APPRENEZ C Language

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

#c

Table des matières

À propos ... 1

Chapitre 1: Démarrer avec le langage C ... 2

Remarques ... 2

Compilateurs communs ... 2

Compiler C version Support ... 2

Style de code (hors sujet ici): ... 3

Bibliothèques et API non couvertes par la norme C (et donc hors sujet ici): ... 4

Versions ... 4

Examples ... 4

Bonjour le monde ... 4

Bonjour c ... 4

Regardons ce programme simple ligne par ligne ... 5

Modifier le programme ... 6

Compiler et exécuter le programme ... 6

Compiler en utilisant GCC ... 6

Utiliser le compilateur clang ... 6

Utilisation du compilateur Microsoft C à partir de la ligne de commande ... 7

Exécuter le programme ... 7

Original "Bonjour, Monde!" dans K & R C ... 7

Chapitre 2: - classification et conversion des personnages ... 9

Examples ... 9

Classification des caractères lus dans un flux ... 9

Classification des caractères d'une chaîne ... 9

introduction ... 10

Chapitre 3: Affirmation ... 13

Introduction ... 13

Syntaxe ... 13

Paramètres ... 13

Remarques ... 13

Examples 14
Condition préalable et postcondition 14
Assertion simple 15
Affirmation statique 15
Affirmation de code inaccessible 16
Assert Messages d'erreur 17
Chapitre 4: Arguments de ligne de commande 19
Syntaxe 19
Paramètres 19
Remarques 19
Examples 20
Impression des arguments de la ligne de commande 20
Imprimer les arguments dans un programme et convertir en valeurs entières 21
Utiliser les outils GNU getopt 21
Chapitre 5: Arguments variables 25
Introduction 25
Syntaxe 25
Paramètres 25
Remarques 26
Examples 26
Utilisation d'un argument de décompte explicite pour déterminer la longueur de la va_list 26
Utiliser des valeurs de terminateur pour déterminer la fin de va_list 27
Implémenter des fonctions avec une interface semblable à `printf ()` 28
Utiliser une chaîne de format 31
Chapitre 6: Assemblage en ligne 33
Remarques 33
Avantages 33
Les inconvénients 33
Examples 33
gcc Basic asm support 33
gcc Support asm étendu 34
Assemblage en ligne gcc dans les macros 35

Chapitre 7: Atomique 37
Syntaxe 37
Remarques 37
Examples 38
atomiques et opérateurs 38
Chapitre 8: Booléen 39
Remarques 39
Examples 39
Utiliser stdbool.h 39
Utiliser #define 39
Utilisation de _Bool de type intrinsèque (intégré) 40
Entiers et pointeurs dans les expressions booléennes. 40
Définir un type bool en utilisant typedef 41
Chapitre 9: Champs de bits 43
Introduction 43
Syntaxe 43
Paramètres 43
Remarques 43
Examples 43
Champs de bits 43
Utilisation de champs de bits sous forme de petits entiers 45
Alignement du champ binaire 45
Quand les champs de bits sont-ils utiles? 46
À ne pas faire pour les champs de bits 47
Chapitre 10: Classes de stockage 49
Introduction 49
Syntaxe 49
Remarques 49
Durée de stockage 50
Durée de stockage statique 51
Durée de stockage des threads 51

Durée de stockage automatique 51
Liaison externe et interne 51
Examples 52
typedef 52
auto 52
statique 53
externe 54
registre 55
_Thread_local 56
Chapitre 11: commentaires 57
Introduction 57
Syntaxe 57
Examples 57
/ * * / commentaires délimités 57
// commentaires délimités 58
Commenter en utilisant le préprocesseur 58
Piège possible dû aux trigraphes 59
Chapitre 12: Communication interprocessus (IPC) 60
Introduction 60
Examples 60
Sémaphores 60
Exemple 1.1: Course avec des threads 61
Exemple 1.2: Évitez les courses avec les sémaphores 62
Chapitre 13: Compilation 65
Introduction 65
Remarques 65
Examples 66
Le lieur 67
Invocation implicite de l'éditeur de liens 67
Invocation explicite de l'éditeur de liens 67
Options pour l'éditeur de liens 67
Autres options de compilation 68

Types de fichier 68

Le préprocesseur 70

Le compilateur 72

Les phases de traduction 73

Chapitre 14: Comportement défini par la mise en œuvre 74

Remarques 74

Vue d'ensemble 74

Programmes et processeurs 74

Général 74

Traduction source 75

Environnement d'exploitation 75

Les types 76

Formulaire source 77

Évaluation 77

Comportement d'exécution 77

Préprocesseur 78

Bibliothèque standard 79

Général 79

Fonctions d'environnement à virgule flottante 79

Fonctions liées aux paramètres régionaux 79

Fonctions mathématiques 79

Les signaux 80

Divers 80

Fonctions de traitement de fichiers 80

Fonctions d'E / S 81

Fonctions d'allocation de mémoire 81

Fonctions d'environnement système 81

Fonctions de date et heure 82

Fonctions d'E / S à caractères larges 82

Examples 82

Décalage à droite d'un entier négatif 82

Affectation d'une valeur hors plage à un entier 83

Allouer zéro octet 83

Représentation d'entiers signés 83
Chapitre 15: Comportement non défini 84
Introduction 84
Remarques 84
Examples 86
Déréférencer un pointeur nul 86
Modifier un objet plus d'une fois entre deux points de séquence 86
Déclaration de retour manquante dans la fonction de retour de valeur 87
Débordement d'entier signé 88
Utilisation d'une variable non initialisée 89
Déréférencer un pointeur à variable au-delà de sa durée de vie 90
Division par zéro 91
Accéder à la mémoire au-delà du bloc attribué 91
Copie de mémoire superposée 92
Lecture d'un objet non initialisé qui n'est pas soutenu par la mémoire 93
Course de données 93
Valeur de lecture du pointeur libéré 95
Modifier le littéral de chaîne 95
Libérer deux fois la mémoire 96
Utiliser un spécificateur de format incorrect dans printf 96
La conversion entre les types de pointeurs produit un résultat incorrectement aligné 96
Ajout ou soustraction de pointeur non borné correctement 97
Modification d'une variable const à l'aide d'un pointeur 98
Passer un pointeur nul à la conversion de printf% s 98
Liaison incohérente d'identificateurs 99
Utiliser fflush sur un flux d'entrée 99
Déplacement de bits en utilisant des nombres négatifs ou au-delà de la largeur du type 100
Modification de la chaîne renvoyée par les fonctions getenv, strerror et setlocale 100
Retour d'une fonction déclarée avec le spécificateur de fonction `_Noreturn` ou` noreturn` 101
Chapitre 16: Contraintes 103
Remarques 103
Examples 103

Noms de variables en double dans la même portée....103
Opérateurs arithmétiques unaires....104
Chapitre 17: Conversions implicites et explicites....105
Syntaxe....105
Remarques....105
Examples....105
Conversions entières dans les appels de fonction....105
Conversions du pointeur dans les appels de fonction....106
Chapitre 18: Cordes....108
Introduction....108
Syntaxe....108
Examples....108
Calculez la longueur: strlen ()....108
Copie et concaténation: strcpy (), strcat ()....109
Comparaison: strcmp (), strncmp (), strcasecmp (), strncasecmp ()....110
Tokenisation: strtok (), strtok_r () et strtok_s ()....112
Rechercher la première / dernière occurrence d'un caractère spécifique: strchr (), strrchr....114
Itération sur les caractères d'une chaîne....116
Introduction de base aux chaînes....116
Création de tableaux de chaînes....117
strstr....118
Littéraux de chaîne....119
Remettre une chaîne à zéro....120
strspn et strcspn....121
Copier des chaînes....122
Les affectations de pointeur ne copient pas les chaînes....122
Copie de chaînes à l'aide de fonctions standard....123
strcpy()....123
snprintf()....123
strncat()....124
strncpy()....124
Convertir les chaînes en nombre: atoi (), atof () (dangereux, ne les utilisez pas)....125

données formatées en chaîne lecture / écriture 126
Convertir en toute sécurité des chaînes en nombre: fonctions strtoX 127
Chapitre 19: Créer et inclure des fichiers d'en-tête 129
Introduction 129
Examples 129
introduction 129
Idempotence 130
Gardes de tête 130
La directive #pragma once 130
Auto-confinement 131
Recommandation: les fichiers d'en-tête doivent être autonomes. 131
Règles historiques 131
Règles modernes 131
Vérification de l'auto-confinement 132
Minimalité 132
Inclure ce que vous utilisez (IWYU) 133
Notation et Divers 133
Références croisées 135
Chapitre 20: Déclaration vs définition 136
Remarques 136
Examples 136
Comprendre la déclaration et la définition 136
Chapitre 21: Déclarations 138
Remarques 138
Examples 138
Appeler une fonction depuis un autre fichier C 138
Utilisation d'une variable globale 139
Utiliser des constantes globales 140
introduction 142
Typedef 145
Utilisation de la règle de droite ou de spirale pour déchiffrer la déclaration C 145
Chapitre 22: Effets secondaires 150

Examples....150
Opérateurs avant / après incrémentation / décrémentation....150
Chapitre 23: Énoncés d'itération / boucles: pour, pendant et après....152
Syntaxe....152
Remarques....152
Relevé d'itération / boucles sous contrôle de la tête....152
Relevé d'itération / boucles contrôlées au pied....152
Examples....152
Pour la boucle....152
En boucle....153
Boucle Do-While....153
Structure et flux de contrôle dans une boucle for....154
Boucles infinies....155
Loop Unrolling et Duff's Device....156
Chapitre 24: Entrée / sortie formatée....158
Examples....158
Impression de la valeur d'un pointeur sur un objet....158
Utiliser <inttypes.h> et uintptr_t....158
Histoire pré-standard:....159
Impression de la différence des valeurs de deux pointeurs sur un objet....159
Spécificateurs de conversion pour l'impression....160
La fonction printf ()....162
Modificateurs de longueur....162
Drapeaux de format d'impression....164
Chapitre 25: Énumérations....166
Remarques....166
Examples....166
Énumération simple....166
Exemple 1....166
Exemple 2....167
Typedef enum....168
Enumération avec valeur en double....169

énumération constante sans nom de type 169
Chapitre 26: Fichiers et flux d'E / S 171
Syntaxe 171
Paramètres 171
Remarques 171
Chaînes de mode: 171
Examples 172
Ouvrir et écrire dans un fichier 172
fprintf 173
Exécuter le processus 174
Récupère les lignes d'un fichier en utilisant getline () 174
Fichier d'entrée example.txt 175
Sortie 175
Exemple d'implémentation de getline() 176
Ouvrir et écrire dans un fichier binaire 178
fscanf () 179
Lire les lignes d'un fichier 180
Chapitre 27: Génération de nombres aléatoires 183
Remarques 183
Examples 183
Génération de nombres aléatoires de base 183
Génératrice à permutation permutée 184
Restreindre la génération à une plage donnée 185
Génération Xorshift 185
Chapitre 28: Gestion de la mémoire 187
Introduction 187
Syntaxe 187
Paramètres 187
Remarques 187
Examples 188
Libérer de la mémoire 188
Allouer de la mémoire 189

Allocation Standard....189
Zéro mémoire....190
Mémoire Alignée....190
Réaffecter la mémoire....191
Tableaux multidimensionnels de taille variable....192
realloc (ptr, 0) n'est pas équivalent à free (ptr)....193
Gestion de la mémoire définie par l'utilisateur....194
alloca: allouer de la mémoire sur la pile....195
Résumé....195
Recommandation....196
Chapitre 29: Idiomes de programmation C courants et pratiques de développeur....197
Examples....197
Comparer littéral et variable....197
Ne laissez pas la liste des paramètres d'une fonction vierge - utilisez void....197
Chapitre 30: Initialisation....201
Examples....201
Initialisation des variables en C....201
Initialisation des structures et des tableaux de structures....203
Utiliser des initialiseurs désignés....203
Initialiseurs désignés pour les éléments de tableau....203
Initialiseurs désignés pour les structures....204
Initialiseur désigné pour les syndicats....204
Initialiseurs désignés pour les tableaux de structures, etc.....205
Spécification de plages dans les initialiseurs de tableau....205
Chapitre 31: Inlining....207
Examples....207
Fonctions d'inline utilisées dans plus d'un fichier source....207
principal c:....207
source1.c:....207
source2.c:....207
headerfile.h:....208
Chapitre 32: Instructions de saut....210

Syntaxe 210
Remarques 210
Voir également 210
Examples 210
Utiliser goto pour sauter des boucles imbriquées 210
Utiliser le retour 211
Retourner une valeur 211
Ne rien retourner 212
Utiliser la pause et continuer 212
Chapitre 33: La gestion des erreurs 214
Syntaxe 214
Remarques 214
Examples 214
errno 214
strerror 214
perror 215
Chapitre 34: Les opérateurs 216
Introduction 216
Syntaxe 216
Remarques 216
Examples 218
Opérateurs relationnels 218
Est égal à "==" 218
Pas égal à "! =" 218
Ne pas "!" 219
Plus grand que ">" 219
Moins que "<" 219
Supérieur ou égal à "> =" 219
Inférieur ou égal à "<=" 219
Opérateurs d'affectation 220
Opérateurs arithmétiques 221

Arithmétique de base 221
Opérateur Additionnel 221
Opérateur de soustraction 222
Opérateur de multiplication 222
Opérateur de division 223
Opérateur Modulo 223
Opérateurs d'incrémentation / décrémentation 224
Opérateurs logiques 224
Logique ET 224
OU logique 225
Logique NON 225
Évaluation en court-circuit 225
Incrémenter / Décrémenter 226
Opérateur conditionnel / opérateur ternaire 226
Opérateur de virgule 227
Opérateur de casting 228
taille de l'opérateur 228
Avec un type comme opérande 228
Avec une expression comme opérande 228
Arithmétique du pointeur 229
Ajout de pointeur 229
Soustraction de pointeur 230
Opérateurs d'accès 230
Membre d'objet 230
Membre d'objet pointé 230
Adresse de 231
Déréférence 231
Indexage 231
Interchangeabilité de l'indexation 231
Opérateur d'appel de fonction 232
Opérateurs sur les bits 232
_Alignof 234

Comportement en court-circuit des opérateurs logiques 234

Chapitre 35: Les syndicats 237

Examples 237

Différence entre struct et union 237

Utiliser les unions pour réinterpréter les valeurs 237

Écrire à un membre du syndicat et lire d'un autre membre 238

Chapitre 36: Listes liées 240

Remarques 240

Liste liée individuellement 240

Structure de données 240

Liste doublement liée 240

Structure de données 240

Topoliges 240

Linéaire ou ouverte 240

Circulaire ou anneau 241

Procédures 241

Lier 241

Faire une liste liée de façon circulaire 241

Faire une liste liée linéairement 242

Insertion 242

Examples 243

Insertion d'un nœud au début d'une liste liée séparément 243

Explication pour l'insertion de nœuds 244

Insérer un noeud à la nième position 245

Inverser une liste chaînée 246

Explication de la méthode de la liste inversée 247

Une liste doublement liée 248

Chapitre 37: Littéraux composés 252

Syntaxe 252

Remarques 252

Examples 252

Définition / Initialisation des littéraux composés 252
Exemples de la norme C, C11-§6.5.2.5 / 9: 252
Littéral composé avec désignateurs 253
Littéral composé sans spécifier la longueur du tableau 253
Littéral composé dont la longueur de l'initialiseur est inférieure à celle spécifiée 254
Littéral composé en lecture seule 254
Littéral composé contenant des expressions arbitraires 254
Chapitre 38: Littéraux pour les nombres, les caractères et les chaînes 255
Remarques 255
Examples 255
Littéraux entiers 255
Littéraux de chaîne 256
Littéraux à virgule flottante 256
Littéraux de caractère 257
Chapitre 39: Mathématiques standard 259
Syntaxe 259
Remarques 259
Examples 259
Reste à virgule flottante double précision: fmod () 259
Précision simple et double virgule flottante en double précision: fmodf (), fmodl () 260
Fonctions d'alimentation - pow (), powf (), powl () 261
Chapitre 40: Multithreading 263
Introduction 263
Syntaxe 263
Remarques 263
Examples 263
C11 Threads exemple simple 263
Chapitre 41: Paramètres de fonction 265
Remarques 265
Examples 265
Utilisation de paramètres de pointeur pour renvoyer plusieurs valeurs 265

Passer des tableaux à des fonctions ... 265
Voir également ... 266
Les paramètres sont passés par valeur ... 266
Ordre d'exécution des paramètres de la fonction ... 267
Exemple de fonction renvoyant une structure contenant des valeurs avec des codes d'erreur ... 267
Chapitre 42: Passer des tableaux 2D à des fonctions ... 269
Examples ... 269
Passer un tableau 2D à une fonction ... 269
Utilisation de tableaux plats comme tableaux 2D ... 275
Chapitre 43: Pièges communs ... 277
Introduction ... 277
Examples ... 277
Mélanger des entiers signés et non signés dans des opérations arithmétiques ... 277
Écrire par erreur = au lieu de == lors de la comparaison ... 277
Utilisation inconsidérée de points-virgules ... 279
Oublier d'allouer un octet supplémentaire pour \ 0 ... 280
Oublier de libérer de la mémoire (fuites de mémoire) ... 280
Copier trop ... 282
Oubliant de copier la valeur de retour de realloc dans un fichier temporaire ... 282
Comparer des nombres à virgule flottante ... 283
Faire une mise à l'échelle supplémentaire dans l'arithmétique du pointeur ... 284
Les macros sont des remplacements de chaînes simples ... 285
Erreurs de référence non définies lors de la liaison ... 287
Incompréhension du tableau ... 289
Passer des tableaux non adjacents à des fonctions qui attendent des tableaux multidimensio ... 292
Utiliser des constantes de caractères au lieu de littéraux de chaîne et vice versa ... 293
Ignorer les valeurs de retour des fonctions de bibliothèque ... 294
Le caractère de nouvelle ligne n'est pas utilisé lors d'un appel scanf () classique ... 295
Ajouter un point-virgule à un #define ... 296
Les commentaires sur plusieurs lignes ne peuvent pas être imbriqués ... 297
Dépasser les limites du réseau ... 298
Fonction récursive - manquant la condition de base ... 299
Vérification de l'expression logique contre 'true' ... 301

Les littéraux à virgule flottante sont de type double par défaut 302
Chapitre 44: Pointeurs 303
Introduction 303
Syntaxe 303
Remarques 303
Examples 303
Erreurs courantes 303
Ne pas vérifier les échecs d'allocation 304
Utiliser des nombres littéraux au lieu de sizeof lors de la demande de mémoire 304
Fuites de mémoire 305
Erreurs logiques 305
Créer des pointeurs pour empiler des variables 305
Incrémenter / décrémenter et déréférencer 307
Déréférencer un pointeur 307
Déréférencer un pointeur à une structure 307
Pointeurs de fonction 309
Voir également 310
Initialisation des pointeurs 310
Mise en garde: 311
Mise en garde: 311
Adresse de l'opérateur (&) 311
Arithmétique du pointeur 311
void * pointeurs comme arguments et renvoyer des valeurs aux fonctions standard 312
Constants 312
Pointeurs uniques 312
Pointeur vers le pointeur 313
Même astérisque, significations différentes 315
Prémisse 315
Exemple 315
Conclusion 316
Pointeur vers le pointeur 316

introduction 317

Pointeurs et tableaux 319

Comportement polymorphe avec des pointeurs de vide 319

Chapitre 45: Pointeurs de fonction 321

Introduction 321

Syntaxe 321

Examples 321

Assigner un pointeur de fonction 321

Renvoi de pointeurs de fonction à partir d'une fonction 322

Les meilleures pratiques 322

Utiliser typedef 322

Exemple: 323

Prendre des pointeurs de contexte. 324

Exemple 324

Voir également 324

introduction 325

Usage 325

Syntaxe 325

Mnémonique pour écrire des pointeurs de fonction 326

Les bases 326

Chapitre 46: Points de séquence 328

Remarques 328

Examples 328

Expressions séquencées 329

Expressions sans séquence 329

Expressions indéterminées 330

Chapitre 47: Portée de l'identifiant 332

Examples 332

Portée du bloc 332

Fonction Prototype Scope 332

Portée du fichier 333

Portée de la fonction ... 334

Chapitre 48: Préprocesseur et Macros ... 336

Introduction ... 336

Remarques ... 336

Examples ... 336

Inclusion conditionnelle et modification de signature de fonction conditionnelle ... 336

Inclusion du fichier source ... 339

Remplacement Macro ... 339

Directive d'erreur ... 340

#if 0 pour bloquer les sections de code ... 341

Coller de jeton ... 342

Macros prédéfinies ... 342

Macros prédéfinies obligatoires ... 342

Autres macros prédéfinies (non obligatoire) ... 343

Header Inclure les gardes ... 344

Mise en œuvre de la prévention ... 347

__cplusplus pour utiliser des C externes dans du code C ++ compilé avec C ++ ... 350

Macros de type fonction ... 351

La macro des arguments Variadic ... 352

Chapitre 49: Qualificatifs de type ... 355

Remarques ... 355

Qualifications de haut niveau ... 355

Qualifications du sous-type de pointeur ... 355

Examples ... 356

Variables non modifiables (const) ... 356

Attention ... 356

Variables volatiles ... 357

Chapitre 50: Relevés de sélection ... 359

Examples ... 359

if () Déclarations ... 359

if () ... else instructions et syntaxe ... 359

switch instructions () ... 360

if () ... else Ladder Chaînage deux ou plus if () ... else instructions 362
Imbriqué if () ... else VS if () .. sinon Ladder 363
Chapitre 51: Sélection générique 365
Syntaxe 365
Paramètres 365
Remarques 365
Examples 365
Vérifier si une variable est d'un certain type qualifié 365
Macro d'impression de type générique 366
Sélection générique basée sur plusieurs arguments 366
Chapitre 52: Séquence de caractères multi-caractères 369
Remarques 369
Examples 369
Trigraphes 369
Digraphes 370
Chapitre 53: Structs 372
Introduction 372
Examples 372
Structures de données simples 372
Typedef Structs 372
Pointeurs vers les structures 374
Membres de tableau flexibles 376
Déclaration de type 376
Effets sur la taille et le rembourrage 377
Usage 377
Le 'struct hack' 378
Compatibilité 378
Passer des structures à des fonctions 379
Programmation par objets utilisant des structures 380
Chapitre 54: Structure rembourrage et emballage 383
Introduction 383
Remarques 383

Examples 383
Structures d'emballage 383
Emballage de structure 384
Rembourrage de la structure 384
Chapitre 55: Tableaux 386
Introduction 386
Syntaxe 386
Remarques 386
Examples 387
Déclaration et initialisation d'un tableau 387
Effacer le contenu du tableau (mise à zéro) 388
Longueur du tableau 389
Définition de valeurs dans les tableaux 390
Définir un tableau et un élément de tableau d'accès 391
Allouer et initialiser à zéro un tableau avec une taille définie par l'utilisateur 392
Itérer à travers un tableau efficacement et ordre de rangée 392
Tableaux multidimensionnels 394
Itérer à travers un tableau en utilisant des pointeurs 397
Passer des tableaux multidimensionnels à une fonction 397
Voir également 398
Chapitre 56: Test des frameworks 399
Introduction 399
Remarques 399
Examples 399
CppUTest 399
Cadre de test d'unité 400
CMocka 401
Chapitre 57: Threads (natifs) 403
Syntaxe 403
Remarques 403
Les bibliothèques C connues pour prendre en charge les threads C11 sont les suivantes: 403
C bibliothèques qui ne supportent pas les threads C11, pourtant: 403

Examples.....404
Commencer plusieurs discussions.....404
Initialisation par un thread.....404
Chapitre 58: Traitement du signal.....406
Syntaxe.....406
Paramètres.....406
Remarques.....406
Examples.....407
Traitement du signal avec “signal ()”.....407
Chapitre 59: Type d'aliasing et efficace.....410
Remarques.....410
Examples.....411
Les types de caractères ne sont pas accessibles via des types autres que des caractères......411
Type efficace.....412
Violer les règles strictes d'aliasing.....413
restreindre la qualification.....413
Changer d'octets.....414
Chapitre 60: Typedef.....416
Introduction.....416
Syntaxe.....416
Remarques.....416
Examples.....417
Typedef pour les structures et les syndicats.....417
Utilisations simples de Typedef.....418
Pour donner des noms courts à un type de données.....418
Améliorer la portabilité.....418
Pour spécifier un usage ou améliorer la lisibilité.....419
Typedef pour les pointeurs de fonction.....419
Chapitre 61: Types de données.....422
Remarques.....422
Examples.....422
Types entiers et constantes.....422

Littéraux de chaîne....424

Types entiers de largeur fixe (depuis C99)....425

Constantes à virgule flottante....425

Interprétation des déclarations....426

Exemples....427

Plusieurs déclarations....427

Interprétation alternative....428

Chapitre 62: Valgrind....429

Syntaxe....429

Remarques....429

Examples....429

Courir Valgrind....429

Ajouter des drapeaux....429

Octets perdus - Oublier de libérer....429

Erreurs les plus fréquentes rencontrées lors de l'utilisation de Valgrind....430

Chapitre 63: X-macros....432

Introduction....432

Remarques....432

Examples....432

Utilisation triviale des macros X pour printfs....432

Valeur Enum et Identifiant....433

Extension: Donne la macro X comme argument....433

Génération de code....434

Ici, nous utilisons des macros X pour déclarer un enum contenant 4 commandes et une carte....434

De même, nous pouvons générer une table de saut pour appeler les fonctions par la valeur e....435

Crédits....436

À propos

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

It is an unofficial and free C Language 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 Language.

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 le langage C

Remarques

C est un langage de programmation informatique impératif à usage général, prenant en charge la programmation structurée, la portée de la variable lexicale et la récursivité, tandis qu'un système de type statique empêche de nombreuses opérations imprévues. De par sa conception, C fournit des modèles qui correspondent efficacement aux instructions de la machine, et trouve donc une utilisation durable dans les applications précédemment codées en langage assembleur, y compris les systèmes d'exploitation, et divers logiciels d'application allant des superordinateurs aux systèmes embarqués. .

Malgré ses capacités de bas niveau, le langage a été conçu pour encourager la programmation multi-plateforme. Un programme C conforme aux normes et portable peut être compilé pour une très grande variété de plates-formes informatiques et de systèmes d'exploitation avec peu de modifications apportées à son code source. Le langage est devenu disponible sur un très large éventail de plates-formes, des microcontrôleurs intégrés aux superordinateurs.

C a été développé à l'origine par Dennis Ritchie entre 1969 et 1973 dans les laboratoires Bell et utilisé pour réimplémenter les systèmes d' exploitation Unix. Il est depuis devenu l'un des langages de programmation les plus utilisés de tous les temps, avec des compilateurs C de différents fournisseurs disponibles pour la majorité des architectures et des systèmes d'exploitation informatiques existants.

Compilateurs communs

Le processus de compilation d'un programme C diffère entre les compilateurs et les systèmes d'exploitation. La plupart des systèmes d'exploitation sont livrés sans compilateur, vous devrez donc en installer un. Certains choix de compilateurs courants sont les suivants:

- GCC, la collection de compilateurs GNU
- clang: un front-end familial en langage C pour LLVM
- MSVC, outils de génération Microsoft Visual C / C ++

Les documents suivants devraient vous donner un bon aperçu de la manière de commencer à utiliser quelques-uns des compilateurs les plus courants:

- Démarrer avec Microsoft Visual C
- Démarrer avec GCC

Compiler C version Support

Notez que les compilateurs ont différents niveaux de prise en charge du standard C, beaucoup d'entre eux ne supportant toujours pas complètement C99. Par exemple, à partir de 2015, MSVC prend en charge une grande partie de C99, mais il comporte encore des exceptions importantes

pour la prise en charge du langage lui-même (par exemple, le prétraitement semble non conforme) et pour la bibliothèque C (par exemple `<tgmath.h>`). documentent-ils nécessairement leurs "choix dépendants de la mise en œuvre"? Wikipedia a un tableau montrant le support offert par certains compilateurs populaires.

Certains compilateurs (notamment GCC) ont proposé, ou continuent d'offrir, des *extensions de compilateur* qui implémentent des fonctionnalités supplémentaires que les producteurs de compilateurs jugent nécessaires, utiles ou susceptibles de faire partie d'une future version C, mais ne font actuellement partie d'aucun standard C. Comme ces extensions sont spécifiques au compilateur, elles peuvent être considérées comme non compatibles entre elles et les développeurs de compilateurs peuvent les supprimer ou les modifier dans des versions ultérieures du compilateur. L'utilisation de telles extensions peut généralement être contrôlée par les indicateurs du compilateur.

De plus, de nombreux développeurs ont des compilateurs qui ne prennent en charge que les versions spécifiques de C imposées par l'environnement ou la plate-forme qu'ils ciblent.

Si vous sélectionnez un compilateur, il est recommandé de choisir un compilateur prenant en charge la dernière version de C autorisée pour l'environnement cible.

Style de code (hors sujet ici):

Parce que l' espace blanc est insignifiante en C (qui est, il ne modifie pas le fonctionnement du code), les programmeurs utilisent souvent l' espace blanc pour rendre le code plus facile à lire et à comprendre, ce qu'on appelle le *style de code.* C'est un ensemble de règles et de directives utilisées lors de l'écriture du code source. Il couvre des problèmes tels que la manière dont les lignes doivent être mises en retrait, si des espaces ou des tabulations doivent être utilisés, comment les accolades doivent être placées, comment utiliser des espaces autour des opérateurs et des parenthèses, comment nommer les variables, etc.

Le style de code n'est pas couvert par la norme et est principalement basé sur l'opinion (différentes personnes trouvent différents styles plus faciles à lire), en tant que tel, il est généralement considéré comme hors sujet sur SO. Le conseil primordial sur le style dans son propre code est que la cohérence est primordiale - choisissez ou créez un style et respectez-le. Il suffit d'expliquer qu'il existe divers styles nommés d'usage courant qui sont souvent choisis par les programmeurs plutôt que de créer leur propre style.

Certains styles de retrait courants sont: le style K & R, le style Allman, le style GNU, etc. Certains de ces styles ont des variantes différentes. Allman, par exemple, est utilisé comme Allman ordinaire ou comme variante populaire, Allman-8. Des informations sur certains styles populaires sont disponibles sur Wikipedia . Ces noms de style proviennent des normes que les auteurs ou les organisations publient souvent pour les nombreuses personnes contribuant à leur code, afin que chacun puisse facilement lire le code lorsqu'il connaît le style, comme le guide de formatage GNU le document sur les normes de codage GNU .

Certaines conventions de nommage courantes sont: UpperCamelCase, lowerCamelCase, lower_case_with_underscore, ALL_CAPS, etc. Ces styles sont combinés de différentes manières

pour être utilisés avec différents objets et types (par exemple, les macros utilisent souvent le style ALL_CAPS)

Le style K & R est généralement recommandé pour une utilisation dans la documentation SO, tandis que les styles plus ésotériques, tels que Pico, sont déconseillés.

Bibliothèques et API non couvertes par la norme C (et donc hors sujet ici):

- API POSIX (couvrant par exemple PThreads , Sockets , Signals)

Versions

Version	la norme	Date de publication
K & R	n / a	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO / IEC 9899: 1990	1990-12-20
C95	ISO / IEC 9899 / AMD1: 1995	1995-03-30
C99	ISO / IEC 9899: 1999	1999-12-16
C11	ISO / IEC 9899: 2011	2011-12-15

Examples

Bonjour le monde

Pour créer un programme C simple qui affiche *"Hello, World"* à l'écran, utilisez un éditeur de texte pour créer un nouveau fichier (par exemple, `hello.c` - l'extension du fichier doit être `.c`) contenant le code source suivant:

Bonjour c

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

Démo en direct sur Coliru

Regardons ce programme simple ligne par ligne

```
#include <stdio.h>
```

Cette ligne indique au compilateur d'inclure le contenu du fichier d'en-tête de bibliothèque standard `stdio.h` dans le programme. Les en-têtes sont généralement des fichiers contenant des déclarations de fonction, des macros et des types de données, et vous devez inclure le fichier d'en-tête avant de les utiliser. Cette ligne inclut `stdio.h` pour pouvoir appeler la fonction `puts()` .

Voir plus sur les en-têtes.

```
int main(void)
```

Cette ligne commence la définition d'une fonction. Il indique le nom de la fonction (`main`), le type et le nombre d'arguments attendus (`void` , c'est-à-dire none) et le type de valeur renvoyé par cette fonction (`int`). L'exécution du programme commence dans la fonction `main()` .

```
{
    …
}
```

Les accolades sont utilisées par paires pour indiquer où commence et se termine un bloc de code. Ils peuvent être utilisés de nombreuses manières, mais dans ce cas ils indiquent où la fonction commence et se termine.

```
    puts("Hello, World");
```

Cette ligne appelle la fonction `puts()` pour sortir du texte sur la sortie standard (l'écran, par défaut), suivi d'une nouvelle ligne. La chaîne à afficher est comprise dans les parenthèses.

`"Hello, World"` est la chaîne qui sera écrite à l'écran. En C, chaque valeur littérale de chaîne doit figurer entre les guillemets `"…"` .

Voir plus sur les chaînes.

Dans les programmes C, chaque instruction doit être terminée par un point-virgule (ie `;`).

```
    return 0;
```

Lorsque nous avons défini `main()` , nous l'avons déclaré comme une fonction renvoyant un `int` , ce qui signifie qu'il doit retourner un entier. Dans cet exemple, nous retournons la valeur entière 0, qui est utilisée pour indiquer que le programme s'est terminé avec succès. Après le `return 0;`

déclaration, le processus d'exécution se terminera.

Modifier le programme

Les éditeurs de texte simples incluent `vim` ou `gedit` sous Linux ou `Notepad` sous Windows. Les éditeurs multi-plateformes incluent également `Visual Studio Code` ou le `Sublime Text` .

L'éditeur doit créer des fichiers texte, pas RTF ou autre format.

Compiler et exécuter le programme

Pour exécuter le programme, ce fichier source (`hello.c`) doit d'abord être compilé dans un fichier exécutable (par exemple, `hello` sur le système Unix / Linux ou `hello.exe` sur Windows). Ceci est fait en utilisant un compilateur pour le langage C.

Voir plus sur la compilation

Compiler en utilisant GCC

GCC (GNU Compiler Collection) est un compilateur C largement utilisé. Pour l'utiliser, ouvrez un terminal, utilisez la ligne de commande pour accéder à l'emplacement du fichier source, puis exécutez:

```
gcc hello.c -o hello
```

Si aucune erreur n'est trouvée dans le code source (`hello.c`), le compilateur créera un **fichier binaire** , dont le nom est donné par l'argument de l'option de ligne de commande `-o` (`hello`). C'est le fichier exécutable final.

Nous pouvons également utiliser les options d'avertissement `-Wall -Wextra -Werror` , qui aident à identifier les problèmes pouvant entraîner l'échec du programme ou produire des résultats inattendus. Ils ne sont pas nécessaires pour ce programme simple mais c'est une manière de les ajouter:

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

Utiliser le compilateur clang

Pour compiler le programme en utilisant `clang` vous pouvez utiliser:

```
clang -Wall -Wextra -Werror -o hello hello.c
```

De par leur conception, les options de ligne de commande `clang` sont similaires à celles de GCC.

Utilisation du compilateur Microsoft C à partir de la ligne de commande

Si vous utilisez le compilateur Microsoft `cl.exe` sur un système Windows qui prend en charge Visual Studio et si toutes les variables d'environnement sont définies, cet exemple C peut être compilé à l'aide de la commande suivante qui produira un fichier exécutable `hello.exe` dans le répertoire (Il existe des options d'avertissement telles que `/W3` pour `cl` , à peu près analogues à `-Wall` etc. pour GCC ou clang).

```
cl hello.c
```

Exécuter le programme

Une fois compilé, le fichier binaire peut alors être exécuté en tapant `./hello` dans le terminal. Lors de l'exécution, le programme compilé affichera `Hello, World` , suivi d'une nouvelle ligne, à l'invite de commande.

Original "Bonjour, Monde!" dans K & R C

Ce qui suit est l'original "Hello, World!" programme du livre The C Programming Language de Brian Kernighan et Dennis Ritchie (Ritchie était le développeur original du langage de programmation C chez Bell Labs), dénommé "K & R":

K & R

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Notez que le langage de programmation C n'était pas standardisé au moment de la rédaction de la première édition de ce livre (1978), et que ce programme ne sera probablement pas compilé sur la plupart des compilateurs modernes à moins qu'ils soient invités à accepter le code C90.

Ce tout premier exemple du livre de K & R est maintenant considéré comme de mauvaise qualité, en partie parce qu’il manque un type de retour explicite pour `main()` et en partie parce qu’il ne contient pas d’instruction de `return` . La deuxième édition du livre a été écrite pour l'ancienne norme C89. En C89, le type de `main` par défaut est `int` , mais l'exemple de K & R ne renvoie pas de valeur définie à l'environnement. Dans les normes C99 et ultérieures, le type de retour est obligatoire, mais il est prudent de ne pas inclure l'instruction `return` de `main` (et uniquement de `main`), en raison d'un cas particulier introduit avec C99 5.1.2.2.3 - cela équivaut à renvoyer 0 , ce qui indique le succès.

La forme recommandée et la plus portable de `main` pour les systèmes hébergés est `int main (void)` lorsque le programme n'utilise aucun argument de ligne de commande, ou `int main(int`

`argc, char **argv)` lorsque le programme utilise les arguments de ligne de commande.

C90 §5.1.2.2.3 Terminaison du **programme**

> Un retour de l'appel initial à la fonction `main` revient à appeler la fonction `exit` avec la valeur renvoyée par la fonction `main` comme argument. Si la fonction `main` exécute un retour qui ne spécifie aucune valeur, l'état de la terminaison renvoyé à l'environnement hôte n'est pas défini.

C90 §6.6.6.4 **La déclaration de `return`**

> Si une instruction de `return` sans expression est exécutée et que la valeur de l'appel de fonction est utilisée par l'appelant, le comportement est indéfini. Atteindre le `}` qui termine une fonction équivaut à exécuter une instruction de `return` sans expression.

C99 §5.1.2.2.3 Résiliation du **programme**

> Si le type de retour de la fonction `main` est un type compatible avec `int` , un retour de l'appel initial à la fonction `main` revient à appeler la fonction `exit` avec la valeur renvoyée par la fonction `main` comme argument; Si vous atteignez le `}` qui termine la fonction `main` , la valeur 0 est renvoyée. Si le type de retour n'est pas compatible avec `int` , l'état de la terminaison renvoyé à l'environnement hôte n'est pas spécifié.

Lire Démarrer avec le langage C en ligne: https://riptutorial.com/fr/c/topic/213/demarrer-avec-le-langage-c

Chapitre 2: - classification et conversion des personnages

Examples

Classification des caractères lus dans un flux

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(FILE *f) {
  chartypes types = { 0, 0, 0 };
  int ch;

  while ((ch = fgetc(f)) != EOF) {
    types.space += !!isspace(ch);
    types.alnum += !!isalnum(ch);
    types.punct += !!ispunct(ch);
  }

  return types;
}
```

La fonction de `classify` lit les caractères d'un flux et compte le nombre d'espaces, les caractères alphanumériques et les signes de ponctuation. Cela évite plusieurs écueils.

- Lors de la lecture d'un caractère dans un flux, le résultat est enregistré sous la forme d'un `int` , car sinon il y aurait une ambiguïté entre la lecture de `EOF` (marqueur de fin de fichier) et un caractère ayant le même modèle de bit.
- Les fonctions de classification des caractères (par exemple, `isspace`) s'attendent à ce que leur argument *soit représentable en tant que caractère `unsigned char` ou en tant que valeur de la macro `EOF`* . Puisque c'est exactement ce que retourne `fgetc` , il n'y a pas besoin de conversion ici.
- La valeur de retour des fonctions de classification de caractères ne fait que distinguer zéro (signifiant `false`) et non nul (ce qui signifie `true`). Pour compter le nombre d'occurrences, cette valeur doit être convertie en 1 ou 0, ce qui se fait par la double négation, `!!` .

Classification des caractères d'une chaîne

```
#include <ctype.h>
#include <stddef.h>

typedef struct {
```

```
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(const char *s) {
  chartypes types = { 0, 0, 0 };
  const char *p;
  for (p= s; p != '\0'; p++) {
    types.space += !!isspace((unsigned char)*p);
    types.alnum += !!isalnum((unsigned char)*p);
    types.punct += !!ispunct((unsigned char)*p);
  }

  return types;
}
```

La fonction de `classify` examine tous les caractères d'une chaîne et compte le nombre d'espaces, les caractères alphanumériques et les signes de ponctuation. Cela évite plusieurs écueils.

- Les fonctions de classification des caractères (par exemple, `isspace`) s'attendent à ce que leur argument *soit représentable en tant que caractère `unsigned char` ou en tant que valeur de la macro `EOF`* .
- L'expression `*p` est de type `char` et doit donc être convertie pour correspondre à la formulation ci-dessus.
- Le `char` type est défini comme équivalent soit `signed char` ou `unsigned char` .
- Lorsque `char` est équivalent à `unsigned char` , il n'y a pas de problème, puisque toutes les valeurs possibles du `char` type est représentable comme `unsigned char` .
- Lorsque `char` est équivalent au caractère `signed char` , il doit être converti en caractère `unsigned char` avant d'être transmis aux fonctions de classification des caractères. Et bien que la valeur du caractère puisse changer à cause de cette conversion, c'est exactement ce à quoi ces fonctions s'attendent.
- La valeur de retour des fonctions de classification de caractères ne fait que distinguer zéro (signifiant `false`) et non nul (ce qui signifie `true`). Pour compter le nombre d'occurrences, cette valeur doit être convertie en 1 ou 0, ce qui se fait par la double négation, `!!` .

introduction

L'en-tête `ctype.h` fait partie de la bibliothèque standard C. Il fournit des fonctions pour classer et convertir les caractères.

Toutes ces fonctions prennent un paramètre, un `int` qui doit être soit *EOF,* soit représentable en tant que caractère non signé.

Les noms des fonctions de classement sont préfixés par 'is'. Chacun renvoie une valeur non nulle entière (TRUE) si le caractère qui lui est transmis satisfait à la condition associée. Si la condition n'est pas satisfaite, la fonction retourne une valeur zéro (FALSE).

Ces fonctions de classification fonctionnent comme indiqué, en supposant que les paramètres régionaux C par défaut:

```
int a;
int c = 'A';
a = isalpha(c); /* Checks if c is alphabetic (A-Z, a-z), returns non-zero here. */
a = isalnum(c); /* Checks if c  is alphanumeric (A-Z, a-z, 0-9), returns non-zero here. */
a = iscntrl(c); /* Checks is c is a control character (0x00-0x1F, 0x7F), returns zero here. */
a = isdigit(c); /* Checks if c is a digit (0-9), returns zero here. */
a = isgraph(c); /* Checks if c has a graphical representation (any printing character except
space), returns non-zero here. */
a = islower(c); /* Checks if c is a lower-case letter (a-z), returns zero here. */
a = isprint(c); /* Checks if c is any printable character (including space), returns non-zero
here. */
a = isupper(c); /* Checks if c is a upper-case letter (a-z), returns zero here. */
a = ispunct(c); /* Checks if c is a punctuation character, returns zero here. */
a = isspace(c); /* Checks if c is a white-space character, returns zero here. */
a = isupper(c); /* Checks if c is an upper-case letter (A-Z), returns non-zero here. */
a = isxdigit(c); /* Checks if c is a hexadecimal digit (A-F, a-f, 0-9), returns non-zero here.
*/
```

C99

```
a = isblank(c); /* Checks if c is a blank character (space or tab), returns non-zero here. */
```

Il y a deux fonctions de conversion. Ceux-ci sont nommés en utilisant le préfixe «to». Ces fonctions prennent le même argument que celles ci-dessus. Cependant, la valeur de retour n'est pas un simple zéro ou non nul, mais l'argument passé a changé d'une certaine manière.

Ces fonctions de conversion fonctionnent comme indiqué, en supposant la locale C par défaut:

```
int a;
int c = 'A';

/* Converts c to a lower-case letter (a-z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'a' here.
 */
a = tolower(c);

/* Converts c to an upper-case letter (A-Z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'A' here.
 */
a = toupper(c);
```

Les informations ci-dessous sont extraites de la mise en correspondance de cplusplus.com sur la manière dont l'ensemble ASCII à 127 caractères d'origine est pris en compte par chacune des fonctions de type classification (a • indique que la fonction renvoie une valeur non nulle pour ce caractère)

Valeurs ASCII	personnages	iscntrl	isblank	isspace	isupper	est plus bas	isalpha	isc
0x00 .. 0x08	NUL, (autres codes de contrôle)	•						

Valeurs ASCII	personnages	iscntrl	isblank	isspace	isupper	est plus bas	isalpha	isc
0x09	onglet ('\ t')	•	•	•				
0x0A .. 0x0D	(codes de contrôle d'espaces blancs: '\ f', '\ v', '\ n', '\ r')	•		•				
0x0E .. 0x1F	(autres codes de contrôle)	•						
0x20	espace (' ')		•	•				
0x21 .. 0x2F	! "# $% & '() * +, -. /							
0x30 .. 0x39	0123456789							•
0x3a .. 0x40	:; <=>? @							
0x41 .. 0x46	A B C D E F				•		•	
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				•		•	
0x5B .. 0x60	[] ^ _ `							
0x61 .. 0x66	a B c d e F					•	•	
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•	•	
0x7B .. 0x7E	{} ~ bar							
0x7F	(DEL)	•						

Lire - classification et conversion des personnages en ligne: https://riptutorial.com/fr/c/topic/6846/-ctype-h----classification-et-conversion-des-personnages

Chapitre 3: Affirmation

Introduction

Une ***assertion*** est un prédicat que la condition présentée doit être vraie au moment où l'assertion est rencontrée par le logiciel. Les plus ***simples*** sont ***les assertions simples*** , validées au moment de l'exécution. Cependant, ***les assertions statiques*** sont vérifiées au moment de la compilation.

Syntaxe

- affirmer (expression)
- static_assert (expression, message)
- _Static_assert (expression, message)

Paramètres

Paramètre	Détails
expression	expression de type scalaire.
message	chaîne littérale à inclure dans le message de diagnostic.

Remarques

Les deux `assert` et `static_assert` sont des macros définies dans `assert.h` .

La définition de `assert` dépend de la macro `NDEBUG` qui n'est pas définie par la bibliothèque standard. Si `NDEBUG` est défini, `assert` est un no-op:

```
#ifdef NDEBUG
#  define assert(condition) ((void) 0)
#else
#  define assert(condition) /* implementation defined */
#endif
```

L'opinion varie selon que `NDEBUG` doit toujours être utilisé pour les compilations de production.

- Le pro-camp soutient que `assert` appels `abort` et les messages d'affirmation ne sont pas utiles pour les utilisateurs finaux, de sorte que le résultat n'est pas utile à l' utilisateur. Si vous avez des conditions fatales à vérifier dans le code de production, vous devez utiliser les `if/else` ordinaires et `exit` ou `quick_exit` pour terminer le programme. Contrairement à l' `abort` , elles permettent au programme de faire du nettoyage (via des fonctions enregistrées avec `atexit` ou `at_quick_exit`).

- Le con camp fait valoir que `assert` appels ne doivent jamais tirer dans le code de production, mais si elles le font, la condition est cochée, il y a quelque chose de mal et de façon spectaculaire le programme se conduisent mal pire si l' exécution se poursuit. Par conséquent, il est préférable d’avoir des assertions actives dans le code de la production car si elles se déclenchent, l’enfer est déjà déchaîné.
- Une autre option consiste à utiliser un système d'assertions maison qui exécute toujours la vérification mais gère les erreurs différemment entre le développement (lorsque l' `abort` est approprié) et la production (où une «erreur interne inattendue - veuillez contacter le support technique» peut être plus appropriée).

`static_assert` développe en `_Static_assert` qui est un mot clé. La condition est vérifiée au moment de la compilation, la `condition` doit donc être une expression constante. Il n'est pas nécessaire que cela soit traité différemment entre le développement et la production.

Examples

Condition préalable et postcondition

Un cas d'utilisation pour l'assertion est la condition préalable et la post-condition. Cela peut être très utile pour maintenir invariant et concevoir par contrat . Pour un exemple, une longueur est toujours nulle ou positive, donc cette fonction doit renvoyer une valeur nulle ou positive.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
```

```
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```

Assertion simple

Une assertion est une déclaration utilisée pour affirmer qu'un fait doit être vrai lorsque cette ligne de code est atteinte. Les assertions sont utiles pour s'assurer que les conditions attendues sont remplies. Lorsque la condition passée à une assertion est vraie, il n'y a pas d'action. Le comportement sur les conditions fausses dépend des indicateurs du compilateur. Lorsque des assertions sont activées, une fausse entrée provoque un arrêt immédiat du programme. Lorsqu'elles sont désactivées, aucune action n'est entreprise. Il est courant d'activer les assertions dans les versions internes et de débogage, et de les désactiver dans les versions de publication, bien que les assertions soient souvent activées dans la version. (Le fait que la terminaison soit meilleure ou pire que les erreurs dépend du programme.) Les assertions ne doivent être utilisées que pour intercepter des erreurs de programmation internes, ce qui signifie généralement que des paramètres incorrects ont été transmis.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);

    printf("x = %d\n", x);
    return 0;
}
```

Sortie possible avec `NDEBUG` non défini:

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Sortie possible avec `NDEBUG` défini:

```
x = -1
```

Il est `NDEBUG` définir `NDEBUG` globalement afin de pouvoir facilement compiler votre code avec toutes les assertions `NDEBUG` ou désactivées. Un moyen simple de le faire est de définir `NDEBUG` comme une option pour le compilateur ou de le définir dans un en-tête de configuration partagé (par exemple, `config.h`).

Affirmation statique

C11

Les assertions statiques sont utilisées pour vérifier si une condition est vraie lorsque le code est compilé. Si ce n'est pas le cas, le compilateur doit émettre un message d'erreur et arrêter le processus de compilation.

Une assertion statique est une assertion qui est vérifiée au moment de la compilation, pas au moment de l'exécution. La condition doit être une expression constante et, si false, une erreur de compilation. Le premier argument, la condition qui est vérifiée, doit être une expression constante et le second un littéral de chaîne.

Contrairement à assert, `_Static_assert` est un mot-clé. Une macro de commodité `static_assert` est définie dans `<assert.h>` .

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

C99

Avant C11, il n'y avait pas de support direct pour les assertions statiques. Cependant, en C99, les assertions statiques pouvaient être émulées avec des macros qui déclencheraient un échec de compilation si la condition de compilation était fausse. À la différence de `_Static_assert` , le second paramètre doit être un nom de jeton approprié afin de pouvoir créer un nom de variable avec lui. Si l'assertion échoue, le nom de la variable apparaît dans l'erreur du compilateur, car cette variable a été utilisée dans une déclaration de tableau syntaxiquement incorrecte.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##__##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

Avant C99, vous ne pouviez pas déclarer des variables à des emplacements arbitraires dans un bloc, vous devriez donc être extrêmement prudent lorsque vous utilisez cette macro, en vous assurant qu'elle apparaît uniquement lorsqu'une déclaration de variable serait valide.

Affirmation de code inaccessible

Au cours du développement, lorsque certains chemins de code doivent être empêchés d'atteindre le flux de contrôle, vous pouvez utiliser `assert(0)` pour indiquer qu'une telle condition est erronée:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;
```

```
    default:
        assert(0);
}
```

Chaque fois que l'argument de la macro `assert()` évalué faux, la macro écrit des informations de diagnostic dans le flux d'erreur standard, puis abandonne le programme. Ces informations incluent le fichier et le numéro de ligne de l'instruction `assert()` et peuvent être très utiles lors du débogage. Les assertions peuvent être désactivées en définissant la macro `NDEBUG` .

Une autre façon de terminer un programme en cas d'erreur est d'utiliser les fonctions de bibliothèque standard `exit` , `quick_exit` ou `abort` . `exit` et `quick_exit` prenez un argument qui peut être renvoyé à votre environnement. `abort()` (et donc `assert`) peut être une terminaison très grave de votre programme, et certains nettoyages qui seraient normalement effectués à la fin de l'exécution peuvent ne pas être exécutés.

Le principal avantage d’ `assert()` est qu’il imprime automatiquement les informations de débogage. L'appel à `abort()` présente l'avantage de ne pas pouvoir être désactivé comme une assertion, mais cela risque de ne pas entraîner l'affichage d'informations de débogage. Dans certaines situations, l'utilisation des deux concepts peut être bénéfique:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
   ...
} else if (color == COLOR_BLUE) {
   ...
} else {
   assert(0), abort();
}
```

Lorsque les assertions sont *activées* , l'appel `assert()` imprimera les informations de débogage et mettra fin au programme. L'exécution n'atteint jamais l'appel `abort()` . Lorsque les assertions sont *désactivées* , l'appel `assert()` ne fait rien et `abort()` est appelé. Cela garantit que le programme se termine *toujours* pour cette condition d'erreur; L'activation et la désactivation n'affectent que les effets de l'impression ou non de la sortie de débogage.

Vous ne devriez jamais laisser un tel `assert` dans le code de production, car les informations de débogage ne sont pas utiles pour les utilisateurs finaux et parce que `abort` est généralement une terminaison beaucoup trop sévère qui empêchent les gestionnaires de nettoyage qui sont installés pour la `exit` ou `quick_exit` à courir.

Assert Messages d'erreur

Un truc existe qui peut afficher un message d'erreur avec une assertion. Normalement, vous écrivez un code comme celui-ci

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

Si l'assertion a échoué, un message d'erreur ressemblerait à

> L'assertion a échoué: p! = NULL, fichier main.c, ligne 5

Cependant, vous pouvez également utiliser un AND logique (`&&`) pour donner un message d'erreur.

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Maintenant, si l'assertion échoue, un message d'erreur indiquera quelque chose comme ceci

> Echec de l'assertion: p! = NULL && "fonction f: p ne peut pas être NULL", fichier main.c, ligne 5

La raison pour laquelle cela fonctionne est qu'une chaîne littérale est toujours évaluée à zéro (true). L'ajout de `&& 1` à une expression booléenne n'a aucun effet. Ainsi, l'ajout de `&& "error message"` n'a aucun effet, sauf que le compilateur affichera l'intégralité de l'expression ayant échoué.

Lire Affirmation en ligne: https://riptutorial.com/fr/c/topic/555/affirmation

Chapitre 4: Arguments de ligne de commande

Syntaxe

- int main (int argc, char * argv [])

Paramètres

Paramètre	Détails
argc	nombre d'arguments - initialisé au nombre d'arguments séparés par des espaces donnés au programme à partir de la ligne de commande, ainsi qu'au nom du programme lui-même.
argv	vecteur d'argument - initialisé à un tableau de `char` -pointers (chaînes) contenant les arguments (et le nom du programme) donnés sur la ligne de commande.

Remarques

Le programme AC s'exécutant dans un environnement hébergé (le type normal, par opposition à un environnement autonome) doit avoir une fonction `main` . Il est traditionnellement défini comme suit:

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

Notez que `argv` peut également être et est très souvent défini comme étant un caractère `char **argv` ; le comportement est le même. En outre, les noms de paramètres peuvent être modifiés car ils ne sont que des variables locales dans la fonction, mais `argc` et `argv` sont conventionnels et vous devriez les utiliser.

Pour les fonctions `main` où le code n'utilise aucun argument, utilisez `int main(void)` .

Les deux paramètres sont initialisés au démarrage du programme:

- `argc` est initialisé au nombre d'arguments séparés par des espaces donnés au programme à partir de la ligne de commande, ainsi qu'au nom du programme lui-même.
- `argv` est un tableau de `char` -pointers (chaînes) contenant les arguments (et le nom du programme) donnés sur la ligne de commande.
- Certains systèmes étendent les arguments de ligne de commande "dans le shell", d'autres non. Sur Unix si l'utilisateur tape `myprogram *.txt` le programme recevra une liste de fichiers texte; sous Windows, il recevra la chaîne " `*.txt` ".

Remarque: Avant d'utiliser `argv` , vous devrez peut-être vérifier la valeur de `argc` . En théorie, `argc` pourrait être `0` , et si `argc` est zéro, alors il n'y a pas d'argument et `argv[0]` (équivalent à `argv[argc]`) est un pointeur nul. Ce serait un système inhabituel avec un environnement hébergé si vous rencontriez ce problème. De même, il est possible, bien que très inhabituel, qu'il n'y ait aucune information sur le nom du programme. Dans ce cas, `argv[0][0] == '\0'` - le nom du programme peut être vide.

Supposons que nous démarrions le programme comme ceci:

```
./some_program abba banana mamajam
```

Alors `argc` est égal à `4` , et les arguments de ligne de commande:

- `argv[0]` pointe sur `"./some_program"` (le nom du programme) si le nom du programme est disponible dans l'environnement hôte. Sinon une chaîne vide `""` .
- `argv[1]` désigne `"abba"` ,
- `argv[2]` désigne `"banana"` ,
- `argv[3]` désigne `"mamajam"` ,
- `argv[4]` contient la valeur `NULL` .

Voir aussi Que devrait `main()` retourner en C et C ++ pour les guillemets complets du standard.

Examples

Impression des arguments de la ligne de commande

Après avoir reçu les arguments, vous pouvez les imprimer comme suit:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
    }
}
```

Remarques

1. Le paramètre `argv` peut également être défini comme étant `char *argv[]` .
2. `argv[0]` *peut* contenir le nom du programme lui-même (en fonction de l'exécution du programme). Le premier argument "réel" de la ligne de commande est à `argv[1]` , et c'est la raison pour laquelle la variable de boucle `i` est initialisée à 1.
3. Dans l'instruction print, vous pouvez utiliser `*(argv + i)` au lieu de `argv[i]` - il évalue la même chose, mais est plus verbeux.
4. Les crochets autour de la valeur de l'argument aident à identifier le début et la fin. Cela peut être inestimable s'il y a des blancs, des nouvelles lignes, des retours chariot ou d'autres caractères inhabituels dans l'argument. Certaines variantes de ce programme sont un outil utile pour déboguer des scripts shell dans lesquels vous devez comprendre ce que contient

la liste d'arguments (bien qu'il existe des alternatives de shell simples, presque équivalentes).

Imprimer les arguments dans un programme et convertir en valeurs entières

Le code suivant imprimera les arguments au programme et le code tentera de convertir chaque argument en un nombre (à un `long`):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

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

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);
        }
        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno ==
ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);
        }
        else {
            printf("Argument %d is a number, and the value is: %ld\n",
                   i, argument_numValue);
        }
    }
    return 0;
}
```

LES RÉFÉRENCES:

- strtol () renvoie une valeur incorrecte
- Utilisation correcte de strtol

Utiliser les outils GNU getopt

Les options de ligne de commande pour les applications ne sont pas traitées différemment des arguments de ligne de commande par le langage C. Ce ne sont que des arguments qui, dans un environnement Linux ou Unix, commencent traditionnellement par un tiret (`-`).

Avec glibc dans un environnement Linux ou Unix, vous pouvez utiliser les outils getopt pour définir, valider et analyser facilement les options de ligne de commande du reste de vos arguments.

Ces outils s'attendent à ce que vos options soient formatées conformément aux normes de codage GNU , qui sont une extension de ce que POSIX spécifie pour le format des options de

ligne de commande.

L'exemple ci-dessous illustre la gestion des options de ligne de commande avec les outils GNU getopt.

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage (FILE *fp, const char *path)
{
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]\n", basename);
    fprintf (fp, "  -h, --help\t\t"
                 "Print this help and exit.\n");
    fprintf (fp, "  -f, --file[=FILENAME]\t"
                 "Write all output to a file (defaults to out.txt).\n");
    fprintf (fp, "  -m, --msg=STRING\t"
                 "Output a particular message rather than 'Hello world'.\n");
}

/* parse command-line options and print message */
int main(int argc, char *argv[])
{
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;

    /* table of all supported options in their long form.
     * fields: name, has_arg, flag, val
     * `has_arg` specifies whether the associated long-form option can (or, in
     * some cases, must) have an argument. the valid values for `has_arg` are
     * `no_argument`, `optional_argument`, and `required_argument`.
     * if `flag` points to a variable, then the variable will be given a value
     * of `val` when the associated long-form option is present at the command
     * line.
     * if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
     * when the associated long-form option is found amongst the command-line
     * arguments.
     */
    struct option longopts[] = {
        { "help", no_argument, &help_flag, 1 },
        { "file", optional_argument, NULL, 'f' },
        { "msg", required_argument, NULL, 'm' },
        { 0 }
    };

    /* infinite loop, to be broken when we are done parsing options */
    while (1) {
        /* getopt_long supports GNU-style full-word "long" options in addition
         * to the single-character "short" options which are supported by
         * getopt.
         * the third argument is a collection of supported short-form options.
```

```
         * these do not necessarily have to correlate to the long-form options.
         * one colon after an option indicates that it has an argument, two
         * indicates that the argument is optional. order is unimportant.
         */
        opt = getopt_long (argc, argv, "hf::m:", longopts, 0);

        if (opt == -1) {
            /* a return value of -1 indicates that there are no more options */
            break;
        }

        switch (opt) {
        case 'h':
            /* the help_flag and value are specified in the longopts table,
             * which means that when the --help option is specified (in its long
             * form), the help_flag variable will be automatically set.
             * however, the parser for short-form options does not support the
             * automatic setting of flags, so we still need this code to set the
             * help_flag manually when the -h option is specified.
             */
            help_flag = 1;
            break;
        case 'f':
            /* optarg is a global variable in getopt.h. it contains the argument
             * for this option. it is null if there was no argument.
             */
            printf ("outarg: '%s'\n", optarg);
            strncpy (filename, optarg ? optarg : "out.txt", sizeof (filename));
            /* strncpy does not fully guarantee null-termination */
            filename[sizeof (filename) - 1] = '\0';
            break;
        case 'm':
            /* since the argument for this option is required, getopt guarantees
             * that aptarg is non-null.
             */
            strncpy (message, optarg, sizeof (message));
            message[sizeof (message) - 1] = '\0';
            break;
        case '?':
            /* a return value of '?' indicates that an option was malformed.
             * this could mean that an unrecognized option was given, or that an
             * option which requires an argument did not include an argument.
             */
            usage (stderr, argv[0]);
            return 1;
        default:
            break;
        }
    }

    if (help_flag) {
        usage (stdout, argv[0]);
        return 0;
    }

    if (filename[0]) {
        fp = fopen (filename, "w");
    } else {
        fp = stdout;
    }
```

```
    if (!fp) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    fprintf (fp, "%s\n", message);

    fclose (fp);
    return 0;
}
```

Il peut être compilé avec `gcc` :

```
gcc example.c -o example
```

Il prend en charge trois options de ligne de commande (`--help` , `--file` et `--msg`). Tous ont aussi une "forme courte" (`-h` , `-f` et `-m`). Les options "fichier" et "msg" acceptent toutes deux des arguments. Si vous spécifiez l'option "msg", son argument est requis.

Les arguments pour les options sont formatés comme suit:

- `--option=value` (pour les options de formulaire long)
- `-ovalue` ou `-o"value"` (pour les options `-ovalue`)

Lire Arguments de ligne de commande en ligne: https://riptutorial.com/fr/c/topic/1285/arguments-de-ligne-de-commande

Chapitre 5: Arguments variables

Introduction

Les arguments variables sont utilisés par les fonctions de la famille printf (`printf` , `fprintf` , etc.) et d’autres pour permettre à une fonction d’être appelée avec un nombre différent d’arguments à chaque fois, d’où le nom *varargs* .

Pour implémenter des fonctions à l'aide de la fonctionnalité d'arguments variables, utilisez `#include <stdarg.h>` .

Pour appeler des fonctions qui prennent un nombre variable d'arguments, assurez-vous qu'il existe un prototype complet avec les points de suspension suivants dans la portée: `void err_exit(const char *format, ...);` par exemple.

Syntaxe

- void va_start (va_list ap, *dernier*); / * Démarrer le traitement des arguments variadiques; *last* est le dernier paramètre de la fonction avant les points de suspension ("...") * /
- *tapez* va_arg (va_list ap, *tapez*); / * Récupère le prochain argument variadic dans la liste; assurez-vous de passer le bon type *promu* * /
- void va_end (va_list ap); / * Fin du traitement des arguments * /
- void va_copy (va_list dst, va_list src); / * C99 ou plus récent: copie la liste des arguments, c.-à-d. La position actuelle dans le traitement des arguments, dans une autre liste (par exemple pour passer plusieurs fois les arguments) * /

Paramètres

Paramètre	Détails
`va_list` ap	pointeur d'argument, position actuelle dans la liste des arguments variadiques
dernier	nom du dernier argument de la fonction non variadique, le compilateur trouve donc le bon endroit pour commencer à traiter les arguments variadiques; ne peut pas être déclaré comme une variable de `register` , une fonction ou un type de tableau
type	type **promu** de l'argument variadic à lire (par exemple, `int` pour un argument `short int`)
`va_list` src	pointeur d'argument courant à copier
`va_list` dst	nouvelle liste d'arguments à remplir

Remarques

Les fonctions `va_start` , `va_arg` , `va_end` et `va_copy` sont en fait des macros.

Veillez à *toujours* appeler `va_start` premier, et une seule fois, et à appeler `va_end` dernier, et une seule fois, et à chaque point de sortie de la fonction. Ne pas le faire *peut* fonctionner sur *votre* système mais n'est sûrement **pas** portable et invite donc des bogues.

Prenez soin de déclarer votre fonction correctement, c'est-à-dire avec un prototype, et tenez compte des restrictions sur le *dernier* argument non-variadique (pas de `register` , pas de fonction ou de type tableau). Il n'est pas possible de déclarer une fonction qui ne prend que des arguments variadiques, car au moins un argument non-variadic est nécessaire pour pouvoir commencer le traitement des arguments.

Lorsque vous appelez `va_arg` , vous devez demander le type d'argument **promu** , à savoir:

- `short` est promu en `int` (et `unsigned short` est également promu en `int` sauf si `sizeof(unsigned short) == sizeof(int)` , auquel cas il est promu `unsigned int`).
- `float` est promu à `double` .
- `signed char` est promu dans `int` ; `unsigned char` est également promu dans `int` sauf si `sizeof(unsigned char) == sizeof(int)` , ce qui est rarement le cas.
- `char` est généralement promu en `int` .
- Les types C99 comme `uint8_t` ou `int16_t` sont également promus.

Le traitement des arguments variadiques historiques (K & R) est déclaré dans `<varargs.h>` mais ne doit pas être utilisé car il est obsolète. Le traitement standard des arguments variadiques (celui décrit ici et déclaré dans `<stdarg.h>`) a été introduit dans C89; la macro `va_copy` été introduite dans C99 mais fournie par de nombreux compilateurs avant cela.

Examples

Utilisation d'un argument de décompte explicite pour déterminer la longueur de la va_list

Avec toute fonction variadique, la fonction doit savoir interpréter la liste des arguments variables. Avec les fonctions `printf()` ou `scanf()` , la chaîne de format indique à la fonction à quoi s’attendre.

La technique la plus simple consiste à transmettre un nombre explicite des autres arguments (qui sont normalement tous du même type). Ceci est démontré dans la fonction variadic dans le code ci-dessous qui calcule la somme d'une série d'entiers, où il peut y avoir un nombre quelconque d'entiers mais ce compte est spécifié en tant qu'argument avant la liste des arguments variables.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
    int sum = 0;
```

```
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
      sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

Utiliser des valeurs de terminateur pour déterminer la fin de va_list

Avec toute fonction variadique, la fonction doit savoir interpréter la liste des arguments variables. L'approche «traditionnelle» (illustrée par `printf`) consiste à spécifier le nombre d'arguments en amont. Cependant, ce n'est pas toujours une bonne idée:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6)  /* Disaster */
```

Parfois, il est plus robuste d'ajouter un terminateur explicite, illustré par la fonction `execlp()` POSIX. Voici une autre fonction pour calculer la somme d'une série de nombres `double` :

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
  double sum = 0;
  va_list va;

  va_start(va, x);
  for (; !isnan(x); x = va_arg(va, double)) {
    sum += x;
  }
  va_end(va);

  return sum;
}

int main (void) {
  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Bonnes valeurs de terminateur:

- entier (censé être tout positif ou non négatif) - `0` ou `-1`
- types à virgule flottante - `NAN`
- types de pointeurs - `NULL`
- types d'énumérateur - une valeur spéciale

Implémenter des fonctions avec une interface semblable à `printf ()`

Une utilisation courante des listes d'arguments de longueur variable consiste à implémenter des fonctions qui constituent un wrapper mince autour de la famille de fonctions `printf()` . Un exemple est un ensemble de fonctions de rapport d’erreur.

errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif
```

Ceci est un exemple simple; de tels paquets peuvent être très complexes. Normalement, les programmeurs utiliseront soit `errmsg()` soit `warnmsg()` , qui utilisent eux-mêmes `verrmsg()` interne. Si quelqu'un a besoin d'en faire plus, alors la fonction `verrmsg()` exposée sera utile. Vous pouvez éviter d' exposer jusqu'à ce que vous avez besoin pour cela (YAGNI - vous n'êtes pas en avoir besoin), mais la nécessité lèveras à terme (vous en aurez besoin - YAGNI).

errmsg.c

Ce code n'a besoin que de transférer les arguments variadiques à la fonction `vfprintf()` pour générer une erreur standard. Il signale également le message d'erreur du système correspondant au numéro d'erreur du système (`errno`) transmis aux fonctions.

```
#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}
```

```
void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}
```

Utiliser `errmsg.h`

Vous pouvez maintenant utiliser ces fonctions comme suit:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer),
filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

Si les appels système `open()` ou `read()` échouent, l'erreur est écrite dans l'erreur standard et le programme se termine avec le code de sortie 1. Si l'appel système `close()` échoue, l'erreur est simplement imprimée en tant que message d'avertissement. le programme continue.

Vérification de l'utilisation correcte des formats `printf()`

Si vous utilisez GCC (le compilateur GNU C, qui fait partie de la collection de compilateurs GNU), ou utilisez Clang, vous pouvez demander au compilateur de vérifier que les arguments que vous transmettez aux fonctions de message d'erreur correspondent à ce qu'attend `printf()` . Comme tous les compilateurs ne prennent pas en charge l'extension, celle-ci doit être compilée de manière conditionnelle, ce qui est un peu délicat. Cependant, la protection qu'elle procure en vaut la peine.

Tout d'abord, nous devons savoir comment détecter que le compilateur est GCC ou Clang émulant GCC. La réponse est que GCC définit `__GNUC__` pour indiquer cela.

Voir les attributs de fonction communs pour plus d'informations sur les attributs - en particulier l'attribut `format` .

errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format(printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
        PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
        PRINTFLIKE(2, 3);

#endif
```

Maintenant, si vous faites une erreur comme:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(où `%d` devrait être `%s`), alors le compilateur se plaindra:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function ‘main’:
erruse.c:20:64: error: format ‘%d’ expects argument of type ‘int’, but argument 4 has type
‘const char *’ [-Werror=format=]
         errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                                           ~^
                                                           %s
cc1: all warnings being treated as errors
$
```

Utiliser une chaîne de format

L'utilisation d'une chaîne de format fournit des informations sur le nombre et le type attendus des arguments variadic ultérieurs de manière à éviter la nécessité d'un argument de décompte explicite ou d'une valeur de terminaison.

L'exemple ci-dessous montre une fonction qui encapsule la fonction standard `printf()` , ne permettant l'utilisation que d'arguments variadiques du type `char` , `int` et `double` (en format décimal à virgule flottante). Ici, comme avec `printf()` , le premier argument de la fonction wrapping est la chaîne de format. A mesure que la chaîne de formatage est analysée, la fonction est capable de déterminer s'il ya un autre argument variadique attendu et quel est son type.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note
type promotion from char to int */
                    break;
                case 'd' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
                    break;

                case 'f' :
                    f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
                    break;
                default :
                    f = -1; /* invalid format specifier */
                    break;
            }
        }
        else
        {
            f = printf("%c", *format); /* print any other characters */
        }

        if (f < 0) /* check for errors */
        {
            printed = f;
            break;
        }
```

```
        else
        {
            printed += f;
        }
        ++format; /* move on to next character in string */
    }

    va_end(ap); /* end variadic argument processing */

    return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

    y = simple_printf("There are %d characters in this sentence", x);
    simple_printf("\n%d were printed\n", y);
}
```

Lire Arguments variables en ligne: https://riptutorial.com/fr/c/topic/455/arguments-variables

Chapitre 6: Assemblage en ligne

Remarques

L'assemblage en ligne est la pratique consistant à ajouter des instructions d'assemblage au milieu du code source C. Aucune norme ISO C ne nécessite le support d'un assemblage en ligne. Comme il n'est pas requis, la syntaxe de l'assemblage en ligne varie d'un compilateur à l'autre. Même s'il est généralement pris en charge, il y a très peu de raisons d'utiliser l'assemblage en ligne et de nombreuses raisons de ne pas le faire.

