prendre des paramètres supplémentaires - constants par rapport à la liste, des macros variadic peuvent être utilisées

```
//a walkaround for Visual studio
#define EXPAND(x) x

#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
```

```
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(racoon, __VA_ARGS__))
```

Le premier argument est fourni par la `LIST` , tandis que le reste est fourni par l'utilisateur dans l'appel de `LIST` . Par exemple:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

étendra à

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

#pragma une fois

La plupart des implémentations C ++, mais pas toutes, prennent en charge la directive `#pragma once` qui garantit que le fichier n'est inclus qu'une seule fois dans une seule compilation. Il ne fait partie d'aucune norme ISO C ++. Par exemple:

```
// Foo.h
#pragma once

class Foo
{
};
```

Alors que `#pragma once` évite certains problèmes associés aux gardes d'inclusion , un `#pragma` - par définition dans les standards - est intrinsèquement un hook spécifique au compilateur, et sera ignoré par les compilateurs qui ne le supportent pas. Les projets qui utilisent `#pragma once` doivent être modifiés pour être conformes aux normes.

Avec certains compilateurs, en particulier ceux qui utilisent des en - têtes précompilés , `#pragma once` peut, `#pragma once` accélérer considérablement le processus de compilation. De même, certains préprocesseurs accélèrent la compilation en suivant les en-têtes utilisés, notamment les gardes. L'avantage net, lorsque les deux `#pragma once` sont utilisées, dépend de l'implémentation et peut être une augmentation ou une diminution des temps de compilation.

`#pragma once` combiné avec les inclusions a été la mise en page recommandée pour les fichiers d' en- tête lors de l' écriture des applications MFC sur les fenêtres, et a été générée par de Visual Studio `add class` , `add dialog` , `add windows` des `add windows` assistants. Il est donc très courant de les trouver combinées dans les applications Windows C ++.

Opérateurs de préprocesseur

`#` opérateur ou opérateur de chaîne est utilisé pour convertir un paramètre de macro en un littéral de chaîne. Il ne peut être utilisé qu'avec les macros ayant des arguments.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")

PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor""\n");
```

Le compilateur concatène deux chaînes et l'argument final `printf()` sera un littéral de chaîne avec un caractère de nouvelle ligne à sa fin.

Le préprocesseur ignore les espaces avant ou après l'argument de macro. Donc, la déclaration imprimée ci-dessous nous donnera le même résultat.

```
PRINT(   This line will be converted to string by preprocessor );
```

Si le paramètre du littéral de chaîne nécessite une séquence d'échappement comme avant un guillemet double (), il sera automatiquement inséré par le préprocesseur.

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor""\n");
```

`##` operator ou Token pasting operator permet de concaténer deux paramètres ou jetons d'une macro.

```
// preprocessor will combine the variable and the x
#define PRINT(x) printf("variable" #x " = %d", variable##x)

int variableY = 15;
PRINT(Y);
//compiler sees
printf("variable""Y"" = %d", variableY);
```

et le résultat final sera

```
variableY = 15
```

Lire Préprocesseur en ligne: https://riptutorial.com/fr/cplusplus/topic/1098/preprocesseur

Chapitre 95: priorité de l'opérateur

Remarques

Les opérateurs sont listés de haut en bas, en ordre décroissant. Les opérateurs avec le même numéro ont la même priorité et la même associativité.

1. `::`
2. Les opérateurs postfixés: `[]` `()` `T(...)` `.` `->` `++` `--` `dynamic_cast` `static_cast` `reinterpret_cast` `const_cast` `typeid`
3. Les opérateurs de préfixe unaire: `++` `--` `*` `&` `+` `-` `!` `~` `sizeof` `new` `delete` `delete[]` ; la notation de style C, `(T)...` ; (C ++ 11 et supérieur) `sizeof...` `alignof` `noexcept`
4. `.*` et `->*`
5. `*` , `/` et `%` , opérateurs arithmétiques binaires
6. `+` et `-` , opérateurs arithmétiques binaires
7. `<<` et `>>`
8. `<` , `>` , `<=` , `>=`
9. `==` et `!=`
10. `&` , l'opérateur ET binaire
11. `^`
12. `|`
13. `&&`
14. `||`
15. `?:` (opérateur conditionnel ternaire)
16. `=` , `*=` , `/=` , `%=` , `+=` , `-=` , `>>=` , `<<=` , `&=` , `^=` , `|=`
17. `throw`
18. `,` (l'opérateur de virgule)

L'attribution, l'assignation composée et les opérateurs conditionnels ternaires sont associatifs à droite. Tous les autres opérateurs binaires sont associatifs à gauche.

Les règles pour l'opérateur conditionnel ternaire sont un peu plus compliquées que les simples règles de priorité peuvent exprimer.

- Un opérande se lie moins à un `?` à sa gauche ou a `:` à sa droite que pour tout autre opérateur. Effectivement, le deuxième opérande de l'opérateur conditionnel est analysé comme s'il était entre parenthèses. Cela permet une expression telle que `a ? b , c : d` être syntaxiquement valide.
- Un opérande se lie plus étroitement à un `?` sur son droit qu'à un opérateur d'affectation ou `throw` sur sa gauche, donc `a = b ? c : d` est équivalent à `a = (b ? c : d)` et `throw a ? b : c` équivaut à `throw (a ? b : c)` .
- Un opérande se lie plus étroitement à un opérateur d'affectation à sa droite qu'à `:` à sa gauche, donc `a ? b : c = d` est équivalent à `a ? b : (c = d)` .

Examples

Opérateurs arithmétiques

Les opérateurs arithmétiques en C ++ ont la même priorité qu'en mathématiques:

La multiplication et la division ont laissé une associativité (signifiant qu'elles seront évaluées de gauche à droite) et elles ont une priorité plus élevée que l'addition et la soustraction, qui ont également une associativité.

On peut aussi forcer la préséance de l'expression en utilisant des parenthèses `( )`. Tout comme vous le feriez en mathématiques normales.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Opérateurs logiques ET et OU

Ces opérateurs ont la priorité habituelle en C ++: AND avant OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

Ce code est équivalent à ce qui suit:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

L'ajout de parenthèses ne modifie pas le comportement, cependant, il facilite la lecture. En ajoutant ces parenthèses, il n'y a aucune confusion quant à l'intention de l'auteur.

Logique && et || opérateurs: court-circuit

&& a préséance sur ||, cela signifie que les parenthèses sont placées pour évaluer ce qui serait évalué ensemble.

c ++ utilise l'évaluation des courts-circuits dans && et || ne pas faire d'exécutions inutiles. Si le côté gauche de || renvoie vrai le côté droit n'a plus besoin d'être évalué.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :=====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
}
```

Opérateurs Unaires

Les opérateurs unaires agissent sur l'objet sur lequel ils sont appelés et ont la priorité. (Voir

remarques)

Lorsqu'elle est utilisée postfixe, l'action se produit uniquement après l'évaluation de toute l'opération, ce qui conduit à des calculs arithmétiques intéressants:

```
int a = 1;
++a;            // result: 2
a--;            // result: 1
int minusa=-a;  // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Lire priorité de l'opérateur en ligne: https://riptutorial.com/fr/cplusplus/topic/3895/priorite-de-l-operateur

Chapitre 96: Profilage

Examples

Profilage avec gcc et gprof

Le profileur GNU gprof, gprof , vous permet de profiler votre code. Pour l'utiliser, vous devez effectuer les étapes suivantes:

1. Construire l'application avec des paramètres pour générer des informations de profilage
2. Générer des informations de profilage en exécutant l'application intégrée
3. Afficher les informations de profilage générées avec gprof

Afin de créer l'application avec des paramètres pour générer des informations de profilage, nous ajoutons l'indicateur `-pg` . Ainsi, par exemple, nous pourrions utiliser

```
$ gcc -pg *.cpp -o app
```

ou

```
$ gcc -O2 -pg *.cpp -o app
```

et ainsi de suite.

Une fois l'application, disons `app` , est construite, exécutez-la comme d'habitude:

```
$ ./app
```

Cela devrait produire un fichier appelé `gmon.out` .

Pour voir les résultats du profilage, exécutez maintenant

```
$ gprof app gmon.out
```

(notez que nous fournissons à la fois l'application et la sortie générée).

Bien sûr, vous pouvez également canaliser ou rediriger:

```
$ gprof app gmon.out | less
```

et ainsi de suite.

Le résultat de la dernière commande doit être une table dont les lignes sont les fonctions et dont les colonnes indiquent le nombre d'appels, le temps total passé, le temps passé (c'est-à-dire le

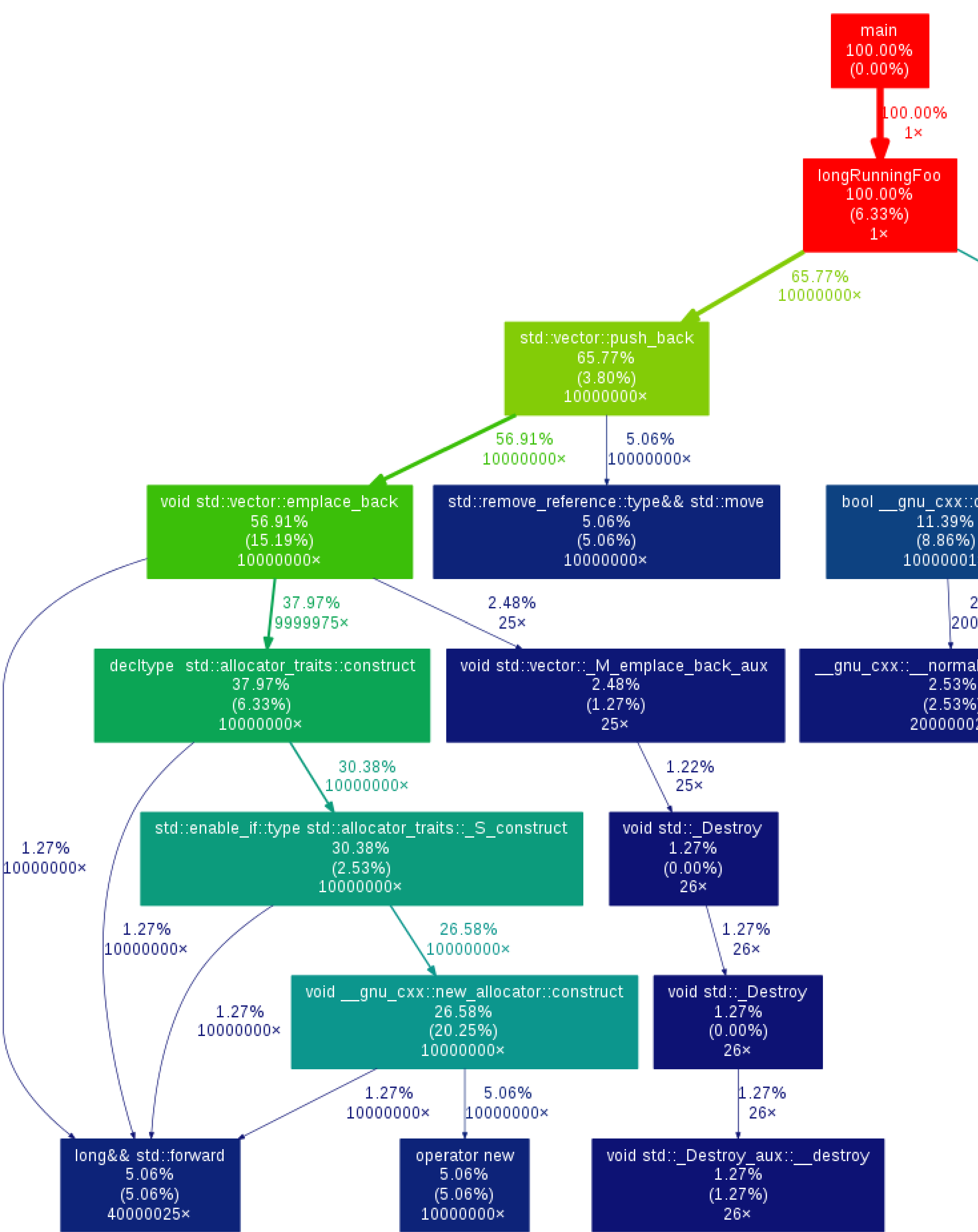
temps passé hors fonction des enfants).

Générer des diagrammes callgraph avec gperf2dot

Pour les applications plus complexes, les profils d'exécution à plat peuvent être difficiles à suivre. C'est pourquoi de nombreux outils de profilage génèrent également une forme d'information annotée sur le callgraph.

gperf2dot convertit la sortie texte de nombreux profileurs (perf Linux, callgrind, oprofile, etc.) en un diagramme callgraph. Vous pouvez l'utiliser en exécutant votre profileur (exemple pour `gprof`):

```
# compile with profiling flags
g++ *.cpp -pg
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



Profilage de l'utilisation du processeur avec les outils gcc et Google Perf

Google Perf Tools fournit également un profileur de processeur, avec une interface légèrement plus conviviale. Pour l'utiliser:

1. Installer les outils Google Perf
2. Compilez votre code comme d'habitude
3. Ajoutez la bibliothèque du profileur `libprofiler` au chemin de chargement de votre bibliothèque au moment de l'exécution
4. Utiliser `pprof` pour générer un profil d'exécution à plat ou un diagramme de callgraph

Par exemple:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000
./main
```

où:

- `CPUPROFILE` indique le fichier de sortie pour les données de profilage
- `CPUPROFILE_FREQUENCY` indique la fréquence d'échantillonnage du profileur;

Utilisez `pprof` pour post-traiter les données de profilage.

Vous pouvez générer un profil d'appel plat sous forme de texte:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text --lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
      22  32.8%  32.8%       67 100.0% longRunningFoo ??:0
      20  29.9%  62.7%       20  29.9% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
       4   6.0%  68.7%        4   6.0% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
       3   4.5%  73.1%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:388
       3   4.5%  77.6%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:401
       2   3.0%  80.6%        2   3.0% __munmap /build/eglibc-3GlaMS/eglibc-
2.19/misc/../sysdeps/unix/syscall-template.S:81
       2   3.0%  83.6%       12  17.9% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:298
       2   3.0%  86.6%        2   3.0% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:385
       2   3.0%  89.6%        2   3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:26
       1   1.5%  91.0%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
       1   1.5%  92.5%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
       1   1.5%  94.0%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:293
       1   1.5%  95.5%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
```

```
2.19/stdlib/random.c:296
       1   1.5%  97.0%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:371
       1   1.5%  98.5%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:381
       1   1.5% 100.0%        1   1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:28
       0   0.0% 100.0%       67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-
2.19/csu/libc-start.c:287
       0   0.0% 100.0%       67 100.0% _start ??:0
       0   0.0% 100.0%       67 100.0% main ??:0
       0   0.0% 100.0%       14  20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:27
       0   0.0% 100.0%       27  40.3% std::vector::_M_emplace_back_aux ??:0
```

... ou vous pouvez générer un callgraph annoté dans un pdf avec:

```
pprof --pdf ./main main.prof > out.pdf
```

Lire Profilage en ligne: https://riptutorial.com/fr/cplusplus/topic/5347/profilage

Chapitre 97: RAII: l'acquisition de ressources est une initialisation

Remarques

RAII signifie **R** eSource **A** cquisition **I** s nitialization **I.** Aussi appelé occasionnellement SBRM (Scope-Based Resource Management) ou RRID (Resource Release Is Destruction), RAII est un langage utilisé pour lier les ressources à la durée de vie de l'objet. En C ++, le destructeur d'un objet s'exécute toujours lorsqu'un objet est hors de portée. Nous pouvons en tirer parti pour lier le nettoyage des ressources à la destruction des objets.

Chaque fois que vous devez acquérir une ressource (par exemple, un verrou, un descripteur de fichier, un tampon alloué) que vous devrez éventuellement libérer, vous devriez envisager d'utiliser un objet pour gérer cette gestion des ressources pour vous. Le déroulement de la pile se produira indépendamment de l'exception ou de la sortie anticipée de la portée. Ainsi, l'objet gestionnaire de ressources nettoiera la ressource pour vous sans que vous ayez à examiner attentivement tous les chemins de code actuels et futurs possibles.

Il convient de noter que RAII ne libère pas complètement le développeur de la durée de vie des ressources. Un cas est évidemment un appel crash ou exit (), qui empêchera les destructeurs d'être appelés. Étant donné que le système d'exploitation va nettoyer les ressources locales telles que la mémoire après la fin d'un processus, ce n'est pas un problème dans la plupart des cas. Cependant, avec les ressources système (c.-à-d. Les canaux nommés, les fichiers de verrouillage, la mémoire partagée), vous avez toujours besoin d'installations pour traiter un processus qui ne s'est pas nettoyé. vérifiez que le processus avec le pid existe réellement, puis agissez en conséquence.

Une autre situation est celle où un processus Unix appelle une fonction de la famille exec, c'est-à-dire après un fork-exec pour créer un nouveau processus. Ici, le processus fils aura une copie complète de la mémoire des parents (y compris les objets RAII), mais une fois qu’il aura été appelé, aucun des destructeurs ne sera appelé dans ce processus. D'un autre côté, si un processus est forké et qu'aucun des processus n'appelle exec, toutes les ressources sont nettoyées dans les deux processus. Ceci est correct uniquement pour toutes les ressources qui ont été réellement dupliquées dans fork, mais avec les ressources système, les deux processus auront uniquement une référence à la ressource (c.-à-d. Le chemin vers un fichier de verrouillage) et tenteront tous deux l'autre processus à échouer.

Examples

Verrouillage

Mauvais verrouillage:

```
std::mutex mtx;
```

```
void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock();   // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock();       // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock();       // Must also force unlock in the presence of
        throw;              // exceptions and allow the exception to continue.
    }
}
```

C'est la mauvaise façon d'implémenter le verrouillage et le déverrouillage du mutex. Pour garantir la publication correcte du mutex avec `unlock()` le programmeur doit s'assurer que tous les flux entraînant la sortie de la fonction entraînent un appel à `unlock()` . Comme indiqué ci-dessus, il s'agit d'un processus fragile car les responsables doivent continuer à suivre le modèle manuellement.

En utilisant une classe spécialement conçue pour implémenter RAII, le problème est trivial:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx);   // constructor locks.
                                           // destructor unlocks. destructor call
                                           // guaranteed by language.
    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

`lock_guard` est un modèle de classe extrêmement simple qui appelle simplement `lock()` sur son argument dans son constructeur, conserve une référence à l'argument et appelle `unlock()` sur l'argument de son destructeur. C'est-à-dire que lorsque le `lock_guard` est hors de portée, le `mutex` est garanti pour être déverrouillé. Peu importe si la raison pour laquelle elle est hors de portée est une exception ou un retour anticipé - tous les cas sont traités; quel que soit le flux de contrôle, nous avons garanti que nous allons débloquer correctement.

Enfin / ScopeExit

Pour les cas où nous ne voulons pas écrire de classes spéciales pour gérer une ressource, nous pouvons écrire une classe générique:

```
template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;
private:
    Function f;
};
// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)};
}
```

Et son exemple d'utilisation

```
void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&](){ v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}
```

Remarque (1): Certaines discussions sur la définition du destructeur doivent être prises en compte pour gérer les exceptions:

- `~Finally() noexcept { f(); }` : `std::terminate` est appelé en cas d'exception
- `~Finally() noexcept(noexcept(f())) { f(); }` : terminate () est appelé uniquement en cas d'exception lors du déroulement de la pile.
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */} }` Aucun `std::terminate` n'a été appelé, mais nous ne pouvons pas gérer les erreurs (même pour le déroulement de la pile).

ScopeSuccess (c ++ 17)

C ++ 17

Grâce à `int std::uncaught_exceptions()` , nous pouvons implémenter une action qui ne s'exécute qu'en cas de succès (aucune exception renvoyée dans la portée). Auparavant `bool std::uncaught_exception()` permet juste de détecter si **un** déroulement de la pile est en cours d' exécution.

```
#include <exception>
#include <iostream>

template <typename F>
```

```
class ScopeSuccess
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Sortie:

```
Success 1
```

ScopeFail (c ++ 17)

C ++ 17

Grâce à `int std::uncaught_exceptions()` , nous pouvons implémenter une action qui ne s'exécute

qu'en cas d'échec (exception levée dans la portée). Auparavant `bool std::uncaught_exception()` permet juste de détecter si **un** déroulement de la pile est en cours d' exécution.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Sortie:

```
Failure 2
```

Lire RAII: l'acquisition de ressources est une initialisation en ligne:
https://riptutorial.com/fr/cplusplus/topic/1320/raii--l-acquisition-de-ressources-est-une-initialisation

Chapitre 98: Recherche de nom dépendante de l'argument

Examples

Quelles fonctions sont trouvées

Les fonctions se trouvent d'abord en collectant un ensemble de *"classes associées"* et *"espaces de noms associés"* qui incluent un ou plusieurs des éléments suivants, en fonction du type d'argument `T` Tout d'abord, montrons les règles pour les noms de classes, d'énumération et de template de classe.

- Si `T` est une classe imbriquée, l'énumération des membres, puis la classe qui l'entoure.
- Si `T` est une énumération (il peut *aussi* s'agir d'un membre de classe!), Son espace de noms le plus interne.
- Si `T` est une classe (il peut *aussi* être imbriqué!), Toutes ses classes de base et la classe elle-même. L'espace de noms le plus interne de toutes les classes associées.
- Si `T` est un `ClassTemplate<TemplateArguments>` (c'est *aussi* une classe!), Les classes et espaces de noms associés aux arguments de type modèle, l'espace de noms de tout argument de modèle de modèle et la classe environnante de tout argument de modèle, un modèle de membre.

Il existe maintenant quelques règles pour les types intégrés.

- Si `T` est un pointeur sur `U` ou tableau de `U` , les classes et espaces de noms associés à `U` Exemple: `void (*fptr)(A); f(fptr);` , inclut les espaces de noms et les classes associés à `void(A)` (voir la règle suivante).
- Si `T` est un type de fonction, les classes et espaces de noms associés aux types de paramètre et de retour. Exemple: `void(A)` comprend les espaces de noms et les classes associés à `A`
- Si `T` est un pointeur sur un membre, les classes et les espaces de noms associés au type de membre (peuvent s'appliquer à la fois aux fonctions de pointeur et à celles de membre de données!). Exemple: `BA::*p; void (A::*pf)(B); f(p); f(pf);` inclut les espaces de noms et les classes associés à `A` , `B` , `void(B)` (qui applique la puce ci-dessus pour les types de fonctions).

Toutes les fonctions et tous les modèles de tous les espaces de noms associés sont détectés par une recherche dépendante des arguments. De plus, *on trouve des fonctions d'ami d'espace de nommage et d'étendue déclarées dans les classes associées* , qui ne sont normalement pas visibles. L'utilisation des directives est cependant ignorée.

Tous les exemples d'appels suivants sont valides, sans qualifier `f` par le nom de l'espace de nom dans l'appel.

```
namespace A {
```

```
    struct Z { };
    namespace I { void g(Z); }
    using namespace I;

    struct X { struct Y { }; friend void f(Y) { } };
    void f(X p) { }
    void f(std::shared_ptr<X> p) { }
}

// example calls
f(A::X());
f(A::X::Y());
f(std::make_shared<A::X>());

g(A::Z()); // invalid: "using namespace I;" is ignored!
```

Lire Recherche de nom dépendante de l'argument en ligne: https://riptutorial.com/fr/cplusplus/topic/5163/recherche-de-nom-dependante-de-l-argument

Chapitre 99: Récursivité en C ++

Examples

Utilisation de la récursion de la queue et de la récursion de style Fibonnaci pour résoudre la séquence Fibonnaci

La manière la plus simple et la plus évidente d'utiliser la récursivité pour obtenir le Nième terme de la séquence Fibonnaci est la suivante:

```
int get_term_fib(int n)
{
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return get_term_fib(n - 1) + get_term_fib(n - 2);
}
```

Cependant, cet algorithme ne s'adapte pas aux termes plus élevés: pour les `n` plus grands, le nombre d'appels de fonctions à effectuer augmente de manière exponentielle. Cela peut être remplacé par une simple récursion de la queue.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
  if (n == 0)
    return prev;
  if (n == 1)
    return curr;
  return get_term_fib(n - 1, curr, prev + curr);
}
```

Chaque appel à la fonction calcule maintenant immédiatement le terme suivant dans la séquence Fibonnaci, de sorte que le nombre d'appels de fonction évolue linéairement avec `n` .

Récursivité avec mémo

Les fonctions récursives peuvent devenir assez coûteuses. Si ce sont des fonctions pures (fonctions qui renvoient toujours la même valeur lorsqu'elles sont appelées avec les mêmes arguments et qui ne dépendent ni ne modifient l'état externe), elles peuvent être considérablement accélérées au détriment de la mémoire en stockant les valeurs déjà calculées.

Ce qui suit est une implémentation de la séquence de Fibonacci avec mémo:

```
#include <map>

int fibonacci(int n)
{
  static std::map<int, int> values;
```

```
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Notez que malgré l'utilisation de la formule de récurrence simple, cette fonction est $ O (n) $ au premier appel. Sur les appels suivants avec la même valeur, c'est bien sûr $ O (1) $.

Notez cependant que cette implémentation n'est pas réentrante. En outre, il ne permet pas de se débarrasser des valeurs stockées. Une autre implémentation serait de permettre à la carte d'être transmise en tant qu'argument supplémentaire:

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Pour cette version, l'appelant doit conserver la carte avec les valeurs stockées. Cela a l'avantage que la fonction est maintenant réentrante et que l'appelant peut supprimer des valeurs inutiles, économisant ainsi de la mémoire. Il présente l'inconvénient de rompre l'encapsulation; l'appelant peut modifier la sortie en renseignant la carte avec des valeurs incorrectes.

Lire Récursivité en C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/5693/recursivite-en-c-plusplus

Chapitre 100: Renvoyer plusieurs valeurs d'une fonction

Introduction

Il existe de nombreuses situations où il est utile de renvoyer plusieurs valeurs d'une fonction: par exemple, si vous souhaitez saisir un article et renvoyer le prix et le nombre en stock, cette fonctionnalité pourrait être utile. Il y a plusieurs façons de faire cela en C ++, et la plupart impliquent la STL. Cependant, si vous souhaitez éviter la STL pour une raison quelconque, il existe encore plusieurs façons de procéder, y compris les `structs/classes` et les `arrays` .

Examples

Utilisation des paramètres de sortie

Les paramètres peuvent être utilisés pour renvoyer une ou plusieurs valeurs; ces paramètres doivent être des pointeurs ou des références non `const` .

Les références:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
    e = a * b;
    f = a / b;
}
```

Pointeurs:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {
    *c = a + b;
    *d = a - b;
    *e = a * b;
    *f = a / b;
}
```

Certaines bibliothèques ou frameworks utilisent un paramètre 'OUT' vide `#define` pour que les paramètres de sortie de la signature de la fonction soient clairement indiqués. Cela n'a aucun impact fonctionnel et sera compilé, mais rend la signature de la fonction un peu plus claire;

```
#define OUT

void calculate(int a, int b, OUT int& c) {
    c = a + b;
}
```

Utiliser std :: tuple

C ++ 11

Le type `std::tuple` peut regrouper un nombre quelconque de valeurs, incluant potentiellement des valeurs de différents types, en un seul objet de retour:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
   return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

En C ++ 17, une liste d'initialisation contreventée peut être utilisée:

C ++ 17

```
std::tuple<int, int, int, int> foo(int a, int b)    {
    return {a + b, a - b, a * b, a / b};
}
```

La récupération des valeurs du `tuple` renvoyé peut être fastidieuse, nécessitant l'utilisation de la fonction `std::get` template:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Si les types peuvent être déclarés avant que la fonction ne retourne, alors `std::tie` peut être utilisé pour décompresser un `tuple` dans des variables existantes:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

Si l'une des valeurs renvoyées n'est pas nécessaire, `std::ignore` peut être utilisé:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

C ++ 17

Les liaisons structurées peuvent être utilisées pour éviter `std::tie` :

```
auto [add, sub, mul, div] = foo(5,12);
```

Si vous voulez retourner un tuple de références lvalue au lieu d'un tuple de valeurs, utilisez `std::tie` à la place de `std::make_tuple` .

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
  if (b<a)
    return std::tie(b,a);
```

```
  else
    return std::tie(a,b);
}
```

qui permet

```
void increase_least(int& a, int& b) {
  std::get<0>(minmax(a,b))++;
}
```

Dans certains cas rares, vous utiliserez `std::forward_as_tuple` au lieu de `std::tie` ; soyez prudent si vous le faites, car les délais peuvent ne pas durer assez longtemps pour être consommés.

Utiliser std :: array

C ++ 11

Le conteneur `std::array` peut regrouper un nombre fixe de valeurs de retour. Ce nombre doit être connu à la compilation et toutes les valeurs de retour doivent être du même type:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

Cela remplace les tableaux de style c de la forme `int bar[4]` . L'avantage étant que différentes fonctions std de `c++` peuvent désormais être utilisées. Il fournit également des fonctions utiles comme membres `at` ce qui est une fonction d'accès membre en sécurité avec le contrôle lié, et la `size` qui vous permet de retourner la taille du tableau sans calcul.

Utiliser std :: pair

Le struct template `std::pair` peut regrouper *exactement* deux valeurs de retour, de deux types quelconques:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

Avec C ++ 11 ou version ultérieure, une liste d'initialisation peut être utilisée à la place de `std::make_pair` :

C ++ 11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

Les valeurs individuelles de la `std::pair` retournée peuvent être récupérées en utilisant les `first` et

`second` objets membres de la paire:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Sortie:

> dix

En utilisant struct

Une `struct` peut être utilisée pour regrouper plusieurs valeurs de retour:

C ++ 11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

auto calc = foo(5, 12);
```

C ++ 11

Au lieu d'affecter des champs individuels, un constructeur peut être utilisé pour simplifier la construction des valeurs renvoyées:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
     return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

Les résultats individuels renvoyés par la fonction `foo()` peuvent être récupérés en accédant aux variables membres de la `struct calc` :

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Sortie:

17 -7 60 0

Remarque: Lors de l'utilisation d'une `struct` , les valeurs renvoyées sont regroupées dans un seul objet et accessibles à l'aide de noms significatifs. Cela permet également de réduire le nombre de variables externes créées dans la portée des valeurs renvoyées.

C ++ 17

Pour décompresser une `struct` renvoyée par une fonction, des liaisons structurées peuvent être utilisées. Cela place les paramètres de sortie sur un pied d'égalité avec les paramètres in:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

La sortie de ce code est identique à celle ci-dessus. La `struct` est toujours utilisée pour renvoyer les valeurs de la fonction. Cela vous permet de traiter les champs individuellement.

Fixations structurées

C ++ 17

C ++ 17 introduit des liaisons structurées, ce qui facilite encore la gestion de plusieurs types de retour, car vous n'avez pas besoin de vous appuyer sur `std::tie()` ni de décompresser manuellement un tuple:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

Les liaisons structurées peuvent être utilisées par défaut avec `std::pair` , `std::tuple` et tout type dont les membres de données non statiques sont tous des membres directs publics ou des membres d'une classe de base non ambiguë:

```
struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
```

```
auto& x = result.x;
auto& y = result.y;
```

Si vous faites votre type "tuple-like", il fonctionnera automatiquement avec votre type. Un tuple semblable est un type avec approprié `tuple_size` , `tuple_element` et s'écrit: `get`

```
namespace my_ns {
    struct my_type {
        int x;
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

maintenant cela fonctionne:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}
```

Utilisation d'un objet de consommation

Nous pouvons fournir à un consommateur qui sera appelé les multiples valeurs pertinentes:

C ++ 11

```
template <class F>
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});
```

C'est ce qu'on appelle le "style de passage continu" .

Vous pouvez adapter une fonction renvoyant un tuple dans une fonction de style de continuation via:

C ++ 17

```
template<class Tuple>
struct continuation {
  Tuple t;
  template<class F>
  decltype(auto) operator->*(F&& f)&&{
    return std::apply( std::forward<F>(f), std::move(t) );
  }
};
std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
  std::cout << "sum is " << sum << '\n';
};
```

les versions plus complexes étant accessibles en écriture en C ++ 14 ou C ++ 11.

Utiliser std :: vector

Un `std::vector` peut être utile pour renvoyer un nombre dynamique de variables du même type. L'exemple suivant utilise `int` comme type de données, mais un `std::vector` peut contenir n'importe quel type trivialement copiable:

```
#include <vector>
#include <iostream>

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
```

```
    std::vector<int> v = fillVectorFrom(1, 10);

    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Utilisation de l'itérateur de sortie

Plusieurs valeurs du même type peuvent être renvoyées en passant un itérateur de sortie à la fonction. Ceci est particulièrement courant pour les fonctions génériques (comme les algorithmes de la bibliothèque standard).

Exemple:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
}
```

Exemple d'utilisation:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Lire Renvoyer plusieurs valeurs d'une fonction en ligne:
https://riptutorial.com/fr/cplusplus/topic/487/renvoyer-plusieurs-valeurs-d-une-fonction

Chapitre 101: Résolution de surcharge

Remarques

La résolution de surcharge se produit dans plusieurs situations différentes

- Appels aux fonctions surchargées nommées. Les candidats sont toutes les fonctions trouvées par la recherche de nom.
- Appels à un objet de classe. Les candidats sont généralement tous les opérateurs d'appel de fonctions surchargés de la classe.
- Utilisation d'un opérateur Les candidats sont les fonctions d'opérateur surchargées dans la portée de l'espace de noms, les fonctions d'opérateur surchargées dans l'objet de classe de gauche (le cas échéant) et les opérateurs intégrés.
- Résolution de surcharge pour trouver la fonction ou le constructeur d'opérateur de conversion correct à appeler pour une initialisation
  - Pour l'initialisation directe sans liste (`Class c(value)`), les candidats sont des constructeurs de `Class` .
  - Pour une initialisation de copie sans liste (`Class c = value`) et pour rechercher la fonction de conversion définie par l'utilisateur à invoquer dans une séquence de conversion définie par l'utilisateur. Les candidats sont les constructeurs de `Class` et si la source est un objet de classe, ses fonctions d'opérateur de conversion.
  - Pour l'initialisation d'une non-classe à partir d'un objet de classe (`Nonclass c = classObject`). Les candidats sont les fonctions d'opérateur de conversion de l'objet d'initialisation.
  - Pour initialiser une référence avec un objet de classe (`R &r = classObject`), lorsque la classe a des fonctions d'opérateur de conversion qui fournissent des valeurs pouvant être directement liées à `r` . Les candidats sont des fonctions d'opérateur de conversion.
  - Pour l'initialisation de liste d'un objet de classe non agrégé (`Class c{1, 2, 3}`), les candidats sont les constructeurs de listes d'initialisation pour une première résolution de surcharge. Si cela ne permet pas de trouver un candidat viable, une résolution de surcharge est effectuée avec les constructeurs de `Class` comme candidats.

Examples

Correspondance exacte

Une surcharge sans conversions nécessaire pour les types de paramètres ou uniquement les conversions nécessaires entre les types qui sont toujours considérés comme des correspondances exactes est préférable à une surcharge qui nécessite d'autres conversions pour appeler.

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

Lorsqu'un argument est lié à une référence au même type, la correspondance est considérée comme ne nécessitant pas de conversion même si la référence est plus qualifiée cv.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

Aux fins de la résolution de surcharge, le type "tableau de `T` " est considéré comme correspondant exactement au type "pointeur sur `T` ", et le type de fonction `T` est considéré comme correspondant exactement au pointeur de fonction de type `T*` , même si les deux nécessitent conversions

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Catégorisation de l'argument au coût du paramètre

La résolution de surcharge partitionne le coût de la transmission d'un argument à un paramètre en quatre catégories différentes, appelées "séquences". Chaque séquence peut inclure zéro, une ou plusieurs conversions

- Séquence de conversion standard

  ```
  void f(int a); f(42);
  ```

- Séquence de conversion définie par l'utilisateur

  ```
  void f(std::string s); f("hello");
  ```

- Séquence de conversion des ellipses

  ```
  void f(...); f(42);
  ```

- Séquence d'initialisation de la liste

  ```
  void f(std::vector<int> v); f({1, 2, 3});
  ```

Le principe général est que les séquences de conversion standard sont les moins chères, suivies des séquences de conversion définies par l'utilisateur, suivies des séquences de conversion des

points de suspension.

Un cas particulier est la séquence d'initialisation de la liste, qui ne constitue pas une conversion (une liste d'initialisation n'est pas une expression avec un type). Son coût est déterminé en le définissant comme étant équivalent à l'une des trois autres séquences de conversion, en fonction du type de paramètre et de la forme de la liste d'initialisation.

Recherche de nom et contrôle d'accès

La résolution de la surcharge se produit *après la* recherche du nom. Cela signifie qu'une fonction mieux adaptée ne sera pas sélectionnée par une résolution de surcharge si elle perd la recherche de nom:

```
void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};
```

La résolution de la surcharge se produit *avant la* vérification de l'accès. Une fonction inaccessible peut être sélectionnée par résolution de surcharge si elle correspond mieux qu’une fonction accessible.

```
class C {
  public:
    static void f(double x);
  private:
    static void f(int x);
};
C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.
```

De même, la résolution de la surcharge se produit sans vérifier si l'appel résultant est bien formé en ce qui concerne l' `explicit` :

```
struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit
```

Surcharge sur la référence de transfert

Vous devez être très prudent lorsque vous fournissez une surcharge de référence de transfert car elle peut trop bien correspondre:

```
struct A {
    A() = default;           // #1
```

```
    A(A const& ) = default;  // #2

    template <class T>
    A(T&& );                 // #3
};
```

L'intention ici était que `A` soit copiable, et que nous ayons cet autre constructeur qui pourrait initialiser un autre membre. Toutefois:

```
A a;     // calls #1
A b(a);  // calls #3!
```

Il y a deux correspondances viables pour l'appel de construction:

```
A(A const& ); // #2
A(A& );       // #3, with T = A&
```

Les deux sont des correspondances exactes, mais le `#3` fait référence à un objet moins qualifié *cv* que le `#2` , il a donc la meilleure séquence de conversion standard et est la meilleure fonction viable.