Avantages

1. **Performances** En écrivant les instructions d'assemblage spécifiques pour une opération, vous pouvez obtenir de meilleures performances que le code assembleur généré par le compilateur. Notez que ces gains de performance sont rares. Dans la plupart des cas, vous pouvez obtenir de meilleurs résultats tout en réorganisant votre code C afin que l'optimiseur puisse faire son travail.
2. **Interface matérielle Le** code de démarrage du pilote de périphérique ou du processeur peut nécessiter un code d'assemblage pour accéder aux registres appropriés et pour garantir que certaines opérations se produisent dans un ordre spécifique avec un délai spécifique entre les opérations.

Les inconvénients

1. **Portabilité du compilateur La** syntaxe de l'assemblage en ligne n'est pas garantie d'un même compilateur à l'autre. Si vous écrivez du code avec un assemblage en ligne qui doit être pris en charge par différents compilateurs, utilisez des macros de préprocesseur (`#ifdef`) pour vérifier quel compilateur est utilisé. Ensuite, écrivez une section d'assemblage en ligne distincte pour chaque compilateur pris en charge.
2. **Portabilité du processeur** Vous ne pouvez pas écrire un assemblage en ligne pour un processeur x86 et attendez-vous à ce qu'il fonctionne sur un processeur ARM. L'assemblage en ligne est destiné à être écrit pour un processeur ou une famille de processeurs spécifique. Si vous avez un assemblage en ligne que vous souhaitez prendre en charge sur différents processeurs, utilisez des macros de préprocesseur pour vérifier le processeur pour lequel le code est compilé et pour sélectionner la section de code d'assemblage appropriée.
3. **Modifications futures des performances** L'assemblage en ligne peut être écrit en anticipant les retards en fonction d'une certaine vitesse d'horloge du processeur. Si le programme est compilé pour un processeur avec une horloge plus rapide, le code de l'assembly peut ne pas fonctionner comme prévu.

Examples

gcc Basic asm support

La prise en charge de l'assembly de base avec gcc a la syntaxe suivante:

```
asm [ volatile ] ( AssemblerInstructions )
```

où `AssemblerInstructions` est le code d'assemblage direct du processeur donné. Le mot clé volatile est facultatif et n'a aucun effet car gcc n'optimise pas le code dans une instruction asm de base. `AssemblerInstructions` instructions d'assemblage peuvent contenir plusieurs instructions d'assemblage. Une instruction asm de base est utilisée si vous avez une routine asm devant exister en dehors d'une fonction C. L'exemple suivant provient du manuel GCC:

```
/* Note that this code will not compile with -masm=intel */
#define DebugBreak() asm("int $3")
```

Dans cet exemple, vous pouvez alors utiliser `DebugBreak()` à d'autres endroits de votre code pour exécuter l'instruction d'assemblage `int $3` . Notez que même si gcc ne modifie aucun code dans une instruction asm de base, l'optimiseur peut toujours déplacer les instructions asm consécutives. Si vous avez plusieurs instructions d'assemblage qui doivent apparaître dans un ordre spécifique, incluez-les dans une seule instruction asm.

gcc Support asm étendu

La prise en charge asm étendue dans gcc a la syntaxe suivante:

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

où `AssemblerTemplate` est le modèle de l'instruction assembleur, `OutputOperands` est une variable C pouvant être modifiée par le code assembleur, `InputOperands` est une variable C utilisée comme paramètre d'entrée, `Clobbers` est une liste ou un registre modifié par le code assembleur et `GotoLabels` sont des étiquettes de déclaration goto qui peuvent être utilisées dans le code d'assemblage.

Le format étendu est utilisé dans les fonctions C et constitue l'utilisation la plus courante de l'assemblage en ligne. Vous trouverez ci-dessous un exemple du noyau Linux pour l'échange d'octets de nombres 16 bits et 32 bits pour un processeur ARM:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#if __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
```

```
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Chaque section asm utilise la variable `x` comme paramètre d'entrée et de sortie. La fonction C renvoie alors le résultat manipulé.

Avec le format asm étendu, gcc peut optimiser les instructions d'assemblage dans un bloc asm en suivant les mêmes règles que celles utilisées pour optimiser le code C. Si vous souhaitez que votre section asm reste intacte, utilisez le mot-clé `volatile` pour la section asm.

Assemblage en ligne gcc dans les macros

Nous pouvons mettre des instructions d'assemblage dans une macro et utiliser la macro comme vous appelleriez une fonction.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

L'utilisation d'instructions d'assemblage intégrées au code C peut améliorer le temps d'exécution d'un programme. Ceci est très utile dans les situations critiques pour le temps, comme les algorithmes cryptographiques tels que AES. Par exemple, pour une opération de décalage simple nécessaire dans l'algorithme AES, nous pouvons substituer une instruction d'assemblage `Rotate Right` directe à l'opérateur C shift `>>` .

Dans une implémentation de 'AES256', dans la fonction 'AddRoundKey ()' nous avons des instructions comme ceci:

```
unsigned int w;          // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;     // hold 8 bit, MSB, leftmost group of 8-bits
subkey[1] = w >> 16;     // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;      // hold 8 bit, second group of 8-bit from right
subkey[3] = w;           // hold 8 bit, LSB, rightmost group of 8-bits
```

```
/// subkey <- w
```

Ils attribuent simplement la valeur de bit de `w` au tableau de `subkey` .

Nous pouvons changer trois shift + assign et une autre assigner une expression avec un seul assemblage. `Rotate Right` operation.

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

Le résultat final est exactement le même.

Lire Assemblage en ligne en ligne: https://riptutorial.com/fr/c/topic/4263/assemblage-en-ligne

Chapitre 7: Atomique

Syntaxe

- `#ifdef __STDC_NO_ATOMICS__`
- `# error this implementation needs atomics`
- `#endif`
- `#include <stdatomic.h>`
- _Atomic counter non signé = ATOMIC_VAR_INIT (0);

Remarques

Atomics dans le cadre du langage C est une fonctionnalité optionnelle disponible depuis C11.

Leur objectif est de garantir un accès sans race aux variables partagées entre différents threads. Sans qualification atomique, l'état d'une variable partagée serait indéfini si deux threads y accèdent simultanément. Par exemple, une opération d'incrémentation (`++`) peut être divisée en plusieurs instructions d'assembleur, une lecture, l'addition elle-même et une instruction de stockage. Si un autre thread effectuait la même opération, ses deux séquences d'instructions pourraient être entrelacées et entraîner un résultat incohérent.

- **Types:** Tous les types d'objets à l'exception des types de tableaux peuvent être qualifiés avec `_Atomic` .
- **Opérateurs:** Tous les **opérateurs de** lecture-modification-écriture (par exemple `++` ou `*=`) sur ceux-ci sont garantis atomiques.
- **Opérations:** Il existe d'autres opérations spécifiées en tant que fonctions génériques de type, par exemple `atomic_compare_exchange` .
- **Threads: il** est garanti que l'accès à ces **threads** ne produira pas de race de données lorsqu'ils sont accessibles par différents threads.
- **Gestionnaires de signaux:** Les types atomiques sont appelés sans *verrouillage* si toutes les opérations sur eux sont sans état. Dans ce cas, ils peuvent également être utilisés pour traiter les changements d'état entre un flux de contrôle normal et un gestionnaire de signal.
- Un seul type de données est garanti sans verrou: `atomic_flag` . C'est un type minimal dont les opérations sont destinées à être mappées sur des instructions matérielles efficaces de test et de configuration.

D'autres moyens d'éviter les conditions de `mtx_t` sont disponibles dans l'interface de thread de C11, en particulier un type de mutex `mtx_t` pour exclure mutuellement les threads d'accès aux données critiques ou les sections critiques du code. Si les atomes ne sont pas disponibles, ils doivent être utilisés pour empêcher les courses.

Examples

atomiques et opérateurs

Les variables atomiques sont accessibles simultanément entre différents threads sans créer de conditions de course.

```
/* a global static variable that is visible by all threads */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);


int myThread(void* a) {
  ++active;         // increment active race free
  // do something
  --active;         // decrement active race free
  return 0;
}
```

Toutes les opérations lvalue (opérations qui modifient l'objet) autorisées pour le type de base sont autorisées et ne conduisent pas à des conditions de concurrence entre les différents threads qui y accèdent.

- Les opérations sur des objets atomiques sont généralement des ordres de grandeur plus lents que les opérations arithmétiques normales. Cela inclut également les opérations simples de chargement ou de stockage. Vous ne devez donc les utiliser que pour des tâches critiques.
- Opérations arithmétiques habituelles et affectation telles que `a = a+1;` sont en fait trois opérations sur `a` : tout d' abord une charge, puis addition et enfin un magasin. Ce n'est *pas* gratuit. Seule l'opération `a += 1;` et `a++;` sont.

Lire Atomique en ligne: https://riptutorial.com/fr/c/topic/4924/atomique

Chapitre 8: Booléen

Remarques

Pour utiliser le type prédéfini `_Bool` et l'en-tête `<stdbool.h>` , vous devez utiliser les versions C99 / C11 de C.

Pour éviter les avertissements du compilateur et éventuellement les erreurs, utilisez uniquement l'exemple `typedef` / `define` si vous utilisez C89 et les versions précédentes du langage.

Examples

Utiliser stdbool.h

C99

L'utilisation du fichier d'en-tête système `stdbool.h` vous permet d'utiliser `bool` comme un type de données booléen. `true 1` et `false true 0` .

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;  /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x)  /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`bool` est juste une belle orthographe pour le type de données `_Bool` . Il a des règles spéciales lorsque les nombres ou les pointeurs y sont convertis.

Utiliser #define

C de toutes les versions, traitera efficacement toute valeur entière autre que `0` comme `true` pour les opérateurs de comparaison et la valeur entière `0` comme étant `false` . Si vous ne disposez pas de `_Bool` ou `bool` partir de C99, vous pouvez simuler un type de données booléen en C à l'aide de macros `#define` , et vous pouvez toujours trouver ces éléments dans le code hérité.

```
#include <stdio.h>

#define bool int
#define true 1
```

```
#define false 0

int main(void) {
    bool x = true;  /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

N'introduisez pas cela dans un nouveau code car la définition de ces macros pourrait être en `<stdbool.h>` avec les utilisations modernes de `<stdbool.h>` .

Utilisation de _Bool de type intrinsèque (intégré)

C99

Ajouté dans la version C standard C99, `_Bool` est également un type de données C natif. Il est capable de contenir les valeurs `0` (pour *false*) et `1` (pour *true*).

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

`_Bool` est un type entier mais a des règles spéciales pour les conversions d'autres types. Le résultat est analogue à l'utilisation d'autres types dans les expressions `if` . Dans ce qui suit

```
_Bool z = X;
```

- Si `X` a un type arithmétique (est un type quelconque de nombre), `z` devient `0` si `X == 0` . Sinon, `z` devient `1` .
- Si `X` a un type de pointeur, `z` devient `0` si `X` est un pointeur nul et `1` sinon.

Pour utiliser les orthographes plus belles `bool` , `false` et `true` vous devez utiliser `<stdbool.h>` .

Entiers et pointeurs dans les expressions booléennes.

Tous les entiers ou pointeurs peuvent être utilisés dans une expression interprétée comme "valeur de vérité".

```
int main(int argc, char* argv[]) {
  if (argc % 4) {
    puts("arguments number is not divisible by 4");
  } else {
    puts("argument number is divisible by 4");
  }
...
```

L'expression `argc % 4` est évaluée et conduit à l'une des valeurs `0` , `1` , `2` ou `3` . Le premier, `0` est la seule valeur qui est "false" et amène l'exécution dans la partie `else` . Toutes les autres valeurs sont "vraies" et vont dans la partie `if` .

```
double* A = malloc(n*sizeof *A);
if (!A) {
   perror("allocation problems");
   exit(EXIT_FAILURE);
}
```

Ici, le pointeur `A` est évalué et s'il s'agit d'un pointeur nul, une erreur est détectée et le programme se ferme.

Beaucoup de gens préfèrent écrire quelque chose comme `A == NULL` , mais si vous faites de telles comparaisons avec d'autres expressions compliquées, les choses deviennent rapidement difficiles à lire.

```
char const* s = ....;   /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

Pour ce faire, vous devez analyser un code compliqué dans l'expression et être sûr de la préférence de l'opérateur.

```
char const* s = ....;   /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

est relativement facile à capturer: si le pointeur est valide, nous vérifions si le premier caractère est différent de zéro et vérifions ensuite s'il s'agit d'une lettre.

Définir un type bool en utilisant typedef

Étant donné que la plupart des débogueurs ne connaissent pas les macros `#define` , mais peuvent vérifier les constantes d' `enum` , il peut être souhaitable de faire quelque chose comme ceci:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
```

```
/* Modern C code might expect these to be macros. */
# ifndef bool
#  define bool bool
# endif
# ifndef true
#  define true true
# endif
# ifndef false
#  define false false
# endif
#else
# include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

Cela permet aux compilateurs pour les versions historiques de C de fonctionner, mais reste compatible avec le futur si le code est compilé avec un compilateur C moderne.

Pour plus d'informations sur `typedef` , voir Typedef , pour plus d'informations sur `enum` voir Enumérations

Lire Booléen en ligne: https://riptutorial.com/fr/c/topic/3336/booleen

Chapitre 9: Champs de bits

Introduction

La plupart des variables de C ont une taille qui est un nombre entier d'octets. Les champs de bits font partie d'une structure qui n'occupe pas nécessairement un nombre entier d'octets; ils peuvent n'importe quel nombre de bits. Plusieurs champs de bits peuvent être regroupés dans une seule unité de stockage. Ils font partie de la norme C, mais de nombreux aspects sont définis pour la mise en œuvre. Ils sont l'une des parties les moins portables de C.

Syntaxe

- identificateur de spécificateur de type: taille;

Paramètres

Paramètre	La description
spécificateur de type	`signed` , `unsigned` , `int` ou `_Bool`
identifiant	Le nom de ce champ dans la structure
Taille	Le nombre de bits à utiliser pour ce champ

Remarques

Les seuls types portables pour les champs de bits sont `signed` , `unsigned` ou `_Bool` . Le type plain `int` peut être utilisé, mais le standard indique (§6.7.2¶5) *… pour les champs de bits, il est défini par l'implémentation si le spécificateur `int` désigne le même type que `signed int` ou le même type que `unsigned int` .*

D'autres types d'entiers peuvent être autorisés par une implémentation spécifique, mais leur utilisation n'est pas portable.

Examples

Champs de bits

Un simple champ de bits peut être utilisé pour décrire des éléments pouvant comporter un nombre spécifique de bits.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
```

```
    unsigned int encoderTurns  : 4;
    unsigned int _reserved     : 5;
};
```

Dans cet exemple, nous considérons un codeur avec 23 bits de simple précision et 4 bits pour décrire le multi-tour. Les champs de bits sont souvent utilisés lors de l'interfaçage avec du matériel qui génère des données associées à un nombre spécifique de bits. Un autre exemple pourrait être la communication avec un FPGA, où le FPGA écrit des données dans votre mémoire en sections de 32 bits, ce qui permet des lectures de matériel:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb1On  : 1;
            unsigned int bulb2On  : 1;
            unsigned int bulb1Off : 1;
            unsigned int bulb2Off : 1;
            unsigned int jetOn    : 1;
        };
        unsigned int data;
   };
};
```

Pour cet exemple, nous avons montré une construction couramment utilisée pour pouvoir accéder aux données dans ses bits individuels, ou pour écrire le paquet de données dans son ensemble (en émulant ce que le FPGA peut faire). Nous pourrions alors accéder aux bits comme ceci:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

Ceci est valable, mais conformément à la norme C99 6.7.2.1, article 10:

> L'ordre d'attribution des champs de bits au sein d'une unité (d'ordre élevé à faible ou faible à élevé) est défini par la mise en œuvre.

Vous devez être conscient de l’endianness lors de la définition des champs de bits de cette manière. En tant que tel, il peut être nécessaire d’utiliser une directive de préprocesseur pour vérifier l’état de la machine. Un exemple de ceci suit:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
    uint8_t commFailure :1;
    uint8_t hardwareFailure :1;
    uint8_t _reserved :6;
#else
    uint8_t _reserved :6;
    uint8_t hardwareFailure :1;
    uint8_t commFailure :1;
#endif
    };
```

```
    uint8_t data;
} hardwareStatus;
```

Utilisation de champs de bits sous forme de petits entiers

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Alignement du champ binaire

Les champs de bits permettent de déclarer des champs de structure plus petits que la largeur des caractères. Les champs de bits sont implémentés avec un masque de niveau octet ou de niveau mot. L'exemple suivant résulte en une structure de 8 octets.

```
struct C
{
    short s;            /* 2 bytes */
    char  c;            /* 1 byte */
    int   bit1 : 1;     /* 1 bit */
    int   nib  : 4;     /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int   sept : 7;     /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

Les commentaires décrivent une disposition possible, mais comme la norme stipule que *l'alignement de l'unité de stockage adressable n'est pas spécifié* , d'autres dispositions sont également possibles.

Un champ de bit sans nom peut avoir n'importe quelle taille, mais il ne peut pas être initialisé ou référencé.

Un champ de bits de largeur nulle ne peut pas recevoir de nom et aligne le champ suivant sur la limite définie par le type de données du champ de bits. Ceci est réalisé en remplissant des bits entre les champs de bits.

La taille de la structure 'A' est de 1 octet.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

Dans la structure B, le premier champ de bits sans nom saute 2 bits; le champ binaire de largeur zéro après `c2` provoque le démarrage de `c3` à partir de la limite du caractère (donc 3 bits sont ignorés entre `c2` et `c3` . Il y a 3 bits de remplissage après `c4` . La taille de la structure est donc de 2 octets.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char    : 2;    /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char    : 0;    /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

Quand les champs de bits sont-ils utiles?

Un champ de bits est utilisé pour regrouper de nombreuses variables en un seul objet, similaire à une structure. Cela permet une utilisation réduite de la mémoire et est particulièrement utile dans un environnement intégré.

```
 e.g. consider the following variables having the ranges as given below.
 a --> range 0 - 3
 b --> range 0 - 1
 c --> range 0 - 7
 d --> range 0 - 1
 e --> range 0 - 1
```

Si nous déclarons ces variables séparément, chacune doit comporter au moins un entier sur 8 bits et l’espace total requis sera de 5 octets. De plus, les variables n'utiliseront pas toute la plage d'un entier non signé de 8 bits (0-255). Ici, nous pouvons utiliser des champs de bits.

```
typedef struct {
   unsigned int a:2;
   unsigned int b:1;
   unsigned int c:3;
   unsigned int d:1;
   unsigned int e:1;
} bit_a;
```

Les champs de bits dans la structure sont accessibles de la même manière que toute autre structure. Le programmeur doit veiller à ce que les variables soient écrites dans la plage. En dehors des limites, le comportement est indéfini.

```
int main(void)
{
   bit_a bita_var;
   bita_var.a = 2;              // to write into element a
   printf ("%d",bita_var.a);    // to read from element a.
   return 0;
}
```

Souvent, le programmeur veut mettre à zéro l'ensemble des champs de bits. Cela peut être fait élément par élément, mais il existe une seconde méthode. Créez simplement une union de la structure ci-dessus avec un type non signé supérieur ou égal à la taille de la structure. Ensuite, l'ensemble des champs de bits peut être mis à zéro en remettant à zéro cet entier non signé.

```
typedef union {
    struct {
       unsigned int a:2;
       unsigned int b:1;
       unsigned int c:3;
       unsigned int d:1;
       unsigned int e:1;
    };
    uint8_t data;
} union_bit;
```

L'utilisation est la suivante

```
int main(void)
{
   union_bit un_bit;
   un_bit.data = 0x00;        // clear the whole bit-field
   un_bit.a = 2;              // write into element a
   printf ("%d",un_bit.a);    // read from element a.
   return 0;
}
```

En conclusion, les champs de bits sont couramment utilisés dans les situations de contraintes de mémoire où vous avez beaucoup de variables qui peuvent prendre des plages limitées.

À ne pas faire pour les champs de bits

1. Les tableaux de champs de bits, les pointeurs vers les champs de bits et les fonctions renvoyant des champs de bits ne sont pas autorisés.
2. L'opérateur d'adresse (&) ne peut pas être appliqué aux membres du champ binaire.
3. Le type de données d'un champ de bits doit être suffisamment large pour contenir la taille du champ.
4. L'opérateur `sizeof()` ne peut pas être appliqué à un champ de bits.
5. Il n'y a aucun moyen de créer un `typedef` pour un champ de bits isolé (bien que vous puissiez certainement créer un `typedef` pour une structure contenant des champs de bits).

```
typedef struct mybitfield
{
    unsigned char c1 : 20;   /* incorrect, see point 3 */
```

```
    unsigned char c2 : 4;     /* correct */
    unsigned char c3 : 1;
    unsigned int x[10]: 5;    /* incorrect, see point 1 */
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { … };
    printf("Address of a.c2 is %p\n", &a.c2);      /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

Lire Champs de bits en ligne: https://riptutorial.com/fr/c/topic/1930/champs-de-bits

Chapitre 10: Classes de stockage

Introduction

Une classe de stockage est utilisée pour définir la portée d'une variable ou d'une fonction. En connaissant la classe de stockage d'une variable, nous pouvons déterminer la durée de vie de cette variable pendant l'exécution du programme.

Syntaxe

- [auto | register | static | extern] <type de données> <nom de la variable> [= <valeur>];
- [static _Thread_local | extern _Thread_local | _Thread_local] <Type de données> <Nom de la variable> [= <Valeur>]; / * depuis> = C11 * /
- Exemples:
- typedef int *foo* ;
- extern int *foo* [2];

Remarques

Les spécificateurs de classe de stockage sont les mots-clés qui peuvent apparaître à côté du type de niveau supérieur d'une déclaration. L'utilisation de ces mots-clés affecte la durée de stockage et le couplage de l'objet déclaré, selon qu'il est déclaré à portée de fichier ou à portée de bloc:

Mot-clé	Durée de stockage	Lien	Remarques
`static`	Statique	Interne	Définit la liaison interne pour les objets à portée de fichier; définit la durée de stockage statique pour les objets à portée de bloc.
`extern`	Statique	Externe	Implicite et donc redondant pour les objets définis au niveau du fichier qui ont également un initialiseur. Lorsqu'elle est utilisée dans une déclaration à portée de fichier sans initialiseur, cela indique que la définition doit être trouvée dans une autre unité de traduction et sera résolue au moment de la liaison.
`auto`	Automatique	Sans importance	Implicite et donc redondant pour les objets déclarés à portée de bloc.
`register`	Automatique	Sans	Ne concerne que les objets avec une durée de

Mot-clé	Durée de stockage	Lien	Remarques
		importance	stockage automatique. Indique que la variable doit être stockée dans un registre. Une contrainte imposée est que l'on ne peut pas utiliser l'opérateur unaire `&` "address of" sur un tel objet, et par conséquent l'objet ne peut pas être aliasé.
`typedef`	Sans importance	Sans importance	Pas un spécificateur de classe de stockage dans la pratique, mais fonctionne comme un point de vue syntaxique. La seule différence est que l'identifiant déclaré est un type plutôt qu'un objet.
`_Thread_local`	Fil	Interne externe	Introduit en C11, pour représenter *la durée de stockage des threads* . S'il est utilisé à portée de bloc, il doit également inclure `extern` ou `static` .

Chaque objet a une durée de stockage associée (indépendamment de la portée) et une liaison (pertinente pour les déclarations à portée de fichier uniquement), même lorsque ces mots-clés sont omis.

L'ordre des spécificateurs de classe de stockage en fonction des spécificateurs de type de niveau supérieur (`int` , `unsigned` , `short` , etc.) et des qualificateurs de niveau supérieur (`const` , `volatile`) n'est pas appliqué. Ces deux déclarations sont donc valides:

```
int static const unsigned a = 5; /* bad practice */
static const unsigned int b = 5; /* good practice */
```

Toutefois, il est conseillé de placer les spécificateurs de classe de stockage en premier, puis tous les qualificatifs de type, puis le spécificateur de type (`void` , `char` , `int` , `signed long` , `unsigned long long` , `long double` ...).

Tous les spécificateurs de classe de stockage ne sont pas légaux à une certaine portée:

```
register int x; /* legal at block scope, illegal at file scope */
auto int y; /* same */

static int z; /* legal at both file and block scope */
extern int a; /* same */

extern int b = 5; /* legal and redundant at file scope, illegal at block scope */

/* legal because typedef is treated like a storage class specifier syntactically */
int typedef new_type_name;
```

Durée de stockage

La durée de stockage peut être statique ou automatique. Pour un objet déclaré, il est déterminé en fonction de son étendue et des spécificateurs de classe de stockage.

Durée de stockage statique

Les variables avec une durée de stockage statique sont présentes tout au long de l'exécution du programme et peuvent être déclarées à la fois à la portée du fichier (avec ou sans `static`) et à la portée du bloc (en mettant explicitement la `static`). Ils sont généralement alloués et initialisés par le système d'exploitation au démarrage du programme et récupérés à la fin du processus. En pratique, les formats exécutables ont des sections dédiées pour ces variables (`data` , `bss` et `rodata`) et ces sections entières du fichier sont mappées en mémoire à certaines plages.

Durée de stockage des threads

C11

Cette durée de stockage a été introduite en C11. Ce n'était pas disponible dans les normes C précédentes. Certains compilateurs fournissent une extension non standard avec une sémantique similaire. Par exemple, gcc prend en `__thread` spécificateur `__thread` qui peut être utilisé dans les normes C précédentes qui n'avaient pas `_Thread_local` .

Les variables avec une durée de stockage des threads peuvent être déclarées à la fois dans la portée du fichier et dans la portée du bloc. S'il est déclaré au niveau du bloc, il doit également utiliser un spécificateur de stockage `static` ou `extern` . Sa durée de vie est l'exécution complète du *thread* dans lequel il a été créé. C'est le seul spécificateur de stockage pouvant apparaître à côté d'un autre spécificateur de stockage.

Durée de stockage automatique

Les variables avec une durée de stockage automatique ne peuvent être déclarées qu'au niveau du bloc (directement dans une fonction ou dans un bloc de cette fonction). Ils ne sont utilisables que pendant la période entre l'entrée et la sortie de la fonction ou du bloc. Une fois la variable hors de portée (soit en revenant de la fonction, soit en quittant le bloc), son stockage est automatiquement désalloué. Toute autre référence à la même variable à partir de pointeurs est invalide et conduit à un comportement indéfini.

Dans les implémentations classiques, les variables automatiques sont situées à certains décalages dans le cadre de la pile d'une fonction ou dans des registres.

Liaison externe et interne

La liaison ne concerne que les objets (fonctions et variables) déclarés au niveau du fichier et

affecte leur visibilité sur différentes unités de traduction. Les objets avec liaison externe sont visibles dans toutes les autres unités de traduction (à condition que la déclaration appropriée soit incluse). Les objets avec liaison interne ne sont pas exposés à d'autres unités de traduction et ne peuvent être utilisés que dans l'unité de traduction où ils sont définis.

Examples

typedef

Définit un nouveau type basé sur un type existant. Sa syntaxe reflète celle d'une déclaration de variable.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;

/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

Bien que ce ne soit pas techniquement une classe de stockage, un compilateur le traitera comme une classe, car aucune des autres classes de stockage n'est autorisée si le mot-clé `typedef` est utilisé.

Les `typedef` s sont importants et ne doivent pas être remplacés par la macro `#define` .

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

cependant,

```
#define int newType
newType *ptr;        // Even though macros are exact replacements to words, this doesn't
result to a pointer to variable of type 'newType' aka int
```

auto

Cette classe de stockage indique qu'un identifiant a une durée de stockage automatique. Cela signifie qu'une fois que l'étendue dans laquelle l'identifiant a été défini se termine, l'objet désigné par l'identifiant n'est plus valide.

Puisque tous les objets, ne vivant pas dans la portée globale ou étant déclarés `static` , ont une durée de stockage automatique par défaut lorsqu'ils sont définis, ce mot clé présente un intérêt historique et ne doit pas être utilisé:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

statique

La classe de stockage `static` remplit différentes fonctions, en fonction de l'emplacement de la déclaration dans le fichier:

1. Pour confiner l'identifiant à cette unité de traduction uniquement (scope = fichier).

```
/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);
```

2. Pour enregistrer des données à utiliser avec le prochain appel d'une fonction (scope = block):

```
void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
                       * entire execution of the program; initialized to 0 on
                       * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
                * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}
```

Ce code imprime:

```
static int a = 10, int b = 10
```

```
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10
```

Les variables statiques conservent leur valeur même lorsqu'elles sont appelées depuis plusieurs threads différents.

C99

3. Utilisé dans les paramètres de fonction pour indiquer qu'un tableau doit avoir un nombre minimal constant d'éléments et un paramètre non nul:

   ```
   /* a is expected to have at least 512 elements. */
   void printInts(int a[static 512])
   {
       size_t i;
       for (i = 0; i < 512; ++i)
           printf("%d\n", a[i]);
   }
   ```

   Le nombre requis d'éléments (ou même un pointeur non nul) n'est pas nécessairement vérifié par le compilateur et les compilateurs ne sont pas tenus de vous informer de quelque manière que ce soit si vous ne disposez pas d'assez d'éléments. Si un programmeur transmet moins de 512 éléments ou un pointeur nul, le comportement non défini est le résultat. Comme il est impossible d'appliquer cette règle, il faut faire particulièrement attention lors de la transmission d'une valeur pour ce paramètre à une telle fonction.

externe

Utilisé pour **déclarer un objet ou une fonction** défini ailleurs (et qui a *un lien externe*). En général, il est utilisé pour déclarer un objet ou une fonction à utiliser dans un module autre que celui dans lequel l'objet ou la fonction correspondant est défini:

```
/* file1.c */
int foo = 2;  /* Has external linkage since it is declared at file scope. */
```

```
/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
}
```

C99

Les choses deviennent un peu plus intéressantes avec l'introduction du mot clé en `inline` dans C99:

```
/* Should usually be place in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
   Creates an external function definition of `bar` for use by other files.
   The compiler is allowed to choose between the inline version and the external
   definition when `bar` is called. Without this line, `bar` would only be an inline
   function, and other files would not be able to call it. */
extern void bar(int);
```

registre

Des astuces pour le compilateur selon lesquelles l'accès à un objet doit être aussi rapide que possible. Si le compilateur utilise réellement le conseil est défini par l'implémentation; il peut simplement le traiter comme équivalent à `auto` .

La seule propriété qui est définitivement différente pour tous les objets déclarés avec `register` est que leur adresse ne peut pas être calculée. Ce `register` peut être un bon outil pour assurer certaines optimisations:

```
register size_t size = 467;
```

est un objet qui ne peut jamais *créer d'alias* car aucun code ne peut transmettre son adresse à une autre fonction où il pourrait être modifié de manière inattendue.

Cette propriété implique également qu'un tableau

```
register int array[5];
```

ne peut pas se désintégrer en un pointeur vers son premier élément (c'est-à-dire que le `array` transforme en `&array[0]`). Cela signifie que les éléments d'un tel tableau ne sont pas accessibles et que le tableau lui-même ne peut pas être transmis à une fonction.

En fait, la seule utilisation légale d'un tableau déclaré avec une classe de stockage de `register` est la `sizeof` opérateur; tout autre opérateur aurait besoin de l'adresse du premier élément du tableau. Pour cette raison, les tableaux ne devraient généralement pas être déclarés avec le mot-clé `register` car cela les rend inutiles pour autre chose que le calcul de la taille du tableau entier, ce qui peut se faire aussi facilement sans le mot-clé `register` .

La classe de stockage des `register` est plus appropriée pour les variables définies dans un bloc et accessibles avec une fréquence élevée. Par exemple,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
{
    register int k, sum;
```

```
    for(k = 1, sum = 0; k < 6; sum += k, k++);
        printf("\t%d\n",sum);
}
```

C11

L'opérateur `_Alignof` peut également être utilisé avec `register` tableaux de `register` .

_Thread_local

C11

C'était un nouveau spécificateur de stockage introduit en C11 avec le multi-threading. Ce n'est pas disponible dans les normes C précédentes.

Indique *la durée de stockage des threads* . Une variable déclarée avec le `_Thread_local` stockage `_Thread_local` indique que l'objet est *local pour ce thread* et que sa durée de vie correspond à l'exécution complète du thread dans lequel il est créé. Il peut également apparaître avec `static` ou `extern` .

```
#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
     * Running this program will print different addresses for i, showing
     * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)
{
    thrd_t id[SIZE];
    int arr[SIZE] = {1, 2, 3, 4, 5};

    /* create 5 threads. */
    for(int i = 0; i < SIZE; i++) {
        thrd_create(&id[i], thread_func, &arr[i]);
    }

    /* wait for threads to complete. */
    for(int i = 0; i < SIZE; i++) {
        thrd_join(id[i], NULL);
    }
}
```

Lire Classes de stockage en ligne: https://riptutorial.com/fr/c/topic/3597/classes-de-stockage

Chapitre 11: commentaires

Introduction

Les commentaires sont utilisés pour indiquer quelque chose à la personne qui lit le code. Les commentaires sont traités comme un blanc par le compilateur et ne changent rien à la signification réelle du code. Deux syntaxes sont utilisées pour les commentaires dans C, l'original `/* */` et le plus récent `//` . Certains systèmes de documentation utilisent des commentaires spécialement formatés pour aider à produire la documentation du code.

Syntaxe

- `/*...*/`
- `//...` (C99 et versions ultérieures uniquement)

Examples

/ * * / commentaires délimités

Un commentaire commence par une barre oblique suivie immédiatement d'un astérisque (`/*`) et se termine dès qu'un astérisque immédiatement suivi d'une barre oblique (`*/`) est rencontré. Tout ce qui se trouve entre ces combinaisons de caractères est un commentaire et est traité comme un espace vide (ignoré) par le compilateur.

```
/* this is a comment */
```

Le commentaire ci-dessus est un commentaire sur une seule ligne. Les commentaires de ce type `/*` peuvent couvrir plusieurs lignes, comme ceci:

```
/* this is a
multi-line
comment */
```

Bien que cela ne soit pas strictement nécessaire, une convention de style commune avec les commentaires multi-lignes consiste à placer des espaces et des astérisques sur les lignes après le premier, et `/*` et `*/` sur les nouvelles lignes, de sorte qu'ils soient tous alignés:

```
/*
 * this is a
 * multi-line
 * comment
 */
```

Les astérisques supplémentaires n'ont aucun effet fonctionnel sur le commentaire car aucun d'entre eux n'a de barre oblique associée.

Ces `/*` type de commentaires peut être utilisé sur leur propre ligne, à la fin d'une ligne de code, ou même dans les lignes de code:

```
/* this comment is on its own line */
if (x && y) { /*this comment is at the end of a line */
    if ((complexCondition1) /* this comment is within a line of code */
            && (complexCondition2)) {
        /* this comment is within an if, on its own line */
    }
}
```

Les commentaires ne peuvent pas être imbriqués. C'est parce que tout `/*` ultérieur sera ignoré (dans le cadre du commentaire) et le premier `*/` atteint sera traité comme mettant fin au commentaire. Le commentaire dans l'exemple suivant *ne fonctionnera pas* :

```
/* outer comment, means this is ignored => /* attempted inner comment */ <= ends the comment,
not this one => */
```

Pour commenter des blocs de code qui contiennent des commentaires de ce type, qui seraient autrement imbriqués, consultez l'exemple de commentaire utilisant le préprocesseur ci-dessous.

// commentaires délimités

C99

C99 a introduit l'utilisation des commentaires sur une seule ligne de style C ++. Ce type de commentaire commence par deux barres obliques et se termine à la fin d'une ligne:

```
// this is a comment
```

Ce type de commentaire n'autorise pas les commentaires sur plusieurs lignes, mais il est possible de créer un bloc de commentaires en ajoutant plusieurs commentaires sur une seule ligne l'un après l'autre:

```
// each of these lines are a single-line comment
// note how each must start with
// the double forward-slash
```

Ce type de commentaire peut être utilisé sur sa propre ligne ou à la fin d'une ligne de code. Cependant, comme ils sont exécutés *à la fin de la ligne* , ils *ne* peuvent *pas* être utilisés dans une ligne de code.

```
// this comment is on its own line
if (x && y) { // this comment is at the end of a line
    // this comment is within an if, on its own line
}
```

Commenter en utilisant le préprocesseur

De gros morceaux de code peuvent également être "commentés" en utilisant les directives de préprocesseur `#if 0` et `#endif` . Ceci est utile lorsque le code contient des commentaires sur plusieurs lignes qui, autrement, ne seraient pas imbriqués.

```
#if 0 /* Starts the "comment", anything from here on is removed by preprocessor */

/* A large amount of code with multi-line comments */
int foo()
{
    /* lots of code */
    ...

    /* ... some comment describing the if statement ... */
    if (someTest) {
        /* some more comments */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* code from here on is "uncommented" (included in compiled executable) */
...
```

Piège possible dû aux trigraphes

C99

Lors de l'écriture de commentaires `//` délimités, il est possible de faire une erreur typographique qui affecte leur fonctionnement attendu. Si on tape:

```
int x = 20;  // Why did I do this??/
```

Le `/` à la fin était une faute de frappe mais maintenant sera interprété dans `\` . C'est parce que le `??/` forme un trigraphe .

Le `??/` trigraphes est en fait une longhand pour la notation `\` , qui est le symbole de continuation. Cela signifie que le compilateur pense que la ligne suivante est une continuation de la ligne en cours, c'est-à-dire une suite du commentaire, ce qui peut ne pas être ce qui est prévu.

```
int foo = 20; // Start at 20 ??/
int bar = 0;

// The following will cause a compilation error (undeclared variable 'bar')
// because 'int bar = 0;' is part of the comment on the preceding line
bar += foo;
```

Lire commentaires en ligne: https://riptutorial.com/fr/c/topic/10670/commentaires

Chapitre 12: Communication interprocessus (IPC)

Introduction

Les mécanismes de communication entre processus (IPC) permettent à différents processus indépendants de communiquer entre eux. La norme C ne fournit aucun mécanisme IPC. Par conséquent, tous ces mécanismes sont définis par le système d'exploitation hôte. POSIX définit un ensemble étendu de mécanismes IPC; Windows définit un autre ensemble; et d'autres systèmes définissent leurs propres variantes.

Examples

Sémaphores

Les sémaphores sont utilisés pour synchroniser les opérations entre deux processus ou plus. POSIX définit deux ensembles différents de fonctions de sémaphore:

1. 'System V IPC' - `semctl()` , `semop()` , `semget()` .
2. «Semaphores POSIX» - `sem_close()` , `sem_destroy()` , `sem_getvalue()` , `sem_init()` , `sem_open()` , `sem_post()` , `sem_trywait()` , `sem_unlink()` .

Cette section décrit les sémaphores System V IPC, appelés ainsi car ils proviennent de Unix System V.

Tout d'abord, vous devez inclure les en-têtes requis. Anciennes versions de POSIX requises `#include <sys/types.h>` ; POSIX moderne et la plupart des systèmes ne l'exigent pas.

```
#include <sys/sem.h>
```

Ensuite, vous devrez définir une clé à la fois pour le parent et pour l'enfant.

```
#define KEY 0x1111
```

Cette clé doit être la même dans les deux programmes ou ne pas faire référence à la même structure IPC. Il existe des moyens de générer une clé convenue sans coder en dur sa valeur.

Ensuite, en fonction de votre compilateur, vous pouvez ou non avoir besoin de faire cette étape: déclarer une union pour les opérations de sémaphore.

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};
```

Ensuite, définissez vos `semwait` *try* (`semwait`) et *raise* (`semsignal`). Les noms P et V proviennent du néerlandais

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Maintenant, commencez par obtenir l'identifiant de votre sémaphore IPC.

```
int id;
// 2nd argument is number of semaphores
// 3rd argument is the mode (IPC_CREAT creates the semaphore set if needed)
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {
    /* error handling code */
}
```

Dans le parent, initialisez le sémaphore pour avoir un compteur de 1.

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the
value of the semaphore to that specified by the union u
    /* error handling code */
}
```

Maintenant, vous pouvez décrémenter ou incrémenter le sémaphore selon vos besoins. Au début de votre section critique, vous décrémentez le compteur en utilisant la fonction `semop()` :

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

Pour incrémenter le sémaphore, vous utilisez `&v` au lieu de `&p` :

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Notez que chaque fonction retourne `0` en cas de succès et `-1` en cas d'échec. Ne pas vérifier ces états de retour peut entraîner des problèmes dévastateurs.

Exemple 1.1: Course avec des threads

Le programme ci-dessous aura un processus de `fork` un enfant et parent et enfant tenteront d'imprimer des caractères sur le terminal sans aucune synchronisation.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```
int main()
{
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

Sortie (1er tour):

```
aAABaBCbCbDDcEEcddeFFGGHHeffgghh
```

(2ème manche):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

Compiler et exécuter ce programme devrait vous donner un résultat différent à chaque fois.

Exemple 1.2: Évitez les courses avec les sémaphores

En modifiant l' *exemple 1.1* pour utiliser les sémaphores, nous avons:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};

struct sembuf p = { 0, -1, SEM_UNDO};
struct sembuf v = { 0, +1, SEM_UNDO};

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget"); exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {
        perror("semctl"); exit(12);
    }
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }

            sleep(rand() % 2);
        }
    }
    else
    {
```

```
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }

            sleep(rand() % 2);
        }
    }
}
```

Sortie:

```
aabbAABBCCccddeeDDffEEFFGGHHgghh
```

Compiler et exécuter ce programme vous donnera le même résultat à chaque fois.

Lire Communication interprocessus (IPC) en ligne:
https://riptutorial.com/fr/c/topic/10564/communication-interprocessus--ipc-

Chapitre 13: Compilation

Introduction

Le langage C est traditionnellement un langage compilé (par opposition à interprété). Le standard C définit les ***phases de traduction*** , et le produit de leur application est une image de programme (ou un programme compilé). En c11 , les phases sont listées au § 5.1.1.2.

Remarques

Extension de nom de fichier	La description
`.c`	Fichier source. Contient généralement des définitions et du code.
`.h`	En tête de fichier. Contient généralement des déclarations.
`.o`	Fichier objet. Code compilé en langage machine.
`.obj`	Extension alternative pour les fichiers objets.
`.a`	Fichier de bibliothèque. Paquet de fichiers objet.
`.dll`	Bibliothèque de liens dynamiques sous Windows.
`.so`	Objet partagé (bibliothèque) sur de nombreux systèmes de type Unix.
`.dylib`	Bibliothèque de liens dynamiques sur OSX (variante Unix).
`.exe` , `.com`	Fichier exécutable Windows. Formé en liant des fichiers d'objet et des fichiers de bibliothèque. Dans les systèmes de type Unix, il n'y a pas d'extension de nom de fichier spécial pour le fichier exécutable.

Drapeaux de compilation POSIX c99	La description
`-o filename`	Nom du fichier de sortie, par ex. (`bin/program.exe` , `program`)
`-I directory`	rechercher des en-têtes dans le `direrctory` .
`-D name`	définir le `name` macro
`-L directory`	rechercher des bibliothèques dans le `directory` .
`-l name`	bibliothèque de liens `libname` .

Les compilateurs sur les plates-formes POSIX (Linux, mainframes, Mac) acceptent généralement ces options, même si elles ne s'appellent pas `c99` .

- Voir aussi c99 - Compiler les programmes C standard

Drapeaux GCC (GNU Compiler Collection)	La description
`-Wall`	Active tous les messages d'avertissement communément acceptés pour être utiles.
`-Wextra`	Permet plus de messages d'avertissement, peut être trop bruyant.
`-pedantic`	Forcer les avertissements lorsque le code viole la norme choisie.
`-Wconversion`	Activer les avertissements sur la conversion implicite, utilisez-les avec précaution.
`-c`	Compile les fichiers source sans liaison.
`-v`	Imprime les informations de compilation.

- `gcc` accepte les drapeaux POSIX et beaucoup d'autres.
- De nombreux autres compilateurs sur les plates-formes POSIX (`clang` , compilateurs spécifiques à un fournisseur) utilisent également les indicateurs répertoriés ci-dessus.
- Voir également Invoquer GCC pour de nombreuses autres options.

Drapeaux TCC (Tiny C Compiler)	La description
`-Wimplicit-function-declaration`	Avertir de la déclaration de fonction implicite.
`-Wunsupported`	Avertissez à propos des fonctionnalités GCC non prises en charge qui sont ignorées par TCC.
`-Wwrite-strings`	Les constantes de chaîne doivent être de type const char * au lieu de char *.
`-Werror`	Abandonner la compilation si des avertissements sont émis.
`-Wall`	Activer tous les avertissements, sauf les `-Werror` , `-Wunusupported` et `-Wwrite strings` .

Examples

Le lieur

Le travail de l'éditeur de liens consiste à lier un groupe de fichiers objets (fichiers `.o`) à un exécutable binaire. Le processus de *liaison* implique principalement la *résolution des adresses symboliques en adresses numériques* . Le résultat du processus de liaison est normalement un programme exécutable.

Pendant le processus de liaison, l'éditeur de liens récupère tous les modules d'objet spécifiés sur la ligne de commande, ajoute un *code de démarrage* spécifique au système et tente de résoudre toutes *les* références *externes* du module objet avec *des définitions externes* dans d'autres fichiers objets peut être spécifié directement sur la ligne de commande ou peut être implicitement ajouté via des bibliothèques). Il va ensuite affecter *des adresses de chargement* pour les fichiers objet, c'est-à-dire qu'il spécifie où le code et les données vont se retrouver dans l'espace adresse du programme fini. Une fois qu'il a les adresses de chargement, il peut remplacer toutes les adresses symboliques dans le code objet par des adresses numériques "réelles" dans l'espace adresse de la cible. Le programme est prêt à être exécuté maintenant.

Cela inclut à la fois les fichiers objet créés par le compilateur à partir de vos fichiers de code source et les fichiers objets pré-compilés pour vous et collectés dans des fichiers de bibliothèque. Ces fichiers portent des noms se `.a` par `.a` ou `.so` , et vous n'avez normalement pas besoin de les connaître, car l'éditeur de liens sait où se trouvent la plupart d'entre eux et les relie automatiquement si nécessaire.

Invocation implicite de l'éditeur de liens

Comme le pré-processeur, l'éditeur de liens est un programme distinct, souvent appelé `ld` (mais Linux utilise `collect2` , par exemple). Tout comme le pré-processeur, l'éditeur de liens est automatiquement appelé pour vous lorsque vous utilisez le compilateur. Ainsi, la manière normale d'utiliser l'éditeur de liens est la suivante:

```
% gcc foo.o bar.o baz.o -o myprog
```

Cette ligne indique au compilateur de relier trois fichiers objets (`foo.o` , `bar.o` et `baz.o`) dans un fichier exécutable binaire nommé `myprog` . Maintenant, vous avez un fichier appelé `myprog` que vous pouvez exécuter et qui, espérons-le, fera quelque chose de cool et / ou utile.

Invocation explicite de l'éditeur de liens

Il est possible d'appeler directement l'éditeur de liens, mais cela est rarement conseillé et est généralement très spécifique à la plate-forme. C'est-à-dire que les options qui fonctionnent sous Linux ne fonctionneront pas nécessairement sous Solaris, AIX, macOS, Windows et tout autre plate-forme. Si vous travaillez avec GCC, vous pouvez utiliser `gcc -v` pour voir ce qui est exécuté en votre nom.

Options pour l'éditeur de liens

L'éditeur de liens prend également des arguments pour modifier son comportement. La

commande suivante indique à gcc de lier `foo.o` et `bar.o` , mais inclut également la bibliothèque `ncurses` .

```
% gcc foo.o bar.o -o foo -lncurses
```

C'est en fait (plus ou moins) équivalent à

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(bien que `libncurses.so` puisse être `libncurses.a` , qui est juste une archive créée avec `ar`). Notez que vous devez lister les bibliothèques (par chemin ou via les options `-lname`) après les fichiers objets. Avec les bibliothèques statiques, l'ordre dans lequel elles sont spécifiées est important. souvent, avec les bibliothèques partagées, l'ordre n'a pas d'importance.

Notez que sur de nombreux systèmes, si vous utilisez des fonctions mathématiques (à partir de `<math.h>`), vous devez spécifier `-lm` pour charger la bibliothèque de mathématiques, mais Mac OS X et macOS Sierra ne le requièrent pas. Il existe d'autres bibliothèques qui sont des bibliothèques distinctes sur Linux et d'autres systèmes Unix, mais pas sur les threads macOS - POSIX, et les bibliothèques POSIX en temps réel et réseau sont des exemples. Par conséquent, le processus de liaison varie entre les plates-formes.

Autres options de compilation

C'est tout ce que vous devez savoir pour commencer à compiler vos propres programmes C. En règle générale, nous vous recommandons également d'utiliser l'option de ligne de commande `-Wall` :

```
% gcc -Wall -c foo.cc
```

L'option `-Wall` fait que le compilateur vous avertit des constructions de code légales mais douteuses, et vous aidera à détecter beaucoup de bogues très tôt.

Si vous voulez que le compilateur vous envoie plus d'avertissements (y compris les variables déclarées mais non utilisées, en oubliant de retourner une valeur, etc.), vous pouvez utiliser cet ensemble d'options, car `-Wall` , malgré le nom, ne tourne pas *tous les avertissements possibles* sur:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>     -Wmissing-declarations -Wredundant-decls -Wshadow …
```

Notez que `clang` a une option `-Weverything` qui `-Weverything` vraiment tous les avertissements dans le `clang` .

Types de fichier

La compilation de programmes C nécessite de travailler avec cinq types de fichiers:

1. **Fichiers source** : ces fichiers contiennent des définitions de fonctions et portent des noms se `.c` par `.c` par convention. Remarque: `.cc` et `.cpp` sont des fichiers C ++; *pas de* fichiers C par exemple, `foo.c`

2. **Fichiers d'en-tête** : ces fichiers contiennent des prototypes de fonctions et diverses instructions de pré-traitement (voir ci-dessous). Ils sont utilisés pour permettre aux fichiers de code source d'accéder à des fonctions définies en externe. Les fichiers d'en-tête se terminent par `.h` par convention.
   par exemple, `foo.h`

3. **Fichiers d'objets** : Ces fichiers sont produits en tant que sortie du compilateur. Ils sont constitués de définitions de fonctions sous forme binaire, mais ils ne sont pas exécutables par eux-mêmes. Les fichiers objets se terminent par `.o` bien que sur certains systèmes d'exploitation (par exemple Windows, MS-DOS), ils se terminent souvent par `.obj` .
   par exemple, `foo.o` `foo.obj`

4. **Exécutables binaires** : Ils sont produits en tant que sortie d'un programme appelé "linker". Le lieur relie un certain nombre de fichiers objets pour produire un fichier binaire pouvant être directement exécuté. Les exécutables binaires n'ont pas de suffixe spécial sur les systèmes d'exploitation Unix, bien qu'ils se terminent généralement par `.exe` sous Windows.
   par exemple, `foo` `foo.exe`

5. **Bibliothèques** : Une bibliothèque est un fichier binaire compilé mais n'est pas en soi un exécutable (c.-à-d. Qu'il n'y a pas de fonction `main()` dans une bibliothèque). Une bibliothèque contient des fonctions pouvant être utilisées par plusieurs programmes. Une bibliothèque doit être livrée avec des fichiers d'en-tête contenant des prototypes pour toutes les fonctions de la bibliothèque. Ces fichiers d'en-tête doivent être référencés (par exemple, `#include <library.h>`) dans tout fichier source utilisant la bibliothèque. L'éditeur de liens doit alors être référé à la bibliothèque pour que le programme puisse être compilé avec succès. Il existe deux types de bibliothèques: statique et dynamique.

   - **Bibliothèque statique** : Une bibliothèque statique (fichiers `.a` pour les systèmes POSIX et les fichiers `.lib` pour Windows - à ne pas confondre avec les fichiers de bibliothèque d'importation de DLL , qui utilisent également l'extension `.lib`) est intégrée de manière statique dans le programme. Les bibliothèques statiques ont l'avantage que le programme sait exactement quelle version d'une bibliothèque est utilisée. Par ailleurs, les tailles des exécutables sont plus grandes car toutes les fonctions de bibliothèque utilisées sont incluses.
     par exemple, `libfoo.a` `foo.lib`
   - **Bibliothèque dynamique** : Une bibliothèque dynamique (fichiers `.so` pour la plupart des systèmes POSIX, `.dylib` pour les fichiers OSX et `.dll` pour Windows) est liée dynamiquement au moment de l'exécution par le programme. Celles-ci sont parfois appelées bibliothèques partagées, car une image de bibliothèque peut être partagée par de nombreux programmes. Les bibliothèques dynamiques ont l'avantage de prendre moins d'espace disque si plusieurs applications utilisent la bibliothèque. En outre, ils autorisent les mises à jour de la bibliothèque (corrections de bogues) sans avoir à reconstruire les fichiers exécutables.
     par exemple, `foo.so` `foo.dylib` `foo.dll`

Le préprocesseur

Avant que le compilateur C ne commence à compiler un fichier de code source, le fichier est traité dans une phase de prétraitement. Cette phase peut être réalisée par un programme séparé ou être complètement intégrée dans un exécutable. Dans tous les cas, il est appelé automatiquement par le compilateur avant que la compilation proprement dite ne commence. La phase de prétraitement convertit votre code source en un autre code source ou unité de traduction en appliquant des remplacements textuels. Vous pouvez le considérer comme un code source "modifié" ou "étendu". Cette source étendue peut exister en tant que fichier réel dans le système de fichiers, ou elle peut uniquement être stockée en mémoire pendant une courte période avant d'être traitée ultérieurement.

Les commandes du préprocesseur commencent par le signe dièse ("#"). Il existe plusieurs commandes de préprocesseur; deux des plus importants sont:

1. **Définit** :

   `#define` est principalement utilisé pour définir des constantes. Par exemple,

   ```
   #define BIGNUM 1000000
   int a = BIGNUM;
   ```

   devient

   ```
   int a = 1000000;
   ```

   `#define` est utilisé de cette manière pour éviter d'avoir à écrire explicitement une valeur constante à différents endroits dans un fichier de code source. Ceci est important dans le cas où vous devez changer la valeur constante plus tard; il est beaucoup moins enclin à le modifier une fois, dans le `#define` , que de devoir le changer à plusieurs endroits dispersés dans le code.

   Étant donné que `#define` uniquement des recherches et des remplacements avancés, vous pouvez également déclarer des macros. Par exemple:

   ```
   #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
   // in the function:
   a = x;
   ISTRUE(a);
   ```

   devient:

   ```
   // in the function:
   a = x;
   do {
       a = a ? 1 : 0;
   } while(0);
   ```

   En première approximation, cet effet est à peu près identique à celui des fonctions en ligne,

mais le préprocesseur ne fournit pas de vérification de type pour les macros `#define` . Ceci est bien connu pour être sujet aux erreurs et leur utilisation nécessite une grande prudence.

Notez également que le préprocesseur remplacera également les commentaires par des blancs, comme expliqué ci-dessous.

2. **Comprend** :

   `#include` est utilisé pour accéder aux définitions de fonctions définies en dehors d'un fichier de code source. Par exemple:

   ```
   #include <stdio.h>
   ```

   oblige le préprocesseur à coller le contenu de `<stdio.h>` dans le fichier de code source à l'emplacement de l'instruction `#include` avant qu'il soit compilé. `#include` est presque toujours utilisé pour inclure des fichiers d'en-tête, qui sont principalement des fichiers contenant des déclarations de fonctions et des instructions `#define` . Dans ce cas, nous utilisons `#include` pour pouvoir utiliser des fonctions telles que `printf` et `scanf` , dont les déclarations se trouvent dans le fichier `stdio.h` . Les compilateurs C ne vous autorisent pas à utiliser une fonction à moins qu'elle n'ait déjà été déclarée ou définie dans ce fichier; `#include` instructions `#include` sont donc le moyen de réutiliser le code précédemment écrit dans vos programmes C.

3. **Opérations logiques** :

   ```
   #if defined A || defined B
   variable = another_variable + 1;
   #else
   variable = another_variable * 2;
   #endif
   ```

   sera changé pour:

   ```
   variable = another_variable + 1;
   ```

   si A ou B ont été définis quelque part dans le projet auparavant. Si ce n'est pas le cas, le préprocesseur le fera bien sûr:

   ```
   variable = another_variable * 2;
   ```

   Ceci est souvent utilisé pour le code, qui s'exécute sur différents systèmes ou compile sur différents compilateurs. Comme il existe des définitions globales, spécifiques au compilateur / système, vous pouvez tester ces définitions et toujours laisser le compilateur utiliser le code qu'il compilera à coup sûr.

4. **commentaires**

   Le préprocesseur remplace tous les commentaires du fichier source par des espaces simples. Les commentaires sont indiqués par `//` jusqu'à la fin de la ligne, ou une

combinaison de parenthèses ouvrantes `/*` et de fermeture `*/` .

Le compilateur

Une fois que le pré-processeur C a inclus tous les fichiers d'en-tête et développé toutes les macros, le compilateur peut compiler le programme. Pour ce faire, il convertit le code source C en un fichier de code objet, qui se termine par `.o` et contient la version binaire du code source. Le code objet n'est pas directement exécutable. Pour rendre un exécutable, vous devez également ajouter du code pour toutes les fonctions de la bibliothèque `#include` d dans le fichier (ce n'est pas la même chose que d'inclure les déclarations, ce que fait `#include`). C'est le travail de l'éditeur de liens .

En général, la séquence exacte d'invocation d'un compilateur C dépend beaucoup du système que vous utilisez. Ici, nous utilisons le compilateur GCC, même s'il faut noter que beaucoup d'autres compilateurs existent:

```
% gcc -Wall -c foo.c
```

`%` est l'invite de commande du système d'exploitation. Cela indique au compilateur d'exécuter le pré-processeur sur le fichier `foo.c` , puis de le compiler dans le fichier de code objet `foo.o` L'option `-c` signifie compiler le fichier de code source dans un fichier objet mais pas invoquer l'éditeur de liens. Cette option `-c` est disponible sur les systèmes POSIX, tels que Linux ou macOS; d'autres systèmes peuvent utiliser une syntaxe différente.

Si votre programme entier se trouve dans un fichier de code source, vous pouvez plutôt faire ceci:

```
% gcc -Wall foo.c -o foo
```

Cela dit au compilateur d'exécuter le pré-processeur sur `foo.c` , de le compiler puis de le lier pour créer un exécutable appelé `foo` . L'option `-o` indique que le mot suivant sur la ligne est le nom du fichier exécutable binaire (programme). Si vous ne spécifiez pas `-o` , (si vous tapez simplement `gcc foo.c`), l'exécutable sera nommé `a.out` pour des raisons historiques.

En général, le compilateur effectue quatre étapes lors de la conversion d'un fichier `.c` fichier exécutable:

1. **pré-traitement** - développe textuellement les directives `#include` et les macros `#define` dans votre fichier `.c`
2. **compilation** - convertit le programme en assembleur (vous pouvez arrêter le compilateur à cette étape en ajoutant l'option `-S`)
3. **assembly** - convertit l'assemblage en code machine
4. **linkage** - lie le code objet aux bibliothèques externes pour créer un exécutable

Notez également que le nom du compilateur que nous utilisons est GCC, ce qui signifie à la fois "compilateur GNU C" et "collection de compilateurs GNU", selon le contexte. D'autres compilateurs C existent. Pour les systèmes d'exploitation de type Unix, beaucoup d'entre eux portent le nom `cc` , pour le "compilateur C", qui constitue souvent un lien symbolique vers un autre compilateur. Sur les systèmes Linux, `cc` est souvent un alias pour GCC. Sur macOS ou OS-X, cela

indique le contraire.

Les normes POSIX `c99` actuellement `c99` comme nom de compilateur C - il prend en charge le standard C99 par défaut. Les versions antérieures de POSIX `c89` comme compilateur. POSIX exige également que ce compilateur comprenne les options `-c` et `-o` que nous avons utilisées ci-dessus.

Remarque: L'option `-Wall` présente dans les deux exemples `gcc` indique au compilateur d'imprimer des avertissements sur les constructions douteuses, ce qui est fortement recommandé. C'est également une bonne idée d'ajouter d'autres options d'avertissement , par exemple `-Wextra` .

Les phases de traduction

À partir de la norme C 2011, répertoriée au *§ 5.1.1.2 Phases de traduction* , la traduction du code source en image de programme (par exemple, l'exécutable) est répertoriée pour se produire en 8 étapes ordonnées.

1. L'entrée du fichier source est associée au jeu de caractères source (si nécessaire). Les trigraphes sont remplacées à cette étape.
2. Les lignes de continuation (lignes se terminant par `\`) sont épissées avec la ligne suivante.
3. Le code source est analysé en espaces et en jetons de prétraitement.
4. Le préprocesseur est appliqué, qui exécute les directives, développe les macros et applique les pragmas. Chaque fichier source extrait par `#include` subit les phases de traduction 1 à 4 (récursivement si nécessaire). Toutes les directives liées au préprocesseur sont alors supprimées.
5. Les valeurs du jeu de caractères source dans les constantes de caractères et les littéraux de chaîne sont associées au jeu de caractères d'exécution.
6. Les littéraux de chaîne adjacents sont concaténés.
7. Le code source est analysé en jetons, qui constituent l'unité de traduction.
8. Les références externes sont résolues et l'image du programme est formée.

Une implémentation d'un compilateur C peut combiner plusieurs étapes, mais l'image résultante doit toujours se comporter comme si les étapes ci-dessus avaient eu lieu séparément dans l'ordre indiqué ci-dessus.

Lire Compilation en ligne: https://riptutorial.com/fr/c/topic/1337/compilation

Chapitre 14: Comportement défini par la mise en œuvre

Remarques

Vue d'ensemble

Le standard C décrit la syntaxe du langage, les fonctions fournies par la bibliothèque standard et le comportement des processeurs C conformes (en gros, les compilateurs) et des programmes C conformes. En ce qui concerne le comportement, la norme spécifie pour la plupart des comportements particuliers pour les programmes et les processeurs. D'un autre côté, certaines opérations ont *un comportement non défini,* explicite ou implicite - de telles opérations doivent toujours être évitées, car vous ne pouvez pas vous fier à elles. Entre les deux, il y a une variété de comportements *définis par l'implémentation* . Ces comportements peuvent varier entre les processeurs C, les runtimes et les bibliothèques standard (collectivement, les *implémentations*), mais ils sont cohérents et fiables pour toute implémentation donnée, et les implémentations conformes documentent leur comportement dans chacun de ces domaines.

Il est parfois raisonnable qu'un programme repose sur un comportement défini par l'implémentation. Par exemple, si le programme est de toute façon spécifique à un environnement d'exploitation particulier, il est peu probable que le recours à des comportements définis par l'implémentation pour les processeurs communs pour cet environnement pose problème. Alternativement, on peut utiliser des directives de compilation conditionnelle pour sélectionner les comportements définis par l'implémentation appropriés pour l'implémentation utilisée. Dans tous les cas, il est essentiel de savoir quelles opérations ont un comportement défini pour la mise en œuvre, de manière à les éviter ou à prendre une décision éclairée quant à savoir si et comment les utiliser.

Le reste de ces remarques constitue une liste de tous les comportements et caractéristiques définis dans la norme C2011, avec des références à la norme. Beaucoup d'entre eux utilisent la terminologie de la norme . Certains autres s'appuient plus généralement sur le contexte de la norme, comme les huit étapes de la traduction du code source dans un programme, ou la différence entre les implémentations hébergées et autonomes. Certains, particulièrement surprenants ou remarquables, sont présentés en caractères gras. Tous les comportements décrits ne sont pas pris en charge par les normes C antérieures, mais ils ont généralement un comportement défini par la mise en œuvre dans toutes les versions de la norme qui les prennent en charge.

Programmes et processeurs

Général

- **Le nombre de bits dans un octet** (3.6 / 3). Au moins `8` , la valeur réelle peut être

interrogée avec la macro `CHAR_BIT` .

- Quels messages de sortie sont considérés comme des "messages de diagnostic" (3.10 / 1)

Traduction source

- La manière dont les caractères multi-octets du fichier source physique sont associés au jeu de caractères source (5.1.1.2/1).
- Si les séquences non vides d'espaces blancs non newline sont remplacées par des espaces simples pendant la phase de traduction 3 (5.1.1.2/1)
- Les caractères de l'ensemble d'exécution auxquels les caractères littéraux et les caractères des constantes de chaîne sont convertis (pendant la phase de traduction 5) lorsqu'il n'y a pas d'autre caractère correspondant (5.1.1.2/1).

Environnement d'exploitation

- La manière dont les messages de diagnostic à émettre sont identifiés (5.1.1.3/1).
- Le nom et le type de la fonction appelée au démarrage dans une implémentation autonome (5.1.2.1/1).
- Quelles bibliothèques sont disponibles dans une implémentation autonome, au-delà d'un ensemble minimal spécifié (5.1.2.1/1).
- L'effet de la fin du programme dans un environnement autonome (5.1.2.1/2).
- Dans un environnement hébergé, toute signature autorisée pour la fonction `main()` autre que `int main(int argc, char *arg[])` et `int main(void)` (5.1.2.2.1 / 1).
- La manière dont une implémentation hébergée définit les chaînes pointées par le second argument de `main()` (5.1.2.2.1 / 2).
- Ce qui constitue un "dispositif interactif" aux fins des sections 5.1.2.3 (Exécution du programme) et 7.21.3 (Fichiers) (5.1.2.3/7).
- Toute restriction sur les objets auxquels les routines d'interruption de traitement font référence dans une implémentation optimisée (5.1.2.3/10).
- Dans une implémentation autonome, si plusieurs threads d'exécution sont pris en charge (5.1.2.4/1).
- Les valeurs des membres du jeu de caractères d'exécution (5.2.1 / 1).
- Les `char` valeurs correspondant aux séquences d'échappement alphabétiques définies (5.2.2 / 3).
- **Les limites et caractéristiques numériques entières et à virgule flottante** (5.2.4.2/1).

- La précision des opérations arithmétiques à virgule flottante et des conversions de la bibliothèque standard à partir de représentations internes en virgule flottante en représentations de chaînes (5.2.4.2.2 / 6).

- La valeur de la macro `FLT_ROUNDS` , qui code le mode d'arrondi à virgule flottante par défaut (5.2.4.2.2 / 8).

- Les comportements d'arrondi caractérisés par des valeurs de `FLT_ROUNDS` supérieures à 3 ou inférieures à -1 (5.2.4.2.2 / 8).

- La valeur de la macro `FLT_EVAL_METHOD` , qui caractérise le comportement d'évaluation en virgule flottante (5.2.4.2.2 / 9).

- Comportement caractérisé par des valeurs de `FLT_EVAL_METHOD` inférieures à -1 (5.2.4.2.2 / 9).

- Les valeurs des macros `FLT_HAS_SUBNORM` , `DBL_HAS_SUBNORM` et `LDBL_HAS_SUBNORM` , caractérisant si les formats à virgule flottante standard prennent en charge les nombres inférieurs à la normale (5.2.4.2.2 / 10)

Les types

- Le résultat de la tentative d'accès (indirectement) à un objet avec une durée de stockage de thread à partir d'un thread autre que celui avec lequel l'objet est associé (6.2.4 / 4)

- La valeur d'une `char` dans laquelle un caractère à l'extérieur de l'ensemble d'exécution de base a été affectée (6.2.5 / 3).

- Les types entiers signés étendus pris en charge, le cas échéant (6.2.5 / 4), et tous les mots clés d'extension utilisés pour les identifier.

- **Si `char` a la même représentation et le même comportement que le caractère `signed char` ou le caractère `unsigned char`** (6.2.5 / 15). Peut être interrogé avec `CHAR_MIN` , qui est soit `0` ou `SCHAR_MIN` si `char` est non signé ou signé, respectivement.

- **Le nombre, l'ordre et le codage des octets dans les représentations des objets** , sauf si explicitement spécifié par la norme (6.2.6.1/2).

- **Laquelle des trois formes de représentation entière reconnues s'applique dans une situation donnée, et si certains modèles de bits des objets entiers sont des représentations de pièges** (6.2.6.2/2).

- L'exigence d'alignement de chaque type (6.2.8 / 1).

- Si et dans quels contextes, tous les alignements étendus sont pris en charge (6.2.8 / 3).

- L'ensemble des alignements étendus pris en charge (6.2.8 / 4).

- Les nombres de conversion entiers de tous les types entiers signés étendus les uns par rapport aux autres (6.3.1.1/1).

- **L'affectation d'une valeur hors plage à un entier signé** (6.3.1.3/3).
- Lorsqu'une valeur à portée mais non représentable est attribuée à un objet à virgule flottante, comment la valeur représentable stockée dans l'objet est choisie parmi les deux valeurs représentables les plus proches (6.3.1.4/2 ; 6.3.1.5/1 ; 6.4.4.2 / 3).
- **Le résultat de la conversion d'un entier en un type de pointeur** , à l'exception des expressions constantes entières avec la valeur `0` (6.3.2.3/5).

Formulaire source

- Les emplacements dans les directives `#pragma` où les jetons de nom d'en-tête sont reconnus (6.4 / 4).
- Les caractères, y compris les caractères multi-octets autres que le soulignement, les lettres latines non accentuées, les noms de caractères universels et les chiffres décimaux pouvant apparaître dans les identificateurs (6.4.2.1/1).
- **Le nombre de caractères significatifs dans un identifiant** (6.4.2.1/5).
- À quelques exceptions près, la manière dont les caractères source d'une constante de caractère entier sont mappés sur des caractères d'ensemble d'exécution (6.4.4.4/2 ; 6.4.4.4/10).
- Paramètres régionaux courants utilisés pour calculer la valeur d'une constante de caractère large et la plupart des autres aspects de la conversion de nombreuses constantes de ce type (6.4.4.4/11).
- Si les jetons littéraux à chaînes larges préfixés différemment peuvent être concaténés et, si tel est le cas, le traitement de la séquence de caractères multi-octets résultante (6.4.5 / 5)
- Les paramètres régionaux utilisés lors de la phase de traduction 7 pour convertir les chaînes de caractères larges en séquences de caractères multi-octets, et leur valeur lorsque le résultat n'est pas représentable dans le jeu de caractères d'exécution (6.4.5 / 6).
- La manière dont les noms d'en-tête sont mappés aux noms de fichiers (6.4.7 / 2).

Évaluation

- `FP_CONTRACT` si et comment les expressions à virgule flottante sont contractées lorsque `FP_CONTRACT` n'est pas utilisé (6.5 / 8).
- **Les valeurs des résultats des opérateurs `sizeof` et `_Alignof`** (6.5.3.4/5).
- La taille du type de résultat de la soustraction de pointeur (6.5.6 / 9).
- **Le résultat du décalage à droite d'un entier signé avec une valeur négative** (6.5.7 / 5).

Comportement d'exécution

- La mesure dans laquelle le mot-clé de `register` est effectif (6.7.1 / 6).
- Si le type d'un champ de bits déclaré comme `int` est le même que `unsigned int` ou `signed int` (6.7.2 / 5).
- Quels types de champs de bits peuvent prendre, autres que `_Bool` , facultativement qualifiés, `_Bool` `signed int` et `unsigned int` ; si les champs de bits peuvent avoir des types atomiques (6.7.2.1/5).
- Aspects de la manière dont les implémentations organisent le stockage pour les champs de bits (6.7.2.1/11).
- L'alignement des membres non bitfield des structures et des unions (6.7.2.1/14).
- Le type sous-jacent pour chaque type énuméré (6.7.2.2/4).
- Ce qui constitue un "accès" à un objet de type `volatile` qualifié (6.7.3 / 7).
- L'efficacité des déclarations de fonctions en `inline` (6.7.4 / 6).

Préprocesseur

- Si les constantes de caractères sont converties en valeurs entières de la même manière dans les conditions préprocesseurs que dans les expressions ordinaires, et si une constante à un seul caractère peut avoir une valeur négative (6.10.1 / 4).
- Les emplacements ont recherché les fichiers désignés dans une directive `#include` (6.10.2 / 2-3).
- La manière dont un nom d'en-tête est formé à partir des jetons d'une directive `#include` multi-token (6.10.2 / 4).
- La limite pour l'imbrication `#include` (6.10.2 / 6).
- Si un caractère `\` est inséré avant le `\` introduction d'un nom de caractère universel dans le résultat de l'opérateur `#` du préprocesseur (6.10.3.2/2).
- Le comportement de la directive de prétraitement `#pragma` pour les pragmas autres que `STDC` (6.10.6 / 1).
- La valeur des macros `__DATE__` et `__TIME__` si aucune date ou heure de traduction, respectivement, n'est disponible (6.10.8.1/1).
- Le codage de caractères interne utilisé pour `wchar_t` si la macro `__STDC_ISO_10646__` n'est pas définie (6.10.8.2/1).
- Le codage de caractères interne utilisé pour `char32_t` si la macro `__STDC_UTF_32__` n'est pas défini (6.10.8.2/1).

Bibliothèque standard

Général

- Le format des messages émis lorsque les assertions échouent (7.2.1.1/2).

Fonctions d'environnement à virgule flottante

- Toutes les exceptions à virgule flottante supplémentaires au-delà de celles définies par la norme (7.6 / 6).
- Tous les modes d'arrondi à virgule flottante supplémentaires au-delà de ceux définis par la norme (7.6 / 8).
- Tout environnement à virgule flottante supplémentaire au-delà de ceux définis par la norme (7.6 / 10).
- La valeur par défaut du commutateur d'accès à l'environnement à virgule flottante (7.6.1 / 2).
- La représentation des indicateurs d'état à virgule flottante enregistrés par `fegetexceptflag()` (7.6.2.2/1).
- `feraiseexcept()` si la fonction `feraiseexcept()` déclenche en plus une exception à virgule flottante "inexacte" à chaque fois qu'elle génère une exception à virgule flottante "overflow" ou " underflow" (7.6.2.3/2).

Fonctions liées aux paramètres régionaux

- Les chaînes de paramètres régionaux autres que `"C"` supportées par `setlocale()` (7.11.1.1/3).

Fonctions mathématiques

- Les types représentés par `float_t` et `double_t` lorsque la macro `FLT_EVAL_METHOD` a une valeur différente de `0` , `1` et `2` (7.12 / 2).
- Toute classification à virgule flottante prise en charge au-delà de celles définies par la norme (7.12 / 6).
- La valeur renvoyée par le `math.h` fonctionne en cas d'erreur de domaine (7.12.1 / 2).
- La valeur renvoyée par le `math.h` fonctionne en cas d'erreur de pôle (7.12.1 / 3).
- La valeur renvoyée par le `math.h` fonctionne lorsque le résultat est sous-jacent, et indique si `errno` est défini sur `ERANGE` et si une exception à virgule flottante est déclenchée dans ces circonstances (7.12.1 / 6).

- La valeur par défaut du commutateur de contraction FP (7.12.2 / 2).
- Si les fonctions `fmod()` renvoient 0 ou `fmod()` une erreur de domaine lorsque leur second argument est 0 (7.12.10.1/3).
- Si les fonctions `remainder()` renvoient 0 ou génèrent une erreur de domaine lorsque leur second argument est 0 (7.12.10.2/3).
- Nombre de bits significatifs dans les modules quotient calculés par les fonctions `remquo()` (7.12.10.3/2).
- Si les fonctions `remquo()` renvoient 0 ou `remquo()` une erreur de domaine lorsque leur second argument est 0 (7.12.10.3/3).

Les signaux

- L'ensemble complet des signaux pris en charge, leur sémantique et leur traitement par défaut (7.14 / 4).
- Lorsqu'un signal est déclenché et qu'un gestionnaire personnalisé est associé à ce signal, quels signaux, le cas échéant, sont bloqués pendant la durée de l'exécution du gestionnaire (7.14.1.1/3).
- Les signaux autres que `SIGFPE` , `SIGILL` et `SIGSEGV` provoquent un comportement indéfini lors du retour d'un gestionnaire de signal personnalisé (7.14.1.1/3).
- Quels signaux sont initialement configurés pour être ignorés (indépendamment de leur traitement par défaut; 7.14.1.1/6).

Divers

- La constante de pointeur null spécifique à laquelle la macro `NULL` développe (7.19 / 3).

Fonctions de traitement de fichiers

- Si la dernière ligne d'un flux de texte nécessite une nouvelle ligne de terminaison (7.21.2 / 2).
- Nombre de caractères nuls ajoutés automatiquement à un flux binaire (7.21.2 / 3).
- La position initiale d'un fichier ouvert en mode annexe (7.21.3 / 1).
- Si une écriture sur un flux de texte provoque la troncature du flux (7.21.3 / 2).
- Prise en charge de la mise en mémoire tampon des flux (7.21.3 / 3).
- Indique si des fichiers de longueur nulle existent (7.21.3 / 4).
- Les règles pour composer des noms de fichiers valides (7.21.3 / 8).

- Si le même fichier peut être simultanément ouvert plusieurs fois (7.21.3 / 8).
- La nature et le choix de l'encodage pour les caractères multi-octets (7.21.3 / 10).
- Comportement de la fonction `remove()` lorsque le fichier cible est ouvert (7.21.4.1/2).
- Comportement de la fonction `rename()` lorsque le fichier cible existe déjà (7.21.4.2/2).
- Si les fichiers créés via la fonction `tmpfile()` sont supprimés dans le cas où le programme se termine anormalement (7.21.4.3/2).
- Quel mode change sous quelles circonstances sont autorisées via `freopen()` (7.21.5.4/3).

Fonctions d'E / S

- Laquelle des représentations autorisées des valeurs infinies et non-nombre de FP est produite par les fonctions printf () - family (7.21.6.1/8).
- La manière dont les pointeurs sont formatés par la fonction `printf()` (7.21.6.1/8).
- Le comportement de la famille `scanf()` fonctionne lorsque le caractère `-` apparaît dans une position interne de la liste de scrutation d'un `[` champ (7.21.6.2/12)).
- La plupart des aspects de la transmission des fonctions `p` de la famille `scanf()` (7.21.6.2/12).
- La valeur d' `errno` définie par `fgetpos()` en cas d'échec (7.21.9.1/2).
- La valeur d' `errno` définie par `fsetpos()` en cas d'échec (7.21.9.3/2).
- La valeur d' `errno` définie par `ftell()` en cas d'échec (7.21.9.4/3).
- La signification des fonctions de famille `strtod()` de certains aspects pris en charge d'un formatage NaN (7.22.1.3p4).
- `strtod()` si les `strtod()` la famille `strtod()` placent `errno` sur `ERANGE` lorsque le résultat est sous-développé (7.22.1.3/10).

Fonctions d'allocation de mémoire

- Le comportement des attributions de mémoire fonctionne lorsque le nombre d'octets requis est 0 (7.22.3 / 1).

Fonctions d'environnement système

- Quels nettoyages sont effectués, le cas échéant, et quel statut est renvoyé au système d'exploitation hôte lorsque la fonction `abort()` est appelée (7.22.4.1/2).
- Quel statut est renvoyé à l'environnement hôte lorsque `exit()` est appelé (7.22.4.4/5).

- La gestion des flux ouverts et quel statut est renvoyé à l'environnement hôte lorsque `_Exit()` est appelé (7.22.4.5/2).
- L'ensemble des noms d'environnement accessibles via `getenv()` et la méthode de modification de l'environnement (7.22.4.6/2).
- La valeur de retour de la fonction `system()` (7.22.4.8/3).

Fonctions de date et heure

- Le fuseau horaire local et l'heure d'été (7.27.1 / 1).
- La plage et la précision des temps représentables via les types `clock_t` et `time_t` (7.27.1 / 4).
- Le début de l'ère qui sert de référence pour les temps renvoyés par la fonction `clock()` (7.27.2.1/3).
- Le début de l'époque qui sert de référence pour les temps renvoyés par la fonction `timespec_get()` (lorsque la base de temps est `TIME_UTC` ; 7.27.2.5/3).
- Le remplacement `strftime()` du spécificateur de conversion `%Z` dans les paramètres régionaux "C" (7.27.3.5/7).

Fonctions d'E / S à caractères larges

- Laquelle des représentations autorisées des valeurs infinies et non-nombre de FP est produite par les `wprintf()` la famille `wprintf()` (7.29.2.1/8).
- La manière dont les pointeurs sont formatés par les `wprintf()` la famille `wprintf()` (7.29.2.1/8).
- Le comportement de la famille `wscanf()` fonctionne lorsque le caractère `-` apparaît dans une position interne de la liste de scrutation d'un `[` champ (7.29.2.2/12)).
- La plupart des aspects des `wscanf()` fonctions -family de la remise des `p` champs (7.29.2.2/12).
- La signification des fonctions de la famille `wstrtod()` de certains aspects du formatage NaN pris en charge (7.29.4.1.1 / 4).
- `wstrtod()` si les `wstrtod()` la famille `wstrtod()` placent `errno` à `ERANGE` lorsque le résultat est sous- jacent (7.29.4.1.1 / 10).

Examples

Décalage à droite d'un entier négatif