La solution ici est de toujours contraindre ces constructeurs (par exemple en utilisant SFINAE):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
    >
A(T&& );
```

Le trait de type ici est d'exclure toute classe `A` ou dérivée publiquement et sans ambiguïté de `A` , ce qui rendrait ce constructeur mal formé dans l'exemple décrit précédemment (et donc supprimé de l'ensemble de surcharge). En conséquence, le constructeur de copie est appelé - ce que nous voulions.

Étapes de la résolution de surcharge

Les étapes de la résolution de la surcharge sont les suivantes:

1. Rechercher des fonctions candidates via la recherche de nom. Les appels non qualifiés effectueront à la fois une recherche régulière non qualifiée et une recherche dépendante des arguments (le cas échéant).

2. Filtrer l'ensemble des fonctions candidates sur un ensemble de fonctions *viables* . Une fonction viable pour laquelle il existe une séquence de conversion implicite entre les arguments avec lesquels la fonction est appelée et les paramètres pris par la fonction.

   ```
   void f(char);          // (1)
   void f(int ) = delete; // (2)
   void f();              // (3)
   void f(int& );         // (4)
   ```

```
f(4); // 1,2 are viable (even though 2 is deleted!)
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to
      //     a non-const lvalue reference
```

3. Choisissez le meilleur candidat viable. Une fonction viable `F1` est une fonction meilleure qu'une autre fonction viable `F2` si la séquence de conversion implicite pour chaque argument dans `F1` n'est pas pire que la séquence de conversion implicite correspondante dans `F2` , et ...:

   3.1. Pour certains arguments, la séquence de conversion implicite pour cet argument dans `F1` est une meilleure séquence de conversion que pour cet argument dans `F2` , ou

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

   3.2. Dans une conversion définie par l'utilisateur, la séquence de conversion standard du retour de `F1` au type de destination est une meilleure séquence de conversion que celle du type de retour de `F2` , ou

```
struct A
{
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

   3.3. Dans une liaison de référence directe, `F1` n'a pas le même type de référence que `F2` ou

```
struct A
{
    operator X&();  // #1
    operator X&&(); // #2
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

   3.4. `F1` n'est pas une spécialisation de modèle de fonction, mais `F2` est ou

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

   3.5. `F1` et `F2` sont tous deux des spécialisations de modèles de fonctions, mais `F1` est plus spécialisé que `F2` .

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized
```

La commande ici est significative. La meilleure vérification de la séquence de conversion se produit avant la vérification du modèle par rapport au modèle. Cela conduit à une erreur commune avec la surcharge sur la référence de transfert:

```
struct A {
    A(A const& ); // #1

    template <class T>
    A(T&& );      // #2, not constrained
};

A a;
A b(a); // calls #2!
        // #1 is not a template but #2 resolves to
        // A(A& ), which is a less cv-qualified reference than #1
        // which makes it a better implicit conversion sequence
```

S'il n'y a pas de meilleur candidat viable à la fin, l'appel est ambigu:

```
void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous
```

Promotions arithmétiques et conversions

La conversion d'un type entier en type promu correspondant est préférable à sa conversion en un autre type entier.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Il est préférable de `double` un `float` que de le convertir en un autre type en virgule flottante.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Les conversions arithmétiques autres que les promotions ne sont ni meilleures ni pires les unes que les autres.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous

void g(long x);
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Par conséquent, pour éviter toute ambiguïté lors de l'appel d'une fonction `f` avec des arguments intégraux ou à virgule flottante de n'importe quel type standard, un total de huit surcharges est nécessaire, de sorte que pour chaque type d'argument possible, une surcharge corresponde exactement ou la surcharge unique avec le type d’argument promu sera sélectionnée.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Surcharge dans une hiérarchie de classes

Les exemples suivants utiliseront cette hiérarchie de classes:

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

La conversion du type de classe dérivé en type de classe de base est préférable à celle des conversions définies par l'utilisateur. Cela s'applique lors du passage par valeur ou par référence, ainsi que lors de la conversion de pointeur en dérivé en pointeur à base.

```
struct Unrelated {
    Unrelated(B b);
};
void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

Une conversion de pointeur de classe dérivée en classe de base est également préférable à la conversion en `void*` .

```
void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)
```

S'il y a plusieurs surcharges dans la même chaîne d'héritage, la surcharge de classe de base la

plus dérivée est préférable. Ceci est basé sur un principe similaire à la répartition virtuelle: l'implémentation "la plus spécialisée" est choisie. Cependant, la résolution de la surcharge se produit toujours au moment de la compilation et ne sera jamais implicitement convertie.

```
void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable
```

Pour les pointeurs vers les membres, qui sont contravariants par rapport à la classe, une règle similaire s'applique dans la direction opposée: la classe dérivée la moins dérivée est préférable.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Surcharge de constance et de volatilité

Passer un argument de pointeur à un paramètre `T*` , si possible, est préférable à le passer à un paramètre `const T*` .

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
       // constness is only a "tie-breaker" rule
```

De même, il est préférable de transmettre un argument à un paramètre `T&` , si possible, à un paramètre `const T&` même si les deux ont un classement exact.

```
void f(int& r);
void f(const int& r);
int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable
```

Cette règle s'applique également aux fonctions membres qualifiées const, où il est important d'autoriser l'accès mutable aux objets non const et l'accès immuable aux objets const.

```
class IntVector {
```

```
  public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
  private:
    // ...
    int* m_data;
};
IntVector v1;
int* data1 = v1.data();       // Vector::data() is better than Vector::data() const;
                              // data1 can be used to modify the vector's data
const IntVector v2;
const int* data2 = v2.data(); // only Vector::data() const is viable;
                              // data2 can't be used to modify the vector's data
```

De la même manière, une surcharge volatile sera moins privilégiée qu'une surcharge non volatile.

```
class AtomicInt {
  public:
    // ...
    int load();
    int load() volatile;
  private:
    // ...
};
AtomicInt a1;
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1
```

Lire Résolution de surcharge en ligne: https://riptutorial.com/fr/cplusplus/topic/2021/resolution-de-surcharge

Chapitre 102: RTTI: Informations sur le type d'exécution

Examples

Nom d'un type

Vous pouvez récupérer le nom d'un type défini par l'implémentation à l'exécution en utilisant la fonction membre `.name()` de l'objet `std::type_info` renvoyé par `typeid` .

```
#include <iostream>
#include <typeinfo>

int main()
{
    int speed = 110;

    std::cout << typeid(speed).name() << '\n';
}
```

Sortie (définie par la mise en œuvre):

```
int
```

dynamic_cast

Utilisez `dynamic_cast<>()` comme une fonction qui vous aide à passer à travers une hiérarchie d'héritage (description principale).

Si vous devez effectuer un travail non polymorphe sur certaines classes dérivées `B` et `C` , mais avez reçu la `class A` base `class A` , écrivez comme ceci:

```
class A { public: virtual ~A(){} };

class B: public A
{ public: void work4B(){} };

class C: public A
{ public: void work4C(){} };

void non_polymorphic_work(A* ap)
{
  if (B* bp =dynamic_cast<B*>(ap))
    bp->work4B();
  if (C* cp =dynamic_cast<C*>(ap))
    cp->work4C();
}
```

Le mot-clé de typeid

Le mot - clé `typeid` est un opérateur unaire qui fournit des informations sur le type d'exécution de son opérande si le type de l'opérande est un type de classe polymorphe. Il retourne une lvalue de type `const std::type_info` . La qualification cv de niveau supérieur est ignorée.

```
struct Base {
    virtual ~Base() = default;
};
struct Derived : Base {};
Base* b = new Derived;
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` peut également être appliqué directement à un type. Dans ce cas, les premières références de niveau supérieur sont supprimées, puis la qualification cv de niveau supérieur est ignorée. Ainsi, l'exemple ci-dessus aurait pu être écrit avec `typeid(Derived)` au lieu de `typeid(Derived{})` :

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

Si `typeid` est appliqué à toute expression qui n'est *pas* de type de classe polymorphe, l'opérande n'est pas évalué et les informations de type renvoyées sont pour le type statique.

```
struct Base {
    // note: no virtual destructor
};
struct Derived : Base {};
Derived d;
Base& b = d;
assert(typeid(b) == typeid(Base)); // not Derived
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Quand utiliser qui jette en c ++

Utilisez **dynamic_cast** pour convertir les pointeurs / références dans une hiérarchie d'héritage.

Utilisez **static_cast** pour les conversions de type ordinaire.

Utilisez **reinterpret_cast** pour une **réinterprétation** de bas niveau des modèles de bits. Utilisez avec une extrême prudence.

Utilisez **const_cast** pour jeter const / volatile. Évitez ceci à moins que vous soyez bloqué en utilisant une API const-incorrecte.

Lire RTTI: Informations sur le type d'exécution en ligne:
https://riptutorial.com/fr/cplusplus/topic/3129/rtti--informations-sur-le-type-d-execution

Chapitre 103: Sémantique de valeur et de référence

Examples

Copie en profondeur et support de déplacement

Si un type souhaite avoir une sémantique de valeur et qu'il doit stocker des objets dynamiquement alloués, alors, lors des opérations de copie, le type devra allouer de nouvelles copies de ces objets. Il doit également le faire pour la copie.

Ce type de copie s'appelle une "copie profonde". Il prend efficacement ce qui aurait été la sémantique de référence et la transforme en sémantique de valeur:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }
};
```

C ++ 11

La sémantique de déplacement permet à un type comme `Value` d'éviter de copier réellement ses données référencées. Si l'utilisateur utilise la valeur d'une manière qui provoque un déplacement, le "copié" de l'objet peut être laissé vide des données référencées:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
```

```
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

En effet, on peut même rendre un tel type non copiable, si l'on veut interdire les copies profondes tout en permettant de déplacer l'objet.

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) = delete;
  Value &operator=(const Value &val) = delete;
```

```
  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

Nous pouvons même appliquer la règle de zéro, en utilisant l' `unique_ptr` :

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  unique_ptr<Inner []>array_; //Move-only type.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //No need to explicitly delete. Or even declare.
  ~Value() = default; {delete[] array_;}

  //No need to explicitly delete. Or even declare.
  Value(const Value &val) = default;
  Value &operator=(const Value &val) = default;

  //Will perform an element-wise move.
  Value(Value &&val) noexcept = default;

  //Will perform an element-wise move.
  Value &operator=(Value &&val) noexcept = default;
};
```

Définitions

Un type a une sémantique de valeur si l'état observable de l'objet est fonctionnellement distinct de tous les autres objets de ce type. Cela signifie que si vous copiez un objet, vous avez un nouvel objet, et les modifications du nouvel objet ne seront en aucun cas visibles depuis l'ancien objet.

La plupart des types C ++ de base ont une sémantique de valeur:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

La plupart des types définis de bibliothèque standard ont également une sémantique de valeur:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

Un type est dit avoir une sémantique de référence si une instance de ce type peut partager son état observable avec un autre objet (externe à celui-ci), de sorte que la manipulation d'un objet entraînera un changement d'état dans un autre objet.

Les pointeurs C ++ ont une sémantique de valeur par rapport à l'objet vers lequel ils pointent, mais ils ont une sémantique de référence en ce qui concerne l' *état* de l'objet vers lequel ils pointe:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

Les références C ++ ont également une sémantique de référence.

Lire Sémantique de valeur et de référence en ligne:
https://riptutorial.com/fr/cplusplus/topic/1955/semantique-de-valeur-et-de-reference

Chapitre 104: Sémaphore

Introduction

Les sémaphores ne sont pas disponibles en C ++ pour le moment, mais peuvent facilement être implémentés avec un mutex et une variable de condition.

Cet exemple provient de:

C ++ 0x n'a pas de sémaphores? Comment synchroniser les threads?

Examples

Sémaphore C ++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
    Semaphore (int count_ = 0)
    : count(count_)
    {
    }

    inline void notify( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        cout << "thread " << tid <<  " notify" << endl;
        //notify the waiting thread
        cv.notify_one();
    }
    inline void wait( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        while(count == 0) {
            cout << "thread " << tid << " wait" << endl;
            //wait on the mutex until notify is called
            cv.wait(lock);
            cout << "thread " << tid << " run" << endl;
        }
        count--;
    }
private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Classe de sémaphore en action

La fonction suivante ajoute quatre threads. Trois threads sont en compétition pour le sémaphore, dont le nombre est de un. Un thread plus lent appelle `notify_one()` , permettant à l'un des threads

en attente de continuer.

Le résultat est que `s1` commence immédiatement à tourner, faisant en sorte que le `count` utilisation du sémaphore reste inférieur à 1. Les autres threads attendent à tour de rôle la variable condition jusqu'à ce que notify () soit appelé.

```
int main()
{
    Semaphore sem(1);

    thread s1([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.wait( 1 );
        }
    });
    thread s2([&]() {
        while(true){
            sem.wait( 2 );
        }
    });
    thread s3([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::milliseconds(600));
            sem.wait( 3 );
        }
    });
    thread s4([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.notify( 4 );
        }
    });


    s1.join();
    s2.join();
    s3.join();
    s4.join();

    ...
    }
```

Lire Sémaphore en ligne: https://riptutorial.com/fr/cplusplus/topic/9785/semaphore

Chapitre 105: Séparateurs de chiffres

Examples

Séparateur de chiffres

Les littéraux numériques de plus de quelques chiffres sont difficiles à lire.

- Prononcez 7237498123.
- Comparez 237498123 avec 237499123 pour l'égalité.
- Décidez si 237499123 ou 20249472 est plus grand.

`C++14` définie offre simple Mark `'` comme un séparateur de chiffres, des chiffres et des littéraux définis par l' utilisateur. Cela peut aider les lecteurs humains à analyser plus facilement de grands nombres.

C ++ 14

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'0011ll;
```

Les guillemets simples sont ignorés lors de la détermination de sa valeur.

Exemple:

- Les littéraux `1048576` , `1'048'576` , `0X100000` , `0x10'0000` et `0'004'000'000` ont tous la même valeur.
- Les littéraux `1.602'176'565e-19` et `1.602176565e-19` ont la même valeur.

La position des guillemets simples n'est pas pertinente. Tous les éléments suivants sont équivalents:

C ++ 14

```
long long a1 = 123456789ll;
long long a2 = 123'456'789ll;
long long a3 = 12'34'56'78'9ll;
long long a4 = 12345'6789ll;
```

Il est également autorisé dans `user-defined` littéraux `user-defined` par l' `user-defined` :

C ++ 14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Lire Séparateurs de chiffres en ligne: https://riptutorial.com/fr/cplusplus/topic/10595/separateurs-

de-chiffres

Chapitre 106: SFINAE (échec de substitution n'est pas une erreur)

Examples

enable_if

`std::enable_if` est un utilitaire pratique pour utiliser des conditions booléennes pour déclencher SFINAE. Il est défini comme suit:

```
template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};
```

Autrement dit, `enable_if<true, R>::type` est un alias pour `R` , alors que `enable_if<false, T>::type` est mal formé car cette spécialisation de `enable_if` n'a pas de `type` membre.

`std::enable_if` peut être utilisé pour contraindre les templates:

```
int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }
```

Ici, un appel à `negate(1)` échouerait à cause de l'ambiguïté. Mais la deuxième surcharge n'est pas destinée à être utilisée pour les types intégraux, nous pouvons donc ajouter:

```
int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }
```

Maintenant, l'instanciation de la `negate<int>` entraînerait un échec de substitution car `!std::is_arithmetic<int>::value` est `false` . En raison de SFINAE, il ne s’agit pas d’une erreur grave, ce candidat est simplement supprimé de l’ensemble de surcharge. En conséquence, `negate(1)` n'a qu'un seul candidat viable - qui est alors appelé.

Quand l'utiliser

Il convient de garder à l'esprit que `std::enable_if` est une aide *en plus* de SFINAE, mais ce n'est pas ce qui fait que SFINAE fonctionne en premier lieu. Considérons ces deux alternatives pour implémenter des fonctionnalités similaires à `std::size` , à savoir une `size(arg)` jeu de surcharge

`size(arg)` qui produit la taille d'un conteneur ou d'un tableau:

```
// for containers
template<typename Cont>
auto size1(Cont const& cont) -> decltype( cont.size() );

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);
```

En supposant que `is_sizeable` est écrit correctement, ces deux déclarations doivent être exactement équivalentes à SFINAE. Quel est le plus facile à écrire et quel est le plus facile à revoir et à comprendre en un coup d'œil?

Considérons maintenant comment nous pourrions implémenter des aides arithmétiques qui évitent le débordement d'entier signé en faveur d'un comportement modulaire ou de bouclage. Ce qui veut dire que, par exemple, `incr(i, 3)` serait le même que `i += 3` sauf que le résultat serait toujours défini même si `i` est un `int` avec la valeur `INT_MAX` . Ce sont deux alternatives possibles:

```
// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);
```

Encore une fois, quel est le plus facile à écrire et quel est le plus facile à revoir et à comprendre en un coup d'œil?

La force de `std::enable_if` réside dans la manière dont il joue avec le refactoring et la conception des API. Si `is_sizeable<Cont>::value` est censé `cont.size()` si `cont.size()` est valide, alors il suffit d'utiliser l'expression telle qu'elle apparaît pour `size1` , bien que cela dépende du fait que `is_sizeable` soit utilisé à plusieurs endroits ou non. . Contraste avec `std::is_signed` qui reflète son

intention beaucoup plus clairement que lorsque son implémentation fuit dans la déclaration de `incr1` .

void_t

C ++ 11

`void_t` est une méta-fonction qui mappe tout (nombre de) types à taper `void` . Le but premier de `void_t` est de faciliter l'écriture des caractères de type.

`std::void_t` fera partie de C ++ 17, mais d'ici là, il est extrêmement simple à implémenter:

```
template <class...> using void_t = void;
```

Certains compilateurs nécessitent une implémentation légèrement différente:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

L'application principale de `void_t` consiste à écrire des traits de type qui vérifient la validité d'une instruction. Par exemple, vérifions si un type a une fonction membre `foo()` qui ne prend aucun argument:

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

Comment cela marche-t-il? Lorsque j'essaye d'instancier `has_foo<T>::value` , le compilateur essaiera de rechercher la meilleure spécialisation pour `has_foo<T, void>` . Nous avons deux options: la primaire et cette secondaire qui consiste à instancier cette expression sous-jacente:

- Si `T` a une fonction membre `foo()` , alors quel que soit le type qui retourne est converti en `void` , et la spécialisation est préférable au primaire sur la base des règles de classement partiel de modèle. Donc `has_foo<T>::value` sera `true`
- Si `T` *n'a pas* une telle fonction membre (ou nécessite plus d'un argument), la substitution échoue pour la spécialisation et nous n'avons que le modèle principal à utiliser. Par conséquent, `has_foo<T>::value` est `false` .

Un cas plus simple:

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T&>> : std::true_type {};
```

cela n'utilise pas `std::declval` ou `decltype` .

Vous remarquerez peut-être un schéma commun d'un argument vide. Nous pouvons prendre ceci en compte:

```
struct details {
  template<template<class...>class Z, class=void, class...Ts>
  struct can_apply:
    std::false_type
  {};
  template<template<class...>class Z, class...Ts>
  struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...>:
    std::true_type
  {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

qui cache l'utilisation de `std::void_t` et fait que `can_apply` agit comme un indicateur `can_apply` si le type fourni en tant que premier argument du template est bien formé après y avoir substitué les autres types. Les exemples précédents peuvent maintenant être réécrits en utilisant `can_apply` comme:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>;    // Is T& well formed for T?
```

et:

```
template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply< dot_foo_r, T >;    // Is T.foo() well formed for T?
```

ce qui semble plus simple que les versions originales.

Il existe des propositions post-C ++ 17 pour les traits `std` similaires à `can_apply` .

L'utilité de `void_t` été découverte par Walter Brown. Il a fait une magnifique présentation à CppCon 2016.

le decltype de fin dans les modèles de fonction

C ++ 11

L'une des fonctions contraignantes consiste à utiliser `decltype` fin pour spécifier le type de retour:

```
namespace details {
```

```
    using std::to_string;

    // this one is constrained on being able to call to_string(T)
    template <class T>
    auto convert_to_string(T const& val, int )
        -> decltype(to_string(val))
    {
        return to_string(val);
    }

    // this one is unconstrained, but less preferred due to the ellipsis argument
    template <class T>
    std::string convert_to_string(T const& val, ... )
    {
        std::ostringstream oss;
        oss << val;
        return oss.str();
    }
}

template <class T>
std::string convert_to_string(T const& val)
{
    return details::convert_to_string(val, 0);
}
```

Si j'appelle `convert_to_string()` avec un argument avec lequel je peux invoquer `to_string()` , j'ai deux fonctions viables pour les `details::convert_to_string()` . Le premier est préférable car la conversion de `0` en `int` est une meilleure séquence de conversion implicite que la conversion de `0` en `...`

Si j'appelle `convert_to_string()` avec un argument à partir duquel je ne peux pas appeler `to_string()` , la première instanciation de modèle de fonction entraîne un échec de substitution (il n'y a pas de `decltype(to_string(val))`). Par conséquent, ce candidat est supprimé de l'ensemble de surcharge. Le second modèle de fonction n'est pas contraint, il est donc sélectionné et nous passons par l' `operator<<(std::ostream&, T)` . Si celui-ci n'est pas défini, nous avons une erreur de compilation avec une pile de modèles sur la ligne `oss << val` .

Qu'est-ce que SFINAE?

SFINAE représente **S** ubstitution **D** tout manquement **I** S OT **N A** n **E** rror. Un code mal formé résultant de la substitution de types (ou de valeurs) pour instancier un modèle de fonction ou un modèle de classe n'est **pas** une erreur de compilation difficile, il est uniquement traité comme un échec de déduction.

Les échecs de déduction sur les modèles de fonction d'instanciation ou les spécialisations de modèle de classe suppriment ce candidat de l'ensemble considéré - comme si ce candidat en échec n'existait pas au départ.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }
```

```
int vals[10];
begin(vals); // OK. The first function template substitution fails because
             // vals.begin() is ill-formed. This is not an error! That function
             // is just removed from consideration as a viable overload candidate,
             // leaving us with the array overload.
```

Seuls les échecs de substitution dans le **contexte immédiat** sont considérés comme des échecs de déduction, tous les autres sont considérés comme des erreurs graves.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

enable_if_all / enable_if_any

C ++ 11

Exemple de motivation

Lorsque vous avez un pack de modèles variadic dans la liste des paramètres du modèle, comme dans l'extrait de code suivant:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

La bibliothèque standard (antérieure à C ++ 17) n'offre aucun moyen direct d'écrire **enable_if** pour imposer des contraintes SFINAE à ***tous les paramètres*** d' `Args` ou à l' ***un des paramètres*** d' `Args` . C ++ 17 offre `std::conjunction` et `std::disjunction` qui résolvent ce problème. Par exemple:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };

/// C++17: SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };
```

Si vous ne disposez pas de C ++ 17, il existe plusieurs solutions pour y parvenir. L'une d'entre elles consiste à utiliser une classe de base et **des spécialisations partielles** , comme le montrent les réponses à cette question .

Alternativement, on peut aussi implémenter le comportement de `std::conjunction` et de `std::disjunction` de manière assez directe. Dans l'exemple suivant, je vais présenter les

implémentations et les combiner avec `std::enable_if` pour produire deux alias: `enable_if_all` et `enable_if_any` , qui font exactement ce qu'ils sont supposés sémantiquement. Cela peut fournir une solution plus évolutive.

Implémentation de `enable_if_all` et `enable_if_any`

Commençons par émuler `std::conjunction` et `std::disjunction` utilisant respectivement `seq_and` et `seq_or` :

```
/// Helper for prior to C++14.
template<bool B, class T, class F >
using conditional_t = typename std::conditional<B,T,F>::type;

/// Emulate C++17 std::conjunction.
template<bool...> struct seq_or: std::false_type {};
template<bool...> struct seq_and: std::true_type {};

template<bool B1, bool... Bs>
struct seq_or<B1,Bs...>:
  conditional_t<B1,std::true_type,seq_or<Bs...>> {};

template<bool B1, bool... Bs>
struct seq_and<B1,Bs...>:
  conditional_t<B1,seq_and<Bs...>,std::false_type> {};
```

Ensuite, la mise en œuvre est assez simple:

```
template<bool... Bs>
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;

template<bool... Bs>
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

Finalement, quelques assistants:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

Usage

L'utilisation est également simple:

```
    /// SFINAE constraints on all of the parameters in Args.
    template<typename ...Args,
             enable_if_all_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };
```

```
    /// SFINAE constraints on any of the parameters in Args.
    template<typename ...Args,
             enable_if_any_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };
```

est détecté

Pour généraliser la création type_trait: basée sur SFINAE il y a des traits expérimentaux `detected_or` , `detected_t` , `is_detected` .

Avec les paramètres du template `typename Default` , `template <typename...> Op` et `typename ... Args` :

- `is_detected` : alias de `std::true_type` ou `std::false_type` fonction de la validité de `Op<Args...>`
- `detected_t` : alias `Op<Args...>` ou `nonesuch` selon la validité de `Op<Args...>` .
- `detected_or` : alias d'une struct avec `value_t` qui est `is_detected` , et le `type` qui est `Op<Args...>` ou `Default` en fonction de validité `Op<Args...>`

qui peut être implémenté en utilisant `std::void_t` pour SFINAE comme suit:

C ++ 17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };

} // namespace detail

// special type to indicate detection failure
struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
```

```
using detected_or = detail::detector<Default, void, Op, Args...>;
```

Les caractéristiques permettant de détecter la présence de la méthode peuvent alors être simplement implémentées:

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
              "Unexpected");

static_assert(std::is_same<void, // Default
                           detected_or<void, foo_type, C1, char>>::value,
              "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
              "Unexpected");
```

Résolution de surcharge avec un grand nombre d'options

Si vous avez besoin de choisir entre plusieurs options, l'activation d'une seule via `enable_if<>` peut être assez compliquée, car plusieurs conditions doivent également être annulées.

L'ordre entre les surcharges peut être sélectionné à l'aide de l'héritage, c.-à-d.

Au lieu de tester la chose qui doit être bien formée et de tester la négation de toutes les autres conditions de version, nous testons plutôt ce dont nous avons besoin, de préférence dans un `decltype` de retour dans un retour.
Cela peut laisser plusieurs options bien formées, nous distinguons celles utilisant des 'tags', similaires aux tags iterator-trait (`random_access_tag` et al). Cela fonctionne car une correspondance directe est préférable à une classe de base, ce qui est mieux qu'une classe de base d'une classe de base, etc.

```
#include <algorithm>
#include <iterator>

namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};
```

```
    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }

    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }

}

// this is the function the user calls. it will dispatch the call
// to the correct implementation with the help of 'tags'.
template<typename T>
void stable_sort(T& t)
{
    // use an N that is higher that any used above.
    // this will pick the highest overload that is well formed.
    detail::stable_sort(t, detail::pick<10>{});
}
```

Il existe d'autres méthodes couramment utilisées pour différencier les surcharges, telles que la correspondance exacte étant meilleure que la conversion, meilleure que les points de suspension.

Cependant, tag-dispatch peut s'étendre à un nombre illimité de choix et est un peu plus clair dans l'intention.

Lire SFINAE (échec de substitution n'est pas une erreur) en ligne: https://riptutorial.com/fr/cplusplus/topic/1169/sfinae--echec-de-substitution-n-est-pas-une-erreur-

Chapitre 107: Side by Side Comparaisons des exemples classiques en C ++ résolus via C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17

Examples

Boucler à travers un conteneur

En C ++, le bouclage via un conteneur de séquence `c` peut être effectué en utilisant les index suivants:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

Bien que simples, ces écritures sont sujettes à des erreurs sémantiques communes, comme un opérateur de comparaison erroné ou une variable d'indexation incorrecte:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
                     ^~~~~~~~~~~~~~^
```

Le bouclage peut également être réalisé pour tous les conteneurs utilisant des itérateurs, avec les mêmes inconvénients:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C ++ 11 a introduit des boucles basées sur des plages et `auto` mots `auto` clés `auto` , permettant au code de devenir:

```
for(auto& x : c) x = 0;
```

Ici, les seuls paramètres sont le conteneur `c` et une variable `x` contenant la valeur actuelle. Cela évite les erreurs de sémantique précédemment signalées.

Selon le standard C ++ 11, l'implémentation sous-jacente est équivalente à:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

Dans une telle implémentation, l'expression `auto begin = c.begin(), end = c.end();` les forces `begin` et `end` pour être du même type, alors que la `end` n'est jamais incrémentée, ni déréférencée. La boucle basée sur les plages ne fonctionne donc que pour les conteneurs définis par un itérateur / itérateur de paires. Le standard C ++ 17 assouplit cette contrainte en changeant l'implémentation en:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

Ici, le `begin` et la `end` peuvent être de différents types, du moment qu'ils peuvent être comparés pour l'inégalité. Cela permet de parcourir plusieurs conteneurs, par exemple un conteneur défini par un itérateur / sentinelle.

Lire Side by Side Comparaisons des exemples classiques en C ++ résolus via C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 en ligne: https://riptutorial.com/fr/cplusplus/topic/7134/side-by-side-comparaisons-des-exemples-classiques-en-c-plusplus-resolus-via-c-plusplus-vs-c-plusplus-11-vs-c-plusplus-14-vs-c-plusplus-17

Chapitre 108: Singleton Design Pattern

Remarques

Un **Singleton** est conçu pour garantir qu'une classe ne possède qu'une seule instance *et* fournit un point d'accès global à celle-ci. Si vous ne souhaitez qu'une seule instance *ou* un point d'accès global pratique, mais pas les deux, envisagez d'autres options avant de passer au singleton.

Les variables globales *peuvent* rendre la raison du code plus difficile. Par exemple, si l'une des fonctions appelantes n'est pas satisfaite des données qu'elle reçoit d'un Singleton, vous devez maintenant rechercher en premier lieu ce qui donne en premier lieu les données singleton incorrectes.

Les singletons encouragent également le couplage , un terme utilisé pour décrire deux composants du code qui sont réunis, réduisant ainsi la propre mesure d'auto-confinement de chaque composant.

Les singletons ne sont pas compatibles avec la concurrence. Lorsqu'une classe dispose d'un point d'accès global, chaque thread a la possibilité d'y accéder, ce qui peut entraîner des blocages et des conditions de course.

Enfin, l'initialisation différée peut entraîner des problèmes de performances si elle est initialisée au mauvais moment. La suppression de l'initialisation différée supprime également certaines des fonctionnalités qui rendent Singleton intéressant en premier lieu, comme le polymorphisme (voir Sous-classes).

Sources: Patterns de programmation de jeux par Robert Nystrom

Examples

Initialisation paresseuse

Cet exemple a été retiré de la section `Q & A` ici: http://stackoverflow.com/a/1008289/3807729

Voir cet article pour un design simple pour un singleton paresseux évalué avec destruction garantie:
Quelqu'un peut-il me fournir un échantillon de Singleton en c ++?

Le classique paresseux a évalué et correctement détruit le singleton.

```
class S
{
    public:
        static S& getInstance()
        {
            static S    instance; // Guaranteed to be destroyed.
                                  // Instantiated on first use.
```

```
            return instance;
        }
    private:
        S() {};                   // Constructor? (the {} brackets) are needed here.

        // C++ 03
        // ========
        // Dont forget to declare these two. You want to make sure they
        // are unacceptable otherwise you may accidentally get copies of
        // your singleton appearing.
        S(S const&);              // Don't Implement
        void operator=(S const&); // Don't implement

        // C++ 11
        // =======
        // We can use the better technique of deleting the methods
        // we don't want.
    public:
        S(S const&)               = delete;
        void operator=(S const&)  = delete;

        // Note: Scott Meyers mentions in his Effective Modern
        //       C++ book, that deleted functions should generally
        //       be public as it results in better error messages
        //       due to the compilers behavior to check accessibility
        //       before deleted status
};
```

Voir cet article sur quand utiliser un singleton: (pas souvent)
Singleton: Comment devrait-il être utilisé

Voir cet article sur l'ordre d'initialisation et comment faire face:
Ordre d'initialisation des variables statiques
Recherche des problèmes d'ordre d'initialisation statiques C ++

Voir cet article décrivant des vies:
Quelle est la durée de vie d'une variable statique dans une fonction C ++?

Voir cet article qui traite de certaines implications pour les singletons:
Instance Singleton déclarée comme variable statique de la méthode GetInstance

Voir cet article qui explique pourquoi le verrouillage à double vérification ne fonctionnera pas sous C ++:
Quels sont les comportements non définis courants qu'un programmeur C ++ doit connaître?

Des sous-classes

```
class API
{
public:
    static API& instance();

    virtual ~API() {}

    virtual const char* func1() = 0;
```

```
    virtual void func2() = 0;

protected:
    API() {}
    API(const API&) = delete;
    API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1()  override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM = LINUX
    static LinuxAPI instance;
#endif
    return instance;
}
```

Dans cet exemple, un simple commutateur de compilateur lie la classe `API` à la sous-classe appropriée. De cette manière, l' `API` est accessible sans être couplée à un code spécifique à la plate-forme.

Filet Singeton

C ++ 11

Les normes C ++ 11 garantissent que l'initialisation des objets de portée de fonction est initialisée de manière synchronisée. Cela peut être utilisé pour implémenter un singleton thread-safe avec initialisation différée .

```
class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};
```

Singleton de désinitialisation statique.

Il y a des fois avec plusieurs objets statiques où vous devez pouvoir garantir que le *singleton* ne sera pas détruit tant que tous les objets statiques utilisant le *singleton* n'en auront plus besoin.

Dans ce cas, `std::shared_ptr` peut être utilisé pour maintenir le *singleton en* vie pour tous les utilisateurs, même lorsque les destructeurs statiques sont appelés à la fin du programme:

```
class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};
```

REMARQUE: Cet exemple apparaît comme une réponse dans la section Q & R ici.

Lire Singleton Design Pattern en ligne: https://riptutorial.com/fr/cplusplus/topic/2713/singleton-design-pattern

Chapitre 109: Spécificateurs de classe de stockage

Introduction

Les spécificateurs de classe de stockage sont des mots clés pouvant être utilisés dans les déclarations. Ils n'affectent pas le type de déclaration, mais modifient généralement la manière dont l'entité est stockée.

Remarques

Il existe six spécificateurs de classe de stockage, mais pas tous dans la même version du langage: `auto` (jusqu'à C ++ 11), `register` (jusqu'à C ++ 17), `static` , `thread_local` (depuis C ++ 11), `extern` et `mutable` .

Selon la norme,

> Au plus, un *spécificateur de classe de stockage* doit apparaître dans une *déclaration-spécificateur-seq* donnée *,* sauf que `thread_local` peut apparaître avec `static` ou `extern` .

Une déclaration ne peut contenir aucun spécificateur de classe de stockage. Dans ce cas, le langage spécifie un comportement par défaut. Par exemple, par défaut, une variable déclarée à la portée du bloc a implicitement une durée de stockage automatique.

Examples

mutable

Un spécificateur qui peut être appliqué à la déclaration d'un membre de données non statique et non référencé d'une classe. Un membre mutable d'une classe n'est pas `const` quand l'objet est `const` .

```
class C {
    int x;
    mutable int times_accessed;
  public:
    C(): x(0), times_accessed(0) {
    }
    int get_x() const {
        ++times_accessed; // ok: const member function can modify mutable data member
        return x;
    }
    void set_x(int x) {
        ++times_accessed;
        this->x = x;
    }
```

```
};
```

C ++ 11

Une deuxième signification pour `mutable` été ajoutée en C ++ 11. Quand il suit la liste de paramètres d'un lambda, il supprime le `const` implicite sur l'opérateur d'appel de la fonction de lambda. Par conséquent, un lambda mutable peut modifier les valeurs des entités capturées par copie. Voir lambda mutable pour plus de détails.

```
std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                  [start]() mutable { return start++; });
    return result;
}
```

Notez que `mutable` n'est *pas* un spécificateur de classe de stockage lorsqu'il est utilisé de cette manière pour former un lambda modifiable.

registre

C ++ 17

Un spécificateur de classe de stockage qui indique au compilateur qu'une variable sera fortement utilisée. Le mot "register" est lié au fait qu'un compilateur pourrait choisir de stocker une telle variable dans un registre de CPU afin de pouvoir y accéder en moins de cycles d'horloge. Il était obsolète en C ++ 11.

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

Les variables locales et les paramètres de fonction peuvent être déclarés `register` . Contrairement à C, C ++ ne place aucune restriction sur ce qui peut être fait avec une variable de `register` . Par exemple, il est valable de prendre l'adresse d'une variable de `register` , mais cela peut empêcher le compilateur de stocker réellement une telle variable dans un registre.

C ++ 17

Le `register` mots clés est inutilisé et réservé. Un programme qui utilise le `register` mots-clés est mal formé.

statique

Le spécificateur de classe de stockage `static` a trois significations différentes.

1. Donne un lien interne à une variable ou une fonction déclarée dans la portée de l'espace de

noms.

```
// internal function; can't be linked to
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Déclare une variable pour avoir une durée de stockage statique (sauf s'il s'agit de `thread_local`). Les variables d'espace de nommage sont implicitement statiques. Une variable locale statique est initialisée une seule fois, la première fois que le contrôle passe par sa définition et n'est pas détruite à chaque fois que son étendue est supprimée.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";
}
```

3. Appliqué à la déclaration d'un membre de la classe, déclare que ce membre est un membre statique .

```
struct S {
    static S* create() {
        return new S;
    }
};
int main() {
    S* s = S::create();
}
```

Notez que dans le cas d'un membre de données statique d'une classe, 2 et 3 s'appliquent simultanément: le mot clé `static` transforme le membre en un membre de données statique et en fait une variable avec une durée de stockage statique.

auto

C ++ 03

Déclare une variable pour avoir une durée de stockage automatique. Il est redondant, car la durée de stockage automatique est déjà la valeur par défaut pour la portée du bloc et le spécificateur automatique n'est pas autorisé pour la portée de l'espace de noms.

```
void f() {
    auto int x; // equivalent to: int x;
    auto y;     // illegal in C++; legal in C89
}
auto int z;     // illegal: namespace-scope variable cannot be automatic
```

En C ++ 11, `auto` signification `auto` change complètement et n'est plus un spécificateur de classe de stockage, mais est plutôt utilisée pour la déduction de type .

externe

Le spécificateur de classe de stockage `extern` peut modifier une déclaration de l'une des trois manières suivantes, en fonction du contexte:

1. Il peut être utilisé pour déclarer une variable sans la définir. En règle générale, cela est utilisé dans un fichier d'en-tête pour une variable qui sera définie dans un fichier de mise en œuvre distinct.

```
// global scope
int x;             // definition; x will be default-initialized
extern int y;      // declaration; y is defined elsewhere, most likely another TU
extern int z = 42; // definition; "extern" has no effect here (compiler may warn)
```

2. Il donne un lien externe à une variable dans la portée de l'espace de noms, même si `const` ou `constexpr` aurait autrement eu un lien interne.

```
// global scope
const int w = 42;            // internal linkage in C++; external linkage in C
static const int x = 42;     // internal linkage in both C++ and C
extern const int y = 42;     // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                             // it's in an unnamed namespace
}
```

3. Elle redéfinit une variable à portée de bloc si elle a déjà été déclarée avec la liaison. Sinon, il déclare une nouvelle variable avec linkage, qui est un membre de l'espace de nommage le plus proche.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
}
void g() {
    extern int y; // y has external linkage; refers to global y defined elsewhere
}
```

Une fonction peut également être déclarée `extern` , mais cela n'a aucun effet. Il est généralement utilisé comme un indice pour le lecteur qu'une fonction déclarée ici est définie dans une autre unité de traduction. Par exemple:

```
void f();        // typically a forward declaration; f defined later in this TU
extern void g(); // typically not a forward declaration; g defined in another TU
```

Dans le code ci-dessus, si `f` était changé en `extern` et `g` en non- `extern` , cela n'affecterait pas du tout l'exactitude ou la sémantique du programme, mais pourrait induire le lecteur en erreur.

Lire Spécificateurs de classe de stockage en ligne:
https://riptutorial.com/fr/cplusplus/topic/9225/specificateurs-de-classe-de-stockage

Chapitre 110: Spécifications de liaison

Introduction

Une spécification de liaison indique au compilateur de compiler les déclarations de manière à ce qu'elles puissent être associées à des déclarations écrites dans un autre langage, tel que C.

Syntaxe

- extern *string-literal* { *declaration-seq* (*opt*)}
- *déclaration de chaîne de caractères* externe

Remarques

La norme exige que tous les compilateurs prennent en charge `extern "C"` pour permettre à C ++ d'être compatible avec C, et `extern "C++"` , qui peut être utilisé pour remplacer une spécification de liaison englobante et restaurer la valeur par défaut. Les autres spécifications de liaison prises en charge sont définies par la mise en œuvre .

Examples

Gestionnaire de signal pour un système d'exploitation de type Unix

Étant donné qu'un gestionnaire de signaux sera appelé par le noyau à l'aide de la convention d'appel C, nous devons indiquer au compilateur d'utiliser la convention d'appel C lors de la compilation de la fonction.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
    bind(...);
    listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

Rendre un en-tête de bibliothèque C compatible avec C ++

L'en-tête de bibliothèque AC peut généralement être inclus dans un programme C ++, puisque la

plupart des déclarations sont valides en C et en C ++. Par exemple, considérez le `foo.h` suivant:

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

La définition de `make_foo` est compilée et distribuée séparément avec l'en-tête sous forme d'objet.

Un programme C ++ peut `#include <foo.h>` , mais le compilateur ne saura pas que la fonction `make_foo` est définie comme un symbole C et essaiera probablement de la rechercher avec un nom déformé et ne pourra pas la localiser. Même si elle peut trouver la définition de `make_foo` dans la bibliothèque, toutes les plates - formes utilisent les mêmes conventions d'appel pour C et C ++, et le compilateur C ++ utilisera la convention d' appel C ++ lors de l' appel `make_foo` , ce qui est susceptible de provoquer une erreur de segmentation si `make_foo` s'attend à être appelé avec la convention d'appel C.

La manière de remédier à ce problème consiste à envelopper presque toutes les déclarations de l'en-tête dans un bloc `extern "C"` .

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

#ifdef __cplusplus
}  /* end of "extern C" block */
#endif
```

Maintenant, quand `foo.h` est inclus dans un programme C, il apparaîtra simplement comme des déclarations ordinaires, mais quand `foo.h` est inclus dans un programme C ++, `make_foo` sera dans un bloc `extern "C"` et le compilateur saura le rechercher. un nom non modifié et utilisez la convention d'appel C.

Lire Spécifications de liaison en ligne: https://riptutorial.com/fr/cplusplus/topic/9268/specifications-de-liaison

Chapitre 111: static_assert

Syntaxe

- static_assert (*bool_constexpr* , *message*)
- static_assert (*bool_constexpr*) / * Depuis C ++ 17 * /

Paramètres

Paramètre	Détails
bool_constexpr	Expression à vérifier
message	Message à imprimer lorsque *bool_constexpr* est *faux*

Remarques

Contrairement aux assertions d'exécution , les assertions statiques sont vérifiées à la compilation et sont également appliquées lors de la compilation de générations optimisées.

Examples

static_assert

Les assertions signifient qu'une condition doit être vérifiée et si elle est fausse, c'est une erreur. Pour `static_assert()` , ceci est fait à la compilation.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

Un `static_assert()` a un premier paramètre obligatoire, la condition, qui est un constexpr de bool. Il *pourrait* avoir un deuxième paramètre, le message, qui est un littéral de chaîne. A partir de C ++ 17, le second paramètre est facultatif. avant cela, c'est obligatoire.

C ++ 17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

Il est utilisé lorsque:

- En général, une vérification à la compilation est requise sur un type de valeur constexpr
- Une fonction de modèle doit vérifier certaines propriétés d'un type passé
- On veut écrire des cas de test pour:
  - métafonctions de modèle
  - fonctions constexpr
  - métaprogrammation macro
- Certaines définitions sont requises (par exemple, version C ++)
- Portage du code hérité, assertions sur `sizeof(T)` (par exemple, int 32 bits)
- Certaines fonctionnalités du compilateur sont requises pour que le programme fonctionne (emballage, optimisation de la classe de base vide, etc.)

Notez que `static_assert()` ne participe pas à SFINAE : ainsi, lorsque des surcharges / spécialisations supplémentaires sont possibles, il ne faut pas l’utiliser à la place des techniques de métaprogrammation des modèles (comme `std::enable_if<>`). Il peut être utilisé dans le code de modèle lorsque la surcharge / spécialisation attendue est déjà trouvée, mais des vérifications supplémentaires sont requises. Dans de tels cas, il peut fournir des messages d'erreur plus concrets que de s'appuyer sur SFINAE pour cela.

Lire static_assert en ligne: https://riptutorial.com/fr/cplusplus/topic/3822/static-assert

Chapitre 112: std :: any

Remarques

La classe `std::any` fournit un conteneur de type sécurisé auquel nous pouvons mettre des valeurs uniques de tout type.

Examples

Utilisation de base

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
  std::any_cast<int>(an_object);
} catch(std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Sortie

```
hello world
Wrong type
42
```

Lire std :: any en ligne: https://riptutorial.com/fr/cplusplus/topic/7894/std----any

Chapitre 113: std :: array

Paramètres

Paramètre	Définition
`class T`	Spécifie le type de données des membres du groupe
`std::size_t N`	Spécifie le nombre de membres dans le tableau

Remarques

L'utilisation d'un `std::array` nécessite l'inclusion de l'en-tête `<array>` aide de `#include <array>` .

Examples

Initialisation d'un tableau std ::

Initialisation de `std::array<T, N>` , où `T` est un type scalaire et `N` le nombre d'éléments de type `T`

Si `T` est un type scalaire, `std::array` peut être initialisé des manières suivantes:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Initialiser `std::array<T, N>` , où `T` est un type non scalaire et `N` est le nombre d'éléments de type `T`

Si `T` est un type non scalaire, `std::array` peut être initialisé des manières suivantes:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
```

```
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Accès aux éléments

1. `at(pos)`

Renvoie une référence à l'élément à la position `pos` avec vérification des bornes. Si `pos` n'est pas dans la plage du conteneur, une exception de type `std::out_of_range` est lancée.

La complexité est constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) `operator[pos]`

Renvoie une référence à l'élément à la position `pos` sans vérification des bornes. Si `pos` n'est pas dans la plage du conteneur, une erreur de *violation de segmentation à l'* exécution peut se produire. Cette méthode fournit un accès aux éléments équivalent aux tableaux classiques et à celui-ci plus efficace qu’à `at(pos)` .

La complexité est constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

Cette fonction **non-membre** renvoie une référence à l'élément à la position de la **constante de compilation** `pos` sans vérification des bornes. Si `pos` n'est pas dans la plage du conteneur, une erreur de *violation de segmentation à l'* exécution peut se produire.

La complexité est constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Renvoie une référence au premier élément du conteneur. L'appel de `front()` sur un conteneur vide n'est pas défini.

La complexité est constante O (1).

Note: Pour un conteneur c, l’expression `c.front()` est équivalente à `*c.begin()` .

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) **`back()`**

Renvoie la référence au dernier élément du conteneur. Le rappel `back()` sur un conteneur vide n'est pas défini.

La complexité est constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) **`data()`**

Renvoie le pointeur sur le tableau sous-jacent servant de stockage d'élément. Le pointeur est tel que la `range [data(); data() + size())` est toujours une plage valide, même si le conteneur est vide (`data()` n'est pas déréférencable dans ce cas).

La complexité est constante O (1).

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
}
```

Vérification de la taille du tableau

L'un des principaux avantages de `std::array` par rapport au tableau de style `C` est que nous pouvons vérifier la taille du tableau en utilisant la fonction membre `size()`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Itérer à travers le tableau

`std::array` étant un conteneur STL, peut utiliser une boucle basée sur une plage similaire à d'autres conteneurs comme un `vector`

```
int main() {
     std::array<int, 3> arr = { 1, 2, 3 };
     for (auto i : arr)
         cout << i << '\n';
}
```

Changer tous les éléments du tableau à la fois

Le membre function `fill()` peut être utilisé sur `std::array` pour modifier les valeurs immédiatement après l'initialisation

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

Lire std :: array en ligne: https://riptutorial.com/fr/cplusplus/topic/2712/std----array

Chapitre 114: std :: atomics

Examples

types atomiques

Chaque instanciation et spécialisation complète du modèle `std::atomic` définit un type atomique. Si un thread écrit sur un objet atomique alors qu'un autre thread en lit un, le comportement est bien défini (voir le modèle de mémoire pour plus de détails sur les courses de données)

De plus, les accès aux objets atomiques peuvent établir une synchronisation inter-thread et ordonner des accès mémoire non atomiques comme spécifié par `std::memory_order` .

std :: atomic peut être instancié avec n'importe quel `TriviallyCopyable type T. std::atomic` n'est ni copiable ni mobile.

La bibliothèque standard fournit des spécialisations du modèle std :: atomic pour les types suivants:

1. Une spécialisation complète du type `bool` et de son nom typedef est définie, traitée comme un `std::atomic<T>` non spécialisé, sauf qu’elle a une disposition standard, un constructeur par défaut trivial, des destructeurs triviaux et prend en charge la syntaxe d’initialisation globale:

Nom typedef	Spécialisation complète
`std::atomic_bool`	`std::atomic<bool>`

2) Spécialisations complètes et typedefs pour les types intégraux, comme suit:

Nom typedef	Spécialisation complète
`std::atomic_char`	`std::atomic<char>`
`std::atomic_char`	`std::atomic<char>`
`std::atomic_schar`	`std::atomic<signed char>`
`std::atomic_uchar`	`std::atomic<unsigned char>`
`std::atomic_short`	`std::atomic<short>`
`std::atomic_ushort`	`std::atomic<unsigned short>`
`std::atomic_int`	`std::atomic<int>`
`std::atomic_uint`	`std::atomic<unsigned int>`
`std::atomic_long`	`std::atomic<long>`
`std::atomic_ulong`	`std::atomic<unsigned long>`

Nom typedef	Spécialisation complète
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>

Nom typedef	Spécialisation complète
`std::atomic_size_t`	`std::atomic<std::size_t>`
`std::atomic_ptrdiff_t`	`std::atomic<std::ptrdiff_t>`
`std::atomic_intmax_t`	`std::atomic<std::intmax_t>`
`std::atomic_uintmax_t`	`std::atomic<std::uintmax_t>`

Exemple simple d'utilisation de std :: atomic_int

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::memory_order_relaxed
#include <thread>         // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
  foo.store(x,std::memory_order_relaxed);     // set value atomically
}

void print_foo() {
  int x;
  do {
    x = foo.load(std::memory_order_relaxed);  // get value atomically
  } while (x==0);
  std::cout << "foo: " << x << '\n';
}

int main ()
{
  std::thread first (print_foo);
  std::thread second (set_foo,10);
  first.join();
  //second.join();
  return 0;
}
//output: foo: 10
```

Lire std :: atomics en ligne: https://riptutorial.com/fr/cplusplus/topic/7475/std----atomics

Chapitre 115: std :: carte

Remarques

- Pour utiliser l'un des `std::map` ou `std::multimap` le fichier d'en-tête `<map>` doit être inclus.
- `std::map` et `std::multimap` conservent leurs éléments triés en fonction de l'ordre croissant des clés. Dans le cas de `std::multimap` , aucun tri n'est effectué pour les valeurs de la même clé.
- La différence fondamentale entre `std::map` et `std::multimap` est que `std::map` one n'autorise pas les valeurs en double pour la même clé, contrairement à `std::multimap` .
- Les cartes sont implémentées comme des arbres de recherche binaires. Ainsi, `search()` , `insert()` , `erase()` prend en moyenne Θ (log n). Pour une opération à temps constant, utilisez `std::unordered_map` .
- `size()` et `empty()` ont une complexité temporelle de Θ (1), le nombre de nœuds est mis en cache pour éviter de traverser un arbre à chaque appel de ces fonctions.

Examples

Accès aux éléments

Un `std::map` prend en entrée des paires `(key, value)` .

Prenons l'exemple suivant de l'initialisation `std::map` :

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),
                                        std::make_pair("docs-beta", 1) };
```

Dans un `std::map` , les éléments peuvent être insérés comme suit:

```
ranking["stackoverflow"]=2;
ranking["docs-beta"]=1;
```

Dans l'exemple ci-dessus, si le `stackoverflow` la clé est déjà présent, sa valeur sera mise à jour à 2. S'il n'est pas déjà présent, une nouvelle entrée sera créée.

Dans une `std::map` , il est possible d'accéder directement aux éléments en donnant la clé en tant qu'index:

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Notez que l'utilisation de l' `operator[]` sur la carte insèrera effectivement *une nouvelle valeur* avec la clé interrogée dans la carte. Cela signifie que vous ne pouvez pas l'utiliser sur un `const std::map` , même si la clé est déjà stockée dans la carte. Pour empêcher cette insertion, vérifiez si l'élément

existe (par exemple en utilisant `find()`) ou utilisez `at()` comme décrit ci-dessous.

C ++ 11

Les éléments d'un `std::map` peuvent être accédés avec `at()` :

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Notez que `at()` lancera une exception `std::out_of_range` si le conteneur ne contient pas l'élément demandé.

Dans les deux conteneurs `std::map` et `std::multimap` , les éléments sont accessibles à l'aide d'itérateurs:

C ++ 11

```
// Example using begin()
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
auto it = mmp.begin();
std::cout << it->first << " : " << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "1 : docs-beta"
```

Initialisation d'un std :: map ou std :: multimap

`std::map` et `std::multimap` peuvent tous deux être initialisés en fournissant des paires clé-valeur séparées par des virgules. Les paires clé-valeur peuvent être fournies par `{key, value}` ou peuvent être explicitement créées par `std::make_pair(key, value)` . Comme `std::map` n'autorise pas les clés en double et que l'opérateur de la virgule effectue de droite à gauche, la paire à droite sera remplacée par la paire avec la même clé à gauche.

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                     std::make_pair(1, "docs-beta"),
                                     std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange

std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                std::make_pair(1, "docs-beta"),
                                std::make_pair(2, "stackexchange")  };
```

```
// 1 docs-beta
// 2 stackoverflow
```

Les deux pourraient être initialisés avec un itérateur.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                               {6, 7} };
                       // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                        // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Supprimer des éléments

Supprimer tous les éléments:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap
```

Supprimer l'élément de quelque part avec l'aide de l'itérateur:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}
```

Supprimer tous les éléments d'une plage:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
it++; //moved first cursor on first {3, 4}
std::advance(it2,3);  //moved second cursor on first {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Enlever tous les éléments ayant une valeur fournie comme clé:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
```

```
                             // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Suppression d'éléments qui satisfont un prédicat `pred` :

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
   if (pred(*it))
       it = m.erase(it);
   else
       ++it;
}
```

Insertion d'éléments

Un élément peut être inséré dans une `std::map` uniquement si sa clé n'est pas déjà présente dans la carte. Étant donné par exemple:

```
std::map< std::string, size_t > fruits_count;
```

- Une paire clé-valeur est insérée dans une `std::map` via la fonction membre `insert()` . Il nécessite une `pair` en argument:

  ```
  fruits_count.insert({"grapes", 20});
  fruits_count.insert(make_pair("orange", 30));
  fruits_count.insert(pair<std::string, size_t>("banana", 40));
  fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));
  ```

  La fonction `insert()` renvoie une `pair` composée d'un itérateur et d'une valeur `bool` :

  - Si l'insertion a réussi, l'itérateur pointe sur l'élément nouvellement inséré et la valeur `bool` est `true` .
  - S'il y avait déjà un élément avec la même `key` , l'insertion échoue. Lorsque cela se produit, l'itérateur pointe vers l'élément à l'origine du conflit, et la valeur `bool` est `false` .

  La méthode suivante peut être utilisée pour combiner une opération d'insertion et de recherche:

  ```
  auto success = fruits_count.insert({"grapes", 20});
  if (!success.second) {           // we already have 'grapes' in the map
      success.first->second += 20; // access the iterator to update the value
  }
  ```

- Pour plus de commodité, le `std::map` fournit l'opérateur de l'indice pour accéder aux éléments de la carte et en insérer de nouveaux s'ils n'existent pas:

  ```
  fruits_count["apple"] = 10;
  ```

Bien que plus simple, il empêche l'utilisateur de vérifier si l'élément existe déjà. Si un élément est manquant, `std::map::operator[]` crée implicitement, l'initialisant avec le constructeur par défaut avant de le remplacer par la valeur fournie.

- `insert()` peut être utilisé pour ajouter plusieurs éléments à la fois en utilisant une liste de paires contreventée. Cette version de insert () renvoie void:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` peut également être utilisé pour ajouter des éléments en utilisant des itérateurs indiquant les valeurs de début et de fin de `value_type` :

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Exemple:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){            // 'fruit' is already in the map
        ++ret.first->second;    // increment the counter
    }
}
```

La complexité temporelle pour une opération d'insertion est O (log n) car `std::map` est implémenté sous forme d'arborescence.

C ++ 11

Une `pair` peut être construite explicitement en utilisant `make_pair()` et `emplace()` :

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

Si nous savons où le nouvel élément sera inséré, nous pouvons utiliser `emplace_hint()` pour spécifier un `hint` itérateur. Si le nouvel élément peut être inséré juste avant l’ `hint` , l’insertion peut être effectuée à temps constant. Sinon, il se comporte comme `emplace()` :

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Itération sur std :: map ou std :: multimap

`std::map` ou `std::multimap` pourrait être parcouru par les moyens suivants:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

//Range based loop - since C++11
for(const auto &x: mmp)
    std::cout<< x.first <<":"<< x.second << std::endl;

//Forward iterator for loop: it would loop through first element to last element
//it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
std::cout<< it->first <<":"<< it->second << std::endl; //Do something with iterator

//Backward iterator for loop: it would loop through last element to first element
//it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
std::cout<< it->first <<" "<< it->second << std::endl; //Do something with iterator
```

En itérant sur un `std::map` ou un `std::multimap` , l'utilisation de `auto` est préférable pour éviter les conversions implicites inutiles (voir cette réponse SO pour plus de détails).

Recherche dans std :: map ou dans std :: multimap

Il existe plusieurs façons de rechercher une clé dans `std::map` ou dans `std::multimap` .

- Pour obtenir l'itérateur de la première occurrence d'une clé, la fonction `find()` peut être utilisée. Il retourne `end()` si la clé n'existe pas.

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
else
    std::cout << "Value does not exist!" << std::endl;

it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "Value does not exist!" << std::endl; // This line would be executed.
```

- Une autre façon de déterminer si une entrée existe dans `std::map` ou dans `std::multimap` consiste à utiliser la fonction `count()` , qui compte le nombre de valeurs associées à une clé donnée. Puisque `std::map` n'associe qu'une seule valeur à chaque clé, sa fonction `count()` ne peut renvoyer que 0 (si la clé n'est pas présente) ou 1 (si elle est présente). Pour `std::multimap` , `count()` peut renvoyer des valeurs supérieures à 1 car il peut y avoir plusieurs valeurs associées à la même clé.

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0) // 3 exists as a key in map
    std::cout << "The key exists!" << std::endl; // This line would be executed.
else
    std::cout << "The key does not exist!" << std::endl;
```

Si vous vous souciez de savoir si un élément existe, `find` est strictement meilleur: il documente votre intention et, pour les `multimaps` , il peut s'arrêter une fois que le premier

élément correspondant a été trouvé.

- Dans le cas de `std::multimap` , il pourrait y avoir plusieurs éléments ayant la même clé. Pour obtenir cette plage, la fonction `equal_range()` est utilisée, qui renvoie respectivement `std::pair` avec une limite inférieure d'itérateur (inclus) et une limite supérieure (exclusive). Si la clé n'existe pas, les deux itérateurs pointe vers la `end()` .

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
    // prints: 6, 5
    //         6, 7
```

Vérification du nombre d'éléments

Le conteneur `std::map` a une fonction membre `empty()` , qui renvoie `true` ou `false` , selon que la carte est vide ou non. Le membre function `size()` renvoie le nombre d'éléments stockés dans un `std::map` :

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Types de cartes

Carte régulière

Une carte est un conteneur associatif contenant des paires clé-valeur.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

Dans l'exemple ci-dessus, `std::string` est le type de *clé* et `size_t` est une *valeur* .

La clé agit comme un index dans la carte. Chaque clé doit être unique et doit être commandée.

- Si vous avez besoin de plusieurs éléments avec la même clé, pensez à utiliser le `multimap` (expliqué ci-dessous)
- Si votre type de valeur ne spécifie aucun ordre ou que vous souhaitez remplacer le classement par défaut, vous pouvez en fournir un:

```
#include <string>
```

```
#include <map>
#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
               //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

Si le comparateur `StrLess` renvoie `false` pour deux clés, elles sont considérées comme identiques même si leur contenu réel est différent.

Multi-Map

Multimap permet de stocker plusieurs paires clé-valeur avec la même clé dans la carte. Sinon, son interface et sa création sont très similaires à la carte normale.

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

Hash-Map (carte non ordonnée)

Une carte de hachage stocke des paires clé-valeur similaires à une carte normale. Cependant, il ne commande pas les éléments par rapport à la clé. Au lieu de cela, une valeur de hachage pour la clé est utilisée pour accéder rapidement aux paires clé-valeur nécessaires.

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Les cartes non ordonnées sont généralement plus rapides, mais les éléments ne sont pas stockés dans un ordre prévisible. Par exemple, l'itération de tous les éléments d'un `unordered_map` donne les éléments dans un ordre apparemment aléatoire.

Création de std :: map avec les types définis par l'utilisateur comme clé

Pour pouvoir utiliser une classe comme clé dans une carte, il `copiable` que la clé soit `copiable` et `assignable` . L'ordre dans la carte est défini par le troisième argument du modèle (et l'argument du constructeur, s'il est utilisé). Par *défaut,* `std::less<KeyType>` , par défaut l'opérateur `<` , mais il n'est pas nécessaire d'utiliser les valeurs par défaut. Il suffit d'écrire un opérateur de comparaison (de préférence en tant qu'objet fonctionnel):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
```

```
        // ...
    }
};
```

En C ++, le prédicat "compare" doit être un ordre faible strict . En particulier, `compare(X,X)` doit retourner `false` pour tout `X` Si `CmpMyType()(a, b)` renvoie true, alors `CmpMyType()(b, a)` doit retourner false, et si les deux renvoient false, les éléments sont considérés égaux (membres de la même classe d'équivalence).

Strict Commande Faible

C'est un terme mathématique pour définir une relation entre deux objets.
Sa définition est la suivante:

> Deux objets x et y sont équivalents si f (x, y) et f (y, x) sont faux. Notez qu'un objet est toujours (par l'invariant de l'irréflexivité) équivalent à lui-même.

En termes de C ++, cela signifie que si vous avez deux objets d'un type donné, vous devez renvoyer les valeurs suivantes par rapport à l'opérateur <.

```
X    a;
X    b;

Condition:                  Test:     Result
a is equivalent to b:       a < b     false
a is equivalent to b        b < a     false

a is less than b            a < b     true
a is less than b            b < a     false

b is less than a            a < b     false
b is less than a            b < a     true
```

Comment vous définissez équivalent / moins dépend totalement du type de votre objet.

Lire std :: carte en ligne: https://riptutorial.com/fr/cplusplus/topic/681/std----carte

Chapitre 116: std :: forward_list

Introduction

`std::forward_list` est un conteneur qui prend en charge l'insertion et le retrait rapides d'éléments de n'importe où dans le conteneur. L'accès aléatoire rapide n'est pas pris en charge. Il est implémenté en tant que liste à lien unique et n'a pratiquement pas de surcharge par rapport à son implémentation en C. Comparé à `std::list` ce conteneur offre un stockage plus efficace lorsque l'itération bidirectionnelle n'est pas nécessaire.

Remarques

L'ajout, la suppression et le déplacement des éléments dans la liste, ou sur plusieurs listes, n'invalident pas les itérateurs faisant actuellement référence à d'autres éléments de la liste. Cependant, un itérateur ou une référence faisant référence à un élément est invalidé lorsque l'élément correspondant est supprimé (via erase_after) de la liste. std :: forward_list répond aux exigences de Container (sauf pour la fonction membre taille et la complexité de cet opérateur == est toujours linéaire), AllocatorAwareContainer et SequenceContainer.

Examples

Exemple

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::forward_list<std::string> words3(words1);
```

```
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Sortie:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

Les méthodes

Nom de la méthode	Définition
`operator=`	assigne des valeurs au conteneur
`assign`	assigne des valeurs au conteneur
`get_allocator`	renvoie l'allocateur associé
------	------
Accès aux éléments	
`front`	accéder au premier élément
------	------
Les itérateurs	
`before_begi` n	renvoie un itérateur à l'élément avant de commencer
`cbefore_begin`	renvoie un itérateur constant à l'élément avant de commencer
`begin`	renvoie un itérateur au début
`cbegin`	retourne un itérateur const au début
`end`	renvoie un itérateur à la fin
`cend`	renvoie un itérateur à la fin
Capacité	
`empty`	vérifie si le conteneur est vide
`max_size`	renvoie le nombre maximum possible d'éléments

Nom de la méthode	Définition
Modificateurs	
`clear`	efface le contenu
`insert_after`	insère des éléments après un élément
`emplace_after`	construit des éléments sur place après un élément
`erase_after`	efface un élément après un élément
`push_front`	insère un élément au début
`emplace_front`	construit un élément sur place au début
`pop_front`	supprime le premier élément
`resize`	change le nombre d'éléments stockés
`swap`	échange le contenu
Les opérations	
`merge`	fusionne deux listes triées
`splice_after`	déplace des éléments d'une autre forward_list
`remove`	supprime les éléments satisfaisant des critères spécifiques
`remove_if`	supprime les éléments satisfaisant des critères spécifiques
`reverse`	inverse l'ordre des éléments
`unique`	supprime les éléments en double consécutifs
`sort`	trie les éléments

Lire std :: forward_list en ligne: https://riptutorial.com/fr/cplusplus/topic/9703/std----forward-list

Chapitre 117: std :: function: Pour envelopper n'importe quel élément appelable

Examples

Usage simple

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

std :: function utilisé avec std :: bind

Pensez à une situation où nous devons rappeler une fonction avec des arguments. `std::function` utilisé avec `std::bind` donne une construction très puissante, comme indiqué ci-dessous.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }

};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ": " << i << std::endl;
    }
```

```
    void DoSomethingOnA()
    {
        anObjA.foo();
    }

    A anObjA;
};

int main(int argc, char *argv[])
{
     B anObjB;
     anObjB.DoSomethingOnA();
}
```

std :: function avec lambda et std :: bind

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
    std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                             [capture_value](int param) -> int { return 7 + capture_value *
param; })
              << std::endl;
    // result: 15 ==  value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
    std::cout << c << std::endl;
```

```
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

frais généraux de la fonction

`std::function` peut entraîner une surcharge importante. Comme `std::function` a [sémantique de valeur] [1], il doit copier ou déplacer l'appelant donné vers lui-même. Mais comme il peut prendre des callables d'un type arbitraire, il devra fréquemment allouer de la mémoire dynamiquement pour ce faire.

Certaines implémentations de `function` ont une soi-disant "optimisation des petits objets", où les petits types (tels que les pointeurs de fonction, les pointeurs membres ou les foncteurs avec très peu d'état) seront stockés directement dans l'objet `function` . Mais même cela ne fonctionne que si le type est `noexcept` mouvement constructible. De plus, le standard C ++ ne nécessite pas que toutes les implémentations en fournissent une.

Considérer ce qui suit:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;

void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

Un paramètre de modèle serait la solution préférée pour `SortMyContainer` , mais supposons que ce ne soit pas possible ou souhaitable pour une raison quelconque. `SortMyContainer` n'a pas besoin de stocker `pred` au-delà de son propre appel. Et pourtant, `pred` peut bien allouer de la mémoire si le foncteur qui lui est donné est de taille non triviale.

`function` alloue de la mémoire car elle a besoin de quelque chose à copier / déplacer; `function` prend possession du callable qui lui est donné. Mais `SortMyContainer` n'a pas besoin de *posséder* l'appelable; c'est juste le référencer. Donc, utiliser la `function` ici est excessif; c'est peut-être efficace, mais ce n'est peut-être pas le cas.

Il n'y a pas de type de fonction de bibliothèque standard qui référence simplement un appelable. Il faudra donc trouver une autre solution ou choisir de vivre avec les frais généraux.

De plus, la `function` n'a aucun moyen efficace de contrôler d'où proviennent les allocations de mémoire pour l'objet. Oui, il a des constructeurs qui prennent un `allocator` , mais [de nombreuses implémentations ne les implémentent pas correctement ... ou même pas *du tout*] [2].

C ++ 17

Les constructeurs de `function` qui prennent un `allocator` ne font plus partie du type. Par

conséquent, il n'y a aucun moyen de gérer l'allocation.

L'appel d'une `function` est également plus lent que l'appel direct du contenu. Comme toute instance de `function` peut contenir un appel, l'appel via une `function` doit être indirect. La surcharge de la `function` appel est dans l'ordre d'un appel de fonction virtuel.

Liaison std :: function à un autre type appelable

```
/*
 * This example show some ways of using std::function to call
 *  a) C-like function
 *  b) class-member function
 *  c) operator()
 *  d) lambda function
 *
 * Function call can be made:
 *  a) with right arguments
 *  b) argumens with different order, types and count
 */
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;



// simple function to be called
double foo_fn(int x, float y, double z)
{
  double res = x + y + z;
  std::cout << "foo_fn called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
  return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
    // this member function has different signature - but it can be used too
    // please not that argument order is changed too
    double foo_fn_4(int x, double z, float y, long xx)
    {
        double res = x + y + z + xx;
```

```
        std::cout << "foo_struct::foo_fn_4 called with arguments: "
                << x << ", " << z << ", " << y << ", " << xx
                << " result is : " << res
                << std::endl;
        return res;
    }
    // overloaded operator() makes whole object to be callable
    double operator()(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::operator() called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
};


int main(void)
{
  // typedefs
  using function_type = std::function<double(int, float, double)>;

  // foo_struct instance
  foo_struct fs;

  // here we will store all binded functions
  std::vector<function_type> bindings;

  // var #1 - you can use simple function
  function_type var1 = foo_fn;
  bindings.push_back(var1);

  // var #2 - you can use member function
  function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
  bindings.push_back(var2);

  // var #3 - you can use member function with different signature
  // foo_fn_4 has different count of arguments and types
  function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, 0l);
  bindings.push_back(var3);

  // var #4 - you can use object with overloaded operator()
  function_type var4 = fs;
  bindings.push_back(var4);

  // var #5 - you can use lambda function
  function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lambda  called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    };
  bindings.push_back(var5);

  std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
            << std::endl;
```

```
  for (auto f : bindings)
      f(1, 2, 3);

}
```

Vivre

Sortie:

```
Test stored functions with arguments: x = 1, y = 2, z = 3
foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6
lambda  called with arguments: 1, 2, 3 result is : 6
```

Stocker des arguments de fonction dans std :: tuple

Certains programmes doivent donc stocker des arguments pour les appels futurs de certaines fonctions.

Cet exemple montre comment appeler n'importe quelle fonction avec des arguments stockés dans std :: tuple

```
#include <iostream>
#include <functional>
#include <tuple>
#include <iostream>

// simple function to be called
double foo_fn(int x, float y, double z)
{
   double res =  x + y + z;
   std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z
             << " res=" << res;
   return res;
}

// helpers for tuple unrolling
template<int ...> struct seq {};
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };

// invocation helper
template<typename FN, typename P, int ...S>
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)
{
   return fn(std::get<S>(params) ...);
}
// call function with arguments stored in std::tuple
template<typename Ret, typename ...Args>
Ret call_fn(const std::function<Ret(Args...)>& fn,
            const std::tuple<Args...>& params)
{
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());
```

```
}


int main(void)
{
  // arguments
  std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);
  // function to call
  std::function<double(int, float, double)> fn = foo_fn;

  // invoke a function with stored arguments
  call_fn(fn, t);
}
```

Vivre

Sortie:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

Lire std :: function: Pour envelopper n'importe quel élément appelable en ligne: https://riptutorial.com/fr/cplusplus/topic/2294/std----function--pour-envelopper-n-importe-quel-element-appelable

Chapitre 118: std :: integer_sequence

Introduction

Le modèle de classe `std::integer_sequence<Type, Values...>` représente une séquence de valeurs de type `Type` où `Type` est l'un des types entiers intégrés. Ces séquences sont utilisées lors de l'implémentation de modèles de classe ou de fonction bénéficiant d'un accès positionnel. La bibliothèque standard contient également des types "factory" qui créent des séquences ascendantes de valeurs entières à partir du nombre d'éléments.

Examples

Tournez un std :: tuple en paramètres de fonction

Un `std::tuple<T...>` peut être utilisé pour transmettre plusieurs valeurs. Par exemple, il pourrait être utilisé pour stocker une séquence de paramètres dans une forme de file d'attente. Lors du traitement d'un tel tuple, ses éléments doivent être transformés en arguments d'appel de fonction:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// ----------------------------------------------------------------------------
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\n";
}

// ----------------------------------------------------------------------------
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
}
```

```
// ---------------------------------------------------------------------------
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

Tant qu'une classe supporte `std::get<I>(object)` et `std::tuple_size<T>::value` elle peut être développée avec la fonction `process()` ci-dessus. La fonction elle-même est entièrement indépendante du nombre d'arguments.

Créer un pack de paramètres composé d'entiers

`std::integer_sequence` lui-même consiste à maintenir une séquence d'entiers pouvant être transformée en un paquet de paramètres. Sa principale valeur est la possibilité de créer des modèles de classe "factory" en créant ces séquences:

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << '\n';
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

Le modèle de fonction `print_sequence()` utilise un `std::initializer_list<bool>` lors de l'extension de la séquence entière pour garantir l'ordre d'évaluation et ne pas créer de variable [array] inutilisée.

Transforme une séquence d'indices en copies d'un élément

L'extension du paquet de paramètres d'index dans une expression virgule avec une valeur crée une copie de la valeur pour chacun des index. Malheureusement, gcc et clang pense que l'indice n'a pas d' effet et mettre en garde à ce sujet (gcc peut être réduit au silence par coulée l'indice d'

`void`):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
    auto array = make_array<20>(std::string("value"));
    std::copy(array.begin(), array.end(),
              std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << "\n";
}
```

Lire std :: integer_sequence en ligne: https://riptutorial.com/fr/cplusplus/topic/8315/std----integer-sequence

Chapitre 119: std :: iomanip

Examples

std :: setw

```
int val = 10;
// val will be printed to the extreme left end of the  output console:
std::cout << val << std::endl;
 // val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;
```

Cela produit:

```
10
        10
1234567890
```

(où la dernière ligne est là pour aider à voir les décalages de caractères).

Parfois, nous devons définir la largeur du champ de sortie, généralement lorsque nous avons besoin d'une sortie structurée et appropriée. Cela peut être fait en utilisant **`std::setw`** de **std :: iomanip** .

La syntaxe de `std::setw` est la suivante:

```
std::setw(int n)
```

où n est la longueur du champ de sortie à définir

std :: setprecision

Lorsqu'il est utilisé dans une expression `out << setprecision(n)` ou `in >> setprecision(n)` , définit le paramètre de précision du flux ou bien exactement n. Le paramètre de cette fonction est un entier, qui est une nouvelle valeur pour la précision.

Exemple:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
              << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
              << "max precision:         "
              << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
```

```
            << pi << '\n';
}
//Output
//default precision (6): 3.14159
//std::precision(10):    3.141592654
//max precision:         3.141592653589793239
```

std :: setfill

Utilisé dans une expression `out << setfill(c)` définit le caractère de remplissage du flux sur `c` .

Remarque: Le caractère de remplissage actuel peut être obtenu avec `std::ostream::fill` .

Exemple:

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
          << "setfill('*'): " << std::setfill('*')
                              << std::setw(10) << 42 << '\n';
}
//output::
//default fill:         42
//setfill('*'): ********42
```

std :: setiosflags

Lorsqu'elle est utilisée dans une expression `out << setiosflags(mask)` ou `in >> setiosflags(mask)` , définit tous les indicateurs de format du flux comme étant spécifiés par le masque.

Liste de tous les `std::ios_base::fmtflags` :

- `dec` - utilise la base décimale pour les E / S entières
- `oct` - utilise une base octale pour les E / S entières
- `hex` - utilise la base hexadécimale pour l'entier E / S
- `basefield` - `dec|oct|hex|0` utile pour les opérations de masquage
- `left` - réglage gauche (ajouter des caractères de remplissage à droite)
- `right` - ajustement à droite (ajoute des caractères de remplissage à gauche)
- `internal` - ajustement interne (ajoute des caractères de remplissage au point désigné interne)
- `adjustfield` - `left|right|internal` . Utile pour les opérations de masquage
- `scientific` - génère des types à virgule flottante en utilisant la notation scientifique ou la notation hexadécimale si elle est combinée à
- `fixed` - génère des types à virgule flottante en utilisant une notation fixe ou une notation hexadécimale si elle est associée à des éléments scientifiques
- `floatfield` - `scientific|fixed|(scientific|fixed)|0` . Utile pour les opérations de masquage
- `boolalpha` - insérer et extraire le type `bool` au format alphanumérique
- `showbase` - génère un préfixe indiquant la base numérique pour la sortie entière, nécessite l'indicateur de devise dans les E / S monétaires

- `showpoint` - génère un caractère de point décimal sans condition pour la sortie de nombre à virgule flottante
- `showpos` - génère un caractère `+` pour une sortie numérique non négative
- `skipws` - `skipws` les espaces blancs avant certaines opérations de saisie
- `unitbuf` la sortie après chaque opération de sortie
- `uppercase` - remplace certaines lettres minuscules par leurs équivalents majuscules dans certaines opérations de sortie

Exemple de manipulateurs:

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
  int l_iTemp = 47;
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl;
  //output: 57
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl;
  //output: 2f
  std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl;
  //output 2F
  std::cout<<std::setfill('0')<<std::setw(12);
  std::cout<<std::resetiosflags(std::ios_base::uppercase);
  std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl;
  //output: 00000000002f

  std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
  std::cout<<std::setfill('.')<<std::setw(10);
  std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl;
  //output: 47........

  std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
  std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
  std::cout<<std::setw(10)<<l_iTemp<<std::endl;
  //output +#######47

  double l_dTemp = -1.2;
  double pi = 3.14159265359;
  std::cout<<pi<<"    "<<l_dTemp<<std::endl;
  //output +3.14159   -1.2
  std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl;
  //output -1.20000
  std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl;
  //output: +3.141593e+00
  std::cout<<std::resetiosflags(std::ios_base::floatfield);
  std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
  //output: +3.141593
  bool b = true;
  std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
  //output: true
  return 0;
}
```

Lire std :: iomanip en ligne: https://riptutorial.com/fr/cplusplus/topic/6936/std----iomanip

Chapitre 120: std :: optionnel

Examples

introduction

Les options (également appelés types Maybe) sont utilisés pour représenter un type dont le contenu peut ou non être présent. Ils sont implémentés en C ++ 17 en tant que classe `std::optional` . Par exemple, un objet de type `std::optional<int>` peut contenir une valeur de type `int` ou ne contenir aucune valeur.

Les options sont généralement utilisées pour représenter une valeur qui n'existe pas ou comme type de retour à partir d'une fonction qui peut ne pas renvoyer un résultat significatif.

Autres approches optionnelles

Il y a beaucoup d'autres approches pour résoudre le problème que `std::optional` résout, mais aucune n'est assez complète: utiliser un pointeur, utiliser une sentinelle ou utiliser une `pair<bool, T>` .

Facultatif vs pointeur

Dans certains cas, nous pouvons fournir un pointeur vers un objet existant ou `nullptr` pour indiquer un échec. Mais ceci est limité aux cas où des objets existent déjà - `optional` , en tant que type de valeur, peut également être utilisé pour renvoyer de nouveaux objets sans avoir recours à l'allocation de mémoire.

Facultatif vs Sentinel

Un idiome commun consiste à utiliser une valeur spéciale pour indiquer que la valeur n'a pas de sens. Cela peut être 0 ou -1 pour les types intégraux, ou `nullptr` pour les pointeurs. Cependant, cela réduit l'espace des valeurs valides (vous ne pouvez pas différencier un 0 valide et un 0 sans signification) et de nombreux types n'ont pas le choix naturel pour la valeur sentinelle.

Facultatif vs `std::pair<bool, T>`

Un autre idiome commun est de fournir une paire, où l'un des éléments est un `bool` indiquant si la valeur est significative ou non.

Cela suppose que le type de valeur soit default-constructible en cas d'erreur, ce qui n'est pas possible pour certains types et possible mais indésirable pour d'autres. Un `optional<T>` , en cas d'erreur, n'a pas besoin de construire quoi que ce soit.

Utiliser des options pour représenter l'absence de valeur

Avant C ++ 17, avoir des pointeurs avec une valeur `nullptr` représentait généralement l'absence d'une valeur. C'est une bonne solution pour les objets volumineux qui ont été alloués dynamiquement et qui sont déjà gérés par des pointeurs. Cependant, cette solution ne fonctionne pas bien pour les types petits ou primitifs tels que `int` , qui sont rarement alloués ou gérés de manière dynamique par des pointeurs. `std::optional` fournit une solution viable à ce problème commun.

Dans cet exemple, `struct Person` est défini. Il est possible pour une personne d'avoir un animal de compagnie, mais pas nécessaire. Par conséquent, le membre `pet` de `Person` est déclaré avec un wrapper `std::optional` .