```
int signed_integer = -1;

// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Affectation d'une valeur hors plage à un entier

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is
// 127, the behavior of this assignment is implementation-defined:
signed char integer;
integer = 128;
```

Allouer zéro octet

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Représentation d'entiers signés

Chaque type entier signé peut être représenté dans l'un des trois formats; c'est l'implémentation qui est utilisée. L'implémentation utilisée pour tout type entier signé au moins aussi large que `int` peut être déterminée à l'exécution à partir des deux bits de poids faible de la représentation de la valeur `-1` dans ce type, comme ceci:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };
#define SIGN_REP(T) ((T)-1 & (T)3)

switch (SIGN_REP(long)) {
   case sign_magnitude: { /* do something */ break; }
   case ones_compl:     { /* do otherwise */ break; }
   case twos_compl:     { /* do yet else  */ break; }
   case 0:  { _Static_assert(SIGN_REP(long), "bogus sign representation"); }
}
```

Le même modèle s'applique à la représentation des types plus étroits, mais ils ne peuvent pas être testés par cette technique car les opérandes de `&` sont soumis aux "conversions arithmétiques habituelles" avant que le résultat ne soit calculé.

Lire Comportement défini par la mise en œuvre en ligne: https://riptutorial.com/fr/c/topic/4832/comportement-defini-par-la-mise-en-ouvre

Chapitre 15: Comportement non défini

Introduction

En C, certaines expressions génèrent *un comportement indéfini* . Le standard choisit explicitement de ne pas définir comment un compilateur doit se comporter s'il rencontre une telle expression. En conséquence, un compilateur est libre de faire ce qu'il juge nécessaire et peut produire des résultats utiles, des résultats inattendus, voire un crash.

Le code qui appelle UB peut fonctionner comme prévu sur un système spécifique avec un compilateur spécifique, mais ne fonctionnera probablement pas sur un autre système, ou avec un compilateur, une version de compilateur ou des paramètres de compilateur différents.

Remarques

Qu'est-ce qu'un comportement indéfini (UB)?

Le comportement non défini est un terme utilisé dans la norme C. La norme C11 (ISO / IEC 9899: 2011) définit le terme comportement indéfini comme

> comportement, lors de l'utilisation d'une structure de programme non portable ou erronée ou de données erronées, pour lesquels la présente Norme internationale n'impose aucune exigence

Que se passe-t-il s'il y a UB dans mon code?

Ce sont les résultats qui peuvent survenir en raison d'un comportement indéfini selon la norme:

> NOTE Le comportement indéfini possible peut aller de l'ignorance complète de la situation à des résultats imprévisibles, au comportement lors de la traduction ou à l'exécution du programme d'une manière documentée caractéristique de l'environnement (avec ou sans message de diagnostic), à la fin d'une traduction émission d'un message de diagnostic).

La citation suivante est souvent utilisée pour décrire (de manière moins formelle) des résultats provenant d'un comportement non défini:

> "Lorsque le compilateur rencontre [une construction indéfinie donnée], il est légal de faire voler des démons" (l'implication est que le compilateur peut choisir n'importe quelle manière arbitraire d'interpréter le code sans violer la norme ANSI C)

Pourquoi UB existe-t-il?

Si c'est si mauvais, pourquoi ne l'ont-ils pas simplement défini ou défini?

Un comportement non défini offre davantage d'opportunités d'optimisation; Le compilateur peut légitimement supposer que tout code ne contient pas de comportement indéfini, ce qui lui permet

d'éviter les vérifications à l'exécution et d'effectuer des optimisations dont la validité serait coûteuse ou impossible à prouver autrement.

Pourquoi UB est-il difficile à retrouver?

Il existe au moins deux raisons pour lesquelles un comportement non défini crée des bogues difficiles à détecter:

- Le compilateur n'est pas obligé de vous avertir - et ne peut généralement pas le faire de manière fiable - d'un comportement non défini. En fait, l'exiger de le faire irait directement à l'encontre de la raison d'être d'un comportement indéfini.
- Les résultats imprévisibles pourraient ne pas commencer à se dérouler au point exact de l'opération où se produit la construction dont le comportement n'est pas défini; Un comportement non défini entrave toute l'exécution et ses effets peuvent survenir à tout moment: pendant, après ou même *avant* la construction indéfinie.

Considérez dereference pointeur nul: le compilateur n'est pas obligé de diagnostiquer le déréférencement de pointeur nul, et même ne pourrait pas, car à l'exécution, tout pointeur passé dans une fonction ou dans une variable globale peut être nul. *Et lorsque le déréférencement de pointeur nul se produit, la norme ne prescrit pas que le programme doive se bloquer.* Au lieu de cela, le programme peut tomber en panne plus tôt, plus tard ou ne pas se bloquer du tout; Il pourrait même se comporter comme si le pointeur null indiquait un objet valide et se comporter complètement normalement, uniquement pour se bloquer dans d'autres circonstances.

Dans le cas de déréférencement de pointeur nul, le langage C diffère des langages gérés tels que Java ou C #, où le comportement du déréférencement de pointeur nul est *défini* : une exception est levée à l'heure exacte (`NullPointerException` en Java, `NullReferenceException` en C #) ainsi, ceux venant de Java ou de C # peuvent *croire de manière incorrecte que dans un tel cas, un programme C doit tomber en panne, avec ou sans émission d'un message de diagnostic* .

Information additionnelle

Plusieurs situations de ce type doivent être clairement distinguées:

- Comportement explicitement indéfini, c'est-à-dire que le standard C vous indique explicitement que vous êtes hors limites.
- Comportement implicitement indéfini, où il n'y a tout simplement pas de texte dans la norme prévoyant un comportement pour la situation dans laquelle vous avez amené votre programme.

Rappelez-vous également que dans de nombreux endroits, le comportement de certaines constructions est délibérément indéfini par le standard C pour laisser la place aux développeurs de bibliothèques et de compilateurs de proposer leurs propres définitions. Un bon exemple est celui des signaux et des gestionnaires de signaux, où les extensions de C, telles que la norme du système d'exploitation POSIX, définissent des règles beaucoup plus élaborées. Dans de tels cas, il vous suffit de vérifier la documentation de votre plate-forme; la norme C ne peut rien vous dire.

Notez également que si un comportement indéfini se produit dans le programme, cela ne signifie pas que le seul point où un comportement non défini a eu lieu est problématique, mais un

programme entier devient sans signification.

En raison de telles préoccupations, il est important (surtout que les compilateurs ne nous avertissent pas toujours sur UB) pour que la programmation de personnes en C soit au moins familière avec le genre de choses qui déclenchent un comportement indéfini.

Il convient de noter que certains outils (par exemple des outils d'analyse statique tels que PC-Lint) facilitent la détection d'un comportement indéfini, mais là encore, ils ne peuvent pas détecter toutes les occurrences d'un comportement indéfini.

Examples

Déréférencer un pointeur nul

Ceci est un exemple de déréférencement d'un pointeur NULL, provoquant un comportement indéfini.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

Un standard `NULL` est garanti par le standard C pour se comparer à tout pointeur sur un objet valide, et le déréférencement appelle un comportement non défini.

Modifier un objet plus d'une fois entre deux points de séquence

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Un tel code conduit souvent à des spéculations sur la "valeur résultante" de `i` . Plutôt que de spécifier un résultat, les normes C spécifient que l'évaluation d'une telle expression produit *un comportement indéfini* . Avant C2011, la norme formalisait ces règles en termes de *points de séquence* :

> Entre le point de séquence précédent et suivant, un objet scalaire aura sa valeur stockée modifiée au plus une fois par l'évaluation d'une expression. En outre, la valeur antérieure doit être lue uniquement pour déterminer la valeur à stocker.

(Norme C99, section 6.5, paragraphe 2)

Ce schéma s'est révélé un peu trop grossier, ce qui a eu pour conséquence que certaines expressions présentant un comportement indéfini par rapport à C99 ne devraient pas le faire. C2011 conserve les points de séquence, mais introduit une approche plus nuancée de ce domaine basée sur le *séquençage* et une relation appelée "séquence avant":

> Si un effet secondaire sur un objet scalaire est non séquencé par rapport à un effet secondaire différent sur le même objet scalaire ou à un calcul de valeur utilisant la valeur du même objet scalaire, le comportement est indéfini. S'il existe plusieurs ordres

> autorisés des sous-expressions d'une expression, le comportement n'est pas défini si un effet secondaire non séquencé se produit dans l'un des classements.

(Norme C2011, section 6.5, paragraphe 2)

Les détails complets de la relation "séquencée avant" sont trop longs pour être décrits ici, mais ils complètent les points de séquence plutôt que de les supplanter. Ils ont donc pour effet de définir le comportement de certaines évaluations dont le comportement était auparavant indéfini. En particulier, s'il existe un point de séquence entre deux évaluations, celle qui précède le point de séquence est "séquencée avant" celle qui suit.

L'exemple suivant présente un comportement bien défini:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

L'exemple suivant a un comportement non défini:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-
operators */
```

Comme pour toute forme de comportement indéfini, l'observation du comportement réel d'évaluation des expressions qui ne respectent pas les règles de séquençage n'est pas informative, sauf dans un sens rétrospectif. La norme linguistique ne permet pas d’attendre que de telles observations soient prédictives, même en ce qui concerne le comportement futur du même programme.

Déclaration de retour manquante dans la fonction de retour de valeur

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* Trying to use the (not) returned value causes UB */
  int value = foo();
  return 0;
}
```

Lorsqu'une fonction est déclarée pour renvoyer une valeur, elle doit le faire sur tous les chemins de code possibles. Un comportement indéfini se produit dès que l'appelant (qui attend une valeur de retour) tente d'utiliser la valeur de retour [1] .

Notez que le comportement non défini ne se produit *que si* l'appelant tente d'utiliser / d'accéder à la valeur de la fonction. Par exemple,

```
int foo(void) {
  /* do stuff */
```

```
  /* no return here */
}

int main(void) {
  /* The value (not) returned from foo() is unused. So, this program
   * doesn't cause *undefined behaviour*. */
  foo();
  return 0;
}
```

C99

La fonction `main()` est une exception à cette règle dans la mesure où il est possible de la terminer sans instruction return car une valeur de retour supposée de `0` sera automatiquement utilisée dans ce cas [2] .

---

[1] (*ISO / IEC 9899: 201x* , 6.9.1 / 12)

> Si le} qui termine une fonction est atteint et que la valeur de l'appel de fonction est utilisée par l'appelant, le comportement est indéfini.

[2] (*ISO / IEC 9899: 201x* , 5.1.2.2.3 / 1)

> atteindre le} qui termine la fonction principale renvoie une valeur de 0.

Débordement d'entier signé

Selon le paragraphe 6.5 / 5 de C99 et C11, l'évaluation d'une expression produit un comportement indéfini si le résultat n'est pas une valeur représentable du type de l'expression. Pour les types arithmétiques, cela s'appelle un *débordement* . L'arithmétique entière non signée ne déborde pas car le paragraphe 6.2.5 / 9 s'applique, ce qui réduit le résultat non signé qui serait autrement hors de portée. Il n'y a pas de disposition analogue pour les types entiers *signés* , cependant; celles-ci peuvent déborder et produisent un comportement indéfini. Par exemple,

```
#include <limits.h>      /* to get INT_MAX */

int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

La plupart des instances de ce type de comportement indéfini sont plus difficiles à reconnaître ou à prédire. Le débordement peut en principe provenir de toute opération d'addition, de soustraction ou de multiplication sur des entiers signés (soumis aux conversions arithmétiques habituelles) où il n'y a pas de limites ou de relations efficaces entre les opérandes pour l'empêcher. Par exemple, cette fonction:

```
int square(int x) {
    return x * x;  /* overflows for some values of x */
}
```

est raisonnable, et il fait ce qu'il faut pour des valeurs d'argument assez petites, mais son comportement n'est pas défini pour des valeurs d'argument plus importantes. Vous ne pouvez pas juger de la seule fonction si les programmes qui l’appellent ont un comportement indéfini. Cela dépend des arguments qu'ils lui transmettent.

D'autre part, considérez cet exemple trivial d'arithmétique d'entiers signés à sécurité de débordement:

```
int zero(int x) {
    return x - x;  /* Cannot overflow */
}
```

La relation entre les opérandes de l'opérateur de soustraction assure que la soustraction ne déborde jamais. Ou considérez cet exemple un peu plus pratique:

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++;  /* might overflow */
    while (fgetc(f2) != EOF) count2++;  /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

Tant que les compteurs ne débordent pas individuellement, les opérandes de la soustraction finale seront tous deux non négatifs. Toutes les différences entre deux de ces valeurs sont représentables sous la forme `int` .

Utilisation d'une variable non initialisée

```
int a;
printf("%d", a);
```

La variable `a` est un `int` avec une durée de stockage automatique. L'exemple de code ci-dessus tente d'imprimer la valeur d'une variable non initialisée (`a` n'a jamais été initialisé). Les variables automatiques qui ne sont pas initialisées ont des valeurs indéterminées; leur accès peut entraîner un comportement indéfini.

Remarque: Les variables avec stockage statique ou thread local, y compris les variables globales sans le mot-clé `static` , sont initialisées à zéro ou à leur valeur initialisée. D'où ce qui suit est légal.

```
static int b;
printf("%d", b);
```

Une erreur très courante consiste à ne pas initialiser les variables qui servent de compteurs à 0. Vous leur ajoutez des valeurs, mais comme la valeur initiale est garbage, vous appellerez **Undefined Behavior** , comme dans la question Compilation on terminal et Avertissement. symboles étranges

Exemple:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Sortie:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main
main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]
        counter += i;
        ^~~~~~~
main.c:4:19: note: initialize the variable 'counter' to silence this warning
    int i, counter;
                  ^
                   = 0
1 warning generated.
C02QT2UBFVH6-lm:~ gsamaras$ ./main
32812
```

Les règles ci-dessus s'appliquent également aux pointeurs. Par exemple, les résultats suivants entraînent un comportement indéfini

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Notez que le code ci-dessus en lui-même peut ne pas provoquer une erreur ou une erreur de segmentation, mais essayer de déréférencer ce pointeur ultérieurement entraînerait un comportement indéfini.

Déréférencer un pointeur à variable au-delà de sa durée de vie

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */

int main (void)
{
    int* p;

    p = foo(5);  /* (2) this expression's behavior is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */
```

```
    return 0;
}
```

Certains compilateurs le soulignent utilement. Par exemple, `gcc` avertit avec:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

et `clang` avertit avec:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

pour le code ci-dessus. Mais les compilateurs peuvent ne pas être en mesure d'aider en code complexe.

(1) Le renvoi de la référence à la variable déclarée `static` est défini comme comportement, car la variable n'est pas détruite après avoir quitté l'étendue actuelle.

(2) Conformément à la norme ISO / IEC 9899: 2011 6.2.4 §2, "La valeur d'un pointeur devient indéterminée lorsque l'objet sur lequel il pointe atteint la fin de sa durée de vie."

(3) Déréférencer le pointeur renvoyé par la fonction `foo` est un comportement indéfini car la mémoire à laquelle il fait référence contient une valeur indéterminée.

Division par zéro

```
int x = 0;
int y = 5 / x;  /* integer division */
```

ou

```
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

ou

```
int x = 0;
int y = 5 % x;  /* modulo operation */
```

Pour la deuxième ligne de chaque exemple, où la valeur du deuxième opérande (x) est zéro, le comportement est indéfini.

Notez que la plupart des implémentations de calcul à virgule flottante suivront une norme (par exemple, IEEE 754), auquel cas les opérations comme diviser par zéro auront des résultats cohérents (par exemple, `INFINITY`) même si la norme C indique que l'opération n'est pas définie.

Accéder à la mémoire au-delà du bloc attribué

Un pointeur sur un morceau de mémoire contenant `n` éléments ne peut être déréférencé que s'il se trouve dans la `memory` de `memory` et dans la `memory + (n - 1)` . Déréférencer un pointeur en dehors de cette plage entraîne un comportement indéfini. À titre d'exemple, considérons le code suivant:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

La troisième ligne accède au 4ème élément d'un tableau de 3 éléments seulement, ce qui conduit à un comportement indéfini. De même, le comportement de la deuxième ligne du fragment de code suivant est également mal défini:

```
int array[3];
array[3] = 0;
```

Notez que le fait de pointer devant le dernier élément d'un tableau n'est pas un comportement indéfini (`beyond_array = array + 3` est bien défini ici), mais le déréférencement est (`*beyond_array` est un comportement indéfini). Cette règle s'applique également à la mémoire allouée dynamiquement (comme les tampons créés via `malloc`).

Copie de mémoire superposée

Une grande variété de fonctions de bibliothèque standard ont parmi leurs effets la copie de séquences d'octets d'une région mémoire à une autre. La plupart de ces fonctions ont un comportement indéfini lorsque les régions source et de destination se chevauchent.

Par exemple, ceci ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... tente de copier 10 octets lorsque les zones de mémoire source et de destination se chevauchent de trois octets. Pour visualiser:

```
               overlapping area
               |
               _ _
              |   |
              v   v
T h i s   i s   a n   e x a m p l e \0
^             ^
|             |
|             destination
|
source
```

En raison du chevauchement, le comportement résultant est indéfini.

`memcpy()` , `strcpy()` , `strcat()` , `sprintf()` et `sscanf()` fonctions standard de la bibliothèque avec une limitation de ce type. La norme dit de ces fonctions et de plusieurs autres fonctions:

> Si la copie a lieu entre des objets qui se chevauchent, le comportement est indéfini.

La fonction `memmove()` est la principale exception à cette règle. Sa définition spécifie que la fonction se comporte comme si les données source étaient d'abord copiées dans un tampon temporaire, puis écrites dans l'adresse de destination. Il n'y a pas d'exception pour les régions source et de destination qui se chevauchent, pas plus que nécessaire, donc `memmove()` a un comportement bien défini dans de tels cas.

La distinction reflète une efficacité *vs.* compromis de généralité. La copie telle que ces fonctions se produit généralement entre des régions de mémoire disjointes, et il est souvent possible de savoir, au moment du développement, si une instance particulière de copie de mémoire sera dans cette catégorie. En supposant que le non-chevauchement offre des implémentations comparativement plus efficaces qui ne produisent pas de résultats corrects de manière fiable lorsque l'hypothèse ne tient pas. La plupart des fonctions de la bibliothèque C sont autorisées pour les implémentations les plus efficaces, et `memmove()` remplit les lacunes, servant les cas où la source et la destination peuvent ou se chevauchent. Pour produire l'effet correct dans tous les cas, cependant, il doit effectuer des tests supplémentaires et / ou utiliser une implémentation comparativement moins efficace.

Lecture d'un objet non initialisé qui n'est pas soutenu par la mémoire

C11

La lecture d'un objet provoquera un comportement indéfini si l'objet est [1] :

- non initialisé
- défini avec une durée de stockage automatique
- son adresse n'est jamais prise

La variable a dans l'exemple ci-dessous satisfait toutes ces conditions:

```
void Function( void )
{
    int a;
    int b = a;
}
```

---

[1] (Cité à partir de: ISO: CEI 9899: 201X 6.3.2.1 Lvalues, tableaux et indicateurs de fonction 2)
Si la lvalue désigne un objet de durée de stockage automatique qui aurait pu être déclaré avec la classe de stockage de registre (son adresse n'a jamais été prise) et cet objet n'est pas initialisé (non déclaré avec un initialiseur et aucune affectation n'a été effectuée avant son utilisation)), le comportement est indéfini.

Course de données

C11

C11 a introduit la prise en charge de plusieurs threads d'exécution, ce qui offre la possibilité de faire des courses de données. Un programme contient une course de données si un objet en y accède [1] par deux fils différents, où au moins l' un des accès est non-atomique, au moins on modifie l'objet, et la sémantique de programme ne parviennent pas à assurer que les deux accès ne peut pas chevaucher temporellement. [2] Notez bien que la concurrence réelle des accès impliqués n’est pas une condition pour une course de données; Les courses de données couvrent une classe plus large de problèmes liés aux incohérences (autorisées) dans les vues de la mémoire de différents threads.

Considérez cet exemple:

```
#include <threads.h>

int a = 0;

int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}
```

Les principaux appels fil `thrd_create` pour démarrer une nouvelle fonction de défilement de fil `Function` . Le deuxième thread modifie `a` , et le thread principal lit `a` . Aucun de ces accès n'est atomique, et les deux threads ne font rien individuellement ou conjointement pour s'assurer qu'ils ne se chevauchent pas, il y a donc une course aux données.

Parmi les façons dont ce programme pourrait éviter la course aux données:

- le thread principal pourrait effectuer sa lecture d' `a` avant de démarrer l'autre thread;
- le fil conducteur peut effectuer sa lecture d' `a` après s'être assuré via `thrd_join` que l'autre a pris fin;
- les threads peuvent synchroniser leurs accès via un mutex, chacun verrouillant ce mutex avant d'accéder à `a` et de le déverrouiller ensuite.

Comme le démontre l'option mutex, éviter une course de données ne nécessite pas de garantir un ordre spécifique d'opérations, tel que le thread enfant modifiant `a` avant que le thread principal ne le lise; il suffit (pour éviter une course de données) de s'assurer que pour une exécution donnée, un accès se produira avant l'autre.

---

[1] Modifier ou lire un objet.

[2] (Cité d'après l'ISO: CEI 9889: 201x, section 5.1.2.4 "Exécutions multithread et courses de données")
L'exécution d'un programme contient une course de données si elle contient deux actions en conflit dans des threads différents, dont au moins une n'est pas atomique et aucune ne se produit avant l'autre. Toute course de données de ce type entraîne un comportement indéfini.

Valeur de lecture du pointeur libéré

Même le simple fait de **lire** la valeur d'un pointeur qui a été libéré (c'est-à-dire sans essayer de déréférencer le pointeur) est un comportement non défini (UB), par exemple

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Citant **ISO / IEC 9899: 2011** , section 6.2.4 §2:

> […] La valeur d'un pointeur devient indéterminée lorsque l'objet vers lequel il pointe (ou juste passé) atteint la fin de sa vie.

L'utilisation d'une mémoire indéterminée pour n'importe quoi, y compris une comparaison ou une arithmétique apparemment sans danger, peut avoir un comportement indéfini si la valeur peut être une représentation de piège pour le type.

Modifier le littéral de chaîne

Dans cet exemple de code, le pointeur de caractère `p` est initialisé à l'adresse d'un littéral de chaîne. Essayer de modifier le littéral de chaîne a un comportement indéfini.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

Cependant, modifier un tableau mutable de `char` directement, ou par le biais d'un pointeur, n'est naturellement pas un comportement indéfini, même si son initialiseur est une chaîne littérale. Ce qui suit va bien:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

En effet, le littéral de chaîne est effectivement copié dans le tableau chaque fois que le tableau est initialisé (une fois pour les variables avec une durée statique, chaque fois que le tableau est créé pour les variables avec une durée automatique ou de thread). il est bon de modifier le contenu du tableau.

Libérer deux fois la mémoire

Libérer deux fois la mémoire est un comportement indéfini, par exemple

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Citation de la norme (7.20.3.2. La fonction libre de C99):

> Sinon, si l'argument ne correspond pas à un pointeur précédemment renvoyé par la fonction calloc, malloc ou realloc, ou si l'espace a été libéré par un appel à free ou realloc, le comportement n'est pas défini.

Utiliser un spécificateur de format incorrect dans printf

L'utilisation d'un spécificateur de format incorrect dans le premier argument de `printf` appelle un comportement indéfini. Par exemple, le code ci-dessous appelle un comportement indéfini:

```
long z = 'B';
printf("%c\n", z);
```

Voici un autre exemple

```
printf("%f\n",0);
```

La ligne de code ci-dessus est un comportement indéfini. `%f` attend le double. Cependant 0 est de type `int` .

Notez que votre compilateur peut généralement vous aider à éviter de tels cas si vous activez les indicateurs appropriés lors de la compilation (`-Wformat` in `clang` et `gcc`). Du dernier exemple:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
            ~~    ^
            %d
```

La conversion entre les types de pointeurs produit un résultat incorrectement aligné

Le comportement suivant *peut* être indéfini en raison d'un alignement de pointeur incorrect:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

Le comportement indéfini se produit lorsque le pointeur est converti. Selon C11, si une *conversion*

entre deux types de pointeurs produit un résultat incorrectement aligné (6.3.2.3), le comportement est indéfini . Ici, un `uint32_t` pourrait nécessiter un alignement de 2 ou 4 par exemple.

`calloc` d'autre part est requis pour retourner un pointeur qui est convenablement aligné pour tout type d'objet; Ainsi, `memory_block` est correctement aligné pour contenir un `uint32_t` dans sa partie initiale. Ensuite, sur un système où `uint32_t` a besoin d'un alignement de 2 ou 4, `memory_block + 1` sera une adresse *impaire* et donc mal alignée.

Observez que le standard C demande que l'opération de conversion soit déjà indéfinie. Ceci est imposé parce que sur les plates-formes où les adresses sont segmentées, l'adresse d'octet `memory_block + 1` peut même ne pas avoir une représentation correcte en tant que pointeur entier.

Le fait de convertir `char *` en pointeurs vers d'autres types sans se soucier des exigences d'alignement est parfois utilisé de manière incorrecte pour le décodage de structures empaquetées telles que les en-têtes de fichiers ou les paquets réseau.

Vous pouvez éviter le comportement indéfini résultant d'une conversion de pointeur mal alignée à l'aide de `memcpy` :

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Ici, aucune conversion de pointeur en `uint32_t*` n'a lieu et les octets sont copiés un par un.

Cette opération de copie pour notre exemple ne conduit qu'à une valeur valide de `mvalue` car:

- Nous avons utilisé `calloc` , donc les octets sont correctement initialisés. Dans notre cas, tous les octets ont la valeur `0` , mais toute autre initialisation correcte le ferait.
- `uint32_t` est un type de largeur exacte et n'a pas de bits de remplissage
- Tout modèle de bit arbitraire est une représentation valide pour tout type non signé.

Ajout ou soustraction de pointeur non borné correctement

Le code suivant a un comportement indéfini:

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1;  /* undefined behavior */
char *ptr2 = buffer + 5;  /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char *ptr4 = buffer + 7;  /* undefined behavior */
```

Selon C11, si l'addition ou la soustraction d'un pointeur dans, ou juste au-delà, un objet tableau et un type entier produit un résultat qui ne pointe pas vers le même objet tableau ou juste au-delà, le comportement n'est pas défini (6.5.6).

De plus, il est naturellement indéfini de *déréférencer* un pointeur qui pointe juste au-delà du tableau:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Modification d'une variable const à l'aide d'un pointeur

```
int main (void)
{
    const int foo_readonly = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_readonly; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Citant *ISO / IEC 9899: 201x* , section 6.7.3 §2:

> Si vous tentez de modifier un objet défini avec un type qualifié en utilisant une lvalue avec un type non-qualifié, le comportement est indéfini. [...]

---

(1) Dans GCC, cela peut générer l'avertissement suivant: `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]`

Passer un pointeur nul à la conversion de printf% s

La conversion `%s` de `printf` indique que l'argument correspondant est *un pointeur sur l'élément initial d'un tableau de type caractère* . Un pointeur nul ne pointe pas vers l'élément initial d'un tableau de type caractère, et le comportement des éléments suivants n'est donc pas défini:

```
char *foo = NULL;
printf("%s", foo); /* undefined behavior */
```

Cependant, le comportement indéfini ne signifie pas toujours que le programme se bloque - certains systèmes prennent des mesures pour éviter le plantage qui se produit normalement lorsqu'un pointeur nul est déréférencé. Par exemple, Glibc est connu pour imprimer

```
(null)
```

pour le code ci-dessus. Cependant, ajoutez (juste) une nouvelle ligne à la chaîne de format et vous obtiendrez un plantage:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

Dans ce cas, cela se produit car GCC a une optimisation qui transforme `printf("%s\n", argument);` dans un appel à `puts` avec `puts(argument)` , et `puts` en glibc ne gère pas des pointeurs nuls. Tout ce comportement est conforme à la norme.

Notez que le *pointeur null* est différent d'une *chaîne vide* . Donc, ce qui suit est valide et n'a pas de comportement indéfini. Il ne vous *reste plus* qu'à imprimer une *nouvelle ligne* :

```
char *foo = "";
printf("%s\n", foo);
```

Liaison incohérente d'identificateurs

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 dit:

> Si, au sein d'une unité de traduction, le même identifiant apparaît avec une liaison interne et externe, le comportement est indéfini.

Notez que si une déclaration préalable d'un identifiant est visible, elle comportera le lien de la déclaration précédente. *C11, §6.2.2, 4 le* permet:

> Pour un identi fi cateur déclaré avec le spéci fi cateur extern de classe de stockage dans une portée dans laquelle une déclaration préalable de cet identifiant est visible, 31) si la déclaration antérieure spéci fi e un lien interne ou externe, la liaison de l'identi fi cateur à la déclaration ultérieure est la même que le lien spécifié à la déclaration préalable. Si aucune déclaration préalable n'est visible ou si la déclaration préalable ne spécifie aucun lien, alors l'identi fi cateur a un lien externe.

```
/* 1. This is NOT undefined */
static int var;
extern int var;


/* 2. This is NOT undefined */
static int var;
static int var;

/* 3. This is NOT undefined */
extern int var;
extern int var;
```

Utiliser fflush sur un flux d'entrée

Les normes POSIX et C indiquent explicitement que l'utilisation de `fflush` sur un flux d'entrée est un comportement non défini. La `fflush` est définie uniquement pour les flux de sortie.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);
```

```
    return 0;
}
```

Il n'y a pas de méthode standard pour éliminer les caractères non lus d'un flux d'entrée. D'autre part, certaines implémentations utilisent `fflush` pour effacer le tampon `stdin` . Microsoft définit le comportement de `fflush` sur un flux d'entrée: Si le flux est ouvert pour l'entrée, `fflush` efface le contenu du tampon. Selon POSIX.1-2008, le comportement de `fflush` est indéfini, à moins que le fichier d'entrée ne soit accessible.

Voir Utilisation de `fflush(stdin)` pour plus de détails.

Déplacement de bits en utilisant des nombres négatifs ou au-delà de la largeur du type

Si la valeur du *compte à* rebours est une **valeur négative,** les opérations de *décalage* à *gauche* et à *droite* sont indéfinies [1] :

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

Si *le décalage à gauche* est effectué sur une **valeur négative** , il n'est pas défini:

```
int x = -5 << 3; /* undefined */
```

Si *le décalage à gauche* est effectué sur une **valeur positive** et que le résultat de la valeur mathématique n'est **pas** représentable dans le type, il n'est pas défini [1] :

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
 * in an int. So, this is undefined. */

int x = 5 << 72;
```

Notez que *le décalage vers la droite* sur une **valeur négative** (.eg `-5 >> 3`) n'est *pas* indéfini mais défini par la *mise en œuvre* .

---

[1] Citant *ISO / IEC 9899: 201x* , section 6.5.7:

> Si la valeur de l'opérande de droite est négative ou est supérieure ou égale à la largeur de l'opérande gauche promu, le comportement n'est pas défini.

Modification de la chaîne renvoyée par les fonctions getenv, strerror et setlocale

La modification des chaînes renvoyées par les fonctions standard `getenv()` , `strerror()` et `setlocale()` est indéfinie. Ainsi, les implémentations peuvent utiliser un stockage statique pour ces chaînes.

La fonction getenv (), C11, § 7.22.4.7, 4 dit:

> La fonction getenv renvoie un pointeur sur une chaîne associée au membre de liste correspondant. La chaîne pointée ne doit pas être modifiée par le programme, mais peut être écrasée par un appel ultérieur à la fonction getenv.

La fonction strerror (), C11, §7.23.6.3, 4 dit:

> La fonction strerror renvoie un pointeur sur la chaîne dont le contenu est localpeci fi c. Le tableau désigné ne doit pas être modifié par le programme, mais peut être remplacé par un appel ultérieur à la fonction strerror.

La fonction setlocale (), C11, §7.11.1.1, 8 dit:

> Le pointeur sur la chaîne renvoyé par la fonction setlocale est tel qu'un appel ultérieur avec cette valeur de chaîne et sa catégorie associée restaurera cette partie des paramètres régionaux du programme. La chaîne pointée ne doit pas être modifiée par le programme, mais peut être écrasée par un appel ultérieur à la fonction setlocale.

De même, la `localeconv()` renvoie un pointeur sur `struct lconv` qui ne doit pas être modifié.

La fonction localeconv (), C11, §7.11.2.1, 8 dit:

> La fonction localeconv renvoie un pointeur sur l'objet rempli. La structure pointée par la valeur de retour ne doit pas être modifiée par le programme, mais peut être écrasée par un appel ultérieur à la fonction localeconv.

Retour d'une fonction déclarée avec le spécificateur de fonction `_Noreturn` ou` noreturn`

C11

Le spécificateur de fonction `_Noreturn` été introduit dans C11. L'en-tête `<stdnoreturn.h>` fournit une macro `noreturn` qui se développe en `_Noreturn` . L'utilisation de `_Noreturn` ou `noreturn` partir de `<stdnoreturn.h>` est donc `<stdnoreturn.h>` et équivalente.

Une fonction déclarée avec `_Noreturn` (ou `noreturn`) n'est pas autorisée à retourner à son appelant. Si une telle fonction *ne* retourne à l'appelant, le comportement est indéfini.

Dans l'exemple suivant, `func()` est déclaré avec le spécificateur `noreturn` mais il renvoie à son appelant.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()...\n");
```

```
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

`gcc` et `clang` produisent des avertissements pour le programme ci-dessus:

```
$ gcc test.c
test.c: In function ‘func’:
test.c:9:1: warning: ‘noreturn’ function does return
 }
 ^
$ clang test.c
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

Un exemple d'utilisation de `noreturn` qui a un comportement bien défini:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void my_exit(void);

/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\n");
    exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

Lire Comportement non défini en ligne: https://riptutorial.com/fr/c/topic/364/comportement-non-defini

Chapitre 16: Contraintes

Remarques

Les contraintes sont un terme utilisé dans toutes les spécifications C existantes (récemment ISO-IEC 9899-2011). Ils sont l'une des trois parties du langage décrites à l'article 6 de la norme (avec la syntaxe et la sémantique).

ISO-IEC 9899-2011 définit une contrainte comme étant:

> restriction, syntaxique ou sémantique, par laquelle l'exposition des éléments du langage doit être interprétée

(Veuillez également noter qu'en termes de norme C, une "contrainte d'exécution" n'est pas une sorte de contrainte et comporte des règles très différentes.)

En d'autres termes, une contrainte décrit une règle du langage qui rendrait illégal un programme autrement syntaxiquement valide. À cet égard, les contraintes ressemblent quelque peu à un comportement non défini, tout programme qui ne les suit pas n'est pas défini en termes de langage C.

Les contraintes ont en revanche une différence très significative par rapport aux comportements non définis. Une implémentation est nécessaire pour fournir un message de diagnostic pendant la phase de traduction (partie de la compilation) si une contrainte est violée, ce message peut être un avertissement ou peut arrêter la compilation.

Examples

Noms de variables en double dans la même portée

Un exemple de contrainte tel qu'exprimé dans le standard C est d'avoir deux variables du même nom déclarées dans une portée [1)], par exemple:

```
void foo(int bar)
{
    int var;
    double var;
}
```

Ce code viole la contrainte et doit produire un message de diagnostic à la compilation. Ceci est très utile par rapport à un comportement indéfini, car le développeur sera informé du problème avant l'exécution du programme, ce qui risque de faire n'importe quoi.

Les contraintes ont donc tendance à être des erreurs facilement détectables au moment de la compilation, des problèmes qui entraînent un comportement indéfini mais qui seraient difficiles ou impossibles à détecter au moment de la compilation ne sont donc pas des contraintes.

---

1) libellé exact:

C99

Si un identificateur n'a pas de lien, il ne doit pas y avoir plus d'une déclaration de l'identificateur (dans un déclarant ou un spécificateur de type) de même portée et dans le même espace de nom, sauf pour les étiquettes spécifiées au 6.7.2.3.

Opérateurs arithmétiques unaires

Les opérateurs unary `+` et `-` ne sont utilisables que sur les types arithmétiques, donc si, par exemple, on essaie de les utiliser sur une structure, le programme produira un diagnostic, par exemple:

```
struct foo
{
    bool bar;
};

void baz(void)
{
   struct foo testStruct;
   -testStruct; /* This breaks the constraint so must produce a diagnostic */
}
```

Lire Contraintes en ligne: https://riptutorial.com/fr/c/topic/7397/contraintes

Chapitre 17: Conversions implicites et explicites

Syntaxe

- Conversion explicite (aka "Casting"): expression (type)

Remarques

La " *conversion explicite* " est aussi communément appelée "coulée".

Examples

Conversions entières dans les appels de fonction

Étant donné que la fonction a un prototype approprié, les entiers sont élargis pour les appels aux fonctions selon les règles de la conversion des nombres entiers, C11 6.3.1.3.

> **6.3.1.3 Entiers signés et non signés**
> Lorsqu'une valeur de type entier est convertie en un autre type entier autre que _Bool, si la valeur peut être représentée par le nouveau type, elle reste inchangée.
>
> Sinon, si le nouveau type n'est pas signé, la valeur est convertie en ajoutant ou en soustrayant de manière répétée un de plus que la valeur maximale pouvant être représentée dans le nouveau type jusqu'à ce que la valeur soit dans la plage du nouveau type.
>
> Sinon, le nouveau type est signé et la valeur ne peut pas y être représentée. soit le résultat est défini par l'implémentation ou un signal défini par l'implémentation est généré.

En règle générale, vous ne devez pas tronquer un type signé large à un type signé plus étroit, car les valeurs ne peuvent évidemment pas s'inscrire et il n'ya pas de signification claire. La norme C citée ci-dessus définit ces cas comme étant "définis par l'implémentation", c'est-à-dire qu'ils ne sont pas portables.

L'exemple suivant suppose que `int` une largeur de 32 bits.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u16(uint16_t val) {
```

```
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val);  /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t  u8  = 127;
    uint8_t  s64  = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8);   /* param_u8 val is 127 */
    param_u16(u8);  /* param_u16 val is 127 */
    param_u32(u8);  /* param_u32 val is 127 */
    param_u64(u8);  /* param_u64 val is 127 */
    param_s8(u8);   /* param_s8 val is 127 */
    param_s16(u8);  /* param_s16 val is 127 */
    param_s32(u8);  /* param_s32 val is 127 */
    param_s64(u8);  /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64);  /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64);  /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}
```

Conversions du pointeur dans les appels de fonction

Les conversions de pointeur à `void*` sont implicites, mais toute autre conversion de pointeur doit

être explicite. Bien que le compilateur permette une conversion explicite de tout type de pointeur en donnée à tout autre type de pointeur vers données, l'accès à un objet via un pointeur mal typé est erroné et conduit à un comportement indéfini. Le seul cas autorisé est que les types soient compatibles ou si le pointeur avec lequel vous regardez l'objet est un type de caractère.

```
#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}

int main(void) {


    /* Implicit ptr conversion allowed for void* */
    func_voidp(&data_a);

    /*
     * Explicit ptr conversion for other types
     *
     * Note that here although the have identical definitions,
     * the types are not compatible, and that the this call is
     * erroneous and leads to undefined behavior on execution.
     */
    func_struct_b((struct struct_b*)&data_a);

    /* My output shows: */
    /* func_charp Address of ptr is 0x601030 */
    /* func_voidp Address of ptr is 0x601030 */
    /* func_struct_b Address of ptr is 0x601030 */

    return 0;
}
```

Lire Conversions implicites et explicites en ligne: https://riptutorial.com/fr/c/topic/2529/conversions-implicites-et-explicites

Chapitre 18: Cordes

Introduction

En C, une chaîne n'est pas un type intrinsèque. Une chaîne de caractères est la convention pour avoir un tableau de caractères à une dimension qui se termine par un caractère nul, par un `'\0'` .

Cela signifie qu'une chaîne de caractères avec un contenu de `"abc"` aura quatre caractères `'a'` , `'b'` , `'c'` et `'\0'` .

Voir l'exemple de base de l'introduction aux chaînes .

Syntaxe

- char str1 [] = "Bonjour, monde!"; / * Modifiable * /
- char str2 [14] = "Bonjour tout le monde!"; / * Modifiable * /
- char * str3 = "Bonjour tout le monde!"; /* Non modifiable*/

Examples

Calculez la longueur: strlen ()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\n", len);

    return EXIT_SUCCESS;
}
```

Ce programme calcule la longueur de son deuxième argument d'entrée et stocke le résultat dans `len` . Il imprime ensuite cette longueur au terminal. Par exemple, lorsqu'il est exécuté avec les paramètres `program_name "Hello, world!"` , le programme va sortir `The length of the second argument is 13.` car la chaine `Hello, world!` est composé de 13 caractères.

`strlen` compte tous les **octets** du début de la chaîne jusqu'à, mais sans inclure, le caractère NUL de terminaison, `'\0'` . En tant que tel, il ne peut être utilisé que lorsque la chaîne est *garantie* pour

être terminée par NUL.

Gardez également à l'esprit que si la chaîne contient des caractères Unicode, `strlen` ne vous indiquera pas le nombre de caractères de la chaîne (certains caractères pouvant compter plusieurs octets). Dans de tels cas, vous devez compter vous-même les caractères (*c.* -à-d. Les unités de code). Considérez la sortie de l'exemple suivant:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\n", sizeof(utf8String));
    printf("\"%s\" is %zu bytes\n", asciiString, strlen(asciiString));
    printf("\"%s\" is %zu bytes\n", utf8String, strlen(utf8String));
}
```

Sortie:

```
asciiString has 50 bytes in the array
utf8String has 50 bytes in the array
"Hello world!" is 12 bytes
"Γειά σου Κόσμε!" is 27 bytes
```

Copie et concaténation: strcpy (), strcat ()

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  /* Always ensure that your string is large enough to contain the characters
   * and a terminating NUL character ('\0')!
   */
  char mystring[10];

  /* Copy "foo" into `mystring`, until a NUL character is encountered. */
  strcpy(mystring, "foo");
  printf("%s\n", mystring);

  /* At this point, we used 4 chars of `mystring`, the 3 characters of "foo",
   * and the NUL terminating byte.
   */

  /* Append "bar" to `mystring`. */
  strcat(mystring, "bar");
  printf("%s\n", mystring);

  /* We now use 7 characters of `mystring`: "foo" requires 3, "bar" requires 3
   * and there is a terminating NUL character ('\0') at the end.
   */
```

```
  /* Copy "bar" into `mystring`, overwriting the former contents. */
  strcpy(mystring, "bar");
  printf("%s\n", mystring);

  return 0;
}
```

Les sorties:

```
foo
foobar
bar
```

Si vous ajoutez à, ou à partir ou copiez à partir d'une chaîne existante, assurez-vous qu'il est terminé NUL!

Les littéraux de chaîne (par exemple `"foo"`) seront toujours terminés par le compilateur.

Comparaison: strcmp (), strncmp (), strcasecmp (), strncasecmp ()

Les fonctions `strcase*` ne sont pas la norme C, mais une extension POSIX.

La fonction `strcmp` compare lexicographiquement deux tableaux de caractères à terminaison nulle. Les fonctions renvoient une valeur négative si le premier argument apparaît avant le second dans l'ordre lexicographique, zéro s'il est égal ou positif si le premier argument apparaît après le second dans l'ordre lexicographique.

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "BBB");
    compare("BBB", "CCCCC");
    compare("BBB", "AAAAAA");
    return 0;
}
```

Les sorties:

```
BBB equals BBB
```

```
BBB comes before CCCCC
BBB comes after AAAAAA
```

En tant que `strcmp` , la fonction `strcasecmp` compare également ses arguments lexicographiquement après avoir traduit chaque caractère en son correspondant en minuscule:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}
```

Les sorties:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` et `strncasecmp` comparent au plus n caractères:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
```

```
    return 0;
}
```

Les sorties:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Tokenisation: strtok (), strtok_r () et strtok_s ()

La fonction `strtok` décompose une chaîne en chaînes plus petites, ou jetons, en utilisant un ensemble de délimiteurs.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Sortie:

```
1: [Hello]
2: [world]
```

La chaîne de délimiteurs peut contenir un ou plusieurs délimiteurs et différentes chaînes de délimiteurs peuvent être utilisées avec chaque appel à `strtok` .

Les appels à `strtok` pour continuer à numériser la même chaîne source ne doivent pas transmettre la chaîne source à nouveau, mais plutôt passer `NULL` comme premier argument. Si la même chaîne source *est* transmise, le premier jeton sera ré-unifié. C'est-à-dire qu'avec les mêmes délimiteurs, `strtok` renverrait simplement le premier jeton.

Notez que comme `strtok` n'alloue pas de nouvelle mémoire pour les jetons, *il modifie la chaîne source* . C'est-à-dire que dans l'exemple ci-dessus, la chaîne `src` sera manipulée pour produire les jetons référencés par le pointeur renvoyé par les appels à `strtok` . Cela signifie que la chaîne source ne peut pas être `const` (elle ne peut donc pas être un littéral de chaîne). Cela signifie également que l'identité de l'octet de délimitation est perdue (c'est-à-dire que dans l'exemple, les "," et "!" Sont effectivement supprimés de la chaîne source et que vous ne pouvez pas déterminer le caractère de délimitation correspondant).

Notez également que plusieurs délimiteurs consécutifs dans la chaîne source sont traités comme un seul; dans l'exemple, la deuxième virgule est ignorée.

`strtok` n'est ni thread-safe ni ré-entrant car il utilise un tampon statique lors de l'analyse. Cela signifie que si une fonction appelle `strtok` , aucune fonction qu'elle appelle lorsqu'elle utilise `strtok` ne peut également utiliser `strtok` , et elle ne peut être appelée par aucune fonction utilisant `strtok` .

Un exemple qui démontre les problèmes causés par le fait que `strtok` n'est pas rentrant est le suivant:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Sortie:

```
[1.2]
 [1]
 [2]
```

L'opération attendue est que la boucle `do while` while externe doit créer trois jetons constitués de chaque chaîne de nombres décimaux (`"1.2"` , `"3.5"` , `"4.2"`), pour chacun desquels les appels `strtok` de la boucle interne doivent être séparés. chaînes de caractères numériques (`"1"` , `"2"` , `"3"` , `"5"` , `"4"` , `"2"`).

Cependant, comme `strtok` n'est pas ré-entrant, cela ne se produit pas. Au lieu de cela, le premier `strtok` crée correctement le jeton "1.2 \ 0" et la boucle interne crée correctement les jetons `"1"` et `"2"` . Mais alors le `strtok` dans la boucle externe est à la fin de la chaîne utilisée par la boucle interne et renvoie immédiatement NULL. Les deuxième et troisième sous-chaînes du tableau `src` ne sont pas analysées du tout.

C11

Les bibliothèques C standard ne contiennent pas de version thread-safe ou ré-entrante, mais d'autres le sont, comme POSIX ' `strtok_r` . Notez que sur `strtok_s` , l'équivalent `strtok` , `strtok_s` est `strtok_s` pour les threads.

C11

C11 possède une partie facultative, l'Annexe K, qui offre une version thread-safe et ré-entrante nommée `strtok_s` . Vous pouvez tester la fonctionnalité avec `__STDC_LIB_EXT1__` . Cette partie facultative n'est pas largement prise en charge.

La fonction `strtok_s` diffère de la fonction POSIX `strtok_r` de stocker en dehors de la chaîne en cours de jeton et en vérifiant les contraintes d'exécution. Sur les programmes correctement écrits, les `strtok_s` et `strtok_r` se comportent de la même manière.

L'utilisation de `strtok_s` avec l'exemple donne maintenant la réponse correcte, comme ceci:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif

char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);
```

Et le résultat sera:

```
[1.2]
 [1]
 [2]
[3.5]
 [3]
 [5]
[4.2]
 [4]
 [2]
```

Rechercher la première / dernière occurrence d'un caractère spécifique: strchr (), strrchr ()

Les fonctions `strchr` et `strrchr` trouvent un caractère dans une chaîne, c'est-à-dire dans un tableau de caractères terminé par NUL. `strchr` renvoie un pointeur sur la première occurrence et `strrchr` sur la dernière.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
              toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                                     is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
              toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }

    return EXIT_SUCCESS;
}
```

Sorties (après avoir généré un exécutable nommé `pos`):

```
$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbbAccccAAAAzzz
First position of A in BAbbbbbAccccAAAAzzz is 1.
Last position of A in BAbbbbbAccccAAAAzzz is 15.
$  ./pos qwerty
A is not in qwerty.
```

Une utilisation courante de `strrchr` consiste à extraire un nom de fichier d'un chemin. Par exemple, pour extraire `myfile.txt` de `C:\Users\eak\myfile.txt` :

```
char *getFileName(const char *path)
{
    char *pend;
```

```
    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Itération sur les caractères d'une chaîne

Si nous connaissons la longueur de la chaîne, nous pouvons utiliser une boucle for pour parcourir ses caractères:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Alternativement, nous pouvons utiliser la fonction standard `strlen()` pour obtenir la longueur d'une chaîne si nous ne savons pas quelle est la chaîne:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Enfin, nous pouvons tirer parti du fait que les chaînes de caractères de C sont garanties sans aucune terminaison (ce que nous avons déjà fait lors du passage à `strlen()` dans l'exemple précédent ;-)). Nous pouvons parcourir le tableau quelle que soit sa taille et cesser d'itérer une fois que nous avons atteint un caractère nul:

```
size_t i = 0;
while (string[i] != '\0') {       /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]);    /* Print each character of the string. */
    i++;
}
```

Introduction de base aux chaînes

En C, une **chaîne** est une suite de caractères terminée par un caractère nul ('\ 0').

Nous pouvons créer des chaînes à l'aide de **chaînes littérales** , qui sont des séquences de caractères entourées de guillemets doubles. par exemple, prenez la chaîne littérale `"hello world"` . Les littéraux de chaîne sont automatiquement annulés.

Nous pouvons créer des chaînes en utilisant plusieurs méthodes. Par exemple, nous pouvons déclarer un caractère `char *` et l'initialiser pour désigner le premier caractère d'une chaîne:

```
char * string = "hello world";
```

Lors de l'initialisation d'un caractère `char *` à une chaîne constante comme ci-dessus, la chaîne elle-même est généralement allouée en lecture seule; `string` est un pointeur sur le premier élément du tableau, qui est le caractère `'h'` .

Comme le littéral de chaîne est alloué en mémoire morte, il est non modifiable [1] . Toute tentative de modification entraînera un comportement indéfini , il est donc préférable d'ajouter `const` pour obtenir une erreur de compilation comme celle-ci

```
char const * string = "hello world";
```

Il a un effet similaire à [2]

```
char const string_arr[] = "hello world";
```

Pour créer une chaîne modifiable, vous pouvez déclarer un tableau de caractères et initialiser son contenu à l'aide d'un littéral de chaîne, comme ceci:

```
char modifiable_string[] = "hello world";
```

Ceci est équivalent à ce qui suit:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Comme la deuxième version utilise un initialiseur avec accolades, la chaîne n'est pas automatiquement terminée par un caractère null à moins qu'un caractère `'\0'` soit explicitement inclus dans le tableau de caractères, généralement comme son dernier élément.

---

1 Non modifiable implique que les caractères dans le littéral de chaîne ne peuvent pas être modifiés, mais rappelez-vous que la `string` pointeur peut être modifiée (peut pointer ailleurs ou peut être incrémentée ou décrémentée).

2 Les deux chaînes ont un effet similaire dans le sens où les caractères des deux chaînes ne peuvent pas être modifiés. Il convient de noter que `string` est un pointeur sur `char` et qu'il s'agit d'une valeur l modifiable, de sorte qu'il peut être incrémenté ou pointer vers un autre emplacement alors que le tableau `string_arr` est une valeur l non modifiable, il ne peut pas être modifié.

Création de tableaux de chaînes

Un tableau de chaînes peut signifier plusieurs choses:

1. Un tableau dont les éléments sont des `char *` s
2. Un tableau dont les éléments sont des réseaux de `char` s

Nous pouvons créer un tableau de pointeurs de caractères comme suit:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Rappelez-vous: lorsque nous affectons des littéraux de chaîne à `char *` , les chaînes elles-mêmes sont allouées en mémoire morte. Cependant, le tableau `string_array` est alloué en lecture / écriture. Cela signifie que nous pouvons modifier les pointeurs dans le tableau, mais nous ne pouvons pas modifier les chaînes vers lesquelles ils pointent.

Dans C, le paramètre principal `argv` (le tableau des arguments de ligne de commande transmis lors de l'exécution du programme) est un tableau de caractères `char *` : `char * argv[]` .

Nous pouvons également créer des tableaux de tableaux de caractères. Comme les chaînes sont des tableaux de caractères, un tableau de chaînes est simplement un tableau dont les éléments sont des tableaux de caractères:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

Ceci est équivalent à:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Notez que nous spécifions `4` comme taille de la deuxième dimension du tableau; chacune des chaînes de notre tableau est en réalité de 4 octets, car nous devons inclure le caractère de terminaison nulle.

strstr

```
/* finds the next instance of needle in haystack
   zbpos: the zero-based position to begin searching from
   haystack: the string to search in
   needle: the string that must be found
   returns the next match of `needle` in `haystack`, or -1 if not found
*/
int findnext(int zbpos, const char *haystack, const char *needle)
{
    char *p;

    if (((p = strstr(haystack + zbpos, needle)) != NULL)
        return p - haystack;

    return -1;
}
```

`strstr` recherche l'argument `haystack` (first) de la chaîne pointée par `needle` . Si trouvé, `strstr` renvoie l'adresse de l'occurrence. S'il n'a pas pu trouver d' `needle` , il retourne NULL. Nous utilisons `zbpos` afin de ne pas retrouver la même aiguille encore et encore. Pour sauter la première instance, nous ajoutons un décalage de `zbpos` . Un clone de Notepad pourrait appeler `findnext`

comme ceci, afin de mettre en œuvre son dialogue "Find Next":

```
/*
    Called when the user clicks "Find Next"
    doc: The text of the document to search
    findwhat: The string to find
*/
void onfindnext(const char *doc, const char *findwhat)
{
    static int i;

    if ((i = findnext(i, doc, findwhat)) != -1)
        /* select the text starting from i and ending at i + strlen(findwhat) */
    else
        /* display a message box saying "end of search" */
}
```

Littéraux de chaîne

Les chaînes littérales représentent zéro terminal, statique durée des tableaux de `char` . Comme ils ont une durée de stockage statique, un littéral de chaîne ou un pointeur vers le même tableau sous-jacent peut être utilisé de différentes manières, ce que ne permet pas un pointeur vers un tableau automatique. Par exemple, le retour d'un littéral de chaîne à partir d'une fonction a un comportement bien défini:

```
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

Pour des raisons historiques, les éléments du tableau correspondant à un littéral de chaîne ne sont pas formellement `const` . Néanmoins, toute tentative de les modifier a un comportement indéfini . En règle générale, un programme qui tente de modifier le tableau correspondant à un littéral de chaîne se bloque ou présente un dysfonctionnement.

```
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Lorsqu'un pointeur pointe vers une chaîne littérale - ou où il peut parfois faire - il est conseillé de déclarer référent ce pointeur `const` pour éviter d' engager un tel comportement non défini accidentellement.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

D'un autre côté, un pointeur vers ou dans le tableau sous-jacent d'un littéral de chaîne n'est pas en soi intrinsèquement spécial; sa valeur peut être librement modifiée pour indiquer autre chose:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

En outre, bien que initialisations pour `char` tableaux peuvent avoir la même forme que les littéraux

de chaîne, l' utilisation d'un tel initialiseur ne confère pas les caractéristiques d'une chaîne littérale sur le tableau initialisé. L'initialiseur désigne simplement la longueur et le contenu initial du tableau. En particulier, les éléments sont modifiables s'ils ne sont pas explicitement déclarés `const` :

```
char foo[] = "hello";
foo[0] = 'y';  /* OK! */
```

Remettre une chaîne à zéro

Vous pouvez appeler `memset` pour mettre à zéro une chaîne (ou tout autre bloc de mémoire).

Où `str` est la chaîne à mettre à zéro et `n` est le nombre d'octets de la chaîne.

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


int main(void)
{
  char str[42] = "fortytwo";
  size_t n = sizeof str; /* Take the size not the length. */

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Impressions:

```
'fortytwo'
''
```

Un autre exemple:

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


#define FORTY_STR "forty"
#define TWO_STR "two"

int main(void)
{
  char str[42] = FORTY_STR TWO_STR;
  size_t n = sizeof str; /* Take the size not the length. */
  char * point_to_two = strstr(str, TWO_STR);

```

```
  printf("'%s'\n", str);

  memset(point_to_two, '\0', n);

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Impressions:

```
'fortytwo'
'forty'
''
```

strspn et strcspn

Etant donné une chaîne, `strspn` calcule la longueur de la sous-chaîne initiale (span) constituée uniquement d'une liste de caractères spécifique. `strcspn` est similaire, sauf qu'il calcule la longueur de la sous-chaîne initiale composée de tous les caractères sauf ceux listés:

```
/*
  Provided a string of "tokens" delimited by "separators", print the tokens along
  with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.,";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);

        if (n > 0)
            printf("token found: << %.*s >> (length=%d)\n", n, s, n);

        /* Skip the token now. */
        s += n;
```

```
    }

    printf("== token list exhausted ==\n");

    return 0;
}
```

Les fonctions analogues utilisant des chaînes de caractères larges sont `wcsspn` et `wcscspn` ; ils sont utilisés de la même manière.

Copier des chaînes

Les affectations de pointeur ne copient pas les chaînes

Vous pouvez utiliser le `=` opérateur pour copier des entiers, mais vous ne pouvez pas utiliser le `=` opérateur de copier des chaînes en C. Les chaînes en C sont représentés sous forme de tableaux de caractères avec un nul caractère de fin, donc en utilisant le `=` opérateur enregistre uniquement l'adresse (pointeur) d'une chaîne.

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
    there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

L'exemple ci-dessus a été compilé car nous avons utilisé `char *d` plutôt que `char d[3]` . L'utilisation de ce dernier provoquerait une erreur de compilation. Vous ne pouvez pas affecter de tableaux en C.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];
```

```
    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Copie de chaînes à l'aide de fonctions standard

`strcpy()`

Pour copier des chaînes, la fonction `strcpy()` est disponible dans `string.h` . Un espace suffisant doit être attribué à la destination avant la copie.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}
```

C99

`snprintf()`

Pour éviter le dépassement de la mémoire tampon, `snprintf()` peut être utilisé. Ce n’est pas la meilleure solution en termes de performances car elle doit analyser la chaîne de template, mais c’est la seule fonction de limitation de tampon pour la copie de chaînes facilement disponibles dans la bibliothèque standard, qui peut être utilisée sans étapes supplémentaires.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here!
*/
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */
```

```
    return 0;
}
```

`strncat()`

Une deuxième option, avec de meilleures performances, consiste à utiliser `strncat()` (une version de vérification de dépassement de tampon de `strcat()`) - elle prend un troisième argument indiquant le nombre maximal d’octets à copier:

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
    /* copies up to the first (sizeof(dest) - 1) elements of source into dest,
    then puts a \0 on the end of dest */
```

Notez que cette formulation utilise `sizeof(dest) - 1` ; Ceci est crucial car `strncat()` ajoute toujours un octet nul (bon), mais ne le compte pas dans la taille de la chaîne (cause de confusion et écrasement du tampon).

Notez également que l'alternative - concaténer après une chaîne non vide - est encore plus compliquée. Considérer:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

La sortie est la suivante:

```
23: [Clownfish: Marvin and N]
```

Notez, cependant, que la taille spécifiée comme la longueur n'était *pas* la taille du tableau de destination, mais la quantité d'espace qui lui restait, sans compter l'octet null du terminal. Cela peut causer de gros problèmes de réécriture. C'est aussi un peu de gaspillage; Pour spécifier correctement l'argument de longueur, vous connaissez la longueur des données dans la destination. Vous pouvez donc spécifier l'adresse de l'octet nul à la fin du contenu existant, en évitant que `strncat()` :

```
    strcpy(dst, "Clownfish: ");
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

Cela produit le même résultat que précédemment, mais `strncat()` n'a pas besoin de `strncat()` le contenu existant de `dst` avant de commencer à copier.

`strncpy()`

La dernière option est la fonction `strncpy()` . Bien que vous puissiez penser que cela devrait venir en premier, c’est une fonction plutôt trompeuse qui comporte deux pièges principaux:

1. Si la copie via `strncpy()` frappe la limite du tampon, un caractère nul `strncpy()` ne sera pas écrit.
2. `strncpy()` remplit toujours complètement la destination, avec des octets nuls si nécessaire.

(Cette implémentation bizarre est historique et était initialement destinée à gérer les noms de fichiers UNIX)

La seule façon correcte de l'utiliser est d'assurer manuellement la null-terminaison:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Même alors, si vous avez un gros tampon, il devient très inefficace d'utiliser `strncpy()` raison du remplissage null supplémentaire.

Convertir les chaînes en nombre: atoi (), atof () (dangereux, ne les utilisez pas)

Avertissement: Les fonctions `atoi` , `atol` , `atoll` et `atof` sont intrinsèquement dangereuses, car: *Si la valeur du résultat ne peut pas être représentée, le comportement est indéfini.* (7.20.1p1)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int val;
    if (argc < 2)
    {
        printf("Usage: %s <integer>\n", argv[0]);
        return 0;
    }

    val = atoi(argv[1]);

    printf("String value = %s, Int value = %d\n", argv[1], val);

    return 0;
}
```

Lorsque la chaîne à convertir est un entier décimal valide compris dans la plage, la fonction fonctionne:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

Pour les chaînes qui commencent par un nombre suivi de quelque chose d'autre, seul le nombre initial est analysé:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
123
```

Dans tous les autres cas, le comportement est indéfini:

```
$ ./atoi hello
Formatting the hard disk...
```

En raison des ambiguïtés ci-dessus et de ce comportement indéfini, la famille de fonctions `atoi` ne devrait jamais être utilisée.

- Pour convertir en `long int` , utilisez `strtol()` au lieu de `atol()` .
- Pour convertir en `double` , utilisez `strtod()` au lieu de `atof()` .