```
#include <iostream>
#include <optional>
#include <string>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Utiliser des options pour représenter l'échec d'une fonction

Avant C ++ 17, une fonction représentait généralement l'échec de plusieurs manières:

- Un pointeur nul a été renvoyé.
    - Par exemple, appeler une fonction `Delegate *App::get_delegate()` sur une instance d' `App` n'ayant pas de délégué renverrait `nullptr` .
    - C'est une bonne solution pour les objets qui ont été alloués dynamiquement ou qui sont grands et gérés par des pointeurs, mais qui ne sont pas une bonne solution pour les petits objets qui sont généralement affectés par pile et transmis par copie.
- Une valeur spécifique du type de retour était réservée pour indiquer une défaillance.
    - Par exemple, appeler une fonction `unsigned shortest_path_distance(Vertex a, Vertex b)` sommet a, sommet `unsigned shortest_path_distance(Vertex a, Vertex b)` sur deux sommets non connectés peut renvoyer zéro pour indiquer ce fait.
- La valeur a été associée à une valeur `bool` pour indiquer que la valeur renvoyée était significative.

- Par exemple, appeler une fonction `std::pair<int, bool> parse(const std::string &str)` avec un argument de chaîne qui n'est pas un entier renverrait une paire avec un `int` indéfini et un `bool` défini sur `false` .

Dans cet exemple, John reçoit deux animaux, Fluffy et Furball. La fonction `Person::pet_with_name()` est ensuite appelée pour récupérer les `Person::pet_with_name()` John. Comme John n'a pas d'animal de compagnie nommé Whiskers, la fonction échoue et `std::nullopt` est renvoyé à la place.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;

    std::optional<Animal> pet_with_name(const std::string &name) {
        for (const Animal &pet : pets) {
            if (pet.name == name) {
                return pet;
            }
        }
        return std::nullopt;
    }
};

int main() {
    Person john;
    john.name = "John";

    Animal fluffy;
    fluffy.name = "Fluffy";
    john.pets.push_back(fluffy);

    Animal furball;
    furball.name = "Furball";
    john.pets.push_back(furball);

    std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
    if (whiskers) {
        std::cout << "John has a pet named Whiskers." << std::endl;
    }
    else {
        std::cout << "Whiskers must not belong to John." << std::endl;
    }
}
```

optionnel comme valeur de retour

```
std::optional<float> divide(float a, float b) {
  if (b!=0.f) return a/b;
```

```
  return {};
}
```

Ici, nous renvoyons soit la fraction `a/b` , mais si elle n’est pas définie (ce serait l’infini), nous renvoyons le vide facultatif.

Un cas plus complexe:

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
  using std::begin; using std::end;
  auto b = begin(r), e = end(r);
  auto r = std::find_if(b, e , p );
  using iterator = decltype(r);
  if (r==e)
    return std::optional<iterator>();
  return std::optional<iterator>(r);
}
template<class Range, class T>
auto find( Range&& r, T const& t ) {
  return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}
```

`find( some_range, 7 )` recherche le conteneur ou la plage `some_range` pour quelque chose égal au nombre `7` . `find_if` fait avec un prédicat.

Il retourne soit une option vide si elle n'a pas été trouvée, soit une option contenant un itérateur à l'élément si c'était le cas.

Cela vous permet de faire:

```
if (find( vec, 7 )) {
  // code
}
```

ou même

```
if (auto oit = find( vec, 7 )) {
  vec.erase(*oit);
}
```

sans avoir à jouer avec les itérateurs de début / fin et les tests.

valeur_ou

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
  std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

`value_or` renvoie soit la valeur stockée dans le `value_or` facultatif, soit l'argument s'il n'y a rien dans la mémoire.

Cela vous permet de prendre la valeur facultative-null et de donner un comportement par défaut lorsque vous avez réellement besoin d'une valeur. En procédant de cette façon, la décision "comportement par défaut" peut être repoussée au point où il est préférable et immédiatement nécessaire, au lieu de générer une valeur par défaut au plus profond des entrailles de certains moteurs.

Lire std :: optionnel en ligne: https://riptutorial.com/fr/cplusplus/topic/2423/std----optionnel

Chapitre 121: std :: pair

Examples

Créer une paire et accéder aux éléments

Pair nous permet de traiter deux objets comme un seul objet. Les paires peuvent être facilement construites à l'aide de la fonction de template `std::make_pair` .

Une autre manière consiste à créer une paire et à affecter ses éléments (`first` et `second`) plus tard.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements



    //We can also create a pair and assign the elements later
    std::pair<int,int> p1;
    p1.first = 3;
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //We can also create a pair using a constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;
}
```

Comparer les opérateurs

Les paramètres de ces opérateurs sont `lhs` et `rhs`

- `operator==` teste si les deux éléments sur `lhs` et `rhs` paire sont égales. La valeur `lhs.first == rhs.first` est `true` si `lhs.first == rhs.first` AND `lhs.second == rhs.second` , sinon `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);

if (p1 == p2)
    std::cout << "equals";
else
    std::cout << "not equal"//statement will show this, because they are not identical
```

- `operator!=` teste si des éléments sur `lhs` et `rhs` paire ne sont pas égaux. La valeur `lhs.first`

`!= rhs.first` est `true` si `lhs.first != rhs.first` OU `lhs.second != rhs.second` , sinon retourne `false` .

- `operator<` teste si `lhs.first<rhs.first` , renvoie `true` . Sinon, si `rhs.first<lhs.first` renvoie `false` . Sinon, si `lhs.second<rhs.second` renvoie `true` , sinon, renvoie `false` .
- `operator<=` retourne `!(rhs<lhs)`
- `operator>` renvoie `rhs<lhs`
- `operator>=` retourne `!(lhs<rhs)`

Un autre exemple avec des conteneurs de paires. Il utilise un `operator<` car il doit trier le conteneur.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
                                                   {2, "bar"},
                                                   {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

Lire std :: pair en ligne: https://riptutorial.com/fr/cplusplus/topic/4834/std----pair

Chapitre 122: std :: set et std :: multiset

Introduction

`set` est un type de conteneur dont les éléments sont triés et uniques. `multiset` est similaire, mais, dans le cas de `multiset` , plusieurs éléments peuvent avoir la même valeur.

Remarques

Différents styles de C ++ ont été utilisés dans ces exemples. Faites attention que si vous utilisez un compilateur C ++ 98; une partie de ce code peut ne pas être utilisable.

Examples

Insérer des valeurs dans un ensemble

Trois méthodes d’insertion différentes peuvent être utilisées avec les ensembles.

- Tout d'abord, une simple insertion de la valeur. Cette méthode renvoie une paire permettant à l'appelant de vérifier si l'insertion s'est réellement produite.
- Deuxièmement, un insert en donnant une indication de l'endroit où la valeur sera insérée. L'objectif est d'optimiser le temps d'insertion dans un tel cas, mais il n'est pas courant de savoir où une valeur doit être insérée. **Soyez prudent dans ce cas; la manière de donner un indice diffère avec les versions du compilateur** .
- Enfin, vous pouvez insérer une plage de valeurs en donnant un pointeur de début et de fin. Le premier sera inclus dans l'insertion, le dernier est exclu.

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> sut;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // Basic insert
  sut.insert(7);
  sut.insert(5);
  sut.insert(12);

  ret = sut.insert(23);
  if (ret.second==true)
    std::cout << "# 23 has been inserted!" << std::endl;

  ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert
should fail
  if (ret.second==false)
    std::cout << "# 23 already present in set!" << std::endl;
```

```
  // Insert with hint for optimization
  it = sut.end();
  // This case is optimized for C++11 and above
  // For earlier version, point to the element preceding your insertion
  sut.insert(it, 30);

  // inserting a range of values
  std::set<int> sut2;
  sut2.insert(20);
  sut2.insert(30);
  sut2.insert(45);
  std::set<int>::iterator itStart = sut2.begin();
  std::set<int>::iterator itEnd = sut2.end();

  sut.insert (itStart, itEnd); // second iterator is excluded from insertion

  std::cout << std::endl << "Set under test contains:" << std::endl;
  for (it = sut.begin(); it != sut.end(); ++it)
  {
    std::cout << *it << std::endl;
  }

  return 0;
}
```

La sortie sera:

```
# 23 has been inserted!

# 23 already present in set!



Set under test contains:

5

7

12

20

23

30

45
```

Insertion de valeurs dans un multiset

Toutes les méthodes d'insertion des ensembles s'appliquent également aux multisets. Néanmoins, il existe une autre possibilité, qui fournit une initializer_list:

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
```

```
msut.insert(il);
```

Changer le type de jeu par défaut

`set` et `multiset` ont des méthodes de comparaison par défaut, mais dans certains cas, vous devrez peut-être les surcharger.

Imaginons que nous stockions des valeurs de chaîne dans un ensemble, mais nous savons que ces chaînes ne contiennent que des valeurs numériques. Par défaut, le tri sera une comparaison de chaîne lexicographique, donc l'ordre ne correspondra pas au tri numérique. Si vous voulez appliquer une sorte équivalente à ce que vous auriez avec les valeurs `int` , vous avez besoin d'un foncteur pour surcharger la méthode de comparaison:

```
#include <iostream>
#include <set>
#include <stdlib.h>

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"},
                                                     custom_compare{}); //< Compare object
optional as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
        std::cout << data << std::endl;

    return 0;
}
```

La sortie sera:

```
### Default sort on std::set<std::string> :
1
2
23
290
5
6
### Custom sort on set :
1
2
5
6
23
290

### Lambda sort on set :
6
5
290
23
2
1
```

Dans l'exemple ci-dessus, on peut trouver 3 manières différentes d'ajouter des opérations de comparaison à `std::set` , chacune étant utile dans son propre contexte.

Tri par défaut

Cela utilisera l'opérateur compare de la clé (premier argument du modèle). Souvent, la clé fournira déjà une bonne valeur par défaut pour la fonction `std::less<T>` . A moins que cette fonction soit spécialisée, elle utilise l' `operator<` de l'objet. Ceci est particulièrement utile lorsque d'autres codes essaient également d'utiliser un certain ordre, car cela permet une cohérence sur toute la base de code.

En écrivant le code de cette façon, vous réduirez les efforts de mise à jour de votre code lorsque les modifications clés sont des API, comme par exemple: une classe contenant 2 membres qui devient une classe contenant 3 membres. En mettant à jour l' `operator<` dans la classe, toutes les occurrences seront mises à jour.

Comme vous vous en doutez, utiliser le tri par défaut est une valeur par défaut raisonnable.

Tri personnalisé

L'ajout d'un tri personnalisé via un objet avec un opérateur de comparaison est souvent utilisé lorsque la comparaison par défaut n'est pas conforme. Dans l'exemple ci-dessus, c'est parce que les chaînes font référence à des entiers. Dans d'autres cas, il est souvent utilisé lorsque vous souhaitez comparer des pointeurs (intelligents) basés sur l'objet auquel ils se réfèrent ou parce que vous avez besoin de contraintes différentes pour comparer (exemple: comparer `std::pair` à la valeur du `first`).

Lors de la création d'un opérateur de comparaison, cela devrait être un tri stable. Si le résultat de

l'opérateur de comparaison change après l'insertion, vous aurez un comportement indéfini. Comme bonne pratique, votre opérateur de comparaison ne doit utiliser que les données constantes (membres const, fonctions const ...).

Comme dans l'exemple ci-dessus, vous rencontrerez souvent des classes sans membres en tant qu'opérateurs de comparaison. Cela se traduit par des constructeurs par défaut et des constructeurs de copie. Le constructeur par défaut vous permet d'omettre l'instance au moment de la construction et le constructeur de la copie est requis car l'ensemble prend une copie de l'opérateur de comparaison.

Type Lambda

Les Lambdas sont un moyen plus court d'écrire des objets de fonction. Cela permet d'écrire l'opérateur de comparaison sur moins de lignes, ce qui rend le code global plus lisible.

L'inconvénient de l'utilisation de lambdas est que chaque lambda obtient un type spécifique au moment de la compilation, de sorte que `decltype(lambda)` sera différent pour chaque compilation de la même unité de compilation (fichier cpp) que sur plusieurs unités de compilation). Pour cette raison, il est recommandé d'utiliser des objets de fonction comme opérateur de comparaison lorsqu'ils sont utilisés dans des fichiers d'en-tête.

Cette construction est souvent rencontrée lorsqu'un objet `std::set` est utilisé dans la portée locale d'une fonction, tandis que l'objet fonction est préféré lorsqu'il est utilisé comme argument de fonction ou membre de classe.

Autres options de tri

Comme l'opérateur de comparaison de `std::set` est un argument de modèle, tous les objets appelables peuvent être utilisés en tant qu'opérateur de comparaison et les exemples ci-dessus ne sont que des cas spécifiques. Les seules restrictions de ces objets appelables sont les suivantes:

- Ils doivent être constructibles
- Ils doivent pouvoir être appelés avec 2 arguments du type de la clé. (les conversions implicites sont autorisées, mais pas recommandées car elles peuvent nuire aux performances)

Recherche de valeurs dans set et multiset

Il existe plusieurs manières de rechercher une valeur donnée dans `std::set` ou dans `std::multiset` :

Pour obtenir l'itérateur de la première occurrence d'une clé, la fonction `find()` peut être utilisée. Il retourne `end()` si la clé n'existe pas.

```
std::set<int> sut;
sut.insert(10);
```

```
  sut.insert(15);
  sut.insert(22);
  sut.insert(3); // contains 3, 10, 15, 22

  auto itS = sut.find(10); // the value is found, so *itS == 10
  itS = sut.find(555); // the value is not found, so itS == sut.end()

  std::multiset<int> msut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(15);
  sut.insert(3); // contains 3, 10, 15, 15, 22

  auto itMS = msut.find(10);
```

Une autre méthode consiste à utiliser la fonction `count()` , qui compte combien de valeurs correspondantes ont été trouvées dans `set` / `multiset` (dans le cas d'un `set` , la valeur de retour peut être seulement 0 ou 1). En utilisant les mêmes valeurs que ci-dessus, nous aurons:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

Dans le cas de `std::multiset` , il pourrait y avoir plusieurs éléments ayant la même valeur. Pour obtenir cette plage, la fonction `equal_range()` peut être utilisée. Il renvoie `std::pair` ayant la borne inférieure de l'itérateur (inclus) et la limite supérieure (exclusive) respectivement. Si la clé n'existe pas, les deux itérateurs indiqueraient la valeur supérieure la plus proche (selon la méthode de comparaison utilisée pour trier le `multiset` donné).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'

eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

Suppression de valeurs d'un ensemble

La méthode la plus évidente, si vous voulez simplement réinitialiser votre set / multiset à un ensemble vide, consiste à utiliser `clear` :

```
  std::set<int> sut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(3);
  sut.clear(); //size of sut is 0
```

Ensuite, la méthode d' `erase` peut être utilisée. Il offre des possibilités qui ressemblent quelque peu à l'insertion:

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
  std::cout << *it << std::endl;
}
```

La sortie sera:

```
Set under test contains:

10

15

30
```

Toutes ces méthodes s'appliquent également au `multiset` . Veuillez noter que si vous demandez de supprimer un élément d'un `multiset` , et qu'il est présent plusieurs fois, **toutes les valeurs équivalentes seront supprimées** .

Lire std :: set et std :: multiset en ligne: https://riptutorial.com/fr/cplusplus/topic/9005/std----set-et-std----multiset

Chapitre 123: std :: string

Introduction

Les chaînes sont des objets qui représentent des séquences de caractères. La norme `string` de classe offre une alternative simple, sûr et polyvalent à l' aide de tableaux explicites de `char` s lorsqu'ils traitent avec du texte et d' autres séquences de caractères. La classe de `string` C ++ fait partie de l'espace de noms `std` et a été standardisée en 1998.

Syntaxe

- *// Déclaration de chaîne vide*

  std :: string s;

- *// Construire à partir de const char * (c-string)*

  std :: string s ("Bonjour");

  std :: string s = "Bonjour";

- *// Construire en utilisant un constructeur de copie*

  std :: string s1 ("Bonjour");

  std :: string s2 (s1);

- *// Construire à partir d'une sous-chaîne*

  std :: string s1 ("Bonjour");

  std :: string s2 (s1, 0, 4); *// Copie 4 caractères de la position 0 de s1 dans s2*

- *// Construire à partir d'un tampon de caractères*

  std :: string s1 ("Hello World");
  std :: string s2 (s1, 5); *// Copie les 5 premiers caractères de s1 dans s2*

- *// Construire en utilisant le constructeur de remplissage (char uniquement)*

  std :: string s (5, 'a'); *// s contient aaaaa*

- *// Construit à l'aide du constructeur et de l'itérateur d'intervalle*

  std :: string s1 ("Hello World");

  std :: string s2 (s1.begin (), s1.begin () + 5); *// Copie les 5 premiers caractères de s1 dans s2*

Remarques

Avant d'utiliser `std::string` , vous devez inclure la `string` tête, car elle inclut des fonctions / opérateurs / surcharges que les autres en-têtes (par exemple `iostream`) n'incluent pas.

L'utilisation de const char * constructeur avec un nullptr conduit à un comportement indéfini.

```
std::string oops(nullptr);
std::cout << oops << "\n";
```

La méthode `at` lève une exception `std::out_of_range` si `index >= size()` .

Le comportement de l' `operator[]` est un peu plus compliqué, dans tous les cas il a un comportement indéfini si `index > size()` , mais quand `index == size()` :

C ++ 11

1. Sur une chaîne non-const, le comportement est *indéfini* .
2. Sur une chaîne de caractères const, une référence à un caractère avec la valeur `CharT()` (le caractère *nul*) est renvoyée.

C ++ 11

1. Une référence à un caractère avec la valeur `CharT()` (le caractère *nul*) est renvoyée.
2. La modification de cette référence est *un comportement indéfini* .

Depuis C ++ 14, au lieu d'utiliser `"foo"` , il est recommandé d'utiliser `"foo"s` , car `s` est un suffixe littéral défini par l'utilisateur , qui convertit le caractère `const char* "foo"` en `std::string "foo"` .

Remarque: vous devez utiliser l'espace de noms `std::string_literals` ou `std::literals std::string_literals` pour obtenir le littéral `s` .

Examples

Scission

Utilisez `std::string::substr` pour diviser une chaîne. Il existe deux variantes de cette fonction membre.

Le premier prend une *position* de *départ à* partir de laquelle la sous-chaîne renvoyée doit commencer. La position de départ doit être valide dans la plage `(0, str.length()]` :

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

La seconde prend une position de départ et une *longueur* totale de la nouvelle sous-chaîne.

Quelle que soit la *longueur* , la sous-chaîne ne dépassera jamais la fin de la chaîne source:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(15, 3); // "and"
```

Notez que vous pouvez aussi appeler `substr` sans argument, dans ce cas une copie exacte de la chaîne est retournée

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Remplacement de cordes

Remplacer par la position

Pour remplacer une partie de `std::string` vous pouvez utiliser la méthode `replace` from `std::string` .

`replace` a beaucoup de surcharges utiles:

```
//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

C ++ 14

```
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
```

C ++ 11

```
//8)
```

```
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Remplacer les occurrences d'une chaîne par une autre chaîne

Ne remplacez que la première occurrence de `replace` with `with` in `str` :

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Remplace toutes les occurrences de `replace` par `with` in dans `str` :

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

Enchaînement

Vous pouvez concaténer `std::string` s en utilisant les opérateurs `+` et `+=` surchargés. En utilisant l'opérateur `+` :

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

En utilisant l'opérateur `+=` :

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

Vous pouvez également ajouter des chaînes C, y compris des chaînes de caractères:

```
std::string hello = "Hello";
```

```
std::string world = "world";
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

Vous pouvez également utiliser `push_back()` pour repousser personne `char` s:

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

Il y a aussi `append()` , qui ressemble à `+=` :

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Accéder à un personnage

Il existe plusieurs manières d'extraire des caractères d'une `std::string` et chacun est subtilement différent.

```
std::string str("Hello world!");
```

---

opérateur [] (n)

Renvoie une référence au caractère à l'index n.

`std::string::operator[]` n'est pas contrôlé par des limites et ne lance pas d'exception. L'appelant est responsable de l'affirmation que l'index se situe dans la plage de la chaîne:

```
char c = str[6]; // 'w'
```

---

en (n)

Renvoie une référence au caractère à l'index n.

`std::string::at` *est* vérifié et lancera `std::out_of_range` si l'index n'est pas dans la plage de la chaîne:

```
char c = str.at(7); // 'o'
```

C ++ 11

Remarque: Ces deux exemples entraîneront un comportement indéfini si la chaîne est vide.

---

de face()

Renvoie une référence au premier caractère:

```
char c = str.front(); // 'H'
```

---

arrière()

Renvoie une référence au dernier caractère:

```
char c = str.back(); // '!'
```

Tokenize

Inscrit du moins cher au plus cher à l'exécution:

1. `str::strtok` est la méthode de tokenisation standard la moins chère, elle permet également de modifier le délimiteur entre les jetons, mais cela entraîne 3 difficultés avec le C ++ moderne:

   - `std::strtok` ne peut pas être utilisé sur plusieurs `strings` en même temps (bien que certaines implémentations s'étendent pour supporter ceci, telles que: `strtok_s`)
   - Pour la même raison, `std::strtok` ne peut pas être utilisé sur plusieurs threads simultanément (cela peut cependant être défini par l'implémentation, par exemple: l'implémentation de Visual Studio est thread-safe)
   - L'appel de `std::strtok` modifie le `std::string` sur lequel il opère, il ne peut donc pas être utilisé sur des `const string` `const char*` , des `const char*` s ou des chaînes littérales, pour marquer l'un de ces éléments avec `std::strtok` ou pour fonctionner sur un `std::string` dont le contenu doit être préservé, l'entrée doit être copiée, puis la copie peut être utilisée

   Généralement, le coût de ces options est caché dans le coût d'allocation des jetons, mais si l'algorithme le moins cher est requis et que les difficultés de `std::strtok` ne sont pas surmontables, envisagez une solution manuelle .

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Exemple Live

2. `std::istream_iterator` utilise de manière itérative l'opérateur d'extraction du flux. Si l'entrée `std::string` est délimitée par des espaces blancs, elle peut être étendue à l'option `std::strtok` en éliminant ses difficultés, permettant ainsi la tokenisation en ligne, ce qui prend en charge la génération d'un `const vector<string>` et en ajoutant plusieurs délimiter le caractère d'espace blanc:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
                                        std::istream_iterator<std::string>());
```

Exemple Live

3. Le `std::regex_token_iterator` utilise un `std::regex` pour marquer de manière itérative. Il fournit une définition de délimiteur plus souple. Par exemple, les virgules non délimitées et les espaces blancs:

C ++ 11

```
// String to tokenize
const std::string str{ "The ,qu\\,ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\\\\,]|\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Exemple Live

Voir l' exemple de `regex_token_iterator` pour plus de détails.

Conversion en (const) char *

Afin d'obtenir `const char*` l' accès aux données d'un `std::string` , vous pouvez utiliser de la chaîne `c_str()` la fonction de membre. Gardez à l'esprit que le pointeur n'est valide que tant que l'objet `std::string` est dans la portée et reste inchangé, ce qui signifie que seules des méthodes `const` peuvent être appelées sur l'objet.

C ++ 17

La fonction membre `data()` peut être utilisée pour obtenir un caractère modifiable `char*` , qui peut être utilisé pour manipuler les données de l'objet `std::string` .

C ++ 11

Un caractère modifiable `char*` peut également être obtenu en prenant l'adresse du premier caractère: `&s[0]` . Dans C ++ 11, il est garanti de générer une chaîne bien formée, terminée par un caractère nul. Notez que `&s[0]` est bien formé même si `s` est vide, alors que `&s.front()` est indéfini si `s` est vide.

C ++ 11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
```

```
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);

// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Recherche de caractère (s) dans une chaîne

Pour trouver un caractère ou une autre chaîne, vous pouvez utiliser `std::string::find` . Il renvoie la position du premier caractère du premier match. Si aucune correspondance n'a été trouvée, la fonction renvoie `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

> Trouvé à la position: 21

Les possibilités de recherche sont élargies par les fonctions suivantes:

```
find_first_of     // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of      // Find last occurrence of characters
find_last_not_of  // Find last absence of characters
```

Ces fonctions peuvent vous permettre de rechercher des caractères à la fin de la chaîne et de trouver la casse négative (c.-à-d. Des caractères qui ne sont pas dans la chaîne). Voici un exemple:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

> Trouvé à la position: 6

Remarque: Sachez que les fonctions ci-dessus ne recherchent pas des sous-chaînes, mais plutôt des caractères contenus dans la chaîne de recherche. Dans ce cas, la dernière occurrence de `'g'` été trouvée à la position `6` (les autres caractères n'ont pas été trouvés).

Découpe des caractères au début / à la fin

Cet exemple requiert les en-têtes `<algorithm>` , `<locale>` et `<utility>` .

C ++ 11

Pour *découper* une séquence ou une chaîne, vous devez supprimer tous les éléments (ou caractères) de début et de fin correspondant à un prédicat donné. Nous découpons d'abord les éléments de fin, car cela n'implique aucun déplacement d'éléments, puis nous découpons les éléments principaux. Notez que les généralisations ci-dessous fonctionnent pour tous les types de `std::basic_string` (par exemple `std::string` et `std::wstring`), et accidentellement également pour les conteneurs de séquence (par exemple `std::vector` et `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Découper les éléments de fin implique de trouver le *dernier* élément ne correspondant pas au prédicat et d'effacer à partir de là:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Découper les éléments principaux implique de trouver le *premier* élément ne correspondant pas au prédicat et d'y effacer:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

Pour spécialiser ce qui précède pour couper les espaces dans un `std::string` nous pouvons utiliser la fonction `std::isspace()` comme prédicat:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
```

```
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

De même, nous pouvons utiliser la fonction `std::iswspace()` pour `std::wstring` etc.

Si vous souhaitez créer une *nouvelle* séquence qui est une copie rognée, vous pouvez utiliser une fonction distincte:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Comparaison lexicographique

Deux `std::string` s peuvent être comparés lexicographiquement en utilisant les opérateurs `==` `!=` , `<` , `<=` , `>` Et `>=` :

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

Toutes ces fonctions utilisent la méthode `std::string::compare()` sous-jacente pour effectuer la comparaison et renvoient les valeurs booléennes de commodité. Le fonctionnement de ces fonctions peut être interprété comme suit, quelle que soit l'implémentation réelle:

- opérateur `==` :

  Si `str1.length() == str2.length()` et que chaque paire de caractères correspond, alors retourne `true` , sinon retourne `false` .

- opérateur `!=` :

  Si `str1.length() != str2.length()` ou une paire de caractères ne correspond pas, renvoie `true` , sinon elle renvoie `false` .

- opérateur `<` ou opérateur `>` :

  Trouve la première paire de caractères différente, les compare, puis retourne le résultat booléen.

- opérateur `<=` ou opérateur `>=` :

Trouve la première paire de caractères différente, les compare, puis retourne le résultat booléen.

Remarque: Le terme " **paire de caractères"** désigne les caractères correspondants dans les deux chaînes des mêmes positions. Pour mieux comprendre, si deux exemples de chaînes sont `str1` et `str2` , et que leurs longueurs sont respectivement `n` et `m` , alors les paires de caractères des deux chaînes signifient chaque `str1[i]` et `str2[i]` où *i = 0, 1, 2* `str2[i]` *.., max (n, m)* . Si pour tout *i* où le caractère correspondant n'existe pas, c'est-à-dire lorsque *i* est supérieur ou égal à `n` ou `m` , il serait considéré comme la valeur la plus basse.

Voici un exemple d'utilisation de `<` :

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

Les étapes sont les suivantes:

1. Comparez les premiers caractères, `'B' == 'B'` - continuez.
2. Comparez les deuxièmes caractères, `'a' == 'a'` - continuez.
3. Comparez les troisièmes caractères, `'r' == 'r'` - continuez.
4. La plage `str2` est maintenant épuisée, alors que la plage `str1` a toujours des caractères. Ainsi, `str2 < str1` .

Conversion en std :: wstring

En C ++, les séquences de caractères sont représentées en spécialisant la classe `std::basic_string` avec un type de caractère natif. Les deux principales collections définies par la bibliothèque standard sont `std::string` et `std::wstring` :

- `std::string` est construit avec des éléments de type `char`
- `std::wstring` est construit avec des éléments de type `wchar_t`

Pour convertir entre les deux types, utilisez `wstring_convert` :

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

Afin d'améliorer la convivialité et / ou la lisibilité, vous pouvez définir des fonctions pour effectuer la conversion:

```
#include <string>
#include <codecvt>
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;

std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Exemple d'utilisation:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

C'est certainement plus lisible que `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` .

Veuillez noter que `char` et `wchar_t` n'impliquent pas de codage et ne donne aucune indication de taille en octets. Par exemple, `wchar_t` est généralement implémenté en tant que type de données à 2 octets et contient généralement des données encodées en UTF-16 sous Windows (ou UCS-2 dans les versions antérieures à Windows 2000) et un type de données à 4 octets encodé en UTF-32. Linux Ceci est en contraste avec les nouveaux types `char16_t` et `char32_t` , qui ont été introduits en C ++ 11 et qui sont garantis suffisamment grands pour contenir respectivement tout "caractère" UTF16 ou UTF32 (ou plus précisément, *point de code*).

Utiliser la classe std :: string_view

C ++ 17

C ++ 17 introduit `std::string_view` , qui est simplement une plage non propriétaire de `const char` , implémentable sous la forme d'une paire de pointeurs ou d'un pointeur et d'une longueur. C'est un type de paramètre supérieur pour les fonctions qui nécessitent des données de chaîne non modifiables. Avant C ++ 17, il y avait trois options pour cela:

```
void foo(std::string const& s);      // pre-C++17, single argument, could incur
                                     // allocation if caller's data was not in a string
                                     // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                     // both everywhere
```

```
void foo(const char* s);             // pre-C++17, single argument, but need to call
                                     // strlen()

template <class StringT>
void foo(StringT const& s);          // pre-C++17, caller can pass arbitrary char data
                                     // provider, but now foo() has to live in a header
```

Tous ces éléments peuvent être remplacés par:

```
void foo(std::string_view s);        // post-C++17, single argument, tighter coupling
                                     // zero copies regardless of how caller is storing
                                     // the data
```

*Notez que `std::string_view` **ne peut** modifier ses données sous-jacentes* .

`string_view` est utile lorsque vous souhaitez éviter les copies inutiles.

Il offre un sous-ensemble utile des fonctionnalités de `std::string` , bien que certaines fonctions se comportent différemment:

```
std::string str = "lllloooonnnngggg sssstttrrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

En boucle à travers chaque personnage

C ++ 11

`std::string` supporte les itérateurs, et vous pouvez donc utiliser une boucle *basée sur la distance* pour parcourir chaque caractère:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

Vous pouvez utiliser une boucle "traditionnelle" `for` parcourir chaque caractère:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Conversion en entiers / types à virgule flottante

Un `std::string` contenant un nombre peut être converti en un type entier, ou un type à virgule flottante, en utilisant des fonctions de conversion.

Notez que toutes ces fonctions cessent d'analyser la chaîne d'entrée dès qu'elles rencontrent un caractère non numérique, ainsi `"123abc"` sera converti en `123` .

La famille de fonctions `std::ato*` convertit les chaînes de style C (tableaux de caractères) en types entiers ou à virgule flottante:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

C ++ 11

```
long long num4 = std::atoll(ten.c_str());
```

Cependant, l'utilisation de ces fonctions est déconseillée car elles renvoient `0` si elles ne permettent pas d'analyser la chaîne. Ceci est mauvais car `0` peut aussi être un résultat valide, si par exemple la chaîne d'entrée était "0", il est donc impossible de déterminer si la conversion a réellement échoué.

La nouvelle famille de fonctions `std::sto*` convertit `std::string` s en types entiers ou à virgule flottante et lance des exceptions si elles ne peuvent pas analyser leurs entrées. *Vous devriez utiliser ces fonctions si possible* :

C ++ 11

```
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

De plus, ces fonctions traitent également les chaînes octales et hexadécimales, contrairement à la famille `std::ato*` . Le deuxième paramètre est un pointeur sur le premier caractère non converti dans la chaîne d'entrée (non illustré ici), et le troisième paramètre est la base à utiliser. `0` est la détection automatique d'octal (commençant par `0`) et hexadécimal (commençant par `0x` ou `0X`), et toute autre valeur est la base à utiliser

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16);  // Returns 10
long num4 = std::stol(ten_hex);  // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Conversion entre les encodages de caractères

La conversion entre encodages est facile avec C ++ 11 et la plupart des compilateurs sont capables de la gérer de manière multi-plateforme via les en- `<codecvt>` et `<locale>` .

```
#include <iostream>
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Rappelez-vous que Visual Studio 2015 prend en charge ces conversions, mais qu'un bogue dans leur implémentation de bibliothèque nécessite l'utilisation d'un modèle différent pour `wstring_convert` lorsqu'il s'agit de `char16_t` :

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
  std::basic_string<utf16_char> tmp;
  tmp.resize(utf16.length());
  std::copy(utf16.begin(), utf16.end(), tmp.begin());
  utf8 = conv_utf8_utf16.to_bytes(tmp);
}
void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
  std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
  utf16.clear();
  utf16.resize(tmp.length());
  std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Vérifier si une chaîne est un préfixe d'un autre

C ++ 14

En C ++ 14, cela se fait facilement par `std::mismatch` qui renvoie la première paire incompatible entre deux plages:

```
std::string prefix = "foo";
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Notez que la version de `mismatch()` était antérieure à C ++ 14, mais elle est dangereuse dans le cas où la deuxième chaîne est la plus courte des deux.

C ++ 14

Nous pouvons toujours utiliser la version range-and-demi de `std::mismatch()` , mais nous devons d'abord vérifier que la première chaîne est au maximum aussi grande que la seconde:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

C ++ 17

Avec `std::string_view` , nous pouvons écrire la comparaison directe que nous voulons sans avoir à nous soucier de la surcharge de l'allocation ou de faire des copies:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Conversion en std :: string

`std::ostringstream` peut être utilisé pour convertir n'importe quel type pouvant être traité en une chaîne, en insérant l'objet dans un objet `std::ostringstream` (avec l'opérateur d'insertion de flux `<<`), puis en convertissant l'intégralité de `std::ostringstream` en `std::string` .

Pour `int` par exemple:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Écrire votre propre fonction de conversion, le plus simple:

```
template<class T>
std::string toString(const T& x)
{
  std::ostringstream ss;
  ss << x;
  return ss.str();
}
```

fonctionne mais ne convient pas au code de performance critique.

Les classes définies par l'utilisateur peuvent implémenter l'opérateur d'insertion de flux si désiré:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

C ++ 11

Outre les flux, depuis C ++ 11, vous pouvez également utiliser la fonction `std::to_string` (et `std::to_wstring`) qui est surchargée pour tous les types fondamentaux et retourne la représentation sous forme de chaîne de son paramètre.

```
std::string s = to_string(0x12f3);  // after this the string s contains "4851"
```

Lire std :: string en ligne: https://riptutorial.com/fr/cplusplus/topic/488/std----string

Chapitre 124: std :: variant

Remarques

La variante remplace l' `union` brute. Il est de type sécurisé et sait quel type il est, et il construit et détruit soigneusement les objets qu'il contient.

Il n'est presque jamais vide: ce n'est que dans les cas où le remplacement de son contenu se produit et qu'il est impossible de le rétablir en toute sécurité qu'il se trouve dans un état vide.

Il se comporte un peu comme un `std::tuple` , et un peu comme un `std::optional` .

Utiliser `std::get` et `std::get_if` est généralement une mauvaise idée. La bonne réponse est généralement `std::visit` , qui vous permet de traiter toutes les possibilités tout de suite. `if constexpr` peut être utilisé dans la `visit` si vous devez `if constexpr` branche de votre comportement, plutôt que d'effectuer une série de vérifications à l'exécution qui dupliquent ce que la `visit` fera plus efficacement.

Examples

Utilisation de base std :: variant

Cela crée une variante (une union balisée) qui peut stocker un `int` ou une `string` .

```
std::variant< int, std::string > var;
```

Nous pouvons en stocker un des deux types:

```
var = "hello"s;
```

Et nous pouvons accéder au contenu via `std::visit` :

```
// Prints "hello\n":
visit( [](auto&& e) {
  std::cout << e << '\n';
}, var );
```

en passant dans un objet polymorphe lambda ou une fonction similaire.

Si nous sommes certains de savoir de quel type il s'agit, nous pouvons l'obtenir:

```
auto str = std::get<std::string>(var);
```

mais cela se produira si nous nous trompons. `get_if` :

```
auto* str  = std::get_if<std::string>(&var);
```

renvoie `nullptr` si vous vous trompez.

Les variantes ne garantissent aucune allocation de mémoire dynamique (autre que celle allouée par leurs types contenus). Seul un des types d'une variante y est stocké et dans de rares cas (impliquant des exceptions lors de l'attribution et aucun moyen sûr de revenir en arrière), la variante peut devenir vide.

Les variantes vous permettent de stocker plusieurs types de valeurs dans une variable de manière sûre et efficace. Ce sont fondamentalement des `union` intelligents et sûrs.

Créer des pointeurs de pseudo-méthode

Ceci est un exemple avancé.

Vous pouvez utiliser une variante pour un effacement léger.

```
template<class F>
struct pseudo_method {
  F f;
  // enable C++17 class type deduction:
  pseudo_method( F&& fin ):f(std::move(fin)) {}

  // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
  template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
  friend decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
    // var->*method returns a lambda that perfect forwards a function call,
    // behaving like a method pointer basically:
    return [&](auto&&...args)->decltype(auto) {
      // use visit to get the type of the variant:
      return std::visit(
        [&](auto&& self)->decltype(auto) {
          // decltype(x)(x) is perfect forwarding in a lambda:
          return method.f( decltype(self)(self), decltype(args)(args)... );
        },
        std::forward<Var>(var)
      );
    };
  }
};
```

Cela crée un type qui surcharge l' `operator->*` avec un `Variant` sur le côté gauche.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
  return decltype(self)(self).print( decltype(args)(args)... );
};
```

Maintenant, si nous avons 2 types chacun avec une méthode d' `print` :

```
struct A {
  void print( std::ostream& os ) const {
    os << "A";
```

```
  }
};
struct B {
  void print( std::ostream& os ) const {
    os << "B";
  }
};
```

Notez qu'ils sont des types non liés. Nous pouvons:

```
std::variant<A,B> var = A{};

(var->*print)(std::cout);
```

et il enverra l'appel directement à `A::print(std::cout)` pour nous. Si on initialise la `var` avec `B{}` , elle sera envoyée à `B::print(std::cout)` .

Si nous avons créé un nouveau type C:

```
struct C {};
```

puis:

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

ne parviendra pas à compiler, car il n'y a pas de `C.print(std::cout)` .

L'extension de ce qui précède permettrait de détecter et d'utiliser une `print` fonction libre, éventuellement avec l'utilisation de `if constexpr` dans la pseudo-méthode d' `print` .

Exemple actuel utilisant `boost::variant` à la place de `std::variant` .

Construire un `std :: variant`

Cela ne couvre pas les allocateurs.

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

Lire std :: variant en ligne: https://riptutorial.com/fr/cplusplus/topic/5239/std----variant

Chapitre 125: std :: vector

Introduction

Un vecteur est un tableau dynamique avec un stockage géré automatiquement. Les éléments d'un vecteur peuvent être accédés aussi efficacement que ceux d'un tableau, avec l'avantage que les vecteurs peuvent changer dynamiquement de taille.

En termes de stockage, les données vectorielles sont (généralement) placées dans une mémoire allouée dynamiquement, ce qui nécessite une surcharge mineure; à l'inverse, `C-arrays` et `std::array` utilisent le stockage automatique par rapport à l'emplacement déclaré et n'ont donc pas de surcharge.

Remarques

L'utilisation d'un `std::vector` nécessite l'inclusion de l'en `<vector>` tête `<vector>` en utilisant `#include <vector>` .

Les éléments d'un `std::vector` sont stockés de manière contiguë sur le free store. Il convient de noter que lorsque les vecteurs sont imbriqués comme dans `std::vector<std::vector<int> >` , les éléments de chaque vecteur sont contigus, mais chaque vecteur alloue son propre tampon sous-jacent sur le free store.

Examples

Initialisation d'un std :: vector

Un `std::vector` peut être initialisé de plusieurs façons en le déclarant:

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };  // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };     // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);  // v becomes {6, 6, 6}

std::vector<int> v(4);     // v becomes {0, 0, 0, 0}
```

Un vecteur peut être initialisé à partir d'un autre conteneur de plusieurs manières:

Copier la construction (à partir d'un autre vecteur uniquement), qui copie les données de `v2` :

```
std::vector<int> v(v2);
```

```
std::vector<int> v = v2;
```

C ++ 11

Déplacer la construction (à partir d'un autre vecteur uniquement), ce qui déplace les données de `v2` :

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Iterator (range) copy-construction, qui copie des éléments dans `v` :

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}

// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3);                   // v becomes {1, 2, 3}

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

C ++ 11

Iterator move-construction, utilisant `std::make_move_iterator` , qui déplace les éléments dans `v` :

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end());

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

A l'aide de la fonction membre `assign()` , un `std::vector` peut être réinitialisé après sa construction:

```
v.assign(4, 100);                      // v becomes {100, 100, 100, 100}

v.assign(v2.begin(), v2.begin() + 3);  // v becomes {v2[0], v2[1], v2[2]}

int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4);                // v becomes {2, 3, 4}
```

Insertion d'éléments

Ajout d'un élément à la fin d'un vecteur (par copie / déplacement):

```
struct Point {
  double x, y;
  Point(double x, double y) : x(x), y(y) {}
};
```

```
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p);  // p is copied into the vector.
```

C ++ 11

Ajout d'un élément à la fin d'un vecteur en construisant l'élément en place:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Notez que `std::vector` n'a *pas de* fonction membre `push_front()` pour des raisons de performances. L'ajout d'un élément au début entraîne le déplacement de tous les éléments existants dans le vecteur. Si vous souhaitez insérer fréquemment des éléments au début de votre conteneur, vous pouvez utiliser plutôt `std::list` ou `std::deque` .