C99

- Pour convertir en `long long int` , utilisez `strtoll()` au lieu de `atoll()` .

données formatées en chaîne lecture / écriture

Écrire des données formatées en chaîne

```
int sprintf ( char * str, const char * format, ... );
```

Utilisez la fonction `sprintf` pour écrire des données flottantes dans une chaîne.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Lire des données formatées à partir d'une chaîne

```
int sscanf ( const char * s, const char * format, ...);
```

Utilisez la fonction `sscanf` pour analyser les données formatées.

```
#include <stdio.h>
int main ()
{
  char sentence []="date : 06-06-2012";
  char str [50];
  int year;
  int month;
  int day;
```

```
  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
  return 0;
}
```

Convertir en toute sécurité des chaînes en nombre: fonctions strtoX

C99

Depuis C99, la bibliothèque C possède un ensemble de fonctions de conversion sécurisées qui interprètent une chaîne sous forme de nombre. Leurs noms sont de la forme `strtoX` , où `X` est l'un de `l` , `ul` , `d` , etc. pour déterminer le type de cible de la conversion

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

Ils vérifient qu'une conversion a un débordement ou un débordement:

```
double ret = strtod(argv[1], 0); /* attempt conversion */

/* check the conversion result. */
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return;  /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Si la chaîne ne contient aucun numéro, cette utilisation de `strtod` renvoie `0.0` .

Si cela n'est pas satisfaisant, le paramètre supplémentaire `endptr` peut être utilisé. C'est un pointeur sur le pointeur qui sera dirigé vers la fin du nombre détecté dans la chaîne. S'il est défini sur `0` , comme ci-dessus ou `NULL` , il est simplement ignoré.

Ce paramètre `endptr` indique s'il y a eu une conversion réussie et, le cas échéant, où le numéro s'est terminé:

```
char *check = 0;
double ret = strtod(argv[1], &check); /* attempt conversion */

/* check the conversion result. */
if (argv[1] == check)
    return; /* No number was detected in string */
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return; /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Il existe des fonctions analogues à convertir en types entiers plus larges:

```
long strtol(char const* p, char** endptr, int nbase);
long long strtoll(char const* p, char** endptr, int nbase);
unsigned long strtoul(char const* p, char** endptr, int nbase);
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

Ces fonctions ont un troisième paramètre `nbase` la base numérique dans laquelle le numéro est écrit.

```
long a = strtol("101",   0, 2 ); /* a = 5L */
long b = strtol("101",   0, 8 ); /* b = 65L */
long c = strtol("101",   0, 10); /* c = 101L */
long d = strtol("101",   0, 16); /* d = 257L */
long e = strtol("101",   0, 0 ); /* e = 101L */
long f = strtol("0101",  0, 0 ); /* f = 65L */
long g = strtol("0x101", 0, 0 ); /* g = 257L */
```

La valeur spéciale `0` pour `nbase` signifie que la chaîne est interprétée de la même manière que les littéraux numériques sont interprétés dans un programme C: un préfixe `0x` correspond à une représentation hexadécimale, sinon un `0` est octal et tous les autres nombres sont décimaux.

Ainsi, le moyen le plus pratique d'interpréter un argument de ligne de commande comme un nombre serait

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */

     ...

    return EXIT_SUCCESS;
}
```

Cela signifie que le programme peut être appelé avec un paramètre en octal, décimal ou hexadécimal.

Lire Cordes en ligne: https://riptutorial.com/fr/c/topic/1990/cordes

Chapitre 19: Créer et inclure des fichiers d'en-tête

Introduction

Dans le moderne C, les fichiers d'en-tête sont des outils cruciaux qui doivent être conçus et utilisés correctement. Ils permettent au compilateur de vérifier par recoupement des parties d'un programme compilées de manière indépendante.

Les en-têtes déclarent les types, les fonctions, les macros, etc. nécessaires aux utilisateurs d'un ensemble de fonctions. Tout le code qui utilise l'une de ces fonctionnalités inclut l'en-tête. Tout le code qui définit ces fonctionnalités inclut l'en-tête. Cela permet au compilateur de vérifier que les utilisations et les définitions correspondent.

Examples

introduction

Il existe un certain nombre de directives à suivre lors de la création et de l'utilisation de fichiers d'en-tête dans un projet C:

- Idemopotence

  Si un fichier d'en-tête est inclus plusieurs fois dans une unité de traduction (TU), il ne doit pas interrompre les générations.

- Auto-confinement

  Si vous avez besoin des fonctionnalités déclarées dans un fichier d'en-tête, vous ne devriez pas avoir à inclure explicitement d'autres en-têtes.

- Minimalité

  Vous ne devriez pas être en mesure de supprimer des informations d'un en-tête sans provoquer l'échec des builds.

- Inclure ce que vous utilisez (IWYU)

  Plus préoccupant en C ++ que C, mais néanmoins important en C aussi. Si le code dans une TU (appelez-le `code.c`) utilise directement les fonctionnalités déclarées par un en-tête (appelez-le `"headerA.h"`), alors `code.c` devrait `#include "headerA.h"` directement, même si la TU inclut un autre en-tête (appelez-le `"headerB.h"`) qui, pour le moment, inclut `"headerA.h"` .

Parfois, il peut y avoir des raisons suffisantes pour enfreindre une ou plusieurs de ces directives, mais vous devez être conscient que vous enfreignez la règle et être conscient des conséquences de le faire avant de le casser.

Idempotence

Si un fichier d'en-tête particulier est inclus plusieurs fois dans une unité de traduction (TU), il ne devrait pas y avoir de problèmes de compilation. C'est ce qu'on appelle «l'idempotence»; vos en-têtes doivent être idempotents. Pensez à quel point la vie serait difficile si vous deviez vous assurer que `#include <stdio.h>` n'était inclus qu'une seule fois.

Il y a deux façons de parvenir à idempotence: les gardes-têtes et la directive `#pragma once` .

Gardes de tête

Les protège-têtes sont simples et fiables et conformes à la norme C. Les premières lignes sans commentaire dans un fichier d'en-tête doivent être de la forme:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

La dernière ligne sans commentaire doit être `#endif` , éventuellement avec un commentaire après:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

Tout le code opérationnel, y compris les autres directives `#include` , doit se trouver entre ces lignes.

Chaque nom doit être unique. Souvent, un schéma de nom tel que `HEADER_H_INCLUDED` est utilisé. Un code plus ancien utilise un symbole défini comme garde d'en-tête (par exemple `#ifndef BUFSIZ` dans `<stdio.h>`), mais il n'est pas aussi fiable qu'un nom unique.

Une option serait d'utiliser un hachage MD5 (ou autre) généré pour le nom de la garde d'en-tête. Vous devez éviter d'émuler les schémas utilisés par les en-têtes système qui utilisent fréquemment des noms réservés à l'implémentation - noms commençant par un trait de soulignement suivi d'un autre trait de soulignement ou d'une lettre majuscule.

La directive `#pragma once`

Certains compilateurs prennent également en charge la directive `#pragma once` qui a le même effet que les trois lignes affichées pour les gardes d’en-tête.

```
#pragma once
```

Les compilateurs qui supportent `#pragma once` incluent MS Visual Studio et GCC et Clang. Toutefois, si la portabilité est un problème, il est préférable d’utiliser des gardes d’en-tête ou d’utiliser les deux. Les compilateurs modernes (ceux prenant en charge C89 ou version ultérieure) doivent ignorer, sans commentaire, les pragmas qu'ils ne reconnaissent pas («Tout pragma qui n'est pas reconnu par l'implémentation est ignoré») mais les anciennes versions de GCC n'étaient pas si indulgentes.

Auto-confinement

Les en-têtes modernes doivent être autonomes, ce qui signifie qu'un programme devant utiliser les fonctionnalités définies par `header.h` peut inclure cet en-tête (`#include "header.h"`) et ne pas se soucier de savoir si d'autres en-têtes doivent être inclus en premier.

Recommandation: les fichiers d'en-tête doivent être autonomes.

Règles historiques

Historiquement, ce sujet a été légèrement controversé.

Un autre millénaire, les normes de codage et de codage AT & T d'Indian Hill ont déclaré:

> Les fichiers d'en-tête ne doivent pas être imbriqués. Le prologue d'un fichier d'en-tête doit donc décrire quels autres en-têtes doivent être `#include` d pour que l'en-tête soit fonctionnel. Dans les cas extrêmes, où un grand nombre de fichiers d'en-tête doivent être inclus dans plusieurs fichiers source différents, il est acceptable de mettre tous les fichiers `#include` communs dans un fichier d'inclusion.

C'est l'antithèse de la maîtrise de soi.

Règles modernes

Cependant, depuis lors, les opinions ont évolué dans la direction opposée. Si un fichier source doit utiliser les fonctionnalités déclarées par un en-tête `header.h` , le programmeur doit pouvoir écrire:

```
#include "header.h"
```

et (uniquement sous réserve que les chemins de recherche corrects soient définis sur la ligne de commande), les en-têtes `header.h` nécessaires seront inclus par `header.h` sans avoir besoin d'ajouter d'autres en-têtes au fichier source.

Cela fournit une meilleure modularité pour le code source. Il protège également la source de l'énigme "pourquoi cet en-tête a été ajouté" qui survient après que le code a été modifié et piraté pendant une décennie ou deux.

Les normes de codage de la NASA Goddard Space Flight Center (GSFC) pour C sont l'une des normes les plus modernes - mais sont maintenant un peu difficiles à retrouver. Il indique que les en-têtes doivent être autonomes. Il fournit également un moyen simple de s’assurer que les en-têtes sont autonomes: le fichier d’implémentation de l’en-tête doit inclure l’en-tête comme premier en-tête. S'il n'est pas autonome, ce code ne sera pas compilé.

La justification donnée par GSFC comprend:

> §2.1.1 En-tête inclure la justification
>
> Cette norme exige que l'en-tête d'une unité contienne des instructions `#include` pour tous les autres en-têtes requis par l'en-tête de l'unité. Placer `#include` pour l'en-tête d'unité en premier dans le corps d'unité permet au compilateur de vérifier que l'en-tête contient toutes les instructions `#include` requises.
>
> Une autre conception, non autorisée par cette norme, n'autorise aucune instruction `#include` dans les en-têtes; tous les `#includes` sont effectués dans les fichiers du corps. Les fichiers d'en-tête d'unité doivent alors contenir des instructions #ifdef qui vérifient que les en-têtes requis sont inclus dans le bon ordre.
>
> L'un des avantages de la conception alternative est que la liste `#include` dans le fichier de corps est exactement la liste de dépendances requise dans un fichier make, et cette liste est vérifiée par le compilateur. Avec la conception standard, un outil doit être utilisé pour générer la liste des dépendances. Cependant, tous les environnements de développement recommandés par la branche fournissent un tel outil.
>
> Un inconvénient majeur de la conception alternative est que si la liste d'en-tête requise d'une unité change, chaque fichier qui utilise cette unité doit être modifié pour mettre à jour la liste d'instructions `#include` . En outre, la liste d'en-tête requise pour une unité de bibliothèque de compilateur peut être différente sur différentes cibles.
>
> Un autre inconvénient de la conception alternative est que les fichiers d'en-tête de la bibliothèque du compilateur et les autres fichiers tiers doivent être modifiés pour ajouter les instructions `#ifdef` requises.

Ainsi, l'auto-confinement signifie que:

- Si un en-tête `header.h` besoin d'un nouvel en-tête imbriqué `extra.h` , vous n'avez pas à vérifier chaque fichier source qui utilise `header.h` pour voir si vous devez ajouter des `extra.h` .
- Si un en-tête `header.h` n'a plus besoin d'inclure un en-tête spécifique `notneeded.h` , vous n'avez pas besoin de vérifier chaque fichier source qui utilise `header.h` pour voir si vous pouvez supprimer `notneeded.h` (mais voir Inclure ce que vous utilisez) .
- Vous n'avez pas à établir la séquence correcte pour inclure les en-têtes pré-requis (ce qui nécessite un tri topologique pour effectuer le travail correctement).

Vérification de l'auto-confinement

Voir Liaison avec une bibliothèque statique pour un script `chkhdr` qui peut être utilisé pour tester l'idempotence et l'auto-confinement d'un fichier d'en-tête.

Minimalité

Les en-têtes sont un mécanisme de vérification de la cohérence crucial, mais ils doivent être aussi petits que possible. En particulier, cela signifie qu'un en-tête ne doit pas inclure d'autres en-têtes simplement parce que le fichier d'implémentation aura besoin des autres en-têtes. Un en-tête ne doit contenir que les en-têtes nécessaires à un consommateur des services décrits.

Par exemple, un en-tête de projet ne doit pas inclure `<stdio.h>` sauf si l'une des interfaces de fonction utilise le type `FILE *` (ou l'un des autres types définis uniquement dans `<stdio.h>`). Si une interface utilise `size_t` , le plus petit en-tête suffisant est `<stddef.h>` . De toute évidence, si un autre en-tête qui définit `size_t` est inclus, il n'est pas nécessaire d'inclure également `<stddef.h>` .

Si les en-têtes sont minimes, le temps de compilation est également réduit au minimum.

Il est possible de concevoir des en-têtes dont le seul but est d'inclure beaucoup d'autres en-têtes. Celles-ci s'avèrent rarement être une bonne idée à long terme, car peu de fichiers source auront besoin de toutes les fonctionnalités décrites par tous les en-têtes. Par exemple, un `<standard-ch>` pourrait être conçu, incluant tous les en-têtes C standard, car certains en-têtes ne sont pas toujours présents. Cependant, très peu de programmes utilisent réellement les fonctionnalités de `<locale.h>` ou `<tgmath.h>` .

- Voir aussi Comment lier plusieurs fichiers d'implémentation en C?

Inclure ce que vous utilisez (IWYU)

Le projet Include What You Use de Google, ou IWYU, garantit que les fichiers source incluent tous les en-têtes utilisés dans le code.

Supposons qu'un fichier source `source.c` inclue un en-tête `arbitrary.h` qui à son tour inclut `freeloader.h` , mais que le fichier source utilise aussi explicitement et indépendamment les fonctionnalités de `freeloader.h` . Tout va bien pour commencer. Ensuite, un jour `arbitrary.h` est modifié afin que ses clients n'aient plus besoin des fonctionnalités de `freeloader.h` . Soudain, `source.c` arrête la compilation, car elle ne répond pas aux critères IWYU. Étant donné que le code dans `source.c` utilisait explicitement les fonctionnalités de `freeloader.h` , il aurait dû inclure ce qu'il utilise - il devrait également y avoir eu un `#include "freeloader.h"` explicite dans le source. (Idempotency aurait assuré qu'il n'y avait pas de problème.)

La philosophie IWYU maximise la probabilité que le code continue à se compiler, même avec des modifications raisonnables apportées aux interfaces. De toute évidence, si votre code appelle une fonction qui est ensuite supprimée de l'interface publiée, aucune préparation ne peut empêcher des modifications. C'est pourquoi les modifications apportées aux API sont évitées dans la mesure du possible, et les cycles de dépréciation sont multiples sur plusieurs versions, etc.

Ceci est un problème particulier en C ++ car les en-têtes standard sont autorisés à s'inclure. Le fichier source `file.cpp` peut inclure un en-tête `header1.h` qui sur une plate-forme inclut un autre en-tête `header2.h` . `file.cpp` peut que `file.cpp` utilise également les fonctionnalités de `header2.h` . Ce ne serait pas un problème au départ - le code compilerait parce `header1.h` comprend `header2.h` . Sur une autre plate-forme, ou une mise à niveau de la plate-forme actuelle, `header1.h` pourrait être révisé pour ne plus inclure `header2.h` , puis `file.cpp` cesserait de compiler en conséquence.

IWYU identifiera le problème et recommandera que `header2.h` soit inclus directement dans `file.cpp` . Cela permettrait de continuer à compiler. Des considérations analogues s'appliquent également au code C.

Notation et Divers

Le standard C indique qu'il y a très peu de différence entre les `#include <header.h>` et `#include "header.h"` .

> [`#include <header.h>`] recherche une séquence de lieux définis par l'implémentation pour un en-tête identifié de manière unique par la séquence spécifiée entre les délimiteurs `<` et `>` et provoque le remplacement de cette directive par le contenu entier de l'en-tête. La façon dont les lieux sont spécifiés ou l'en-tête identifié sont définis par la mise en œuvre.
>
> [`#include "header.h"`] provoque le remplacement de cette directive par tout le contenu du fichier source identifié par la séquence spécifiée entre les délimiteurs `"…"` . Le fichier source nommé est recherché d'une manière définie par l'implémentation. Si cette recherche n'est pas prise en charge ou si la recherche échoue, la directive est traitée comme si elle lisait [`#include <header.h>`]…

Ainsi, la forme de guillemets doubles peut regarder plus d'endroits que la forme entre crochets. La norme spécifie par exemple que les en-têtes standard doivent être inclus entre crochets, même si la compilation fonctionne à la place des guillemets. De même, les normes telles que POSIX utilisent le format entre crochets - et vous devriez aussi le faire. Réserver des en-têtes entre guillemets doubles pour les en-têtes définis par le projet. Pour les en-têtes définis en externe (y compris les en-têtes d'autres projets sur lesquels repose votre projet), la notation entre crochets est la plus appropriée.

Notez qu'il doit y avoir un espace entre `#include` et l'en-tête, même si les compilateurs n'acceptent aucun espace. Les espaces sont bon marché.

Un certain nombre de projets utilisent une notation telle que:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

Vous devriez envisager d'utiliser ce contrôle d'espace de noms dans votre projet (c'est probablement une bonne idée). Vous devriez éviter les noms utilisés par les projets existants (en particulier, `sys` et `linux` seraient de mauvais choix).

Si vous utilisez ceci, votre code doit être prudent et cohérent dans l'utilisation de la notation.

N'utilisez pas la notation `#include "../include/header.h"` .

Les fichiers d'en-tête doivent rarement, voire jamais, définir des variables. Bien que vous gardiez les variables globales au minimum, si vous avez besoin d'une variable globale, vous le déclarerez dans un en-tête et le définirez dans un fichier source approprié, et ce fichier source inclura l'en-tête pour vérifier la déclaration et la définition. , et tous les fichiers sources utilisant la variable utiliseront l'en-tête pour le déclarer.

Corollaire: vous ne déclarerez pas de variables globales dans un fichier source - un fichier source ne contiendra que des définitions.

Les fichiers d'en-tête doivent rarement déclarer `static` fonctions `static` , à l'exception notable des fonctions `static inline` qui seront définies dans les en-têtes si la fonction est nécessaire dans plusieurs fichiers source.

- Les fichiers source définissent des variables globales et des fonctions globales.
- Les fichiers sources ne déclarent pas l'existence de variables ou de fonctions globales; ils incluent l'en-tête qui déclare la variable ou la fonction.
- Les fichiers d'en-tête déclarent la variable globale et les fonctions (et les types et autres éléments de support).
- Les fichiers d'en-tête ne définissent aucune variable ou fonction, à l'exception `inline` fonctions en `inline` (`static`).

Références croisées

- Où documenter les fonctions en C?
- Liste des fichiers d'en-tête standard en C et C ++
- Est-ce que `inline` sans `static` ou `extern` utile dans C99?
- Comment utiliser `extern` pour partager des variables entre les fichiers sources?
- Quels sont les avantages d'un chemin relatif tel que `"../include/header.h"` pour un en-tête?
- Optimisation de l'inclusion d'en-tête
- Dois-je inclure chaque en-tête?

Lire Créer et inclure des fichiers d'en-tête en ligne: https://riptutorial.com/fr/c/topic/6257/creer-et-inclure-des-fichiers-d-en-tete

Chapitre 20: Déclaration vs définition

Remarques

Source: Quelle est la différence entre une définition et une déclaration?

Source (Pour les symboles faibles et forts): https://www.amazon.com/Computer-Systems-Programmers-Perspective-2nd/dp/0136108040/

Examples

Comprendre la déclaration et la définition

Une déclaration introduit un identifiant et décrit son type, que ce soit un type, un objet ou une fonction. Une déclaration est ce que le compilateur doit accepter des références à cet identifiant. Ce sont des déclarations:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto                         */
```

Une définition instancie / implémente cet identifiant. C'est ce que l'éditeur de liens a besoin pour relier les références à ces entités. Ce sont des définitions correspondant aux déclarations ci-dessus:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {}  /* prototype is implied in definition, same as double h2(void) */
```

Une définition peut être utilisée à la place d'une déclaration.

Cependant, il doit être défini exactement une fois. Si vous oubliez de définir quelque chose qui a été déclaré et référencé quelque part, alors l'éditeur de liens ne sait pas à quoi lier les références et se plaint des symboles manquants. Si vous définissez quelque chose plus d'une fois, l'éditeur de liens ne sait pas laquelle des définitions pour lier les références à des symboles dupliqués.

Exception:

```
extern int i = 0;  /* defines i */
extern int j;  /* declares j */
```

Cette exception peut être expliquée en utilisant les concepts de "symboles forts vs symboles faibles" (du point de vue d'un éditeur de liens). Veuillez regarder ici (diapositive 22) pour plus

d'explications.

```
/* All are definitions. */
struct S { int a; int b; };             /* defines S */
struct X {                              /* defines X */
    int x;                              /* defines non-static data member x */
};
struct X anX;                                  /* defines anX */
```

Lire Déclaration vs définition en ligne: https://riptutorial.com/fr/c/topic/3104/declaration-vs-definition

Chapitre 21: Déclarations

Remarques

La déclaration d'identifiant faisant référence à un objet ou à une fonction est souvent désignée simplement comme une déclaration d'objet ou de fonction.

Examples

Appeler une fonction depuis un autre fichier C

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/

/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

principal c

```
#include "foo.h"
```

```
int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Compiler et Lier

Tout d' abord, nous *compilons* les deux `foo.c` et `main.c` aux *fichiers objet.* Ici, nous utilisons le compilateur `gcc` , votre compilateur peut avoir un nom différent et avoir besoin d'autres options.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Maintenant, nous les lions ensemble pour produire notre exécutable final:

```
$ gcc -o testprogram foo.o main.o
```

Utilisation d'une variable globale

L'utilisation de variables globales est généralement déconseillée. Cela rend votre programme plus difficile à comprendre et plus difficile à déboguer. Mais parfois, l'utilisation d'une variable globale est acceptable.

global.h

```
#ifndef GLOBAL_DOT_H    /* This is an "include guard" */
#define GLOBAL_DOT_H

/**
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                        * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal;     /* _Define_ my_global. As living in global scope it gets initialised to 0
                     * on program start-up. */
```

principal c

```
#include "global.h"
```

```
int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

Voir aussi Comment utiliser `extern` pour partager des variables entre des fichiers sources?

Utiliser des constantes globales

Les en-têtes peuvent être utilisés pour déclarer des ressources en lecture seule utilisées globalement, comme les tables de chaînes par exemple.

Déclarez ceux d'un en-tête séparé qui est inclus par tout fichier (" *Translation Unit* ") qui veut les utiliser. Il est pratique d'utiliser le même en-tête pour déclarer une énumération associée afin d'identifier toutes les ressources de chaîne:

ressources.h:

```
#ifndef RESOURCES_H
#define RESOURCES_H

typedef enum { /* Define a type describing the possible valid resource IDs. */
  RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be

                              marked as "not in use", "not in list", "undefined", wtf.
                              Will say un-initialised on application level, not on language
level. Initialised uninitialised, so to say ;-)
                              Its like NULL for pointers ;-)*/
  RESOURCE_UNKNOWN = 0,    /* To be used if the application uses some resource ID,
                              for which we do not have a table entry defined, a fall back in
                              case we _need_ to display something, but do not find anything
                              appropriate. */

  /* The following identify the resources we have defined: */
  RESOURCE_OK,
  RESOURCE_CANCEL,
  RESOURCE_ABORT,
  /* Insert more here. */

  RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;


extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes

                                      this, that at linkage-time this symbol will be around.
                                      The 1st const guarantees the strings will not change,
                                      the 2nd const guarantees the string-table entries
                                      will never suddenly point somewhere else as set during
                                      initialisation. */
#endif
```

Pour définir réellement les ressources créées dans un fichier .c associé, il s'agit d'une autre unité de traduction contenant les instances réelles de ce qui a été déclaré dans le fichier d'en-tête (.h)

associé:

resources.c:

```
#include "resources.h" /* To make sure clashes between declaration and definition are
                          recognised by the compiler include the declaring header into
                          the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
  "<unknown>",
  "OK",
  "Cancel",
  "Abort"
};
```

Un programme utilisant ceci pourrait ressembler à ceci:

principal c:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>

#include "resources.h"


int main(void)
{
  EnumResourceID resource_id = RESOURCE_UNDEFINED;

  while ((++resource_id) < RESOURCE_MAX)
  {
    printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
  }

  return EXIT_SUCCESS;
}
```

Compilez les trois fichiers ci-dessus en utilisant GCC, et liez-les pour devenir le fichier `main` du programme, par exemple en utilisant ceci:

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(utilisez ces `-Wall -Wextra -pedantic -Wconversion` pour rendre le compilateur vraiment pointilleux, de sorte que vous ne manquiez de rien avant de poster le code dans SO, direz le monde entier, ou même le déployez en production)

Exécutez le programme créé:

```
$ ./main
```

Et obtenir:

```
resource ID: 0, resource: '<unknown>'
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
resource ID: 3, resource: 'Abort'
```

introduction

Exemple de déclaration:

```
int a; /* declaring single identifier of type int */
```

La déclaration ci-dessus déclare un identifiant unique nommé `a` qui fait référence à un objet de type `int` .

```
int a1, b1; /* declaring 2 identifiers of type int */
```

La seconde déclaration déclare 2 identificateurs nommés `a1` et `b1` qui font référence à d'autres objets avec le même type `int` .

Fondamentalement, la façon dont cela fonctionne est comme ceci - d'abord vous mettez un type , puis vous écrivez une ou plusieurs expressions séparées par une virgule (`,` **qui ne seront pas évaluées à ce stade - et qui devraient autrement être qualifiées de déclarants dans ce contexte**). En écrivant de telles expressions, vous êtes autorisé à appliquer uniquement les opérateurs indirection (`*`), appel de fonction (`( )`) ou indice (ou indexation de tableau - `[ ]`) à un identifiant (vous ne pouvez également utiliser aucun opérateur). L'identifiant utilisé n'a pas besoin d'être visible dans la portée actuelle. Quelques exemples:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	La description
1	Le nom du type entier.
2	Expression non évaluée appliquant l'indirection à un identifiant `z` .
3	Nous avons une virgule indiquant qu'une expression de plus suivra dans la même déclaration.
4	Expression non évaluée appliquant l'indirection à un autre identifiant `x` .
5	Expression non évaluée appliquant l'indirection à la valeur de l'expression `(*c)` .
6	Fin de déclaration

Notez qu'aucun des identificateurs ci-dessus n'était visible avant cette déclaration et que les expressions utilisées ne seraient donc pas valides avant.

Après chaque expression, l'identifiant utilisé est introduit dans la portée actuelle. (Si l'identifiant lui

a été associé, il peut également être déclaré de nouveau avec le même type de liaison afin que les deux identificateurs se réfèrent au même objet ou à la même fonction)

De plus, le signe d'opérateur égal (=) peut être utilisé pour l'initialisation. Si une expression non évaluée (déclarateur) est suivie de = dans la déclaration - on dit que l’identifiant introduit est également en cours d’initialisation. Après le signe = on peut replacer une expression, mais cette fois-ci, elle sera évaluée et sa valeur sera utilisée comme initiale pour l'objet déclaré.

Exemples:

```
int l = 90; /* the same as: */

int l; l = 90; /* if it the declaration of l was in block scope */

int c = 2, b[c]; /* ok, equivalent to: */

int c = 2; int b[c];
```

Plus tard dans votre code, vous êtes autorisé à écrire exactement la même expression à partir de la partie déclaration de l'identifiant nouvellement introduit, en vous donnant un objet du type spécifié au début de la déclaration, en supposant que vous avez attribué des valeurs valides à tous objets dans le chemin. Exemples:

```
void f()
{
    int b2; /* you should be able to write later in your code b2
            which will directly refer to the integer object
            that b2 identifies */

    b2 = 2; /* assign a value to b2 */

    printf("%d", b2); /*ok - should print 2*/

    int *b3; /* you should be able to write later in your code *b3 */

    b3 = &b2; /* assign valid pointer value to b3 */

    printf("%d", *b3); /* ok - should print 2 */

    int **b4; /* you should be able to write later in your code **b4 */

    b4 = &b3;

    printf("%d", **b4); /* ok - should print 2 */

    void (*p)(); /* you should be able to write later in your code (*p)() */

    p = &f; /* assign a valid pointer value */

    (*p)(); /* ok - calls function f by retrieving the
            pointer value inside p -    p
            and dereferencing it -      *p
            resulting in a function
            which is then called -      (*p)() -

            it is not *p() because else first the () operator is
```

```
             applied to p and then the resulting void object is
             dereferenced which is not what we want here */
}
```

La déclaration de `b3` spécifie que vous pouvez potentiellement utiliser la valeur `b3` comme moyen d'accéder à un objet entier.

Bien sûr, pour appliquer l'indirection (`*`) à `b3` , vous devez également y stocker une valeur correcte (voir les pointeurs pour plus d'informations). Vous devez également d'abord stocker une valeur dans un objet avant d'essayer de le récupérer (vous pouvez en savoir plus sur ce problème ici). Nous avons fait tout cela dans les exemples ci-dessus.

```
int a3(); /* you should be able to call a3 */
```

Celui-ci indique au compilateur que vous allez essayer d'appeler `a3` . Dans ce cas, `a3` réfère à la fonction au lieu d'un objet. Une différence entre l'objet et la fonction est que les fonctions auront toujours une sorte de liaison. Exemples:

```
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

Dans l'exemple ci-dessus, les deux déclarations se réfèrent à la même fonction `f2` , alors que si elles déclaraient des objets, alors dans ce contexte (ayant deux portées de bloc différentes), elles seraient deux objets distincts différents.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Maintenant, cela peut sembler compliqué, mais si vous connaissez les opérateurs de priorité, vous aurez 0 problème pour lire la déclaration ci-dessus. Les parenthèses sont nécessaires car l'opérateur `*` a moins de priorité que celui `( )` .

Dans le cas de l'utilisation de l'opérateur subscript, l'expression résultante ne serait pas réellement valide après la déclaration car l'index utilisé (la valeur comprise entre `[` et `]`) sera toujours 1 au-dessus de la valeur maximale autorisée pour cet objet / fonction.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

Mais il devrait être accessible par tous les autres index inférieurs à 5. Exemples:

```
a4[0], a4[1]; a4[4];
```

`a4[5]`

donnera comme résultat UB. Vous trouverez plus d'informations sur les tableaux ici .

```
int (*a5)[5](); /* here a4 could be applied indirection
                indexed up to (but not including) 5
                and called */
```

Malheureusement pour nous, bien que syntaxiquement possible, la déclaration de `a5` est interdite par la norme actuelle.

Typedef

Les typedefs sont des déclarations qui ont le mot-clé `typedef` devant et avant le type. Par exemple:

```
typedef int (*(*t0)())[5];
```

(*vous pouvez techniquement mettre le typedef après le type aussi - comme ceci* `int typedef (*(*t0)())[5];` *mais ceci est déconseillé*)

Les déclarations ci-dessus déclarent un identifiant pour un nom typedef. Vous pouvez l'utiliser comme ça après:

```
t0 pf;
```

Qui aura le même effet que l'écriture:

```
int (*(*pf)())[5];
```

Comme vous pouvez le voir, le nom typedef "enregistre" la déclaration en tant que type à utiliser ultérieurement pour d'autres déclarations. De cette façon, vous pouvez enregistrer certaines frappes. De même que la déclaration utilisant `typedef` est toujours une déclaration, vous n'êtes pas limité uniquement par l'exemple ci-dessus:

```
t0 (*pf1);
```

Est le même que:

```
int (*(**pf1)())[5];
```

Utilisation de la règle de droite ou de spirale pour déchiffrer la déclaration C

La règle "right left" est une règle complètement régulière pour déchiffrer les déclarations C. Cela peut aussi être utile pour les créer.

Lisez les symboles lorsque vous les rencontrez dans la déclaration ...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

Comment appliquer la règle

ÉTAPE 1

Trouvez l'identifiant. C'est votre point de départ. Puis dites-vous, "l'identifiant est". Vous avez commencé votre déclaration.

ÉTAPE 2

Regardez les symboles à droite de l'identifiant. Si, par exemple, vous trouvez `()` , vous savez que c'est la déclaration d'une fonction. Vous auriez alors alors *"l'identifiant est la fonction qui retourne"* . Ou si vous avez trouvé un `[]` , vous diriez *"identifiant est un tableau de"* . Continuez tout droit jusqu'à ce que vous n'ayez plus de symboles OU appuyez sur une parenthèse droite `)` . (Si vous appuyez sur une parenthèse gauche `(` c'est le début d'un symbole `()` , même s'il y a des choses entre les parenthèses).

ÉTAPE 3

Regardez les symboles à gauche de l'identifiant. Si ce n'est pas l'un de nos symboles ci-dessus (par exemple, quelque chose comme "int"), dites-le simplement. Sinon, traduisez-le en anglais en utilisant ce tableau ci-dessus. Continuez à gauche jusqu'à épuisement des symboles OU appuyez sur une parenthèse gauche `(` .

Répétez maintenant les étapes 2 et 3 jusqu'à ce que vous ayez formé votre déclaration.

Voici quelques exemples:

```
int *p[];
```

Tout d'abord, identifiant de recherche:

```
int *p[];
     ^
```

"p est"

Maintenant, déplacez-vous à droite jusqu'à ce que moins de symboles ou de parenthèses droites soient frappés.

```
int *p[];
      ^^
```

"p est un tableau de"

Ne peut plus bouger (sans symboles), déplacez-vous à gauche et trouvez:

```
int *p[];
    ^
```

"p est un tableau de pointeur vers"

Continuez à gauche et trouvez:

```
int *p[];
^^^
```

"p est un tableau de pointeur sur int".

(ou *"p est un tableau dont chaque élément est de type pointeur sur int"*)

Un autre exemple:

```
int *(*func())();
```

Trouvez l'identifiant.

```
int *(*func())();
       ^^^^
```

"func is"

Déplacer vers la droite.

```
int *(*func())();
           ^^
```

"func est la fonction qui retourne"

Ne peut plus bouger à cause de la parenthèse droite, alors déplacez-vous vers la gauche.

```
int *(*func())();
      ^
```

"func est la fonction qui retourne le pointeur sur"

Ne peut plus bouger à cause de la parenthèse gauche, alors continuez comme ça.

```
int *(*func())();
              ^^
```

"func est la fonction renvoyant le pointeur à la fonction retournant"

Je ne peux plus bouger parce que nous n'avons plus de symboles, alors allez à gauche.

```
int *(*func())();
    ^
```

"func est la fonction renvoyant le pointeur à la fonction retournant le pointeur à"

Et enfin, continuez à gauche, car il ne reste plus rien à droite.

```
int *(*func())();
^^^
```

"func est la fonction renvoyant le pointeur à la fonction renvoyant le pointeur à int".

Comme vous pouvez le voir, cette règle peut être très utile. Vous pouvez également l'utiliser pour vérifier vous-même pendant que vous créez des déclarations, et pour vous donner une idée de l'endroit où placer le symbole suivant et si des parenthèses sont nécessaires.

Certaines déclarations ont l'air beaucoup plus compliquées qu'elles ne le sont en raison de la taille des tableaux et des listes d'arguments sous forme de prototype. Si vous voyez `[3]` , cela se lit comme *"array (size 3) of ..."* . Si vous voyez `(char *,int)` qui se lit comme * "function attend (char *, int) et retourne ..."* .

Voici un amusant:

```
int (*(*fun_one)(char *,double))[9][20];
```

Je ne vais pas passer par chacune des étapes pour déchiffrer celle-ci.

* "fun_one est un pointeur sur la fonction qui attend (char *, double) et retourne le pointeur sur le tableau (taille 9) du tableau (taille 20) de int."*

Comme vous pouvez le voir, ce n'est pas si compliqué si vous vous débarrassez de la taille des tableaux et des listes d'arguments:

```
int (*(*fun_one)())[][];
```

Vous pouvez le déchiffrer de cette façon, puis placer les tailles de tableau et les listes d'arguments plus tard.

Quelques mots de conclusion:

Il est tout à fait possible de faire des déclarations illégales en utilisant cette règle, donc une connaissance de ce qui est légal dans C est nécessaire. Par exemple, si ce qui précède a été:

```
int *((*fun_one)())[][];
```

il aurait lu *"fun_one est un pointeur sur la fonction retournant un tableau de tableau de pointeur sur int"* . Une fonction ne pouvant pas retourner un tableau, mais uniquement un pointeur vers un tableau, cette déclaration est illégale.

Les combinaisons illégales comprennent:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
```

```
()[] - cannot have a function that returns an array
```

Dans tous les cas ci-dessus, vous auriez besoin d'un ensemble de parenthèses pour lier un symbole `*` à gauche entre ces symboles `()` et `[]` droite afin que la déclaration soit légale.

Voici quelques exemples supplémentaires:

Légal

```
int i;               an int
int *p;              an int pointer (ptr to an int)
int a[];             an array of ints
int f();             a function returning an int
int **pp;            a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];         a pointer to an array of ints
int (*pf)();         a pointer to a function returning an int
int *ap[];           an array of int pointers (array of ptrs to ints)
int aa[][];          an array of arrays of ints
int *fp();           a function returning an int pointer
int ***ppp;          a pointer to a pointer to an int pointer
int (**ppa)[];       a pointer to a pointer to an array of ints
int (**ppf)();       a pointer to a pointer to a function returning an int
int *(*pap)[];       a pointer to an array of int pointers
int (*paa)[][];      a pointer to an array of arrays of ints
int *(*pfp)();       a pointer to a function returning an int pointer
int **app[];         an array of pointers to int pointers
int (*apa[])[];      an array of pointers to arrays of ints
int (*apf[])();      an array of pointers to functions returning an int
int *aap[][];        an array of arrays of int pointers
int aaa[][][];       an array of arrays of arrays of int
int **fpp();         a function returning a pointer to an int pointer
int (*fpa())[];      a function returning a pointer to an array of ints
int (*fpf())();      a function returning a pointer to a function returning an int
```

Illégal

```
int af[]();          an array of functions returning an int
int fa()[];          a function returning an array of ints
int ff()();          a function returning a function returning an int
int (*pfa)()[];      a pointer to a function returning an array of ints
int aaf[][]();       an array of arrays of functions returning an int
int (*paf)[]();      a pointer to a an array of functions returning an int
int (*pff)()();      a pointer to a function returning a function returning an int
int *afp[]();        an array of functions returning int pointers
int afa[]()[];       an array of functions returning an array of ints
int aff[]()();       an array of functions returning functions returning an int
int *fap()[];        a function returning an array of int pointers
int faa()[][];       a function returning an array of arrays of ints
int faf()[]();       a function returning an array of functions returning an int
int *ffp()();        a function returning a function returning an int pointer
```

Source: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Lire Déclarations en ligne: https://riptutorial.com/fr/c/topic/3729/declarations

Chapitre 22: Effets secondaires

Examples

Opérateurs avant / après incrémentation / décrémentation

En C, il y a deux opérateurs unaires - «++» et «-» qui sont une source de confusion très commune. L'opérateur `++` est appelé *opérateur d'incrémentation* et l'opérateur `--` est appelé l' *opérateur de décrémentation* . Les deux peuvent être utilisés sous forme de *préfixe* ou de *postfixe* . La syntaxe pour la forme de préfixe pour l'opérateur `++` est l' `++operand` et la syntaxe pour la forme postfixe est `operand++` . Lorsqu'il est utilisé sous la forme de préfixe, l'opérande est incrémenté d'abord de `1` et la valeur résultante de l'opérande est utilisée dans l'évaluation de l'expression. Prenons l'exemple suivant:

```
int n, x = 5;
n = ++x; /* x is incremented  by 1(x=6), and result is assigned to n(6) */
         /* this is a short form for two statements: */
         /* x = x + 1; */
         /* n = x ; */
```

Lorsqu'elle est utilisée dans le formulaire postfixe, la valeur actuelle de l'opérande est utilisée dans l'expression, puis la valeur de l'opérande est incrémentée de `1` . Prenons l'exemple suivant:

```
int n, x = 5;
n = x++; /* value of x(5) is assigned first to n(5), and then x is incremented by 1; x(6) */
         /* this is a short form for two statements: */
         /* n = x; */
         /* x = x + 1; */
```

Le fonctionnement de l'opérateur de décrémentation `--` peut être compris de la même manière.

Le code suivant montre ce que chacun fait

```
    int main()
    {
        int a, b, x = 42;
        a = ++x; /* a and x are 43 */
        b = x++; /* b is 43, x is 44 */
        a = x--; /* a is is 44, x is 43 */
        b = --x; /* b and x are 42 */

        return 0;
    }
```

De ce qui précède , il est clair que les opérateurs de poste renvoient la valeur actuelle d'une variable, *puis* le modifier, mais avant les opérateurs modifient la variable *puis* retourner la valeur modifiée.

Dans toutes les versions de C, l'ordre d'évaluation des opérateurs pré et post n'est pas défini. Le

code suivant peut donc renvoyer des sorties inattendues:

```
int main()
{
    int a, x = 42;
    a = x++ + x; /* wrong */
    a = x + x; /* right */
    ++x;

    int ar[10];
    x = 0;
    ar[x] = x++; /* wrong */
    ar[x++] = x; /* wrong */
    ar[x] = x; /* right */
    ++x;
    return 0;
}
```

Notez que c'est également une bonne pratique d'utiliser des opérateurs pré-poste lorsqu'ils sont utilisés seuls dans une déclaration. Regardez le code ci-dessus pour cela.

Notez également que lorsqu'une fonction est appelée, tous les effets secondaires sur les arguments doivent avoir lieu avant l'exécution de la fonction.

```
int foo(int x)
{
    return x;
}

int main()
{
    int a = 42;
    int b = foo(a++);  /* This returns 43, even if it seems like it should return 42 */
    return 0;
}
```

Lire Effets secondaires en ligne: https://riptutorial.com/fr/c/topic/7094/effets-secondaires

Chapitre 23: Énoncés d'itération / boucles: pour, pendant et après

Syntaxe

- /* toutes les versions */
- pour ([expression]; [expression]; [expression]) one_statement
- for ([expression]; [expression]; [expression]) {zéro ou plusieurs déclarations}
- while (expression) one_statement
- while (expression) {zéro ou plusieurs déclarations}
- faire one_statement while (expression);
- faire {une ou plusieurs déclarations} while (expression);
- // depuis C99 en plus du formulaire ci-dessus
- pour (déclaration; [expression]; [expression]) one_statement;
- for (declaration; [expression]; [expression]) {zéro ou plusieurs déclarations}

Remarques

L'énoncé d'itération / les boucles se divisent en deux catégories:

- instruction / boucles d'itération contrôlée par la tête
- instruction d'itération / boucles contrôlées par le pied

Relevé d'itération / boucles sous contrôle de la tête

```
for ([<expression>]; [<expression>]; [<expression>]) <statement>
while (<expression>) <statement>
```

C99

```
for ([declaration expression]; [expression] [; [expression]]) statement
```

Relevé d'itération / boucles contrôlées au pied

```
do <statement> while (<expression>);
```

Examples

Pour la boucle

Afin d'exécuter un bloc de code sur une nouvelle fois, des boucles apparaissent dans l'image. La boucle `for` doit être utilisée lorsqu'un bloc de code doit être exécuté un nombre de fois fixe. Par

exemple, pour remplir un tableau de taille `n` avec les entrées utilisateur, nous devons exécuter `scanf()` `n` fois.

C99

```
#include <stddef.h>          // for size_t

int array[10];               // array of 10 int

for (size_t i = 0; i < 10; i++)    // i starts at 0 and finishes with 9
{
   scanf("%d", &array[i]);
}
```

De cette façon, l'appel de la fonction `scanf()` est exécuté `n` fois (10 fois dans notre exemple), mais n'est écrit qu'une seule fois.

Ici, la variable `i` est l'indice de boucle et il est préférable de le déclarer comme présenté. Le type `size_t` (*type de taille*) doit être utilisé pour tout ce qui compte ou parcourt des objets de données.

Cette façon de déclarer des variables dans le `for` est uniquement disponible pour les compilateurs mis à jour avec le standard C99. Si pour une raison quelconque vous êtes toujours bloqué avec un ancien compilateur, vous pouvez déclarer l'index de la boucle avant la boucle `for` :

C99

```
#include <stddef.h>        /* for size_t */
size_t i;
int array[10];             /* array of 10 int */

for (i = 0; i < 10; i++)       /* i starts at 0 and finishes at 9 */
{
   scanf("%d", &array[i]);
}
```

En boucle

A `while` la boucle est utilisée pour exécuter un morceau de code tandis qu'une condition est vraie. Le `while` en boucle doit être utilisé quand un bloc de code doit être exécuté un nombre variable de fois. Par exemple, le code affiché reçoit l'entrée utilisateur, à condition que l'utilisateur insère des nombres qui ne sont pas `0` . Si l'utilisateur insère `0` , la condition while n'est plus vraie, donc l'exécution quittera la boucle et passera à n'importe quel code suivant:

```
int num = 1;

while (num != 0)
{
    scanf("%d", &num);
}
```

Boucle Do-While

Contrairement `for` boucles `for` et `while` boucles `do-while` vérifient la vérité de la condition à la fin de la boucle, ce qui signifie que le bloc `do` s'exécutera une fois, puis vérifie l'état du `while` en bas du bloc. Ce qui signifie qu'une boucle `do-while` fong sera *toujours* exécutée au moins une fois.

Par exemple, cette boucle `do-while` while aura des nombres d'utilisateurs, jusqu'à ce que la somme de ces valeurs soit supérieure ou égale à `50` :

```
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

`do-while` boucles `do-while` sont relativement rares dans la plupart des styles de programmation.

Structure et flux de contrôle dans une boucle for

```
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

Dans une boucle `for` , la condition de la boucle a trois expressions, toutes facultatives.

- La première expression, `declaration-or-expression` , *initialise* la boucle. Il est exécuté exactement une fois au début de la boucle.

C99

Il peut s'agir d'une déclaration et d'une initialisation d'une variable de boucle ou d'une expression générale. S'il s'agit d'une déclaration, la portée de la variable déclarée est restreinte par l'instruction `for` .

C99

Les versions historiques de C permettaient uniquement une expression, ici, et la déclaration d'une variable de boucle devait être placée avant le `for` .

- La deuxième expression, `expression2` , est la *condition de test* . Il est d'abord exécuté après l'initialisation. Si la condition est `true` , le contrôle entre dans le corps de la boucle. Sinon, il passe à l'extérieur du corps de la boucle à la fin de la boucle. Par la suite, cette condition est vérifiée après chaque exécution du corps ainsi que la déclaration de mise à jour. Lorsque `true` , le contrôle revient au début du corps de la boucle. La condition est généralement destinée à vérifier le nombre d'exécutions du corps de la boucle. C'est le moyen principal de sortir d'une boucle, l'autre étant d'utiliser des instructions de saut .
- La troisième expression, `expression3` , est la *déclaration de mise à jour* . Il est exécuté après chaque exécution du corps de la boucle. Il est souvent utilisé pour incrémenter un nombre

de variables du nombre de fois que le corps de la boucle a été exécuté, et cette variable est appelée un *itérateur* .

Chaque instance d'exécution du corps de la boucle est appelée une *itération* .

Exemple:

C99

```
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

La sortie est la suivante:

```
0123456789
```

Dans l'exemple ci-dessus, `i = 0` est d'abord exécuté, en initialisant `i` . Ensuite, la condition `i < 10` est vérifiée, ce qui est considéré comme `true` . Le contrôle entre dans le corps de la boucle et la valeur de `i` est imprimée. Ensuite, le contrôle passe à `i++` , en mettant à jour la valeur de `i` de 0 à 1. La condition est à nouveau vérifiée et le processus continue. Cela continue jusqu'à ce que la valeur de `i` devienne 10. Ensuite, la condition `i < 10` `false` , après quoi le contrôle sort de la boucle.

Boucles infinies

Une boucle est dite une *boucle infinie* si le contrôle entre mais ne quitte jamais le corps de la boucle. Cela se produit lorsque la condition de test de la boucle n'est jamais évaluée à `false` .

Exemple:

C99

```
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed*/
}
```

Dans l'exemple ci-dessus, la variable `i` , l'itérateur, est initialisée à 0. La condition de test est initialement `true` . Cependant, `i` ne `i` modifié nulle part dans le corps et l'expression de mise à jour est vide. Par conséquent, `i` resterai 0 et la condition de test ne sera jamais évaluée à `false` , conduisant à une boucle infinie.

En supposant qu'il n'y ait pas d' instruction de saut, une autre manière de former une boucle infinie consiste à conserver la condition de façon explicite:

```
while (true)
{
    /* body of the loop */
```

```
}
```

Dans une boucle `for` , l'instruction de condition est facultative. Dans ce cas, la condition est toujours `true` , conduisant à une boucle infinie.

```
for (;;)
{
    /* body of the loop */
}
```

Cependant, dans certains cas, la condition peut être conservé `true` volontairement, avec l'intention de sortir de la boucle en utilisant une instruction de saut comme la `break` .

```
while (true)
{
    /* statements */
    if (condition)
    {
         /* more statements */
         break;
    }
}
```

Loop Unrolling et Duff's Device

Parfois, la boucle directe ne peut pas être entièrement contenue dans le corps de la boucle. En effet, la boucle doit être amorcée par certaines instructions ***B.*** Ensuite, l'itération commence par quelques instructions ***A*** , qui sont ensuite suivies par ***B*** avant de boucler.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

Pour éviter réduire le potentiel / problèmes de pâte à répéter ***B*** deux fois dans le code, le dispositif de Duff pourrait être appliquée pour démarrer la boucle à partir du milieu du `while` corps, en utilisant une instruction switch et tomber par un comportement.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
default:    do_B(); /* FALL THROUGH */
}
```

Le dispositif de Duff a été inventé pour implémenter le déroulement des boucles. Imaginez appliquer un masque à un bloc de mémoire, où `n` est un type intégral signé avec une valeur positive.

```
do {
    *ptr++ ^= mask;
} while (--n > 0);
```

Si `n` était toujours divisible par 4, vous pourriez facilement le dérouler comme suit:

```
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

Mais, avec Device Duff, le code peut suivre cet idiome déroulant qui saute au bon endroit au milieu de la boucle si `n` n'est pas divisible par 4.

```
switch (n % 4) do {
case 0: *ptr++ ^= mask; /* FALL THROUGH */
case 3: *ptr++ ^= mask; /* FALL THROUGH */
case 2: *ptr++ ^= mask; /* FALL THROUGH */
case 1: *ptr++ ^= mask; /* FALL THROUGH */
} while ((n -= 4) > 0);
```

Ce type de déroulement manuel est rarement requis avec les compilateurs modernes, car le moteur d'optimisation du compilateur peut dérouler les boucles au nom du programmeur.

Lire Énoncés d'itération / boucles: pour, pendant et après en ligne:
https://riptutorial.com/fr/c/topic/5151/enonces-d-iteration---boucles--pour--pendant-et-apres

Chapitre 24: Entrée / sortie formatée

Examples

Impression de la valeur d'un pointeur sur un objet

Pour imprimer la valeur d'un pointeur sur un objet (par opposition à un pointeur de fonction), utilisez le spécificateur de conversion `p` . Il est défini pour imprimer uniquement les pointeurs `void` , donc pour imprimer la valeur d'un pointeur non `void` il doit être explicitement converti ("casted *") en `void*` .

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>  /* for printf() */

int main(void)
{
  int i;
  int * p = &i;

  printf("The address of i is %p.\n", (void*) p);

  return EXIT_SUCCESS;
}
```

C99

Utiliser `<inttypes.h>` et `uintptr_t`

Une autre manière d’imprimer des pointeurs en C99 ou version ultérieure utilise le type `uintptr_t` et les macros de `<inttypes.h>` :

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h>    /* for printf() */

int main(void)
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

  return 0;
}
```

En théorie, il pourrait ne pas y avoir de type entier pouvant contenir un pointeur converti en entier (de sorte que le type `uintptr_t` pourrait ne pas exister). En pratique, il existe. Les pointeurs vers les fonctions ne doivent pas nécessairement être convertibles au type `uintptr_t` , même si, à nouveau, ils sont le plus souvent convertibles.

Si le type `uintptr_t` existe, il en va de même pour le type `intptr_t` . Il n'est pas clair pourquoi vous

voudriez jamais traiter les adresses comme des entiers signés, cependant.

K & R C89

Histoire pré-standard:

Avant C89 pendant K & R fois-C il n'y avait pas de type void* (ni en- tête <stdlib.h> , ni prototypes, et donc pas d' int main(void) notation), de sorte que le pointeur a été jeté à long unsigned int et imprimé à l' aide du lx longueur modificateur / spécificateur de conversion.

L'exemple ci-dessous est juste à titre informatif. De nos jours, ce code est invalide, ce qui pourrait très bien provoquer le fameux comportement indéfini .

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%lx.\n", (long unsigned) p);

  return 0;
}
```

Impression de la différence des valeurs de deux pointeurs sur un objet

Soustraire les valeurs de deux pointeurs à un objet donne un entier signé [* 1] . Donc, il serait imprimé en utilisant *au moins* le spécificateur de conversion d .

Pour vous assurer qu'il existe un type suffisamment large pour contenir une telle "différence de pointeur", puisque C99 <stddef.h> définit le type ptrdiff_t . Pour imprimer un ptrdiff_t utilisez le modificateur de longueur t .

C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */


int main(void)
{
  int a[2];
  int * p1 = &a[0], * p2 = &a[1];
  ptrdiff_t pd = p2 - p1;

  printf("p1 = %p\n", (void*) p1);
  printf("p2 = %p\n", (void*) p2);
  printf("p2 - p1 = %td\n", pd);

  return EXIT_SUCCESS;
}
```

Le résultat pourrait ressembler à ceci:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Veuillez noter que la valeur résultante de la différence est mise à l'échelle en fonction de la taille du type sur lequel les pointeurs ont soustrait le point, un `int` ici. La taille d'un `int` pour cet exemple est 4.

---

* [1] Si les deux pointeurs à soustraire ne pointent pas sur le même objet, le comportement est indéfini.

Spécificateurs de conversion pour l'impression

Spécificateur de conversion	Type d'argument	La description
`i` , `d`	int	imprime la décimale
`u`	unsigned int	imprime la décimale
`o`	unsigned int	imprime octal
`x`	unsigned int	imprime hexadécimal, minuscule
`X`	unsigned int	imprime hexadécimal, majuscule
`f`	double	les impressions flottent avec une précision par défaut de 6, si aucune précision n'est donnée (minuscules utilisées pour les nombres spéciaux `nan` et `inf` ou `infinity`)
`F`	double	les impressions flottent avec une précision par défaut de 6, si aucune précision n'est donnée (majuscule utilisée pour les numéros spéciaux `NAN` et `INF` ou `INFINITY`)
`e`	double	les impressions flottent avec une précision par défaut de 6, si aucune précision n'est donnée, en utilisant une notation scientifique utilisant mantisse / exposant; exposant minuscule et numéros spéciaux
`E`	double	les impressions flottent avec une précision par défaut de 6, si aucune précision n'est donnée, en utilisant une notation scientifique utilisant mantisse / exposant; exposant en majuscule et numéros spéciaux
`g`	double	utilise soit `f` ou `e` *[voir ci-dessous]*

Spécificateur de conversion	Type d'argument	La description
`G`	double	utilise soit `F` soit `E` *[voir ci-dessous]*
`a`	double	imprime hexadécimal, minuscule
`A`	double	imprime hexadécimal, majuscule
`c`	carboniser	imprime un caractère unique
`s`	carboniser*	imprime une chaîne de caractères jusqu'à un terminateur `NUL` , ou tronquée à la longueur donnée par précision, si spécifiée
`p`	vide*	imprime la `void` pointeur `void` ; un pointeur non `void` doit être explicitement converti ("cast") en `void*` ; pointeur vers un objet uniquement, **pas** un pointeur de fonction
`%`	n / a	imprime le caractère `%`
`n`	int *	***écrivez*** le nombre d'octets imprimés jusqu'ici dans l' `int` pointé.

Noter que les modificateurs de longueur peuvent être appliqués à `%n` (par exemple, `%hhn` indique qu'un *spécificateur de conversion `n` suivant s'applique à un pointeur sur un argument de caractère `signed char`* , conformément à la norme ISO / IEC 9899: 2011, § 7.21.6.1 ¶7).

Notez que les conversions à virgule flottante appliquées aux types `float` et `double` raison de règles de promotion par défaut - §6.5.2.2 Appels de fonction, ¶7 *La notation par points de suspension dans un déclarant de prototype de fonction entraîne l'arrêt de la conversion après le dernier paramètre déclaré. Les promotions d'argument par défaut sont effectuées sur les arguments de fin.*) Ainsi, les fonctions telles que `printf()` ne sont transmises qu'à `double` valeurs `double` , même si la variable référencée est de type `float` .

Avec les formats `g` et `G` , le choix entre la notation `e` et `f` (ou `E` et `F`) est documenté dans le standard C et dans la spécification POSIX pour [`printf()`] :

> Le double argument représentant un nombre à virgule flottante doit être converti dans le style `f` ou `e` (ou dans le style `F` ou `E` dans le cas d'un spécificateur de conversion `G`), en fonction de la valeur convertie et de la précision. Soit `P` la précision si elle est égale à zéro, 6 si la précision est omise ou 1 si la précision est égale à zéro. Alors, si une conversion avec le style `E` aurait un exposant de `X` :
>
> - Si P> X> = -4, la conversion doit être avec le style `f` (ou `F`) et la précision `P - (X+1)` .
> - Sinon, la conversion doit se faire avec le style `e` (ou `E`) et la précision `P - 1` .
>
> Enfin, à moins que le drapeau "#" ne soit utilisé, tous les zéros à la fin seront

supprimés de la partie fractionnaire du résultat et le caractère décimal sera supprimé s'il ne reste aucune partie fractionnaire.

La fonction printf ()

Accessible via y compris `<stdio.h>` , la fonction `printf()` est l'outil principal utilisé pour imprimer du texte sur la console en C.

```
printf("Hello world!");
// Hello world!
```

Les tableaux de caractères normaux et non formatés peuvent être imprimés par eux-mêmes en les plaçant directement entre les parenthèses.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.

int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Vous pouvez également imprimer des nombres entiers, des nombres à virgule flottante, des caractères et autres en utilisant le caractère d'échappement `%` , suivi d'un caractère ou d'une séquence de caractères indiquant le format, appelé *spécificateur de format* .

Tous les arguments supplémentaires de la fonction `printf()` sont séparés par des virgules et ces arguments doivent être dans le même ordre que les spécificateurs de format. Les arguments supplémentaires sont ignorés, tandis que les arguments mal typés ou l'absence d'arguments provoquent des erreurs ou un comportement indéfini. Chaque argument peut être une valeur littérale ou une variable.

Après une exécution réussie, le nombre de caractères imprimés est renvoyé avec le type `int` . Sinon, un échec renvoie une valeur négative.

Modificateurs de longueur

Les normes C99 et C11 spécifient les modificateurs de longueur suivants pour `printf()` ; leurs significations sont:

Modificateur	Modifie	S'applique à
hh	d, i, o, u, x ou x	`char` , `signed char` ou `unsigned char`
h	d, i, o, u, x ou x	`short int` ou `unsigned short int`
l	d, i, o, u, x ou x	`long int` ou `unsigned long int`
l	a, A, e, E, f, F, g ou	`double` (pour compatibilité avec `scanf()` ; indéfini dans

Modificateur	Modifie	S'applique à
	G	C90)
ll	d, i, o, u, x ou x	`long long int` ou `unsigned long long int`
j	d, i, o, u, x ou x	`intmax_t` ou `uintmax_t`
z	d, i, o, u, x ou x	`size_t` ou le type signé correspondant (`ssize_t` dans POSIX)
t	d, i, o, u, x ou x	`ptrdiff_t` ou le type d'entier non signé correspondant
L	a, A, e, E, f, F, g ou G	`long double`

Si un modificateur de longueur apparaît avec un spécificateur de conversion autre que celui spécifié ci-dessus, le comportement est indéfini.

Microsoft spécifie des modificateurs de longueur différents et ne prend pas explicitement en charge `hh` , `j` , `z` ou `t` .

Modificateur	Modifie	S'applique à
I32	d, i, o, x ou x	`__int32`
I32	o, u, x ou x	`unsigned __int32`
I64	d, i, o, x ou x	`__int64`
I64	o, u, x ou x	`unsigned __int64`
je	d, i, o, x ou X	`ptrdiff_t` (c'est-à-dire `__int32` sur les plates-formes 32 bits, `__int64` sur les plates-formes 64 bits)
je	o, u, x ou x	`size_t` (c'est-à-dire `unsigned __int32` sur les plates-formes 32 bits, `unsigned __int64` sur les plates-formes 64 bits)
l ou l	a, A, e, E, f, g ou G	`long double` (Dans Visual C ++, bien que `long double` soit un type distinct, il a la même représentation interne que le `double` .)
l ou w	c ou c	Caractère large avec les fonctions `printf` et `wprintf` . (Un `wC` type `lc` , `lC` , `wc` ou `wC` est synonyme de `C` dans les fonctions `printf` et `c` dans les fonctions `wprintf` .)
l ou w	s, S ou Z	Chaîne de caractères large avec les fonctions `printf` et `wprintf` . (Un `wS` type `ls` , `lS` , `ws` ou `wS` est synonyme de `S` dans les fonctions `printf` et avec `s` dans les fonctions `wprintf` .)

Notez que les spécificateurs de conversion `C` , `S` et `Z` et les modificateurs de longueur `I` , `I32` , `I64` et `w` sont des extensions Microsoft. Traiter `l` comme un modificateur du `long double` plutôt que du `double` est différent du standard, mais vous aurez du mal à repérer la différence à moins que le `long double` ait une représentation différente du `double` .

Drapeaux de format d'impression

Le standard C (C11 et C99) définit les indicateurs suivants pour `printf()` :

Drapeau	Conversions	Sens
`-`	tout	Le résultat de la conversion doit être justifié à gauche dans le champ. La conversion est justifiée à droite si cet indicateur n'est pas spécifié.
`+`	numérique signé	Le résultat d'une conversion signée doit toujours commencer par un signe («+» ou «-»). La conversion ne doit commencer par un signe que si une valeur négative est convertie si cet indicateur n'est pas spécifié.
`<space>`	numérique signé	Si le premier caractère d'une conversion signée n'est pas un signe ou si une conversion signée ne génère aucun caractère, un `<space>` doit être préfixé au résultat. Cela signifie que si les indicateurs `<space>` et ' `+` ' apparaissent tous deux, le drapeau `<space>` doit être ignoré.
`#`	tout	Indique que la valeur doit être convertie en un autre formulaire. Pour `o` conversion, il faut augmenter la précision, si et seulement si nécessaire, pour forcer le premier chiffre du résultat à être un zéro (si la valeur et la précision sont toutes les deux 0, un seul 0 est imprimé). Pour les spécificateurs de conversion `x` ou `X` , `0x` (ou `0X`) doit être précédé d'un résultat non nul. Pour `a` , `A` , `e` , `E` , `f` , `F` , `g` et `G` spécificateurs de conversion, le résultat contient toujours un caractère radix, même si aucun chiffre suivent le caractère radix. Sans ce drapeau, un caractère de base apparaît dans le résultat de ces conversions uniquement si un chiffre le suit. Pour les spécificateurs de conversion `g` et `G` , les zéros à la fin ne doivent pas être supprimés du résultat normalement. Pour les autres spécificateurs de conversion, le comportement est indéfini.
`0`	numérique	Pour les spécificateurs de conversion d, i, o, u, x, X, a, A, e, E, f, F, g et G, les zéros en tête (à la suite de toute indication de signe ou de base) sont utilisés pour le champ width plutôt que d'effectuer un remplissage d'espace, sauf lors de la conversion d'un infini ou d'un NaN. Si les drapeaux '0' et '-' apparaissent tous les deux, le drapeau '0' est ignoré. Pour les spécificateurs de conversion d, i, o, u, x et X, si une précision est spécifiée, le

Drapeau	Conversions	Sens
		drapeau '0' doit être ignoré. Si les indicateurs '0' et `<apostrophe>` apparaissent tous les deux, les caractères de regroupement sont insérés avant le remplissage zéro. Pour les autres conversions, le comportement est indéfini.

Ces indicateurs sont également pris en charge par Microsoft avec les mêmes significations.

La spécification POSIX pour `printf()` ajoute:

Drapeau	Conversions	Sens
'	i, d, u, f, f, g, g	La partie entière du résultat d'une conversion décimale doit être formatée avec des milliers de caractères de regroupement. Pour les autres conversions, le comportement n'est pas défini. Le caractère de regroupement non monétaire est utilisé.

Lire Entrée / sortie formatée en ligne: https://riptutorial.com/fr/c/topic/3750/entree---sortie-formatee

Chapitre 25: Énumérations

Remarques

Les énumérations se composent du mot clé `enum` et d'un *identificateur* facultatif suivi d'une *liste d'énumérateurs* entourée d'accolades.

Un *identifiant* est de type `int` .

La *liste d'énumérateur* contient au moins un élément *énumérateur* .

Un *énumérateur* peut éventuellement être "affecté" à une expression constante de type `int` .

Un *énumérateur* est constante et est compatible avec soit une `char` , un entier signé ou un entier non signé. Ce qui est utilisé est défini par la mise en œuvre . Dans tous les cas, le type utilisé doit pouvoir représenter toutes les valeurs définies pour l'énumération en question.

Si aucune expression constante n'est "affectée" à un *énumérateur* et qu'il s'agit de la 1ère entrée d'une *liste d'énumérateurs,* elle prend la valeur `0` , sinon elle prend la valeur de l'entrée précédente dans la *liste d'énumérateur* plus 1.

L'utilisation de plusieurs «affectations» peut conduire à différents *énumérateurs* de la même énumération portant les mêmes valeurs.

Examples

Énumération simple

Une énumération est un type de données défini par l'utilisateur constitué de constantes intégrales et chaque constante intégrale reçoit un nom. Le mot clé `enum` est utilisé pour définir le type de données énuméré.

Si vous utilisez `enum` au lieu de `int` ou `string/ char*` , vous augmentez la vérification au moment de la compilation et évitez les erreurs de transmission de constantes non valides, et vous documentez les valeurs légales à utiliser.

Exemple 1

```
enum color{ RED, GREEN, BLUE };

void printColor(enum color chosenColor)
{
    const char *color_name = "Invalid color";
    switch (chosenColor)
    {
       case RED:
         color_name = "RED";
```

```
         break;

       case GREEN:
        color_name = "GREEN";
        break;

       case BLUE:
        color_name = "BLUE";
        break;
    }
    printf("%s\n", color_name);
}
```

Avec une fonction principale définie comme suit (par exemple):

```
int main(){
    enum color chosenColor;
    printf("Enter a number between 0 and 2");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

C99

Exemple 2

(Cet exemple utilise des initialiseurs désignés qui sont standardisés depuis C99.)

```
enum week{ MON, TUE, WED, THU, FRI, SAT, SUN };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   printf("%s\n", dow[day]);
}
```

Le même exemple utilisant la vérification des plages:

```
enum week{ DOW_INVALID = -1,
  MON, TUE, WED, THU, FRI, SAT, SUN,
  DOW_MAX };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   assert(day > DOW_INVALID && day < DOW_MAX);
   printf("%s\n", dow[day]);
}
```

Typedef enum

Il existe plusieurs possibilités et conventions pour nommer une énumération. La première consiste à utiliser un *nom de tag* juste après le mot-clé `enum` .

```
enum color
{
    RED,
    GREEN,
    BLUE
};
```

Cette énumération doit alors toujours être utilisée avec le mot-clé *et* la balise comme ceci:

```
enum color chosenColor = RED;
```

Si nous utilisons directement `typedef` lors de la déclaration de l' `enum` , nous pouvons omettre le nom de la balise, puis utiliser le type sans le mot clé `enum` :

```
typedef enum
{
    RED,
    GREEN,
    BLUE
} color;

color chosenColor = RED;
```

Mais dans ce dernier cas, nous ne pouvons pas l'utiliser en tant que `enum color` , car nous n'avons pas utilisé le nom de tag dans la définition. Une convention courante consiste à utiliser les deux, de sorte que le même nom puisse être utilisé avec ou sans mot-clé `enum` . Cela présente l'avantage particulier d'être compatible avec C ++

```
enum color                /* as in the first example */
{
    RED,
    GREEN,
    BLUE
};
typedef enum color color; /* also a typedef of same identifier */

color chosenColor  = RED;
enum color defaultColor = BLUE;
```

Fonction:

```
void printColor()
{
    if (chosenColor == RED)
    {
        printf("RED\n");
    }
    else if (chosenColor == GREEN)
```

```
    {
        printf("GREEN\n");
    }
    else if (chosenColor == BLUE)
    {
        printf("BLUE\n");
    }
}
```

Pour plus d'informations sur `typedef` voir Typedef

Enumération avec valeur en double

Une valeur d'énumération ne doit en aucun cas être unique:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */


enum Dupes
{
   Base, /* Takes 0 */
   One, /* Takes Base + 1 */
   Two, /* Takes One + 1 */
   Negative = -1,
   AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
  printf("Base = %d\n", Base);
  printf("One = %d\n", One);
  printf("Two = %d\n", Two);
  printf("Negative = %d\n", Negative);
  printf("AnotherZero = %d\n", AnotherZero);

  return EXIT_SUCCESS;
}
```

L'échantillon imprime:

```
Base = 0
One = 1
Two = 2
Negative = -1
AnotherZero = 0
```

énumération constante sans nom de type

Les types d'énumération peuvent également être déclarés sans leur donner un nom:

```
  enum { buffersize = 256, };
  static unsigned char buffer [buffersize] = { 0 };
```

Cela nous permet de définir des constantes de temps de compilation de type `int` qui peuvent,

comme dans cet exemple, être utilisées comme longueur de tableau.

Lire Énumérations en ligne: https://riptutorial.com/fr/c/topic/5460/enumerations

Chapitre 26: Fichiers et flux d'E / S

Syntaxe

- #include <stdio.h> / * Inclure ceci pour utiliser l'une des sections suivantes * /
- FILE * fopen (const char * path, const char * mode); / * Ouvre un flux sur le fichier au *chemin* avec le *mode* spécifié * /
- FILE * freopen (const char * path, const char * mode, FILE * stream); / * Rouvre un flux existant sur le fichier au *chemin* avec le *mode* spécifié * /
- int fclose (flux FILE *); / * Fermer un flux ouvert * /
- size_t fread (void * ptr, taille_t taille, size_t nmemb, flux FILE *); / * Lit à la plupart des éléments *nmemb* de *taille* chacun des octets du *flux* et les écrit dans *ptr* . Renvoie le nombre d'éléments de lecture. * /
- size_t fwrite (const void * ptr, taille_t taille, size_t nmemb, flux FILE *); / * *Ecrit des* éléments *nmemb* de *taille* chacun de *ptr* dans le *flux* . Renvoie le nombre d'éléments écrits. * /
- int fseek (flux FILE *, offset long, int dont); / * Réglez le curseur du flux pour *compenser,* par rapport au décalage dit par *où,* et retourne 0 si elle a réussi. * /
- long ftell (flux FILE *); / * Renvoie le décalage de la position actuelle du curseur depuis le début du flux. * /
- annuler le rembobinage (flux FILE *); / * Définit la position du curseur au début du fichier. * /
- int fprintf (FILE * fout, const char * fmt, ...); / * Ecrit la chaîne de format printf sur fout * /
- FICHIER * stdin; / * Flux d'entrée standard * /
- FILE * stdout; / * Flux de sortie standard * /
- FICHIER * stderr; / * Flux d'erreur standard * /

Paramètres

Paramètre	Détails
mode const char *	Une chaîne décrivant le mode d'ouverture du flux sauvegardé. Voir les remarques pour les valeurs possibles.
d'où	Peut être `SEEK_SET` à définir depuis le début du fichier, `SEEK_END` à définir depuis sa fin ou `SEEK_CUR` à définir par rapport à la valeur actuelle du curseur. Remarque: `SEEK_END` est non portable.

Remarques

Chaînes de mode:

Les chaînes de mode dans `fopen()` et `freopen()` peuvent être l'une de ces valeurs:

- `"r"` : ouvre le fichier en mode lecture seule, le curseur étant placé au début du fichier.

- `"r+"` : Ouvrez le fichier en mode lecture-écriture, avec le curseur défini au début du fichier.
- `"w"` : Ouvrez ou créez le fichier en mode écriture seule, son contenu étant tronqué à 0 octet. Le curseur est défini au début du fichier.
- `"w+"` : Ouvrez ou créez le fichier en mode lecture-écriture, avec son contenu tronqué à 0 octet. Le curseur est défini au début du fichier.
- `"a"` : Ouvrez ou créez le fichier en mode écriture seule, le curseur étant placé à la fin du fichier.
- `"a+"` : ouvrez ou créez le fichier en mode lecture-écriture, le curseur de lecture étant placé au début du fichier. La sortie, cependant, sera *toujours* ajoutée à la fin du fichier.

Chacun de ces modes peut avoir un `b` ajouté après la lettre initiale (par exemple `"rb"` ou `"a+b"` ou `"ab+"`). Le `b` signifie que le fichier doit être traité comme un fichier binaire au lieu d'un fichier texte sur les systèmes où il y a une différence. Cela ne fait aucune différence sur les systèmes de type Unix. c'est important sur les systèmes Windows. (De plus, Windows `fopen` permet un `t` explicite au lieu de `b` pour indiquer «fichier texte» - et de nombreuses autres options spécifiques à la plate-forme.)

C11

- `"wx"` : Créez un fichier texte en mode écriture seule. *Le fichier n'existe peut-être pas* .
- `"wbx"` : Créez un fichier binaire en mode écriture seule. *Le fichier n'existe peut-être pas* .

Le `x` , s'il est présent, doit être le dernier caractère de la chaîne de mode.

Examples

Ouvrir et écrire dans un fichier

```
#include <stdio.h>   /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h>  /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
```

```
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

Ce programme ouvre le fichier avec le nom donné dans l'argument à main, par défaut à `output.txt` si aucun argument n'est donné. Si un fichier portant le même nom existe déjà, son contenu est supprimé et le fichier est traité comme un nouveau fichier vide. Si les fichiers n'existent pas déjà, l'appel `fopen()` le crée.

Si l'appel `fopen()` échoue pour une raison quelconque, il renvoie une valeur `NULL` et définit la valeur de la variable `errno` globale. Cela signifie que le programme peut tester la valeur renvoyée après l'appel à `fopen()` et utiliser `perror()` si `fopen()` échoue.

Si l'appel `fopen()` réussit, il renvoie un pointeur `FILE` valide. Ce pointeur peut alors être utilisé pour référencer ce fichier jusqu'à ce que `fclose()` soit appelé.

La fonction `fputs()` écrit le texte donné dans le fichier ouvert, remplaçant tout contenu précédent du fichier. Comme pour `fopen()` , la fonction `fputs()` définit également la valeur `errno` si elle échoue, bien que dans ce cas la fonction retourne `EOF` pour indiquer l'échec (sinon, elle renvoie une valeur non négative).

La fonction `fclose()` vide les tampons, ferme le fichier et libère la mémoire pointée par `FILE *` . La valeur de retour indique l'achèvement de la même manière que `fputs()` (bien qu'elle renvoie "0" si elle réussit), définissant également la valeur `errno` en cas d'échec.

fprintf

Vous pouvez utiliser `fprintf` sur un fichier comme vous le feriez sur une console avec `printf` . Par exemple, pour garder une trace des gains, des pertes et des cravates, vous pouvez écrire

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);
}
```

Une note de côté: certains systèmes (Windows, notoirement) n'utilisent pas ce que la plupart des programmeurs appellent des fins de ligne "normales". Bien que les systèmes de type UNIX utilisent \ n pour terminer les lignes, Windows utilise une paire de caractères: \ r (retour chariot) et \ n (saut de ligne). Cette séquence est communément appelée CRLF. Cependant, chaque fois que vous utilisez C, vous n'avez pas à vous soucier de ces détails hautement dépendants de la plate-forme. Le compilateur AC est requis pour convertir chaque instance de \ n en une terminaison de ligne correcte. Ainsi, un compilateur Windows convertirait \ n en \ r \ n, mais un compilateur UNIX

le conserverait tel quel.

Exécuter le processus

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}
int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

Ce programme exécute un processus (`netstat`) via `popen()` et lit toute la sortie standard du processus et fait écho à la sortie standard.

Note: `popen()` n'existe pas dans la bibliothèque C standard , mais il fait plutôt partie de POSIX C)

Récupère les lignes d'un fichier en utilisant getline ()

La bibliothèque POSIX C définit la fonction `getline()` . Cette fonction alloue un tampon pour contenir le contenu de la ligne et renvoie la nouvelle ligne, le nombre de caractères de la ligne et la taille du tampon.

Exemple de programme qui obtient chaque ligne de `example.txt` :

```
#include <stdlib.h>
#include <stdio.h>


#define FILENAME "example.txt"

int main(void)
{
  /* Open the file for reading */
  char *line_buf = NULL;
  size_t line_buf_size = 0;
  int line_count = 0;
  ssize_t line_size;
  FILE *fp = fopen(FILENAME, "r");
  if (!fp)
  {
    fprintf(stderr, "Error opening file '%s'\n", FILENAME);
    return EXIT_FAILURE;
```

```
  }

  /* Get the first line of the file. */
  line_size = getline(&line_buf, &line_buf_size, fp);

  /* Loop through until we are done with the file. */
  while (line_size >= 0)
  {
    /* Increment our line count */
    line_count++;

    /* Show the line details */
    printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
        line_size, line_buf_size, line_buf);

    /* Get the next line */
    line_size = getline(&line_buf, &line_buf_size, fp);
  }

  /* Free the allocated line buffer */
  free(line_buf);
  line_buf = NULL;

  /* Close the file now that we are done with it */
  fclose(fp);

  return EXIT_SUCCESS;
}
```

Fichier d'entrée `example.txt`

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that getline() will reallocate the line buffer if the length of a
line is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Sortie

```
line[000001]: chars=000015, buf size=000016, contents: This is a file
line[000002]: chars=000012, buf size=000016, contents:   which has
line[000003]: chars=000015, buf size=000016, contents: multiple lines
line[000004]: chars=000030, buf size=000032, contents:     with various indentation,
line[000005]: chars=000012, buf size=000032, contents: blank lines
line[000006]: chars=000001, buf size=000032, contents:
line[000007]: chars=000001, buf size=000032, contents:
line[000008]: chars=000001, buf size=000032, contents:
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that
getline() will reallocate the line buffer if the length of a line is too long to fit in the
buffer it has been given,
line[000010]: chars=000042, buf size=000160, contents:  and punctuation at the end of the
```

```
lines.
line[000011]: chars=000001, buf size=000160, contents:
```

Dans l'exemple, `getline()` est initialement appelée sans tampon alloué. Lors de ce premier appel, `getline()` alloue un tampon, lit la première ligne et place le contenu de la ligne dans le nouveau tampon. Lors d'appels ultérieurs, `getline()` met à jour le même tampon et ne réalloue le tampon que lorsqu'il n'est plus assez grand pour s'adapter à toute la ligne. Le tampon temporaire est alors libéré lorsque nous en avons fini avec le fichier.

Une autre option est `getdelim()` . C'est la même chose que `getline()` sauf que vous spécifiez le caractère de fin de ligne. Ceci n'est nécessaire que si le dernier caractère de la ligne pour votre type de fichier n'est pas "\ n". `getline()` fonctionne même avec les fichiers texte Windows car avec la fin de la ligne multi-octets (`"\r\n")` '" est toujours le dernier caractère de la ligne.

Exemple d'implémentation de `getline()`

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <stdint.h>


#if !(defined _POSIX_C_SOURCE)
typedef long int ssize_t;
#endif

/* Only include our version of getline() if the POSIX version isn't available. */

#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L

#if !(defined SSIZE_MAX)
#define SSIZE_MAX (SIZE_MAX >> 1)
#endif

ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)
{
  const size_t INITALLOC = 16;
  const size_t ALLOCSTEP = 16;
  size_t num_read = 0;

  /* First check that none of our input pointers are NULL. */
  if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))
  {
    errno = EINVAL;
    return -1;
  }

  /* If output buffer is NULL, then allocate a buffer. */
  if (NULL == *pline_buf)
  {
    *pline_buf = malloc(INITALLOC);
    if (NULL == *pline_buf)
    {
      /* Can't allocate memory. */
      return -1;
    }
```

```
    else
    {
      /* Note how big the buffer is at this time. */
      *pn = INITALLOC;
    }
  }

  /* Step through the file, pulling characters until either a newline or EOF. */

  {
    int c;
    while (EOF != (c = getc(fin)))
    {
      /* Note we read a character. */
      num_read++;

      /* Reallocate the buffer if we need more room */
      if (num_read >= *pn)
      {
        size_t n_realloc = *pn + ALLOCSTEP;
        char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
        if (NULL != tmp)
        {
          /* Use the new buffer and note the new buffer size. */
          *pline_buf = tmp;
          *pn = n_realloc;
        }
        else
        {
          /* Exit with error and let the caller free the buffer. */
          return -1;
        }

        /* Test for overflow. */
        if (SSIZE_MAX < *pn)
        {
          errno = ERANGE;
          return -1;
        }
      }

      /* Add the character to the buffer. */
      (*pline_buf)[num_read - 1] = (char) c;

      /* Break from the loop if we hit the ending character. */
      if (c == '\n')
      {
        break;
      }
    }

    /* Note if we hit EOF. */
    if (EOF == c)
    {
      errno = 0;
      return -1;
    }
  }

  /* Terminate the string by suffixing NUL. */
  (*pline_buf)[num_read] = '\0';
```

```
  return (ssize_t) num_read;
}

#endif
```

Ouvrir et écrire dans un fichier binaire

```
#include <stdlib.h>
#include <stdio.h>


int main(void)
{
   result = EXIT_SUCCESS;

   char file_name[] = "outbut.bin";
   char str[] = "This is a binary file example";
   FILE * fp = fopen(file_name, "wb");

   if (fp == NULL)  /* If an error occurs during the file creation */
   {
     result = EXIT_FAILURE;
     fprintf(stderr, "fopen() failed for '%s'\n", file_name);
   }
   else
   {
     size_t element_size = sizeof *str;
     size_t elements_to_write = sizeof str;

     /* Writes str (_including_ the NUL-terminator) to the binary file. */
     size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
     if (elements_written != elements_to_write)
     {
       result = EXIT_FAILURE;
       /* This works for >=c99 only, else the z length modifier is unknown. */
       fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
         elements_written, elements_to_write);
       /* Use this for <c99: *
       fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
         (unsigned long) elements_written, (unsigned long) elements_to_write);
        */
     }

     fclose(fp);
   }

   return result;
}
```

Ce programme crée et écrit du texte sous la forme binaire via la fonction `fwrite` dans le fichier `output.bin` .

Si un fichier portant le même nom existe déjà, son contenu est supprimé et le fichier est traité comme un nouveau fichier vide.

Un flux binaire est une séquence ordonnée de caractères pouvant enregistrer de manière

transparente des données internes. Dans ce mode, les octets sont écrits entre le programme et le fichier sans aucune interprétation.

Pour écrire des entiers de manière portable, il faut savoir si le format de fichier les attend au format big ou little-endian, et à la taille (généralement 16, 32 ou 64 bits). Le décalage et le masquage des bits peuvent alors être utilisés pour écrire les octets dans le bon ordre. Les nombres entiers dans C ne sont pas garantis d'avoir une représentation de complément à deux (bien que presque toutes les implémentations le soient). Heureusement une conversion non signée *est* garantie à utiliser complément à deux. Le code pour écrire un entier signé dans un fichier binaire est donc un peu surprenant.

```
/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}
```

Les autres fonctions suivent le même schéma avec des modifications mineures de la taille et de l'ordre des octets.

fscanf ()

Disons que nous avons un fichier texte et que nous voulons lire tous les mots de ce fichier, afin de satisfaire certaines exigences.

fichier.txt :

```
This is just
a test file
to be used by fscanf()
```

Ceci est la fonction principale:

```
#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
  FILE *fp;

  if ((fp = fopen("file.txt", "r")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE);
```

```
    }

    printAllWords(fp);

    fclose(fp);

    return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
    char tmp[20];
    int i = 1;

    while (fscanf(fp, "%19s", tmp) != EOF) {
        printf("Word %d: %s\n", i, tmp);
        i++;
    }
}
```

Le résultat sera:

```
Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
Word 10: by
Word 11: fscanf()
```

Lire les lignes d'un fichier

L'en-tête `stdio.h` définit la fonction `fgets()` . Cette fonction lit une ligne d'un flux et la stocke dans une chaîne spécifiée. La fonction arrête de lire le texte du flux lorsque `n - 1` caractères sont lus, le caractère de nouvelle ligne (`'\n'`) est lu ou la fin du fichier (EOF) est atteinte.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
```

```
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
    }
}
```

Appel du programme avec un argument qui est un chemin vers un fichier contenant le texte suivant:

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that the line will be counted as two lines if the length of a line
is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Entraînera la sortie suivante:

```
line[000001]: This is a file
line[000002]:   which has
line[000003]: multiple lines
line[000004]:     with various indentation,
line[000005]: blank lines
line[000006]:
line[000007]:
line[000008]:
line[000009]: a really long line to show that the line will be counted as two lines if the le
line[000010]: ngth of a line is too long to fit in the buffer it has been given,
line[000011]:  and punctuation at the end of the lines.
line[000012]:
```

Cet exemple très simple permet une longueur de ligne maximale fixe, de sorte que les lignes plus

longues seront effectivement comptabilisées comme deux lignes. La fonction `fgets()` nécessite que le code appelant fournisse la mémoire à utiliser comme destination pour la ligne lue.

POSIX met à disposition la fonction `getline()` qui alloue en interne de la mémoire pour agrandir le tampon selon les besoins pour une ligne de longueur quelconque (tant qu'il y a suffisamment de mémoire).

Lire Fichiers et flux d'E / S en ligne: https://riptutorial.com/fr/c/topic/507/fichiers-et-flux-d-e---s

Chapitre 27: Génération de nombres aléatoires

Remarques

En raison des failles de `rand()` , de nombreuses autres implémentations par défaut sont apparues au fil des ans. Parmi ceux-ci sont:

- `arc4random()` (disponible sur OS X et BSD)
- `random()` (disponible sous Linux)
- `drand48()` (disponible sur POSIX)

Examples

Génération de nombres aléatoires de base

La fonction `rand()` peut être utilisée pour générer un nombre entier pseudo-aléatoire compris entre `0` et `RAND_MAX` (`0` et `RAND_MAX` inclus).

`srand(int)` est utilisé pour générer le générateur de nombres pseudo-aléatoires. Chaque fois que `rand()` est ensemencé avec la même graine, il doit produire la même séquence de valeurs. Il ne devrait être utilisé qu'une fois avant d'appeler `rand()` . Elle ne doit pas être répétée de manière répétée ou ré-émise chaque fois que vous souhaitez générer un nouveau lot de nombres pseudo-aléatoires.

La pratique standard consiste à utiliser le résultat du `time(NULL)` comme graine. Si votre générateur de nombres aléatoires doit avoir une séquence déterministe, vous pouvez affecter le générateur avec la même valeur à chaque démarrage du programme. Ceci n'est généralement pas requis pour le code de version, mais est utile dans les exécutions de débogage pour rendre les bogues reproductibles.

Il est conseillé de toujours semer le générateur, s’il n’est pas ensemencé, il se comporte comme s’il était ensemencé de `srand(1)` .

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Sortie possible:

```
Random value between [0, 2147483647]: 823321433
```

Remarques:

La norme C ne garantit pas la qualité de la séquence aléatoire produite. Dans le passé, certaines implémentations de `rand()` avaient de sérieux problèmes de distribution et de caractère aléatoire des nombres générés. **L'utilisation de `rand()` n'est pas recommandée pour les besoins de génération de nombres aléatoires graves, comme la cryptographie.**

Génératrice à permutation permutée

Voici un générateur de nombres aléatoires autonome qui ne repose pas sur `rand()` ou des fonctions de bibliothèque similaires.

Pourquoi voudriez-vous une telle chose? Peut-être que vous ne faites pas confiance au générateur de nombres aléatoires intégré à votre plate-forme, ou peut-être souhaitez-vous une source reproductible de caractère aléatoire indépendante de toute implémentation de bibliothèque particulière.

Ce code est PCG32 de pcg-random.org , un RNG moderne, rapide et polyvalent doté d'excellentes propriétés statistiques. Ce n'est pas cryptographiquement sécurisé, donc ne l'utilisez pas pour la cryptographie.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */

typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}
```

Et voici comment l'appeler:

```
#include <stdio.h>
```

```
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;

    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}
```

Restreindre la génération à une plage donnée

Généralement, lors de la génération de nombres aléatoires, il est utile de générer des nombres entiers compris dans une plage ou une valeur entre 0,0 et 1,0. Bien que le module puisse être utilisé pour réduire la graine à un nombre entier faible, il utilise les bits bas, qui passent souvent par un cycle court, ce qui entraîne un léger déséquilibre de la distribution si N est important par rapport à RAND_MAX.

La macro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produit une valeur de 0.0 à 1.0 - epsilon, donc

```
i = (int)(uniform() * N)
```

fixera `i` à un nombre aléatoire uniforme compris entre 0 et N - 1.

Malheureusement, il y a un défaut technique, car RAND_MAX peut être plus grand qu'une variable de type `double` peut représenter avec précision. Cela signifie que `RAND_MAX + 1.0` évalué à RAND_MAX et que la fonction retourne parfois l'unité. C'est peu probable cependant.

Génération Xorshift

Xorshift , une classe de générateurs de nombres pseudo-aléatoires découverts par George Marsaglia , constitue une *alternative intéressante aux* procédures `rand()` erronées. Le générateur xorshift fait partie des générateurs de nombres aléatoires non cryptographiquement sécurisés les plus rapides. Plus d'informations et d'autres exemples d'implémentations sont disponibles sur la page Wikipedia de xorshift

Exemple d'implémentation

```
#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;
```

```
uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
    return w;
}
```

Lire Génération de nombres aléatoires en ligne: https://riptutorial.com/fr/c/topic/365/generation-de-nombres-aleatoires

Chapitre 28: Gestion de la mémoire

Introduction

Pour gérer la mémoire allouée dynamiquement, la bibliothèque C standard fournit les fonctions `malloc()` , `calloc()` , `realloc()` et `free()` . En C99 et `aligned_alloc()` ultérieures, il y a aussi `aligned_alloc()` . Certains systèmes fournissent également `alloca()` .

Syntaxe

- void * alignment_alloc (alignement size_t, taille size_t); / * Seulement depuis C11 * /
- void * calloc (size_t nelements, size_t size);
- vide gratuit (annulation * ptr);
- void * malloc (taille_taille);
- void * realloc (void * ptr, size_t size);
- void * alloca (taille_taille); / * from alloca.h, pas standard, pas portable, dangereux. * /

Paramètres

prénom	la description
taille (`malloc` , `realloc` et `aligned_alloc`)	taille totale de la mémoire en octets. Pour `aligned_alloc` la taille doit être un multiple entier de l'alignement.
taille (`calloc`)	taille de chaque élément
éléments	nombre d'éléments
ptr	pointeur sur la mémoire allouée précédemment renvoyée par `malloc` , `calloc` , `realloc` ou `aligned_alloc`
alignement	alignement de la mémoire allouée

Remarques

C11

Notez que `aligned_alloc()` est uniquement défini pour C11 ou version ultérieure.

Des systèmes tels que ceux basés sur POSIX fournissent d'autres moyens d'allouer de la mémoire alignée (par exemple, `posix_memalign()`), et ont également d'autres options de gestion de la mémoire (par exemple, `mmap()`).

Examples

Libérer de la mémoire

Il est possible de libérer de la mémoire allouée dynamiquement en appelant free () .

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

La mémoire pointée par `p` est récupérée (soit par l'implémentation de la libc ou par le système d'exploitation sous-jacent) après l'appel à `free()` , donc l'accès à ce bloc de mémoire libéré via `p` entraîne un comportement indéfini . Les pointeurs qui référencent des éléments de mémoire libérés sont communément appelés pointeurs , et présentent un risque de sécurité. De plus, le standard C stipule que même accéder à la valeur d'un pointeur en suspens a un comportement indéfini. Notez que le pointeur `p` lui-même peut être réutilisé comme indiqué ci-dessus.

Veuillez noter que vous ne pouvez appeler `free()` sur les pointeurs renvoyés directement par les fonctions `malloc()` , `calloc()` , `realloc()` et `aligned_alloc()` , ou lorsque la documentation vous indique que la mémoire a été allouée de cette manière (fonctions comme `strdup ()` sont des exemples notables). Libérer un pointeur qui est,

- obtenu en utilisant l'opérateur `&` sur une variable, ou
- au milieu d'un bloc alloué,

est interdit. Une telle erreur ne sera généralement pas diagnostiquée par votre compilateur mais conduira l'exécution du programme dans un état indéfini.

Il existe deux stratégies courantes pour prévenir de tels cas de comportement indéfini.

Le premier et préférable est simple: `p` lui-même cesse-t-il d'exister lorsqu'il n'est plus nécessaire, par exemple:

```
if (something_is_needed())
{

    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
        return -1;
```

```
    }

    /* do whatever is needed with p */

    free(p);
}
```

En appelant `free()` directement avant la fin du bloc contenant (c'est-à-dire le `}`), `p` cesse d'exister. Le compilateur donnera une erreur de compilation sur toute tentative d'utilisation de `p` après cela.

Une deuxième approche consiste à invalider le pointeur lui-même après avoir libéré la mémoire sur laquelle il pointe:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Arguments pour cette approche:

- Sur de nombreuses plates-formes, une tentative de déréférencement d'un pointeur nul provoquera un blocage instantané: erreur de segmentation. Ici, nous obtenons au moins une trace de pile pointant vers la variable utilisée après avoir été libérée.

  Si vous ne définissez pas le pointeur sur `NULL` nous avons un pointeur en attente. Le programme risque encore de se bloquer, mais plus tard, car la mémoire sur laquelle pointe le pointeur sera corrompue. Ces bogues sont difficiles à détecter car ils peuvent entraîner une pile d'appels sans aucun rapport avec le problème initial.

  Cette approche suit donc le concept rapide .

- Vous pouvez libérer un pointeur nul. Le standard C spécifie que `free(NULL)` n'a aucun effet:

  > La fonction libre provoque la désallocation de l’espace pointé par ptr, c’est-à-dire qu’elle est disponible pour une allocation ultérieure. Si ptr est un pointeur nul, aucune action ne se produit. Sinon, si l'argument ne correspond pas à un pointeur précédemment renvoyé par la fonction `calloc` , `malloc` ou `realloc` , ou si l'espace a été libéré par un appel à `free` ou `realloc` , le comportement n'est pas défini.

- Parfois, la première approche ne peut pas être utilisée (par exemple, la mémoire est allouée dans une fonction et libérée beaucoup plus tard dans une fonction complètement différente)

Allouer de la mémoire

Allocation Standard

Les fonctions d'allocation de mémoire dynamique C sont définies dans l'en-tête `<stdlib.h>` . Si l'on souhaite allouer dynamiquement de l'espace mémoire pour un objet, le code suivant peut être utilisé:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

Cela calcule le nombre d'octets que dix `int` s occupent en mémoire, puis demande que le nombre d' octets de `malloc` et affecte le résultat (c. -à l'adresse de départ du morceau de mémoire qui vient d'être créé à l' aide `malloc`) à un pointeur nommé `p` .

Il est recommandé d'utiliser `sizeof` pour calculer la quantité de mémoire à demander, car le résultat de `sizeof` est défini par l'implémentation (sauf pour les *types de caractères* `char` , `signed char` et `unsigned char` , pour lesquels `sizeof` est toujours défini sur `1`).

Comme `malloc` peut ne pas être en mesure de traiter la demande, il peut renvoyer un pointeur null. Il est important de vérifier cela pour empêcher les tentatives ultérieures de déréférencer le pointeur null.

La mémoire allouée dynamiquement à l'aide de `malloc()` peut être redimensionnée à l'aide de `realloc()` ou, lorsqu'elle n'est plus nécessaire, libérée à l'aide de `free()` .

Sinon, déclarez `int array[10];` allouerait la même quantité de mémoire. Cependant, s'il est déclaré dans une fonction sans le mot-clé `static` , il ne sera utilisable que dans la fonction dans laquelle il est déclaré et dans les fonctions qu'il appelle (car le tableau sera alloué sur la pile et l'espace sera libéré pour être réutilisé). la fonction retourne). Sinon, si elle est définie avec `static` à `static` intérieur d'une fonction, ou si elle est définie en dehors d'une fonction, sa durée de vie correspond à la durée de vie du programme. Les pointeurs peuvent également être renvoyés par une fonction, mais une fonction dans C ne peut pas renvoyer un tableau.

Zéro mémoire

La mémoire renvoyée par `malloc` ne peut pas être initialisée à une valeur raisonnable, et il faut veiller à mettre la mémoire à zéro avec `memset` ou à y copier immédiatement une valeur appropriée. `calloc` renvoie également un bloc de la taille souhaitée où tous les bits sont initialisés à `0` . Ce n'est pas nécessairement la même chose que la représentation du zéro à virgule flottante ou d'une constante de pointeur nul.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

Une note sur `calloc` : La plupart des implémentations (couramment utilisées) vont optimiser `calloc()` pour les performances, donc ce sera plus rapide que d'appeler `malloc()` , puis `memset()` , même si l'effet net est identique.

Mémoire Alignée

C11

C11 a introduit une nouvelle fonction `aligned_alloc()` qui alloue de l'espace avec l'alignement donné. Il peut être utilisé si la mémoire à attribuer doit être alignée à certaines limites qui ne peuvent pas être satisfaites par `malloc()` ou `calloc()` . `malloc()` et `calloc()` allouent de la mémoire convenablement alignée pour *tout* type d'objet (l'alignement est donc `alignof(max_align_t)`). Mais avec `aligned_alloc()` des alignements plus importants peuvent être demandés.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

Le standard C11 impose deux restrictions: 1) la *taille* (deuxième argument) demandé doit être un multiple entier de l' *alignement* (premier argument) et 2) la valeur de l' *alignement* doit être un alignement valide pris en charge par l'implémentation. Ne pas répondre à l'un d'eux entraîne un comportement indéfini .

Réaffecter la mémoire

Vous devrez peut-être agrandir ou réduire l'espace de stockage du pointeur après lui avoir affecté de la mémoire. La fonction `void *realloc(void *ptr, size_t size)` l'ancien objet pointé par `ptr` et renvoie un pointeur sur un objet ayant la taille spécifiée par `size` . `ptr` est le pointeur sur un bloc de mémoire précédemment alloué avec `malloc` , `calloc` ou `realloc` (ou un pointeur nul) à réallouer. Le contenu maximal possible de la mémoire d'origine est préservé. Si la nouvelle taille est plus grande, toute mémoire supplémentaire au-delà de l'ancienne taille n'est pas initialisée. Si la nouvelle taille est plus courte, le contenu de la partie réduite est perdu. Si `ptr` est NULL, un nouveau bloc est alloué et un pointeur sur celui-ci est renvoyé par la fonction.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;

    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
```

```
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
       42 15
    */
    printf("%d %d\n", p[0], p[9]);

    free(p);

    return EXIT_SUCCESS;
}
```

L'objet réalloué peut ou non avoir la même adresse que `*p` . Par conséquent, il est important de capturer la valeur de retour de `realloc` qui contient la nouvelle adresse si l'appel est réussi.

Assurez-vous d'attribuer la valeur de retour de `realloc` à un `p temporary` au lieu de l'original. `realloc` renverra null en cas de défaillance, ce qui écraserait le pointeur. Cela perdrait vos données et créerait une fuite de mémoire.

Tableaux multidimensionnels de taille variable

C99

Depuis C99, C possède des tableaux de longueur variable, VLA, qui modélisent les tableaux avec des limites connues uniquement au moment de l'initialisation. Bien qu'il faille faire attention à ne pas allouer de VLA trop volumineux (ils pourraient casser votre pile), utiliser des *pointeurs vers VLA* et les utiliser dans des expressions de `sizeof` est bien.

```
double sumAll(size_t n, size_t m, double A[n][m]) {
    double ret = 0.0;
    for (size_t i = 0; i < n; ++i)
       for (size_t j = 0; j < m; ++j)
          ret += A[i][j]
    return ret;
}

int main(int argc, char *argv[argc+1]) {
   size_t n = argc*10;
   size_t m = argc*8;
   double (*matrix)[m] = malloc(sizeof(double[n][m]));
   // initialize matrix somehow
   double res = sumAll(n, m, matrix);
   printf("result is %g\n", res);
   free(matrix);
}
```

Ici , la `matrix` est un pointeur sur des éléments de type à `double[m]` , et la `sizeof` expression avec `double[n][m]` assure qu ' elle contient un espace pour `n` tels éléments.

Tout cet espace est alloué de manière contiguë et peut ainsi être désalloué par un seul appel à `free` .

La présence de VLA dans le langage affecte également les déclarations possibles de tableaux et de pointeurs dans les en-têtes de fonction. Maintenant, une expression d'entier général est autorisée dans les `[]` des paramètres du tableau. Pour les deux fonctions, les expressions dans `[]` utilisent des paramètres déclarés auparavant dans la liste des paramètres. Pour `sumAll` ce sont les longueurs que le code utilisateur attend pour la matrice. Comme pour tous les paramètres de fonction de tableau dans C, la dimension la plus interne est réécrite dans un type de pointeur, ce qui équivaut à la déclaration

```
double sumAll(size_t n, size_t m, double (*A)[m]);
```

En d'autres termes, `n` ne fait pas vraiment partie de l'interface de la fonction, mais les informations peuvent être utiles pour la documentation et peuvent également être utilisées par les compilateurs pour vérifier les accès hors limites.

Semblablement, pour `main` , l'expression `argc+1` est la longueur minimale que le standard C prescrit pour l'argument `argv` .

Notez que officiellement, le support VLA est facultatif dans C11, mais nous ne connaissons aucun compilateur qui implémente C11 et qui ne les possède pas. Vous pouvez tester avec la macro `__STDC_NO_VLA__` si vous devez.

realloc (ptr, 0) n'est pas équivalent à free (ptr)

`realloc` est *conceptuellement équivalent* à `malloc + memcpy + free` sur l'autre pointeur.

Si la taille de l'espace demandé est égale à zéro, le comportement de `realloc` est défini par l'implémentation. Ceci est similaire pour toutes les fonctions d'allocation de mémoire qui reçoivent un paramètre de `size` de la valeur `0` . De telles fonctions peuvent en fait renvoyer un pointeur non nul, mais cela ne doit jamais être déréférencé.

Ainsi, `realloc(ptr,0)` n’est pas équivalent à `free(ptr)` . Cela pourrait

- être une implémentation "paresseuse" et renvoyer juste `ptr`
- `free(ptr)` , allouer un élément factice et renvoyer cette
- `free(ptr)` et retour `0`
- retourne juste `0` pour échec et ne fais rien d'autre.

Ainsi, en particulier, les deux derniers cas ne peuvent être distingués par le code d'application.

Cela signifie que `realloc(ptr,0)` peut ne pas libérer / désallouer la mémoire, et ne devrait donc jamais être utilisé comme remplacement `free` .

Gestion de la mémoire définie par l'utilisateur

`malloc()` appelle souvent les fonctions sous-jacentes du système d'exploitation pour obtenir des pages de mémoire. Mais la fonction n'a rien de particulier et peut être implémentée dans straight C en déclarant un grand tableau statique et en l'allouant (il est difficile de garantir un alignement correct, en pratique, l'alignement sur 8 octets est presque toujours suffisant).

Pour implémenter un schéma simple, un bloc de contrôle est stocké dans la région de mémoire immédiatement avant le pointeur à renvoyer de l'appel. Cela signifie que `free()` peut être implémenté en soustrayant du pointeur renvoyé et en lisant les informations de contrôle, ce qui correspond généralement à la taille du bloc et à certaines informations lui permettant de revenir dans la liste libre - une liste liée de blocs non alloués.

Lorsque l'utilisateur demande une allocation, la liste libre est recherchée jusqu'à ce qu'un bloc de taille identique ou supérieure au montant demandé soit trouvé, puis si nécessaire, il est divisé. Cela peut entraîner une fragmentation de la mémoire si l'utilisateur effectue continuellement de nombreuses allocations et libère une taille imprévisible et à des intervalles imprévisibles (tous les programmes réels ne se comportent pas comme cela, le système simple est souvent adapté aux petits programmes).

```
/* typical control block */
struct block
{
   size_t size;         /* size of block */
   struct block *next;  /* next block in free list */
   struct block *prev;  /* back pointer to previous block in memory */
   void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

De nombreux programmes nécessitent un grand nombre d'attributions de petits objets de même taille. C'est très facile à mettre en œuvre. Utilisez simplement un bloc avec un pointeur suivant. Donc, si un bloc de 32 octets est requis:

```
union block
{
   union block * next;
   unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* last one, null */
    head = &block[0];
}
```

```
void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head - block;
}
```

Ce système est extrêmement rapide et efficace et peut être générique avec une certaine perte de clarté.

alloca: allouer de la mémoire sur la pile

Avertissement: `alloca` n'est mentionné ici que pour être complet. Il est entièrement non portable (non couvert par les normes communes) et comporte un certain nombre de caractéristiques potentiellement dangereuses qui le rendent inoffensif pour les non-avertis. Le code C moderne devrait le remplacer par des *tableaux à longueur variable* (VLA).

Page de manuel

```
#include <alloca.h>
// glibc version of stdlib.h include alloca.h by default

void foo(int size) {
    char *data = alloca(size);
    /*
      function body;
    */
    // data is automatically freed
}
```

Allouer de la mémoire sur le cadre de la pile de l'appelant, l'espace référencé par le pointeur renvoyé est automatiquement libre lorsque la fonction de l'appelant est terminée.

Bien que cette fonction soit pratique pour la gestion automatique de la mémoire, sachez que demander une allocation importante peut provoquer un débordement de pile et que vous ne pouvez pas utiliser `free` mémoire allouée avec `alloca` (ce qui pourrait causer davantage de débordement de pile).

Pour cette raison, il n'est pas recommandé d'utiliser `alloca` dans une boucle ni une fonction récursive.

Et comme la mémoire est `free` au retour de fonction, vous ne pouvez pas renvoyer le pointeur en tant que résultat de fonction (le comportement serait indéfini).

Résumé

- appel identique à `malloc`
- automatiquement libéré lors du retour de la fonction
- incompatible avec `free` fonctions `realloc` `free` (comportement indéfini)
- le pointeur ne peut pas être renvoyé en tant que résultat de fonction (comportement non défini)
- taille d'allocation limitée par l'espace de pile, qui (sur la plupart des machines) est beaucoup plus petite que l'espace de tas disponible pour l'utilisation par `malloc()`
- éviter d'utiliser `alloca()` et VLAs (tableaux de longueur variable) dans une seule fonction
- `alloca()` n'est pas aussi portable que `malloc()` et al

Recommandation

- N'utilisez pas `alloca()` dans le nouveau code

C99

Alternative moderne.

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

Cela fonctionne là où `alloca()` fonctionne, et fonctionne dans les endroits où `alloca()` ne fonctionne pas (dans les boucles, par exemple). Il suppose une implémentation C99 ou une implémentation C11 qui ne définit pas `__STDC_NO_VLA__` .

Lire Gestion de la mémoire en ligne: https://riptutorial.com/fr/c/topic/4726/gestion-de-la-memoire

Chapitre 29: Idiomes de programmation C courants et pratiques de développeur

Examples

Comparer littéral et variable

Supposons que vous comparez une valeur avec une variable

```
if ( i  == 2) //Bad-way
{
    doSomething;
}
```

Supposons maintenant que vous avez trompés `==` avec `=` . Ensuite, il vous faudra du temps pour le comprendre.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Ensuite, si un signe égal est accidentellement omis, le compilateur se plaindra d'une «tentative d'affectation à un littéral». Cela ne vous protégera pas lorsque vous comparez deux variables, mais chaque petite aide est utile.

Voir ici pour plus d'informations.

Ne laissez pas la liste des paramètres d'une fonction vierge - utilisez void

Supposons que vous créez une fonction qui ne nécessite aucun argument lors de son appel et que vous êtes confronté au dilemme de savoir comment définir la liste de paramètres dans le prototype de fonction et la définition de fonction.

- Vous avez le choix entre conserver la liste de paramètres vide pour le prototype et la définition. Ils ressemblent donc à la déclaration d'appel de fonction dont vous aurez besoin.
- Vous lisez quelque part que l'une des utilisations du mot-clé ***vide*** (il y en a peu), est de définir la liste de paramètres des fonctions qui n'acceptent aucun argument dans leur appel. Donc, c'est aussi un choix.

Alors, quel est le bon choix?

REPONSE: en utilisant le mot-clé ***void***

CONSEILS GÉNÉRAUX: Si une langue fournit certaines fonctionnalités à utiliser dans un but

spécial, il est préférable de les utiliser dans votre code. Par exemple, utiliser `enum` s au lieu de macros `#define` (pour un autre exemple).

C11, section 6.7.6.3 "Déclarateurs de fonctions", paragraphe 10:

> Le cas particulier d'un paramètre sans nom de type void comme seul élément de la liste spécifie que la fonction n'a pas de paramètre.

Le paragraphe 14 de cette même section montre la seule différence:

> … Une liste vide dans un déclarant de fonction qui fait partie d'une définition de cette fonction spécifie que la fonction n'a pas de paramètre. La liste vide dans un déclarant de fonction qui ne fait pas partie d'une définition de cette fonction spécifie qu'aucune information sur le nombre ou les types de paramètres n'est fournie.

Une explication simplifiée fournie par K & R (pages 72-73) pour les éléments ci-dessus:

> En outre, si une déclaration de fonction n'inclut pas d'arguments, comme dans `double atof();` cela signifie aussi que rien ne doit être supposé sur les arguments de `atof` ; tout contrôle des paramètres est désactivé. Cette signification particulière de la liste d'arguments vide est destinée à permettre aux programmes C plus anciens de compiler avec de nouveaux compilateurs. Mais c'est une mauvaise idée de l'utiliser avec de nouveaux programmes. Si la fonction prend des arguments, déclarez-les; s'il ne prend aucun argument, utilisez le `void` .

Voici comment votre prototype de fonction devrait ressembler:

```
int foo(void);
```

Et voici comment la définition de la fonction devrait être:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

Un avantage d'utiliser ce qui précède, plus `int foo()` type de déclaration (ie. Sans utiliser le mot - clé ***void),*** est que le compilateur peut détecter l'erreur si vous appelez votre fonction à l' aide d' une déclaration erronée comme `foo(42)` . Ce type d'instruction d'appel de fonction ne provoquerait aucune erreur si vous laissez la liste de paramètres vide. L'erreur passerait silencieusement, non détectée et le code s'exécuterait toujours.

Cela signifie également que vous devez définir la fonction `main()` comme ceci:

```
int main(void)
{
    ...
```

```
    <statements>
    ...
    return 0;
}
```

Notez que même si une fonction définie avec une liste de paramètres vide ne prend aucun argument, elle ne fournit pas de prototype pour la fonction. Le compilateur ne se plaindra donc pas si la fonction est appelée ultérieurement avec des arguments. Par exemple:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
    return 0;
}
```

Si ce code est enregistré dans le fichier `proto79.c` , il peut être compilé sous Unix avec GCC (version 7.1.0 sur macOS Sierra 10.12.5 utilisée pour la démonstration) comme ceci:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o
proto79
$
```

Si vous compilez avec des options plus strictes, vous obtenez des erreurs:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c:3:13: error: function declaration isn't a prototype [-Werror=strict-prototypes]
 static void parameterless()
             ^~~~~~~~~~~~~
proto79.c: In function 'parameterless':
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]
cc1: all warnings being treated as errors
$
```

Si vous donnez à la fonction le prototype formel `static void parameterless(void)` , la compilation donne des erreurs:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c: In function 'main':
proto79.c:10:5: error: too many arguments to function 'parameterless'
     parameterless(3, "arguments", "provided");
     ^~~~~~~~~~~~~
proto79.c:3:13: note: declared here
 static void parameterless(void)
             ^~~~~~~~~~~~~
$
```

Moral - assurez-vous toujours d'avoir des prototypes et assurez-vous que votre compilateur vous indique quand vous n'obéissez pas aux règles.

Lire Idiomes de programmation C courants et pratiques de développeur en ligne: https://riptutorial.com/fr/c/topic/10543/idiomes-de-programmation-c-courants-et-pratiques-de-developpeur

Chapitre 30: Initialisation

Examples

Initialisation des variables en C

En l'absence d'initialisation explicite, les variables externes et `static` sont garanties pour être initialisées à zéro; Les variables automatiques (y compris `register` variables de `register`) ont *des* valeurs initiales *indéterminées* [1] (c.-à-d.

Les variables scalaires peuvent être initialisées lorsqu'elles sont définies en suivant le nom avec un signe égal et une expression:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

Pour les variables externes et `static` , l'initialiseur doit être une *expression constante* [2] ; l'initialisation est faite une fois, conceptuellement avant le début de l'exécution du programme.

Pour les variables automatiques et de `register` , l'initialiseur n'est pas limité à une constante: il peut s'agir de toute expression impliquant des valeurs précédemment définies, voire des appels de fonctions.

Par exemple, voir l'extrait de code ci-dessous

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

au lieu de

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

En effet, l'initialisation des variables automatiques n'est qu'un raccourci pour les instructions d'affectation. Quelle forme préférer est en grande partie une question de goût. Nous utilisons généralement des affectations explicites, car les initialiseurs dans les déclarations sont plus difficiles à voir et plus éloignés du point d'utilisation. Par contre, les variables ne doivent être déclarées que lorsqu'elles sont sur le point d'être utilisées, dans la mesure du possible.

Initialiser un tableau:

Un tableau peut être initialisé en suivant sa déclaration avec une liste d'initialisateurs entre accolades et séparés par des virgules.

Par exemple, pour initialiser un tableau days avec le nombre de jours de chaque mois:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Notez que janvier est encodé en mois zéro dans cette structure.)

Lorsque la taille du tableau est omise, le compilateur calcule la longueur en comptant les initialiseurs, qui sont au nombre de 12 dans ce cas.

S'il y a moins d'initialiseurs pour un tableau que la taille spécifiée, les autres seront nuls pour tous les types de variables.

C'est une erreur d'avoir trop d'initialiseurs. Il n'y a pas de méthode standard pour spécifier la répétition d'un initialiseur - mais GCC a une extension pour le faire.

C99

Dans C89 / C90 ou les versions antérieures de C, il était impossible d'initialiser un élément au milieu d'un tableau sans fournir toutes les valeurs précédentes.

C99

Avec C99 et supérieur, les initialiseurs désignés vous permettent d'initialiser des éléments arbitraires d'un tableau, en laissant toutes les valeurs non initialisées comme des zéros.

Initialisation des tableaux de caractères:

Les tableaux de caractères sont un cas particulier d'initialisation. une chaîne peut être utilisée à la place des accolades et des virgules:

```
char chr_array[] = "hello";
```

est un raccourci pour le plus long mais équivalent:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Dans ce cas, la taille du tableau est de six (cinq caractères plus le `'\0'`).

[1] Qu'advient-il d'une variable déclarée non initialisée dans C? At-il une valeur?

[2] Notez qu'une *expression constante* est définie comme quelque chose qui peut être évalué à la compilation. Donc, `int global_var = f();` est invalide. Une autre idée fausse commune est de penser une variable qualifiée de `const` comme une *expression constante* . Dans C, `const` signifie "lecture seule" et non "constante de compilation". Ainsi, les définitions globales comme `const int SIZE = 10; int global_arr[SIZE];` et `const int SIZE = 10; int global_var = SIZE;` ne sont pas légales en C.

Initialisation des structures et des tableaux de structures

Les structures et les tableaux de structures peuvent être initialisés par une série de valeurs entre accolades, une valeur par membre de la structure.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 },    // Battle of Hastings
    { 1815,  6, 18 },    // Battle of Waterloo
    { 1805, 10, 21 },    // Battle of Trafalgar
};
```

Notez que l'initialisation du tableau pourrait être écrite sans les accolades intérieures, et dans le passé (avant 1990, par exemple) aurait souvent été écrit sans eux:

```
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Bien que cela fonctionne, ce n'est pas un bon style moderne - vous ne devez pas essayer d'utiliser cette notation dans le nouveau code et corriger les avertissements du compilateur qu'elle génère habituellement.

Voir aussi les initialiseurs désignés .

Utiliser des initialiseurs désignés

C99

C99 a introduit le concept d' *initialiseurs désignés.* Cela vous permet de spécifier quels éléments d'un tableau, d'une structure ou d'une union doivent être initialisés par les valeurs suivantes.

Initialiseurs désignés pour les éléments de tableau

Pour un type simple comme plain `int` :

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

Le terme entre crochets, qui peut être n'importe quelle expression entière constante, spécifie quel

élément du tableau doit être initialisé par la valeur du terme après le signe `=` . Les éléments non spécifiés sont initialisés par défaut, ce qui signifie que des zéros sont définis. L'exemple montre les initialiseurs désignés dans l'ordre; ils ne doivent pas nécessairement être en ordre. L'exemple montre des lacunes; ceux-ci sont légitimes. L'exemple ne montre pas deux initialisations différentes pour le même élément; cela est également autorisé (ISO / CEI 9899: 2011, §6.7.9 Initialisation, ¶19 *L'initialisation doit avoir lieu dans l'ordre de la liste d'initialisation, chaque initialiseur étant fourni pour un sous-objet particulier remplaçant tout initialiseur précédemment listé pour le même sous-objet*).

Dans cet exemple, la taille du tableau n'est pas explicitement définie. Par conséquent, l'index maximal spécifié dans les initialiseurs désignés détermine la taille du tableau, ce qui correspond à 21 éléments dans l'exemple. Si la taille était définie, l'initialisation d'une entrée au-delà de la fin du tableau serait une erreur, comme d'habitude.

Initialiseurs désignés pour les structures

Vous pouvez spécifier quels éléments d'une structure sont initialisés à l'aide de `.` notation d' *`element`* :

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

Si les éléments ne sont pas répertoriés, ils sont initialisés par défaut (mis à zéro).

Initialiseur désigné pour les syndicats

Vous pouvez spécifier quel élément d'une union est initialisé avec un initialiseur désigné.

C89

Avant la norme C, il n'y avait aucun moyen d'initialiser une `union` . La norme C89 / C90 vous permet d’initialiser le premier membre d’une `union` - le choix du membre est donc le premier.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 }
};
```

C11

Notez que C11 vous permet d'utiliser des membres d'union anonymes dans une structure, de sorte que vous n'avez pas besoin `du` nom `du` dans l'exemple précédent:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Initialiseurs désignés pour les tableaux de structures, etc.

Ces constructions peuvent être combinées pour des tableaux de structures contenant des éléments tels que des tableaux, etc. L'utilisation d'ensembles complets d'accolades permet de s'assurer que la notation n'est pas ambiguë.

```
typedef struct Date Date;  // See earlier in this example

struct date_range
{
    Date    dr_from;
    Date    dr_to;
    char    dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to   = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
          },
    [2] = { .dr_from = { .month = 7, .day =  4, .year = 1776 },
            .dr_to   = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
          }
 };
```

Spécification de plages dans les initialiseurs de tableau

GCC fournit une extension qui vous permet de spécifier une plage d'éléments dans un tableau devant recevoir le même initialiseur:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

Les points triples doivent être séparés des nombres, au cas où l'un des points ne serait pas interprété comme faisant partie d'un nombre à virgule flottante (règle de *maximalisation maximale*

).

Lire Initialisation en ligne: https://riptutorial.com/fr/c/topic/4547/initialisation

Chapitre 31: Inlining

Examples

Fonctions d'inline utilisées dans plus d'un fichier source

Pour les petites fonctions appelées souvent, la surcharge associée à l'appel de la fonction peut représenter une fraction significative du temps d'exécution total de cette fonction. Une façon d'améliorer les performances consiste donc à éliminer les frais généraux.

Dans cet exemple, nous utilisons quatre fonctions (plus `main()`) dans trois fichiers sources. Deux d'entre elles (`plusfive()` et `timestwo()`) sont appelées par les deux autres situées dans "source1.c" et "source2.c". Le `main()` est inclus, nous avons donc un exemple de travail.

principal c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

source1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
```

```
    return tmp;
}
```

headerfile.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
  return input * 2;
}
inline int plusfive(int input) {
  return input + 5;
}

#endif
```

Fonctions `timestwo` et `plusfive` s'appellent à la fois par `complicated1` et `complicated2` , qui sont dans des « unités de traduction », ou les fichiers source. Pour les utiliser de cette manière, il faut les définir dans l'en-tête.

Compilez comme ceci, en supposant que gcc:

```
cc -O2 -std=c99 -c -o main.o main.c
cc -O2 -std=c99 -c -o source1.o source1.c
cc -O2 -std=c99 -c -o source2.o source2.c
cc main.o source1.o source2.o -o main
```

Nous utilisons l'option d'optimisation -O2 car certains compilateurs ne sont pas intégrés sans optimisation activée.

L'effet du mot clé en `inline` est que le symbole de la fonction en question n'est pas émis dans le fichier objet. Sinon, une erreur se produirait dans la dernière ligne, où nous lions les fichiers objet pour former le dernier exécutable. Si nous n'avions pas de `inline` , le même symbole serait défini dans les deux fichiers `.o` et une erreur "symbole défini par plusieurs" se produirait.

Dans les situations où le symbole est réellement nécessaire, cela présente l'inconvénient que le symbole n'est pas produit du tout. Il y a deux possibilités pour faire face à cela. La première consiste à ajouter une déclaration `extern` supplémentaire des fonctions intégrées dans l'un des fichiers `.c` . Donc, ajoutez ce qui suit à `source1.c` :

```
extern int timestwo(int input);
extern int plusfive(int input);
```

L'autre possibilité est de définir la fonction avec `static inline` au lieu de `inline` . Cette méthode a l'inconvénient qu'une copie de la fonction en question peut éventuellement être produite dans **chaque** fichier objet produit avec cet en-tête.

Lire Inlining en ligne: https://riptutorial.com/fr/c/topic/7427/inlining

Chapitre 32: Instructions de saut

Syntaxe

- retour val; / * Retourne de la fonction actuelle. val peut être une valeur de tout type qui est convertie en type de retour de la fonction. * /
- revenir; / * Retourne de la fonction void actuelle. * /
- Pause; / * Saute inconditionnellement au-delà de la fin ("saute") d'un relevé d'itération (boucle) ou de la déclaration de commutateur la plus interne. * /
- continuer; / * Saute inconditionnellement au début d'un énoncé d'itération (boucle). * /
- aller à LBL; / * Saute pour étiqueter LBL. * /
- LBL: *statement* / * toute instruction dans la même fonction. * /

Remarques

Ce sont les sauts intégrés à C au moyen de mots-clés.

C a également une autre construction de *saut* , *saut en longueur* , qui est spécifiée avec un type de données, les `jmp_buf` et les bibliothèques de la bibliothèque, `setjmp` et `longjmp` .

Voir également

Énoncés d'itération / boucles: pour, pendant et après

Examples

Utiliser goto pour sauter des boucles imbriquées

Sauter des boucles imbriquées nécessiterait généralement l'utilisation d'une variable booléenne avec une vérification de cette variable dans les boucles. En supposant que nous parcourons `i` et `j` , cela pourrait ressembler à ceci

```
size_t i,j;
for (i = 0; i < myValue && !breakout_condition; ++i) {
    for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
        ... /* Do something, maybe modifying breakout_condition */
            /* When breakout_condition == true the loops end */
     }
}
```

Mais le langage C propose la clause `goto` , qui peut être utile dans ce cas. En l'utilisant avec une étiquette déclarée après les boucles, nous pouvons facilement sortir des boucles.

```
size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
```

```
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

Cependant, souvent, lorsque ce besoin se présente, un `return` pourrait être mieux utilisé. Cette construction est également considérée comme "non structurée" dans la théorie de la programmation structurelle.

Une autre situation où `goto` peut être utile est de sauter à un gestionnaire d'erreur:

```
ptr = malloc(N *  x);
if(!ptr)
  goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
 free(ptr); /* harmless, and necessary if we have further errors */
 return FAILURE;
```

L'utilisation de `goto` permet de garder le flux d'erreur distinct du flux de contrôle de programme normal. Il est toutefois également considéré comme "non structuré" au sens technique.

Utiliser le retour

Retourner une valeur

Un cas couramment utilisé: retour de `main()`

```
#include <stdlib.h> /* for EXIT_xxx macros */

int main(int argc, char ** argv)
{
  if (2 < argc)
  {
    return EXIT_FAILURE; /* The code expects one argument:
                            leave immediately skipping the rest of the function's code */
  }

  /* Do stuff. */

  return EXIT_SUCCESS;
}
```

Notes complémentaires:

1. Pour une fonction ayant un type de retour comme `void` (`void` pas les types `void *` ou apparentés), l'instruction `return` ne doit avoir aucune expression associée; c'est-à-dire que la

seule déclaration de retour autorisée serait le `return;` .

2. Pour une fonction ayant un type de retour non `void` , l'instruction `return` ne doit pas apparaître sans expression.

3. Pour `main()` (et uniquement pour `main()`), une instruction de `return` *explicite* n'est pas requise (en C99 ou version ultérieure). Si l'exécution atteint le terme `}` , une valeur implicite de `0` est renvoyée. Certaines personnes pensent que l'omission de ce `return` est une mauvaise pratique; d'autres suggèrent activement de le laisser de côté.

Ne rien retourner

Retourner d'une fonction `void`

```
void log(const char * message_to_log)
{
  if (NULL == message_to_log)
  {
    return; /* Nothing to log, go home NOW, skip the logging. */
  }

  fprintf(stderr, "%s:%d %s\n", __FILE__, _LINE__, message_to_log);

  return; /* Optional, as this function does not return a value. */
}
```

Utiliser la pause et continuer

Immédiatement `continue` la lecture sur l' entrée non valide ou `break` sur demande de l' utilisateur ou en fin de fichier:

```
#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h>  /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);


int main(void)
{
  int sum = 0;
  printf("Enter digits to be summed up or 0 to exit:\n");

  do
  {
    int c = getchar();
    if (EOF == c)
    {
      printf("Read 'end-of-file', exiting!\n");

      break;
    }

    if ('\n' != c)
```

```
    {
      flush_input_stream(stdin);
    }

    if (!isdigit(c))
    {
      printf("%c is not a digit! Start over!\n", c);

      continue;
    }

    if ('0' == c)
    {
      printf("Exit requested.\n");

      break;
    }

    sum += c - '0';

    printf("The current sum is %d.\n", sum);
  } while (1);

  return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
  size_t i = 0;
  int c;
  while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-
line. */
  {
    ++i;
  }

  if (0 != i)
  {
    fprintf(stderr, "Flushed %zu characters from input.\n", i);
  }
}
```

Lire Instructions de saut en ligne: https://riptutorial.com/fr/c/topic/5568/instructions-de-saut

Chapitre 33: La gestion des erreurs

Syntaxe

- #include <errno.h>
- int errno; / * implémentation définie * /
- #include <string.h>
- char * strerror (int errnum);
- #include <stdio.h>
- perid vide (const char * s);

Remarques

N'oubliez pas que `errno` n'est pas nécessairement une variable, mais que la syntaxe n'est qu'une indication de la manière dont elle *pourrait* être déclarée. Dans de nombreux systèmes modernes avec des interfaces de threads, `errno` est une macro qui se résout en un objet local au thread actuel.

Examples

errno

Lorsqu'une fonction de bibliothèque standard échoue, elle définit souvent errno avec le code d'erreur approprié. Le standard C nécessite au moins 3 valeurs pour errno:

Valeur	Sens
EDOM	Erreur de domaine
ERANGE	Erreur de portée
EILSEQ	Séquence de caractères multi-octets illégale

strerror

Si `perror` n'est pas assez flexible, vous pouvez obtenir une description d'erreur lisible par l'utilisateur en appelant `strerror` partir de `<string.h>` .

```
int main(int argc, char *argv[])
{
    FILE *fout;
    int last_error = 0;

    if ((fout = fopen(argv[1], "w")) == NULL) {
        last_error = errno;
```

```
        /* reset errno and continue */
        errno = 0;
    }

    /* do some processing and try opening the file differently, then */


    if (last_error) {
        fprintf(stderr, "fopen: Could not open %s for writing: %s",
                argv[1], strerror(last_error));
        fputs("Cross fingers and continue", stderr);
    }

    /* do some other processing */

    return EXIT_SUCCESS;
}
```

perror

Pour imprimer un message d'erreur lisible par l'utilisateur sur `stderr` , appelez `perror` depuis <stdio.h>.

```
int main(int argc, char *argv[])
{
   FILE *fout;

   if ((fout = fopen(argv[1], "w")) == NULL) {
      perror("fopen: Could not open file for writing");
      return EXIT_FAILURE;
   }
return EXIT_SUCCESS;
}
```

Cela affichera un message d'erreur concernant la valeur actuelle de `errno` .

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/c/topic/2486/la-gestion-des-erreurs

Chapitre 34: Les opérateurs

Introduction

Un opérateur dans un langage de programmation est un symbole qui indique au compilateur ou à l'interprète d'exécuter une opération mathématique, relationnelle ou logique spécifique et de produire un résultat final.

C a beaucoup d'opérateurs puissants. De nombreux opérateurs C sont des opérateurs binaires, ce qui signifie qu'ils ont deux opérandes. Par exemple, dans `a / b` , `/` est un opérateur binaire qui accepte deux opérandes (`a` , `b`). Il y a des opérateurs unaires qui prennent un opérande (par exemple: `~` , `++`), et un seul opérateur ternaire `? :` .

Syntaxe

- opérateur expr1
- opérateur expr2
- opérateur expr1 expr2
- expr1? expr2: expr3

Remarques

Les opérateurs ont une *arité* , une *priorité* et une *associativité* .

- *Arity* indique le nombre d'opérandes. En C, il existe trois opérateurs différents:
    - Unaire (1 opérande)
    - Binaire (2 opérandes)
    - Ternaire (3 opérandes)
- *La priorité* indique quels opérateurs "se lient" en premier à leurs opérandes. C'est-à-dire quel opérateur a la priorité d'opérer sur ses opérandes. Par exemple, le langage C obéit à la convention selon laquelle la multiplication et la division ont priorité sur l'addition et la soustraction:

    ```
    a * b + c
    ```

    Donne le même résultat que

    ```
    (a * b) + c
    ```

    Si ce n'est pas ce qui était souhaité, la priorité peut être forcée à l'aide de parenthèses, car elles ont la *plus haute* priorité de tous les opérateurs.

```
a * (b + c)
```

Cette nouvelle expression produira un résultat différent des deux expressions précédentes.

Le langage C a de nombreux niveaux de priorité; Un tableau est donné ci-dessous de tous les opérateurs, par ordre décroissant de priorité.

Tableau de préséance

Les opérateurs	Associativité
`() [] -> .`	de gauche à droite
`! ~ ++ -- + - *` (déréférencement) `(type) sizeof`	de droite à gauche
`*` (multiplication) `/ %`	de gauche à droite
`+ -`	de gauche à droite
`<< >>`	de gauche à droite
`< <= > >=`	de gauche à droite
`== !=`	de gauche à droite
`&`	de gauche à droite
`^`	de gauche à droite
`|`	de gauche à droite
`&&`	de gauche à droite
`||`	de gauche à droite
`?:`	de droite à gauche
`= += -= *= /= %= &= ^= |= <<= >>=`	de droite à gauche
`,`	de gauche à droite

- *L'associativité* indique comment les opérateurs d'égale priorité sont liés par défaut et il en existe deux types: de *gauche à droite* et de *droite à gauche* . Un exemple de liaison de *gauche à droite* est l'opérateur de soustraction (`-`). L'expression

```
a - b - c - d
```

a trois soustractions de priorité identique, mais donne le même résultat que

```
((a - b) - c) - d
```

parce que le plus à gauche `-` se lie d'abord à ses deux opérandes.

Un exemple d'associativité de *droite à gauche* sont les opérateurs dereference `*` et post-incrément `++` . Les deux ont la même priorité, donc s'ils sont utilisés dans une expression telle que

```
* ptr ++
```

, cela équivaut à

```
* (ptr ++)
```

parce que l'opérateur unaire le plus à droite (`++`) se lie d'abord à son seul opérande.

Examples

Opérateurs relationnels

Les opérateurs relationnels vérifient si une relation spécifique entre deux opérandes est vraie. Le résultat est évalué à 1 (ce qui signifie *vrai*) ou à 0 (ce qui signifie *faux*). Ce résultat est souvent utilisé pour affecter le flux de contrôle (via `if` , `while` , `for`), mais peut également être stocké dans des variables.

Est égal à "=="

Vérifie si les opérandes fournis sont égaux.

```
1 == 0;         /* evaluates to 0. */
1 == 1;         /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yptr; /* evaluates to 1, the operands point at locations that hold the same value.
*/
```

Attention: cet opérateur ne doit pas être confondu avec l'opérateur d'affectation (`=`)!

Pas égal à "! ="

Vérifie si les opérandes fournis ne sont pas égaux.

```
1 != 0;         /* evaluates to 1. */
1 != 1;         /* evaluates to 0. */
```

```
int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yptr; /* evaluates to 0, the operands point at locations that hold the same value.
*/
```

Cet opérateur renvoie effectivement le résultat opposé à celui de l'opérateur égal (`==`).

Ne pas "!"

Vérifiez si un objet est égal à `0` .

Le `!` peut également être utilisé directement avec une variable comme suit:

```
!someVal
```

Cela a le même effet que:

```
someVal == 0
```

Plus grand que ">"

Vérifie si l'opérande de gauche a une valeur supérieure à l'opérande de droite

```
5 > 4      /* evaluates to 1. */
4 > 5      /* evaluates to 0. */
4 > 4      /* evaluates to 0. */
```

Moins que "<"

Vérifie si l'opérande de gauche a une valeur inférieure à l'opérande de droite

```
5 < 4      /* evaluates to 0. */
4 < 5      /* evaluates to 1. */
4 < 4      /* evaluates to 0. */
```

Supérieur ou égal à "> ="

Vérifie si l'opérande de gauche a une valeur supérieure ou égale à l'opérande de droite.

```
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

Inférieur ou égal à "<="

Vérifie si l'opérande de gauche a une valeur inférieure ou égale à l'opérande de droite.

```
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

Opérateurs d'affectation

Attribue la valeur de l'opérande de droite à l'emplacement de stockage nommé par l'opérande de gauche et renvoie la valeur.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99. Returns 99
                 * (as the character 'c' is represented in the ASCII table with 99).
                 */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string
'foo'. */
```

Plusieurs opérations arithmétiques ont un opérateur d' *affectation composé* .

```
a += b  /* equal to: a = a + b */
a -= b  /* equal to: a = a - b */
a *= b  /* equal to: a = a * b */
a /= b  /* equal to: a = a / b */
a %= b  /* equal to: a = a % b */
a &= b  /* equal to: a = a & b */
a |= b  /* equal to: a = a | b */
a ^= b  /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

Une caractéristique importante de ces affectations composées est que l'expression du côté gauche (`a`) n'est évaluée qu'une seule fois. Par exemple, si `p` est un pointeur

```
*p += 27;
```

dereferences `p` qu'une seule fois, alors que le suivant le fait deux fois.

```
*p = *p + 27;
```

Il convient également de noter que le résultat d'une assignation telle que `a = b` est ce qu'on appelle une *valeur* . Ainsi, l'affectation a en réalité une valeur qui peut ensuite être affectée à une autre variable. Cela permet de chaîner des affectations pour définir plusieurs variables dans une seule instruction.

Cette *valeur* peut être utilisée dans les expressions de contrôle des instructions `if` (ou boucles ou instructions `switch`) qui protègent du code sur le résultat d'une autre expression ou d'un appel de fonction. Par exemple:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Pour cette raison, il faut veiller à éviter une faute de frappe commune pouvant conduire à des bogues mystérieux.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

Cela aura des résultats désastreux, car `a = 1` sera toujours évalué à `1` et donc l'expression de contrôle de l'instruction `if` sera toujours vraie (en savoir plus sur cet écueil commun ici). L'auteur avait certainement l'intention d'utiliser l'opérateur d'égalité (`==`) comme indiqué ci-dessous:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Associativité des opérateurs

```
int a, b = 1, c = 2;
a = b = c;
```

Cela assigne `c` à `b` , qui renvoie `b` , qui est assigné à `a` . Cela se produit parce que tous les opérateurs d'affectation ont une associativité correcte, ce qui signifie que l'opération la plus à droite de l'expression est évaluée en premier et se poursuit de droite à gauche.

Opérateurs arithmétiques

Arithmétique de base

Renvoie une valeur résultant de l'application de l'opérande de gauche à l'opérande de droite, à l'aide de l'opération mathématique associée. Des règles mathématiques normales de commutation s'appliquent (c.-à-d. L'addition et la multiplication sont commutatives, la soustraction, la division et le module ne le sont pas).

Opérateur Additionnel

L'opérateur d'addition (`+`) est utilisé pour ajouter deux opérandes ensemble. Exemple:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a + b; /* c now holds the value 12 */

    printf("%d + %d = %d",a,b,c); /* will output "5 + 7 = 12" */

    return 0;
}
```

Opérateur de soustraction

L'opérateur de soustraction (`-`) est utilisé pour soustraire le second opérande du premier. Exemple:

```
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d",a,b,c); /* will output "10 - 7 = 3" */

    return 0;
}
```

Opérateur de multiplication

L'opérateur de multiplication (`*`) est utilisé pour multiplier les deux opérandes. Exemple:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d",a,b,c); /* will output "5 * 7 = 35" */

    return 0;
}
```

Ne pas confondre avec l'opérateur `*` dereference.

Opérateur de division

L'opérateur de division (`/`) divise le premier opérande par le second. Si les deux opérandes de la division sont des nombres entiers, il retournera une valeur entière et rejettera le reste (utilisez l'opérateur `%` modulo pour calculer et acquérir le reste).

Si l'un des opérandes est une valeur à virgule flottante, le résultat est une approximation de la fraction.

Exemple:

```
#include <stdio.h>

int main (void)
{
    int a = 19 / 2 ; /* a holds value 9   */
    int b = 18 / 2 ; /* b holds value 9   */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4 ; /* d holds value 11  */
    double e = 19 / 2.0 ; /* e holds value 9.5   */
    double f = 18.0 / 2 ; /* f holds value 9.0   */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4 ; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a);    /* Will output "19 / 2 = 9"    */
    printf("18 / 2 = %d\n", b);    /* Will output "18 / 2 = 9"    */
    printf("255 / 2 = %d\n", c);   /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d);    /* Will output "44 / 4 = 11"   */
    printf("19 / 2.0 = %g\n", e);  /* Will output "19 / 2.0 = 9.5"    */
    printf("18.0 / 2 = %g\n", f);  /* Will output "18.0 / 2 = 9"      */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h);  /* Will output "45.0 / 4 = 11.25"  */

    return 0;
}
```

Opérateur Modulo

L'opérateur modulo (`%`) reçoit uniquement des opérandes entiers et est utilisé pour calculer le reste après la division du premier opérande par le second. Exemple:

```
#include <stdio.h>

int main (void) {
    int a = 25 % 2;    /* a holds value 1  */
    int b = 24 % 2;    /* b holds value 0  */
    int c = 155 % 5;   /* c holds value 0  */
    int d = 49 % 25;   /* d holds value 24 */

    printf("25 % 2 = %d\n", a);     /* Will output "25 % 2 = 1"    */
    printf("24 % 2 = %d\n", b);     /* Will output "24 % 2 = 0"    */
    printf("155 % 5 = %d\n", c);    /* Will output "155 % 5 = 0"   */
    printf("49 % 25 = %d\n", d);    /* Will output "49 % 25 = 24"  */
```

```
    return 0;
}
```

Opérateurs d'incrémentation / décrémentation

Les opérateurs d'incrémentation (`a++`) et de décrémentation (`a--`) sont différents en ce sens qu'ils changent la valeur de la variable à laquelle ils sont appliqués sans opérateur d'affectation. Vous pouvez utiliser des opérateurs d'incrémentation et de décrémentation avant ou après la variable. L'emplacement de l'opérateur modifie la synchronisation de l'incrémentation / décrémentation de la valeur avant ou après l'affectation à la variable. Exemple:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;
    int c = 1;
    int d = 4;

    a++;
    printf("a = %d\n",a);    /* Will output "a = 2" */
    b--;
    printf("b = %d\n",b);    /* Will output "b = 3" */

    if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
        printf("This will print\n");    /* This is printed */
    } else {
        printf("This will never print\n");    /* This is not printed */
    }

    if (d-- < 4) {  /* d is decremented after being compared */
        printf("This will never print\n");    /* This is not printed */
    } else {
        printf("This will print\n");    /* This is printed */
    }
}
```

Comme le montre l'exemple de `c` et `d` , les deux opérateurs ont deux formes, la notation préfixe et la notation postfixe. Les deux ont le même effet en incrémentant (`++`) ou en décrémentant (`--`) la variable, mais diffèrent par la valeur qu'ils renvoient: les opérations de préfixe font l'opération en premier et ensuite la valeur, alors que les opérations postfixes déterminent d'abord la valeur à être retourné, puis effectuez l'opération.

En raison de ce comportement potentiellement contre-intuitif, l'utilisation d'opérateurs d'incrémentation / décrémentation à l'intérieur des expressions est controversée.

Opérateurs logiques

Logique ET

Effectue un booléen logique AND-ing des deux opérandes renvoyant 1 si les deux opérandes ne

sont pas à zéro. L'opérateur logique AND est de type `int` .