Insertion d'un élément à n'importe quelle position d'un vecteur:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9);          // v now contains {9, 1, 2, 3}
```

C ++ 11

Insertion d'un élément à n'importe quelle position d'un vecteur en construisant l'élément en place:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9);     // v now contains {1, 9, 2, 3}
```

Insertion d'un autre vecteur à n'importe quelle position du vecteur:

```
std::vector<int> v(4);      // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Insérer un tableau à n'importe quelle position d'un vecteur:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Utilisez `reserve()` avant d'insérer plusieurs éléments si la taille de vecteur résultante est connue à l'avance pour éviter les réallocations multiples (voir taille et capacité du vecteur):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
```

```
    v.emplace_back(i);
```

Veillez à ne pas commettre l'erreur d'appeler `resize()` dans ce cas, ou vous créez par inadvertance un vecteur avec 200 éléments, dont seuls les cent derniers auront la valeur souhaitée.

Itération sur std :: vector

Vous pouvez parcourir un `std::vector` de plusieurs manières. Pour chacune des sections suivantes, `v` est défini comme suit:

```
std::vector<int> v;
```

Itération dans la direction avant

C ++ 11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);

// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

C ++ 11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Itération dans le sens inverse

C ++ 14

```
// There is no standard way to use range based for for this.
// See below for alternatives.

// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Bien qu'il n'y ait pas de moyen intégré d'utiliser la plage pour renverser l'itération; il est relativement simple de résoudre ce problème. La plage basée sur les utilisations `begin()` et `end()` pour obtenir des itérateurs et simuler cela avec un objet wrapper peut atteindre les résultats requis.

C ++ 14

```
template<class C>
struct ReverseRange {
  C c; // could be a reference or a copy, if the original was a temporary
  ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
  ReverseRange(ReverseRange&&)=default;
  ReverseRange& operator=(ReverseRange&&)=delete;
  auto begin() const {return std::rbegin(c);}
  auto end()   const {return std::rend(c);}
};
// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Application d'éléments const

Depuis C ++ 11, les `cbegin()` et `cend()` vous permettent d'obtenir un *itérateur constant* pour un

vecteur, même si le vecteur est non-const. Un itérateur constant vous permet de lire mais pas de modifier le contenu du vecteur, ce qui est utile pour appliquer la correction const:

C ++ 11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

C ++ 17

as_const étend cette itération à l'intervalle:

```
for (auto const& e : std::as_const(v)) {
  std::cout << e << '\n';
}
```

Ceci est facile à implémenter dans les versions antérieures de C ++:

C ++ 14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
  return t;
}
```

Une note sur l'efficacité

Puisque la classe `std::vector` est essentiellement une classe qui gère un tableau contigu alloué dynamiquement, le même principe expliqué ici s'applique aux vecteurs C ++. L'accès au contenu du vecteur par index est beaucoup plus efficace lorsque l'on suit le principe de l'ordre des lignes principales. Bien entendu, chaque accès au vecteur met également son contenu de gestion dans le cache, mais comme cela a été débattu plusieurs fois (notamment ici et ici), la différence de performances pour itérer sur un `std::vector` par rapport à un tableau brut est négligeable. Ainsi , le même principe d'efficacité pour les tableaux bruts en C applique également pour l 'C ++ `std::vector` .

Accéder aux éléments

Il y a deux manières principales d'accéder aux éléments dans un `std::vector`

- accès par index
- itérateurs

Accès par index:

Cela peut être fait soit avec l'opérateur subscript `[]` , soit avec la fonction membre `at()` .

Les deux renvoient une référence à l'élément à la position respective dans le `std::vector` (à moins que ce ne soit un `vector<bool>`), de sorte qu'il puisse être lu et modifié (si le vecteur n'est pas `const`).

`[]` et `at()` diffèrent par le fait que `[]` ne garantit aucune vérification des limites, alors que `at()` fait. L'accès aux éléments où `index < 0` ou `index >= size` est un comportement indéfini pour `[]` , alors que `at()` génère une exception `std::out_of_range` .

Remarque: Les exemples ci-dessous utilisent l'initialisation de style C ++ 11 pour plus de clarté, mais les opérateurs peuvent être utilisés avec toutes les versions (sauf si marqué C ++ 11).

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
```

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Comme la méthode `at()` effectue la vérification des limites et peut générer des exceptions, elle est plus lente que `[]` . Cela rend `[]` code préféré où la sémantique de l'opération garantit que l'index est dans les limites. Dans tous les cas, les accès aux éléments des vecteurs se font à temps constant. Cela signifie que l'accès au premier élément du vecteur a le même coût (en temps) d'accès au deuxième élément, au troisième élément, etc.

Par exemple, considérez cette boucle

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Ici, nous savons que la variable d'index `i` est toujours dans les limites, donc ce serait un gaspillage de cycles CPU pour vérifier que `i` dans les limites pour chaque appel à l' `operator[]` .

Les fonctions membres `front()` et `back()` permettent respectivement un accès de référence au premier et au dernier élément du vecteur. Ces positions sont fréquemment utilisées et les accesseurs spéciaux peuvent être plus lisibles que leurs alternatives en utilisant `[]` :

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose

int a = v.front();   // a is 4, v.front() is equivalent to v[0]
v.front() = 3;       // v now contains {3, 5, 6}
int b = v.back();    // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7;        // v now contains {3, 5, 7}
```

Note : C'est un comportement indéfini pour appeler `front()` ou `back()` sur un vecteur vide. Vous devez vérifier que le conteneur n'est pas vide à l'aide de la fonction membre `empty()` qui vérifie si le conteneur est vide avant d'appeler `front()` ou `back()` . Voici un exemple simple d'utilisation de 'empty ()' pour tester un vecteur vide:

```
int main ()
{
  std::vector<int> v;
  int sum (0);

  for (int i=1;i<=10;i++) v.push_back(i);//create and initialize the vector

  while (!v.empty())//loop through until the vector tests to be empty
  {
     sum += v.back();//keep a running total
     v.pop_back();//pop out the element which removes it from the vector
  }

  std::cout << "total: " << sum << '\n';//output the total to the user

  return 0;
}
```

L'exemple ci-dessus crée un vecteur avec une séquence de nombres de 1 à 10. Puis il extrait les éléments du vecteur jusqu'à ce que le vecteur soit vide (en utilisant 'empty ()') pour empêcher un comportement indéfini. Ensuite, la somme des nombres dans le vecteur est calculée et affichée pour l'utilisateur.

C ++ 11

La méthode `data()` renvoie un pointeur sur la mémoire brute utilisée par `std::vector` pour stocker ses éléments en interne. Ceci est le plus souvent utilisé lors du passage des données vectorielles au code hérité qui attend un tableau de style C.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4;            // v now contains {4, 2, 3, 4}
++p;               // p points to 2
*p = 3;            // v now contains {4, 3, 3, 4}
p[1] = 2;          // v now contains {4, 3, 2, 4}
*(p + 2) = 1;      // v now contains {4, 3, 2, 1}
```

C ++ 11

Avant C ++ 11, la méthode `data()` peut être simulée en appelant `front()` et en prenant l'adresse de la valeur renvoyée:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

Cela fonctionne parce que les vecteurs sont toujours garantis pour stocker leurs éléments dans des emplacements de mémoire contigus, en supposant que le contenu du vecteur ne remplace pas l' `operator&` unaire. Si c'est le cas, vous devrez `std::addressof` en pré-C ++ 11. Cela suppose également que le vecteur n'est pas vide.

Itérateurs:

Les itérateurs sont expliqués plus en détail dans l'exemple "Itération sur `std::vector` " et les itérateurs d' article. En bref, ils agissent de manière similaire aux pointeurs vers les éléments du vecteur:

C ++ 11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;        // i is 4
++it;
i = *it;            // i is 5
*it = 6;            // v contains { 4, 6, 6 }
auto e = v.end();   // e points to the element after the end of v. It can be
                    // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();      // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();      // true
```

Il est cohérent avec le standard que les itérateurs de `std::vector<T>` *soient en* réalité des `T*` , mais la plupart des bibliothèques standard ne le font pas. Ne pas le faire améliore à la fois les messages d'erreur, intercepte le code non portable et peut être utilisé pour instrumenter les itérateurs avec des contrôles de débogage dans les versions sans publication. Ensuite, dans les versions release, la classe qui entoure le pointeur sous-jacent est optimisée.

Vous pouvez conserver une référence ou un pointeur sur un élément d'un vecteur pour un accès indirect. Ces références ou pointeurs vers les éléments du `vector` restent stables et l'accès reste défini à moins que vous ajoutiez / supprimiez des éléments à ou avant l'élément dans le `vector` , ou que vous ne modifiiez la capacité du `vector` . C'est la même chose que la règle pour invalider les itérateurs.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;     // p points to 2
```

```
v.insert(v.begin(), 0);     // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;           // p points to 1
v.reserve(10);              // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;           // p points to 1
v.erase(v.begin());         // p is now invalid, accessing *p is a undefined behavior.
```

Utiliser std :: vector comme un tableau C

Il existe plusieurs manières d'utiliser un `std::vector` comme un tableau C (par exemple, pour la compatibilité avec les bibliothèques C). Cela est possible car les éléments d'un vecteur sont stockés de manière contiguë.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

Contrairement aux solutions basées sur les normes C ++ précédentes (voir ci-dessous), la fonction membre `.data()` peut également être appliquée aux vecteurs vides, car elle ne provoque pas de comportement indéfini dans ce cas.

Avant C ++ 11, vous prendriez l'adresse du premier élément du vecteur pour obtenir un pointeur équivalent, si le vecteur n'est pas vide, ces deux méthodes sont interchangeables:

```
int* p = &v[0];      // combine subscript operator and 0 literal

int* p = &v.front(); // explicitly reference the first element
```

Remarque: Si le vecteur est vide, `v[0]` et `v.front()` sont pas définis et ne peuvent pas être utilisés.

Lorsque vous stockez l'adresse de base des données du vecteur, notez que de nombreuses opérations (telles que `push_back` , `resize` , etc.) peuvent modifier l'emplacement de la mémoire de données du vecteur, ce qui invalide les pointeurs de données précédents . Par exemple:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Invalidation de l'itérateur / pointeur

Les itérateurs et les pointeurs pointant dans un `std::vector` peuvent devenir invalides, mais uniquement lors de l'exécution de certaines opérations. L'utilisation d'itérateurs / pointeurs non valides entraînera un comportement indéfini.

Les opérations qui invalident les itérateurs / pointeurs incluent:

- Toute opération d’insertion modifiant la `capacity` du `vector` invalidera *tous les* itérateurs / pointeurs:

```
vector<int> v(5); // Vector has a size of 5; capacity is unknown.
int *p1 = &v[0];
v.push_back(2);   // p1 may have been invalidated, since the capacity was unknown.

v.reserve(20);    // Capacity is now at least 20.
int *p2 = &v[0];
v.push_back(4);   // p2 is *not* invalidated, since the size of `v` is now 7.
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the
                          // requested capacity of 20, so `p2` is (probably) invalidated.
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

C ++ 11

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // Iterators were invalidated.
```

- Toute opération d'insertion, qui n'augmente pas la capacité, invalidera toujours les itérateurs / pointeurs pointant vers des éléments à la position d'insertion et au-delà. Cela inclut l'itérateur `end` :

```
vector<int> v(5);
v.reserve(20);                 // Capacity is at least 20.
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity
                               // did not change, `p1` remains valid.
int *p3 = &v[v.size() - 1];
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Toute opération de suppression invalidera les itérateurs / pointeurs pointant vers les éléments supprimés et vers tous les éléments au-delà des éléments supprimés. Cela inclut l'itérateur `end` :

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (copy, move ou else) et `clear()` invalideront tous les itérateurs / pointeurs pointant dans le vecteur.

Supprimer des éléments

Supprimer le dernier élément:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back();                           // v becomes {1, 2}
```

Supprimer tous les éléments:

```
std::vector<int> v{ 1, 2, 3 };
v.clear();                              // v becomes an empty vector
```

Supprimer un élément par index:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3);                 // v becomes {1, 2, 3, 5, 6}
```

Remarque: pour un `vector` supprimant un élément qui n'est pas le dernier élément, tous les éléments au-delà de l'élément supprimé doivent être copiés ou déplacés pour combler le vide, voir la note ci-dessous et std :: list .

Supprimer tous les éléments d'une plage:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5);  // v becomes {1, 6}
```

Remarque: Les méthodes ci-dessus ne modifient pas la capacité du vecteur, mais uniquement la taille. Voir Taille et capacité du vecteur .

La méthode d' `erase` , qui supprime une série d'éléments, est souvent utilisée dans le cadre de l'idiome d' **effacement-suppression** . C'est-à-dire que `std::remove` déplace d'abord certains éléments à la fin du vecteur, puis les `erase` . Ceci est une opération relativement inefficace pour tous les indices inférieurs au dernier index du vecteur car tous les éléments après les segments effacés doivent être déplacés vers de nouvelles positions. Pour les applications critiques qui nécessitent une suppression efficace d'éléments arbitraires dans un conteneur, consultez std :: list .

Suppression d'éléments par valeur:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Suppression d'éléments par condition:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
```

```
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Supprimer des éléments par lambda, sans créer de fonction de prédicat supplémentaire

C ++ 11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
     [](auto& element){return element > 3;} ), v.end()
);
```

Suppression d'éléments par condition à partir d'une boucle:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
    else
        ++it;             // manually set 'it' to the next element in v
}
```

Bien qu'il soit important de *ne pas* augmenter , `it` en cas de suppression, vous devriez envisager d' utiliser une autre méthode lors de l' effacement puis à plusieurs reprises dans une boucle. Considérez `remove_if` pour une manière plus efficace.

Suppression d'éléments par condition à partir d'une boucle inverse:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
```

```
    } else
        ++it;
}
```

Notez quelques points pour la boucle précédente:

- Compte tenu d' un itérateur inverse , `it` en montrant un élément, la méthode de `base` donne le pointage régulier (non-retour) iterator au même élément.
- `vector::erase(iterator)` efface l'élément pointé par un itérateur et renvoie un itérateur à l'élément qui suit l'élément donné.
- `reverse_iterator::reverse_iterator(iterator)` construit un itérateur inverse à partir d'un itérateur.

Mettez tout à fait, la ligne `it = rev_itr(v.erase(it.base()))` dit: prendre l'itérateur inverse `it` , ont `v` d' effacement de l'élément pointé par son iterator régulière; prendre l'itérateur résultant, construire un itérateur inverse de celui - ci, et l' affecter à l'itérateur inverse `it` .

La suppression de tous les éléments à l'aide de `v.clear()` ne libère pas de mémoire (`capacity()` du vecteur reste inchangé). Pour récupérer de l'espace, utilisez:

```
std::vector<int>().swap(v);
```

C ++ 11

`shrink_to_fit()` libère la capacité vectorielle inutilisée:

```
v.shrink_to_fit();
```

Le `shrink_to_fit` ne garantit pas vraiment de récupérer de l'espace, mais la plupart des implémentations actuelles le font.

Recherche d'un élément dans std :: vector

La fonction `std::find` , définie dans l’en-tête `<algorithm>` , peut être utilisée pour trouver un élément dans un `std::vector` .

`std::find` utilise l' `operator==` pour comparer les éléments pour l'égalité. Il renvoie un itérateur au premier élément de la plage qui est égal à la valeur.

Si l'élément en question n'est pas trouvé, `std::find` renvoie `std::vector::end` (ou `std::vector::cend` si le vecteur est `const`).

C ++ 11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );
```

```
std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

C ++ 11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Si vous devez effectuer de nombreuses recherches dans un grand vecteur, vous pouvez envisager de trier d'abord le vecteur avant d'utiliser l'algorithme `binary_search` .

Pour trouver le premier élément dans un vecteur qui remplit une condition, `std::find_if` peut être utilisé. En plus des deux paramètres donnés à `std::find` , `std::find_if` accepte un troisième argument qui est un objet fonction ou un pointeur de fonction vers une fonction de prédicat. Le prédicat doit accepter un élément du conteneur en tant qu'argument et renvoyer une valeur convertible en `bool` sans modifier le conteneur:

C ++ 11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

C ++ 11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Conversion d'un tableau en std :: vector

Un tableau peut facilement être converti en un `std::vector` en utilisant `std::begin` et `std::end` :

C ++ 11

```
int values[5] = { 1, 2, 3, 4, 5 }; // source array

std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector

for(auto &x: v)
    std::cout << x << " ";
std::cout << std::endl;
```

> 1 2 3 4 5

```
int main(int argc, char* argv[]) {
    // convert main arguments into a vector of strings.
    std::vector<std::string>  args(argv, argv + argc);
}
```

Une liste_initialisation C ++ 11 <> peut également être utilisée pour initialiser le vecteur à la fois

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

vecteur : L'exception à tant de règles, tant de règles

La norme (section 23.3.7) spécifie qu’une spécialisation du `vector<bool>` est fournie, qui optimise l’espace en empaquetant les valeurs `bool` , de sorte que chacune ne prenne qu’un seul bit. Comme les bits ne sont pas adressables en C ++, cela signifie que plusieurs exigences sur le `vector` ne sont pas placées sur le `vector<bool>` :

- Les données stockées ne doivent pas nécessairement être contiguës, donc un `vector<bool>` ne peut pas être transmis à une API C qui attend un tableau `bool` .
- `at()` , `operator []` et dereferencing des itérateurs ne renvoient pas de référence à `bool` . Au lieu de cela, ils renvoient un objet proxy qui simule (imparfaitement) une référence à un `bool` en surchargeant ses opérateurs d'affectation. Par exemple, le code suivant peut ne pas être valide pour `std::vector<bool>` , car le déréférencement d'un itérateur ne renvoie pas de référence:

C ++ 11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

De même, les fonctions qui attendent un `bool&` argument ne peuvent pas être utilisées avec le résultat de l' `operator []` ou `at()` appliqué au `vector<bool>` , ou avec le résultat du déréférencement de son itérateur:

```
  void f(bool& b);
  f(v[0]);             // error
  f(*v.begin());       // error
```

L'implémentation de `std::vector<bool>` dépend à la fois du compilateur et de l'architecture. La spécialisation est implémentée en plaçant `n` Booleans dans la section adressable la plus basse. Ici, `n` est la taille en bits de la mémoire adressable la plus basse. Dans la plupart des systèmes modernes, il s'agit d'un octet ou de 8 bits. Cela signifie qu'un octet peut stocker 8 valeurs booléennes. C'est une amélioration par rapport à l'implémentation traditionnelle où 1 valeur booléenne est stockée dans 1 octet de mémoire.

Remarque: L'exemple ci-dessous montre les valeurs binaires possibles des octets individuels dans un `vector<bool>` traditionnel vs optimisé `vector<bool>` . Cela ne sera pas toujours vrai dans toutes les architectures. C'est toutefois un bon moyen de visualiser l'optimisation. Dans les exemples ci-dessous, un octet est représenté par [x, x, x, x, x, x, x, x].

*`std::vector<char>` **traditionnel** stockant 8 valeurs booléennes:*

C ++ 11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Représentation binaire:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

*`std::vector<bool>` **spécialisé** stockant 8 valeurs booléennes:*

C ++ 11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Représentation binaire:

```
[1,0,0,0,1,0,1,1]
```

Notez que dans l'exemple ci-dessus, dans la version traditionnelle de `std::vector<bool>` , 8 valeurs booléennes occupent 8 octets de mémoire, alors que dans la version optimisée de `std::vector<bool>` , elles n'utilisent que 1 octet de mémoire. Mémoire. Il s'agit d'une amélioration significative de l'utilisation de la mémoire. Si vous devez transmettre un `vector<bool>` à une API de

style C, vous devrez peut-être copier les valeurs dans un tableau ou trouver un meilleur moyen d'utiliser l'API si la mémoire et les performances sont menacées.

Taille et capacité du vecteur

La taille du vecteur est simplement le nombre d'éléments dans le vecteur:

1. La **taille** actuelle du vecteur est interrogée par la fonction membre `size()` . La fonction Convenience `empty()` renvoie `true` si la taille est 0:

```
vector<int> v = { 1, 2, 3 }; // size is 3
const vector<int>::size_type size = v.size();
cout << size << endl; // prints 3
cout << boolalpha << v.empty() << endl; // prints false
```

2. Le vecteur construit par défaut commence par une taille de 0:

```
vector<int> v; // size is 0
cout << v.size() << endl; // prints 0
```

3. L'ajout de `N` éléments à vector augmente la **taille** de `N` (par exemple, par les fonctions `push_back()` , `insert()` ou `resize()`).

4. La suppression de `N` éléments du vecteur diminue la **taille** de `N` (par exemple par les fonctions `pop_back()` , `erase()` ou `clear()`).

5. Vector a une limite supérieure de taille spécifique à l'implémentation, mais vous risquez de manquer de RAM avant de l'atteindre:

```
vector<int> v;
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Erreur commune: la **taille** n'est pas nécessairement (ou même généralement) `int` :

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

La capacité vectorielle diffère de la **taille** . Alors que la **taille** est simplement le nombre d'éléments dont dispose actuellement le vecteur, la **capacité** correspond au nombre d'éléments alloués / réservés à la mémoire. Cela est utile, car une (ré) attribution trop fréquente de trop grandes tailles peut être coûteuse.

1. La **capacité** actuelle du vecteur est interrogée par la fonction membre `capacity()` . **La capacité** est toujours supérieure ou égale à la **taille** :

```
vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
const vector<int>::size_type capacity = v.capacity();
cout << capacity << endl; // prints number >= 3
```

2. Vous pouvez réserver manuellement la capacité par fonction de `reserve( N )` (elle change la capacité vectorielle en `N`):

```
// !!!bad!!!evil!!!
vector<int> v_bad;
for( int i = 0; i < 10000; ++i ) {
    v_bad.push_back( i ); // possibly lot of reallocations
}

// good
vector<int> v_good;
v_good.reserve( 10000 ); // good! only one allocation
for( int i = 0; i < 10000; ++i ) {
    v_good.push_back( i ); // no allocations needed anymore
}
```

3. Vous pouvez demander que la capacité excédentaire soit libérée par `shrink_to_fit()` mais l'implémentation n'a pas à vous obéir. Ceci est utile pour conserver la mémoire utilisée:

```
vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but
possibly false)
```

Vector gère en partie automatiquement la capacité, lorsque vous ajoutez des éléments, elle peut décider de croître. Les implémenteurs aiment utiliser 2 ou 1,5 pour le facteur de croissance (le nombre d'or serait la valeur idéale - mais peu pratique en raison du nombre rationnel). D'un autre côté, le vecteur ne diminue généralement pas automatiquement. Par exemple:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!

v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Vecteurs concaténants

Un `std::vector` peut être ajouté à un autre en utilisant la fonction membre `insert()` :

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

Cependant, cette solution échoue si vous essayez d’ajouter un vecteur à lui-même, car le standard spécifie que les itérateurs donnés à `insert()` ne doivent pas appartenir à la même plage que les éléments de l’objet récepteur.

c ++ 11

Au lieu d'utiliser les fonctions membres du vecteur, les fonctions `std::begin()` et `std::end()` peuvent être utilisées:

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

C'est une solution plus générale, par exemple, car `b` peut également être un tableau. Cependant, cette solution ne vous permet pas non plus d'ajouter un vecteur à lui-même.

Si l'ordre des éléments dans le vecteur de réception n'a pas d'importance, compte tenu du nombre d'éléments dans chaque vecteur peut éviter des opérations de copie inutiles:

```
if (b.size() < a.size())
  a.insert(a.end(), b.begin(), b.end());
else
  b.insert(b.end(), a.begin(), a.end());
```

Réduire la capacité d'un vecteur

Un `std::vector` augmente automatiquement sa capacité au moment de l'insertion, mais ne réduit jamais sa capacité après la suppression de l'élément.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
v.erase(v.begin() + 50, v.end());  // Reduces the size from 100 to 50 (v.size() == 50),
                                   // but not the capacity (v.capacity() == old_capacity)
```

Pour réduire sa capacité, nous pouvons copier le contenu d'un vecteur dans un nouveau vecteur temporaire. Le nouveau vecteur aura la capacité minimale nécessaire pour stocker tous les éléments du vecteur d'origine. Si la réduction de taille du vecteur d'origine était significative, la réduction de capacité pour le nouveau vecteur sera probablement significative. Nous pouvons alors échanger le vecteur d'origine avec le vecteur temporaire pour conserver sa capacité minimale:

```
std::vector<int>(v).swap(v);
```

C ++ 11

En C ++ 11, nous pouvons utiliser la fonction membre `shrink_to_fit()` pour un effet similaire:

```
v.shrink_to_fit();
```

Remarque: La fonction membre `shrink_to_fit()` est une demande et ne garantit pas la réduction de la capacité.

Utilisation d'un vecteur trié pour la recherche rapide d'éléments

L' `<algorithm>` tête `<algorithm>` fournit un certain nombre de fonctions utiles pour travailler avec des vecteurs triés.

Une condition préalable importante pour travailler avec des vecteurs triés est que les valeurs stockées sont comparables à `<` .

Un vecteur non trié peut être trié en utilisant la fonction `std::sort()` :

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Les vecteurs `std::lower_bound()` permettent une recherche efficace des éléments en utilisant la fonction `std::lower_bound()` . Contrairement à `std::find()` , cette fonction effectue une recherche binaire efficace sur le vecteur. L'inconvénient est qu'il ne donne que des résultats valides pour les plages d'entrées triées:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Remarque: Si la valeur demandée ne fait pas partie du vecteur, `std::lower_bound()` renverra un itérateur au premier élément *supérieur* à la valeur demandée. Ce comportement nous permet d'insérer un nouvel élément à sa place dans un vecteur déjà trié:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

Si vous devez insérer beaucoup d'éléments à la fois, il peut être plus efficace d'appeler `push_back()` pour tous les éléments et d'appeler ensuite `std::sort()` une fois tous les éléments insérés. Dans ce cas, le coût accru du tri peut compenser le coût réduit de l'insertion de nouveaux éléments à la fin du vecteur et non au milieu.

Si votre vecteur contient plusieurs éléments de la même valeur, `std::lower_bound()` essaiera de renvoyer un itérateur au premier élément de la valeur recherchée. Cependant, si vous devez insérer un nouvel élément *après* le dernier élément de la valeur recherchée, vous devez utiliser la fonction `std::upper_bound()` car cela entraînera moins de décalage des éléments:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

Si vous avez besoin à la fois des itérateurs de limite supérieure et inférieure, vous pouvez utiliser la fonction `std::equal_range()` pour les récupérer efficacement avec un seul appel:

```
std::pair<std::vector<int>::iterator,
          std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

Pour tester si un élément existe dans un vecteur trié (bien que non spécifique aux vecteurs), vous pouvez utiliser la fonction `std::binary_search()` :

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Fonctions de retour de grands vecteurs

C ++ 11

En C ++ 11, les compilateurs doivent se déplacer implicitement d'une variable locale renvoyée. De plus, la plupart des compilateurs peuvent effectuer une élimination de la copie dans de nombreux cas et éviter tout mouvement. En conséquence, le retour d'objets volumineux pouvant être déplacés à moindre coût ne nécessite plus de manipulation particulière:

```
#include <vector>
#include <iostream>

// If the compiler is unable to perform named return value optimization (NRVO)
// and elide the move altogether, it is required to move from v into the return value.
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}

int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // print vector
    for (auto value : vec)
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "

    std::cout << std::endl;

    return 0;
}
```

C ++ 11

Avant C ++ 11, l'élision des copies était déjà autorisée et implémentée par la plupart des compilateurs. Cependant, en raison de l'absence de sémantique de déplacement, dans le code ou le code hérité qui doit être compilé avec les anciennes versions du compilateur qui n'implémentent

pas cette optimisation, vous pouvez trouver des vecteurs transmis comme arguments de sortie pour empêcher la copie inutile:

```
#include <vector>
#include <iostream>

// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() {// declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}
```

Rechercher max et min Element et index respectif dans un vecteur

Pour trouver le plus grand ou le plus petit élément stocké dans un vecteur, vous pouvez utiliser les méthodes `std::max_element` et `std::min_element` , respectivement. Ces méthodes sont définies dans l' `<algorithm>` tête `<algorithm>` . Si plusieurs éléments sont équivalents au plus grand (plus petit) élément, les méthodes renvoient l'itérateur au premier élément de ce type. Renvoie `v.end()` pour les vecteurs vides.

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(),v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(),v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Sortie:

> maxElementIndex: 3, maxElement: 10
> minElementIndex: 1, minElement: 2

C ++ 11

Les éléments minimum et maximum dans un vecteur peuvent être récupérés en même temps en utilisant la méthode `std::minmax_element` , qui est également définie dans l' `<algorithm>` tête `<algorithm>`

:

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Sortie:

> élément minimum: 2
> élément maximum: 10

Matrices utilisant des vecteurs

Les vecteurs peuvent être utilisés comme une matrice 2D en les définissant comme un vecteur de vecteurs.

Une matrice à 3 lignes et 4 colonnes avec chaque cellule initialisée à 0 peut être définie comme suit:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

C ++ 11

La syntaxe permettant de les initialiser à l'aide de listes d'initialisation ou autres est similaire à celle d'un vecteur normal.

```
  std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                           {4,5,6,7},
                                           {8,9,10,11}
                                         };
```

Les valeurs dans un tel vecteur peuvent être accédées de la même manière qu'un tableau 2D

```
int var = matrix[0][2];
```

Itérer sur toute la matrice est similaire à celui d'un vecteur normal mais avec une dimension supplémentaire.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

C ++ 11

```
for(auto& row: matrix)
```

```
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

Un vecteur de vecteurs est un moyen pratique de représenter une matrice, mais ce n'est pas le plus efficace: les vecteurs individuels sont dispersés dans la mémoire et la structure de données n'est pas compatible avec le cache.

De plus, dans une matrice appropriée, la longueur de chaque ligne doit être la même (ce n’est pas le cas pour un vecteur de vecteurs). La flexibilité supplémentaire peut être source d'erreurs.

Lire std :: vector en ligne: https://riptutorial.com/fr/cplusplus/topic/511/std----vector

Chapitre 126: Structures de données en C ++

Examples

Implémentation de la liste liée en C ++

Création d'un nœud de liste

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Créer une classe de liste

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
    ~List();
};
```

Insérer un nouveau noeud au début de la liste

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Insérer un nouveau noeud à la fin de la liste

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
```

```
    {
        ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

Insérer à une position particulière dans la liste

```
void List::insertAtPos(int pos,int val)
{
    listNode *newnode=new listNode(val);
    if(pos==1)
    {
        //as head
        newnode->next=this->head;
        this->head=newnode;
        return;
    }
    pos--;
    listNode *ptr=this->head;
    while(ptr!=NULL && --pos)
    {
        ptr=ptr->next;
    }
    if(ptr==NULL)
    return;//not enough elements
    newnode->next=ptr->next;
    ptr->next=newnode;
}
```

Supprimer un noeud de la liste

```
void List::remove(int toBeRemoved)//removing an element
{
    if(this->head==NULL)
    return; //empty
    if(this->head->data==toBeRemoved)
    {
        //first node to be removed
        listNode *temp=this->head;
        this->head=this->head->next;
        delete(temp);
        return;
    }
    listNode *ptr=this->head;
    while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
    ptr=ptr->next;
    if(ptr->next==NULL)
    return;//not found
    listNode *temp=ptr->next;
    ptr->next=ptr->next->next;
    delete(temp);
}
```

Imprimer la liste

```
void List::print()//printing the list
```

```
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" " ;
        ptr=ptr->next;
    }
    cout<<endl;
}
```

Destructeur pour la liste

```
 List::~List()
{
    listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

Lire Structures de données en C ++ en ligne:
https://riptutorial.com/fr/cplusplus/topic/7485/structures-de-donnees-en-c-plusplus

Chapitre 127: Structures de synchronisation de fil

Introduction

Travailler avec des threads peut nécessiter des techniques de synchronisation si les threads interagissent. Dans cette rubrique, vous pouvez trouver les différentes structures fournies par la bibliothèque standard pour résoudre ces problèmes.

Examples

std :: shared_lock

Un shared_lock peut être utilisé conjointement avec un verrou unique pour autoriser plusieurs lecteurs et écrivains exclusifs.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
    public:
        string getPhoneNo( const std::string & name )
        {
            shared_lock<shared_timed_mutex> r(_protect);
            auto it =  _phonebook.find( name );
            if ( it == _phonebook.end() )
                return (*it).second;
            return "";
        }
        void addPhoneNo ( const std::string & name, const std::string & phone )
        {
            unique_lock<shared_timed_mutex> w(_protect);
            _phonebook[name] = phone;
        }

        shared_timed_mutex _protect;
        unordered_map<string,string>  _phonebook;
    };
```

std :: call_once, std :: once_flag

`std::call_once` assure l'exécution d'une fonction exactement une fois par des threads concurrents. Il jette `std::system_error` au cas où il ne pourrait pas terminer sa tâche.

Utilisé conjointement avec s `td::once_flag` .

```
#include <mutex>
#include <iostream>

std::once_flag flag;
void do_something(){
      std::call_once(flag, [](){std::cout << "Happens once" << std::endl;});

      std::cout << "Happens every time" << std::endl;
}
```

Verrouillage d'objet pour un accès efficace.

Vous voulez souvent verrouiller l’objet entier pendant que vous effectuez plusieurs opérations sur celui-ci. Par exemple, si vous devez examiner ou modifier l'objet à l'aide d' *itérateurs* . Chaque fois que vous avez besoin d'appeler plusieurs fonctions membres, il est généralement plus efficace de verrouiller l'objet entier plutôt que les fonctions membres individuelles.

Par exemple:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

public:
    // This returns a scoped lock that can be shared by multiple
    // readers at the same time while excluding any writers
    [[nodiscard]]
    reading_lock lock_for_reading() const { return reading_lock(mtx); }

    // This returns a scoped lock that is exclusing to one
    // writer preventing any readers
    [[nodiscard]]
    updates_lock lock_for_updates() { return updates_lock(mtx); }

    char* data() { return buf; }
    char const* data() const { return buf; }

    char* begin() { return buf; }
    char const* begin() const { return buf; }

    char* end() { return buf + sizeof(buf); }
    char const* end() const { return buf + sizeof(buf); }

    std::size_t size() const { return sizeof(buf); }

private:
    char buf[1024];
    mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

Lors du calcul d'une somme de contrôle, l'objet est verrouillé pour la lecture, permettant ainsi aux autres threads qui veulent lire à partir de l'objet en même temps.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
    auto lock = buf.lock_for_reading();

    for(auto c: buf)
        sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

    return sum;
}
```

L'effacement de l'objet met à jour ses données internes. Vous devez donc utiliser un verrou exclusif.

```
void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}
```

Lors de l'obtention de plusieurs verrous, veillez à toujours acquérir les verrous dans le même ordre pour tous les threads.

```
void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}
```

note: il vaut mieux utiliser std :: deferred :: lock et appeler std :: lock

std :: condition_variable_any, std :: cv_status

Une généralisation de `std::condition_variable` , `std::condition_variable_any` fonctionne avec tout type de structure BasicLockable.

`std::cv_status` comme statut de retour pour une variable de condition a deux codes de retour possibles:

- std :: cv_status :: no_timeout: Il n'y avait pas de délai d'attente, la variable de condition a été notifiée
- std :: cv_status :: no_timeout: La variable de condition a expiré

Lire Structures de synchronisation de fil en ligne: https://riptutorial.com/fr/cplusplus/topic/9794/structures-de-synchronisation-de-fil

Chapitre 128: Surcharge de l'opérateur

Introduction

En C ++, il est possible de définir des opérateurs tels que `+` et `->` pour les types définis par l'utilisateur. Par exemple, l'en-tête <string> définit un opérateur `+` pour concaténer des chaînes. Cela se fait en définissant une *fonction d'opérateur* à l'aide du mot-clé `operator` .

Remarques

Les opérateurs pour les types intégrés ne peuvent pas être modifiés, les opérateurs ne peuvent être surchargés que pour les types définis par l'utilisateur. C'est-à-dire qu'au moins l'un des opérandes doit être d'un type défini par l'utilisateur.

Les opérateurs suivants *ne peuvent pas* être surchargés:

- L'accès membre ou l'opérateur "point" `.`
- Le pointeur sur l'opérateur d'accès membre `.*`
- L'opérateur de résolution de portée, `::`
- L'opérateur conditionnel ternaire, `?:`
- `dynamic_cast` , `static_cast` , `reinterpret_cast` , `const_cast` , `typeid` , `sizeof` , `alignof` et `noexcept`
- Les directives de prétraitement `#` et `##` qui sont exécutées avant toute information de type sont disponibles.

Certains opérateurs **ne** doivent **pas** (99,98% du temps) surcharger:

- `&&` et `||` (préférez plutôt utiliser la conversion implicite en `bool`)
- `,`
- L'adresse de l'opérateur (unaire `&`)

Pourquoi? Parce qu'ils surchargent les opérateurs auxquels un autre programmeur ne s'attend jamais, entraînant un comportement différent de celui anticipé.

Par exemple, l'utilisateur a défini `&&` et `||` ces opérateurs de surcharge perdent leur évaluation de , court-circuit et perdent leurs propriétés de séquençage spéciales (17 C ++) , la question de séquençage applique également `,` les surcharges de l' opérateur.

Examples

Opérateurs arithmétiques

Vous pouvez surcharger tous les opérateurs arithmétiques de base:

- `+` et `+=`
- `-` et `-=`
- `*` et `*=`

- `/` et `/=`
- `&` et `&=`
- `|` et `|=`
- `^` et `^=`
- `>>` et `>>=`
- `<<` et `<<=`

La surcharge pour tous les opérateurs est la même. *Faites défiler pour une explication*

Surcharge en dehors de `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Surcharge à l'intérieur de `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Note: `operator+` devrait retourner par une valeur non-const, car renvoyer une référence n'aurait pas de sens (elle renvoie un *nouvel* objet) et ne renverrait pas une valeur `const` (vous ne devriez généralement pas retourner par `const`). Le premier argument est passé par valeur, pourquoi? Car

1. Vous ne pouvez pas modifier l'objet d'origine (`Object foobar = foo + bar;` ne devrait pas modifier `foo` après tout, cela n'aurait aucun sens)
2. Vous ne pouvez pas le faire `const` , parce que vous devez être en mesure de modifier l'objet (car l' `operator+` est mis en œuvre en termes d' `operator+=` , qui modifie l'objet)

Passer par `const&` serait une option, mais vous devrez alors faire une copie temporaire de l'objet passé. En passant par valeur, le compilateur le fait pour vous.

`operator+=` renvoie une référence à lui-même, car il est alors possible de les enchaîner (n'utilisez cependant pas la même variable, ce serait un comportement indéfini dû à des points de séquence).

Le premier argument est une référence (nous voulons le modifier), mais pas `const` , car alors vous ne pourriez pas le modifier. Le second argument ne doit pas être modifié, et donc, pour des raisons de performances, il est passé par `const&` (le passage par la référence const est plus rapide que par valeur).

Opérateurs unaires

Vous pouvez surcharger les 2 opérateurs unaires:

- `++foo` et `foo++`
- `--foo` et `foo--`

La surcharge est la même pour les deux types (`++` et `--`). *Faites défiler pour une explication*

Surcharge en dehors de `class` / `struct` :

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Surcharge à l'intérieur de `class` / `struct` :

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Remarque: L'opérateur de préfixe renvoie une référence à lui-même, afin que vous puissiez continuer à y effectuer des opérations. Le premier argument est une référence, car l'opérateur de préfixe change l'objet, c'est aussi la raison pour laquelle il n'est pas `const` (vous ne pourriez pas le modifier autrement).

L'opérateur postfix renvoie par valeur une valeur temporaire (la valeur précédente) et ne peut donc pas être une référence, car il s'agirait d'une référence à une valeur temporaire, qui serait inutile à la fin de la fonction, car la variable temporaire s'éteint. de portée). Il ne peut pas non plus être `const` , car vous devriez pouvoir le modifier directement.

Le premier argument est une référence non `const` à l'objet "calling", car s'il était `const` , vous ne seriez pas en mesure de le modifier, et si ce n'était pas une référence, vous ne modifieriez pas la valeur d'origine.

C'est à cause de la copie nécessaire dans les surcharges d'opérateur postfix qu'il vaut mieux prendre l'habitude d'utiliser prefix ++ au lieu de postfix ++ dans `for` loops. Du point de vue de la boucle `for` , ils sont généralement fonctionnellement équivalents, mais l'utilisation du préfixe ++, en particulier avec les "grosses" classes avec beaucoup de membres à copier, peut présenter un léger avantage. Exemple d'utilisation de prefix ++ dans une boucle for:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Opérateurs de comparaison

Vous pouvez surcharger tous les opérateurs de comparaison:

- `==` et `!=`
- `>` et `<`
- `>=` et `<=`

La méthode recommandée pour surcharger tous ces opérateurs consiste à implémenter uniquement 2 opérateurs (`==` et `<`), puis à les utiliser pour définir le reste. *Faites défiler pour une explication*

Surcharge en dehors de `class` / `struct` :

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Surcharge à l'intérieur de `class` / `struct` :

```
//Note that the functions are const, because if they are not const, you wouldn't be able
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

Les opérateurs renvoient évidemment un `bool` , indiquant `true` ou `false` pour l'opération correspondante.