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

OU logique

Effectue un OU booléen logique des deux opérandes retournant 1 si l'un des opérandes est non nul. L'opérateur logique OU est de type `int` .

```
0 || 0  /* Returns 0. */
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

Logique NON

Effectue une négation logique. L'opérateur logique NOT est de type `int` . L'opérateur NOT vérifie si au moins un bit est égal à 1, si tel est le cas, il renvoie 0. Sinon, il renvoie 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Évaluation en court-circuit

Il y a quelques propriétés essentielles communes à la fois `&&` et `||` :

- l'opérande de gauche (LHS) est entièrement évalué avant que l'opérande de droite (RHS) soit évalué,
- il y a un point de séquence entre l'évaluation de l'opérande gauche et l'opérande droit,
- et, plus important encore, l’opérande de droite n’est pas du tout évalué si le résultat de l’opérande de gauche détermine le résultat global.

Cela signifie que:

- si le LHS est évalué à «vrai» (non nul), l'ERS de `||` ne sera pas évalué (parce que le résultat de «vrai OU quelque chose» est «vrai»),
- Si le LHS est évalué à "false" (zéro), le RHS de `&&` ne sera pas évalué (car le résultat de "false AND importe" est "false").

Ceci est important car cela vous permet d'écrire du code tel que:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
```

```
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

Si une valeur négative est transmise à la fonction, la `value >= 0` terme est évaluée à false et la `value < NUM_NAMES` terme n'est pas évaluée.

Incrémenter / Décrémenter

Les opérateurs d'incrémentation et de décrémentation existent sous forme de *préfixe* et de *postfixe* .

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;        /* increments a by one, and returns new value; a == 2, tmp == 2  */
tmp = a++;        /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;        /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;        /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Notez que les opérations arithmétiques n'introduisent pas de points de séquence , de sorte que certaines expressions avec des opérateurs `++` ou `--` peuvent introduire un comportement indéfini .

Opérateur conditionnel / opérateur ternaire

Évalue son premier opérande et, si la valeur résultante n'est pas égale à zéro, évalue son deuxième opérande. Sinon, il évalue son troisième opérande, comme illustré dans l'exemple suivant:

```
a = b ? c : d;
```

est équivalent à:

```
if (b)
    a = c;
else
    a = d;
```

Ce pseudo-code le représente: `condition ? value_if_true : value_if_false` . Chaque valeur peut être le résultat d'une expression évaluée.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

L'opérateur conditionnel peut être imbriqué. Par exemple, le code suivant détermine le plus grand des trois nombres:

```
big= a > b ? (a > c ? a : c)
```

```
          : (b > c ? b : c);
```

L'exemple suivant écrit même des entiers dans un fichier et des entiers impairs dans un autre fichier:

```
#include<stdio.h>

int main()
{
    FILE *even, *odds;
    int n = 10;
    size_t k = 0;

    even = fopen("even.txt", "w");
    odds = fopen("odds.txt", "w");

    for(k = 1; k < n + 1; k++)
    {
        k%2==0 ? fprintf(even, "\t%5d\n", k)
               : fprintf(odds, "\t%5d\n", k);
    }
    fclose(even);
    fclose(odds);

    return 0;
}
```

L'opérateur conditionnel associe de droite à gauche. Considérer ce qui suit:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

Comme l'association est de droite à gauche, l'expression ci-dessus est évaluée comme

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

Opérateur de virgule

Évalue son opérande gauche, supprime la valeur résultante, puis évalue son opérande de droits et result renvoie la valeur de son opérande le plus à droite.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

L'opérateur de virgule introduit un point de séquence entre ses opérandes.

Notez que la *virgule* utilisée dans les fonctions appelant des arguments séparés n'est PAS l' *opérateur virgule* , mais plutôt un *séparateur* différent de l' *opérateur virgule* . Par conséquent, il n'a pas les propriétés de l' *opérateur de virgule* .

L'appel `printf()` ci-dessus contient à la fois l' *opérateur de virgule* et le *séparateur* .

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

```
/*           ^        ^ this is a comma operator */
/*           this is a separator */
```

L'opérateur virgule est souvent utilisé dans la section d'initialisation ainsi que dans la section de mise à jour d'une boucle `for` . Par exemple:

```
for(k = 1; k < 10; printf("\%d\\n", k), k += 2);   /*outputs the odd numbers below 9/*

/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\\n", k, sumk);
```

Opérateur de casting

Effectue une conversion *explicite* dans le type donné à partir de la valeur résultant de l'évaluation de l'expression donnée.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Ici, la valeur de `x` est convertie en `double` , la division favorise également la valeur de `y` pour `double` , et le résultat de la division, un `double` est transmis à `printf` pour impression.

taille de l'opérateur

Avec un type comme opérande

Evalue dans la taille en octets, de type `size_t` , des objets du type donné. Nécessite des parenthèses autour du type.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-dependent. */
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by parentheses! */
```

Avec une expression comme opérande

Évalue la taille en octets, de type `size_t` , des objets du type de l'expression donnée. L'expression elle-même n'est pas évaluée. Les parenthèses ne sont pas requises; Cependant, comme l'expression doit être unaire, il est recommandé de toujours les utiliser.

```
char ch = 'a';
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always 1 for all platforms. */
printf("%zu\n", sizeof ch);  /* Valid, will output the size of a char object, which is always 1 for all platforms. */
```

Ajout de pointeur

Étant donné un pointeur et un type scalaire `N` , évalue en un pointeur le `N` ème élément du type pointé qui succède directement à l'objet pointé en mémoire.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

Peu importe si le pointeur est utilisé comme valeur d'opérande ou comme valeur scalaire. Cela signifie que des choses telles que `3 + arr` sont valides. Si `arr[k]` est le membre `k+1` d'un tableau, alors `arr+k` est un pointeur sur `arr[k]` . En d'autres termes, `arr` ou `arr+0` est un pointeur sur `arr[0]` , `arr+1` est un pointeur sur `arr[2]` , etc. En général, `*(arr+k)` est identique à `arr[k]` .

Contrairement à l'arithmétique habituelle, l'ajout de `1` à un pointeur sur un `int` ajoutera `4` octets à la valeur d'adresse actuelle. Comme les noms de tableaux sont des pointeurs constants, `+` est le seul opérateur que nous pouvons utiliser pour accéder aux membres d'un tableau via la notation du pointeur en utilisant le nom du tableau. Cependant, en définissant un pointeur sur un tableau, nous pouvons obtenir plus de flexibilité pour traiter les données dans un tableau. Par exemple, nous pouvons imprimer les membres d'un tableau comme suit:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

En définissant un pointeur sur le tableau, le programme ci-dessus est équivalent à ce qui suit:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", ptr[k]);
        /* or   printf("\n\t%d", *(ptr + k)); */
        /* or   printf("\n\t%d", *ptr++); */
    }
    return 0;
```

```
}
```

Voir que les membres du tableau `arr` sont accédés en utilisant les opérateurs `+` et `++` . Les autres opérateurs pouvant être utilisés avec le pointeur `ptr` sont `-` et `--` .

Soustraction de pointeur

Étant donné deux pointeurs sur le même type, évalue en un objet de type `ptrdiff_t` qui contient la valeur scalaire qui doit être ajoutée au second pointeur afin d'obtenir la valeur du premier pointeur.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("*(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

Opérateurs d'accès

Les opérateurs d'accès au membre (dot `.` Et flèche `->`) sont utilisés pour accéder à un membre d'une `struct` .

Membre d'objet

Évalue la valeur indiquant l'objet qui est membre de l'objet accédé.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */
```

Membre d'objet pointé

Sucre syntaxique pour déréférencement suivi de l'accès des membres. Effectivement, une expression de la forme `x->y` est un raccourci pour `(*x).y` - mais l'opérateur de la flèche est beaucoup plus clair, surtout si les pointeurs de la structure sont imbriqués.

```
struct MyStruct
{
    int x;
    int y;
};
```

```
struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */
```

Adresse de

Le unaire `&` opérateur est l'adresse de l'opérateur. Il évalue l'expression donnée, où l'objet résultant doit être une lvalue. Ensuite, il évalue un objet dont le type est un pointeur sur le type de l'objet résultant et contient l'adresse de l'objet résultant.

```
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-
defined A. */
```

Déréférence

L'opérateur unaire `*` supprime un pointeur. Il évalue la valeur résultant du déréférencement du pointeur résultant de l'évaluation de l'expression donnée.

```
int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */
```

Indexage

L’indexation est du sucre syntaxique pour l’ajout de pointeur suivi d’un déréférencement. En effet, une expression de la forme `a[i]` est équivalente à `*(a + i)` - mais la notation en indice explicite est préférable.

```
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

Interchangeabilité de l'indexation

Ajouter un pointeur à un entier est une opération commutative (c’est-à-dire que l’ordre des opérandes ne change pas le résultat) donc `pointer + integer == integer + pointer` .

Une conséquence de ceci est que `arr[3]` et `3[arr]` sont équivalents.

```
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

L'utilisation d'une expression `3[arr]` au lieu de `arr[3]` n'est généralement pas recommandée, car elle affecte la lisibilité du code. Il a tendance à être populaire dans les concours de programmation obscurcis.

Opérateur d'appel de fonction

Le premier opérande doit être un pointeur de fonction (un désignateur de fonction est également acceptable car il sera converti en pointeur sur la fonction), identifiant la fonction à appeler et tous les autres opérandes, le cas échéant, sont appelés collectivement les arguments de l'appel de fonction. . Evalue dans la valeur de retour résultant de l'appel de la fonction appropriée avec les arguments respectifs.

```
int myFunction(int x, int y)
{
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference
explicitly */
```

Opérateurs sur les bits

Les opérateurs binaires peuvent être utilisés pour effectuer des opérations au niveau du bit sur les variables.
Vous trouverez ci-dessous une liste des six opérateurs binaires pris en charge dans C:

symbole	Opérateur
Et	binaire ET
|	bit à bit inclus OU
^	exclusif binaire OU (XOR)
~	pas du tout (son complément)
<<	décalage gauche logique
>>	décalage logique droite

Le programme suivant illustre l'utilisation de tous les opérateurs binaires:

```
#include <stdio.h>
```

```
int main(void)
{
   unsigned int a = 29;    /* 29 = 0001 1101 */
   unsigned int b = 48;    /* 48 = 0011 0000 */
   int c = 0;

   c = a & b;              /* 32 = 0001 0000 */
   printf("%d & %d = %d\n", a, b, c );

   c = a | b;              /* 61 = 0011 1101 */
   printf("%d | %d = %d\n", a, b, c );

   c = a ^ b;              /* 45 = 0010 1101 */
   printf("%d ^ %d = %d\n", a, b, c );

   c = ~a;                 /* -30 = 1110 0010 */
   printf("~%d = %d\n", a, c );

   c = a << 2;             /* 116 = 0111 0100 */
   printf("%d << 2 = %d\n", a, c );

   c = a >> 2;             /* 7 = 0000 0111 */
   printf("%d >> 2 = %d\n", a, c );

   return 0;
}
```

Les opérations binaires avec des types signés doivent être évitées car le bit de signe de cette représentation binaire a une signification particulière. Des restrictions particulières s’appliquent aux opérateurs de quarts:

- Le décalage à gauche d'un bit dans le bit signé est erroné et conduit à un comportement indéfini.
- Le décalage à droite d'une valeur négative (avec le bit de signe 1) est défini par l'implémentation et n'est donc pas portable.
- Si la valeur de l'opérande droit d'un opérateur de décalage est négative ou est supérieure ou égale à la largeur de l'opérande gauche promu, le comportement n'est pas défini.

Masquage:

Le masquage fait référence au processus d'extraction des bits souhaités à partir d'une variable (ou à la transformation des bits souhaités) à l'aide d'opérations binaires logiques. L'opérande (constante ou variable) utilisé pour effectuer le masquage est appelé *masque* .

Le masquage est utilisé de différentes manières:

- Décider du modèle de bit d'une variable entière.
- Pour copier une partie d'un motif de bit donné dans une nouvelle variable, alors que le reste de la nouvelle variable est rempli de 0 (à l'aide du bit AND)
- Pour copier une partie d'un motif de bit donné dans une nouvelle variable, alors que le reste de la nouvelle variable est rempli de 1 (à l'aide d'un bit OU).

- Pour copier une partie d'un motif de bit donné dans une nouvelle variable, le reste du motif de bit d'origine est inversé dans la nouvelle variable (à l'aide d'un OU exclusif au niveau du bit).

La fonction suivante utilise un masque pour afficher le modèle de bit d'une variable:

```
#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;
    word = CHAR_BIT * sizeof(int);
    mask = mask << (word - 1);    /* shift 1 to the leftmost position */
    for(i = 1; i <= word; i++)
    {
        x = (u & mask) ? 1 : 0;  /* identify the bit */
        printf("%d", x);         /* print bit value */
        mask >>= 1;              /* shift mask to the right by 1 bit */
    }
}
```

_Alignof

C11

Interroge l'exigence d'alignement pour le type spécifié. L'exigence d'alignement est une puissance intégrale positive de 2 représentant le nombre d'octets entre lesquels deux objets du type peuvent être alloués. En C, l'exigence d'alignement est mesurée en `size_t` .

Le nom du type peut ne pas être un type incomplet ni un type de fonction. Si un tableau est utilisé comme type, le type de l'élément de tableau est utilisé.

Cet opérateur est souvent accessible via la macro de commodité `alignof` de `<stdalign.h>` .

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Sortie possible:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://fr.cppreference.com/w/c/language/_Alignof

Comportement en court-circuit des opérateurs logiques

La mise en court-circuit est une fonctionnalité qui permet d'évaluer certaines parties d'une condition (if / while / ...) lorsque cela est possible. Dans le cas d'une opération logique sur deux opérandes, le premier opérande est évalué (à true ou false) et s'il y a un verdict (le premier opérande est faux avec &&, le premier opérande est vrai avec ||) le deuxième opérande est non évalué.

Exemple:

```
#include <stdio.h>

int main(void) {
  int a = 20;
  int b = -5;

  /* here 'b == -5' is not evaluated,
     since a 'a != 20' is false. */
  if (a != 20 && b == -5) {
    printf("I won't be printed!\n");
  }

  return 0;
}
```

Vérifiez vous-même:

```
#include <stdio.h>

int print(int i) {
  printf("print function %d\n", i);
  return i;
}

int main(void) {
  int a = 20;

  /* here 'print(a)' is not called,
     since a 'a != 20' is false. */
  if (a != 20 && print(a)) {
    printf("I won't be printed!\n");
  }

  /* here 'print(a)' is called,
     since a 'a == 20' is true. */
  if (a == 20 && print(a)) {
    printf("I will be printed!\n");
  }

  return 0;
}
```

Sortie:

```
$ ./a.out
print function 20
I will be printed!
```

La mise en court-circuit est importante lorsque vous souhaitez éviter d'évaluer des termes coûteux (en termes de calcul). De plus, cela peut fortement affecter le déroulement de votre programme, comme dans ce cas: Pourquoi ce programme imprime-t-il "forked!" 4 fois?

Lire Les opérateurs en ligne: https://riptutorial.com/fr/c/topic/256/les-operateurs

Chapitre 35: Les syndicats

Examples

Différence entre struct et union

Cela montre que les membres d'union partagent la mémoire et que les membres de la structure ne partagent pas la mémoire.

```
#include <stdio.h>
#include <string.h>

union My_Union
{
  int variable_1;
  int variable_2;
};

struct My_Struct
{
  int variable_1;
  int variable_2;
};

int main (void)
{
  union My_Union u;
  struct My_Struct s;
  u.variable_1 = 1;
  u.variable_2 = 2;
  s.variable_1 = 1;
  s.variable_2 = 2;
  printf ("u.variable_1: %i\n", u.variable_1);
  printf ("u.variable_2: %i\n", u.variable_2);
  printf ("s.variable_1: %i\n", s.variable_1);
  printf ("s.variable_2: %i\n", s.variable_2);
  printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
  printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
  return 0;
}
```

Utiliser les unions pour réinterpréter les valeurs

Certaines implémentations C permettent au code d'écrire sur un membre d'un type d'union puis de lire sur un autre afin d'effectuer une sorte de réinterprétation du cast (analyse du nouveau type en tant que représentation binaire de l'ancien).

Il est important de noter que ceci n'est pas autorisé par la norme C actuelle ou passée et entraînera un comportement indéfini, mais les compilateurs proposent néanmoins une extension très commune (vérifiez donc vos documents de compilation si vous envisagez de le faire). .

Un exemple réel de cette technique est l'algorithme "Racine Carré Inverse Rapide" qui repose sur

les détails d'implémentation des nombres à virgule flottante IEEE 754 pour effectuer une racine carrée inverse plus rapide que l'utilisation des opérations à virgule flottante. (ce qui est très dangereux et enfreint la règle de l'aliasing strict) ou par une union (qui est toujours un comportement non défini mais fonctionne dans de nombreux compilateurs):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input;     /* Assign to the float member */
    i = x.intMember;           /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i;           /* Assign to the integer member */
    f = x.floatMember;         /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

Cette technique était largement utilisée en infographie et dans les jeux dans le passé en raison de sa plus grande rapidité par rapport aux opérations en virgule flottante.

Écrire à un membre du syndicat et lire d'un autre membre

Les membres d'un syndicat partagent le même espace en mémoire. Cela signifie que l'écriture sur un membre écrase les données dans tous les autres membres et que la lecture d'un membre entraîne les mêmes données que la lecture de tous les autres membres. Cependant, comme les membres d'union peuvent avoir différents types et tailles, les données lues peuvent être interprétées différemment, voir http://www.riptutorial.com/c/example/9399/using-unions-to-reinterpret-values

L'exemple simple ci-dessous montre une union avec deux membres, tous deux du même type. Cela montre que l'écriture dans le membre `m_1` entraîne la lecture de la valeur écrite du membre `m_2` et l'écriture dans le membre `m_2` entraîne la lecture de la valeur écrite du membre `m_1` .

```
#include <stdio.h>

union my_union /* Define union */
{
    int m_1;
    int m_2;
};

int main (void)
{
    union my_union u;             /* Declare union */
    u.m_1 = 1;                    /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
```

```
    u.m_2 = 2;                    /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}
```

Résultat

```
u.m_2: 1
u.m_1: 2
```

Lire Les syndicats en ligne: https://riptutorial.com/fr/c/topic/7645/les-syndicats

Chapitre 36: Listes liées

Remarques

Le langage C ne définit pas une structure de données de liste chaînée. Si vous utilisez C et avez besoin d'une liste chaînée, vous devez soit utiliser une liste liée à partir d'une bibliothèque existante (telle que GLib), soit écrire votre propre interface de liste chaînée. Cette rubrique présente des exemples de listes liées et de listes à double lien pouvant servir de point de départ pour écrire vos propres listes chaînées.

Liste liée individuellement

La liste contient des nœuds composés d'un lien appelé suivant.

Structure de données

```
struct singly_node
{
  struct singly_node * next;
};
```

Liste doublement liée

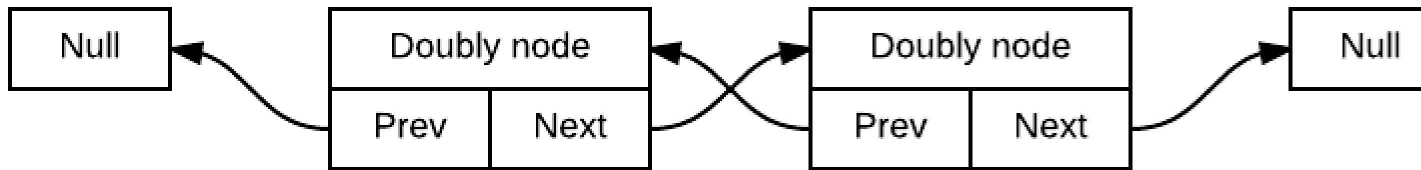
La liste contient des nœuds composés de deux liens appelés précédent et suivant. Les liens font normalement référence à un nœud de même structure.

Structure de données

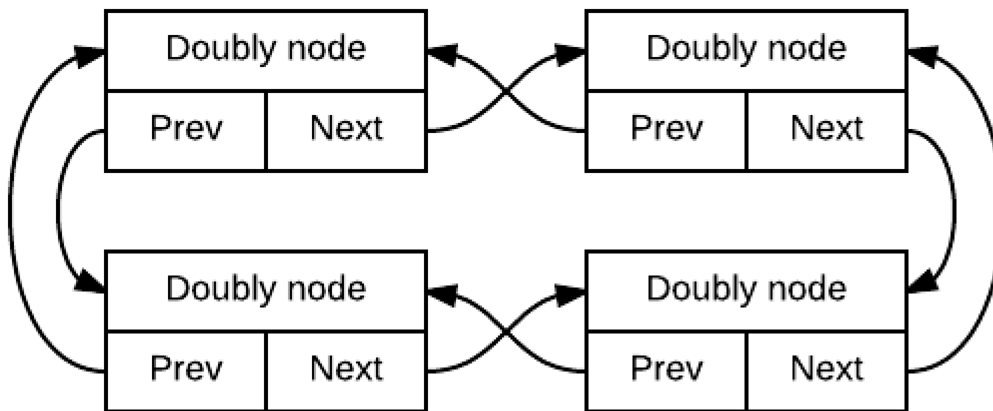
```
struct doubly_node
{
  struct doubly_node * prev;
  struct doubly_node * next;
};
```

Topoliges

Linéaire ou ouverte



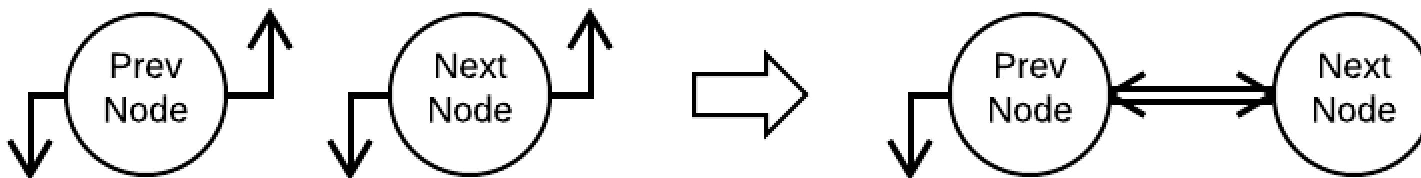
Circulaire ou anneau



Procédures

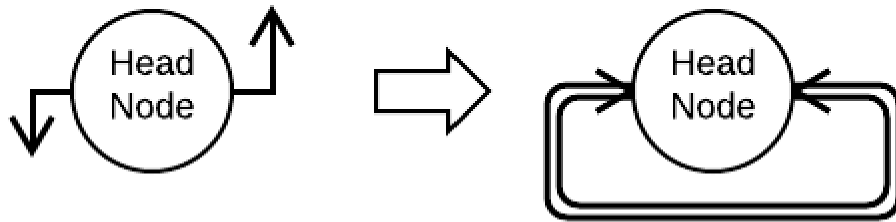
Lier

Lier deux nœuds ensemble.



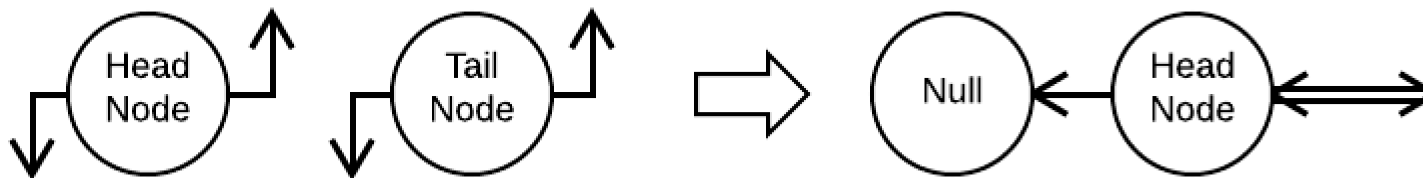

```
void doubly_node_bind (struct doubly_node * prev, struct doubly_node * next)
{
  prev->next = next;
  next->prev = prev;
}
```

Faire une liste liée de façon circulaire




```
void doubly_node_make_empty_circularly_list (struct doubly_node * head)
{
  doubly_node_bind (head, head);
}
```

Faire une liste liée linéairement




```
void doubly_node_make_empty_linear_list (struct doubly_node * head, struct doubly_node * tail)
{
  head->prev = NULL;
  tail->next = NULL;
  doubly_node_bind (head, tail);
}
```

Insertion

Supposons qu'une liste vide contient toujours un nœud au lieu de NULL. Les procédures d’insertion ne doivent alors pas prendre en compte la valeur NULL.

```
void doubly_node_insert_between
(struct doubly_node * prev, struct doubly_node * next, struct doubly_node * insertion)
{
  doubly_node_bind (prev, insertion);
  doubly_node_bind (insertion, next);
}

void doubly_node_insert_before
(struct doubly_node * tail, struct doubly_node * insertion)
{
  doubly_node_insert_between (tail->prev, tail, insertion);
}

void doubly_node_insert_after
```

```
(struct doubly_node * head, struct doubly_node * insertion)
{
  doubly_node_insert_between (head, head->next, insertion);
}
```

Examples

Insertion d'un nœud au début d'une liste liée séparément

Le code ci-dessous demandera des chiffres et continuera à les ajouter au début d'une liste chaînée.

```
/* This program will demonstrate inserting a node at the beginning of a linked list */

#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};


void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
  struct Node* headNode;
  headNode = NULL; /* Initialize our first node pointer to be NULL. */
  size_t listSize, i;
  do {
    printf("How many numbers would you like to input?\n");
  } while(1 != scanf("%zu", &listSize));

  for (i = 0; i < listSize; i++) {
    int numToAdd;
    do {
      printf("Enter a number:\n");
    } while (1 != scanf("%d", &numToAdd));

    insert_node (&headNode, numToAdd);
    printf("Current list after your inserted node: \n");
    print_list(headNode);
  }

  return 0;
}

void print_list (struct Node *head) {
  struct node* currentNode = head;

  /* Iterate through each link. */
  while (currentNode != NULL) {
      printf("Value: %d\n", currentNode->data);
      currentNode = currentNode -> next;
  }
}
```

```
void insert_node (struct Node **head, int nodeValue) {
  struct Node *currentNode = malloc(sizeof *currentNode);
  currentNode->data = nodeValue;
  currentNode->next = (*head);

  *head = currentNode;
}
```

Explication pour l'insertion de nœuds

Afin de comprendre comment nous ajoutons des nœuds au début, examinons les scénarios possibles:

1. La liste est vide, nous devons donc ajouter un nouveau nœud. Dans ce cas, notre mémoire ressemble à ceci: `HEAD` est un pointeur sur le premier nœud:

```
| HEAD | --> NULL
```

La ligne `currentNode->next = *headNode;` assignera la valeur de `currentNode->next` pour être `NULL` puisque `headNode` commence à l'origine à la valeur `NULL` .

Maintenant, nous voulons définir notre pointeur de nœud principal pour qu'il pointe vers notre nœud actuel.

```
  -----      -------------
 |HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
  -----      -------------
```

Ceci est fait avec `*headNode = currentNode;`

2. La liste est déjà remplie. nous devons ajouter un nouveau nœud au début. Par souci de simplicité, commençons par 1 nœud:

```
 -----    -----------
 HEAD --> FIRST NODE --> NULL
 -----    -----------
```

Avec `currentNode->next = *headNode` , la structure de données ressemble à ceci:

```
 ---------        -----    ---------------------
 currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
 ---------        -----    ---------------------
```

Ce qui doit évidemment être modifié depuis que `*headNode` doit pointer sur `currentNode` .

```
 ----    -----------    ---------------
 HEAD -> currentNode -->     NODE       -> NULL
 ----    -----------    ---------------
```

Ceci est fait avec `*headNode = currentNode;`

Insérer un noeud à la nième position

Jusqu'à présent, nous avons envisagé d' insérer un nœud au début d'une liste liée séparément . Cependant, la plupart du temps, vous souhaiterez également pouvoir insérer des nœuds ailleurs. Le code écrit ci-dessous montre comment il est possible d'écrire une fonction `insert()` pour insérer des noeuds *n'importe où* dans les listes liées.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
  struct Node *head = NULL; /* Initialize the list to be empty */

  /* Insert nodes at positions with values: */
  head = insert(head, 1, 0);
  head = insert(head, 100, 1);
  head = insert(head, 21, 2);
  head = insert(head, 2, 3);
  head = insert(head, 5, 4);
  head = insert(head, 42, 2);

  print_list(head);
  return 0;
}

struct Node* insert(struct Node* head, int value, size_t position) {
  size_t i = 0;
  struct Node *currentNode;

  /* Create our node */
  currentNode = malloc(sizeof *currentNode);
  /* Check for success of malloc() here! */

  /* Assign data */
  currentNode->data = value;

  /* Holds a pointer to the 'next' field that we have to link to the new node.
     By initializing it to &head we handle the case of insertion at the beginning. */
  struct Node **nextForPosition = &head;
  /* Iterate to get the 'next' field we are looking for.
     Note: Insert at the end if position is larger than current number of elements. */
  for (i = 0; i < position && *nextForPosition != NULL; i++) {
      /* nextForPosition is pointing to the 'next' field of the node.
         So *nextForPosition is a pointer to the next node.
         Update it with a pointer to the 'next' field of the next node. */
      nextForPosition = &(*nextForPosition)->next;
  }
```

```
  /* Here, we are taking the link to the next node (the one our newly inserted node should
  point to) by dereferencing nextForPosition, which points to the 'next' field of the node
  that is in the position we want to insert our node at.
  We assign this link to our next value. */
  currentNode->next = *nextForPosition;

  /* Now, we want to correct the link of the node before the position of our
  new node: it will be changed to be a pointer to our new node. */
  *nextForPosition = currentNode;

  return head;
}

void print_list (struct Node* head) {
  /* Go through the list of nodes and print out the data in each node */
  struct Node* i = head;
  while (i != NULL) {
    printf("%d\n", i->data);
    i = i->next;
  }
}
```

Inverser une liste chaînée

Vous pouvez également effectuer cette tâche de manière récursive, mais j'ai choisi dans cet exemple d'utiliser une approche itérative. Cette tâche est utile si vous insérez tous vos nœuds au début d'une liste chaînée . Voici un exemple:

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
  int data;
  struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);
void reverse_list(struct Node **headNode);


int main(void) {
  int i;
  struct Node *head = NULL;

  for(i = 1; i <= NUM_ITEMS; i++) {
    insert_node(&head, i, i);
  }
  print_list(head);

  printf("I will now reverse the linked list\n");
  reverse_list(&head);
  print_list(head);
  return 0;
}
```

```
void print_list(struct Node *headNode) {
  struct Node *iterator;

  for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
    printf("Value: %d\n", iterator->data);
  }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
  int i;
  struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *nodeBeforePosition = *headNode;

  currentNode->data = nodeValue;

  if(position == 1) {
    currentNode->next = *headNode;
    *headNode = currentNode;
    return;
  }

  for (i = 0; i < position - 2; i++) {
    nodeBeforePosition = nodeBeforePosition->next;
  }

  currentNode->next = nodeBeforePosition->next;
  nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
  struct Node *iterator = *headNode;
  struct Node *previousNode = NULL;
  struct Node *nextNode = NULL;

  while (iterator != NULL) {
    nextNode = iterator->next;
    iterator->next = previousNode;
    previousNode = iterator;
    iterator = nextNode;
  }

  /* Iterator will be NULL by the end, so the last node will be stored in
  previousNode.  We will set the last node to be the headNode */
  *headNode = previousNode;
}
```

Explication de la méthode de la liste inversée

Nous commençons le `previousNode` par `NULL` , puisque nous savons à la première itération de la boucle, si nous cherchons le noeud avant le premier noeud principal, ce sera `NULL` . Le premier nœud deviendra le dernier nœud de la liste et la variable suivante sera naturellement `NULL` .

Fondamentalement, l'idée d'inverser la liste des liens ici est que nous inversons les liens eux-mêmes. Le membre suivant de chaque nœud deviendra le nœud avant, comme ceci:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Où chaque nombre représente un nœud. Cette liste deviendrait:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Enfin, la tête devrait plutôt pointer vers le 5ème nœud et chaque nœud devrait pointer vers le nœud précédent.

Le nœud 1 devrait pointer sur `NULL` car il n'y avait rien avant lui. Le nœud 2 doit pointer sur le nœud 1, le nœud 3 doit pointer sur le nœud 2, et cetera.

Cependant, il y a *un petit problème* avec cette méthode. Si nous rompons le lien vers le nœud suivant et le changeons pour le nœud précédent, nous ne pourrons pas traverser le nœud suivant dans la liste, car le lien vers celui-ci a disparu.

La solution à ce problème consiste simplement à stocker l'élément suivant dans une variable (`nextNode`) avant de modifier le lien.

Une liste doublement liée

Un exemple de code montrant comment les nœuds peuvent être insérés dans une liste à double lien, comment la liste peut facilement être inversée et comment elle peut être imprimée en sens inverse.

```
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of use */
struct Node {
  /* Sometimes a key is also stored and used in the functions */
  int data;
  struct Node* next;
  struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
  /* Sometimes in a doubly linked list the last node is also stored */
  struct Node *head = NULL;

  printf("Insert a node at the beginning of the list.\n");
  insert_at_beginning(&head, 5);
  print_list(head);

  printf("Insert a node at the beginning, and then print the list backwards\n");
  insert_at_beginning(&head, 10);
  print_list_backwards(head);

  printf("Insert a node at the end, and then print the list forwards.\n");
```

```
  insert_at_end(&head, 15);
  print_list(head);

  free_list(head);

  return 0;
}

void print_list_backwards(struct Node *headNode) {
  if (NULL == headNode)
  {
    return;
  }
  /*
  Iterate through the list, and once we get to the end, iterate backwards to print
  out the items in reverse order (this is done with the pointer to the previous node).
  This can be done even more easily if a pointer to the last node is stored.
  */
  struct Node *i = headNode;
  while (i->next != NULL) {
    i = i->next; /* Move to the end of the list */
  }

  while (i != NULL) {
    printf("Value: %d\n", i->data);
    i = i->previous;
  }
}

void print_list(struct Node *headNode) {
  /* Iterate through the list and print out the data member of each node */
  struct Node *i;
  for (i = headNode; i != NULL; i = i->next) {
    printf("Value: %d\n", i->data);
  }
}

void insert_at_beginning(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }
  /*
  This is done similarly to how we insert a node at the beginning of a singly linked
  list, instead we set the previous member of the structure as well
  */
  currentNode = malloc(sizeof *currentNode);

  currentNode->next = NULL;
  currentNode->previous = NULL;
  currentNode->data = value;

  if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
  }

  currentNode->next = *pheadNode;
```

```
  (*pheadNode)->previous = currentNode;
  *pheadNode = currentNode;
}

void insert_at_end(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }

  /*
  This can, again be done easily by being able to have the previous element.  It
  would also be even more useful to have a pointer to the last node, which is commonly
  used.
  */

  currentNode = malloc(sizeof *currentNode);
  struct Node *i = *pheadNode;

  currentNode->data = value;
  currentNode->next = NULL;
  currentNode->previous = NULL;

  if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
  }

  while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
  }

  i->next = currentNode;
  currentNode->previous = i;
}

void free_list(struct Node *node) {
  while (node != NULL) {
    struct Node *next = node->next;
    free(node);
    node = next;
  }
}
```

Notez que parfois, stocker un pointeur sur le dernier nœud est utile (il est plus efficace de simplement pouvoir sauter directement à la fin de la liste que de devoir parcourir la fin):

```
struct Node *lastNode = NULL;
```

Dans ce cas, il est nécessaire de le mettre à jour en cas de modification de la liste.

Parfois, une clé est également utilisée pour identifier des éléments. C'est simplement un membre de la structure Node:

```
struct Node {
  int data;
```

```
  int key;
  struct Node* next;
  struct Node* previous;
};
```

La clé est ensuite utilisée lorsque des tâches sont effectuées sur un élément spécifique, comme la suppression d'éléments.

Lire Listes liées en ligne: https://riptutorial.com/fr/c/topic/560/listes-liees

Chapitre 37: Littéraux composés

Syntaxe

- (type) {liste d'initialisation}

Remarques

C standard dit dans C11-§6.5.2.5 / 3:

> Une expression postfixe composée d'un nom de type entre parenthèses suivi d'une accolade contenant une liste d'initialisateurs est un *littéral composé* . Il fournit un objet non nommé dont la valeur est donnée par la liste d'initialisation. [99)]

et la note de bas de page 99 dit:

> Notez que cela diffère d'une expression de distribution. Par exemple, un transtypage spécifie une conversion en types scalaires ou **nuls** uniquement, et le résultat d'une expression de distribution n'est pas une lvalue.

Notez que:

> Les littéraux de chaîne et les littéraux composés avec des types qualifiés de constants n'ont pas besoin de désigner des objets distincts. [101)]
>
> 101) Cela permet aux implémentations de partager le stockage pour les littéraux de chaîne et les littéraux composés constants avec des représentations identiques ou superposées.

L'exemple est donné en standard:
C11-§6.5.2.5 / 13:

> Tout comme les littéraux de chaîne, les littéraux composés qualifiés peuvent être placés en mémoire morte et peuvent même être partagés. Par exemple,
>
> ```
> (const char []){"abc"} == "abc"
> ```
>
> peut donner 1 si le stockage des littéraux est partagé.

Examples

Définition / Initialisation des littéraux composés

Un littéral composé est un objet sans nom créé dans la portée où il est défini. Le concept a été introduit pour la première fois dans la norme C99. Un exemple de littéral composé est

Exemples de la norme C, C11-§6.5.2.5 / 9:

```
int *p = (int [2]){ 2, 4 };
```

`p` est initialisé à l'adresse du premier élément d'un tableau sans nom de deux ints.

Le littéral composé est une lvalue. La durée de stockage de l'objet non nommé est soit statique (si le littéral apparaît dans la portée du fichier), soit automatique (si le littéral apparaît dans la portée du bloc) et dans ce dernier cas la durée de vie de l'objet

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

> On attribue à `p` l'adresse du premier élément d'un tableau de deux ints, le premier ayant la valeur pointée précédemment par `p` et le second, zéro. [...]

Ici, `p` reste valide jusqu'à la fin du bloc.

Littéral composé avec désignateurs

(également de C11)

```
  struct point {
    unsigned x;
    unsigned y;
  };

  extern void drawline(struct point, struct point);

 // used somewhere like this
 drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

Une ligne de `drawline` fictive reçoit deux arguments de type `struct point` . Le premier a des valeurs de coordonnées `x == 1` et `y == 1` , tandis que le second a `x == 3` et `y == 4`

Littéral composé sans spécifier la longueur du tableau

```
int *p = (int []){ 1, 2, 3};
```

Dans ce cas, la taille du tableau n'est pas spécifiée, alors elle sera déterminée par la longueur de l'initialiseur.

Littéral composé dont la longueur de l'initialiseur est inférieure à celle spécifiée

```
int *p = (int [10]){1, 2, 3};
```

le reste des éléments du littéral composé sera initialisé à `0` implicitement.

Littéral composé en lecture seule

Notez qu'un littéral composé est une lvalue et que ses éléments peuvent donc être modifiables. Un littéral composé en *lecture seule* peut être spécifié à l'aide du qualificateur `const` sous la forme `(const int[]){1,2}` .

Littéral composé contenant des expressions arbitraires

À l'intérieur d'une fonction, un littéral composé, comme pour toute initialisation depuis C99, peut avoir des expressions arbitraires.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

Lire Littéraux composés en ligne: https://riptutorial.com/fr/c/topic/4135/litteraux-composes

Chapitre 38: Littéraux pour les nombres, les caractères et les chaînes

Remarques

Le terme **littéral** est couramment utilisé pour décrire une séquence de caractères dans un code C qui désigne une valeur constante telle qu'un nombre (par exemple `0`) ou une chaîne (par exemple, `"C"`). Strictement parlant, le standard utilise le terme **constant** pour les constantes entières, les constantes flottantes, les constantes d'énumération et les constantes de caractères, réservant le terme «littéral» aux littéraux de chaîne, mais ce n'est pas un usage courant.

Les littéraux peuvent avoir des **préfixes** ou des **suffixes** (mais pas les deux) qui sont des caractères supplémentaires pouvant démarrer ou terminer un littéral pour modifier son type par défaut ou sa représentation.

Examples

Littéraux entiers

Les littéraux entiers sont utilisés pour fournir des valeurs intégrales. Trois bases numériques sont supportées, indiquées par des préfixes:

Base	Préfixe	Exemple
Décimal	Aucun	`5`
Octal	`0`	`0345`
Hexadécimal	`0x` ou `0X`	`0x12AB` , `0X12AB` , `0x12ab` , `0x12Ab`

Notez que cette écriture n'inclut aucun signe, donc les littéraux entiers sont toujours positifs. Quelque chose comme `-1` est traité comme une expression avec un entier littéral (`1`) qui est annulé par un `-`

Le type d'un littéral entier décimal est le premier type de données pouvant correspondre à la valeur d' `int` et de `long` . Depuis C99, `long long` est également pris en charge pour les très grands littéraux.

Le type d'un littéral entier octal ou hexadécimal est le premier type de données pouvant correspondre à la valeur de `int` , `unsigned` , `long` et `unsigned long` . Depuis C99, `long long` et `unsigned long long` sont également pris en charge pour les très grands littéraux.

En utilisant différents suffixes, le type par défaut d'un littéral peut être modifié.

Suffixe	Explication
`L` , `l`	`long int`
`LL` , `ll` (depuis C99)	`long long int`
`U` , `u`	`unsigned`

Les suffixes U et L / LL peuvent être combinés dans n'importe quel ordre et cas. C'est une erreur de dupliquer des suffixes (par exemple, fournir deux suffixes `U`) même s'ils ont des cas différents.

Littéraux de chaîne

Les littéraux de chaîne sont utilisés pour spécifier des tableaux de caractères. Ce sont des séquences de caractères entre guillemets (par exemple `"abcd"` et ont le type `char*`).

Le préfixe `L` fait du littéral un tableau de caractères large, de type `wchar_t*` . Par exemple, `L"abcd"` .

Depuis C11, il existe d'autres préfixes de codage, similaires à `L` :

préfixe	type de base	codage
aucun	`char`	dépend de la plate-forme
`L`	`wchar_t`	dépend de la plate-forme
`u8`	`char`	UTF-8
`u`	`char16_t`	généralement UTF-16
`U`	`char32_t`	généralement UTF-32

Pour les deux derniers, il peut être interrogé avec des macros de test de fonctionnalité si le codage correspond effectivement au codage UTF correspondant.

Littéraux à virgule flottante

Les littéraux à virgule flottante sont utilisés pour représenter les nombres réels signés. Les suffixes suivants peuvent être utilisés pour spécifier le type d'un littéral:

Suffixe	Type	Exemples
aucun	`double`	`3.1415926` `-3E6`
`f` , `F`	`float`	`3.1415926f` `2.1E-6F`
`l` , `L`	`long double`	`3.1415926L` `1E126L`

Pour utiliser ces suffixes, le littéral *doit* être un littéral à virgule flottante. Par exemple, `3f` est une

erreur, puisque `3` est un littéral entier, tandis que `3.f` ou `3.0f` sont corrects. Pour le `long double` , la recommandation est de toujours utiliser le capital `L` pour des raisons de lisibilité.

Littéraux de caractère

Les littéraux de caractères sont un type spécial de littéraux entiers utilisés pour représenter un caractère. Ils sont entre guillemets simples, par exemple `'a'` et ont le type `int` . La valeur du littéral est une valeur entière en fonction du jeu de caractères de la machine. Ils ne permettent pas les suffixes.

Le préfixe `L` précédant un littéral de caractère en fait un caractère large de type `wchar_t` . De même, les préfixes C11 `u` et `U` font des caractères larges de type `char16_t` et `char32_t` , respectivement.

Lorsque vous souhaitez représenter certains caractères spéciaux, tels qu'un caractère non imprimable, des séquences d'échappement sont utilisées. Les séquences d'échappement utilisent une séquence de caractères traduite dans un autre personnage. Toutes les séquences d'échappement se composent de deux caractères ou plus, le premier d'entre eux étant une barre oblique inverse `\` . Les caractères qui suivent immédiatement la barre oblique inverse déterminent le caractère littéral interprété par la séquence.

Séquence d'échappement	Personnage représenté
`\b`	Retour arrière
`\f`	Flux de formulaire
`\n`	Line feed (nouvelle ligne)
`\r`	Retour de voiture
`\t`	Onglet horizontal
`\v`	Onglet vertical
`\\`	Barre oblique inverse
`\'`	Guillemet simple
`\"`	Guillemet double
`\?`	Point d'interrogation
`\nnn`	Valeur octale
`\xnn` ...	Valeur hexadécimale

C89

Séquence d'échappement	Personnage représenté
`\a`	Alerte (bip, cloche)

C99

Séquence d'échappement	Personnage représenté
`\unnnn`	Nom de personnage universel
`\Unnnnnnnn`	Nom de personnage universel

Un nom de caractère universel est un point de code Unicode. Un nom de caractère universel peut correspondre à plusieurs caractères. Les chiffres `n` sont interprétés comme des chiffres hexadécimaux. Selon l'encodage UTF en cours d' utilisation, une séquence de nom de caractère universel peut donner lieu à un point de code qui se compose de plusieurs caractères, au lieu d'un seul normale `char` caractère.

Lors de l'utilisation de la séquence d'échappement du saut de ligne en mode texte, elle est convertie en octet ou en octet de nouvelle ligne spécifique au système d'exploitation.

La séquence d'échappement de point d'interrogation est utilisée pour éviter les trigraphes . Par exemple, `??/` est compilé en tant que trigraphe représentant un caractère barre oblique inverse `'\'` , mais en utilisant `?\?/` Entraînerait la *chaîne* `"??/"` .

Il peut y avoir un, deux ou trois nombres octaux `n` dans la séquence d'échappement de la valeur octale.

Lire Littéraux pour les nombres, les caractères et les chaînes en ligne:
https://riptutorial.com/fr/c/topic/3455/litteraux-pour-les-nombres--les-caracteres-et-les-chaines

Chapitre 39: Mathématiques standard

Syntaxe

- #include <math.h>
- double pow (double x, double y);
- float powf (float x, float y);
- long double double (long double x, long double y);

Remarques

1. Pour `-lm` lien avec la bibliothèque mathématique, utilisez `-lm` avec les indicateurs gcc.
2. Un programme portable qui doit vérifier une erreur à partir d'une fonction mathématique doit définir `errno` à zéro et appeler l'appel suivant `feclearexcept(FE_ALL_EXCEPT);` avant d'appeler une fonction mathématique. A son retour de la fonction mathématique, si `errno` est différent de zéro, ou si l'appel suivant renvoie un non-zéro `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` alors une erreur s'est produite dans la fonction mathématique. Lisez la page de manuel de math_error pour plus d'informations.

Examples

Reste à virgule flottante double précision: fmod ()

Cette fonction renvoie le reste en virgule flottante de la division de `x/y` . La valeur renvoyée a le même signe que x.

```
#include <math.h> /* for fmod() */
#include <stdio.h> /* for printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);

    printf("%lf\n", modulus); /* f is the same as lf. */

    return 0;
}
```

Sortie:

```
4.90000
```

Important: utilisez cette fonction avec précaution, car elle peut renvoyer des valeurs inattendues en raison du fonctionnement des valeurs à virgule flottante.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f\n", fmod(1, 0.1));
    printf("%19.17f\n", fmod(1, 0.1));
    return 0;
}
```

Sortie:

```
0.1
0.09999999999999995
```

Précision simple et double virgule flottante en double précision: fmodf (), fmodl ()

C99

Ces fonctions retournent le reste en virgule flottante de la division de `x/y` . La valeur renvoyée a le même signe que x.

Précision unique:

```
#include <math.h> /* for fmodf() */
#include <stdio.h> /* for printf() */

int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f\n", modulus); /* lf would do as well as modulus gets promoted to double. */
}
```

Sortie:

```
4.90000
```

Double Double Precision:

```
#include <math.h> /* for fmodl() */
#include <stdio.h> /* for printf() */

int main(void)
{
    long double x = 10.0;
    long double y = 5.1;

    long double modulus = fmodl(x, y);
```

```
    printf("%Lf\n", modulus); /* Lf is for long double. */
}
```

Sortie:

```
4.90000
```

Fonctions d'alimentation - pow (), powf (), powl ()

L'exemple de code suivant calcule la somme des séries *1 + 4 (3 + 3 ^ 2 + 3 ^ 3 + 3 ^ 4 + ... + 3 ^ N) en* utilisant la famille pow () de la bibliothèque mathématique standard.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
        double pwr, sum=0;
        int i, n;

        printf("\n1+4(3+3^2+3^3+3^4+...+3^N)=?\nEnter N:");
        scanf("%d",&n);
        if (n<=0) {
                printf("Invalid power N=%d", n);
                return -1;
        }

        for (i=0; i<n+1; i++) {
                errno = 0;
                feclearexcept(FE_ALL_EXCEPT);
                pwr = powl(3,i);
                if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                        FE_UNDERFLOW)) {
                        perror("Math Error");
                }
                sum += i ? pwr : 0;
                printf("N= %d\tS= %g\n", i, 1+4*sum);
        }

        return 0;
}
```

Exemple de sortie:

```
1+4(3+3^2+3^3+3^4+...+3^N)=?
Enter N:10
N= 0    S= 1
N= 1    S= 13
N= 2    S= 49
N= 3    S= 157
N= 4    S= 481
N= 5    S= 1453
N= 6    S= 4369
N= 7    S= 13117
```

```
N= 8    S= 39361
N= 9    S= 118093
N= 10    S= 354289
```

Lire Mathématiques standard en ligne: https://riptutorial.com/fr/c/topic/3170/mathematiques-standard

Chapitre 40: Multithreading

Introduction

En C11, il existe une bibliothèque de threads standard, `<threads.h>` , mais aucun compilateur connu ne l'implémente encore. Ainsi, pour utiliser le multithreading en C, vous devez utiliser des implémentations spécifiques à la plate-forme telles que la bibliothèque de threads POSIX (souvent appelée pthreads) à l'aide de l'en-tête `pthread.h` .

Syntaxe

- thrd_t // Type d'objet complet défini par l'implémentation identifiant un thread
- int thrd_create (thrd_t * thr, thrd_start_t func, void * arg); // Crée un fil
- int thrd_equal (thrd_t thr0, thrd_t thr1); // Vérifie si les arguments font référence au même thread
- thr_t thrd_current (void); // Retourne l'identifiant du thread qui l'appelle
- int thrd_sleep (const struct timespec * duration, struct timespec * restant); // Suspendre l'exécution du thread d'appel pendant au moins un temps donné
- annuler thrd_yield (vide); // Autorise l'exécution d'autres threads au lieu du thread qui l'appelle
- _Noreturn void thrd_exit (int res); // Termine le thread le thread qui l'appelle
- int thrd_detatch (thrd_t thr; // Détache un thread donné de l'environnement actuel
- int thrd_join (thrd_t thr, int * res); // Bloque le thread actuel jusqu'à la fin du thread

Remarques

L'utilisation de threads peut introduire un comportement non défini supplémentaire tel qu'un http://www.riptutorial.com/c/example/2622/data-race . Pour un accès sans race aux variables partagées entre différents threads, C11 fournit la fonctionnalité de `mtx_lock()` ou le type de données (facultatif) http://www.riptutorial.com/c/topic/4924/atomics et les fonctions associées dans `stdatomic.h` .

Examples

C11 Threads exemple simple

```
#include <threads.h>
#include <stdio.h>

int run(void *arg)
{
    printf("Hello world of C11 threads.");

    return 0;
}
```

```
int main(int argc, const char *argv[])
{
    thrd_t thread;
    int result;

    thrd_create(&thread, run, NULL);

    thrd_join(&thread, &result);

    printf("Thread return %d at the end\n", result);
}
```

Lire Multithreading en ligne: https://riptutorial.com/fr/c/topic/10489/multithreading

Chapitre 41: Paramètres de fonction

Remarques

En C, il est courant d'utiliser des valeurs de retour pour indiquer des erreurs qui se produisent; et pour retourner des données en utilisant des pointeurs passés. Cela peut être fait pour plusieurs raisons; y compris ne pas avoir à allouer de la mémoire sur le tas ou à l'aide d'allocation statique au point où la fonction est appelée.

Examples

Utilisation de paramètres de pointeur pour renvoyer plusieurs valeurs

Un modèle courant en C, pour imiter facilement le retour de plusieurs valeurs d'une fonction, consiste à utiliser des pointeurs.

```
#include <stdio.h>

void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

Passer des tableaux à des fonctions

```
int getListOfFriends(size_t size, int friend_indexes[]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = i;
  }
}
```

C99 C11

```
/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
   In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {
```

```
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = 1;
  }
}
```

Ici, le `static` à l'intérieur du paramètre `[]` du paramètre function demande que le tableau d'arguments ait au moins autant d'éléments que spécifié (éléments de `size`). Pour pouvoir utiliser cette fonctionnalité, nous devons nous assurer que le paramètre `size` est placé avant le paramètre array dans la liste.

Utilisez `getListOfFriends()` comme ceci:

```
#define LIST_SIZE (50)

int main(void) {
  size_t size_of_list = LIST_SIZE;
  int friends_indexes[size_of_list];

  getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                     address of its 1st element:
                                                                      &friend_indexes[0] */

  /* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

  return 0;
}
```

Voir également

Passer des tableaux multidimensionnels à une fonction

Les paramètres sont passés par valeur

En C, tous les paramètres de fonction sont passés par valeur, donc la modification de ce qui est passé dans les fonctions appelées n'affectera pas les variables locales des fonctions appelantes.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

Vous pouvez utiliser des pointeurs pour laisser les fonctions appelées modifier les variables locales des fonctions appelantes. Notez que ce n'est pas une *référence par référence,* mais que les *valeurs des* pointeurs pointant vers les variables locales sont transmises.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

Cependant, le renvoi de l'adresse d'une variable locale à l'appelé entraîne un comportement indéfini. Voir Déréférencement d'un pointeur sur variable au-delà de sa durée de vie .

Ordre d'exécution des paramètres de la fonction

L'ordre d'exécution des paramètres n'est pas défini dans la programmation en C. Ici, il peut être exécuté de gauche à droite ou de droite à gauche. La commande dépend de l'implémentation.

```
#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}
```

Exemple de fonction renvoyant une structure contenant des valeurs avec des codes d'erreur

La plupart des exemples de fonctions renvoyant une valeur impliquent de fournir un pointeur comme argument pour permettre à la fonction de modifier la valeur pointée, comme ci-dessous. La valeur de retour réelle de la fonction est généralement un type tel que `int` pour indiquer le statut du résultat, qu'il fonctionne ou non.

```
int func (int *pIvalue)
{
    int iRetStatus = 0;             /* Default status is no change */
```

```
    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45;    /* Modify the value pointed to */
        iRetStatus = 1;              /* indicate value was changed */
    }

    return iRetStatus;               /* Return an error code */
}
```

Cependant, vous pouvez également utiliser une `struct` comme valeur de retour, ce qui vous permet de renvoyer à la fois un statut d'erreur et d'autres valeurs. Par exemple.

```
typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
}  RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}
```

Cette fonction pourrait alors être utilisée comme l'exemple suivant.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Ou il pourrait être utilisé comme suit.

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Lire Paramètres de fonction en ligne: https://riptutorial.com/fr/c/topic/1006/parametres-de-fonction

Chapitre 42: Passer des tableaux 2D à des fonctions

Examples

Passer un tableau 2D à une fonction

Passer un tableau 2D à une fonction semble simple et évident et nous écrivons volontiers:

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Mais le compilateur, ici GCC en version 4.9.4, ne l'apprécie pas bien.

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c11 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:16:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, n, m);
        ^
passarr.c:8:6: note: expected ‘int **’ but argument is of type ‘int (*)[2]’
 void fun1(int **, int, int);
```

Les raisons en sont doubles: le problème principal est que les tableaux ne sont pas des pointeurs et le second inconvénient est ce que l’on appelle le *déclin du pointeur* . Passer un tableau à une fonction amènera le tableau à un pointeur sur le premier élément du tableau - dans le cas d'un tableau 2d, il se désintègre en un pointeur vers la première ligne car les tableaux C sont classés

en premier.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Il *est* nécessaire de passer le nombre de lignes, elles ne peuvent pas être calculées.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int rows = ROWS;

  /* works here because array_2d is still in scope and still an array */
  printf("MAIN: %zu\n",sizeof(array_2D)/sizeof(array_2D[0]));

  fun1(array_2D, rows);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
  int i, j;
  int n, m;
```

```
  n = rows;
  /* Works, because that information is passed (as "COLS").
     It is also redundant because that value is known at compile time (in "COLS"). */
  m = (int) (sizeof(a[0])/sizeof(a[0][0]));

  /* Does not work here because the "decay" in "pointer decay" is meant
     literally--information is lost. */
  printf("FUN1: %zu\n",sizeof(a)/sizeof(a[0]));

  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

C99

Le nombre de colonnes est prédéfini et donc fixé au moment de la compilation, mais le prédécesseur de la norme C actuelle (ISO / IEC 9899: 1999, actuellement ISO / IEC 9899: 2011) a implémenté des VLA (TODO: link it) et Bien que la norme actuelle le rende optionnel, presque tous les compilateurs C modernes le supportent (TODO: vérifiez si MS Visual Studio le supporte maintenant).

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];

  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = (i + 1) * (j + 1);
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
```

```
  int i, j;
  int n, m;

  n = rows;
  /* Does not work anymore, no sizes are specified anymore
  m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
  m = cols;


  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Cela ne fonctionne pas, le compilateur se plaint:

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function 'fun1':
passarr.c:168:7: error: invalid use of array with unspecified bounds
       printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

Cela devient un peu plus clair si nous `void fun1(int **a, int rows, int cols)` intentionnellement une erreur dans l'appel de la fonction en changeant la déclaration pour `void fun1(int **a, int rows, int cols)` . Cela amène le compilateur à se plaindre d'une manière différente mais tout aussi nébuleuse

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function 'main':
passarr.c:208:8: warning: passing argument 1 of 'fun1' from incompatible pointer type
   fun1(array_2D, rows, cols);
        ^
passarr.c:185:6: note: expected 'int **' but argument is of type 'int (*)[(sizetype)(cols)]'
 void fun1(int **, int rows, int cols);
```

Nous pouvons réagir de plusieurs manières, dont l'une est de tout ignorer et de faire des jonglages illisibles:

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
```

```
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, *( (*a) + (i * cols + j)));
    }
  }
}
```

Ou nous le faisons bien et transmettons les informations nécessaires à l' `fun1` . Pour ce faire, nous devons réorganiser les arguments à `fun1` : la taille de la colonne doit `fun1` la déclaration du tableau. Pour rester plus lisible, la variable qui contient le nombre de lignes a également changé d’emplacement et est maintenant la première.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int (*)[]);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
```

```
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(rows, cols, array_2D);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int (*a)[cols])
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Cela semble gênant pour certaines personnes, qui estiment que l'ordre des variables ne devrait pas avoir d'importance. Ce n'est pas vraiment un problème, il suffit de déclarer un pointeur et de le laisser pointer vers le tableau.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int **);

int main(int argc, char **argv)
{
  int rows, cols, i, j;


  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }
```

```
  // a "rows" number of pointers to "int". Again a VLA
  int *a[rows];
  // initialize them to point to the individual rows
  for (i = 0; i < rows; i++) {
      a[i] = array_2D[i];
  }

  fun1(rows, cols, a);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int **a)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Utilisation de tableaux plats comme tableaux 2D

La solution la plus simple consiste souvent à faire passer des baies 2D et supérieures sous forme de mémoire plate.

```
/* create 2D array with dimensions determined at runtime */
double *matrix = malloc(width * height * sizeof(double));

/* initialise it (for the sake of illustration we want 1.0 on the diagonal) */
int x, y;
for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* pass it to a subroutine */
 manipulate_matrix(matrix, width, height);


/* do something with the matrix, e.g. scale by 2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;

    for (y = 0; y < height; y++)
```

```
    {
        for (x = 0; x < width; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
}
```

Lire Passer des tableaux 2D à des fonctions en ligne: https://riptutorial.com/fr/c/topic/6862/passer-des-tableaux-2d-a-des-fonctions

Chapitre 43: Pièges communs

Introduction

Cette section traite de certaines des erreurs courantes qu'un programmeur C devrait connaître et qu'il devrait éviter. Pour plus d'informations sur certains problèmes inattendus et leurs causes, reportez-vous à la section Comportement non défini.

Examples

Mélanger des entiers signés et non signés dans des opérations arithmétiques

Il est généralement pas une bonne idée de mélanger `signed` et `unsigned` entiers dans les opérations arithmétiques. Par exemple, qu'est-ce qui sortira de l'exemple suivant?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Comme 1000 est supérieur à -1, vous vous attendez à ce que la sortie soit `a is more than b` , mais ce ne sera pas le cas.

Les opérations arithmétiques entre différents types intégraux sont effectuées dans un type commun défini par les conversions arithmétiques dites habituelles (voir la spécification du langage, 6.3.1.8).

Dans ce cas, le "type commun" est `unsigned int` , car, comme indiqué dans les conversions arithmétiques habituelles ,

> 714 Sinon, si l'opérande qui a un type d'entier non signé a un rang supérieur ou égal au rang du type de l'autre opérande, l'opérande avec un type d'entier signé est converti dans le type de l'opérande avec un type d'entier non signé.

Cela signifie que `int` opérande `b` va se converti en `unsigned int` avant la comparaison.

Lorsque -1 est converti en un `unsigned int` le résultat est la valeur `unsigned int` maximale `unsigned int` possible, qui est supérieure à 1000, ce qui signifie que `a > b` est faux.

Écrire par erreur = au lieu de == lors de la comparaison

L'opérateur `=` est utilisé pour l'affectation.

L'opérateur `==` est utilisé pour la comparaison.

Il faut faire attention à ne pas mélanger les deux. Parfois, on écrit par erreur

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

quand ce qui était vraiment recherché est:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

Le premier attribue la valeur de y à x et vérifie si cette valeur est non nulle, au lieu de faire une comparaison, ce qui équivaut à:

```
if ((x = y) != 0) {
    /* logic */
}
```

Il y a des moments où tester le résultat d'une affectation est destiné et est couramment utilisé, car cela évite d'avoir à dupliquer le code et de devoir traiter la première fois spécialement. Comparer

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

contre

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
        switch (c) {
        ...
        }
        c = getopt_long(argc, argv, short_options, long_options, &option_index);
}
```

Les compilateurs modernes reconnaîtront ce modèle et ne préviendront pas lorsque l’affectation est entre parenthèses comme ci-dessus, mais peuvent avertir pour d’autres utilisations. Par exemple:

```
if (x = y)         /* warning */

if ((x = y))       /* no warning */
```

```
if ((x = y) != 0)  /* no warning; explicit */
```

Certains programmeurs utilisent la stratégie consistant à mettre la constante à gauche de l'opérateur (communément appelée conditions Yoda). Étant donné que les constantes sont des valeurs de valeur, ce style entraînera une erreur du compilateur si le mauvais opérateur a été utilisé.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

Cependant, cela réduit considérablement la lisibilité du code et n'est pas considéré comme nécessaire si le programmeur suit les bonnes pratiques de codage C, et ne permet pas de comparer deux variables afin que ce ne soit pas une solution universelle. De plus, de nombreux compilateurs modernes peuvent donner des avertissements lorsque du code est écrit avec les conditions de Yoda.

Utilisation inconsidérée de points-virgules

Soyez prudent avec les points-virgules. Exemple suivant

```
if (x > a);
   a = x;
```

signifie en réalité:

```
if (x > a) {}
a = x;
```

ce qui signifie que `x` sera assigné à `a` dans tous les cas, ce qui pourrait ne pas être ce que vous vouliez à l'origine.

Parfois, manquer un point-virgule causera également un problème imperceptible:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

Le point-virgule derrière return est manqué, donc day = date [0] sera renvoyé.

Une technique pour éviter ce problème et d'autres problèmes similaires consiste à toujours utiliser des accolades sur les conditionnels et les boucles multi-lignes. Par exemple:

```
if (x > a) {
    a = x;
}
```

Oublier d'allouer un octet supplémentaire pour \ 0

Lorsque vous copiez une chaîne dans un tampon `malloc` , n'oubliez jamais d'ajouter 1 à `strlen` .

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

Ceci est dû au fait que `strlen` n'inclut pas le `\0` final dans la longueur. Si vous prenez l'approche `WRONG` (comme indiqué ci-dessus), lors de l'appel de `strcpy` , votre programme invoquera un comportement indéfini.

Cela s'applique également aux situations où vous lisez une chaîne de longueur maximale connue à partir de `stdin` ou d'une autre source. Par exemple

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer);  /* Ensure that the buffer is not overflowed */
```

Oublier de libérer de la mémoire (fuites de mémoire)

Une bonne pratique de programmation consiste à libérer toute mémoire allouée directement par votre propre code ou implicitement en appelant une fonction interne ou externe, telle qu'une API de bibliothèque telle que `strdup()` . Si vous ne parvenez pas à libérer de la mémoire, vous risquez d'introduire une fuite de mémoire, qui pourrait s'accumuler dans une quantité considérable de mémoire qui n'est pas disponible pour votre programme (ou le système), entraînant éventuellement des pannes ou un comportement indéfini. Les problèmes sont plus susceptibles de se produire si la fuite est occasionnée de manière répétée dans une fonction en boucle ou récursive. Le risque d'échec du programme augmente au fur et à mesure qu'un programme qui fuit est long. Parfois, des problèmes apparaissent instantanément; d'autres fois, les problèmes ne seront pas visibles pendant des heures, voire des années de fonctionnement constant. Les défaillances de mémoire peuvent être catastrophiques, selon les circonstances.

La boucle infinie suivante est un exemple de fuite qui finira par épuiser la fuite de mémoire disponible en appelant `getline()` , une fonction qui alloue implicitement une nouvelle mémoire, sans libérer cette mémoire.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* The loop below leaks memory as fast as it can */
```

```
    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */

        line = NULL;
    }

    return 0;
}
```

En revanche, le code ci-dessous utilise également la fonction `getline()` , mais cette fois, la mémoire allouée est correctement libérée, évitant une fuite.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;

            /* Handle failure such as setting flag, breaking out of loop and/or exiting */
        }

        /* <do whatever> */

        free(line);
        line = NULL;

    }

    return 0;
}
```

Une fuite de mémoire n'a pas toujours des conséquences tangibles et n'est pas nécessairement un problème fonctionnel. Bien que les «meilleures pratiques» dictent la libération rigoureuse de la mémoire aux points et conditions stratégiques, afin de réduire l’empreinte mémoire et les risques d’épuisement de la mémoire, il peut y avoir des exceptions. Par exemple, si un programme est limité en durée et en portée, le risque d'échec de l'allocation peut être considéré comme trop petit pour vous inquiéter. Dans ce cas, contourner la désallocation explicite pourrait être considéré comme acceptable. Par exemple, la plupart des systèmes d'exploitation modernes libèrent automatiquement toute la mémoire consommée par un programme lorsqu'il se termine, qu'il soit dû à une défaillance de programme, à un appel système à `exit()` , à la fin du processus ou à la fin de `main()` . Libérer de manière explicite la mémoire au moment de la fin imminente du programme pourrait en fait être redondant ou introduire une pénalité de performance.

L'allocation peut échouer si la mémoire disponible est insuffisante et que les échecs de traitement doivent être pris en compte aux niveaux appropriés de la pile d'appels. `getline()` , présenté ci-dessus, est un cas d'utilisation intéressant, car il s'agit d'une fonction de bibliothèque qui alloue

non seulement la mémoire qu'il laisse à l'appelant pour la libérer, mais qui peut échouer pour diverses raisons. Par conséquent, il est essentiel, lors de l'utilisation d'une API C, de lire la *documentation (page de manuel)* et d'accorder une attention particulière aux conditions d'erreur et à l'utilisation de la mémoire et de savoir quelle couche logicielle supporte la libération de la mémoire renvoyée.

Une autre méthode courante de gestion de la mémoire consiste à définir systématiquement les pointeurs de mémoire sur NULL immédiatement après la libération de la mémoire référencée par ces pointeurs, afin que leur validité soit vérifiée à tout moment (par exemple, NULL / non-NULL). peut entraîner des problèmes graves tels que l'obtention de données inutiles (opération de lecture), la corruption de données (opération d'écriture) et / ou un plantage de programme. Dans la plupart des systèmes d'exploitation modernes, libérer l'emplacement mémoire 0 (`NULL`) est un NOP (par exemple, il est inoffensif), comme l'exige le standard C - en définissant un pointeur sur NULL, il n'y a aucun risque de double libération de mémoire est passé à `free()` . Gardez à l'esprit que la double libération de mémoire peut entraîner des échecs très longs, compliqués et *difficiles à diagnostiquer* .

Copier trop

```
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

Si l'utilisateur entre une chaîne de plus de 7 caractères (- 1 pour la terminaison nulle), la mémoire derrière le tampon `buf` sera écrasé. Cela se traduit par un comportement indéfini. Les pirates informatiques malveillants l’exploitent souvent pour écraser l’adresse de retour et la remplacer par l’adresse du code malveillant du pirate.

Oubliant de copier la valeur de retour de realloc dans un fichier temporaire

Si `realloc` échoue, il renvoie `NULL` . Si vous affectez la valeur du tampon d'origine à la valeur de retour de `realloc` , et s'il renvoie `NULL` , le tampon d'origine (l'ancien pointeur) est perdu, entraînant une *fuite de mémoire* . La solution consiste à copier dans un pointeur temporaire, et si cela est temporaire NULL, **puis** copiez dans le réel tampon.

```
char *buf, *tmp;

buf = malloc(...);
...

/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");

/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Comparer des nombres à virgule flottante

Les types à virgule flottante (`float` , `double` et `long double`) ne peuvent pas représenter avec précision certains nombres car ils ont une précision finie et représentent les valeurs dans un format binaire. Tout comme nous avons des nombres décimaux répétés dans la base 10 pour des fractions telles que 1/3, il y a des fractions qui ne peuvent pas être représentées finement dans le binaire (comme 1/3, mais aussi et surtout 1/10). Ne pas comparer directement les valeurs à virgule flottante; utilisez plutôt un delta.

```
#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h>  // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}
```

Un autre exemple:

```
gcc -O3   -g   -I./inc   -std=c11   -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-
prototypes  -Wold-style-definition       rd11.c -o rd11 -L./lib -lsoq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
```

```
        epsilon /= 10.0;
    }
    return 0;
}
```

Sortie:

```
0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Faire une mise à l'échelle supplémentaire dans l'arithmétique du pointeur

Dans l'arithmétique du pointeur, l'entier à ajouter ou à soustraire au pointeur est interprété non pas comme un changement d' *adresse* mais comme un nombre d' *éléments* à déplacer.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}
```

Ce code effectue une mise à l'échelle supplémentaire dans le pointeur de calcul affecté à `ptr2` . Si `sizeof(int)` est 4, ce qui est typique dans les environnements 32 bits modernes, l'expression signifie "8 éléments après le `array[0]` ", qui est hors plage, et appelle *un comportement indéfini* .

Pour avoir `ptr2` point à 2 éléments après le `array[0]` , vous devez simplement ajouter 2.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

L'arithmétique explicite des pointeurs utilisant des opérateurs additifs peut être source de confusion, donc l'utilisation de l'indice de tableau peut être meilleure.

```
#include <stdio.h>
```

```
int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

`E1[E2]` est identique à `(*((E1)+(E2)))` (N1570 6.5.2.1, paragraphe 2), et `&(E1[E2])` est équivalent à `((E1)+(E2))` (N1570 6.5.3.2, note de bas de page 102).

Si l'arithmétique du pointeur est préférée, il est possible de convertir le pointeur pour adresser un type de données différent, ce qui peut permettre l'adressage par octet. Attention cependant: l' endianisme peut devenir un problème, et lancer des types autres que «pointeur sur caractère» conduit à des problèmes d'alias stricts .

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3};  // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array;  // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory.  ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */

    return 0;
}
```

Les macros sont des remplacements de chaînes simples

Les macros sont des remplacements de chaîne simples. (Strictement parlant, ils fonctionnent avec des jetons de prétraitement, pas des chaînes arbitraires).

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Vous pouvez vous attendre à ce que ce code imprime `9` (`3*3`), mais en réalité `5` sera imprimé car la macro sera étendue à `1+2*1+2` .

Vous devez envelopper les arguments et l'expression de macro entière entre parenthèses pour éviter ce problème.

```
#include <stdio.h>
```

```
#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Un autre problème est que les arguments d'une macro ne sont pas garantis pour être évalués une fois; ils peuvent ne pas être évalués du tout ou peuvent être évalués plusieurs fois.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Dans ce code, la macro sera étendue à `((a++) <= (10) ? (a++) : (10))` . Etant donné `a++` (`0`) est inférieur à `10` , `a++` sera évalué deux fois et la valeur de `a` et de ce qui est renvoyé par `MIN` différente de celle à laquelle vous pouvez vous attendre.

Cela peut être évité en utilisant des fonctions, mais notez que les types seront corrigés par la définition de la fonction, alors que les macros peuvent être (trop) flexibles avec les types.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Maintenant, le problème de la double évaluation est résolu, mais cette fonction `min` ne peut pas traiter `double` données `double` sans les tronquer, par exemple.

Les directives de macro peuvent être de deux types:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

Ce qui distingue ces deux types de macros est le caractère qui suit l'identifiant après `#define` : si c'est un *lparen* , c'est une macro de type fonction; sinon, c'est une macro de type objet. Si l’intention est d’écrire une macro de type fonction, il ne doit pas y avoir d’espace blanc entre la fin du nom de la macro et `(` . Cochez cette option pour une explication détaillée.

C99

En C99 ou plus tard, vous pouvez utiliser `static inline int min(int x, int y) { … }` .