Tous les opérateurs prennent leurs arguments par `const&` , car la seule chose que font les opérateurs est de comparer, donc ils ne doivent pas modifier les objets. Passer par `&` (référence) est plus rapide que par valeur, et pour s'assurer que les opérateurs ne le modifient pas, il s'agit d'une référence `const` .

Notez que les opérateurs à l'intérieur de la `class` / `struct` sont définis comme `const` , la raison en est que sans les fonctions `const` , la comparaison des objets `const` ne serait pas possible, car le compilateur ne sait pas que les opérateurs ne modifient rien.

Opérateurs de conversion

Vous pouvez surcharger les opérateurs de type afin que votre type puisse être implicitement converti dans le type spécifié.

L'opérateur de conversion **doit** être défini dans une `class` / `struct` :

```
operator T() const { /* return something */ }
```

Remarque: l'opérateur est `const` pour permettre la conversion des objets `const` .

Exemple:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
    // ^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";
```

```
//Ok
const char* copyoftext = t;
```

Opérateur d'indice de tableau

Vous pouvez même surcharger l'opérateur d'index de tableau `[]` .

Vous devez **toujours** (99,98% du temps) implémenter 2 versions, une version `const` et une version non `const` , car si l'objet est `const` , il ne devrait pas pouvoir modifier l'objet renvoyé par `[]` .

Les arguments sont passés par `const&` non par valeur car le passage par référence est plus rapide que par valeur et `const` pour que l'opérateur ne modifie pas l'index accidentellement.

Les opérateurs renvoient par référence, car de par leur conception, vous pouvez modifier le retour de l'objet `[]` , à savoir:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

Vous ne pouvez surcharger **que** dans une `class` / `struct` :

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Plusieurs opérateurs d'indice, `[][]...` , peuvent être obtenus via des objets proxy. L'exemple suivant d'une classe de matrice simple de rangée majeure montre ceci:

```
template<class T>
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
```

```
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Opérateur d'appel de fonction

Vous pouvez surcharger l'opérateur d'appel de fonction `()` :

La surcharge doit être faite à l'intérieur d'une `class` / `struct` :

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
    //return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

Par exemple:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};
```

```
//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Opérateur d'assignation

L'opérateur d'affectation est l'un des opérateurs les plus importants car il vous permet de modifier le statut d'une variable.

Si vous ne surchargez pas l'opérateur d'affectation pour votre `class` / `struct` , il est automatiquement généré par le compilateur: l'opérateur d'affectation généré automatiquement exécute une "attribution membre", c'est-à-dire en invoquant des opérateurs d'affectation sur chaque membre. à l'autre, un membre à la fois. L'opérateur d'affectation doit être surchargé lorsque l'attribution simple par membres n'est pas adaptée à votre `class` / `struct` , par exemple si vous devez effectuer une **copie approfondie** d'un objet.

La surcharge de l'opérateur d'affectation `=` est facile, mais vous devez suivre quelques étapes simples.

1. **Test d'auto-affectation.** Ce contrôle est important pour deux raisons:
    - une auto-affectation est une copie inutile, donc cela n'a aucun sens de l'exécuter;
    - l'étape suivante ne fonctionnera pas dans le cas d'une auto-affectation.
2. **Nettoyez les anciennes données.** Les anciennes données doivent être remplacées par de nouvelles. Maintenant, vous pouvez comprendre la deuxième raison de l'étape précédente: si le contenu de l'objet a été détruit, une auto-affectation échouera à effectuer la copie.
3. **Copiez tous les membres.** Si vous surchargez l'opérateur d'assignation pour votre `class` ou votre `struct` , il n'est pas généré automatiquement par le compilateur. Vous devrez donc vous charger de copier tous les membres de l'autre objet.
4. **Retourner** `*this` . L'opérateur retourne par lui-même par référence, car il permet le chaînage (ie `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Note: `other` est passé par `const&` , car l'objet assigné ne doit pas être modifié, et le passage par référence est plus rapide que par valeur, et pour s'assurer `operator=` ne le modifie pas accidentellement, c'est `const` .

L'opérateur d'affectation ne **peut** être surchargé que dans la `class` / `struct` , car la valeur de gauche de `=` est **toujours** la `class` / `struct` elle `struct` même. Le définir comme une fonction libre n'a pas cette garantie et est interdit à cause de cela.

Lorsque vous le déclarez dans la `class` / `struct` , la valeur de gauche est implicitement la `class` / `struct` elle `struct` même, donc cela ne pose aucun problème.

Opérateur binaire NON

Surcharger le bit bit NOT (`~`) est assez simple. *Faites défiler pour une explication*

Surcharge en dehors de `class` / `struct` :

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Surcharge à l'intérieur de `class` / `struct` :

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Note: `operator~` renvoie par valeur, car il doit renvoyer une nouvelle valeur (la valeur modifiée), et non une référence à la valeur (ce serait une référence à l'objet temporaire, qui aurait une valeur de mémoire dès que l'opérateur est fait). Non `const` soit parce que le code d'appel devrait pouvoir le modifier par la suite (par exemple `int a = ~a + 1;` devrait être possible).

A l' intérieur de la `class` / `struct` vous devez faire un objet temporaire, parce que vous ne pouvez pas modifier `this` , car il modifierait l'objet original, qui ne devrait pas être le cas.

Opérateurs de décalage de bits pour E / S

Les opérateurs `<<` et `>>` sont couramment utilisés comme opérateurs "write" et "read":

- `std::ostream` surcharges `<<` pour écrire des variables dans le flux sous-jacent (exemple: `std::cout`)
- `std::istream` surcharge `>>` pour lire depuis le flux sous-jacent vers une variable (exemple: `std::cin`)

La façon dont ils le font est similaire si vous voulez les surcharger "normalement" en dehors de la `class` / `struct` , sauf que spécifier les arguments ne sont pas du même type:

- Le type de retour est le flux que vous voulez surcharger (par exemple, `std::ostream`) passé par référence, pour permettre le chaînage (Chaînage: `std::cout << a << b;`). Exemple: `std::ostream&`
- `lhs` serait le même que le type de retour
- `rhs` est le type que vous voulez autoriser à surcharger (c'est-à-dire `T`), passé par `const&` au lieu de valeur pour des raisons de performances (`rhs` ne doit pas être modifié de toute façon). Exemple: `const Vector&` .

Exemple:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Nombres complexes revisités

Le code ci-dessous implémente un type de nombre complexe très simple pour lequel le champ sous-jacent est automatiquement promu, suivant les règles de promotion du type du langage, sous l'application des quatre opérateurs de base (+, -, * et /) avec un membre d'un champ différent (que ce soit un autre `complex<T>` ou un type scalaire).

Ceci est destiné à être un exemple global couvrant la surcharge de l'opérateur, parallèlement à l'utilisation de base des modèles.

```
#include <type_traits>

namespace not_std{

using std::decay_t;

//----------------------------------------------------------------
// complex< value_t >
//----------------------------------------------------------------

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
```

```
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}
    {}

    template<typename other_value_t>
    explicit complex(const complex<other_value_t> &other)
    : x{static_cast<const value_t &>(other.x)}
    , y{static_cast<const value_t &>(other.y)}
    {}

    complex &operator = (const complex &) = default;
    complex &operator = (complex &&) = default;
    complex(const complex &) = default;
    complex(complex &&) = default;
    complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//-------------------------------------------------------------------
// operator - (negation)
//-------------------------------------------------------------------

template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//-------------------------------------------------------------------
```

```
// operator +
//------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//------------------------------------------------------------------
// operator -
//------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//------------------------------------------------------------------
// operator *
//------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
        };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }
```

```
//------------------------------------------------------------------
// operator /
//------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
        };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
         b.x * s,
        -b.y * s
        };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std


int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};

    // All of these are complex<double>
    auto c0 = a + b;
    auto c1 = a - b;
    auto c2 = a * b;
```

```
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}
```

Opérateurs nommés

Vous pouvez étendre C ++ avec des opérateurs nommés qui sont "cités" par des opérateurs C ++ standard.

Nous commençons par une bibliothèque de douzaines de lignes:

```
namespace named_operator {
  template<class D>struct make_operator{constexpr make_operator(){}};

  template<class T, char, class O> struct half_apply { T&& lhs; };

  template<class Lhs, class Op>
  half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
    return {std::forward<Lhs>(lhs)};
  }

  template<class Lhs, class Op, class Rhs>
  auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
  -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
  {
    return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
  }
}
```

cela ne fait rien encore.

Premièrement, ajouter des vecteurs

```
namespace my_ns {
  struct append_t : named_operator::make_operator<append_t> {};
  constexpr append_t append{};
```

```
  template<class T, class A0, class A1>
  std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const&
rhs ) {
      lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
      return std::move(lhs);
  }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

auto c = a *append* b;
```

Le noyau ici est que nous définissons un objet `append` de type `append_t:named_operator::make_operator<append_t>` .

Nous surchargons alors named_invoke (lhs, append_t, rhs) pour les types que nous voulons à droite et à gauche.

La bibliothèque surcharge `lhs*append_t` , renvoyant un objet `half_apply` temporaire. Il surcharge également `half_apply*rhs` pour appeler `named_invoke( lhs, append_t, rhs )` .

Nous devons simplement créer le jeton `append_t` approprié et effectuer un `named_invoke` la signature ADL-friendly `named_invoke` de la signature appropriée, et tout se `named_invoke` et fonctionne.

Pour un exemple plus complexe, supposons que vous souhaitiez une multiplication d'éléments des éléments d'un tableau std ::

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
  return [](auto&& f) {
    return f( std::integral_constant<std::size_t, Is>{}... );
  };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
  struct e_times_t : named_operator::make_operator<e_times_t> {};
  constexpr e_times_t e_times{};

  template<class L, class R, std::size_t N,
    class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
  >
  std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N>
const& rhs ) {
    using result_type = std::array<Out, N>;
    auto index_over_N = indexer<N>();
    return index_over_N([&](auto...is)->result_type {
      return {{
        (lhs[is] * rhs[is])...
      }};
    });
  }
}
```

exemple vivant .

Ce code de tableau élémentaire peut être étendu pour fonctionner sur des tuples ou des paires ou des tableaux de style C, ou même des conteneurs de longueur variable si vous décidez quoi faire si les longueurs ne correspondent pas.

Vous pouvez également saisir un type d'opérateur élémentaire et obtenir `lhs *element_wise<'+'>* rhs` .

L'écriture d'opérateurs de produit `*dot*` et `*cross*` est également une utilisation évidente.

L'utilisation de `*` peut être étendue pour prendre en charge d'autres délimiteurs, comme `+` . La précision des délimiteurs détermine la précision de l'opérateur nommé, ce qui peut être important lors de la traduction des équations de physique en C ++ avec une utilisation minimale des extra `()` .

Avec un léger changement dans la bibliothèque ci-dessus, nous pouvons prendre en charge les opérateurs `->*then*` et étendre `std::function` avant la mise à jour du standard, ou écrire monadic `->*bind*` . Il pourrait également avoir un opérateur nommé avec état, où nous passerions soigneusement la `Op` à la fonction d'invocation finale, permettant ainsi:

```
named_operator<'*'> append = [](auto lhs, auto&& rhs) {
  using std::begin; using std::end;
  lhs.insert( end(lhs), begin(rhs), end(rhs) );
  return std::move(lhs);
};
```

générer un opérateur nommé contenant le conteneur en C ++ 17.

Lire Surcharge de l'opérateur en ligne: https://riptutorial.com/fr/cplusplus/topic/562/surcharge-de-l-operateur

Chapitre 129: Surcharge du modèle de fonction

Remarques

- Une fonction normale n'est jamais liée à un modèle de fonction, malgré le même nom, même type.
- Un appel de fonction normal et un appel de modèle de fonction généré sont différents même s'ils partagent le même nom, le même type de retour et la même liste d'arguments

Examples

Qu'est-ce qu'une surcharge de modèle de fonction valide?

Un modèle de fonction peut être surchargé selon les règles de surcharge de la fonction non-modèle (même nom, mais différents types de paramètres) et en plus, la surcharge est valide si

- Le type de retour est différent ou
- La liste des paramètres du modèle est différente, sauf pour la désignation des paramètres et la présence d'arguments par défaut (ils ne font pas partie de la signature)

Pour une fonction normale, la comparaison de deux types de paramètres est facile pour le compilateur, car il contient tous les informat. Mais un type dans un modèle peut ne pas encore être déterminé. Par conséquent, la règle à appliquer lorsque deux types de paramètres sont égaux est approximative et indique que les types et valeurs non dépendants doivent correspondre et que l’orthographe des types et expressions dépendants doit être la même (plus précisément, ils doivent être conformes à la règles ODR), sauf que les paramètres du modèle peuvent être renommés. Cependant, si des orthographes si différentes, deux valeurs dans les types sont considérées différentes, mais instancient toujours aux mêmes valeurs, la surcharge est invalide, mais aucun diagnostic n'est requis du compilateur.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

C'est une surcharge valide, car "T" et "T *" sont des orthographes différentes. Mais ce qui suit est invalide, sans diagnostic requis

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }

template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Lire Surcharge du modèle de fonction en ligne:
https://riptutorial.com/fr/cplusplus/topic/4164/surcharge-du-modele-de-fonction

Chapitre 130: Systèmes de construction

Introduction

C ++, comme C, a une histoire longue et variée concernant les workflows de compilation et les processus de compilation. Aujourd'hui, C ++ dispose de plusieurs systèmes de construction populaires utilisés pour compiler des programmes, parfois pour plusieurs plates-formes au sein d'un même système de construction. Ici, quelques systèmes de construction seront examinés et analysés.

Remarques

Actuellement, il n'existe pas de système de génération universel ou dominant pour C ++, à la fois populaire et multi-plateforme. Cependant, il existe plusieurs systèmes de construction majeurs attachés aux principales plates-formes / projets, le plus notable étant GNU Make avec le système d'exploitation GNU / Linux et NMAKE avec le système de projet Visual C ++ / Visual Studio.

De plus, certains environnements de développement intégrés (IDE) incluent également des systèmes de construction spécialisés à utiliser spécifiquement avec l'EDI natif. Certains générateurs de système de génération peuvent générer ces formats de système / projet de génération IDE natifs, tels que CMake pour Eclipse et Microsoft Visual Studio 2012.

Examples

Générer un environnement de construction avec CMake

CMake génère des environnements de construction pour presque tous les compilateurs ou IDE à partir d'une seule définition de projet. Les exemples suivants montreront comment ajouter un fichier CMake au code C ++ interplate -forme "Hello World" .

Les fichiers CMake sont toujours nommés "CMakeLists.txt" et devraient déjà exister dans le répertoire racine de chaque projet (et éventuellement dans les sous-répertoires). Un fichier CMakeLists.txt de base ressemble à ceci:

```
cmake_minimum_required(VERSION 2.4)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

Voyez-le vivre sur Coliru .

Ce fichier indique à CMake le nom du projet, la version du fichier à attendre et les instructions pour générer un exécutable appelé "HelloWorld" nécessitant `main.cpp` .

Générez un environnement de compilation pour votre compilateur / IDE installé à partir de la ligne

de commande:

```
> cmake .
```

Construisez l'application avec:

```
> cmake --build .
```

Cela génère l'environnement de génération par défaut pour le système, en fonction du système d'exploitation et des outils installés. Gardez le code source propre de tous les artefacts de construction avec l'utilisation des versions "out-of-source":

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake peut également abstraire les commandes de base du shell de plate-forme de l'exemple précédent:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake inclut des générateurs pour un certain nombre d'outils de construction et d'EDI courants. Pour générer des makefiles pour `nmake` Visual Studio :

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Compiler avec GNU make

introduction

GNU Make (styled `make`) est un programme dédié à l'automatisation de l'exécution des commandes shell. GNU Make est un programme spécifique de la famille Make. Make reste populaire parmi les systèmes d'exploitation de type Unix ou de type POSIX, y compris ceux dérivés du noyau Linux, Mac OS X et BSD.

GNU Make est particulièrement remarquable pour être associé au projet GNU, qui est attaché au système d’exploitation GNU / Linux. GNU Make dispose également de versions compatibles exécutant différentes versions de Windows et de Mac OS X. Il s’agit également d’une version très stable dont l’importance historique reste populaire. C'est pour ces raisons que GNU Make est souvent enseigné avec C et C ++.

Règles de base

Pour compiler avec make, créez un Makefile dans votre répertoire de projet. Votre Makefile peut être aussi simple que:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called
without target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
    @$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
    @rm -f $(EXE)
```

> **REMARQUE: Assurez-vous absolument que les indentations sont avec un onglet, pas avec quatre espaces. Sinon, vous aurez une erreur de** `Makefile:10: *** missing separator. Stop.`

Pour l'exécuter depuis la ligne de commande, procédez comme suit:

```
$ cd ~/Path/to/project
$ make
$ ls
app  main.cpp  Makefile

$ ./app
Hello World!
```

```
$ make clean
$ ls
main.cpp  Makefile
```

Constructions incrémentielles

Lorsque vous commencez à avoir plus de fichiers, make devient plus utile. Et si vous **modifiiez a.cpp** mais pas **b.cpp** ? Recompiler **b.cpp** prendrait plus de temps.

Avec la structure de répertoires suivante:

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

Ce serait un bon Makefile:

Makefile

```
CXX=g++
CXXFLAGS=-Wall -Wextra -pedantic
EXE=app

SRCS_GLOB=src/*.cpp
SRCS=$(wildcard $(SRCS_GLOB))
OBJS=$(SRCS:.cpp=.o)

all: $(EXE)

$(EXE): $(OBJS)
    @$(CXX) -o $@ $(OBJS)

depend: .depend

.depend: $(SRCS)
    @-rm -f ./.depend
    @$(CXX) $(CXXFLAGS) -MM $^>>./.depend

clean:
    -rm -f $(EXE)
    -rm $(OBJS)
    -rm *~
    -rm .depend

include .depend
```

Regardez à nouveau les onglets. Ce nouveau Makefile garantit que vous ne recompilez que les fichiers modifiés, minimisant ainsi le temps de compilation.

Documentation

Pour plus d'informations sur make, consultez la documentation officielle de la Free Software Foundation , la documentation de stackoverflow et la réponse élaborée de dmckee sur stackoverflow .

Construire avec des SCons

Vous pouvez créer le code C ++ "multiplateforme" "Hello World" , en utilisant Scons - Un outil de construction de logiciels en langage Python .

Tout d'abord, créez un fichier appelé `SConstruct` (notez que SCons recherchera un fichier avec ce nom exact par défaut). Pour l'instant, le fichier devrait être dans un répertoire tout au long de votre `hello.cpp` . Ecrire dans le nouveau fichier la ligne

```
Program('hello.cpp')
```

Maintenant, à partir du terminal, lancez les `scons` . Vous devriez voir quelque chose comme

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: done building targets.
```

(bien que les détails varient en fonction de votre système d'exploitation et du compilateur installé).

Les classes `Environment` et `Glob` vous aideront à configurer davantage les éléments à construire. Par exemple, le fichier `SConstruct`

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
    )

env.Program('hello', Glob('src/*.cpp'))
```

construit l'exécutable `hello` , en utilisant tous les fichiers `cpp` dans `src` . Son `CPPPATH` est `/usr/include/boost` et spécifie le standard C ++ 11.

Ninja

introduction

Le site Web de son projet décrit le système de construction Ninja comme «un petit système de construction axé sur la vitesse». Ninja est conçu pour que ses fichiers soient générés par des générateurs de fichiers système de génération, et adopte une approche de bas niveau pour créer des systèmes, contrairement aux gestionnaires de systèmes de génération de niveau supérieur comme CMake ou Meson.

Ninja est principalement écrit en C ++ et Python et a été créé comme alternative au système de construction SCons du projet Chromium.

NMAKE (utilitaire de maintenance de programme Microsoft)

introduction

NMAKE est un utilitaire de ligne de commande développé par Microsoft pour être utilisé principalement avec Microsoft Visual Studio et / ou les outils de ligne de commande Visual C ++.

NMAKE est un système de génération qui fait partie de la famille de systèmes de génération Make, mais possède certaines fonctionnalités distinctes des programmes Make de type Unix, comme la syntaxe de chemin de fichier spécifique à Windows (elle-même différente des chemins de fichier de style Unix).

Autotools (GNU)

introduction

Les Autotools sont un groupe de programmes qui créent un système de génération GNU pour un logiciel donné. C'est une suite d'outils qui fonctionnent ensemble pour produire diverses ressources de construction, telles qu'un fichier Makefile (à utiliser avec GNU Make). Autotools peut donc être considéré comme un générateur de système de facto.

Certains programmes Autotools notables incluent:

- Autoconf
- Automake (à ne pas confondre avec `make`)

En général, Autotools est conçu pour générer le script compatible Unix et Makefile pour permettre à la commande suivante de construire (ainsi que d'installer) la plupart des packages (dans le cas simple):

```
./configure && make && make install
```

En tant que tel, Autotools a également une relation avec certains gestionnaires de paquets, en particulier ceux qui sont attachés à des systèmes d'exploitation conformes aux normes POSIX.

Lire Systèmes de construction en ligne: https://riptutorial.com/fr/cplusplus/topic/8200/systemes-de-

construction

Chapitre 131: Tableaux

Introduction

Les tableaux sont des éléments du même type placés dans des emplacements mémoire adjacents. Les éléments peuvent être référencés individuellement par un identifiant unique avec un index ajouté.

Cela vous permet de déclarer plusieurs valeurs de variable d'un type spécifique et d'y accéder individuellement sans avoir à déclarer une variable pour chaque valeur.

Examples

Taille du tableau: tapez safe au moment de la compilation.

```
#include <stddef.h>     // size_t, ptrdiff_t

//----------------------------------- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
    -> Size
{ return n; }


//----------------------------------- Usage:

#include <iostream>
using namespace std;
auto main()
    -> int
{
    int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
    Size const  n       = n_items( a );
    int         b[n]    = {};       // An array of the same size as a.

    (void) b;
    cout << "Size = " << n << "\n";
}
```

Le langage C pour la taille du tableau, `sizeof(a)/sizeof(a[0])` acceptera un pointeur comme argument et donnera généralement un résultat incorrect.

Pour C ++ 11

en utilisant C ++ 11 vous pouvez faire:

```
std::extent<decltype(MyArray)>::value;
```

Exemple:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Jusqu'à C ++ 17 (à paraître) Le C ++ n'avait pas de langage de base intégré ni d'utilitaire de bibliothèque standard pour obtenir la taille d'un tableau, mais cela peut être implémenté en passant le tableau *par référence* à un modèle de fonction, comme montré ci-dessus. Point précis mais important: le paramètre de taille de modèle est un `size_t` , quelque peu incompatible avec le type de résultat de la fonction `Size` signée, afin de prendre en charge le compilateur g ++ qui insiste parfois sur `size_t` pour la correspondance de modèle.

Avec C ++ 17 et versions ultérieures, on peut utiliser `std::size` , qui est spécialisé pour les tableaux.

Tableau brut de taille dynamique

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }

auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';

    delete[] a;
}
```

Un programme qui déclare un tableau `T a[n];` où `n` est déterminé à l'exécution, peut être compilé avec certains compilateurs qui prennent en charge *les tableaux de longueurs variadiques* C99 (VLA) en tant qu'extension de langage. Mais les VLA ne sont pas supportés par le standard C ++. Cet exemple montre comment allouer manuellement un tableau de taille dynamique via une `new[]` expression `new[]` ,

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

… Alors utilisez-le et enfin désallouez-le via une expression `delete[]` :

```
delete[] a;
```

Le tableau affecté ici a des valeurs indéterminées, mais il peut être initialisé à zéro en ajoutant simplement une parenthèse vide `()` , comme ceci: `new int[n]()` . Plus généralement, pour un type d'article arbitraire, une *initialisation de valeur est effectuée* .

Dans le cadre d'une fonction dans une hiérarchie d'appels, ce code ne serait pas exempt d'exception, car une exception avant l'expression `delete[]` (et après la `new[]`) provoquerait une fuite de mémoire. Une façon de résoudre ce problème consiste à automatiser le nettoyage via, par exemple, un pointeur intelligent `std::unique_ptr` . Mais une meilleure façon de le résoudre consiste à utiliser simplement un `std::vector` : c'est ce que `std::vector` est là pour ça.

Extension du tableau de taille dynamique en utilisant std :: vector.

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` est un modèle de classe de bibliothèque standard qui fournit la notion d'un tableau de taille variable. Il prend en charge toute la gestion de la mémoire, et le tampon est contigu donc un pointeur vers le tampon (par exemple, `&v[0]` ou `v.data()`) peut être transmis aux fonctions API nécessitant un tableau brut. Un `vector` peut même être développé au moment de l'exécution, par exemple via la fonction membre `push_back` qui ajoute un élément.

La complexité de la séquence de *n* opérations `push_back` , y compris la copie ou le déplacement impliqué dans les extensions vectorielles, est amortie O (*n*). «Amorti»: en moyenne.

En interne, ceci est généralement obtenu par le fait que le vecteur *double* sa taille de tampon, sa

capacité, lorsqu'un plus grand tampon est nécessaire. Par exemple, pour un tampon commençant à la taille 1 et doublé à plusieurs reprises selon les besoins pour $n = 17$ appels `push_back` , cela implique 1 + 2 + 4 + 8 + 16 = 31 opérations de copie, ce qui est inférieur à $2 \times n$. plus généralement la somme de cette séquence ne peut dépasser $2 \times n$.

Par rapport à l'exemple de tableau brut de taille dynamique, ce code `vector` ne nécessite pas que l'utilisateur fournisse (et connaisse) le nombre d'éléments à la fois. Au lieu de cela, le vecteur est simplement développé selon les besoins, pour chaque nouvelle valeur d'élément spécifiée par l'utilisateur.

Une matrice de matrice brute de taille fixe (c'est-à-dire un tableau brut 2D).

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Sortie:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

C ++ ne supporte pas la syntaxe spéciale pour indexer un tableau multidimensionnel. Au lieu de cela, un tel tableau est considéré comme un tableau de tableaux (éventuellement de tableaux, etc.), et la notation d'index simple ordinaire `[ i ]` est utilisée pour chaque niveau. Dans l'exemple ci-dessus, `m[y]` fait référence à la ligne `y` de `m` , où `y` est un index basé sur zéro. Ensuite , cette ligne peut être indexé à son tour, par exemple , `m[y][x]` , qui se réfère à la `x` ième élément - ou colonne - de la ligne `y` .

C'est-à-dire que le dernier index varie le plus rapidement, et dans la déclaration, la plage de cet index, qui est ici le nombre de colonnes par ligne, est la dernière et la plus petite taille spécifiée.

Comme C ++ ne prend pas en charge les tableaux de taille dynamique, hormis l'allocation dynamique, une matrice de taille dynamique est souvent implémentée en tant que classe. Alors, la notation d'indexation matricielle brute `m[y][x]` a un certain coût, soit en exposant l'implémentation (de sorte qu'une vue d'une matrice transposée devient pratiquement impossible) ou en ajoutant une surcharge et un léger inconvénient un objet proxy de l' `operator[]` . Et donc, la notation d'indexation pour une telle abstraction peut et sera généralement différente, à la fois en apparence et dans l'ordre des index, par exemple `m(x,y)` ou `m.at(x,y)` ou `m.item(x,y)` .

Une matrice de taille dynamique utilisant std :: vector pour le stockage.

Malheureusement, à partir de C ++ 14, il n'y a pas de classe de matrice de taille dynamique dans la bibliothèque standard C ++. Classes de matrice qui prennent en charge la taille dynamique sont toutefois disponibles à partir d' un certain nombre de 3 ^{e} bibliothèques du parti, y compris la bibliothèque Matrix Boost (une sous-bibliothèque au sein de la bibliothèque Boost).

Si vous ne voulez pas de dépendance sur Boost ou une autre bibliothèque, la matrice de taille dynamique d'un homme pauvre en C ++ est comme

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

… Où le `vector` est `std::vector` . La matrice est ici créée en copiant un vecteur de ligne *n* fois où *n* est le nombre de lignes, ici 3. Il a l'avantage de fournir la même notation d'indexation `m[y][x]` que pour une matrice de matrice brute de taille fixe, mais c'est un peu inefficace car il implique une allocation dynamique pour chaque ligne, et c'est un peu dangereux car il est possible de redimensionner une ligne par inadvertance.

Une approche plus sûre et efficace consiste à utiliser un seul vecteur en tant que *stockage* pour la matrice et à mapper le code client (*x* , *y*) sur un index correspondant dans ce vecteur:

```
// A dynamic size matrix using std::vector for storage.

//--------------------------------------------- Machinery:
#include <algorithm>        // std::copy
#include <assert.h>         // assert
#include <initializer_list> // std::initializer_list
#include <vector>           // std::vector
#include <stddef.h>         // ptrdiff_t

namespace my {
    using Size = ptrdiff_t;
    using std::initializer_list;
    using std::vector;

    template< class Item >
    class Matrix
    {
    private:
        vector<Item>    items_;
        Size            n_cols_;

        auto index_for( Size const x, Size const y ) const
            -> Size
```

```
        { return y*n_cols_ + x; }

    public:
        auto n_rows() const -> Size { return items_.size()/n_cols_; }
        auto n_cols() const -> Size { return n_cols_; }

        auto item( Size const x, Size const y )
            -> Item&
        { return items_[index_for(x, y)]; }

        auto item( Size const x, Size const y ) const
            -> Item const&
        { return items_[index_for(x, y)]; }

        Matrix(): n_cols_( 0 ) {}

        Matrix( Size const n_cols, Size const n_rows )
            : items_( n_cols*n_rows )
            , n_cols_( n_cols )
        {}

        Matrix( initializer_list< initializer_list<Item> > const& values )
            : items_()
            , n_cols_( values.size() == 0? 0 : values.begin()->size() )
        {
            for( auto const& row : values )
            {
                assert( Size( row.size() ) == n_cols_ );
                items_.insert( items_.end(), row.begin(), row.end() );
            }
        }
    };
}  // namespace my

//--------------------------------------------- Usage:
using my::Matrix;

auto some_matrix()
    -> Matrix<int>
{
    return
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };
}

#include <iostream>
#include <iomanip>
using namespace std;
auto main() -> int
{
    Matrix<int> const m = some_matrix();
    assert( m.n_cols() == 7 );
    assert( m.n_rows() == 3 );
    for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
    {
        for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
        {
            cout << setw( 4 ) << m.item( x, y );        // ← Note: not `m[y][x]`!
```

```
        }
        cout << '\n';
    }
}
```

Sortie:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

Le code ci-dessus n'est pas de qualité industrielle: il est conçu pour montrer les principes de base et répondre aux besoins des étudiants qui apprennent le C ++.

Par exemple, on peut définir des surcharges d' `operator()` pour simplifier la notation d'indexation.

Initialisation du tableau

Un tableau est juste un bloc d'emplacements de mémoire séquentiels pour un type de variable spécifique. Les tableaux sont alloués de la même manière que les variables normales, mais avec des crochets ajoutés à son nom `[]` qui contiennent le nombre d'éléments qui rentrent dans la mémoire du tableau.

L'exemple suivant d'un tableau utilise le type `int` , le nom de la variable `arrayOfInts` et le nombre d'éléments `[5]` que le tableau peut `arrayOfInts` :

```
int arrayOfInts[5];
```

Un tableau peut être déclaré et initialisé en même temps comme ceci

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

Lors de l'initialisation d'un tableau en listant tous ses membres, il n'est pas nécessaire d'inclure le nombre d'éléments à l'intérieur des crochets. Il sera automatiquement calculé par le compilateur. Dans l'exemple suivant, c'est 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

Il est également possible d'initialiser uniquement les premiers éléments tout en allouant plus d'espace. Dans ce cas, la définition de la longueur entre parenthèses est obligatoire. Ce qui suit va allouer un tableau de longueur 5 avec une initialisation partielle, le compilateur initialise tous les éléments restants avec la valeur standard du type d'élément, dans ce cas zéro.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Les tableaux d'autres types de données de base peuvent être initialisés de la même manière.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize
```

```
char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

Il est également important de noter que lors de l'accès aux éléments du tableau, l'index (ou la position) de l'élément du tableau commence à 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Lire Tableaux en ligne: https://riptutorial.com/fr/cplusplus/topic/3017/tableaux

Chapitre 132: Techniques de refactoring

Introduction

Le refactoring se réfère à la modification du code existant en une version améliorée. Bien que le refactoring soit souvent effectué lors de la modification du code pour ajouter des fonctionnalités ou corriger des bogues, le terme fait particulièrement référence à l'amélioration du code sans nécessairement ajouter des fonctionnalités ou corriger des bogues.

Examples

Refactoring à pied

Voici un programme qui pourrait bénéficier de la refactorisation. C'est un programme simple utilisant C ++ 11 qui est conçu pour calculer et imprimer tous les nombres premiers de 1 à 100 et est basé sur un programme qui a été publié sur CodeReview pour révision.

```
#include <iostream>
#include <vector>
#include <cmath>

int main()
{
    int l = 100;
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < l; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                std::cout << no << "\n";
                break;
            }
        }
        if (isprime) {
            std::cout << no << " ";
            primes.push_back(no);
        }
    }
    std::cout << "\n";
}
```

La sortie de ce programme ressemble à ceci:

> 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

La première chose que nous remarquons est que le programme ne parvient pas à imprimer `2` qui

est un nombre premier. Nous pourrions simplement ajouter une ligne de code pour simplement imprimer cette constante sans modifier le reste du programme, mais il serait peut-être préférable de *refactoriser* le programme pour le diviser en deux parties - une qui crée la liste de nombres premiers et une autre les imprime. . Voici comment cela pourrait ressembler:

```
#include <iostream>
#include <vector>
#include <cmath>

std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}

int main()
{
    std::vector<int> primes = prime_list(100);
    for (std::size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << ' ';
    }
    std::cout << '\n';
}
```

En essayant cette version, nous voyons que cela fonctionne effectivement correctement maintenant:

> 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

L'étape suivante consiste à noter que la deuxième clause `if` n'est pas vraiment nécessaire. La logique dans la boucle recherche les facteurs premiers de chaque nombre donné jusqu'à la racine carrée de ce nombre. Cela fonctionne parce que s'il y a des facteurs premiers d'un nombre, au moins l'un d'entre eux doit être inférieur ou égal à la racine carrée de ce nombre. Reprenant juste cette fonction (le reste du programme reste le même), nous obtenons ce résultat:

```
std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
```

```
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}
```

Nous pouvons aller plus loin, changer les noms de variables pour les rendre un peu plus descriptifs. Par exemple, `primecount` n'est pas vraiment un décompte de nombres premiers. Au lieu de cela, il s'agit d'une variable d'index dans le vecteur des nombres premiers connus. De plus, alors que `no` est parfois utilisé comme abréviation pour "numéro", en écriture mathématique, il est plus courant d'utiliser `n` . Nous pouvons également apporter certaines modifications en éliminant la `break` et en déclarant les variables plus proches de leur utilisation.

```
std::vector<int> prime_list(int limit)
{
    std::vector<int> primes{2};
    for (int n = 3; n < limit; n += 2) {
        bool isprime = true;
        for (int i=0; isprime && primes[i] <= std::sqrt(n); ++i) {
            isprime &= (n % primes[i] != 0);
        }
        if (isprime) {
            primes.push_back(n);
        }
    }
    return primes;
}
```

Nous pouvons aussi refactoriser `main` pour utiliser un "range-for" pour le rendre un peu plus net:

```
int main()
{
    std::vector<int> primes = prime_list(100);
    for (auto p : primes) {
        std::cout << p << ' ';
    }
    std::cout << '\n';
}
```

Ceci est juste un moyen de refactoring pourrait être fait. D'autres pourraient faire des choix différents. Cependant, le but du refactoring reste le même, à savoir améliorer la lisibilité et éventuellement les performances du code sans nécessairement ajouter de fonctionnalités.

Aller au nettoyage

Dans les bases de code C ++ qui étaient auparavant C, on peut trouver le modèle `goto cleanup` . Comme la commande `goto` rend le workflow d'une fonction plus difficile à comprendre, cela est souvent évité. Souvent, il peut être remplacé par des instructions de retour, des boucles, des fonctions. Cependant, avec le `goto cleanup` il faut se débarrasser de la logique de nettoyage.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
       goto cleanup;  //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

En C ++, on pourrait utiliser RAII pour résoudre ce problème:

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
       return FALSE;  //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

À partir de là, on pourrait continuer à refactoriser le code actuel. Par exemple en remplaçant le `VectorRAII` par `std::unique_ptr` ou `std::vector` .

Lire Techniques de refactoring en ligne: https://riptutorial.com/fr/cplusplus/topic/7600/techniques-de-refactoring

Chapitre 133: Test d'unité en C ++

Introduction

Le test unitaire est un niveau de test logiciel qui valide le comportement et l'exactitude des unités de code.

En C ++, les "unités de code" font souvent référence à des classes, des fonctions ou des groupes de l'un ou de l'autre. Les tests unitaires sont souvent effectués à l'aide de "frameworks de test" spécialisés ou de "bibliothèques de test" qui utilisent souvent des modes de syntaxe ou d'utilisation non triviaux.