C11

En C11, vous pouvez écrire une expression 'type-generic' pour `min` .

```
#include <stdio.h>

#define min(x, y) _Generic((x), \
                        long double: min_ld, \
                        unsigned long long: min_ull, \
                        default: min_i \
                        )(x, y)

#define gen_min(suffix, type) \
    static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

L 'expression générique pourrait être étendue avec davantage de types tels que les invocations de macro `gen_min` écrites en `double` , `float` , `long long` , `unsigned long` , `long` , `unsigned` - et appropriées.

Erreurs de référence non définies lors de la liaison

L'une des erreurs de compilation les plus courantes se produit pendant la phase de liaison. L'erreur ressemble à ceci:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

Alors regardons le code qui a généré cette erreur:

```
int foo(void);

int main(int argc, char **argv)
```

```
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

On voit ici une *déclaration* de foo (`int foo();`) mais pas de *définition* (fonction réelle). Nous avons donc fourni le compilateur avec l'en-tête de fonction, mais aucune fonction de ce type n'a été définie, de sorte que l'étape de compilation passe mais que l'éditeur de liens quitte avec une erreur de `Undefined reference` non définie.
Pour corriger cette erreur dans notre petit programme, il suffirait d'ajouter une *définition* pour foo:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Maintenant, ce code va compiler. Une autre situation se présente où la source de `foo()` trouve dans un fichier source séparé, `foo.c` (et il existe un en-tête `foo.h` pour déclarer `foo()` inclus dans `foo.c` et `undefined_reference.c`). Ensuite, le correctif consiste à lier à la fois le fichier objet de `foo.c` et `undefined_reference.c` , ou de compiler les deux fichiers source:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

Ou:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

Un cas plus complexe est celui où les bibliothèques sont impliquées, comme dans le code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;
```

```
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

Le code est syntaxiquement correct, la déclaration de `pow()` existe depuis `#include <math.h>` , nous essayons donc de compiler et de lier, mais nous obtenons une erreur comme celle-ci:

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

Cela se produit car la *définition* de `pow()` n'a pas été trouvée lors de la phase de liaison. Pour corriger cela, nous devons spécifier que nous voulons `-lm` lien avec la bibliothèque mathématique appelée `libm` en spécifiant l' `-lm` . (Notez qu'il existe des plates-formes telles que macOS où `-lm` n'est pas nécessaire, mais lorsque vous obtenez la référence non définie, la bibliothèque est nécessaire.)

Donc, nous exécutons à nouveau l'étape de compilation, en spécifiant cette fois la bibliothèque (après les fichiers source ou objet):

```
$ gcc no_library_in_link.c -lm -o library_in_link_cmd
$ ./library_in_link_cmd 2 4
2.000000 to the power of 4.000000 = 16.000000
$
```

Et il fonctionne!

Incompréhension du tableau

Un problème courant dans le code qui utilise des tableaux multidimensionnels, des tableaux de pointeurs, etc. est le fait que `Type**` et `Type[M][N]` sont fondamentalement différents:

```
#include <stdio.h>
```

```
void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Exemple de sortie du compilateur:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-
Wincompatible-pointer-types]
         print_strings(strings, 4);
                       ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
     void print_strings(char **strings, size_t n)
```

L'erreur indique que le tableau `s` de la fonction `main` est transmis à la fonction `print_strings` , qui attend un type de pointeur différent de celui reçu. Il comprend également une note exprimant le type attendu par `print_strings` et le type qui lui a été transmis depuis le `main` .

Le problème est dû à quelque chose appelé *désintégration de tableau* . Que se passe-t-il lorsque `s` avec son type `char[4][20]` (tableau de 4 tableaux de 20 caractères) est passé à la fonction est-il transformé en un pointeur vers son premier élément comme si vous aviez écrit `&s[0]` , qui a le type `char (*)[20]` (pointeur sur 1 tableau de 20 caractères). Cela se produit pour tout tableau, y compris un tableau de pointeurs, un tableau de tableaux de tableaux (tableaux 3D) et un tableau de pointeurs vers un tableau. Vous trouverez ci-dessous un tableau illustrant ce qui se produit lorsqu'un tableau se désintègre. Les modifications apportées à la description du type sont mises en évidence pour illustrer ce qui se passe:

Avant la pourriture		Après la décomposition	
`char [20]`	**tableau de (20 caractères)**	`char *`	**pointeur vers (1 caractère)**
`char [4][20]`	**tableau de (4 tableaux** de 20 caractères)	`char (*)[20]`	**pointeur vers (1 tableau** de 20 caractères)
`char *[4]`	**tableau de (4 pointeurs** à 1 caractère)	`char **`	**pointeur vers (1 pointeur** vers 1 caractère)
`char [3][4][20]`	**tableau de (3 tableaux** de 4 tableaux de 20 caractères)	`char (*)[4][20]`	**pointeur vers (1 tableau** de 4 tableaux de 20 caractères)

Avant la pourriture		Après la décomposition	
`char (*[4])[20]`	**tableau de (4 pointeurs** à 1 tableau de 20 caractères)	`char (**)[20]`	**pointeur sur (1 pointeur** sur 1 tableau de 20 caractères)

Si un tableau peut se transformer en un pointeur, on peut dire qu'un pointeur peut être considéré comme un tableau contenant au moins un élément. Une exception à ceci est un pointeur nul, qui ne pointe vers rien et n'est par conséquent pas un tableau.

La désintégration des tableaux ne se produit qu'une fois. Si un tableau est devenu un pointeur, il s’agit maintenant d’un pointeur et non d’un tableau. Même si vous avez un pointeur vers un tableau, rappelez-vous que le pointeur peut être considéré comme un tableau contenant au moins un élément, de sorte que le déclin du tableau s’est déjà produit.

En d'autres termes, un pointeur sur un tableau (`char (*)[20]`) ne deviendra jamais un pointeur sur un pointeur (caractère `char **`). Pour corriger la fonction `print_strings` , faites simplement en sorte qu’elle reçoive le bon type:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

Un problème se pose lorsque vous voulez que la fonction `print_strings` soit générique pour tout tableau de caractères: et s'il y avait 30 caractères au lieu de 20? Ou 50? La réponse est d'ajouter un autre paramètre avant le paramètre array:

```
#include <stdio.h>

/*
 * Note the rearranged parameters and the change in the parameter name
 * from the previous definitions:
 *      n (number of strings)
 *   => scount (string count)
 *
 * Of course, you could also use one of the following highly recommended forms
 * for the `strings` parameter instead:
 *
 *    char strings[scount][ccount]
 *    char strings[][ccount]
 */
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
```

```
}
```

La compilation ne génère aucune erreur et produit le résultat attendu:

```
Example 1
Example 2
Example 3
Example 4
```

Passer des tableaux non adjacents à des fonctions qui attendent des tableaux multidimensionnels "réels"

Lors de l'allocation de tableaux multidimensionnels avec `malloc` , `calloc` et `realloc` , il est courant d'allouer les tableaux internes avec plusieurs appels (même si l'appel n'apparaît qu'une seule fois, il peut être en boucle):

```
/* Could also be `int **` with malloc used to allocate outer array. */
int *array[4];
int i;

/* Allocate 4 arrays of 16 ints. */
for (i = 0; i < 4; i++)
    array[i] = malloc(16 * sizeof(*array[i]));
```

La différence en octets entre le dernier élément de l'un des tableaux internes et le premier élément du tableau interne suivant peut ne pas être égale à 0 comme avec un tableau multidimensionnel "réel" (par exemple, `int array[4][16];`) :

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Compte tenu de la taille de `int` , vous obtenez une différence de 8128 octets (8132-4), ce qui correspond à 2032 éléments de tableau `int` dimensionnés.

Si vous devez utiliser un tableau alloué dynamiquement avec une fonction qui attend un tableau multidimensionnel "réel", vous devez allouer un objet de type `int *` et utiliser l'arithmétique pour effectuer des calculs:

```
void func(int M, int N, int *array);
...

/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */
int *array;
int M = 4, N = 16;
array = calloc(M, N * sizeof(*array));
array[i * N + j] = 1;
func(M, N, array);
```

Si `N` est une macro ou un entier au lieu d'une variable, le code peut simplement utiliser la notation plus naturelle du tableau 2D après avoir alloué un pointeur à un tableau:

```
void func(int M, int N, int *array);
#define N 16
void func_N(int M, int (*array)[N]);
...

int M = 4;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;

/* Cast to `int *` works here because `array` is a single block of M*N ints with no gaps,
   just like `int array2[M * N];` and `int array3[M][N];` would be. */
func(M, N, (int *)array);
func_N(M, array);
```

C99

Si `N` n'est pas une macro ou un entier, le `array` pointera vers un tableau de longueur variable (VLA). Cela peut encore être utilisé avec `func` en coulant dans `int *` et une nouvelle fonction `func_vla` remplacerait `func_N` :

```
void func(int M, int N, int *array);
void func_vla(int M, int N, int array[M][N]);
...

int M = 4, N = 16;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
func(M, N, (int *)array);
func_vla(M, N, array);
```

C11

Remarque : les VLA sont facultatifs à partir de C11. Si votre implémentation prend en charge C11 et définit la macro `__STDC_NO_VLA__` à 1, vous êtes bloqué avec les méthodes pré-C99.

Utiliser des constantes de caractères au lieu de littéraux de chaîne et vice versa

En C, les constantes de caractères et les littéraux de chaîne sont des choses différentes.

Un caractère entouré de guillemets simples comme `'a'` est une *constante de caractère* . Une constante de caractère est un entier dont la valeur est le code de caractère correspondant au caractère. Comment interpréter les constantes de caractères avec plusieurs caractères comme `'abc'` est défini par l'implémentation.

Zéro ou plusieurs caractères entourés de guillemets comme `"abc"` est un *littéral de chaîne* . Un littéral de chaîne est un tableau non modifiable dont les éléments sont de type `char` . La chaîne dans les guillemets plus le caractère nul de terminaison sont le contenu, donc `"abc"` a 4 éléments (`{'a', 'b', 'c', '\0'}`)

Dans cet exemple, une constante de caractère est utilisée lorsqu'un littéral de chaîne doit être

utilisé. Cette constante de caractère sera convertie en un pointeur de manière définie par l'implémentation et le pointeur converti a peu de chance d'être valide. Cet exemple invoquera donc *un comportement indéfini* .

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

Dans cet exemple, un littéral de chaîne est utilisé où une constante de caractère doit être utilisée. Le pointeur converti à partir du littéral de chaîne sera converti en un entier d'une manière définie par l'implémentation, et il sera converti en `char` d'une manière définie par l'implémentation. (Comment convertir un entier en un type signé qui ne peut pas représenter la valeur à convertir est défini par l'implémentation, et si `char` est signé est également défini par l'implémentation.) La sortie sera sans objet.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

Dans presque tous les cas, le compilateur se plaindra de ces mélanges. Si ce n'est pas le cas, vous devez utiliser davantage d'options d'avertissement du compilateur, ou il est recommandé d'utiliser un meilleur compilateur.

Ignorer les valeurs de retour des fonctions de bibliothèque

Presque toutes les fonctions de la bibliothèque standard C renvoient quelque chose en cas de succès, et une autre erreur. Par exemple, `malloc` renverra un pointeur sur le bloc de mémoire alloué par la fonction en cas de succès et, si la fonction ne parvient pas à allouer le bloc de mémoire demandé, un pointeur nul. Donc, vous devriez toujours vérifier la valeur de retour pour un débogage plus facile.

C'est mauvais:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation
violation, unless your system has a lot of memory */
```

C'est bon:

```
#include <stdlib.h>
#include <stdio.h>
```

```
int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

De cette façon, vous savez immédiatement la cause de l'erreur, sinon vous risquez de passer des heures à chercher un bug dans un endroit complètement erroné.

Le caractère de nouvelle ligne n'est pas utilisé lors d'un appel scanf () classique

Quand ce programme

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

est exécuté avec cette entrée

```
42
life
```

la sortie sera `42 ""` au lieu de `42 "life"` attendue.

C'est parce qu'un caractère de nouvelle ligne après `42` n'est pas consommé dans l'appel de `scanf()` et qu'il est consommé par `fgets()` avant de lire la `life` . Ensuite, `fgets()` arrête de lire avant de lire la `life` .

Pour éviter ce problème, une méthode utile lorsque la longueur maximale d'une ligne est connue - lors de la résolution de problèmes dans le système de juge en ligne, par exemple - évite d'utiliser directement `scanf()` et de lire toutes les lignes via `fgets()` . Vous pouvez utiliser `sscanf()` pour analyser les lignes lues.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Une autre méthode consiste à lire jusqu'à ce que vous frappiez un caractère de nouvelle ligne après avoir utilisé `scanf()` et avant d'utiliser `fgets()` .

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Ajouter un point-virgule à un #define

Il est facile de se perdre dans le préprocesseur C et de le traiter comme partie intégrante de C, mais c'est une erreur car le préprocesseur n'est qu'un mécanisme de substitution de texte. Par exemple, si vous écrivez

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

le code se développe à

```
int arr[100;];
```

qui est une erreur de syntaxe. La solution consiste à supprimer le point-virgule de la ligne `#define` . Il est presque toujours une erreur de mettre fin à un `#define` avec un point-virgule.

Les commentaires sur plusieurs lignes ne peuvent pas être imbriqués

En C, les commentaires sur plusieurs lignes, / * et * /, ne pas imbriquer.

Si vous annotez un bloc de code ou une fonction en utilisant ce style de commentaire:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Vous ne pourrez pas le commenter facilement:

```
//Trying to comment out the block...
/*

/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

//Causes an error on the line below...
*/
```

Une solution consiste à utiliser les commentaires de style C99:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
```

```
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Maintenant, le bloc entier peut être facilement commenté:

```
/*

// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Une autre solution consiste à éviter de désactiver le code en utilisant la syntaxe de commentaire, en utilisant `#ifdef` directives de préprocesseur `#ifdef` ou `#ifndef` . Ces directives *sont* imbriquées, vous laissant libre de commenter votre code dans le style que vous préférez.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifdef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
#endif
```

Certains guides vont jusqu'à recommander que les sections de code *ne* soient *jamais* commentées et que, si le code doit être désactivé temporairement, on puisse avoir recours à une directive `#if 0` .

Voir [#if 0 pour bloquer les sections de code](#) .

Dépasser les limites du réseau

Les tableaux sont basés sur zéro, c'est-à-dire que l'index commence toujours à 0 et se termine par la longueur du tableau d'index moins 1. Ainsi, le code suivant ne générera pas le premier élément du tableau et affichera la valeur finale qu'il imprime.

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 1; x <= 5; x++) //Looping from 1 till 5.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Sortie: `2 3 4 5 GarbageValue`

Ce qui suit illustre la manière correcte d'obtenir la sortie souhaitée:

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 0; x < 5; x++) //Looping from 0 till 4.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Sortie: `1 2 3 4 5`

Il est important de connaître la longueur d'un tableau avant de l'utiliser, sinon vous risquez de corrompre le tampon ou de provoquer une erreur de segmentation en accédant à des emplacements de mémoire hors limites.

Fonction récursive - manquant la condition de base

Calculer la factorielle d'un nombre est un exemple classique de fonction récursive.

Manquant la condition de base:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}
```

```
int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Sortie typique: `Segmentation fault: 11`

Le problème avec cette fonction est qu'elle ferait une boucle infinie, provoquant une erreur de segmentation - elle nécessite une condition de base pour arrêter la récursivité.

Condition de base déclarée:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
       return 1;
    }
    else
    {
       return n * factorial(n - 1);
    }
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Sortie de l'échantillon

```
Factorial 3 = 6
```

Cette fonction se terminera dès qu'elle atteindra la condition `n` est égale à 1 (à condition que la valeur initiale de `n` soit suffisamment petite - la limite supérieure est `12` lorsque `int` est une quantité de 32 bits).

Règles à suivre:

1. Initialiser l'algorithme Les programmes récursifs nécessitent souvent une valeur de départ pour commencer. Ceci est accompli soit en utilisant un paramètre passé à la fonction, soit en fournissant une fonction de passerelle non récursive mais qui définit les valeurs de départ pour le calcul récursif.
2. Vérifiez si la ou les valeurs en cours de traitement correspondent au cas de base. Si oui, traitez et renvoyez la valeur.
3. Redéfinissez la réponse en termes de sous-problèmes ou de sous-problèmes plus petits ou plus simples.
4. Exécutez l'algorithme sur le sous-problème.

5. Combinez les résultats dans la formulation de la réponse.
6. Renvoyer les résultats.

Source: Fonction récursive

Vérification de l'expression logique contre 'true'

Le standard C original n'avait pas de type booléen intrinsèque, donc `bool` , `true` et `false` n'avaient aucune signification inhérente et étaient souvent définis par les programmeurs. Typiquement, `true` serait défini comme 1 et `false` serait défini sur 0.

C99

C99 ajoute le type `_Bool` et l'en-tête `<stdbool.h>` qui définit `bool` (en expansion à `_Bool`), `false` et `true` . Cela vous permet également de redéfinir `bool` , `true` et `false` , mais note qu’il s’agit d’une fonctionnalité obsolète.

Plus important encore, les expressions logiques traitent tout ce qui est considéré comme nul comme faux et toute évaluation non nulle comme vraie. Par exemple:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == true)  /* Comparison only succeeds if true is 0x80 and bitField
has that bit set */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Dans l'exemple ci-dessus, la fonction tente de vérifier si le bit supérieur est défini et renvoie `true` si c'est le cas. Cependant, en vérifiant explicitement la valeur `true` , l'instruction `if` réussira uniquement si `(bitfield & 0x80)` évalué à `true` , ce qui est généralement `1` et très rarement `0x80` . Soit explicitement vérifier avec le cas que vous attendez:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Ou évaluez toute valeur non nulle comme vraie.

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Les littéraux à virgule flottante sont de type double par défaut

Des précautions doivent être prises lors de l'initialisation des variables de type `float` à des valeurs littérales ou en les comparant avec des valeurs littérales, car les littéraux à virgule flottante tels que `0.1` sont de type `double` . Cela peut entraîner des surprises:

```
#include <stdio.h>
int main() {
    float  n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Ici, `n` est initialisé et arrondi à une précision unique, avec pour résultat la valeur 0.10000000149011612. Ensuite, `n` est reconverti en double précision pour être comparé à `0.1` littéral (ce qui équivaut à 0,10000000000000001), ce qui entraîne une non-concordance.

Outre les erreurs d'arrondi, le mélange de variables `float` avec `double` littéraux `double` entraînera de mauvaises performances sur les plates-formes qui ne prennent pas en charge le matériel pour une double précision.

Lire Pièges communs en ligne: https://riptutorial.com/fr/c/topic/2006/pieges-communs

Chapitre 44: Pointeurs

Introduction

Un pointeur est un type de variable pouvant stocker l'adresse d'un autre objet ou d'une fonction.

Syntaxe

- <Type de données> * <Nom de la variable>;
- int * ptrToInt;
- void * ptrToVoid; / * C89 + * /
- struct someStruct * ptrToStruct;
- int ** ptrToPtrToInt;
- int arr [longueur]; int * ptrToFirstElem = arr; / * Pour <C99 'length doit être une constante de compilation, pour> = C11 il faudra peut-être en avoir une. * /
- int * arrayOfPtrsToInt [length]; / * Pour <C99 'length doit être une constante de compilation, pour> = C11 il faudra peut-être en avoir une. * /

Remarques

La position de l'astérisque n'affecte pas le sens de la définition:

```
/* The * operator binds to right and therefore these are all equivalent. */
int *i;
int * i;
int* i;
```

Cependant, lors de la définition de plusieurs pointeurs à la fois, chacun nécessite son propre astérisque:

```
int *i, *j; /* i and j are both pointers */
int* i, j;  /* i is a pointer, but j is an int not a pointer variable */
```

Un tableau de pointeurs est également possible, où un astérisque est donné avant le nom de la variable du tableau:

```
int *foo[2]; /* foo is a array of pointers, can be accessed as *foo[0] and *foo[1] */
```

Examples

Erreurs courantes

Une mauvaise utilisation des pointeurs est souvent une source de bogues pouvant inclure des bogues de sécurité ou des plantages de programmes, le plus souvent dus à des erreurs de

segmentation.

Ne pas vérifier les échecs d'allocation

L'allocation de mémoire n'est pas garantie pour réussir et peut à la place renvoyer un pointeur `NULL` . L'utilisation de la valeur renvoyée, sans vérifier si l'allocation a réussi, appelle un comportement non défini . Cela conduit généralement à un plantage, mais rien ne garantit qu'un crash se produira, ce qui peut également entraîner des problèmes.

Par exemple, manière dangereuse:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Manière sûre:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Utiliser des nombres littéraux au lieu de sizeof lors de la demande de mémoire

Pour une configuration compilateur / machine donnée, les types ont une taille connue. Cependant, il n'existe pas de norme définissant que la taille d'un type donné (autre que `char`) sera la même pour toutes les configurations du compilateur / machine. Si le code utilise 4 au lieu de `sizeof(int)` pour l'allocation de mémoire, cela peut fonctionner sur la machine d'origine, mais le code n'est pas nécessairement portable à d'autres machines ou compilateurs. Les tailles fixes pour les types doivent être remplacées par `sizeof(that_type)` ou `sizeof(*var_ptr_to_that_type)` .

Allocation non portable:

```
 int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
 long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Allocation portable:

```
 int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
 long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

Ou mieux encore:

```
 int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
 long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Fuites de mémoire

Si vous ne désallouez pas la mémoire à l'aide de `free` vous obtenez une accumulation de mémoire non réutilisable, qui n'est plus utilisée par le programme; cela s'appelle une fuite de mémoire . Les fuites de mémoire gaspillent des ressources de mémoire et peuvent entraîner des défaillances d'allocation.

Erreurs logiques

Toutes les allocations doivent suivre le même schéma:

1. Allocation en utilisant `malloc` (ou `calloc`)
2. Utilisation pour stocker des données
3. Désaffectation en utilisant `free`

Ne pas adhérer à ce modèle, comme utiliser la mémoire après un appel à `free` (pointeur) ou avant un appel à `malloc` (pointeur sauvage), appeler deux fois `free` ("double free"), etc., provoque généralement une erreur de segmentation et entraîne un crash du programme.

Ces erreurs peuvent être transitoires et difficiles à déboguer - par exemple, la mémoire libérée n'est généralement pas récupérée immédiatement par le système d'exploitation et les pointeurs en suspens peuvent donc persister pendant un certain temps et sembler fonctionner.

Sur les systèmes sur lesquels il fonctionne, Valgrind est un outil précieux pour identifier les fuites de mémoire et leur emplacement initial.

Créer des pointeurs pour empiler des variables

Créer un pointeur ne prolonge pas la vie de la variable pointée. Par exemple:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Ici, `x` a *une durée de stockage automatique* (communément appelée allocation de *pile*). Comme il est alloué sur la pile, sa durée de vie est aussi longue que `myFunction` s'exécute; une fois `myFunction` sortie, la variable `x` est détruite. Cette fonction obtient l'adresse de `x` (en utilisant `&x`), et la renvoie à l'appelant, laissant l'appelant avec un pointeur sur une variable inexistante. Tenter d'accéder à cette variable invoquera alors un comportement indéfini .

La plupart des compilateurs ne nettoient pas réellement un cadre de pile après la fermeture de la fonction, ce qui permet de déréférencer le pointeur retourné et vous donne souvent les données attendues. Lorsqu'une autre fonction est appelée, la mémoire pointée peut être écrasée et il semble que les données pointées ont été corrompues.

Pour résoudre ce problème, `malloc` le stockage de la variable à renvoyer et renvoyez un pointeur vers le stockage nouvellement créé, ou exigez qu'un pointeur valide soit transmis à la fonction au lieu d'en renvoyer un, par exemple:

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    {
        /* Use solution1() */

        int *foo = solution1();
        if (foo == NULL)
        {
            /* Something went wrong */
            return 1;
        }

        printf("The value set by solution1() is %i\n", *foo);
        /* Will output: "The value set by solution1() is 10" */

        free(foo);    /* Tidy up */
    }

    {
        /* Use solution2() */

        int bar;
        solution2(&bar);

        printf("The value set by solution2() is %i\n", bar);
        /* Will output: "The value set by solution2() is 10" */
    }

    return 0;
}
```

Incrémenter / décrémenter et déréférencer

Si vous écrivez `*p++` pour incrémenter ce qui est indiqué par `p` , vous avez tort.

La post-incrémentation / décrémentation est exécutée avant le déréférencement. Par conséquent, cette expression incrémentera le pointeur `p` lui-même et renverra ce qui a été pointé par `p` avant de l'incrémenter sans le modifier.

Vous devriez écrire `(*p)++` pour incrémenter ce qui est indiqué par `p` .

Cette règle s'applique également à la post-décrémentation: `*p--` décrémentera le pointeur `p` lui-même, pas ce qui est indiqué par `p` .

Déréférencer un pointeur

```
int a = 1;
int *a_pointer = &a;
```

Pour déréférencer `a_pointer` et changer la valeur de a, nous utilisons l'opération suivante

```
*a_pointer = 2;
```

Cela peut être vérifié en utilisant les instructions d'impression suivantes.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

Cependant, on pourrait se tromper de déréférencer un pointeur `NULL` ou non valide. Ce

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

est généralement un comportement indéfini . `p1` ne peut pas être déréférencé car il pointe vers une adresse `0xbad` qui peut ne pas être une adresse valide. Qui sait ce qu'il y a? Il peut s'agir de la mémoire du système d'exploitation ou de la mémoire d'un autre programme. Le seul code temporel comme celui-ci est le développement intégré, qui stocke des informations particulières à des adresses codées en dur. `p2` ne peut pas être déréférencé car il est `NULL` , ce qui est invalide.

Déréférencer un pointeur à une structure

Disons que nous avons la structure suivante:

```
struct MY_STRUCT
```

```
{
    int my_int;
    float my_float;
};
```

Nous pouvons définir `MY_STRUCT` pour omettre le mot `struct` clé `struct` , nous n'avons donc pas à taper `struct MY_STRUCT` chaque fois que nous l'utilisons. Ceci est cependant facultatif.

```
typedef struct MY_STRUCT MY_STRUCT;
```

Si nous avons alors un pointeur sur une instance de cette structure

```
MY_STRUCT *instance;
```

Si cette instruction apparaît à la portée du fichier, l' `instance` sera initialisée avec un pointeur nul au démarrage du programme. Si cette instruction apparaît dans une fonction, sa valeur est indéfinie. La variable doit être initialisée pour pointer sur une variable `MY_STRUCT` valide ou sur un espace alloué dynamiquement avant de pouvoir être déréférencé. Par exemple:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

Lorsque le pointeur est valide, nous pouvons le déréférencer pour accéder à ses membres en utilisant l'une des deux notations suivantes:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

Bien que ces deux méthodes fonctionnent, il est préférable d'utiliser l'opérateur arrow `->` plutôt que la combinaison de parenthèses, l'opérateur dereference `*` et le point `.` opérateur car il est plus facile à lire et à comprendre, en particulier avec les utilisations imbriquées.

Une autre différence importante est illustrée ci-dessous:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

Dans ce cas, `copy` contient une copie du contenu de l' `instance` . Changer `my_int` de `copy` ne le changera pas par `instance` .

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

Dans ce cas, `ref` est une référence à une `instance` . Changer `my_int` utilisant la référence le changera par `instance` .

Il est courant d'utiliser des pointeurs vers des structures en tant que paramètres dans les fonctions, plutôt que les structures elles-mêmes. L'utilisation des structures en tant que

paramètres de fonction pourrait entraîner un débordement de la pile si la structure est volumineuse. L'utilisation d'un pointeur sur une structure n'utilise que suffisamment d'espace de pile pour le pointeur, mais peut entraîner des effets secondaires si la fonction modifie la structure transmise à la fonction.

Pointeurs de fonction

Les pointeurs peuvent également être utilisés pour pointer des fonctions.

Prenons une fonction de base:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Maintenant, définissons un pointeur du type de cette fonction:

```
int (*my_pointer)(int, int);
```

Pour en créer un, utilisez simplement ce modèle:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

Nous devons ensuite affecter ce pointeur à la fonction:

```
my_pointer = &my_function;
```

Ce pointeur peut maintenant être utilisé pour appeler la fonction:

```
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Bien que cette syntaxe semble plus naturelle et cohérente avec les types de base, les pointeurs de fonction d'attribution et de déréférencement ne nécessitent pas l'utilisation des opérateurs `&` et `*` . Ainsi, l'extrait suivant est également valide:

```
/* Attribution without the & operator */
my_pointer = my_function;
```

```
/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

Pour augmenter la lisibilité des pointeurs de fonction, les typedefs peuvent être utilisés.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Une autre astuce de lisibilité est que le standard C permet de simplifier un pointeur de fonction dans les arguments comme ci-dessus (mais pas dans la déclaration de variable) à quelque chose qui ressemble à un prototype de fonction; ainsi, les éléments suivants peuvent être utilisés de manière équivalente pour les définitions de fonctions et les déclarations:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

Voir également

Pointeurs de fonction

Initialisation des pointeurs

L'initialisation du pointeur est un bon moyen d'éviter les pointeurs sauvages. L'initialisation est simple et n'est pas différente de l'initialisation d'une variable.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

         /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */

    ...
}
```

Dans la plupart des systèmes d'exploitation, l'utilisation involontaire d'un pointeur initialisé à `NULL` entraîne souvent une panne immédiate du programme, facilitant ainsi l'identification de la cause du problème. L'utilisation d'un pointeur non initialisé peut souvent causer des bogues difficiles à diagnostiquer.

Mise en garde:

Le résultat du déréférencement d'un pointeur `NULL` n'est pas défini, il *ne provoquera* donc *pas nécessairement un blocage* même si cela est le comportement naturel du système d'exploitation sur lequel le programme s'exécute. Les optimisations du compilateur peuvent masquer la panne, provoquer la panne avant ou après le point dans le code source auquel le déréférencement du pointeur null s'est produit ou provoquer la suppression inattendue de certaines parties du code contenant le déréférencement du pointeur nul du programme. Les versions de débogage ne présentent généralement pas ces comportements, mais cela n'est pas garanti par la norme de langage. Tout autre comportement inattendu et / ou indésirable est également autorisé.

Comme `NULL` ne pointe jamais sur une variable, sur la mémoire allouée ou sur une fonction, il est possible de l'utiliser comme valeur de garde.

Mise en garde:

Habituellement, `NULL` est défini comme `(void *)0` . Mais cela ne signifie pas que l'adresse mémoire attribuée est `0x0` . Pour plus de précisions, reportez - vous à C-faq pour les pointeurs NULL

Notez que vous pouvez également initialiser des pointeurs pour contenir des valeurs autres que NULL.

```
int i1;

int main()
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Adresse de l'opérateur (&)

Pour tout objet (variable, tableau, union, struct, pointeur ou fonction), l'opérateur d'adresse unaire peut être utilisé pour accéder à l'adresse de cet objet.

Supposer que

```
int i = 1;
int *p = NULL;
```

Alors, une déclaration `p = &i;` , copie l'adresse de la variable `i` dans le pointeur `p` .

Il est exprimé en `p` ***points à*** `i` .

`printf("%d\n", *p);` imprime 1, qui est la valeur de `i` .

Arithmétique du pointeur

S'il vous plaît voir ici: Arithmétique Pointeur

void * pointeurs comme arguments et renvoyer des valeurs aux fonctions standard

K & R

`void*` est un type attrape tous les pointeurs vers les types d'objet. Un exemple de ceci est utilisé avec la fonction `malloc` , qui est déclarée comme

```
void* malloc(size_t);
```

Le type de retour d'un pointeur à un autre signifie qu'il est possible d'affecter la valeur de retour de `malloc` à un pointeur vers tout autre type d'objet:

```
int* vector = malloc(10 * sizeof *vector);
```

Il est généralement considéré comme une bonne pratique de *ne pas* intégrer explicitement les valeurs dans les pointeurs de vide. Dans le cas spécifique de `malloc()` c'est parce qu'avec un transtypage explicite, le compilateur peut par ailleurs supposer, mais sans avertir, un type de retour incorrect pour `malloc()` , si vous oubliez d'inclure `stdlib.h` . Il s'agit également d'utiliser le comportement correct des pointeurs de vide pour mieux se conformer au principe DRY (ne vous répétez pas); comparez ce qui suit à ce qui suit, dans lequel le code suivant contient plusieurs endroits supplémentaires inutiles où une faute de frappe pourrait causer des problèmes:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

De même, des fonctions telles que

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

ont leurs arguments spécifiés comme `void *` car l'adresse de n'importe quel objet, quel que soit son type, peut être transmise. Ici aussi, un appel ne doit pas utiliser de transtypage

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Constants

Pointeurs uniques

- Pointeur vers un `int`

  Le pointeur peut pointer vers différents entiers et les `int` peuvent être modifiés via le pointeur. Cet exemple de code assigne b à pointer sur `int b` puis modifie la valeur de `b` à `100`

.

```
int b;
int* p;
p = &b;    /* OK */
*p = 100;  /* OK */
```

- Pointeur sur un `const int`

  Le pointeur peut pointer vers différents entiers mais la valeur de l' `int` ne peut pas être modifiée via le pointeur.

```
int b;
const int* p;
p = &b;    /* OK */
*p = 100;  /* Compiler Error */
```

- `const` pointeur sur `int`

  Le pointeur ne peut pointer que sur un `int` mais la valeur de l' `int` peut être modifiée via le pointeur.

```
int a, b;
int* const p = &b; /* OK as initialisation, no assignment */
*p = 100;  /* OK */
p = &a;    /* Compiler Error */
```

- `const` pointeur sur `const int`

  Le pointeur ne peut pointer que vers un `int` et l' `int` ne peut pas être modifié par le pointeur.

```
int a, b;
const int* const p = &b; /* OK as initialisation, no assignment */
p = &a;   /* Compiler Error */
*p = 100; /* Compiler Error */
```

Pointeur vers le pointeur

- Pointeur sur un pointeur vers un `int`

  Ce code assigne l'adresse de `p1` au double pointeur `p` (qui pointe alors sur `int* p1` (qui pointe sur `int`)).

  Puis change de `p1` pour pointer sur `int a` . Puis change la valeur de a pour être 100.

```
void f1(void)
{
  int a, b;
  int *p1;
  int **p;
  p1 = &b; /* OK */
```

```
  p = &p1; /* OK */
  *p = &a; /* OK */
  **p = 100; /* OK */
}
```

- Pointeur vers un pointeur vers un `const int`

```
 void f2(void)
{
  int b;
  const int *p1;
  const int **p;
  p = &p1; /* OK */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- Pointeur sur `const` pointeur sur un `int`

```
void f3(void)
{
  int b;
  int *p1;
  int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* OK */
}
```

- `const` pointeur vers pointeur vers `int`

```
void f4(void)
{
  int b;
  int *p1;
  int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’ */
  *p = &b; /* OK */
  **p = 100; /* OK */
}
```

- Pointeur sur `const` pointeur sur `const int`

```
void f5(void)
{
  int b;
  const int *p1;
  const int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- `const` pointeur vers pointeur vers `const int`

```
void f6(void)
{
  int b;
  const int *p1;
  const int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’ */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- `const` pointeur vers `const` pointeur vers `int`

```
void f7(void)
{
  int b;
  int *p1;
  int * const * const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’  */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* OK */
}
```

Même astérisque, significations différentes

Prémisse

L'aspect le plus déroutant entourant la syntaxe de pointeur en C et C ++ est qu'il existe en réalité deux significations différentes qui s'appliquent lorsque le symbole du pointeur, l'astérisque (`*`), est utilisé avec une variable.

Exemple

Tout d'abord, vous utilisez `*` pour **déclarer** une variable de pointeur.

```
int i = 5;
/* 'p' is a pointer to an integer, initialized as NULL */
int *p = NULL;
/* '&i' evaluates into address of 'i', which then assigned to 'p' */
p = &i;
/* 'p' is now holding the address of 'i' */
```

Lorsque vous ne déclarez pas (ou multipliez), `*` est utilisé pour **déréférencer** une variable de pointeur:

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

Lorsque vous souhaitez qu'une variable de pointeur existante contienne l'adresse d'une autre variable, vous **n'utilisez pas** `*` , mais faites-le comme ceci:

```
p = &another_variable;
```

Une confusion commune entre les débutants de la programmation en C survient lorsqu'ils déclarent et initialisent une variable de pointeur en même temps.

```
int *p = &i;
```

Puisque `int i = 5;` et `int i; i = 5;` donner le même résultat, certains d'entre eux pourraient penser `int *p = &i;` et `int *p; *p = &i;` donner le même résultat aussi. Le fait est que non, `int *p; *p = &i;` tentera de déférer un pointeur **non initialisé** qui entraînera UB. N'utilisez jamais `*` lorsque vous ne déclarez ni ne déréférence un pointeur.

Conclusion

L'astérisque (`*`) a deux significations distinctes dans C par rapport aux pointeurs, selon l'endroit où il est utilisé. Lorsqu'elle est utilisée dans une **déclaration de variable** , la valeur du côté droit du côté égal doit être une ***valeur de pointeur*** vers une ***adresse*** en mémoire. Lorsqu'il est utilisé avec une **variable** déjà **déclarée** , l'astérisque ***déréférencera*** la valeur du pointeur, le suivant à l'emplacement pointé dans la mémoire, et permettant d'attribuer ou de récupérer la valeur stockée à cet endroit.

À emporter

Il est important de se soucier de vos P et Q, pour ainsi dire, quand il s'agit de pointeurs. Soyez conscient du moment où vous utilisez l'astérisque et de ce que cela signifie lorsque vous l'utilisez là. Si vous négligez ce petit détail, vous risquez d’avoir un comportement bogué et / ou indéfini auquel vous ne souhaitez pas avoir à faire face.

Pointeur vers le pointeur

En C, un pointeur peut faire référence à un autre pointeur.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */

  return EXIT_SUCCESS;
}
```

Mais, référence et référence directement n'est pas autorisé.

```
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
  int*** pppA = &&&A;  /* Compilation error here! */

  ...
```

introduction

Un pointeur est déclaré comme toute autre variable, sauf qu'un astérisque (`*`) est placé entre le type et le nom de la variable pour indiquer qu'il s'agit d'un pointeur.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid
object yet */
```

Pour déclarer deux variables de pointeur du même type, dans la même déclaration, utilisez le symbole astérisque avant chaque identifiant. Par exemple,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int
*/
```

L'opérateur d'adresse ou de référence désigné par une esperluette (`&`) donne l'adresse d'une variable donnée qui peut être placée dans un pointeur de type approprié.

```
int value = 1;
pointer = &value;
```

L'opérateur d'indirection ou de déréférencement désigné par un astérisque (`*`) récupère le contenu d'un objet désigné par un pointeur.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

Si le pointeur pointe sur une structure ou un type d'union, vous pouvez le déréférencer et accéder directement à ses membres en utilisant l'opérateur `->` :

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

En C, un pointeur est un type de valeur distinct qui peut être réaffecté, sinon il est traité comme une variable à part entière. Par exemple, l'exemple suivant imprime la valeur du pointeur (variable) lui-même.

```
printf("Value of the pointer itself: %p\n", (void *)pointer);
/* Value of the pointer itself: 0x7ffcd41b06e4 */
/* This address will be different each time the program is executed */
```

Comme un pointeur est une variable mutable, il est possible qu'il ne pointe pas vers un objet valide, soit en définissant la valeur null

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

ou simplement en contenant un modèle de bit arbitraire qui n'est pas une adresse valide. La dernière est une très mauvaise situation, car elle ne peut pas être testée avant que le pointeur ne soit déréférencé, il n'y a qu'un test pour le cas où un pointeur est nul:

```
if (!pointer) exit(EXIT_FAILURE);
```

Un pointeur ne peut être déréférencé que s'il pointe vers un objet *valide* , sinon le comportement n'est pas défini. De nombreuses implémentations modernes peuvent vous aider en soulevant un type d'erreur tel qu'une erreur de segmentation et en interrompant l'exécution, mais d'autres peuvent simplement laisser votre programme dans un état invalide.

La valeur renvoyée par l'opérateur de déréférence est un alias mutable à la variable d'origine, de sorte qu'il peut être modifié en modifiant la variable d'origine.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Les pointeurs sont également ré-assignables. Cela signifie qu'un pointeur pointant sur un objet peut être utilisé ultérieurement pour pointer vers un autre objet du même type.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Comme toute autre variable, les pointeurs ont un type spécifique. Par exemple, vous ne pouvez pas affecter l'adresse d'un `short int` à un pointeur `long int` . Un tel comportement est appelé punition de type et est interdit dans C, bien qu'il y ait quelques exceptions.

Bien que le pointeur doive être d'un type spécifique, la mémoire allouée pour chaque type de pointeur est égale à la mémoire utilisée par l'environnement pour stocker les adresses, plutôt que la taille du type pointé.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof (int*));      /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof (int));      /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof (char*));    /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof (char));    /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof (short*));  /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof (short));  /* size 2 bytes */
    return 0;
}
```

(Remarque: si vous utilisez Microsoft Visual Studio, qui ne prend pas en charge les normes C99 ou C11, vous devez utiliser `%Iu` [1] au lieu de `%zu` dans l'exemple ci-dessus.)

Notez que les résultats ci-dessus peuvent varier d'un environnement à l'autre en termes de nombre, mais que tous les environnements affichent des tailles égales pour différents types de pointeurs.

Extrait basé sur des informations de l'Université de Cardiff C Pointers Introduction

Pointeurs et tableaux

Les pointeurs et les tableaux sont intimement liés en C. Les tableaux en C sont toujours situés dans des emplacements contigus en mémoire. L'arithmétique du pointeur est toujours mise à l'échelle en fonction de la taille de l'élément pointé. Donc, si nous avons un tableau de trois doubles et un pointeur sur la base, `*ptr` réfère au premier double, `*(ptr + 1)` au second, `*(ptr + 2)` au troisième. Une notation plus pratique consiste à utiliser la notation de tableau `[]` .

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;

/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

Donc, essentiellement ptr et le nom du tableau sont interchangeables. Cette règle signifie également qu'un tableau se désintègre en un pointeur lorsqu'il est transmis à un sous-programme.

```
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

Un pointeur peut pointer sur un élément d'un tableau ou sur l'élément situé au-delà du dernier élément. C'est cependant une erreur de définir un pointeur sur une autre valeur, y compris l'élément avant le tableau. (La raison en est que sur les architectures segmentées, l'adresse avant le premier élément peut traverser une limite de segment, le compilateur garantit que cela ne se produit pas pour le dernier élément plus un).

Note de bas de page 1: Les informations de format Microsoft peuvent être trouvées via `printf()` et la syntaxe de spécification de format .

Comportement polymorphe avec des pointeurs de vide

La fonction de bibliothèque standard `qsort()` est un bon exemple de la façon dont on peut utiliser

des pointeurs vides pour faire fonctionner une seule fonction sur une grande variété de types différents.

```
void qsort (
    void *base,                                  /* Array to be sorted */
    size_t num,                                  /* Number of elements in array */
    size_t size,                                 /* Size in bytes of each element */
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

Le tableau à trier est transmis en tant que pointeur vide, de sorte qu'un tableau de tout type d'élément peut être utilisé. Les deux arguments suivants indiquent à `qsort()` combien d’éléments il doit attendre dans le tableau et quelle est la taille, en octets, de chaque élément.

Le dernier argument est un pointeur de fonction vers une fonction de comparaison qui prend elle-même deux pointeurs de vide. En faisant en sorte que l'appelant fournisse cette fonction, `qsort()` peut trier efficacement les éléments de tout type.

Voici un exemple d'une telle fonction de comparaison, pour comparer des flotteurs. Notez que toute fonction de comparaison passée à `qsort()` doit avoir cette signature de type. La manière dont il est rendu polymorphe consiste à convertir les arguments du pointeur vide en pointeurs du type d’élément que nous souhaitons comparer.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

Puisque nous savons que qsort utilisera cette fonction pour comparer les flottants, nous renvoyons les arguments du pointeur vide à des pointeurs flottants avant de les déréférencer.

Maintenant, l'utilisation de la fonction polymorphe qsort sur un tableau "array" de longueur "len" est très simple:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Lire Pointeurs en ligne: https://riptutorial.com/fr/c/topic/1108/pointeurs

Chapitre 45: Pointeurs de fonction

Introduction

Les pointeurs de fonction sont des pointeurs qui pointent vers des fonctions plutôt que des types de données. Ils peuvent être utilisés pour autoriser la variabilité de la fonction à appeler lors de l'exécution.

Syntaxe

- returnType (* nom) (paramètres)
- typedef returnType (* name) (paramètres)
- typedef returnType Name (paramètres);
  Nom nom;
- typedef returnType Name (paramètres);
  typedef Nom * NomPtr;

Examples

Assigner un pointeur de fonction

```
#include <stdio.h>

/* increment: take number, increment it by one, and return it */
int increment(int i)
{
    printf("increment %d by 1\n", i);
    return i + 1;
}

/* decrement: take number, decrement it by one, and return it */
int decrement(int i)
{
    printf("decrement %d by 1\n", i);
    return i - 1;
}

int main(void)
{
    int num = 0;          /* declare number to increment */
    int (*fp)(int);       /* declare a function pointer */

    fp = &increment;      /* set function pointer to increment function */
    num = (*fp)(num);     /* increment num */
    num = (*fp)(num);     /* increment num a second time */

    fp = &decrement;      /* set function pointer to decrement function */
    num = (*fp)(num);     /* decrement num */
```

```
    printf("num is now: %d\n", num);
    return 0;
}
```

Renvoi de pointeurs de fonction à partir d'une fonction

```
#include <stdio.h>

enum Op
{
  ADD = '+',
  SUB = '-',
};


/* add: add a and b, return result */
int add(int a, int b)
{
    return a + b;
}

/* sub: subtract b from a, return result */
int sub(int a, int b)
{
    return a - b;
}

/* getmath: return the appropriate math function */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Les meilleures pratiques

Utiliser typedef

Il peut être utile d'utiliser un `typedef` au lieu de déclarer le pointeur de fonction à la main.

La syntaxe pour déclarer un `typedef` pour un pointeur de fonction est la suivante:

```
typedef returnType (*name)(parameters);
```

Exemple:

Posit que nous avons une fonction, `sort` , qui attend un pointeur de fonction sur une fonction `compare` telle que:

> compare - Une fonction de comparaison pour deux éléments à fournir à une fonction de tri.
>
> "compare" est censé renvoyer 0 si les deux éléments sont considérés égaux, une valeur positive si le premier élément passé est "plus grand" dans un certain sens que le dernier élément et sinon la fonction retourne une valeur négative (ce qui signifie que le premier élément est "moindre" que celui-ci).

Sans `typedef` nous `typedef` un pointeur de fonction comme argument à une fonction de la manière suivante:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {
    /* inside of this block, the function is named "compare" */
}
```

Avec un `typedef` , on écrirait:

```
typedef int (*compare_func)(const void *, const void *);
```

et puis nous pourrions changer la signature de la fonction de `sort` en:

```
void sort(compare_func func) {
    /* In this block the function is named "func" */
}
```

les deux définitions de `sort` accepteraient n'importe quelle fonction de la forme

```
int compare(const void *arg1, const void *arg2) {
    /* Note that the variable names do not have to be "elem1" and "elem2" */
}
```

Les pointeurs de fonction sont le seul endroit où vous devez inclure la propriété de pointeur du type, par exemple, n'essayez pas de définir des types comme `typedef struct something_struct *something_type`

. Cela vaut même pour une structure avec des membres qui ne sont pas censés accéder directement aux appelants de l'API, par exemple le type de `FILE` stdio.h (qui, comme vous le constaterez maintenant, n'est pas un pointeur).

Prendre des pointeurs de contexte.

Un pointeur de fonction devrait presque toujours prendre un vide * fourni par l'utilisateur comme pointeur de contexte.

Exemple

```
/* function minimiser, details unimportant */
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)
{
    ...
    /* repeatedly make calls like this */
    temp = (*fptr)(testx, testy, ctx);
}

/* the function we are minimising, sums two cubics */
double *cubics(double x, double y, void *ctx)
{
    double *coeffsx = ctx;
    double *coeffsy = coeffx + 4;

    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];

}

void caller()
{
    /* context, the coefficients of the cubics */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}
```

L'utilisation du pointeur de contexte signifie que les paramètres supplémentaires n'ont pas besoin d'être codés en dur dans la fonction pointée ou nécessitent l'utilisation de globales.

La fonction de bibliothèque `qsort()` ne suit pas cette règle et on peut souvent s’en sortir sans contexte pour des fonctions de comparaison triviales. Mais pour quelque chose de plus compliqué, le pointeur de contexte devient essentiel.

Voir également

Fonctions pointeurs

introduction

Tout comme `char` et `int` , une fonction est une caractéristique fondamentale de C. En tant que tel, vous pouvez déclarer un pointeur sur un: cela signifie que vous pouvez passer à *quelle fonction appeler* une autre fonction pour l'aider à faire son travail. Par exemple, si vous aviez une fonction `graph()` qui affichait un graphique, vous pouviez transmettre *cette fonction à* `graph()` .

```
// A couple of external definitions to make the example clearer
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// The graph() function.
// Pass in the bounds: the minimum and maximum X and Y that should be plotted.
// Also pass in the actual function to plot.
void graph(double minX, double minY,
           double maxX, double maxY,
           ???? *fn) {            // See below for syntax

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);         // Get y for this x by calling passed-in fn()

        if (minY<=y && y<maxY) {
            plotXY(x, y);         // Plot calculated point
        } // if
    } for
} // graph(minX, minY, maxX, maxY, fn)
```

Usage

Donc, le code ci-dessus représentera graphiquement les fonctions que vous y avez passées - à condition que cette fonction réponde à certains critères: à savoir que vous transmettiez un `double` et obteniez un `double` . Il y a beaucoup de fonctions comme celle-ci - `sin()` , `cos()` , `tan()` , `exp()` etc. - mais il y en a beaucoup qui ne le sont pas, comme `graph()` lui-même!

Syntaxe

Alors, comment spécifiez-vous les fonctions que vous pouvez passer dans `graph()` et celles que vous ne pouvez pas? La méthode conventionnelle consiste à utiliser une syntaxe difficile à lire ou à comprendre:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

Le problème ci-dessus est que deux choses essaient d'être définies en même temps: la structure de la fonction et le fait que c'est un pointeur. Alors, divisez les deux définitions! Mais en utilisant `typedef` , une meilleure syntaxe (plus facile à lire et à comprendre) peut être obtenue.

Mnémonique pour écrire des pointeurs de fonction

Toutes les fonctions C sont en fait des pointeurs vers un point de la mémoire du programme où du code existe. L'utilisation principale d'un pointeur de fonction consiste à fournir un "rappel" à d'autres fonctions (ou à simuler des classes et des objets).

La syntaxe d'une fonction, définie plus bas sur cette page, est la suivante:

returnType (* nom) (paramètres)

Un mnémonique pour écrire une définition de pointeur de fonction est la procédure suivante:

1. Commencez par écrire une déclaration de fonction normale: `returnType name(parameters)`
2. Enveloppez le nom de la fonction avec la syntaxe du pointeur: `returnType (*name)(parameters)`

Les bases

Tout comme vous pouvez avoir un pointeur sur un **int** , **char** , **float** , **array / string** , **struct** , etc., vous pouvez avoir un pointeur sur une fonction.

La déclaration du pointeur prend la *valeur de retour de la fonction* , le *nom de la fonction* et le *type d'arguments / paramètres qu'il reçoit* .

Supposons que la fonction suivante soit déclarée et initialisée:

```
int addInt(int n, int m){
    return n+m;
}
```

Vous pouvez déclarer et initialiser un pointeur sur cette fonction:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

Si vous avez une fonction vide, cela pourrait ressembler à ceci:

```
void Print(void){
    printf("look ma' - no hands, only pointers!\n");
}
```

Ensuite, déclarer le pointeur serait:

```
void (*functionPtrPrint)(void) = Print;
```

Pour accéder à la fonction elle-même, il faut déréférencer le pointeur:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum
(*functionPtrPrint)(); //will print the text in Print function
```

Comme on le voit dans des exemples plus avancés de ce document, déclarer un pointeur sur une fonction peut être désordonné si la fonction est transmise avec plus de quelques paramètres. Si vous avez quelques pointeurs vers des fonctions ayant une "structure" identique (même type de valeur de retour et même type de paramètres), mieux vaut utiliser la commande **typedef** pour vous éviter de taper du code et rendre le code plus clair:

```
typedef int (*ptrInt)(int, int);

int Add(int i, int j){
    return i+j;
}

int Multiply(int i, int j){
    return i*j;
}

int main()
{
    ptrInt ptr1 = Add;
    ptrInt ptr2 = Multiply;

    printf("%d\n", (*ptr1)(2,3)); //will print 5
    printf("%d\n", (*ptr2)(2,3)); //will print 6
    return 0;
}
```

Vous pouvez également créer un **tableau de pointeurs de fonctions** . Si tous les pointeurs ont la même "structure":

```
int (*array[2]) (int x, int y); // can hold 2 function pointers
array[0] = Add;
array[1] = Multiply;
```

Vous pouvez en apprendre plus ici et ici .

Il est également possible de définir un tableau de pointeurs de fonctions de différents types, bien que cela nécessiterait de lancer quand vous voulez accéder à la fonction spécifique. Vous pouvez en apprendre plus ici .

Lire Pointeurs de fonction en ligne: https://riptutorial.com/fr/c/topic/250/pointeurs-de-fonction

Chapitre 46: Points de séquence

Remarques

Norme internationale ISO / IEC 9899: 201x Langages de programmation - C

> L'accès à un objet volatile, la modification d'un objet, la modification d'un fichier ou l'appel d'une fonction effectuant l'une de ces opérations sont tous *des effets secondaires* , à savoir des modifications de l'état de l'environnement d'exécution.
>
> La présence d'un *point de séquence* entre l'évaluation des expressions A et B implique que chaque calcul de valeur et effet secondaire associé à A est séquencé avant chaque calcul de valeur et tout effet secondaire associé à B.

Voici la liste complète des points de séquence de l'annexe C du projet de publication préalable en ligne 2011 de la norme du langage C:

> **Points de séquence**
>
> 1 Les points de séquence décrits au 5.1.2.3 sont les suivants:
>
> - Entre les évaluations du désignateur de fonction et les arguments réels dans un appel de fonction et l'appel réel. (6.5.2.2).
> - Entre les évaluations des premier et second opérandes des opérateurs suivants: ET logiques `&&` (6.5.13); OU logique `||` (6.5.14); virgule `,` (6.5.17).
> - Entre les évaluations du premier opérande du conditionnel `? :` opérateur et celui des deuxième et troisième opérandes sont évalués (6.5.15).
> - La fin d'un déclarateur complet: les déclarants (6.7.6);
> - Entre l'évaluation d'une expression complète et la prochaine expression complète à évaluer. Les expressions suivantes sont des expressions complètes: un initialiseur qui ne fait pas partie d'un littéral composé (6.7.9); l'expression dans une expression (6.8.3); l'expression de contrôle d'une instruction de sélection (`if` ou `switch`) (6.8.4); l'expression de contrôle d'une instruction `while` ou `do` (6.8.5); chacune des expressions (facultatives) d'une déclaration `for` (6.8.5.3); l'expression (facultatif) dans une déclaration de `return` (6.8.6.4).
> - Immédiatement avant qu'une fonction de bibliothèque ne retourne (7.1.4).
> - Après les actions associées à chaque spécificateur de conversion de fonction d'entrée / sortie formaté (7.21.6, 7.29.2).
> - Immédiatement avant et immédiatement après chaque appel à une fonction de comparaison, ainsi qu'entre tout appel à une fonction de comparaison et tout mouvement des objets transmis comme arguments à cet appel (7.22.5).

Examples

Expressions séquencées

Les expressions suivantes sont *séquencées* :

```
a && b
a || b
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

Dans tous les cas, l'expression `a` est pleinement évaluée et *tous les effets secondaires sont appliqués* avant que `b` ou `c` soient évalués. Dans le quatrième cas, seul l'un des `b` ou `c` sera évalué. Dans le dernier cas, `b` est entièrement évalué et tous les effets secondaires sont appliqués avant que `c` soit évalué.

Dans tous les cas, l'évaluation de l'expression `a` est *séquencée avant* les évaluations de `b` ou `c` (alternativement, les évaluations de `b` et `c` sont *séquencées après* évaluation de `a`).

Ainsi, les expressions comme

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

avoir un comportement bien défini.

Expressions sans séquence

C11

Les expressions suivantes ne sont pas *séquencées* :

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

Dans les exemples ci-dessus, l'expression `a` peut être évaluée avant ou après l'expression `b` , `b` peut être évaluée avant `a` , ou même peuvent être mélangées si elles correspondent à plusieurs instructions.

Une règle similaire s'applique aux appels de fonction:

```
f(a, b);
```

Ici , non seulement `a` et `b` sont non séquencée (par exemple la `,` l' opérateur dans un appel de

fonction *ne* produit *pas* un point de séquence) , mais aussi `f` , l'expression qui détermine la fonction qui doit être appelée.

Les effets secondaires peuvent être appliqués immédiatement après l'évaluation ou différés jusqu'à un stade ultérieur.

Des expressions comme

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

ou

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

va donner *un comportement indéfini* parce que

- une modification d'un objet et tout autre accès à celui-ci doit être séquencé
- l'ordre d'évaluation et l'ordre dans lequel *les effets secondaires* [1] sont appliqués ne sont pas spécifiés.

---

1 Toute modification de l'état de l'environnement d'exécution.

Expressions indéterminées

Les appels de fonction comme `f(a)` impliquent toujours un point de séquence entre l'évaluation des arguments et l'indicatif (ici `f` et `a`) et l'appel réel. Si deux appels de ce type ne sont pas séquencés, les deux appels de fonction sont indéfiniment séquencés, c'est-à-dire que l'un est exécuté avant l'autre et que l'ordre n'est pas spécifié.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
}
```

Cette double modification implicite du `counter` lors de l'évaluation des arguments `printf` est valide, nous ne savons tout simplement pas lequel des appels vient en premier. Comme la commande n'est pas spécifiée, elle peut varier et ne peut pas être utilisée. L'impression pourrait donc être:

> la commande est 0 1

ou

> la commande est 1 0

La déclaration analogue à ce qui précède sans appel de fonction intermédiaire

```
printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

a un comportement indéfini car il n'y a pas de point de séquence entre les deux modifications du `counter` .

Lire Points de séquence en ligne: https://riptutorial.com/fr/c/topic/1275/points-de-sequence

Chapitre 47: Portée de l'identifiant

Examples

Portée du bloc

Un identificateur a une portée de bloc si sa déclaration correspondante apparaît dans un bloc (la déclaration de paramètre dans la définition de fonction s'applique). La portée se termine à la fin du bloc correspondant.

Aucune autre entité avec le même identifiant ne peut avoir la même portée, mais les étendues peuvent se chevaucher. En cas de chevauchement des étendues, la seule visible est celle déclarée dans la portée la plus profonde.

```
#include <stdio.h>

void test(int bar)                   // bar has scope test function block
{
    int foo = 5;                     // foo has scope test function block
    {
        int bar = 10;                // bar has scope inner block, this overlaps with previous
test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                                // end of scope for inner bar
    printf("%d %d\n", foo, bar);     // 5 5, here bar is test:bar
}                                    // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;         // foo has scope main function block

    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                        // end of scope for main:foo
```

Fonction Prototype Scope

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope.  These names
   are not significant outside the prototype itself.  This is demonstrated
   below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

    orange = test_function(orange);
    printf("%d\r\n", orange); //orange = 6
```

```
    return 0;
}

int test_function(int fruit)
{
    fruit += 1;
    return fruit;
}
```

Notez que vous obtenez des messages d'erreur déroutants si vous introduisez un nom de type dans un prototype:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

Avec GCC 6.3.0, ce code (fichier source `dc11.c`) produit:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: 'struct whatever' declared inside parameter list will not be visible
outside of this definition or declaration [-Werror]
     int function(struct whatever *arg);
                         ^~~~~~~~
dc11.c:9:9: error: conflicting types for 'function'
     int function(struct whatever *arg)
         ^~~~~~~~
dc11.c:1:9: note: previous declaration of 'function' was here
     int function(struct whatever *arg);
         ^~~~~~~~
cc1: all warnings being treated as errors
$
```

Placez la définition de la structure avant la déclaration de la fonction, ou ajoutez `struct whatever;` comme une ligne avant la déclaration de fonction, et il n'y a pas de problème. Vous ne devez pas introduire de nouveaux noms de type dans un prototype de fonction car il n'y a aucun moyen d'utiliser ce type, et donc aucun moyen de définir ou d'utiliser cette fonction.

Portée du fichier

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
   It can be used anywhere after the declaration until the end of
   the translation unit. */
static int foo;
```

```
void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

Portée de la fonction

La portée de la fonction est la portée spéciale pour les **étiquettes** . Cela est dû à leur propriété inhabituelle. Une **étiquette** est visible à travers toute la fonction, elle est définie et on peut y sauter (en utilisant l'instruction `goto` *`label`*) depuis n'importe quel point de la même fonction. Bien que cela ne soit pas utile, l'exemple suivant illustre ce point:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
  OUTSIDE:
    if (a!=0) {
        int i=0;
      INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

`INSIDE` peut sembler défini *à* l' *intérieur* du bloc `if` , comme c'est le cas pour `i` quel champ est le bloc, mais ce n'est pas le cas. Il est visible dans toute la fonction en tant qu'instruction `goto INSIDE;` illustre. Il ne peut donc pas y avoir deux étiquettes avec le même identifiant dans une même fonction.

Une utilisation possible est le schéma suivant pour réaliser des nettoyages complexes corrects des ressources allouées:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
   double* a = malloc(sizeof(double[34]));
   if (!a) {
      fprintf(stderr,"can't allocate\n");
      return;                 /* No point in freeing a if it is null */
   }
   FILE* b = fopen("some_file","r");
```

```
    if (!b) {
        fprintf(stderr,"can't open\n");
        goto CLEANUP1;          /* Free a; no point in closing b */
    }
    /* do something reasonable */
    if (error) {
        fprintf(stderr,"something's wrong\n");
        goto CLEANUP2;       /* Free a and close b to prevent leaks */
    }
    /* do yet something else */
CLEANUP2:
    close(b);
CLEANUP1:
    free(a);
}
```

Les étiquettes telles que `CLEANUP1` et `CLEANUP2` sont des identificateurs spéciaux qui se comportent différemment de tous les autres identificateurs. Ils sont visibles de partout à l' intérieur de la fonction, même dans des endroits qui sont exécutées avant l'instruction étiquetée, ou même dans des endroits qui ne pourrait jamais être atteint si aucun des `goto` est exécutée. Les étiquettes sont souvent écrites en minuscule plutôt qu'en majuscule.

Lire Portée de l'identifiant en ligne: https://riptutorial.com/fr/c/topic/1804/portee-de-l-identifiant

Chapitre 48: Préprocesseur et Macros

Introduction

Toutes les commandes du préprocesseur commencent par le symbole dièse (dound) `#` . La macro AC est simplement une commande de préprocesseur définie à l'aide de la directive de préprocesseur `#define` . Pendant la phase de prétraitement, le préprocesseur C (une partie du compilateur C) remplace simplement le corps de la macro où son nom apparaît.

Remarques

Lorsqu'un compilateur rencontre une macro dans le code, il effectue un remplacement de chaîne simple, aucune opération supplémentaire n'est effectuée. Pour cette raison, les modifications apportées par le préprocesseur ne respectent pas la portée des programmes C - par exemple, une définition de macro ne se limite pas à être dans un bloc, elle n'est donc pas affectée par un `'}'` qui termine une instruction de bloc.

Le préprocesseur n'est, de par sa conception, pas complet - il existe plusieurs types de calcul que le préprocesseur ne peut effectuer seul.

Les compilateurs ont généralement un indicateur de ligne de commande (ou paramètre de configuration) qui nous permet d'arrêter la compilation après la phase de prétraitement et d'inspecter le résultat. Sur les plates-formes POSIX, cet indicateur est `-E` . Ainsi, l'exécution de gcc avec cet indicateur imprime la source étendue à stdout:

```
$ gcc -E cprog.c
```

Souvent, le préprocesseur est implémenté en tant que programme séparé, qui est appelé par le compilateur. Le nom commun de ce programme est `cpp` . Un certain nombre de préprocesseurs émettent des informations complémentaires, telles que des informations sur les numéros de ligne, qui sont utilisées par les phases ultérieures de compilation pour générer des informations de débogage. Dans le cas où le préprocesseur est basé sur gcc, l'option -P supprime ces informations.

```
$ cpp -P cprog.c
```

Examples

Inclusion conditionnelle et modification de signature de fonction conditionnelle

Pour inclure conditionnellement un bloc de code, le préprocesseur a plusieurs directives (par exemple, `#if` , `#ifdef` , `#else` , `#endif` , etc.).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
 * has been defined
 */
#ifdef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Les opérateurs relationnels C normaux peuvent être utilisés pour la condition `#if`

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

Les directives `#if` se comportent de la même façon `if` instruction C `if` , elles ne doivent contenir que des expressions constantes intégrales et aucune projection. Il prend en charge un opérateur unaire supplémentaire, `defined( identifier )` , qui renvoie `1` si l'identifiant est défini, et `0` sinon.

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Modification de la signature de la fonction conditionnelle

Dans la plupart des cas, une version d'une application devrait avoir le moins de charge possible. Toutefois, lors du test d'une version intermédiaire, des journaux supplémentaires et des informations sur les problèmes détectés peuvent être utiles.

Par exemple, supposons qu'il y ait une fonction `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` qui, lorsque vous souhaitez créer un test, générera un journal sur son utilisation. Cependant, cette fonction est utilisée à plusieurs endroits et il est souhaitable que lors de la génération du journal, une partie de l'information consiste à savoir d'où provient la fonction appelée.

Donc, en utilisant la compilation conditionnelle, vous pouvez avoir quelque chose comme ceci dans le fichier d'inclusion déclarant la fonction. Cela remplace la version standard de la fonction par une version de débogage de la fonction. Le préprocesseur est utilisé pour remplacer les appels à la fonction `SerOpPluAllRead()` par des appels à la fonction `SerOpPluAllRead_Debug()` avec deux arguments supplémentaires, le nom du fichier et le numéro de ligne où la fonction est utilisée.

La compilation conditionnelle est utilisée pour choisir de remplacer ou non la fonction standard par une version de débogage.

```
#if 0
// function declaration and prototype for our debug version of the function.
SHORT   SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with
additional arguments.
#define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead_Debug(pPif,usLock,__FILE__,__LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

Cela vous permet de remplacer la version standard de la fonction `SerOpPluAllRead()` par une version qui fournira le nom du fichier et le numéro de ligne dans le fichier où la fonction est appelée.

Il y a une considération importante: *tout fichier utilisant cette fonction doit inclure le fichier d'en-tête où cette approche est utilisée pour que le préprocesseur puisse modifier la fonction. Sinon, vous verrez une erreur de l'éditeur de liens.*

La définition de la fonction ressemblerait à ce qui suit. Qu'est-ce que cette source fait est de demander que le préprocesseur renomme la fonction `SerOpPluAllRead()` pour être `SerOpPluAllRead_Debug()` et de modifier la liste des arguments pour inclure deux arguments supplémentaires, un pointeur sur le nom du fichier où la fonction a été appelée et le numéro de ligne dans le fichier dans lequel la fonction est utilisée.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT    SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char  xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen (aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf (xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d", pPif-
>husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
```

```
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

Inclusion du fichier source

Les utilisations les plus courantes des directives de prétraitement `#include` sont les suivantes:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` remplace l'instruction par le contenu du fichier auquel il est fait référence. Les crochets d'angle (<>) font référence aux fichiers d'en-tête installés sur le système, tandis que les guillemets ("") sont destinés aux fichiers fournis par l'utilisateur.

Les macros elles-mêmes peuvent développer d'autres macros une fois, comme l'illustre cet exemple:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"
    /*  and so on */
#else
    #define INCFILE  "versN.h"
#endif
/* ... */
#include INCFILE
```

Remplacement Macro

La forme la plus simple de remplacement de macro consiste à définir une `manifest constant` , comme dans

```
#define ARRSIZE 100
int array[ARRSIZE];
```

Ceci définit une macro de *type fonction* qui multiplie une variable par `10` et stocke la nouvelle valeur:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b);   // good: ((b) *= 10);
TIMES10(c);   // good: ((c) *= 10);
TIMES10(5);   // bad:  ((5) *= 10);
```

Le remplacement est effectué avant toute autre interprétation du texte du programme. Dans le premier appel à `TIMES10` le nom `A` de la définition est remplacé par `b` et le texte ainsi étendu est alors mis à la place de l'appel. Notez que cette définition de `TIMES10` n’est pas équivalente à

```
#define TIMES10(A) ((A) = (A) * 10)
```

parce que cela pourrait évaluer le remplacement de `A` , deux fois, ce qui peut avoir des effets secondaires indésirables.

Ce qui suit définit une macro de type fonction dont la valeur est le maximum de ses arguments. Il présente l'avantage de fonctionner pour tous les types compatibles d'arguments et de générer du code en ligne sans la surcharge des appels de fonctions. Il présente l'inconvénient d'évaluer une ou l'autre de ses arguments une seconde fois (y compris les effets secondaires) et de générer plus de code qu'une fonction si elle est invoquée plusieurs fois.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43);              /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j);       /* i == 4 */
```

De ce fait, les macros qui évaluent leurs arguments plusieurs fois sont généralement évitées dans le code de production. Depuis C11, il y a la fonctionnalité `_Generic` qui permet d'éviter de telles invocations multiples.

Les parenthèses abondantes dans les extensions de macro (à droite de la définition) garantissent que les arguments et l'expression résultante sont correctement liés et s'intègrent bien dans le contexte dans lequel la macro est appelée.

Directive d'erreur

Si le préprocesseur rencontre une directive `#error` , la compilation est interrompue et le message de diagnostic inclus est imprimé.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Sortie possible:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

#if 0 pour bloquer les sections de code

Si vous envisagez de supprimer des sections de code ou de les désactiver temporairement, vous pouvez les commenter avec un commentaire de bloc.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
*/
```

Cependant, si le code source que vous avez entouré d'un commentaire de bloc contient des commentaires de style bloc dans la source, la fin * / des commentaires de bloc existants peut rendre votre nouveau commentaire de bloc non valide et entraîner des problèmes de compilation.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;

    /* Return 5 */
    return i;
}
*/
```

Dans l'exemple précédent, les deux dernières lignes de la fonction et le dernier '* /' sont vus par le compilateur, il serait donc compilé avec des erreurs. Une méthode plus sûre consiste à utiliser une directive `#if 0` autour du code que vous souhaitez bloquer.