Cette rubrique passera en revue différentes stratégies et bibliothèques ou frameworks de tests unitaires.

Examples

Test Google

Google Test est un framework de test C ++ géré par Google. Cela nécessite de construire la bibliothèque `gtest` et de la lier à votre structure de test lors de la création d'un fichier de `gtest` test.

Exemple minimal

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// Build command: g++ main.cpp -lgtest
```

Capture

Catch est une bibliothèque en-tête uniquement qui vous permet d'utiliser à la fois le style de test d'unité TDD et BDD .

L'extrait suivant provient de la page de documentation de Catch sur ce lien :

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Idéalement, ces tests seront rapportés comme suit lors de l'exécution:

```
Scenario: vectors can be sized and resized
     Given: A vector with some items
      When: more capacity is reserved
      Then: the capacity changes but not the size
```

Lire Test d'unité en C ++ en ligne: https://riptutorial.com/fr/cplusplus/topic/9928/test-d-unite-en-c-plusplus

Chapitre 134: Transmission parfaite

Remarques

La transmission parfaite nécessite le *transfert des références* afin de préserver les qualificateurs de référence des arguments. De telles références n'apparaissent que dans un *contexte déduit* . C'est:

```
template<class T>
void f(T&& x) // x is a forwarding reference, because T is deduced from a call to f()
{
    g(std::forward<T>(x)); // g() will receive an lvalue or an rvalue, depending on x
}
```

Ce qui suit n'implique pas un transfert parfait, car `T` n'est pas déduit de l'appel du constructeur:

```
template<class T>
struct a
{
    a(T&& x); // x is a rvalue reference, not a forwarding reference
};
```

C ++ 17

C ++ 17 autorisera la déduction des arguments du modèle de classe. Le constructeur de "a" dans l'exemple ci-dessus deviendra un utilisateur d'une référence de transfert

```
a example1(1);
  // same as a<int> example1(1);

int x = 1;
a example2(x);
  // same as a<int&> example2(x);
```

Examples

Fonctions d'usine

Supposons que nous voulions écrire une fonction de fabrique qui accepte une liste arbitraire d'arguments et transmet ces arguments sans modification à une autre fonction. Un exemple d'une telle fonction est `make_unique` , qui est utilisé pour construire en toute sécurité une nouvelle instance de `T` et renvoyer un `unique_ptr<T>` propriétaire de l'instance.

Les règles de langage concernant les modèles variadiques et les références rvalue nous permettent d’écrire une telle fonction.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
```

```
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

L'utilisation d'ellipses `...` indique un paquet de paramètres, qui représente un nombre arbitraire de types. Le compilateur étendra ce pack de paramètres au nombre d'arguments correct sur le site d'appel. Ces arguments sont ensuite transmis au constructeur de `T` utilisant `std::forward` . Cette fonction est nécessaire pour préserver les qualificateurs ref des arguments.

```
struct foo
{
    foo() {}
    foo(const foo&) {}                    // copy constructor
    foo(foo&&) {}                         // copy constructor
    foo(int, int, int) {}
};

foo f;
auto p1 = make_unique<foo>(f);            // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Lire Transmission parfaite en ligne: https://riptutorial.com/fr/cplusplus/topic/1750/transmission-parfaite

Chapitre 135: Tri

Remarques

La famille de fonctions `std::sort` se trouve dans la bibliothèque d' `algorithm` .

Examples

Tri des conteneurs de séquence avec un ordre spécifique

Si les opérateurs d'un conteneur sont déjà surchargés, `std::sort` peut être utilisé avec des foncteurs spécialisés pour trier dans l'ordre croissant ou décroissant:

C ++ 11

```
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

//sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

//sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

//Or just:
std::sort(v.rbegin(), v.rend());
```

C ++ 14

En C ++ 14, il n'est pas nécessaire de fournir l'argument de modèle pour les objets de fonction de comparaison et de laisser l'objet déduire en fonction de ce qu'il reçoit:

```
std::sort(v.begin(), v.end(), std::less<>());     // ascending order
std::sort(v.begin(), v.end(), std::greater<>());  // descending order
```

Tri des conteneurs de séquence par un opérateur moins surchargé

Si aucune fonction de `std::sort` n'est passée, `std::sort` ordonnera les éléments en appelant `operator<` sur des paires d'éléments, ce qui doit renvoyer un type convertible contextuellement en `bool` (ou simplement `bool`). Les types de base (entiers, flottants, pointeurs, etc.) ont déjà été créés dans des opérateurs de comparaison.

Nous pouvons surcharger cet opérateur pour que l'appel de `sort` par défaut fonctionne sur les

types définis par l'utilisateur.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    // Use variable to provide total order operator less
    //`this` always represents the left-hand side of the compare.
    bool operator<(const Base &b) const {
        return this->variable < b.variable;
    }

    int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using operator<(const Base &b) function
    std::sort(vector.begin(), vector.end());
    std::sort(deque.begin(), deque.end());
    // List must be sorted differently due to its design
    list.sort();

    return 0;
}
```

Tri des conteneurs de séquence à l'aide de la fonction de comparaison

```
// Include sequence containers
#include <vector>
#include <deque>
```

```
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
    std::sort(vector.begin(), vector.end(), compare);
    std::sort(deque.begin(), deque.end(), compare);
    list.sort(compare);

    return 0;
}
```

Tri des conteneurs de séquence à l'aide d'expressions lambda (C ++ 11)

C ++ 11

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>

// Include sorting algorithm
#include <algorithm>
```

```
class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};


int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector <Base> vector = {a, b};
    std::deque <Base> deque = {a, b};
    std::list <Base> list = {a, b};
    std::array <Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
      [](const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [](const Base &a, const Base &b) {
                     return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}
```

Tri et séquence des conteneurs

`std::sort` , trouvé dans l' `algorithm` tête de bibliothèque standard, est un algorithme de bibliothèque standard pour trier une plage de valeurs, définie par une paire d'itérateurs. `std::sort` prend comme dernier paramètre un foncteur utilisé pour comparer deux valeurs; c'est comme ça qu'il détermine la commande. Notez que `std::sort` n'est pas stable .

La fonction de comparaison *doit* imposer un ordre strict, faible sur les éléments. Une simple comparaison inférieure à (ou supérieure à) suffit.

Un conteneur avec des itérateurs à accès aléatoire peut être trié à l'aide de l'algorithme `std::sort` :

C ++ 11

```
#include <vector>
#include <algorithm>
```

```
std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` exige que ses itérateurs soient des itérateurs à accès aléatoire. Les conteneurs de séquence `std::list` et `std::forward_list` (nécessitant C ++ 11) ne fournissent pas d'itérateurs d'accès aléatoire, ils ne peuvent donc pas être utilisés avec `std::sort` . Cependant, ils ont des fonctions de membre de `sort` qui implémentent un algorithme de tri qui fonctionne avec leurs propres types d'itérateurs.

C ++ 11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

Leurs fonctions de `sort` membres `sort` toujours la liste entière, de sorte qu'elles ne peuvent pas trier un sous-ensemble d'éléments. Cependant, puisque `list` et `forward_list` ont des opérations d'épissage rapides, vous pouvez extraire les éléments à trier de la liste, les trier, puis les réutiliser là où ils étaient assez efficacement comme ceci:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

tri avec std :: map (croissant et décroissant)

Cet exemple trie les éléments dans l'ordre **croissant** d'une **clé à l'** aide d'une carte. Vous pouvez utiliser n'importe quel type, y compris la classe, au lieu de `std::string` , dans l'exemple ci-dessous.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1,      "Earth"));
    sorted_map.insert(std::make_pair(0.532,  "Mars"));
```

```
    sorted_map.insert(std::make_pair(10.97,  "Jupiter"));
    sorted_map.insert(std::make_pair(9.14,   "Saturn"));
    sorted_map.insert(std::make_pair(3.981,  "Uranus"));
    sorted_map.insert(std::make_pair(3.865,  "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius)" << '\n';
    }
}
```

Sortie:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

Si des entrées avec des clés égales sont possibles, utilisez `multimap` au lieu de `map` (comme dans l'exemple suivant).

Pour trier les éléments de manière **décroissante** , déclarez la carte avec un foncteur de comparaison correct (`std::greater<>`):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6,   "bug"));
    sorted_map.insert(std::make_pair(4,   "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2,   "chicken"));
    sorted_map.insert(std::make_pair(0,   "fish"));
    sorted_map.insert(std::make_pair(4,   "horse"));
    sorted_map.insert(std::make_pair(8,   "spider"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (has " << entry.first << " legs)" << '\n';
    }
}
```

Sortie

```
centipede (has 100 legs)
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
```

```
horse (has 4 legs)
chicken (has 2 legs)
fish (has 0 legs)
```

Tri des tableaux intégrés

L'algorithme de `sort` trie une séquence définie par deux itérateurs. Ceci est suffisant pour trier un tableau intégré (également appelé c-style).

C ++ 11

```
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

Avant C ++ 11, la fin du tableau devait être "calculée" en utilisant la taille du tableau:

C ++ 11

```
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Lire Tri en ligne: https://riptutorial.com/fr/cplusplus/topic/1675/tri

Chapitre 136: Type de retour

Syntaxe

- *nom_fonction* ([*noeuds_fonction*]) [*attributs_fonction*] [*qualificateurs_fonctionnels*] `->` *type-retour-fin* [*clause_exigences*]

Remarques

La syntaxe ci-dessus montre une déclaration de fonction complète en utilisant un type de fin, où les crochets indiquent une partie facultative de la déclaration de fonction (comme la liste d'arguments si une fonction non-arg).

En outre, la syntaxe du type de retour final interdit de définir une classe, une union ou un type enum dans un type de retour de fin (notez que cela n'est pas autorisé non plus dans un type de retour principal). En dehors de cela, les types peuvent être épelés de la même manière après le `->` qu'ils seraient ailleurs.

Examples

Évitez de qualifier un nom de type imbriqué

```
class ClassWithAReallyLongName {
  public:
    class Iterator { /* ... */ };
    Iterator end();
};
```

Définition de la `end` du membre avec un type de retour de fin:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Définition de la `end` du membre sans type de retour de fin:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

Le type de retour final est recherché dans la portée de la classe, tandis qu'un type de retour principal est recherché dans la portée de l'espace de nommage et peut donc nécessiter une qualification "redondante".

Expressions lambda

Un lambda ne *peut* avoir qu'un type de retour final; la syntaxe du type de retour principal n'est pas applicable à lambdas. Notez que dans de nombreux cas, il n'est pas nécessaire de spécifier un type de retour pour un lambda.

```
struct Base {};
struct Derived1 : Base {};
struct Derived2 : Base {};
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Lire Type de retour en ligne: https://riptutorial.com/fr/cplusplus/topic/4142/type-de-retour

Chapitre 137: Type de retour Covariance

Remarques

La covariance d'un paramètre ou d'une valeur de retour pour une fonction membre virtuelle `m` est celle où son type `T` devient plus spécifique dans une substitution de classe dérivée de `m` . Le type `T` varie alors (*variance*) en spécificité de la même manière (*co*) que les classes fournissant `m` . C ++ fournit un support de langage pour les *types de retour* covariants qui sont des pointeurs bruts ou des références brutes - la covariance est pour le type pointee ou référent.

La prise en charge de C ++ est limitée aux types de retour car les valeurs de retour de fonction sont les seuls **arguments de sortie** purs en C ++, et la covariance est uniquement de type sûr pour un argument de sortie pur. Sinon, le code appelant pourrait fournir un objet de type moins spécifique que celui attendu par le code de réception. Barbara Liskov, professeur au MIT, a étudié cette question et les problèmes de sécurité liés à la variance, et elle est maintenant connue sous le nom de principe de substitution Liskov ou **LSP** .

La prise en charge de la covariance permet essentiellement d’éviter le downcasting et la vérification de type dynamique.

Étant donné que les pointeurs intelligents sont de type classe on ne peut pas utiliser directement le support intégré pour covariance intelligents résultats de pointeur, mais on peut définir *apparemment covariants* non `virtual` fonctions wrapper intelligentes de résultat de pointeur pour une covariante `virtual` fonction qui produit des pointeurs premières.

Examples

1. Exemple de base sans retours covariants, montre pourquoi ils sont souhaitables

```
// 1. Base example not using language support for covariance, dynamic type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{

public:
    Top* clone() const override
    { return new D( *this ); }
};

class DD : public D
{
```

```
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}

    Top* clone() const override
    { return new DD( *this ); }
};

#include <assert.h>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
    cout << boolalpha;

    DD* p1 = new DD();
    Top* p2 = p1->clone();
    bool const  correct_dynamic_type = (typeid( *p2 ) == typeid( DD ));
    cout << correct_dynamic_type << endl;               // "true"

    assert( correct_dynamic_type ); // This is essentially dynamic type checking. :(
    auto p2_most_derived = static_cast<DD*>( p2 );
    cout << p2_most_derived->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

2. Version du résultat de covariant de l'exemple de base, vérification de type statique.

```
// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}
```

```
    DD* /* ← Covariant return */ clone() const override
    { return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

3. Résultat du pointeur intelligent covariant (nettoyage automatique).

```
// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
    { return up( virtual_clone() ); }

    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
    D* /* ← Covariant return */ virtual_clone() const override
    { return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

class DD : public D
{
private:
    int answer_ = 42;

    DD* /* ← Covariant return */ virtual_clone() const override
    { return new DD( *this ); }
```

```
public:
    int answer() const
    { return answer_;}

    unique_ptr<DD> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

#include <iostream>
using namespace std;

int main()
{
    auto  p1 = unique_ptr<DD>(new DD());
    auto  p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    // Cleanup is automated via unique_ptr.
 }
```

Lire Type de retour Covariance en ligne: https://riptutorial.com/fr/cplusplus/topic/5411/type-de-retour-covariance

Chapitre 138: Type effacement

Introduction

L'effacement de type est un ensemble de techniques permettant de créer un type pouvant fournir une interface uniforme à différents types sous-jacents, tout en masquant les informations de type sous-jacentes au client. `std::function<R(A...)>` , qui a la capacité de contenir des objets appelables de différents types, est peut-être l'exemple le plus connu d'effacement de type en C ++.

Examples

Mécanisme de base

L'effacement de type permet de masquer le type d'un objet à l'aide du code, même s'il n'est pas dérivé d'une classe de base commune. Ce faisant, il fournit un pont entre les mondes du polymorphisme statique (modèles; au lieu d'utilisation, le type exact doit être connu au moment de la compilation, mais il n'a pas besoin d'être déclaré conforme à une interface à la définition) et le polymorphisme dynamique (héritage et fonctions virtuelles; sur le lieu d'utilisation, le type exact n'a pas besoin d'être connu au moment de la compilation, mais doit être déclaré conforme à une interface à la définition).

Le code suivant montre le mécanisme de base de type effacement.

```
#include <ostream>

class Printable
{
public:
  template <typename T>
  Printable(T value) : pValue(new Value<T>(value)) {}
  ~Printable() { delete pValue; }
  void print(std::ostream &os) const { pValue->print(os); }

private:
  Printable(Printable const &)        /* in C++1x: =delete */; // not implemented
  void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
  struct ValueBase
  {
      virtual ~ValueBase() = default;
      virtual void print(std::ostream &) const = 0;
  };
  template <typename T>
  struct Value : ValueBase
  {
      Value(T const &t) : v(t) {}
      virtual void print(std::ostream &os) const { os << v; }
      T v;
  };
  ValueBase *pValue;
};
```

Sur le site d'utilisation, seule la définition ci-dessus doit être visible, comme pour les classes de base avec des fonctions virtuelles. Par exemple:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Notez qu'il *ne* s'agit *pas d'* un modèle, mais d'une fonction normale qui doit uniquement être déclarée dans un fichier d'en-tête et qui peut être définie dans un fichier d'implémentation (contrairement aux modèles dont la définition doit être visible sur le lieu d'utilisation).

A la définition du type concret, il n'ya rien à savoir sur `Printable` , il suffit de se conformer à une interface, comme avec les templates:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
  return os << "MyType {" << mc.i << "}";
}
```

On peut maintenant passer un objet de cette classe à la fonction définie ci-dessus:

```
MyType foo = { 42 };
print_value(foo);
```

Effacement à un type régulier avec une vtable manuelle

C ++ prospère sur ce que l'on appelle un type régulier (ou du moins pseudo-régulier).

Un type Regular est un type qui peut être construit, assigné et attribué par copie ou par déplacement, peut être détruit et peut être comparé à une valeur égale. Il peut également être construit sans arguments. Enfin, il prend également en charge quelques autres opérations très utiles dans divers algorithmes et conteneurs `std` .

Ceci est le document racine , mais en C ++ 11 voudrait ajouter le support `std::hash` .

Je vais utiliser l'approche vtable manuelle pour taper l'effacement ici.

```
using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
  return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self);
}};
}
struct regular_vtable {
  void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
  void(*move_assign)(void* dest, void* src); // T&=(T&&)
  bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
  bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
```

```
  std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
  std::type_info const&(*type)(); // typeid(T)
  dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
  return {
    [](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
    [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
    [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T
const*>(rhs); },
    [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T
const*>(lhs),*static_cast<T const*>(rhs)); },
    [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
    []()->decltype(auto){ return typeid(T); },
    [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
  };
}
template<class T>
regular_vtable const* get_regular_vtable() noexcept {
  static const regular_vtable vtable=make_regular_vtable<T>();
  return &vtable;
}

struct regular_type {
  using self=regular_type;
  regular_vtable const* vtable = 0;
  dtor_unique_ptr ptr{nullptr, [](void*){}};

  bool empty() const { return !vtable; }

  template<class T, class...Args>
  void emplace( Args&&... args ) {
    ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
    if (ptr)
      vtable = get_regular_vtable<T>();
    else
      vtable = nullptr;
  }
  friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
    if (lhs.vtable != rhs.vtable) return false;
    return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
  }
  bool before(regular_type const& rhs) const {
    auto const& lhs = *this;
    if (!lhs.vtable || !rhs.vtable)
      return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
    if (lhs.vtable != rhs.vtable)
      return lhs.vtable->type().before(rhs.vtable->type());
    return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
  }
  // technically friend bool operator< that calls before is also required

  std::type_info const* type() const {
    if (!vtable) return nullptr;
    return &vtable->type();
  }
  regular_type(regular_type&& o):
    vtable(o.vtable),
    ptr(std::move(o.ptr))
```

```
  {
    o.vtable = nullptr;
  }
  friend void swap(regular_type& lhs, regular_type& rhs){
    std::swap(lhs.ptr, rhs.ptr);
    std::swap(lhs.vtable, rhs.vtable);
  }
  regular_type& operator=(regular_type&& o) {
    if (o.vtable == vtable) {
      vtable->move_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = std::move(o);
    swap(*this, tmp);
    return *this;
  }
  regular_type(regular_type const& o):
    vtable(o.vtable),
    ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr{nullptr, [](void*){}})
  {
    if (!ptr && vtable) vtable = nullptr;
  }
  regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
      vtable->copy_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
  }
  std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
  }
  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
  >
  regular_type(T&& t) {
    emplace<std::decay_t<T>>(std::forward<T>(t));
  }
};
namespace std {
  template<>
  struct hash<regular_type> {
    std::size_t operator()( regular_type const& r )const {
      return r.hash();
    }
  };
  template<>
  struct less<regular_type> {
    bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
      return lhs.before(rhs);
    }
  };
}
```

exemple vivant .

Un tel type régulier peut être utilisé comme clé pour un `std::map` ou un `std::unordered_map` qui

accepte *tout ce qui est normal* pour une clé, comme:

```
std::map<regular_type, std::any>
```

serait fondamentalement une carte de rien régulier, à tout ce qui peut être copié.

Contrairement à `any` , my `regular_type` ne fait pas d'optimisation de petits objets et ne permet pas de récupérer les données d'origine. Obtenir le type original n'est pas difficile.

L'optimisation des petits objets nécessite que nous `regular_type` un tampon de stockage aligné dans le type `regular_type` , et `regular_type` soigneusement le paramètre de suppression du `ptr` pour ne détruire que l'objet et non le supprimer.

Je commencerais par `make_dtor_unique_ptr` et je lui apprendrais à stocker parfois les données dans un tampon, puis dans le tas s'il n'y a pas de place dans le tampon. Cela peut être suffisant.

Un simple `std :: function`

`std::function` efface quelques opérations. Une des choses qu'il faut, c'est que la valeur stockée soit copiable.

Cela pose des problèmes dans quelques contextes, tels que les lambda stockant des ptrs uniques. Si vous utilisez la `std::function` dans un contexte où la copie importe peu, comme un pool de threads où vous envoyez des tâches à des threads, cette exigence peut entraîner une surcharge.

En particulier, `std::packaged_task<Sig>` est un objet appelable qui est en déplacement uniquement. Vous pouvez stocker un `std::packaged_task<R(Args...)>` dans un `std::packaged_task<void(Args...)>` , mais c'est un moyen assez lourd et obscur de créer un mouvement uniquement classe de type effaçable.

Ainsi la `task` . Cela montre comment vous pouvez écrire un simple type `std::function` . J'ai omis le constructeur de copie (ce qui impliquerait également d'ajouter une méthode de `clone` à `details::task_pimpl<...>`).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
namespace details {
  template<class R, class...Args>
  struct task_pimpl {
    virtual R invoke(Args&&...args) const = 0;
    virtual ~task_pimpl() {};
    virtual const std::type_info& target_type() const = 0;
  };

  // store an F.  invoke(Args&&...) calls the f
  template<class F, class R, class...Args>
  struct task_pimpl_impl:task_pimpl<R,Args...> {
    F f;
    template<class Fin>
```

```
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual R invoke(Args&&...args) const final override {
      return f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };

  // the void version discards the return value of f:
  template<class F, class...Args>
  struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual void invoke(Args&&...args) const final override {
      f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };
};

template<class R, class...Args>
struct task<R(Args...)> {
  // semi-regular:
  task()=default;
  task(task&&)=default;
  // no copy

private:
  // aliases to make some SFINAE code below less ugly:
  template<class F>
  using call_r = std::result_of_t<F const&(Args...)>;
  template<class F>
  using is_task = std::is_same<std::decay_t<F>, task>;
public:
  // can be constructed from a callable F
  template<class F,
    // that can be invoked with Args... and converted-to-R:
    class= decltype( (R)(std::declval<call_r<F>>()) ),
    // and is not this same type:
    std::enable_if_t<!is_task<F>{}, int>* = nullptr
  >
  task(F&& f):
    m_pImpl( make_pimpl(std::forward<F>(f)) )
  {}

  // the meat: the call operator
  R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
  }
  explicit operator bool() const {
    return (bool)m_pImpl;
  }
  void swap( task& o ) {
    std::swap( m_pImpl, o.m_pImpl );
  }
  template<class F>
  void assign( F&& f ) {
```

```
    m_pImpl = make_pimpl(std::forward<F>(f));
  }
  // Part of the std::function interface:
  const std::type_info& target_type() const {
    if (!*this) return typeid(void);
    return m_pImpl->target_type();
  }
  template< class T >
  T* target() {
    return target_impl<T>();
  }
  template< class T >
  const T* target() const {
    return target_impl<T>();
  }
  // compare with nullptr    :
  friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
  friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
  friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }
  friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
  template<class T>
  using pimpl_t = details::task_pimpl_impl<T, R, Args...>;

  template<class F>
  static auto make_pimpl( F&& f ) {
    using dF=std::decay_t<F>;
    using pImpl_t = pimpl_t<dF>;
    return std::make_unique<pImpl_t>(std::forward<F>(f));
  }
  std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;

  template< class T >
  T* target_impl() const {
    return dynamic_cast<pimpl_t<T>*>(m_pImpl.get());
  }
};
```

Pour rendre cette bibliothèque digne de ce nom, vous voudrez ajouter une petite optimisation de mémoire tampon, afin de ne pas stocker tous les appels sur le tas.

L'ajout de SBO nécessiterait une `task(task&&)` par défaut `task(task&&)` , certains `std::aligned_storage_t` dans la classe, un `m_pImpl` `unique_ptr` avec un paramètre qui peut être défini pour détruire uniquement (et ne pas renvoyer la mémoire au tas), et un `emplace_move_to( void* ) = 0` dans la `task_pimpl` .

exemple en direct du code ci-dessus (sans SBO).

Effacement à un tampon contigu de T

Tout effacement de type n'implique pas d'héritage virtuel, d'allocations, de nouveaux emplacements ou même de pointeurs de fonctions.

Ce qui rend l'effacement de type effacement de type, c'est qu'il décrit un (ensemble de) comportement (s), et prend tout type qui prend en charge ce comportement et l'enveloppe. Toutes les informations qui ne figurent pas dans cet ensemble de comportements sont "oubliées" ou

"effacées".

Un `array_view` prend sa plage entrante ou son type de conteneur et efface tout sauf le fait qu'il s'agit d'un tampon contigu de `T`

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} ||
std::is_same< data_t<Src>, std::remove_const_t<T>* >{}>;

template<class T>
struct array_view {
  // the core of the class:
  T* b=nullptr;
  T* e=nullptr;
  T* begin() const { return b; }
  T* end() const { return e; }

  // provide the expected methods of a good contiguous range:
  T* data() const { return begin(); }
  bool empty() const { return begin()==end(); }
  std::size_t size() const { return end()-begin(); }

  T& operator[](std::size_t i)const{ return begin()[i]; }
  T& front()const{ return *begin(); }
  T& back()const{ return *(end()-1); }

  // useful helpers that let you generate other ranges from this one
  // quickly and safely:
  array_view without_front( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin()+i, end()};
  }
  array_view without_back( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin(), end()-i};
  }

  // array_view is plain old data, so default copy:
  array_view(array_view const&)=default;
  // generates a null, empty range:
  array_view()=default;

  // final constructor:
  array_view(T* s, T* f):b(s),e(f) {}
  // start and length is useful in my experience:
  array_view(T* s, std::size_t length):array_view(s, s+length) {}

  // SFINAE constructor that takes any .data() supporting container
  // or other range in one fell swoop:
  template<class Src,
    std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
    std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
  >
  array_view( Src&& src ):
    array_view( src.data(), src.size() )
  {}
```

```
  // array constructor:
  template<std::size_t N>
  array_view( T(&arr)[N] ):array_view(arr, N) {}

  // initializer list, allowing {} based:
  template<class U,
    std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
  >
  array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}
};
```

un `array_view` prend tout conteneur qui prend en charge `.data()` renvoyant un pointeur sur `T` et une méthode `.size()` , ou un tableau, et l'efface pour devenir une plage d'accès aléatoire sur les `T` contigus.

Il peut prendre un `std::vector<T>` , un `std::string<T>` un `std::array<T, N>` un `T[37]` , une liste d'initialiseurs (y compris ceux basés sur `{}`), ou quelque chose d'autre vous `T* x.data()` qui le supporte (via `T* x.data()` et `size_t x.size()`).

Dans ce cas, les données que nous pouvons extraire de la chose que nous effaçons, ainsi que notre état non propriétaire "view", signifient que nous n'avons pas besoin d'allouer de la mémoire ou d'écrire des fonctions personnalisées dépendant du type.

Exemple en direct .

Une amélioration consisterait à utiliser des `data` non membres et une `size` non membre dans un contexte compatible ADL.

Type effacement type effacement avec std :: any

Cet exemple utilise C ++ 14 et `boost::any` . Dans C ++ 17, vous pouvez échanger `std::any` place.

La syntaxe avec laquelle nous nous retrouvons est la suivante:

```
const auto print =
  make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });

super_any<decltype(print)> a = 7;

(a->*print)(std::cout);
```

ce qui est presque optimal.

Cet exemple est basé sur le travail de @dyp et @cpplearner ainsi que sur le mien.

Nous utilisons d'abord un tag pour faire circuler les types:

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

Cette classe de trait récupère la signature stockée avec `any_method` :

Cela crée un type de pointeur de fonction et une fabrique pour les pointeurs de ces fonctions, avec une `any_method` :

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
  template<class T>
  using decorate = std::conditional_t< any_method::is_const, T const, T >;

  using any = decorate<boost::any>;

  using type = R(*)(any&, any_method const*, Args&&...);
  template<class T>
  type operator()( tag_t<T> )const{
    return +[](any& self, any_method const* method, Args&&...args) {
      return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
    };
  }
};
```

`any_method_function::type` est le type d'un pointeur de fonction que nous allons stocker avec l'instance. `any_method_function::operator()` prend un `tag_t<T>` et écrit une instance personnalisée du `any_method_function::type` qui suppose que le `any&` va être un `T`

Nous voulons pouvoir effacer plus d'une méthode à la fois. Nous les regroupons donc dans un tuple et écrivons un wrapper helper pour coller le tuple au stockage statique par type et y placer un pointeur.

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
  return std::make_tuple(
    any_method_function<any_methods>{}(tag<T>)...
  );
}

template<class...methods>
struct any_methods {
private:
  any_method_tuple<methods...> const* vtable = 0;
  template<class T>
  static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
    static const auto table = make_vtable<methods...>(tag<T>);
    return &table;
  }
public:
  any_methods() = default;
```

```
  template<class T>
  any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
  any_methods& operator=(any_methods const&)=default;
  template<class T>
  void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

  template<class any_method>
  auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
  }
};
```

Nous pourrions nous spécialiser dans un cas où la vtable est petite (par exemple, 1 élément) et utiliser des pointeurs directs stockés en classe dans ces cas pour des raisons d’efficacité.

Maintenant, nous commençons le `super_any` . J'utilise `super_any_t` pour rendre la déclaration de `super_any` un peu plus facile.

```
template<class...methods>
struct super_any_t;
```

Ceci recherche les méthodes que le super supporte pour SFINAE et les meilleurs messages d'erreur:

```
template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
    std::integral_constant<bool, std::is_same<M0, method>{}  ||
super_method_applies_helper<super_any_t<Methods...>, method>{}>
{};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} >{};
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};
```

Ensuite, nous créons le type `any_method` . Une `any_method` est un pseudo-méthode-pointeur. Nous créons globalement et `const` ment en utilisant une syntaxe:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

ou en C ++ 17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Notez que l'utilisation d'un non-lambda peut rendre les choses plus poilues, car nous utilisons le type pour une étape de recherche. Cela peut être corrigé, mais rendrait cet exemple plus long qu’il ne l’est déjà. Donc, toujours initialiser une méthode à partir d'un lambda, ou d'un type paramétré sur un lambda.

```
template<class Sig, bool const_method, class F>
struct any_method {
  using signature=Sig;
  enum{is_const=const_method};
private:
  F f;
public:

  template<class Any,
    // SFINAE testing that one of the Anys's matches this type:
    std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
  >
  friend auto operator->*( Any&& self, any_method const& m ) {
    // we don't use the value of the any_method, because each any_method has
    // a unique type (!) and we check that one of the auto*'s in the super_any
    // already has a pointer to us.  We then dispatch to the corresponding
    // any_method_data...

    return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)-
>decltype(auto)
    {
      return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
  }
  any_method( F fin ):f(std::move(fin)) {}

  template<class...Args>
  decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
  }
};
```

Une méthode d'usine, pas nécessaire en C ++ 17 Je crois:

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
  return {std::forward<F>(f)};
}
```

C'est la `any` augmentée. Il est à la fois un `any` , et il porte autour d' un faisceau de pointeurs de fonctions de type effacement qui change chaque fois que le contenu `any` fait:

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
  using vtable=any_methods<methods...>;
```

```
public:
  template<class T,
    std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
  >
  super_any_t( T&& t ):
    boost::any( std::forward<T>(t) )
  {
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
  }

  boost::any& as_any()&{return *this;}
  boost::any&& as_any()&&{return std::move(*this);}
  boost::any const& as_any()const&{return *this;}
  super_any_t()=default;
  super_any_t(super_any_t&& o):
    boost::any( std::move( o.as_any() ) ),
    vtable(o)
  {}
  super_any_t(super_any_t const& o):
    boost::any( o.as_any() ),
    vtable(o)
  {}
  template<class S,
    std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
  >
  super_any_t( S&& o ):
    boost::any( std::forward<S>(o).as_any() ),
    vtable(o)
  {}
  super_any_t& operator=(super_any_t&&)=default;
  super_any_t& operator=(super_any_t const&)=default;

  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>* =nullptr
  >
  super_any_t& operator=( T&& t ) {
    ((boost::any&)*this) = std::forward<T>(t);
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
    return *this;
  }
};
```

Comme nous stockons `any_method` s en tant qu'objets `const` , cela facilite la création d'un `super_any` :

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Code de test:

```
const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p
<< "\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os
<< p << L"\n"; });

int main()
{
  super_any<decltype(print), decltype(wprint)> a = 7;
```

```
    super_any<decltype(print), decltype(wprint)> a2 = 7;

    (a->*print)(std::cout);
    (a->*wprint)(std::wcout);
}
```

exemple vivant .

Initialement posté ici dans une question et une réponse SO (et les personnes mentionnées ci-dessus ont aidé à la mise en œuvre).

Lire Type effacement en ligne: https://riptutorial.com/fr/cplusplus/topic/2872/type-effacement

Chapitre 139: Type Inférence

Introduction

Cette rubrique traite de l'inférence de type impliquant le type `auto` mot-clé disponible à partir de C ++ 11.

Remarques

Il est généralement préférable de déclarer `const` , `&` et `constexpr` chaque fois que vous utilisez `auto` si cela est nécessaire pour éviter les comportements indésirables tels que la copie ou les mutations. Ces indications supplémentaires garantissent que le compilateur ne génère aucune autre forme d'inférence. Il est également déconseillé de trop utiliser `auto` et devrait être utilisé uniquement lorsque la déclaration réelle est très longue, en particulier avec les modèles STL.

Examples

Type de données: Auto

Cet exemple montre les inférences de type de base que le compilateur peut effectuer.

```
auto a = 1;        //     a = int
auto b = 2u;       //     b = unsigned int
auto c = &a;       //     c = int*
const auto  d = c; //     d = const int*
const auto& e = b; //     e = const unsigned int&

auto x = a + b     //     x = int, #compiler warning unsigned and signed

auto v = std::vector<int>;    //     v = std::vector<int>
```

Toutefois, le mot-clé auto n'effectue pas toujours l'inférence de type attendue sans indications supplémentaires pour `&` ou `const` ou `constexpr`

```
//    y = unsigned int,
//    note that y does not infer as const unsigned int&
//    The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Auto lambda

Le mot-clé auto type de données est un moyen pratique pour les programmeurs de déclarer les fonctions lambda. Il est utile de raccourcir la quantité de texte que les programmeurs doivent saisir pour déclarer un pointeur de fonction.

```
auto DoThis = [](int a, int b) { return a + b; };
```

```
//    Do this is of type (int)(*DoThis)(int, int)
//    else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2);    //    c = int
auto d = pDothis(1, 2);   //    d = int

//    using 'auto' shortens the definition for lambda functions
```

Par défaut, si le type de retour des fonctions lambda n'est pas défini, il sera automatiquement déduit des types d'expression de retour.

Ces 3 sont fondamentalement la même chose

```
[](int a, int b) -> int  { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

Boucles et auto

Cet exemple montre comment auto peut être utilisé pour raccourcir la déclaration de type pour les boucles for

```
std::map<int, std::string> Map;
for (auto pair : Map)            //    pair = std::pair<int, std::string>
for (const auto pair : Map)      //    pair = const std::pair<int, std::string>
for (const auto& pair : Map)     //    pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i)        //    i = int
for (auto i = 0; i < Map.size(); ++i)  //    Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i)  //    i = size_t
```

Lire Type Inférence en ligne: https://riptutorial.com/fr/cplusplus/topic/8233/type-inference

Chapitre 140: Type Traits

Remarques

Les traits de type sont des constructions basées sur des modèles utilisées pour comparer et tester les propriétés de différents types au moment de la compilation. Ils peuvent être utilisés pour fournir une logique conditionnelle au moment de la compilation qui peut limiter ou étendre les fonctionnalités de votre code de manière spécifique. La bibliothèque de caractères de type a été introduite avec le standard `c++11` qui fournit un certain nombre de fonctionnalités différentes. Il est également possible de créer vos propres modèles de comparaison de caractères.

Examples

Traits de type standard

C ++ 11

L'en `type_traits` tête `type_traits` contient un ensemble de classes de modèles et de helpers pour transformer et vérifier les propriétés des types au moment de la compilation.

Ces traits sont généralement utilisés dans les modèles pour vérifier les erreurs des utilisateurs, prendre en charge la programmation générique et permettre des optimisations.

La plupart des caractères sont utilisés pour vérifier si un type remplit certains critères. Ceux-ci ont la forme suivante:

```
template <class T> struct is_foo;
```

Si la classe de modèle est instancié avec un type qui répond à certains critères `foo` , puis `is_foo<T>` hérite de `std::integral_constant<bool,true>` (alias `std::true_type`), sinon il hérite de `std::integral_constant<bool,false>` (aka `std::false_type`). Cela donne le trait aux membres suivants:

Les constantes

`static constexpr bool value`

`true` si `T` remplit les critères `foo` , `false` sinon

Les fonctions

`operator bool`

Renvoie la `value`

C ++ 14

`bool operator()`

Renvoie la `value`

Les types

prénom	Définition
`value_type`	`bool`
`type`	`std::integral_constant<bool,value>`

Le trait peut ensuite être utilisé dans des constructions telles que `static_assert` ou `std::enable_if` . Un exemple avec `std::is_pointer` :

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}
```

Il existe également divers traits qui transforment les types, tels que `std::add_pointer` et `std::underlying_type` `std::add_pointer` `std::underlying_type` . Ces traits exposent généralement un `type` membre de type unique qui contient le type transformé. Par exemple, `std::add_pointer<int>::type` est `int*` .

Tapez les relations avec std :: is_same

C ++ 11

La relation de type `std::is_same<T, T>` est utilisée pour comparer deux types. Il évaluera comme booléen, true si les types sont les mêmes et faux si autrement.

par exemple

```
// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout  << std::is_same<unsigned int, int>::value << "\n";
```

La relation de type `std::is_same` fonctionnera également indépendamment des typedefs. Ceci est en fait démontré dans le premier exemple en comparant `int == int32_t` mais ce n'est pas tout à fait clair.

par exemple

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value <<  "\n";
```

Utiliser `std::is_same` pour avertir lorsque vous utilisez incorrectement une classe ou une fonction `std::is_same` sur un modèle.

Lorsqu'il est associé à une affirmation statique, le modèle `std::is_same` peut être un outil précieux pour `std::is_same` utilisation correcte des classes et des fonctions `std::is_same` modèles.

Par exemple, une fonction qui n'autorise que les entrées d'un `int` et un choix de deux structures.

```
#include <type_traits>
struct foo {
  int member;
  // Other variables
};

struct bar {
  char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
  // If type T != foo || T != bar then show error message.
  static_assert(std::is_same<T, foo>::value ||
    std::is_same<T, bar>::value,
    "This function does not support the specified type.");
  return var1.member + var2;
}
```

Traits de type fondamentaux

C ++ 11

Il existe un certain nombre de caractères différents qui comparent des types plus généraux.

Est intégral:

Evalue comme vrai pour tous les types entiers `int` , `char` , `long` , `unsigned int` etc.

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

Est Flottant Point:

Évalue comme vrai pour tous les types à virgule flottante. `float` , `double` , `long double` etc.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

Est Enum:

Évalue comme vrai pour tous les types énumérés, y compris la `enum class` .

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

Est le pointeur:

Évalue comme vrai pour tous les pointeurs.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

Est classe:

Evalue comme vrai pour toutes les classes et struct, à l'exception de la `enum class` .

```
struct FOO {int x, y;};
class BAR {
 public:
  int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

Propriétés du type

C ++ 11

Les propriétés de type comparent les modificateurs pouvant être placés sur différentes variables. L'utilité de ces caractères de type n'est pas toujours évidente.

Remarque: L'exemple ci-dessous n'offre qu'une amélioration sur un compilateur non optimisé. C'est un simple exemple de preuve de concept, plutôt qu'un exemple complexe.

Par exemple, diviser rapidement par quatre.

```
template<typename T>
inline T FastDivideByFour(cont T &var) {
  // Will give an error if the inputted type is not an unsigned integral type.
  static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
    "This function is only designed for unsigned integral types.");
  return (var >> 2);
}
```

Est constante:

Cela sera considéré comme vrai lorsque le type est constant.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

Est volatile:

Cela sera évalué comme vrai lorsque le type est volatile.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

Est signé:

Cela sera considéré comme vrai pour tous les types signés.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Est non signé:

Sera évalué comme vrai pour tous les types non signés.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

Lire Type Traits en ligne: https://riptutorial.com/fr/cplusplus/topic/4750/type-traits

Chapitre 141: Typedef et alias de type

Introduction

Le `typedef` et (depuis C ++ 11) `using` mots - clés peuvent être utilisés pour donner un nouveau nom à un type existant.

Syntaxe

- typedef *type-specifier-seq liste init-déclarator* ;
- *attribut-spécificateur-seq* typedef *décl-spécificateur-seq liste init-déclarateur* ; // depuis C ++ 11
- en utilisant l' *identificateur attribut-spécificateur-seq* (*opt*) = *id-type* ; // depuis C ++ 11

Examples

Syntaxe de base de typedef

Une déclaration `typedef` a la même syntaxe qu'une déclaration de variable ou de fonction, mais elle contient le mot `typedef` . La présence de `typedef` fait que la déclaration déclare un type au lieu d'une variable ou d'une fonction.

```
int T;          // T has type int
typedef int T; // T is an alias for int

int A[100];         // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Une fois qu'un alias de type a été défini, il peut être utilisé indifféremment avec le nom d'origine du type.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

`typedef` ne crée jamais un type distinct. Cela ne donne qu'une autre façon de faire référence à un type existant.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Utilisations plus complexes du typedef

La règle selon laquelle les déclarations `typedef` ont la même syntaxe que les déclarations de variables et de fonctions ordinaires peut être utilisée pour lire et écrire des déclarations plus complexes.

```
void (*f)(int);         // f has type "pointer to function of int returning void"
typedef void (*f)(int); // f is an alias for "pointer to function of int returning void"
```

Ceci est particulièrement utile pour les constructions avec une syntaxe confuse, telles que les pointeurs vers les membres non statiques.

```
void (Foo::*pmf)(int);         // pmf has type "pointer to member function of Foo taking int
                               // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                               // taking int and returning void"
```

Il est difficile de se souvenir de la syntaxe des déclarations de fonctions suivantes, même pour les programmeurs expérimentés:

```
void (Foo::*Foo::f(const char*))(int);
int (&g())[100];
```

`typedef` peut être utilisé pour faciliter leur lecture et leur écriture:

```
typedef void (Foo::pmf)(int);  // pmf is a pointer to member function type
pmf Foo::f(const char*);       // f is a member function of Foo

typedef int (&ra)[100];        // ra means "reference to array of 100 ints"
ra g();                        // g returns reference to array of 100 ints
```

Déclarer plusieurs types avec typedef

Le mot-clé `typedef` est un spécificateur, il s'applique donc séparément à chaque déclarant. Par conséquent, chaque nom déclaré fait référence au type que ce nom aurait en l'absence de `typedef` .

```
int *x, (*p)();         // x has type int*, and p has type int(*)()
typedef int *x, (*p)(); // x is an alias for int*, while p is an alias for int(*)()
```

Déclaration d'alias avec "using"

C ++ 11

La syntaxe d' `using` est très simple: le nom à définir va du côté gauche et la définition du côté droit. Pas besoin de scanner pour voir où est le nom.

```
using I = int;
using A = int[100];             // array of 100 ints
```

```
using FP = void(*)(int);          // pointer to function of int returning void
using MP = void (Foo::*)(int);   // pointer to member function of Foo of int returning void
```

La création d'un alias de type avec `using` a exactement le même effet que la création d'un alias de type avec `typedef` . C'est simplement une syntaxe alternative pour accomplir la même chose.

Contrairement au `typedef` , l' `using` peut être basée sur des modèles. Un "template typedef" créé avec `using` est appelé un modèle d'alias .

Lire Typedef et alias de type en ligne: https://riptutorial.com/fr/cplusplus/topic/9328/typedef-et-alias-de-type

Chapitre 142: Types atomiques

Syntaxe

- std :: atomique <T>
- std :: atomic_flag

Remarques

`std::atomic` permet l'accès atomique à un type TriviallyCopyable, il dépend de l'implémentation si cela se fait via des opérations atomiques ou en utilisant des verrous. Le seul type atomique garanti sans verrou est `std::atomic_flag` .

Examples

Accès multithread

Un type atomique peut être utilisé pour lire et écrire en toute sécurité dans un emplacement de mémoire partagé entre deux threads.

Un mauvais exemple susceptible de provoquer une course de données:

```
#include <thread>
#include <iostream>


//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
        std::cout << shared << std::endl;
    }
```

```
    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

L'exemple ci-dessus peut provoquer une lecture corrompue et entraîner un comportement indéfini.

Un exemple avec la sécurité des threads:

```
#include <atomic>
#include <thread>
#include <iostream>


    //function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }


    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

L'exemple ci - dessus est sûr parce que tout `store()` et `load()` opérations du `atomic` type de données protègent le encapsulé `int` d' un accès simultané.

Lire Types atomiques en ligne: https://riptutorial.com/fr/cplusplus/topic/3804/types-atomiques

Chapitre 143: Types sans nom

Examples

Classes sans nom

Contrairement à une classe ou à une structure nommée, les classes et les structures non nommées doivent être instanciées là où elles sont définies et ne peuvent pas avoir de constructeur ou de destructeur.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Membres anonymes

En tant qu'extension non standard de C ++, les compilateurs communs permettent l'utilisation de classes en tant que membres anonymes.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;

    //The anonymous struct's members are accessed as if members of the parent struct
    Example() : inner_b(2), outer_b(4) {
        inner_b = outer_b + 2;
    }
};

Example ex;

//The same holds true for external code referencing the struct
ex.inner_b -= ex.outer_b;
```

Comme alias de type

Les types de classe sans nom peuvent également être utilisés lors de la création d'alias de type, c'est-à-dire via `typedef` et en `using` :

C ++ 11

```
using vec2d = struct {
    float x;
    float y;
};
```

```
typedef struct {
    float x;
    float y;
} vec2d;
```

```
vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

Union anonyme

Les noms de membres d'une union anonyme appartiennent à la portée de la déclaration d'union et doivent être distincts de tous les autres noms de cette portée. L'exemple ici a la même construction que l'exemple de membres anonymes utilisant "struct" mais est conforme aux normes.

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
  Sample sa;
  sa.a =3;
  sa.b =4;
  sa.c =5;
}
```

Lire Types sans nom en ligne: https://riptutorial.com/fr/cplusplus/topic/2704/types-sans-nom

Chapitre 144: Une règle de définition (ODR)

Examples

Multiplier la fonction définie

La conséquence la plus importante de la règle de définition unique est que les fonctions non intégrées avec un lien externe ne doivent être définies qu'une seule fois dans un programme, bien qu'elles puissent être déclarées plusieurs fois. Par conséquent, ces fonctions ne doivent pas être définies dans les en-têtes, car un en-tête peut être inclus plusieurs fois à partir de différentes unités de traduction.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() { std:: cout << "bar"; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

Dans ce programme, la fonction `foo` est définie dans l'en-tête `foo.h` , qui est incluse deux fois: une fois depuis `foo.cpp` et une fois depuis `main.cpp` . Chaque unité de traduction contient donc sa propre définition de `foo` . Notez que les gardes d' `foo.h` dans `foo.h` n'empêchent pas que cela se produise, puisque `foo.cpp` et `main.cpp` chacun *séparément* `foo.h` Le résultat le plus probable de la tentative de création de ce programme est une erreur de lien-time identifiant `foo` comme ayant été définie par une multiplication.

Pour éviter de telles erreurs, il convient de *déclarer les* fonctions dans les en-têtes et de les *définir* dans les fichiers `.cpp` correspondants, à quelques exceptions près (voir d'autres exemples).

Fonctions en ligne

Une fonction déclarée en `inline` peut être définie dans plusieurs unités de traduction, à condition

que toutes les définitions soient identiques. Il doit également être défini dans chaque unité de traduction dans laquelle il est utilisé. Par conséquent, les fonctions en ligne *doivent* être définies dans les en-têtes et il n'est pas nécessaire de les mentionner dans le fichier d'implémentation.

Le programme se comportera comme s'il y avait une seule définition de la fonction.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

Dans cet exemple, la fonction plus simple `foo` est définie en ligne dans le fichier d'en-tête alors que la `bar` fonctions plus compliquée n'est pas intégrée et est définie dans le fichier d'implémentation. Les deux unités de traduction `foo.cpp` et `main.cpp` contiennent des définitions de `foo` , mais ce programme est bien formé puisque `foo` est en ligne.

Une fonction définie dans une définition de classe (qui peut être une fonction membre ou une fonction amie) est *implicitement* intégrée. Par conséquent, si une classe est définie dans un en-tête, les fonctions membres de la classe peuvent être définies dans la définition de classe, même si les définitions peuvent être incluses dans plusieurs unités de traduction:

```
// in foo.h
class Foo {
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
void Foo::baz() {
   // definition
}
```

La fonction `Foo::baz` est définie hors ligne, ce n'est donc *pas* une fonction inline et ne doit pas être

définie dans l'en-tête.

Violation d'ODR via une résolution de surcharge

Même avec des jetons identiques pour les fonctions en ligne, les ODR peuvent être violés si la recherche de noms ne fait pas référence à la même entité. considérons `func` en suivant:

- header.h

```
void overloaded(int);
inline void func() { overloaded('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    func(); // `overloaded` refers to `void overloaded(int)`
}
```

- bar.cpp

```
void overloaded(char); // can come from other include
#include "header.h"

void bar()
{
    func(); // `overloaded` refers to `void overloaded(char)`
}
```

Nous avons une violation de l'ODR comme `overloaded` fait référence à différentes entités en fonction de l'unité de traduction.

Lire Une règle de définition (ODR) en ligne: https://riptutorial.com/fr/cplusplus/topic/4907/une-regle-de-definition--odr-

Chapitre 145: Utiliser la déclaration

Introduction

Une déclaration `using` introduit un nom unique dans la portée actuelle précédemment déclarée ailleurs.

Syntaxe

- en utilisant typename (*opt*) *nested-name-specifier non-qualifié-id* ;
- using :: *unqualified-id* ;

Remarques

Une *déclaration d'utilisation* est distincte d'une directive using , qui indique au compilateur de chercher dans un espace de nom particulier lors de la recherche d' *un* nom. Une *directive* `using namespace` commence par `using namespace` .

Une *déclaration d'utilisation* est également distincte d'une déclaration d'alias, qui donne un nouveau nom à un type existant de la même manière que `typedef` . Une déclaration d'alias contient un signe égal.

Examples

Importation de noms individuellement à partir d'un espace de noms

Une fois que l' `using` est utilisée pour introduire le nom `cout` de l'espace de noms `std` dans la portée de la fonction `main` , l'objet `std::cout` peut être appelé `cout` seul.

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Redéclarer les membres d'une classe de base pour éviter de les masquer

Si une *déclaration using* se produit à la portée de la classe, elle est uniquement autorisée à redéclarer un membre d'une classe de base. Par exemple, l' `using std::cout` n'est pas autorisée dans la portée de la classe.

Souvent, le nom redéclaré est un nom qui serait autrement caché. Par exemple, dans le code ci-dessous, `d1.foo` fait uniquement référence à `Derived1::foo(const char*)` et une erreur de compilation se produira. La fonction `Base::foo(int)` n'est pas du tout prise en compte. Cependant, `d2.foo(42)` convient bien car la *déclaration using* apporte `Base::foo(int)` dans l'ensemble des

entités nommées `foo` dans `Derived2` . La recherche de nom trouve alors `foo` s et la résolution de surcharge sélectionne `Base::foo` .

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42);  // error
    Derived2 d2;
    d2.foo(42);  // OK
}
```

Héritage des constructeurs

C ++ 11

En tant que cas particulier, une *déclaration d'utilisation* à la portée d'une classe peut faire référence aux constructeurs d'une classe de base directe. Ces constructeurs sont ensuite *hérités* par la classe dérivée et peuvent être utilisés pour initialiser la classe dérivée.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

Dans le code ci-dessus, `Derived1` et `Derived2` ont `Derived2` deux des constructeurs qui transmettent les arguments directement au constructeur correspondant de `Base` . `Derived1` effectue le transfert explicitement, alors que `Derived2` , utilisant la fonctionnalité C ++ 11 pour hériter des constructeurs, le fait implicitement.

Lire Utiliser la déclaration en ligne: https://riptutorial.com/fr/cplusplus/topic/9301/utiliser-la-declaration

Chapitre 146: Utiliser std :: unordered_map

Introduction

std :: unordered_map n'est qu'un conteneur associatif. Cela fonctionne sur les clés et leurs cartes. Key comme les noms va, aide à avoir l'unicité dans la carte. Bien que la valeur mappée ne soit qu'un contenu associé à la clé. Les types de données de cette clé et de cette carte peuvent être l'un des types de données prédéfinis ou définis par l'utilisateur.

Remarques

Comme son nom l'indique, les éléments de la carte non ordonnée ne sont pas stockés dans une séquence triée. Ils sont stockés en fonction de leurs valeurs de hachage et, par conséquent, l'utilisation de cartes non ordonnées présente de nombreux avantages, par exemple il suffit de O (1) pour rechercher n'importe quel élément. Il est également plus rapide que les autres conteneurs de carte. Il est également visible de l'exemple qu'il est très facile à mettre en œuvre car l'opérateur ([]) nous aide à accéder directement à la valeur mappée.

Examples

Déclaration et utilisation

Comme déjà mentionné, vous pouvez déclarer une carte non ordonnée de n'importe quel type. Ayons une carte non ordonnée nommée en premier avec la chaîne et le type entier.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Quelques fonctions de base

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value

variable_name.begin(); // iterator to the first element

variable_name.end(); // iterator to the last + 1 element
```

Lire Utiliser std :: unordered_map en ligne: https://riptutorial.com/fr/cplusplus/topic/10540/utiliser-std----unordered-map

Chapitre 147: Variables en ligne

Introduction

Une variable en ligne peut être définie dans plusieurs unités de traduction sans violer la règle à une définition . S'il est défini par une multiplication, l'éditeur de liens fusionnera toutes les définitions en un seul objet dans le programme final.

Examples

Définition d'un membre de données statique dans la définition de classe

Un membre de données statique de la classe peut être entièrement défini dans la définition de classe s'il est déclaré en `inline` . Par exemple, la classe suivante peut être définie dans un en-tête. Avant C ++ 17, il aurait été nécessaire de fournir un fichier `.cpp` contenant la définition de `Foo::num_instances` pour qu'il ne soit défini qu'une seule fois, mais en C ++ 17 les multiples définitions de la variable inline `Foo::num_instances` réfèrent tous au même objet `int` .

```
// warning: not thread-safe...
class Foo {
  public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

En tant que cas particulier, un membre de données statique `constexpr` est implicitement intégré.

```
class MyString {
  public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};
// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

Lire Variables en ligne en ligne: https://riptutorial.com/fr/cplusplus/topic/9265/variables-en-ligne

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec C ++	Adhokshaj Mishra, ankit dassor, aquirdturtle, ArchbishopOfBanterbury, Bakhtiar Hasan, Bart van Nierop, Ben H, Bo Persson, Brandon, Brian, BullshitPingu, cb4, celtschk, Cheers and hth. - Alf, chrisb2244, Cody Gray, Community, cpatricio, Curious, Daemon, Daksh Gupta, Danh, darkpsychic, David Bippes, David G., DeepCoder, Dim_ov, dlemstra, Donald Duck, Dr t, Dylan Little, Edward, emlai, Erick Q., ethanwu10, Fantastic Mr Fox, Florian, GIRISH kuniyal, greatwolf, honk, Humam Helfawi, Hurkyl, Ilyas Mimouni, Isak Combrinck, itzmukeshy7, Jason Watkins, JedaiCoder, Jerry Coffin, Jim Clark, Johan Lundberg, Jon Harper, jotik, Justin, Justin Time, JVApen, K48, Ken Y-N, Keshav Sharma, kiner_shah, krOoze, Leandros, maccard, Malcolm, Malick, Manan Sharma, manetsus, manlio, Marco A., Mark Gardner, MasterHD, Matt, Matt Lord, mnoronha, Muhammad Aladdin, Mustaghees, muXXmit2X, mynameisausten, Nathan Osman, Neil A., Nemanja Boric, neuro, Nicol Bolas, noɥʇʎdʎzɐɹƆ, Optimus Prime, Pavel Strakhov, Peter, Praetorian, Qchmqs, Quirk, RamenChef, Rushikesh Deshpande, SajithP, Sam Cristall, Serikov, Shoe, SirGuy, Soapy, Soul_man, theo2003, ʇolɐəz əɥʇ qoq, Tom K, TriskalJM, Trizzle, UncleZeiv, VermillionAzure, Walter, Wen Qin, Wexiwa, πάντα ῥεῖ, パスカル
2	Algorithmes de bibliothèque standard	Ami Tavory, Barry, Daniel, Duly Kinsky, Edgar Rokyan, Guillaume Pascal, Jarod42, NinjaDeveloper, Patryk Obara, Peter, Riom
3	Alignement	Brian, Marco A., Nicol Bolas
4	Arithmétique en virgule flottante	Xirema
5	auto	Artalus, Barry, celtschk, Daniele Pallastrelli, Edward, Igor Oks, Jarod42, Johan Lundberg, manlio, Yakk
6	Bit Manipulation	A. Sarid, Barry, Cody Gray, CroCo, FedeWar, Jarod42, JVApen, manlio, tambre, Tarod, Trevor Hickey, Алексей Неудачин
7	Boucles	ankit dassor, anotherGatsby, Barry, ChemiCalChems, Chris, ChrisN, Christian Rau, ColleenV, Debanjan Dhar, DrZoo, Edward, emlai, holmicz, honk, Johannes Schaub - litb, Justin

		Time, L.V.Rao, manlio, Nicholas, Nicol Bolas, Null, Ped7g, pmelanson, Pyves, Rakete1111, Sergey, sp2danny, user1336087, VladimirS, Yakk
8	C incompatibilités	パスカル
9	Catégories de valeur	Barry, ChemiCalChems, Curious, fefe, Johannes Schaub - litb, mnoronha, Nicol Bolas, Praetorian, SirGuy
10	Champs de bits	Ajay, Perette Barella
11	Classes / Structures	Alexey Voytenko, anderas, aquirdturtle, Brian, callyalater, chrisb2244, Colin Basnett, Dan Hulme, darkpsychic, Dragma, Fantastic Mr Fox, Firas Moalla, Jarod42, Jerry Coffin, jotik, Justin Time, Kerrek SB, Nicol Bolas, Null, OliPro007, PcAF, Ph03n1x, pingul, Rakete1111, Sándor Mátyás Márton, Sergey, silvergasp, Skywrath, Yakk
12	Compiler et construire	4444, Adhokshaj Mishra, Ami Tavory, ArchbishopOfBanterbury, Barry, Ben Steffan, celtschk, Curious, Donald Duck, Dr t, elvis.dukaj, Fantastic Mr Fox, Florian, greatwolf, Griffin, Isak Combrinck, Jahid, Jarod42, Jason Watkins, Johan Lundberg, jotik, Justin, Justin Time, JVApen, madduci, Malick, manetsus, manlio, Matt, Michael Gaskill, Morten Kristensen, MSD, muXXmit2X, n.m., Nathan Osman, Nemanja Boric, Peter, Quirk, Richard Dally, Sergey, Tharindu Kumara, Toby, Trygve Laugstøl , VermillionAzure
13	Comportement défini par la mise en œuvre	2501, Bo Persson, Brian, Dutow, Jahid, Jarod42, jotik, Justin Time, lz96, manlio, Nicol Bolas, Peter
14	Comportement non défini	Ami Tavory, AndreiM, Ben Steffan, Brian, Cody Gray, cshu, Dovahkiin, Elias Kosunen, emlai, Emma X, FedeWar, fefe, ggrr, GIRISH kuniyal, Hiura, Jeremi Podlasek, Johannes Schaub - litb , JVApen, kd1508, Ken Y-N, manetsus, manlio, Marco A., Mat, mceo, Motti, Naor Hadar, nbro, Nicol Bolas, Peter, Rakete1111, ralismark, RamenChef, Sebastian Ärleryd, Tannin, Trevor Hickey, Tyler Durden
15	Comportement non spécifié	AndreiM, Brian, Jarod42, Yakk
16	Comportements plus indéfinis en C ++	didiz
17	Concurrence avec OpenMP	Andrea Chua, JVApen, Nicol Bolas, Sumurai8

18	constexpr	Ajay, Brian, diegodfrf, mtb, Null
19	Conteneurs C ++	John DiFini
20	Contrôle de flux	anotherGatsby, Brian, JVApen, mkluwe, Qchmqs, RamenChef, Tejendra
21	Conversions de type explicites	4444, Brian, JVApen, Nikola Vasilev
22	Copier Elision	Nicol Bolas, TartanLlama
23	Copier vs assignation	amanuel2, Roland
24	Correct Correct	amanuel2, Justin Time
25	Date et heure en utilisant entête	Edward, marcinj, Naor Hadar, RamenChef
26	decltype	Ajay
27	déduction de type	Barry, Brian, Emmanuel Mathi-Amorim
28	Déplacer la sémantique	Barry, Cheers and hth. - Alf, ChemiCalChems, David Doria, didiz , Guillaume Racicot, Justin Time, JVApen
29	Des exceptions	Alexey Guseynov, Brian, callyalater, Dr t, Jahid, Jarod42, Johan Lundberg, jotik, Martin Ba, Nemanja Boric, Null, Peter, Rakete1111, Ronen Ness
30	Disposition des types d'objet	Brian, Justin Time
31	Entrée / sortie basique en c ++	Daemon, Nicol Bolas, Владимир Стрелец
32	Énumération	Denkkar, Fantastic Mr Fox, Jarod42, Nicol Bolas, SajithP, stackptr, T.C.
33	Erreurs courantes de compilation / édition de liens (GCC)	Asu, immerhart
34	Espaces de noms	anderas, Andrea Chua, Barry, Brian, DeepCoder, emlai, Isak Combrinck, Jarod42, Jérémy Roy, Johannes Schaub - litb, Julien-L, JVApen, Nicol Bolas, Null, Rakete1111, randag, Roland, T.C., tenpercent, Yakk

35	Exemples de serveur client	Abhinav Gauniyal
36	Expressions régulières	honk, Jonathan Mee, Justin, JVApen
37	Fichier I / O	anderas, ankit dassor, Anonymous1847, AProgrammer, Bakhtiar Hasan, bitek, Chachmu, ComicSansMS, didiz, Dietmar Kühl, Dr t , Emanuel Vintilă, Galik, honk, Hurkyl, Jérémie Bolduc, John Strood, JVApen, Loki Astari, manlio, Mathieu K., MikeMB, mindriot, Nicol Bolas, nwp, patmanpato, Rakete1111, RomCoo, Serikov, sheng09, shrike, svgspnr, Алексей Неудачин
38	Fichiers d'en-tête	RamenChef, VermillionAzure
39	Filetage	Alejandro, amchacon, Brian, CaffeineToCode, ComicSansMS, Dair, defube, didiz, Diligent Key Presser, Galik, James Adkison, james large, Jason Watkins, Jeremi Podlasek, mpromonet, Niall, nwp, Rakete1111, Stephen Cross, Sumurai8, Yakk, ysdx, Yuushi
40	Flux C ++	Ami Tavory, didiz, JVApen, mpromonet, Sergey
41	Fonction C ++ "appel par valeur" vs. "appel par référence"	Error - Syntactical Remorse, Henkersmann
42	Fonction de surcharge	Bakhtiar Hasan, Barry, Cody Gray, CoffeeandCode, didiz, Galik, Jatin, Johannes Schaub - litb, Justin Time, JVApen, Rakete1111 , Sean, Sumurai8, Tim Straubinger
43	Fonctions en ligne	amanuel2, Aravind .KEN, Bim, Brian, legends2k
44	Fonctions membres de classe constante	Vijayabhaskarreddy CH, Yakk
45	Fonctions membres non statiques	Justin Time, RamenChef
46	Fonctions membres spéciales	Barry, krOoze, OliPro007, Reuben Thomas, TriskalJM
47	Fonctions membres virtuelles	0x5f3759df, Daksh Gupta, Johan Lundberg, Justin Time, Motti, Sergey, T.C.
48	Futures et promesses	didiz, Nicol Bolas
49	Génération de	Ha., manlio, merlinND, Sumurai8

	nombres aléatoires	
50	Gestion de la mémoire	Andrei, Brian, callyalater, Daksh Gupta, Galik, JVApen, madduci, nnrales, RamenChef, ThyReaper
51	Implémentation du modèle de conception en C ++	Antonio Barreto, datosh, didiz, Jarod42, JVApen, Nikola Vasilev
52	Internationalisation en C ++	John Bargman
53	Itération	Brian, Daniel Käfer, Emmanuel Mathi-Amorim, marquesm91, RamenChef
54	La gestion des ressources	Anonymous1847
55	La norme ISO C ++	Bakhtiar Hasan, Barry, C.W.Holeman II, ComicSansMS, didiz, diegodfrf, Guillaume Pascal, Ivan Kush, Johan Lundberg, Justin Time, JVApen, manlio, Marco A., MSalters, Nicol Bolas, sth, vishal
56	La règle des trois, cinq et zéro	Adrien Descamps, Barry, ChrisN, hello, honk, Johan Lundberg, Justin Time, JVApen, Loki Astari, mpromonet, Nicol Bolas, Nirmal4G, NonNumeric, Null, Peter, relgukxilef, Scott Weldon, T.C., TriskalJM, Venemo
57	Lambdas	Adi Lester, Aganju, Ajay, alain, anderas, Andrea Corbelli, Barry, bcmpinc, Brian, Christopher Oezbek, Community, cpplearner, derekerdmann, Edd, Falias, Firas Moalla, honk, Jean-Baptiste Yunès, Johan Lundberg, Johannes Schaub - litb, John Slegers, JVApen, Loki Astari, Loufylouf, M. Viaz, Mike Dvorkin, Nicol Bolas, Patryk, Praetorian, Rakete1111, RamenChef, Ryan Haining, Sergio, Serikov, Snowhawk, teivaz, Yakk, ygram
58	Le pointeur	amanuel2, Justin Time, RamenChef
59	Les attributs	ibrahim5253, JVApen, Kerrek SB, MathSquared, SingerOfTheFall
60	Les itérateurs	Barry, chrisb2244, cute_ptr, Daniel Jour, Edgar Rokyan, EvgeniyZh, fbrereto, Gal Dreiman, Gaurav Kumar Garg, GIRISH kuniyal, honk, Hurkyl, JPNotADragon, JVApen, Mike H-R, Null, Oz., Sergey, Serikov, tilz0R, Yakk
61	Les portées	deepmax, Error - Syntactical Remorse
62	Les références	Andrea Corbelli, Asu, Daksh Gupta, darkpsychic, rockoder

63	Les syndicats	manlio, ThyReaper, txtechhelp
64	Littéraux	Brian, Nikola Vasilev, RamenChef
65	Littéraux définis par l'utilisateur	Brian, Cid1025, Jarod42, Roland, sigalor, sth
66	Manipulateurs de flux	Nicol Bolas, Владимир Стрелец
67	Métaprogrammation	anderas, Barry, Brian, celtschk, Colin Basnett, DawidPi, deepmax, dmi_, Holt, Jarod42, Justin, manlio, Matthieu M., Nicol Bolas, Oz., rhynodegreat, rtmh, sth, TartanLlama, Venki, W.F., ysdx, πάντα ῥεῖ
68	Métaprogrammation arithmitique	Meena Alfons
69	Modèle de mémoire C ++ 11	NonNumeric
70	Modèle de modèle curieusement récurrent (CRTP)	Barry, Brian, Gabriel, honk, Nicol Bolas, Ryan Haining
71	Modèles	Barry, Benjy Kessler, Brian, callyalater, cb4, celtschk, CodeMouse92, Colin Basnett, DeepCoder, Diligent Key Presser, Eldritch Cheese, eXPerience, FedeWar, Gabriel, Greg, Holt, honk, J_T, Johannes Schaub - litb, Justin, JVApen, Loki Astari, M. Viaz, manlio, Maxito, MSalters, Nicol Bolas, Pontus Gagge, Praetorian, Rakete1111, Ricardo Amores, Ryan Haining, Sergey , SirGuy, Smeeheey, Sumurai8, user1887915, W.F., WMios, Wolf, πάντα ῥεῖ
72	Modèles d'expression	Ajay, BigONotation, celtschk, defube, Jarod42, Jonathan Lee, Roland, T.C., Yakk
73	Mot clé ami	Perette Barella, Sergey
74	mot clé const	Barry, Jarod42, Jatin, Justin, Podgorskiy, tenpercent, ThyReaper
75	mot-clé mutable	Barry, Community, Dean Seo, start2learn, T.C., tenpercent
76	Mots clés	ADITYA, amanuel2, Brian, Danh, John London, Justin Time, JVApen, Kerrek SB, Loki Astari, manlio, Marco A., Nicol Bolas, OliPro007, Rakete1111, RamenChef, Roland, start2learn
77	Mots-clés de déclaration de variable	Brian, RamenChef, start2learn

78	Mots-clés de type	Brian, Justin Time, Omnifarious, RamenChef
79	Mots-clés de type de base	amanuel2, Brian, Kerrek SB, RamenChef
80	Mutex récursif	didiz
81	Mutexes	didiz, hyoslee, JVApen
82	Objets appelables	JVApen, turoni
83	Opérateurs de bits	Loki Astari, Mads Marquart, manlio, txtechhelp, Алексей Неудачин
84	Optimisation	chema989, ralismark
85	Optimisation en C ++	4444, JVApen, lorro, mindriot
86	Outils et techniques de débogage et de prévention du débogage C ++	Adam Trhon, JVApen, King's jester, Misgevolution
87	Pack de paramètres	Marco A.
88	Pimpl Idiom	Danh, Daniele Pallastrelli, emlai, Jordan Chapman, JVApen, manlio, Stephen Cross, Yakk
89	Plier les expressions	AndyG, Barry, cpplearner, Firas Moalla, Marco A., Rakete1111, T.C., Yakk
90	Pointeurs	Baron, daB0bby, FedeWar, Hindrik Stegenga, Nicol Bolas, Nitinkumar Ambekar, Pietro Saccardi, Reverie Wisp, West
91	Pointeurs aux membres	John Burger, start2learn
92	Pointeurs intelligents	Abyx, Ajay, Alexey Voytenko, anderas, Barry, CaffeineToCode, Christopher Oezbek, Cody Gray, ComicSansMS, cpplearner, Daksh Gupta, Danh, Daniele Pallastrelli, DeepCoder, Edward, emlai, foxcub, Francis Cugler, honk, Jack Zhou, Jared Payne, Jarod42, Johan Lundberg, Johannes Schaub - litb, jotik, Justin, JVApen, Kerrek SB, King's jester, Loki Astari, manlio, Marco A., MC93, Menasheh, Meysam, PcAF, Rakete1111, Reuben Thomas, Richard Dally, rodrigo, Roland, sami1592, sth, Sumurai8, tysonite, user3684240, Xirema, Yakk
93	Polymorphisme	A. Sarid, Christophe, Jarod42, Jeremi Podlasek, Justin Time,

		manlio
94	Préprocesseur	alain, callyalater, Cheers and hth. - Alf, CygnusX1, Fantastic Mr Fox, Fox, Francisco P., Ian Ringrose, immerhart, InitializeSahib, Johan Lundberg, Justin, Justin Time, Ken Y-N, Kieran Chandler, krOoze, manlio, Marco A., Maxito, n.m., Nicol Bolas, Peter, phandinhlan, Richard Dally, Sean, signal, silvergasp, Sumurai8, T.C., Tanjim Hossain, tenpercent, The Philomath, Владимир Стрелец, パスカル
95	priorité de l'opérateur	an0o0nym, Brian, didiz, JVApen, start2learn, turoni
96	Profilage	Ami Tavory, paul-g
97	RAII: l'acquisition de ressources est une initialisation	Barry, defube, Jarod42, JVApen, Loki Astari, Niall, Nicol Bolas, RamenChef, Sumurai8, Tannin
98	Recherche de nom dépendante de l'argument	Fanael, Johannes Schaub - litb
99	Récursivité en C ++	celtschk, R_Kapp
100	Renvoyer plusieurs valeurs d'une fonction	aaronsnoswell, Bakhtiar Hasan, Barry, bitek, celtschk, Christopher Oezbek, Community, DeepCoder, Dr t, Ela782, Fantastic Mr Fox, Galik, honk, J_T, Jarod42, Johan Lundberg, Johannes Schaub - litb, John Slegers, Jon Chesterfield, Kevin Katzke, Let_Me_Be, Loki Astari, M. Sadeq H. E., manetsus, Menasheh, Michael Gaskill, mnoronha, Niall, Nicol Bolas, Null, Peter, Rakete1111, Richard Forrest, Ryan Hilbert, Stephen, T.C. , templatetypedef, tenpercent, user3384414, Yakk, Ze Rubeus, パスカル
101	Résolution de surcharge	Barry, Brian, didiz, Johannes Schaub - litb
102	RTTI: Informations sur le type d'exécution	Brian, deepmax, Pankaj Kumar Boora, Roland, Savas Mikail KAPLAN
103	Sémantique de valeur et de référence	JVApen, Nicol Bolas
104	Sémaphore	didiz
105	Séparateurs de	diegodfrf, JVApen

	chiffres	
106	SFINAE (échec de substitution n'est pas une erreur)	Barry, Fox, Jarod42, Jason R, Jonathan Lee, Luc Danton, sp2danny, SU3, w1th0utnam3, Xosdy, Yakk
107	Side by Side Comparaisons des exemples classiques en C ++ résolus via C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17	wasthishelpful
108	Singleton Design Pattern	deepmax, Galik, Jarod42, Johan Lundberg, JVApen, Stradigos
109	Spécificateurs de classe de stockage	Brian, start2learn
110	Spécifications de liaison	Brian
111	static_assert	Jarod42, JVApen, lorro, Marco A., Richard Dally, T.C.
112	std :: any	demonplus, Marco A.
113	std :: array	CinCout, Daksh Gupta, Dinesh Khandelwal, Error - Syntactical Remorse, Malcolm, Nikola Vasilev, plasmacel
114	std :: atomics	Nikola Vasilev
115	std :: carte	Andrea Corbelli, ankit dassor, ChrisN, CinCout, ComicSansMS, CygnusX1, davidsheldon, diegodfrf, Fantastic Mr Fox, foxcub, Galik, honk, jmmut, manetsus, manlio, Meysam, Naveen Mittal, Null, Peter, Richard Dally, rick112358, Savan Morya, user1336087, vdaras, VolkA, Wyzard
116	std :: forward_list	Nikola Vasilev
117	std :: function: Pour envelopper n'importe quel élément appelable	elimad, Evgeniy, Nicol Bolas, Tarod
118	std :: integer_sequence	Dietmar Kühl
119	std :: iomanip	kiner_shah, Nikola Vasilev, Yakk

120	std :: optionnel	Barry, diegodfrf, Jahid, Jared Payne, JVApen, Null, Yakk
121	std :: pair	Ajay, Bim, demonplus, kiner_shah, Nikola Vasilev, Ravi Chandra
122	std :: set et std :: multiset	G-Man, JVApen, Mikitori
123	std :: string	1337ninja, 3442, Andrea Corbelli, Barry, Bim, caps, Christopher Oezbek, cpplearner, crea7or, Curious, drov, Edward, Emil Rowland, emlai, Fantastic Mr Fox, fbrereto, ggrr, Holt, honk, immerhart, Jack, Jahid, Jerry Coffin, Jonathan Mee, jotik, JPNotADragon, jpo38, Justin, JVApen, Ken Y-N, Leandros, Loki Astari, manetsus, manlio, Marc.2377, Matthew, Matthieu M., Meysam, Michael Gaskill, mpromonet, Niall, Null, Rakete1111, RamenChef, Richard Dally, SajithP, Serikov, sigalor, Skipper, Soapy, sth, T.C., Tharindu Kumara, Trevor Hickey, user1336087 , user2176127, W.F., Wolf, Yakk
124	std :: variant	Yakk
125	std :: vector	2power10, A. Sarid, Aaron Stein, alain, Alex Logan, Ami Tavory, anatolyg, anderas, Andy, AndyG, arunmoezhi, Bakhtiar Hasan, Barry, Benjamin Lindley, bluefog, bone, CHess, CinCout, Cody Gray, Colin Basnett, ComicSansMS, Community, cute_ptr, Daksh Gupta, Daniel, Daniel Stradowski, Dario, David G., David Yaw, DeepCoder, diegodfrf, dkg, Dr t, Duly Kinsky, Ed Cottrell, Edward, ehudt, emlai, enrico.bacis, Falias, Fantastic Mr Fox, Fox, foxcub, gaazkam, Galik, gartenriese, granmirupa, Holt, honk, Hurkyl, iliketocode, immerhart, Isak Combrinck, Jarod42, Jason Watkins, JHBonarius, Johan Lundberg, John Slegers, jotik, jpo38, JVApen, Kevin Katzke, krOoze, Loki Astari, lordjohncena, manetsus, manlio, Marco A., Matt, Michael Gaskill , Misha Brukman, MotKohn, Motti, mtk, NageN, Niall, Nicol Bolas , Null, patmanpato, Paul Beckingham, paul-g, Ped7g, Praetorian, Pyves, R. Martinho Fernandes, Rakete1111, Randy Taylor, Richard Dally, Roddy, Romain Vincent, Rushikesh Deshpande, Ryan Hilbert, Saint-Martin, Samar Yadav, Samer Tufail, Sayakiss, Serikov, Shoe, silvergasp, Skipper, solidcell, Stephen, sth, strangeqargo, T.C., Tamarous, theo2003, Tom, towi, Trevor Hickey, TriskalJM, user1336087, user2176127, Vladimir Gamalyan, Wolf, Yakk
126	Structures de données en C ++	Gaurav Sehgal
127	Structures de synchronisation de fil	didiz, Galik, JVApen

128	Surcharge de l'opérateur	ArchbishopOfBanterbury, Archie Gertsman, Ates Goral, Barry, Brian, Candlemancer, chrisb2244, defube, enzom83, James Adkison, Rakete1111, Sergey, start2learn, Xeverous, Yakk
129	Surcharge du modèle de fonction	Johannes Schaub - litb, Kunal Tyagi, RamenChef
130	Systèmes de construction	Ami Tavory, celtschk, Florian, Jahid, Jason Watkins, Justin, JVApen, Nathan Osman, RamenChef, VermillionAzure
131	Tableaux	Cheers and hth. - Alf, Isak Combrinck, manlio, Matthew Brien, Wen Qin, Wolf, ΦХocę Пepeúpa ツ
132	Techniques de refactoring	asantacreu, Cody Gray, Edward, Jarod42, JVApen, RamenChef
133	Test d'unité en C ++	elvis.dukaj, VermillionAzure
134	Transmission parfaite	In silico, Johannes Schaub - litb, Nicol Bolas, Roland
135	Tri	anatolyg, Barry, Daniel, Ivan Kush, maccard, manetsus, manlio, MikeMB, MKAROL, Nicol Bolas, Patrick, Ravi Chandra, SajithP, timrau, Trevor Hickey
136	Type de retour	Brian, define cindy const, Torbjörn
137	Type de retour Covariance	Cheers and hth. - Alf, sorosh_sabz
138	Type effacement	Brian, celtschk, greatwolf, Jarod42, Yakk
139	Type Inférence	Andrea Chua, Jim Clark
140	Type Traits	Jarod42, silvergasp, TartanLlama
141	Typedef et alias de type	Brian
142	Types atomiques	JVApen, Stephen
143	Types sans nom	jotik, Roland, ThyReaper
144	Une règle de définition (ODR)	Brian, Jarod42
145	Utiliser la déclaration	Brian

146	Utiliser std :: unordered_map	tulak.hord
147	Variables en ligne	Brian