```
#if 0
/* #if 0 evaluates to false, so everything between here and the #endif are
 * removed by the preprocessor. */
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
#endif
```

Un avantage est que lorsque vous voulez revenir en arrière et trouver le code, il est beaucoup plus facile de rechercher "# 0" que de rechercher tous vos commentaires.

Un autre avantage très important est que vous pouvez imbriquer du code avec `#if 0` . Cela ne peut pas être fait avec des commentaires.

Une alternative à l'utilisation de `#if 0` consiste à utiliser un nom qui ne sera pas `#defined` mais qui expliquera davantage pourquoi le code est bloqué. Par exemple, s'il existe une fonction qui semble être du code mort inutile, vous pouvez utiliser `#if defined(POSSIBLE_DEAD_CODE)` ou `#if defined(FUTURE_CODE_REL_020201)` pour le code nécessaire une fois que d'autres fonctionnalités sont

en place ou quelque chose de similaire. Ensuite, lors du retour en arrière pour supprimer ou activer cette source, ces sections de source sont faciles à trouver.

Coller de jeton

Le collage de jetons permet de coller deux arguments de macro. Par exemple, `front##back` `frontback` . Un exemple célèbre est l'en-tête `<TCHAR.H>` de Win32. En standard C, on peut écrire `L"string"` pour déclarer une chaîne de caractères large. Cependant, Windows API permet de convertir entre des chaînes de caractères larges et des chaînes de caractères étroites simplement par `#define UNICODE` . Pour implémenter les littéraux de chaîne, `TCHAR.H` utilise cette

```
#ifdef UNICODE
#define TEXT(x) L##x
#endif
```

Chaque fois qu'un utilisateur écrit `TEXT("hello, world")` et que UNICODE est défini, le préprocesseur C concatène `L` et l'argument de la macro. `L` concaténé avec `"hello, world"` donne `L"hello, world"` .

Macros prédéfinies

Une macro prédéfinie est une macro qui est déjà comprise par le préprocesseur C sans que le programme ait besoin de la définir. Les exemples comprennent

Macros prédéfinies obligatoires

- `__FILE__` , qui donne le nom du fichier source actuel (un littéral de chaîne),
- `__LINE__` pour le numéro de ligne actuel (une constante entière),
- `__DATE__` pour la date de compilation (un littéral de chaîne),
- `__TIME__` pour l'heure de compilation (un littéral de chaîne).

Il existe également un identifiant prédéfini, `__func__` (ISO / IEC 9899: 2011 §6.4.2.2), qui n'est *pas* une macro:

> L'identifiant `__func__` doit être implicitement déclaré par le traducteur comme si, immédiatement après l'accolade d'ouverture de chaque définition de fonction, la déclaration:
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> est apparu, où *nom-fonction* est le nom de la fonction lexicale.

`__FILE__` , `__LINE__` et `__func__` sont particulièrement utiles à des fins de débogage. Par exemple:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

Les compilateurs antérieurs à la norme C99 peuvent prendre en `__func__` ou non `__func__` ou

peuvent avoir une macro qui agit de la même manière, nommée différemment. Par exemple, gcc a utilisé `__FUNCTION__` en mode C89.

Les macros ci-dessous permettent de demander des détails sur l'implémentation:

- `__STDC_VERSION__` La version du standard C implémentée. C'est un entier constant utilisant le format `yyyymmL` (la valeur `201112L` pour C11, la valeur `199901L` pour C99; elle n'a pas été définie pour C89 / C90)
- `__STDC_HOSTED__` `1` s'il s'agit d'une implémentation hébergée, sinon `0` .
- `__STDC__` Si `1` , la mise en œuvre est conforme à la norme C.

Autres macros prédéfinies (non obligatoire)

ISO / IEC 9899: 2011 §6.10.9.2 Macros d'environnement:

- `__STDC_ISO_10646__` Une constante entière de la forme `yyyymmL` (par exemple, 199712L). Si ce symbole est défini, chaque caractère de l'ensemble requis Unicode, lorsqu'il est stocké dans un objet de type `wchar_t` , a la même valeur que l'identificateur abrégé de ce caractère. L'ensemble requis Unicode comprend tous les caractères définis par ISO / IEC 10646, ainsi que tous les amendements et corrigenda techniques, à compter de l'année et du mois spécifiés. Si un autre encodage est utilisé, la macro ne doit pas être définie et le codage réel utilisé est défini par la mise en œuvre.

- `__STDC_MB_MIGHT_NEQ_WC__` La constante entière 1, destinée à indiquer que, dans l'encodage de `wchar_t` , un membre du jeu de caractères de base n'a pas besoin d'une valeur de code égale à sa valeur lorsqu'il est utilisé comme caractère unique dans une constante de caractère entier.

- `__STDC_UTF_16__` La constante entière 1, destinée à indiquer que les valeurs de type `char16_t` sont `char16_t` UTF-16. Si un autre encodage est utilisé, la macro ne doit pas être définie et le codage réel utilisé est défini par la mise en œuvre.

- `__STDC_UTF_32__` La constante entière 1, destinée à indiquer que les valeurs de type `char32_t` sont `char32_t` UTF-32. Si un autre encodage est utilisé, la macro ne doit pas être définie et le codage réel utilisé est défini par la mise en œuvre.

ISO / IEC 9899: 2011 §6.10.8.3 Macros à caractéristiques conditionnelles

- `__STDC_ANALYZABLE__` La constante entière 1, destinée à indiquer la conformité aux spécifications de l'annexe L (Analyse).
- `__STDC_IEC_559__` La constante entière 1, destinée à indiquer la conformité aux spécifications de l'annexe F (arithmétique à virgule flottante CEI 60559).
- `__STDC_IEC_559_COMPLEX__` La constante entière 1, destinée à indiquer le respect des spécifications de l'annexe G (arithmétique complexe compatible CEI 60559).
- `__STDC_LIB_EXT1__` La constante entière `201112L` , destinée à indiquer la prise en charge des extensions définies dans l'annexe K (interfaces de vérification des limites).

- `__STDC_NO_ATOMICS__` La constante entière 1, destinée à indiquer que l'implémentation ne prend pas en charge les types atomiques (y compris le `_Atomic` type `_Atomic`) et l'en-tête `<stdatomic.h>` .
- `__STDC_NO_COMPLEX__` La constante entière 1, destinée à indiquer que l'implémentation ne prend pas en charge les types complexes ou l'en-tête `<complex.h>` .
- `__STDC_NO_THREADS__` La constante entière 1, destinée à indiquer que l'implémentation ne prend pas en charge l'en-tête `<threads.h>` .
- `__STDC_NO_VLA__` La constante entière 1, destinée à indiquer que l'implémentation ne prend pas en charge les tableaux de longueur variable ou les types modifiés de manière variable.

Header Inclure les gardes

Presque tous les fichiers d'en-tête doivent suivre l'idiome include guard :

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

Cela garantit que lorsque vous `#include "my-header-file.h"` à plusieurs endroits, vous n'obtenez pas de déclarations en double des fonctions, des variables, etc. Imaginez la hiérarchie de fichiers suivante:

en-tête-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

en-tête-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

principal c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Ce code pose un grave problème: le contenu détaillé de `MyStruct` est défini deux fois, ce qui n'est pas autorisé. Cela entraînerait une erreur de compilation qui peut être difficile à détecter, car un fichier d'en-tête en inclut un autre. Si vous l'avez fait avec des gardes d'en-tête:

en-tête-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

en-tête-2.h

```
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

principal c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Ce serait ensuite élargir à:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H
```

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
}
```

Lorsque le compilateur atteint la seconde inclusion de **header-1.h** , `HEADER_1_H` était déjà défini par l'inclusion précédente. Ergo, cela se résume à ce qui suit:

```
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

Et donc il n'y a pas d'erreur de compilation.

Remarque: il existe plusieurs conventions différentes pour nommer les gardes d'en-tête. Certains utilisateurs aiment l' `HEADER_2_H_` , certains incluent le nom du projet comme `MY_PROJECT_HEADER_2_H` . L'important est de s'assurer que la convention que vous suivez fait en sorte que chaque fichier de votre projet ait une protection d'en-tête unique.

Si les détails de la structure n'étaient pas inclus dans l'en-tête, le type déclaré serait incomplet ou un type opaque . De tels types peuvent être utiles, en masquant les détails de la mise en œuvre aux utilisateurs des fonctions. À bien des égards, le type `FILE` dans la bibliothèque C standard peut être considéré comme un type opaque (bien qu'il ne soit généralement pas opaque, de sorte que les implémentations macro des fonctions E / S standard puissent utiliser les éléments internes de la structure). Dans ce cas, l' `header-1.h` pourrait contenir:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;
```

```
int myFunction(MyStruct *value);

#endif
```

Notez que la structure doit avoir un nom de balise (ici `MyStruct` - qui se trouve dans l'espace de noms des balises, distinct de l'espace de noms des identificateurs ordinaires du nom de `MyStruct`), et que le `{ … }` est omis. Cela dit "il existe un type de structure `struct MyStruct` et il existe un alias pour `MyStruct` ".

Dans le fichier d'implémentation, les détails de la structure peuvent être définis pour que le type soit complet:

```
struct MyStruct {
    …
};
```

Si vous utilisez C11, vous pouvez répéter `typedef struct MyStruct MyStruct;` déclaration sans provoquer une erreur de compilation, mais les versions antérieures de C se plaignent. Par conséquent, il est toujours préférable d'utiliser l'idiome include guard, même si dans cet exemple, il serait facultatif que le code ne soit compilé qu'avec des compilateurs prenant en charge C11.

De nombreux compilateurs prennent en charge la directive `#pragma once` , qui a les mêmes résultats:

my-header-file.h

```
#pragma once

// Code for header file
```

Cependant, `#pragma once` ne fait pas partie du standard C, le code est moins portable si vous l'utilisez.

Quelques en-têtes n'utilisent pas l'idiome include guard. Un exemple spécifique est l'en-tête standard `<assert.h>` . Il peut être inclus plusieurs fois dans une seule unité de traduction, ce qui dépend si la macro `NDEBUG` est définie chaque fois que l'en-tête est inclus. Vous pouvez parfois avoir une exigence analogue; de tels cas seront rares. Normalement, vos en-têtes doivent être protégés par l'idiome de garde inclus.

Mise en œuvre de la prévention

Nous pouvons également utiliser des macros pour rendre le code plus facile à lire et à écrire. Par exemple, nous pouvons implémenter des macros pour implémenter la construction `foreach` dans C pour certaines structures de données telles que les listes à liaisons simples et doubles, les files d'attente, etc.

Voici un petit exemple.

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedListNode
{
    int data;
    struct LinkedListNode *next;
};

#define FOREACH_LIST(node, list) \
     for (node=list; node; node=node->next)

/* Usage */
int main(void)
{
    struct LinkedListNode *list, **plist = &list, *node;
    int i;

    for (i=0; i<10; i++)
    {
         *plist = malloc(sizeof(struct LinkedListNode));
         (*plist)->data = i;
         (*plist)->next = NULL;
         plist          = &(*plist)->next;
    }

    /* printing the elements here */
    FOREACH_LIST(node, list)
    {
        printf("%d\n", node->data);
    }
}
```

Vous pouvez créer une interface standard pour de telles structures de données et écrire une implémentation générique de `FOREACH` comme `FOREACH` :

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CollectionItem_
{
    int data;
    struct CollectionItem_ *next;
} CollectionItem;

typedef struct Collection_
{
    /* interface functions */
    void* (*first)(void *coll);
    void* (*last) (void *coll);
    void* (*next) (void *coll, CollectionItem *currItem);

    CollectionItem *collectionHead;
    /* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
```

```
}

/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
    CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
    item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
    CollectionItem **item = &coll->collectionHead;
    while(*item)
        item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
    Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
    nc->last  = last;
    nc->next  = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection)                      \
    for (node  = (collection)->first(collection);      \
         node != (collection)->last(collection);       \
         node  = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
         Add_Collection(coll, i);
    }

    /* printing the elements here */
    FOREACH(node, coll)
    {
        printf("%d\n", node->data);
    }
```

```
}
```

Pour utiliser cette implémentation générique, implémentez simplement ces fonctions pour votre structure de données.

```
1.  void* (*first)(void *coll);
2.  void* (*last) (void *coll);
3.  void* (*next) (void *coll, CollectionItem *currItem);
```

__cplusplus pour utiliser des C externes dans du code C ++ compilé avec C ++

Il existe des moments où un fichier include doit générer une sortie différente du préprocesseur selon que le compilateur est un compilateur C ou un compilateur C ++ en raison de différences de langage.

Par exemple, une fonction ou un autre composant externe est défini dans un fichier source C mais est utilisé dans un fichier source C ++. Comme C ++ utilise la gestion des noms (ou la décoration des noms) pour générer des noms de fonctions uniques basés sur des types d'arguments de fonction, une déclaration de fonction C utilisée dans un fichier source C ++ provoquera des erreurs de liaison. Le compilateur C ++ modifiera le nom externe spécifié pour la sortie du compilateur en utilisant les règles de gestion de noms pour C ++. Le résultat est des erreurs de lien dues à des externes non trouvés lorsque la sortie du compilateur C ++ est liée à la sortie du compilateur C.

Comme les compilateurs C ne gèrent pas les noms mais que les compilateurs C ++ le font pour tous les libellés externes (noms de fonctions ou noms de variables) générés par le compilateur C ++, une macro de préprocesseur prédéfinie, `__cplusplus` , a été introduite pour permettre la détection du compilateur.

Pour contourner ce problème de sortie de compilateur incompatible pour les noms externes entre C et C ++, la macro `__cplusplus` est définie dans le préprocesseur C ++ et n'est pas définie dans le préprocesseur C. Ce nom de macro peut être utilisé avec la directive `#ifdef` ou le `#ifdef #if` préprocesseur conditionnel avec l'opérateur `defined()` pour indiquer si un code source ou un fichier d'inclusion est en cours de compilation en C ++ ou C.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

Ou vous pourriez utiliser

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

Pour spécifier le nom de fonction correct d'une fonction à partir d'un fichier source C compilé avec le compilateur C utilisé dans un fichier source C ++, vous pouvez rechercher la constante `__cplusplus` définie pour provoquer l' `extern "C" { /* ... */ };` à utiliser pour déclarer des C externes lorsque le fichier d'en-tête est inclus dans un fichier source C ++. Cependant, lorsque compilé avec un compilateur C, le `extern "C" { */ ... */ };` N'est pas utilisé. Cette compilation conditionnelle est nécessaire car `extern "C" { /* ... */ };` est valide en C ++ mais pas en C.

```
#ifdef __cplusplus
// if we are being compiled with a C++ compiler then declare the
// following functions as C functions to prevent name mangling.
extern "C" {
#endif

// exported C function list.
int foo (void);

#ifdef __cplusplus
// if this is a C++ compiler, we need to close off the extern declaration.
};
#endif
```

Macros de type fonction

Les macros de type fonction sont similaires aux fonctions `inline` , elles sont utiles dans certains cas, comme le journal de débogage temporaire:

```
#ifdef DEBUG
# define LOGFILENAME "/tmp/logfile.log"

# define LOG(str) do {                              \
  FILE *fp = fopen(LOGFILENAME, "a");              \
  if (fp) {                                         \
    fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
                 /* don't print null pointer */    \
                 str ?str :"<null>");              \
    fclose(fp);                                     \
  }                                                 \
  else {                                            \
    perror("Opening '" LOGFILENAME "' failed");    \
  }                                                 \
} while (0)
#else
  /* Make it a NOOP if DEBUG is not defined. */
# define LOG(LINE) (void)0
#endif


#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        LOG("There are command line arguments");
    else
        LOG("No command line arguments");
    return 0;
```

```
}
```

Dans les deux cas (avec `DEBUG` ou non), l'appel se comporte de la même manière qu'une fonction avec un type de retour `void` . Cela garantit que les `if/else` sont interprétées comme prévu.

Dans le cas de `DEBUG` , ceci est implémenté via une construction `do { ... } while(0)` . Dans l'autre cas, `(void)0` est une déclaration sans effet secondaire qui est simplement ignorée.

Une alternative pour ce dernier serait

```
#define LOG(LINE) do { /* empty */ } while (0)
```

tel que dans tous les cas, il est syntaxiquement équivalent au premier.

Si vous utilisez GCC, vous pouvez également mettre en œuvre une macro de type fonction qui renvoie le résultat en utilisant une extension GNU non standard - expressions de déclaration . Par exemple:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
        int i, r = 1; \
        for (i = 0; i < Y; ++i) \
                r *= X; \
        r; \ // returned value is result of last operation
})

int main(void)
{
        int result;

        result = POW(2, 3);
        printf("Result: %d\n", result);
}
```

La macro des arguments Variadic

C99

Macros avec des arguments variadic:

Disons que vous voulez créer une macro d'impression pour déboguer votre code, prenons cette macro comme exemple:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Quelques exemples d'utilisation:

La fonction `somefunc()` renvoie -1 si elle échoue et 0 si elle réussit, et elle est appelée depuis de nombreux endroits différents du code:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */

 retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

Que se passe-t-il si l'implémentation de `somefunc()` change, et renvoie maintenant des valeurs différentes correspondant à différents types d'erreur possibles? Vous souhaitez toujours utiliser la macro de débogage et imprimer la valeur d'erreur.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

Pour résoudre ce problème, la macro `__VA_ARGS__` été introduite. Cette macro permet plusieurs paramètres X-macro:

Exemple:

```
 #define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Usage:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

Cette macro vous permet de transmettre plusieurs paramètres et de les imprimer, mais cela vous interdit désormais d’envoyer des paramètres.

```
debug_print("Hey");
```

Cela soulèverait une erreur de syntaxe car la macro attend au moins un argument supplémentaire et le pré-processeur n'ignorerait pas le manque de virgule dans la macro `debug_print()` . Aussi `debug_print("Hey",);` générer une erreur de syntaxe car vous ne pouvez pas garder l'argument passé à la macro vide.

Pour résoudre ce problème, la macro `##__VA_ARGS__` été introduite, cette macro indique que si aucun argument variable n'existe, la virgule est supprimée par le pré-processeur du code.

Exemple:

```
#define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                              printf("\nError occured in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Usage:

```
debug_print("Ret val of somefunc()?");
debug_print("%d",somefunc());
```

Lire Préprocesseur et Macros en ligne: https://riptutorial.com/fr/c/topic/447/preprocesseur-et-macros

Chapitre 49: Qualificatifs de type

Remarques

Les qualificatifs de type sont les mots-clés décrivant la sémantique supplémentaire d'un type. Ils font partie intégrante des signatures de type. Ils peuvent apparaître à la fois au plus haut niveau d'une déclaration (affectant directement l'identifiant) ou à des sous-niveaux (pertinents uniquement pour les pointeurs, affectant les valeurs pointées):

Mot-clé	Remarques
`const`	Empêche la mutation de l'objet déclaré (en apparaissant au plus haut niveau) ou empêche la mutation de la valeur pointée (en apparaissant à côté d'un sous-type de pointeur).
`volatile`	Informe le compilateur que l'objet déclaré (au plus haut niveau) ou la valeur pointée (dans les sous-types de pointeurs) peut changer sa valeur en raison de conditions externes, et pas uniquement en raison du flux de contrôle du programme.
`restrict`	Un indice d'optimisation, uniquement pour les pointeurs. Déclare l'intention que pour la durée de vie du pointeur, aucun autre pointeur ne sera utilisé pour accéder au même objet pointé.

L'ordre des qualificateurs de type en ce qui concerne les spécificateurs de classe de stockage (`static` , `extern` , `auto` , `register`), les modificateurs de type (`signed` , `unsigned` , `short` , `long`) et les spécificateurs de type (`int` , `char` , `double` , etc.) la bonne pratique est de les mettre dans l'ordre susmentionné:

```
static const volatile unsigned long int a = 5; /* good practice */
unsigned volatile long static int const b = 5; /* bad practice */
```

Qualifications de haut niveau

```
/* "a" cannot be mutated by the program but can change as a result of external conditions */
const volatile int a = 5;

/* the const applies to array elements, i.e. "a[0]" cannot be mutated */
const int arr[] = { 1, 2, 3 };

/* for the lifetime of "ptr", no other pointer could point to the same "int" object */
int *restrict ptr;
```

Qualifications du sous-type de pointeur

```
/* "s1" can be mutated, but "*s1" cannot */
const char *s1 = "Hello";

/* neither "s2" (because of top-level const) nor "*s2" can be mutated */
const char *const s2 = "World";

/* "*p" may change its value as a result of external conditions, "**p" and "p" cannot */
char *volatile *p;

/* "q", "*q" and "**q" may change their values as a result of external conditions */
volatile char *volatile *volatile q;
```

Examples

Variables non modifiables (const)

```
const int a = 0; /* This variable is "unmodifiable", the compiler
                    should throw an error when this variable is changed */
int b = 0; /* This variable is modifiable */

b += 10; /* Changes the value of 'b' */
a += 10; /* Throws a compiler error */
```

La qualification `const` ne signifie que nous n'avons pas le droit de modifier les données. Cela ne signifie pas que la valeur ne peut pas changer dans notre dos.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

Pendant l'exécution des autres appels, `*a` peut-être changé, et cette fonction peut donc retourner soit `false` soit `true` .

Attention

Les variables avec qualification de `const` peuvent toujours être modifiées à l'aide de pointeurs:

```
const int a = 0;

int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */
*a_ptr += 10;          /* This has undefined behavior */

printf("a = %d\n", a); /* May print: "a = 10" */
```

Mais le faire est une erreur qui conduit à un comportement indéfini. La difficulté ici est que cela peut se comporter comme prévu dans des exemples simples comme cela, mais ensuite se tromper lorsque le code se développe.

Variables volatiles

Le mot-clé `volatile` indique au compilateur que la valeur de la variable peut changer à tout moment en raison de conditions externes, et pas uniquement en raison du flux de contrôle du programme.

Le compilateur n'optimisera rien de la variable volatile.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;

volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

Il y a deux raisons principales pour utiliser des variables volatiles:

- Pour interfacer avec le matériel qui a des registres d'E / S mappés en mémoire.
- Lors de l'utilisation de variables modifiées en dehors du flux de contrôle du programme (par exemple, dans une routine de service d'interruption)

Voyons cet exemple:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

Le compilateur est autorisé à remarquer que la boucle while ne modifie pas la variable de `quit` et convertit la boucle en une boucle `while (true)` sans fin. Même si la variable `quit` est définie sur le gestionnaire de signaux pour `SIGINT` et `SIGTERM` , le compilateur ne le sait pas.

Déclarer `quit` as `volatile` va dire au compilateur de ne pas optimiser la boucle et le problème sera résolu.

Le même problème se produit lors de l'accès au matériel, comme nous le voyons dans cet exemple:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

Le comportement de l'optimiseur est de lire la valeur de la variable une fois, il n'est pas nécessaire de la relire, car la valeur sera toujours la même. On se retrouve donc avec une boucle infinie. Pour forcer le compilateur à faire ce que nous voulons, nous modifions la déclaration pour:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Lire Qualificatifs de type en ligne: https://riptutorial.com/fr/c/topic/2588/qualificatifs-de-type

Chapitre 50: Relevés de sélection

Examples

if () Déclarations

L'un des moyens les plus simples de contrôler le déroulement du programme consiste à utiliser les instructions de sélection `if` . Si un bloc de code doit être exécuté ou ne pas être exécuté peut être décidé par cette instruction.

La syntaxe de `if` instruction de sélection dans C peut être la suivante:

```
if(cond)
{
  statement(s);  /*to be executed, on condition being true*/
}
```

Par exemple,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Où `a > 1` est une *condition* qui doit être évaluée à `true` pour exécuter les instructions dans le bloc `if` . Dans cet exemple, "a est supérieur à 1" n'est imprimé que si `a > 1` est vrai.

`if` instructions de sélection peuvent omettre les accolades `{` et `}` s'il n'y a qu'une seule instruction dans le bloc. L'exemple ci-dessus peut être réécrit pour

```
if (a > 1)
    puts("a is larger than 1");
```

Cependant, pour l'exécution de plusieurs instructions dans un bloc, les accolades doivent être utilisées.

La *condition* pour `if` peut inclure plusieurs expressions. `if` ne réalisera l'action que si le résultat final de l'expression est vrai.

Par exemple

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

n'exécutera le `printf` et `a++` si les **deux** `a` et `b` sont supérieurs à `1` .

if () ... else instructions et syntaxe

Alors que `if` effectue une action uniquement lorsque sa condition est évaluée à `true` , `if` / `else` vous permet de spécifier les différentes actions lorsque la condition est `true` et que la condition est `false` .

Exemple:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Tout comme l'instruction `if` , lorsque le bloc `if` ou `else` se compose d'une seule instruction, les accolades peuvent être omises (mais cela n'est pas recommandé car cela peut facilement introduire des problèmes involontairement). Cependant, s'il y a plus d'une instruction dans le bloc `if` ou `else` , les accolades doivent être utilisées sur ce bloc particulier.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

switch instructions ()

`switch` instructions `switch` sont utiles lorsque vous souhaitez que votre programme effectue de nombreuses opérations différentes en fonction de la valeur d'une variable de test particulière.

Voici un exemple d'utilisation de l'instruction `switch` :

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

Cet exemple est équivalent à

```
int a = 1;

if (a == 1) {
```

```
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

Si la valeur de `a` est 1 lorsque l'instruction `switch` est utilisée, `a is 1` sera imprimé. Si la valeur de `a` est 2 alors, `a is 2` sera imprimé. Sinon, `a is neither 1 nor 2` sera imprimé.

`case n:` est utilisé pour décrire l'endroit où le flux d'exécution interviendra lorsque la valeur transmise à l'instruction `switch` est *n* . *n* doit être une constante à la compilation et le même *n* peut exister au plus une fois dans une instruction de `switch` .

`default:` est utilisé pour décrire cela lorsque la valeur ne correspond à aucun des choix pour le `case n:` Il est recommandé d'inclure un cas `default` dans chaque instruction de commutateur pour détecter un comportement inattendu.

Une `break;` déclaration est nécessaire pour sauter hors du bloc de `switch` .

Remarque: Si vous oubliez accidentellement d'ajouter une `break` après la fin d'un `case` , le compilateur suppose que vous avez l' intention de « passer » et toutes les déclarations de cas suivantes, le cas échéant, seront exécutées (sauf si une instruction *break* se trouve dans n'importe lequel des cas suivants), que la déclaration de cas suivante corresponde ou non. Cette propriété particulière est utilisée pour implémenter le périphérique de Duff . Ce comportement est souvent considéré comme une faille dans la spécification du langage C.

Voici un exemple qui montre les effets de l’absence de `break;` :

```
int a = 1;

switch (a) {
case 1:
case 2:
    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}
```

Lorsque la valeur de `a` est 1 ou 2, `a is 1 or 2` et `a is 1, 2 or 3` seront tous deux imprimés. Quand `a` est 3, seulement `a is 1, 2 or 3` sera imprimé. Sinon, `a is neither 1, 2 nor 3` sera imprimé.

Notez que le cas `default` n'est pas nécessaire, surtout lorsque le jeu de valeurs que vous obtenez dans le `switch` est terminé et connu à la compilation.

Le meilleur exemple consiste à utiliser un `switch` sur un `enum` .

```
enum msg_type { ACK, PING, ERROR };
```

```
void f(enum msg_type t)
{
  switch (t) {
  case ACK:
    // do nothing
    break;
  case PING:
    // do something
    break;
  case ERROR:
    // do something else
    break;
  }
}
```

Cela présente plusieurs avantages:

- la plupart des compilateurs signaleront un avertissement si vous ne gérez pas une valeur (cela ne serait pas signalé si un cas `default` était présent)
- pour la même raison, si vous ajoutez une nouvelle valeur à l' `enum` , vous serez informé de tous les endroits où vous avez oublié de gérer la nouvelle valeur (avec un cas `default` , vous devrez explorer manuellement votre code pour rechercher ces cas)
- Le lecteur n'a pas besoin de déterminer "ce qui est masqué par la `default:` par `default:` ", s'il existe d'autres valeurs de `enum` ou s'il s'agit d'une protection pour "juste au cas où". Et s'il y a d'autres valeurs d' `enum` , le codeur a-t-il intentionnellement utilisé le cas `default` pour eux ou y a-t-il un bogue qui a été introduit lorsqu'il a ajouté la valeur?
- gérer chaque valeur d' `enum` rend le code explicite, car vous ne pouvez pas vous cacher derrière un caractère générique, vous devez gérer explicitement chacun d'eux.

Néanmoins, vous ne pouvez pas empêcher quelqu'un d'écrire du code maléfique comme:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Ainsi, vous pouvez ajouter une vérification supplémentaire avant votre commutateur pour le détecter, si vous en avez vraiment besoin.

```
void f(enum msg_type t)
{
   if (!is_msg_type_valid(t)) {
      // Handle this unlikely error
   }

   switch(t) {
    // Same code than before
   }
}
```

if () ... else Ladder Chaînage deux ou plus if () ... else instructions

Alors que l'instruction `if ()... else` permet de définir un seul comportement (par défaut) qui se produit lorsque la condition dans `if ()` n'est pas remplie, chaîner deux ou plusieurs `if () ... else` permet de définir un couple plus de comportements avant d' aller à la dernière `else` branche

agissant comme « par défaut », le cas échéant.

Exemple:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) //we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```

Imbriqué if () ... else VS if () .. sinon Ladder

Imbriquées `if()...else` instructions prennent plus de temps d'exécution (elles sont plus lentes) que les `if()...else` parce que les instructions `if()...else` vérifient toutes les instructions conditionnelles internes L'instruction conditionnelle `if()` est satisfaite, alors que l'échelle `if()..else` arrête le test de condition une fois que les instructions conditionnelles `if()` ou `else if()` sont vraies.

Une échelle `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if ((a < b) && (a < c))
  {
    printf("\na = %d is the smallest.", a);
  }
  else if ((b < a) && (b < c))
  {
    printf("\nb = %d is the smallest.", b);
  }
  else if ((c < a) && (c < b))
  {
    printf("\nc = %d is the smallest.", c);
  }
  else
  {
    printf("\nImprove your coding logic");
  }
  return 0;
}
```

Est-ce, dans le cas général, considéré comme meilleur que l'équivalent imbriqué `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if (a < b)
  {
    if (a < c)
      {
        printf("\na = %d is the smallest.", a);
      }
    else
      {
        printf("\nc = %d is the smallest.", c);
      }
  }
  else
  {
    if(b < c)
    {
      printf("\nb = %d is the smallest.", b);
    }
    else
    {
      printf("\nc = %d is the smallest.", c);
    }
  }
  return 0;
}
```

Lire Relevés de sélection en ligne: https://riptutorial.com/fr/c/topic/3073/releves-de-selection

Chapitre 51: Sélection générique

Syntaxe

- _Générique (expression-affectation, liste-générique-assoc)

Paramètres

Paramètre	Détails
liste générique-assoc	association générique **OU liste** générique-assoc, association générique
association générique	nom-type: expression-assignation **OU** par défaut: expression-affectation

Remarques

1. Tous les qualificateurs de type seront supprimés lors de l'évaluation de l'expression primaire `_Generic` .
2. `_Generic` expression primaire `_Generic` est évaluée à la phase de traduction 7 . Ainsi, des phases comme la concaténation de chaînes ont été terminées avant son évaluation.

Examples

Vérifier si une variable est d'un certain type qualifié

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x),  \
        const int *: "a const int",     \
        int *:       "a non-const int", \
        default:     "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Sortie:

```
i is a const int
j is a non-const int
k is of other type
```

Cependant, si la macro générique de type est implémentée comme ceci:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

La sortie est la suivante:

```
i is a non-const int
j is a non-const int
k is of other type
```

En effet, tous les qualificatifs de type sont supprimés pour l'évaluation de l'expression de contrôle d'une expression primaire `_Generic` .

Macro d'impression de type générique

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }
void print_dbl(double x) { printf("double: %g\n", x); }
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \
        int: print_int, \
        double: print_dbl, \
        default: print_default)(X)

int main(void) {
    print(42);
    print(3.14);
    print("hello, world");
}
```

Sortie:

```
int: 42
double: 3.14
unknown argument
```

Notez que si le type n'est ni `int` ni `double` , un avertissement sera généré. Pour éliminer l'avertissement, vous pouvez ajouter ce type à la macro `print(X)` .

Sélection générique basée sur plusieurs arguments

Si une sélection de plusieurs arguments pour une expression générique de type est souhaitée et

que tous les types en question sont des types arithmétiques, un moyen simple d'éviter les expressions `_Generic` imbriquées `_Generic` à utiliser les paramètres dans l'expression de contrôle:

```
int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),                \
                           int:      max_int,      \
                           unsigned: max_unsigned, \
                           default:  max_double)   \
                    ((X), (Y))
```

Ici, l'expression de contrôle `(X)+(Y)` n'est inspectée qu'en fonction de son type et n'est pas évaluée. Les conversions habituelles pour les opérandes arithmétiques sont effectuées pour déterminer le type sélectionné.

Pour une situation plus complexe, une sélection peut être effectuée en fonction de plusieurs arguments pour l'opérateur, en les imbriquant ensemble.

Cet exemple sélectionne entre quatre fonctions implémentées en externe, qui combinent deux arguments int et / ou chaîne et renvoient leur somme.

```
int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char*  a, int b );
int AddStrStr(const char*  a, const char*  b);

#define AddStr(y)                            \
   _Generic((y),         int: AddStrInt,     \
                        char*: AddStrStr,    \
                  const char*: AddStrStr )

#define AddInt(y)                            \
   _Generic((y),          int: AddIntInt,    \
                        char*: AddIntStr,    \
                  const char*: AddIntStr )

#define Add(x, y)                            \
   _Generic((x) ,        int: AddInt(y) ,    \
                       char*: AddStr(y) ,    \
                 const char*: AddStr(y))     \
                         ((x), (y))

int main( void )
{
    int result = 0;
    result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

    const int a = -123;
    char b[] = "4321";
    result = Add( a , b );

    int c = 1;
```

```
    const char d[] = "0";
    result = Add( d , ++c );
}
```

Même s'il semble que l'argument `y` est évalué plus d'une fois, ce n'est pas [1] . Les deux arguments sont évalués une seule fois, à la fin de la macro Add: `( x , y )` , comme dans un appel de fonction ordinaire.

---

[1] (Cité à partir de: ISO: IEC 9899: 201X 6.5.1.1 Sélection générique 3)
L'expression de contrôle d'une sélection générique n'est pas évaluée.

Lire Sélection générique en ligne: https://riptutorial.com/fr/c/topic/571/selection-generique

Chapitre 52: Séquence de caractères multi-caractères

Remarques

Tous les préprocesseurs ne prennent pas en charge le traitement des séquences de trigraphes. Certains compilateurs donnent une option ou un commutateur supplémentaire pour les traiter. D'autres utilisent un programme séparé pour convertir les trigraphes.

Le compilateur GCC ne les reconnaît pas à moins que vous ne le `-trigraphs` explicitement (utilisez `-trigraphs` pour les activer, utilisez `-Wtrigraphs` , une partie de `-Wall` , pour obtenir des avertissements sur les trigraphes).

Comme la plupart des plates-formes utilisées aujourd'hui prennent en charge la totalité des caractères uniques utilisés en C, les digraphes sont préférés aux trigraphes, mais l'utilisation de séquences de caractères multi-caractères est généralement déconseillée.

Aussi, méfiez-vous de l'utilisation accidentelle de trigraphes (`puts("What happened??!!");` , par exemple).

Examples

Trigraphes

Les symboles `[ ] { } ^ \ | ~ #` Sont fréquemment utilisés dans les programmes C, mais dans les fin des années 1980, il y avait des jeux de codes utilisés (variantes ISO 646, par exemple, dans les pays scandinaves) où les positions de caractères ASCII pour ceux - ci ont été utilisés pour les variantes de caractères langue nationale (par exemple `£` pour `#` au Royaume - Uni, `Æ Å æ å ø Ø` pour `{ } { } | \` au Danemark, il n'y avait pas `~` en EBCDIC). Cela signifiait qu'il était difficile d'écrire du code C sur les machines qui utilisaient ces ensembles.

Pour résoudre ce problème, le standard C a suggéré d'utiliser des combinaisons de trois caractères pour produire un seul caractère appelé trigraphe. Un trigraphe est une séquence de trois caractères, dont les deux premiers sont des points d'interrogation.

Voici un exemple simple qui utilise des séquences de trigraphes au lieu de `#` , `{` et `}` :

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

Cela sera modifié par le préprocesseur C en remplaçant les trigraphes par leurs équivalents à caractère unique comme si le code avait été écrit:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
}
```

Trigraph	Équivalent
?? =	#
?? /	\
?? '	^
?? (	[
??)	]
??!	|
?? <	{
??>	}
?? -	~

Notez que les trigraphes sont problématiques car, par exemple, `??/` est une barre oblique inverse et peut affecter la signification des lignes de continuation dans les commentaires, et doivent être reconnus dans les chaînes et les caractères littéraux (par exemple, `'??/??/'` est un simple) caractère, une barre oblique inverse).

Digraphes

C99

En 1994, des alternatives plus lisibles à cinq des trigraphes ont été fournies. Ceux-ci n'utilisent que deux caractères et sont appelés digraphs. Contrairement aux trigraphes, les digraphes sont des jetons. Si un digraphe apparaît dans un autre jeton (littéraux de chaîne ou constantes de caractères, par exemple), il ne sera pas traité comme un digraphe, mais restera tel quel.

Ce qui suit montre la différence avant et après le traitement de la séquence de digraphes.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Qui sera traité de la même façon que:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

Digraph	Équivalent
<:	[
:>	]
<%	{
%>	}
%:	#

Lire Séquence de caractères multi-caractères en ligne:
https://riptutorial.com/fr/c/topic/7111/sequence-de-caracteres-multi-caracteres

Chapitre 53: Structs

Introduction

Les structures permettent de regrouper un ensemble de variables connexes de divers types en une seule unité de mémoire. La structure dans son ensemble peut être référencée par un seul nom ou un seul pointeur; les membres de la structure sont également accessibles individuellement. Les structures peuvent être transmises à des fonctions et renvoyées à partir de fonctions. Ils sont définis à l'aide du mot `struct` clé `struct` .

Examples

Structures de données simples

Les types de données de structure sont un moyen utile de regrouper les données associées et de les faire se comporter comme une seule variable.

Déclarer une `struct` simple qui contient deux membres `int` :

```
struct point
{
    int x;
    int y;
};
```

`x` et `y` sont appelés les *membres* (ou *champs*) de la structure de `point` .

Définir et utiliser des structures:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Les structures peuvent être initialisées à la définition. Ce qui précède est équivalent à:

```
struct point p = {5, 3};
```

Les structures peuvent également être initialisées en utilisant des initialiseurs désignés .

L'accès aux champs se fait également à l'aide de `.` opérateur

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Typedef Structs

La combinaison de `typedef` avec `struct` peut rendre le code plus clair. Par exemple:

```
typedef struct
{
    int x, y;
} Point;
```

par opposition à:

```
struct Point
{
    int x, y;
};
```

pourrait être déclaré comme:

```
Point point;
```

au lieu de:

```
struct Point point;
```

Encore mieux est d'utiliser les éléments suivants

```
typedef struct Point Point;

struct Point
{
    int x, y;
};
```

pour bénéficier des deux définitions possibles du `point` . Une telle déclaration est plus pratique si vous avez d'abord appris C ++, où vous pouvez omettre le mot `struct` clé `struct` si le nom n'est pas ambigu.

`typedef` noms `typedef` pour les structures peuvent être en conflit avec d'autres identificateurs d'autres parties du programme. Certains considèrent cela comme un inconvénient, mais pour la plupart des gens ayant une `struct` et un autre identifiant, cela est très inquiétant. Notoire est par exemple POSIX ' `stat`

```
int stat(const char *pathname, struct stat *buf);
```

où vous voyez une fonction `stat` qui a un argument qui est `struct stat` .

Les structures `typedef` sans nom de tag imposent toujours que toute la déclaration de `struct` soit visible pour le code qui l'utilise. La déclaration de `struct` entière doit alors être placée dans un fichier d'en-tête.

Considérer:

```
#include "bar.h"

```

```
struct foo
{
    bar *aBar;
};
```

Ainsi, avec une `struct typedef` d sans nom de balise, le fichier `bar.h` doit toujours inclure toute la définition de `bar` . Si nous utilisons

```
typedef struct bar bar;
```

dans `bar.h` , les détails de la structure de `bar` peuvent être cachés.

Voir Typedef

Pointeurs vers les structures

Lorsque vous avez une variable contenant une `struct` , vous pouvez accéder à ses champs à l'aide de l'opérateur point (`.`). Cependant, si vous avez un pointeur sur une `struct` , cela ne fonctionnera pas. Vous devez utiliser l'opérateur de flèche (`->`) pour accéder à ses champs. Voici un exemple d'une implémentation extrêmement simple (certains pourraient dire "terrible et simple") d'une pile qui utilise des pointeurs pour `struct` et montre l'opérateur de la flèche.

```
#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
```

```
    }

    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
        }
    }

    /* destroy stack */
    destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}
```

```
/* Pop a value off of the stack. */
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}
```

Membres de tableau flexibles

C99

Déclaration de type

Une structure *avec au moins un membre* peut en outre contenir un seul élément de tableau de longueur non spécifiée à la fin de la structure. Cela s'appelle un membre de tableau flexible:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Effets sur la taille et le rembourrage

Un membre de groupe flexible est considéré comme n'ayant pas de taille lors du calcul de la taille d'une structure, même si le remplissage entre ce membre et le membre précédent de la structure peut toujours exister:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

Le membre de tableau flexible est considéré comme ayant un type de tableau incomplet, sa taille ne peut donc pas être calculée à l'aide de `sizeof` .

Usage

Vous pouvez déclarer et initialiser un objet avec un type de structure contenant un membre de tableau flexible, mais vous ne devez pas tenter d'initialiser le membre du groupe flexible car il est traité comme s'il n'existait pas. Il est interdit d'essayer de faire cela, et des erreurs de compilation en résulteront.

De même, vous ne devez pas tenter d'attribuer une valeur à un élément d'un membre de groupe flexible lors de la déclaration d'une structure de cette manière, car il peut ne pas y avoir suffisamment de remplissage à la fin de la structure pour autoriser les objets requis. Le compilateur ne vous empêchera pas nécessairement de le faire, cependant, cela peut conduire à un comportement indéfini.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

Vous pouvez plutôt choisir d'utiliser `malloc` , `calloc` ou `realloc` pour allouer la structure avec un espace de stockage supplémentaire et la libérer ultérieurement, ce qui vous permet d'utiliser le membre de tableau flexible comme vous le souhaitez:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
```

```
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

C99

Le 'struct hack'

Les membres de tableau flexibles n'existaient pas avant C99 et sont traités comme des erreurs. Une solution commune consiste à déclarer un tableau de longueur 1, une technique appelée «struct hack»:

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

Cela affectera la taille de la structure, cependant, contrairement à un véritable membre de tableau flexible:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

Pour utiliser le membre `flex` tant que membre de tableau flexible, vous devez l'allouer avec `malloc` comme indiqué ci-dessus, sauf que `sizeof(*pe1)` (ou la `sizeof(struct ex1)` équivalente `sizeof(struct ex1)`) sera remplacée par `offsetof(struct ex1, flex)` ou la plus longue, expression de type agnostique `sizeof(*pe1)-sizeof(pe1->flex)` . Alternativement, vous pouvez soustraire 1 de la longueur souhaitée du tableau "flexible" car il est déjà inclus dans la taille de la structure, en supposant que la longueur souhaitée est supérieure à 0. La même logique peut être appliquée aux autres exemples d'utilisation.

Compatibilité

Si la compatibilité avec les compilateurs ne `FLEXMEMB_SIZE` pas en charge les membres de tableau flexibles est souhaitée, vous pouvez utiliser une macro définie comme `FLEXMEMB_SIZE` ci-dessous:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
```

```
};
```

Lorsque vous `offsetof(struct ex1, flex)` objets, vous devez utiliser la forme `offsetof(struct ex1, flex)` pour faire référence à la taille de la structure (à l'exclusion du membre de tableau flexible) car c'est la seule expression qui reste cohérente entre les compilateurs ne pas:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

L'alternative consiste à utiliser le préprocesseur pour soustraire conditionnellement 1 de la longueur spécifiée. En raison du potentiel accru d'incohérence et d'erreur humaine générale sous cette forme, j'ai déplacé la logique dans une fonction distincte:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Passer des structures à des fonctions

En C, tous les arguments sont transmis aux fonctions par valeur, y compris les structures. Pour les petites structures, c'est une bonne chose car cela signifie qu'il n'y a pas de surcharge à accéder aux données via un pointeur. Cependant, il est également très facile de transmettre accidentellement une structure volumineuse entraînant de mauvaises performances, en particulier si le programmeur est habitué à d'autres langages où les arguments sont transmis par référence.

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

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
{
    int param1;
```

```
    char param2[80000];
};

// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}
```

Programmation par objets utilisant des structures

Les structures peuvent être utilisées pour implémenter du code de manière orientée objet. Une structure est similaire à une classe, mais manque les fonctions qui font normalement partie d'une classe, nous pouvons les ajouter en tant que variables de membre de pointeur de fonction. Pour rester avec notre exemple de coordonnées:

```
/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
/* Destructor */
void coordinate_destroy(coordinate *this);
```

Et maintenant le fichier C d'implémentation:

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
```

```
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
        printf("NULL pointer exception!\n");
    }
}
```

Un exemple d'utilisation de notre classe de coordonnées serait:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* Now we can use our objects using our methods and passing the object as parameter */
```

```
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Lire Structs en ligne: https://riptutorial.com/fr/c/topic/1119/structs

Chapitre 54: Structure rembourrage et emballage

Introduction

Par défaut, les compilateurs C disposent de structures permettant d'accéder rapidement à chaque membre, sans encourir de pénalités pour un accès non aligné, un problème avec les machines RISC telles que DEC Alpha et certains processeurs ARM.

Selon l'architecture du processeur et le compilateur, une structure peut occuper plus d'espace en mémoire que la somme des tailles des membres de ses composants. Le compilateur peut ajouter un remplissage entre les membres ou à la fin de la structure, mais pas au début.

L'emballage remplace le remplissage par défaut.

Remarques

Eric Raymond a un article sur The Lost Art de C Structure Packing qui est une lecture utile.

Examples

Structures d'emballage

Par défaut, les structures sont complétées en C. Si vous souhaitez éviter ce comportement, vous devez le demander explicitement. Sous GCC, c'est `__attribute__((__packed__))` . Considérez cet exemple sur une machine 64 bits:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

La structure sera automatiquement remplie pour avoir `8-byte` alignement de `8-byte` et ressemblera à ceci:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

Donc, `sizeof(struct foo)` nous donnera `24` au lieu de `17` . Cela s'est produit à cause d'un

compilateur 64 bits en lecture / écriture depuis / vers la mémoire dans 8 octets de mots à chaque étape et évident lorsque vous essayez d'écrire un caractère `char c;` un octet en mémoire, un octet complet (c'est-à-dire un mot) récupéré et consomme uniquement son premier octet et ses sept octets successifs reste vide et inaccessible pour toute opération de lecture et d'écriture pour le remplissage de la structure.

Emballage de structure

Mais si vous ajoutez l'attribut `packed` , le compilateur n'ajoutera pas de remplissage:

```
struct __attribute__((__packed__)) foo {
    char *p;   /* 8 bytes */
    char c;    /* 1 byte  */
    long x;    /* 8 bytes */
};
```

Maintenant, `sizeof(struct foo)` retournera `17` .

Les structures généralement emballées sont utilisées:

- Pour économiser de l'espace
- Formater une structure de données pour la transmettre sur le réseau sans dépendre de chaque alignement d'architecture de chaque nœud du réseau.

Il faut tenir compte du fait que certains processeurs, tels que ARM Cortex-M0, ne permettent pas un accès mémoire non aligné; dans de tels cas, le compactage de la structure peut entraîner *un comportement indéfini* et provoquer *un* blocage du processeur.

Rembourrage de la structure

Supposons que cette `struct` soit définie et compilée avec un compilateur 32 bits:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

On pourrait s'attendre à ce que cette `struct` n'occupe que 10 octets de mémoire, mais en imprimant `sizeof(str_32)` nous voyons qu'il utilise 12 octets.

Cela s'est produit car le compilateur aligne les variables pour un accès rapide. Un modèle courant est que lorsque le type de base occupe N octets (où N est une puissance de 2 telle que 1, 2, 4, 8, 16 - et rarement plus grande), la variable doit être alignée sur une limite de N octets (un multiple de N octets).

Pour la structure affichée avec `sizeof(int) == 4` et `sizeof(short) == 2` , une disposition commune est:

- `int a;` stocké au décalage 0; taille 4
- `short b;` stocké au décalage 4; taille 2
- remplissage sans nom à l'offset 6; taille 2
- `int c;` stocké au décalage 8; taille 4

Ainsi, `struct test_32` occupe 12 octets de mémoire. Dans cet exemple, il n'y a pas de remplissage de fin.

Le compilateur s'assurera que toutes les variables `struct test_32` sont stockées à partir d'une limite de 4 octets, afin que les membres de la structure soient correctement alignés pour un accès rapide. Les fonctions d'allocation de mémoire telles que `malloc()` , `calloc()` et `realloc()` sont nécessaires pour garantir que le pointeur renvoyé est suffisamment bien aligné pour être utilisé avec n'importe quel type de données.

Vous pouvez vous retrouver avec des situations bizarres comme sur un processeur Intel x86_64 64 bits (par exemple Intel Core i7 - un Mac sous MacOS Sierra ou Mac OS X), où lors de la compilation en mode 32 bits, les compilateurs placent un `double` alignement sur un Limite de 4 octets; mais, sur le même matériel, lors de la compilation en mode 64 bits, les compilateurs placent le `double` aligné sur une limite de 8 octets.

Lire Structure rembourrage et emballage en ligne: https://riptutorial.com/fr/c/topic/4590/structure-rembourrage-et-emballage

Chapitre 55: Tableaux

Introduction

Les tableaux sont des types de données dérivés, représentant une collection ordonnée de valeurs ("éléments") d'un autre type. La plupart des tableaux de C ont un nombre fixe d'éléments d'un type quelconque, et leur représentation stocke les éléments de manière contiguë dans la mémoire sans espaces ni remplissage. C permet des tableaux multidimensionnels dont les éléments sont d'autres tableaux, ainsi que des tableaux de pointeurs.

C prend en charge les tableaux alloués dynamiquement dont la taille est déterminée au moment de l'exécution. C99 et versions ultérieures prennent en charge les tableaux ou les VLA de longueur variable.

Syntaxe

- tapez nom [longueur]; / * Définit le tableau de 'type' avec le nom 'name' et la longueur 'length'. * /
- int arr [10] = {0}; / * Définit un tableau et initialise TOUS les éléments à 0. * /
- int arr [10] = {42}; / * Définit un tableau et initialise les 1er éléments à 42 et le reste à 0. * /
- int arr [] = {4, 2, 3, 1}; / * Définit et initialise un tableau de longueur 4. * /
- arr [n] = valeur; / * Définir la valeur à l'index n. * /
- valeur = arr [n]; / * Récupère la valeur à l'index n. * /

Remarques

Pourquoi avons-nous besoin de tableaux?

Les tableaux permettent d'organiser les objets en un agrégat ayant sa propre signification. Par exemple, les chaînes C sont des tableaux de caractères (`char` s) et une chaîne telle que "Hello, World!" a la signification d'un agrégat qui n'est pas inhérent aux caractères individuellement. De même, les tableaux sont couramment utilisés pour représenter des vecteurs et des matrices mathématiques, ainsi que des listes de nombreux types. De plus, sans moyen de regrouper les éléments, il faudrait s’adresser individuellement, par exemple via des variables distinctes. Non seulement cette opération est compliquée, mais elle ne permet pas d’acquérir facilement des collections de différentes longueurs.

Les tableaux sont implicitement convertis en pointeurs dans la plupart des contextes .

Sauf en tant qu'opérande de l'opérateur `sizeof` opérateur `_Alignof` (C2011) ou opérateur unaire `&` (adresse-de), ou littéral de chaîne utilisé pour initialiser un (autre) tableau, un tableau est implicitement converti en ("decays to") un pointeur sur son premier élément. Cette conversion implicite est étroitement liée à la définition de l'opérateur d'indexation du tableau (`[]`): l'expression `arr[idx]` est définie comme étant équivalente à `*(arr + idx)` . De plus, l'arithmétique du pointeur étant commutative, `*(arr + idx)` est également équivalent à `*(idx + arr)` , qui à son

tour équivaut à `idx[arr]` . Toutes ces expressions sont valides et évaluent à la même valeur, à condition que `idx` ou `arr` soit un pointeur (ou un tableau qui se désintègre en un pointeur), l'autre est un entier et l'entier est un index valide dans le tableau vers lequel pointe le pointeur

Comme cas particulier, observez que `&(arr[0])` est équivalent à `&*(arr + 0)` , ce qui simplifie l' `arr` . Toutes ces expressions sont interchangeables partout où le dernier se désintègre en un pointeur. Cela exprime simplement qu'un tableau se désintègre en un pointeur vers son premier élément.

En revanche, si l'adresse de l'opérateur est appliquée à un tableau de type `T[N]` (*ie* `&arr`), le résultat a le type `T (*)[N]` et pointe sur l'ensemble du tableau. Ceci est différent d'un pointeur vers le premier élément du tableau au moins en ce qui concerne l'arithmétique du pointeur, qui est défini en termes de taille du type pointé.

Les paramètres de fonction ne sont pas des tableaux .

```
void foo(int a[], int n);
void foo(int *a, int n);
```

Bien que la première déclaration de `foo` utilise une syntaxe de type tableau pour le paramètre `a` , cette syntaxe est utilisée pour déclarer un paramètre de fonction déclare ce paramètre comme un *pointeur* sur le type d'élément du tableau. Ainsi, la deuxième signature de `foo()` est sémantiquement identique à la première. Cela correspond à la décroissance des valeurs de tableau en pointeurs où ils apparaissent comme des arguments à un *appel de* fonction, de telle sorte que si une variable et un paramètre de fonction sont déclarés avec le même type de tableau, cette valeur peut être utilisée dans un appel de fonction argument associé au paramètre.

Examples

Déclaration et initialisation d'un tableau

La syntaxe générale pour déclarer un tableau à une dimension est

```
type arrName[size];
```

où `type` peut être un type intégré ou des types définis par l'utilisateur tels que des structures, `arrName` est un identificateur défini par l'utilisateur et la `size` est une constante entière.

Déclarer un tableau (un tableau de 10 variables int dans ce cas) est fait comme ceci:

```
int array[10];
```

il contient maintenant des valeurs indéterminées. Pour vous assurer qu'il contient des valeurs nulles lors de la déclaration, vous pouvez le faire:

```
int array[10] = {0};
```

Les tableaux peuvent aussi avoir des initialiseurs, cet exemple déclare un tableau de 10 `int` , où

les 3 premiers `int` contiendront les valeurs `1` , `2` , `3` , toutes les autres valeurs seront nulles:

```
int array[10] = {1, 2, 3};
```

Dans la méthode d'initialisation ci-dessus, la première valeur de la liste sera affectée au premier membre du tableau, la deuxième valeur sera affectée au second membre du tableau, etc. Si la taille de la liste est inférieure à la taille du tableau, alors, comme dans l'exemple ci-dessus, les membres restants du tableau seront initialisés à zéro. Avec l'initialisation de la liste désignée (ISO C99), une initialisation explicite des membres du groupe est possible. Par exemple,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

Dans la plupart des cas, le compilateur peut déduire la longueur du tableau pour vous, cela peut être réalisé en laissant les crochets vides:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

Déclarer un tableau de longueur nulle n'est pas autorisé.

C99 C11

Des tableaux de longueur variable (VLA en abrégé) ont été ajoutés en C99 et rendus facultatifs dans C11. Ils sont égaux aux tableaux normaux, avec une différence importante: la longueur ne doit pas nécessairement être connue au moment de la compilation. Les VLA ont une durée de stockage automatique. Seuls les pointeurs vers les VLA peuvent avoir une durée de stockage statique.

```
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m];               /* create array with calculated length */
```

Important:

Les VLA sont potentiellement dangereux. Si le tableau `vla` dans l'exemple ci-dessus nécessite plus d'espace sur la pile que disponible, la pile débordera. L'utilisation des VLA est donc souvent déconseillée dans les guides de style, les livres et les exercices.

Effacer le contenu du tableau (mise à zéro)

Parfois, il est nécessaire de définir un tableau à zéro, une fois l'initialisation terminée.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
  int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */
```

```
  size_t i;
  for(i = 0; i < ARRLEN; ++i)
  {
    array[i] = 0;
  }

  return EXIT_SUCCESS;
}
```

Un raccourci courant vers la boucle ci-dessus consiste à utiliser `memset()` partir de `<string.h>` . Passer `array` comme indiqué ci-dessous le fait passer à un pointeur sur son premier élément.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

ou

```
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

Comme dans cet exemple, le `array` *est* un tableau et pas seulement un pointeur sur le premier élément d'un tableau (voir la section Longueur du tableau sur l'importance de cet élément). Une troisième option permettant de désactiver le tableau est possible:

```
 memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Longueur du tableau

Les tableaux ont des longueurs fixes connues dans le cadre de leurs déclarations. Néanmoins, il est possible et parfois pratique de calculer des longueurs de tableau. En particulier, cela peut rendre le code plus flexible lorsque la longueur du tableau est déterminée automatiquement à partir d'un initialiseur:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);
```

Cependant, dans la plupart des contextes où un tableau apparaît dans une expression, il est automatiquement converti en ("se désintègre en") en un pointeur vers son premier élément. Le cas où un tableau est l'opérande de l'opérateur `sizeof` est l'un des quelques rares exceptions. Le pointeur résultant n'est pas lui-même un tableau et ne contient aucune information sur la longueur du tableau à partir duquel il a été dérivé. Par conséquent, si cette longueur est nécessaire avec le pointeur, par exemple lorsque le pointeur est passé à une fonction, il doit être acheminé séparément.

Par exemple, supposons que nous voulions écrire une fonction pour retourner le dernier élément d’un tableau de type `int` . En continuant de ce qui précède, nous pourrions l'appeler ainsi:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

La fonction pourrait être implémentée comme ceci:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Notons en particulier que, bien que la déclaration de paramètre `input` ressemble à celle d'un tableau, **il déclare en fait une `input` tant que pointeur** (vers `int`). Cela équivaut exactement à déclarer `input` comme `int *input` . La même chose serait vraie même si une dimension était donnée. Cela est possible parce que les tableaux ne peuvent jamais être des arguments réels pour les fonctions (ils se désintègrent en pointeurs lorsqu'ils apparaissent dans les expressions d'appel de fonction) et peuvent être considérés comme mnémoniques.

C'est une erreur très courante d'essayer de déterminer la taille d'un tableau à partir d'un pointeur, qui ne peut pas fonctionner. NE FAITES PAS CELA:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]));

    return input[length - 1];  /* Oops -- not the droid we are looking for */
}
```

En fait, cette erreur particulière est si courante que certains compilateurs le reconnaissent et l’avertissent. `clang` , par exemple, émettra l'avertissement suivant:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-
Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
                           ^
note: declared here
int BAD_get_last(int input[])
                     ^
```

Définition de valeurs dans les tableaux

L'accès aux valeurs de tableau se fait généralement entre crochets:

```
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

Comme effet secondaire des opérandes à l'opérateur + étant échangeable (-> loi commutative), ce qui suit est équivalent:

```
*(array + 4) = 5;
*(4 + array) = 5;
```

ainsi les prochaines déclarations sont équivalentes:

```
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

et ces deux aussi:

```
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C n'effectue aucune vérification des limites, l'accès au contenu en dehors du tableau déclaré n'est pas défini (Accès à la mémoire au-delà du bloc alloué):

```
int val;
int array[10];

array[4] = 5;    /* ok */
val = array[4];  /* ok */
array[19] = 20;  /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Définir un tableau et un élément de tableau d'accès

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{

   int n[ ARRLEN ]; /* n is an array of 10 integers */
   size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to

                   be wide enough to address all of the possible available memory.
                   Using signed integers to do so should be considered a special use case,
                   and should be restricted to the uncommon case of being in the need of
                   negative indexes. */

   /* Initialize elements of array n. */
   for ( i = 0; i < ARRLEN ; i++ )
   {
      n[ i ] = i + 100; /* Set element at location i to i + 100. */
   }

   /* Output each array element's value. */
   for (j = 0; j < ARRLEN ; j++ )
   {
      printf("Element[%zu] = %d\n", j, n[j] );
```

```
    }

    return 0;
}
```

Allouer et initialiser à zéro un tableau avec une taille définie par l'utilisateur

```
#include <stdio.h>
#include <stdlib.h>


int main (void)
{
  int * pdata;
  size_t n;

  printf ("Enter the size of the array: ");
  fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

  if (1 != scanf("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
  {
    fprintf("scanf() did not read a in proper value.\n");
    exit(EXIT_FAILURE);
  }

  pdata = calloc(n, sizeof *pdata);
  if (NULL == pdata)
  {
    perror("calloc() failed"); /* Print error. */
    exit(EXIT_FAILURE);
  }

  free(pdata); /* Clean up. */

  return EXIT_SUCCESS;
}
```

Ce programme essaie d'analyser une valeur entière non signée à partir de l'entrée standard, alloue un bloc de mémoire pour un tableau de `n` éléments de type `int` en appelant la fonction `calloc()` . La mémoire est initialisée à tous les zéros par ce dernier.

En cas de succès, la mémoire est `free()` par l'appel à `free()` .

Itérer à travers un tableau efficacement et ordre de rangée

Les tableaux en C peuvent être considérés comme un bloc de mémoire contigu. Plus précisément, la dernière dimension du tableau est la partie contiguë. Nous appelons cela l'ordre des *rangées majeures* . Comprendre cela et le fait qu'un défaut de cache charge une ligne de cache complète dans le cache lors de l' accès aux données non mises en cache pour éviter les défauts de cache suivantes, nous pouvons voir pourquoi l' accès à un tableau de dimension 10000x10000 avec `array[0][0]` serait **potentiellement** charge `array[0][1]` dans le cache, mais accéder au `array[1][0]` juste après génèrerait un second défaut de cache, car il est `sizeof(type)*10000` octets du `array[0][0]` , et donc certainement pas sur la même ligne de cache. C'est pourquoi l'itération comme celle-ci est inefficace:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

Et itérer comme ça est plus efficace:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

Dans le même ordre d'idées, c'est la raison pour laquelle, lorsqu'il s'agit d'un tableau à une dimension et de plusieurs index (disons 2 dimensions ici pour la simplicité avec les index i et j), il est important de parcourir le tableau comme ceci:

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}
```

Ou avec 3 dimensions et index i, j et k:

```
#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
```

```
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}
```

Ou de manière plus générique, lorsque nous avons un tableau avec **N1 x N2 x ... x Nd** éléments, **d** dimensions et indices notés **n1, n2, ..., et** le décalage est calculé comme ceci

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1)\cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

Image / formule extraite de: https://en.wikipedia.org/wiki/Row-major_order

Tableaux multidimensionnels

Le langage de programmation C permet des tableaux multidimensionnels . Voici la forme générale d'une déclaration de tableau multidimensionnel -

```
type name[size1][size2]...[sizeN];
```

Par exemple, la déclaration suivante crée un tableau d'entiers tridimensionnel (5 x 10 x 4):

```
int arr[5][10][4];
```

Tableaux bidimensionnels

La forme la plus simple de tableau multidimensionnel est le tableau à deux dimensions. Un tableau à deux dimensions est essentiellement une liste de tableaux à une dimension. Pour déclarer un tableau entier à deux dimensions de dimensions mxn, nous pouvons écrire comme suit:

```
type arrayName[m][n];
```

Où `type` peut être n'importe quel type de données C valide (`int` , `float` , etc.) et `arrayName` peut être n'importe quel identifiant C valide. Un tableau à deux dimensions peut être visualisé sous forme de tableau avec `m` lignes et `n` colonnes. **Note** : L'ordre *est* important en C. Le tableau `int a[4][3]` n'est pas le même que le tableau `int a[3][4]` . Le nombre de lignes vient en premier car C est un langage de *rang* supérieur.

Un tableau à deux dimensions `a` , qui contient trois lignes et quatre colonnes peut être affiché comme suit:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Ainsi, chaque élément du tableau `a` est identifié par un nom d'élément de la forme `a[i][j]` , où `a` est le nom du tableau, `i` représente quelle ligne et `j` représente quelle colonne. Rappelez-vous que les lignes et les colonnes sont indexées à zéro. Ceci est très similaire à la notation mathématique pour les matrices 2D.

Initialisation de tableaux bidimensionnels

Les tableaux multidimensionnels peuvent être initialisés en spécifiant des valeurs entre parenthèses pour chaque ligne. Ce qui suit définit un tableau avec 3 lignes où chaque ligne a 4 colonnes.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Les accolades imbriquées, qui indiquent la ligne voulue, sont facultatives. L'initialisation suivante est équivalente à l'exemple précédent:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Bien que la méthode de création de tableaux avec des accolades imbriquées soit facultative, elle est fortement recommandée car elle est plus lisible et plus claire.

Accès à des éléments de tableau à deux dimensions

On accède à un élément dans un tableau à deux dimensions en utilisant les indices, à savoir l’index de ligne et l’index de colonne du tableau. Par exemple -

```
int val = a[2][3];
```

L'instruction ci-dessus prendra le 4ème élément de la 3ème ligne du tableau. Laissez-nous vérifier le programme suivant où nous avons utilisé une boucle imbriquée pour gérer un tableau à deux dimensions:

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;
```

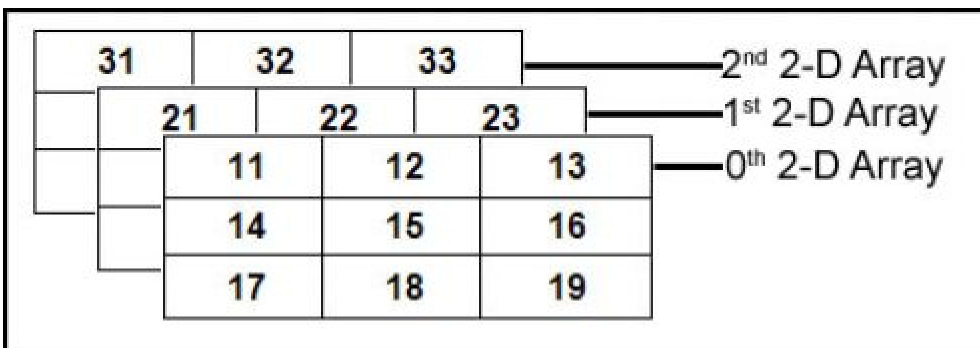
```
   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {

      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Lorsque le code ci-dessus est compilé et exécuté, il produit le résultat suivant:

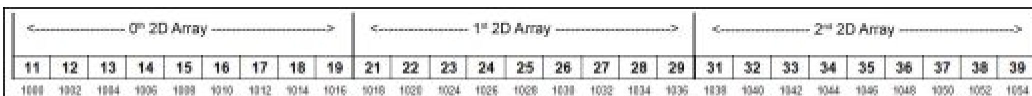
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Tableau tridimensionnel:

Un tableau 3D est essentiellement un tableau de tableaux de tableaux: il s’agit d’un tableau ou d’une collection de tableaux 2D, et un tableau 2D est un tableau de tableaux 1D.



Carte mémoire 3D:



Initialisation d'un tableau 3D:

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

Nous pouvons avoir des tableaux avec un nombre quelconque de dimensions, bien qu'il soit

probable que la plupart des tableaux créés auront une ou deux dimensions.

Itérer à travers un tableau en utilisant des pointeurs

```
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

Ici, dans l'initialisation de `p` dans la première `for` condition de la boucle, le tableau d' `a` *désintègre* à un pointeur sur son premier élément, car elle dans presque tous les endroits où une telle variable tableau est utilisé.

Ensuite, le `++p` effectue l'arithmétique du pointeur sur le pointeur `p` et parcourt un par un les éléments du tableau, en les référençant par `*p` .

Passer des tableaux multidimensionnels à une fonction

Les tableaux multidimensionnels suivent les mêmes règles que les tableaux à une dimension lors de leur transmission à une fonction. Cependant, la combinaison de decay-to-pointer, de priorité des opérateurs et des deux manières différentes de déclarer un tableau multidimensionnel (tableau de tableaux vs tableau de pointeurs) peut rendre la déclaration de ces fonctions non intuitive. L'exemple suivant montre comment transmettre correctement des tableaux multidimensionnels.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}
```

```
/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
   *(x[4]), i.e. an array of 4 pointers to int.  But if it's declared as a
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

Voir également

Passer des tableaux à des fonctions

Lire Tableaux en ligne: https://riptutorial.com/fr/c/topic/322/tableaux

Chapitre 56: Test des frameworks

Introduction

De nombreux développeurs utilisent des tests unitaires pour vérifier que leur logiciel fonctionne comme prévu. Les tests unitaires vérifient les petites unités de logiciels plus gros et vérifient que les résultats correspondent aux attentes. Les frameworks de test facilitent les tests unitaires en fournissant des services de configuration / déconnexion et en coordonnant les tests.

Il existe de nombreux frameworks de tests unitaires pour C. Par exemple, Unity est un framework C pur. Les gens utilisent assez souvent les frameworks de test C ++ pour tester le code C; Il existe également de nombreux frameworks de test C ++.

Remarques

Harnais de test:

TDD - Développement piloté par les tests:

Tester les mécanismes doubles en C:

1. Substitution de temps de liaison
2. Substitution de pointeur de fonction
3. Substitution de préprocesseur
4. Combinaison du temps de liaison et du pointeur de fonction

Remarque sur les frameworks de test C ++ utilisés dans C: L'utilisation de frameworks C ++ pour tester un programme C est une pratique assez courante, comme expliqué ici .

Examples

CppUTest

CppUTest est un framework de type xUnit pour les tests unitaires C et C ++. Il est écrit en C ++ et vise la portabilité et la simplicité dans la conception. Il prend en charge la détection des fuites de mémoire, la création de simulacres et l'exécution de ses tests avec le test Google. Livré avec des scripts d'aide et des exemples de projets pour Visual Studio et Eclipse CDT.

```
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>


TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* Test runner may be provided options, such
```

```
   as to enable colored output, to run only a
   specific test or a group of tests, etc. This
   will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}
```

Un groupe de test peut avoir une méthode `setup()` et une méthode `teardown()` . La méthode d' `setup` est appelée avant chaque test et la méthode `teardown()` est appelée après. Les deux sont facultatifs et l'un ou l'autre peut être omis indépendamment. D'autres méthodes et variables peuvent également être déclarées à l'intérieur d'un groupe et seront disponibles pour tous les tests de ce groupe.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

Cadre de test d'unité

Unity est un framework de test de style xUnit pour les tests unitaires C. Il est entièrement écrit en C et est portable, rapide, simple, expressif et extensible. Il est spécialement conçu pour les tests unitaires des systèmes embarqués.

Un scénario de test simple qui vérifie la valeur de retour d'une fonction peut se présenter comme suit

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

Un fichier de test complet peut ressembler à ceci:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */
```

```
void setUp (void) {} /* Is run before every test, put unit init calls here. */
void tearDown (void) {} /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity est fourni avec des exemples de projets, des fichiers makefiles et des scripts rake Ruby qui facilitent la création de fichiers de test plus longs.

CMocka

CMocka est un framework de test unitaire élégant pour C avec prise en charge des objets fictifs . Il ne nécessite que la bibliothèque standard C, fonctionne sur une gamme de plates-formes informatiques (y compris embarquées) et avec différents compilateurs. Il comprend un didacticiel sur les tests avec des simulacres, la documentation de l'API et divers exemples .

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}

void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}


int main (void)
{
    const struct CMUnitTest tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
```

```
    };

    /* If setup and teardown functions are not
       needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}
```

Lire Test des frameworks en ligne: https://riptutorial.com/fr/c/topic/6779/test-des-frameworks

Chapitre 57: Threads (natifs)

Syntaxe

- `#ifndef __STDC_NO_THREADS__`
- `# include <threads.h>`
- `#endif`
- `void call_once(once_flag *flag, void (*func)(void));`
- `int cnd_broadcast(cnd_t *cond);`
- `void cnd_destroy(cnd_t *cond);`
- `int cnd_init(cnd_t *cond);`
- `int cnd_signal(cnd_t *cond);`
- `int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int cnd_wait(cnd_t *cond, mtx_t *mtx);`
- `void mtx_destroy(mtx_t *mtx);`
- `int mtx_init(mtx_t *mtx, int type);`
- `int mtx_lock(mtx_t *mtx);`
- `int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int mtx_trylock(mtx_t *mtx);`
- `int mtx_unlock(mtx_t *mtx);`
- `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`
- `thrd_t thrd_current(void);`
- `int thrd_detach(thrd_t thr);`
- `int thrd_equal(thrd_t thr0, thrd_t thr1);`
- `_Noreturn void thrd_exit(int res);`
- `int thrd_join(thrd_t thr, int *res);`
- `int thrd_sleep(const struct timespec *duration, struct timespec* remaining);`
- `void thrd_yield(void);`
- `int tss_create(tss_t *key, tss_dtor_t dtor);`
- `void tss_delete(tss_t key);`
- `void *tss_get(tss_t key);`
- `int tss_set(tss_t key, void *val);`

Remarques

Les threads C11 sont une fonctionnalité facultative. Leur absence peut être testée avec `__STDC__NO_THREAD__` . Actuellement (juillet 2016), cette fonctionnalité n'est pas encore implémentée par toutes les bibliothèques C qui prennent en charge C11.

Les bibliothèques C connues pour prendre en charge les threads C11 sont les suivantes:

- musl

C bibliothèques qui ne supportent pas les threads C11, pourtant:

- gnu libc

Examples

Commencer plusieurs discussions

```
#include <stdio.h>
#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
   double factor;
};

int my_thread_func(void* a) {
   struct my_thread_data* d = a;
   // do something with d
   printf("we found %g\n", d->factor);
   // return an success or error code
   return d->factor > 1.0;
}


int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // reserve space for the arguments for the threads
    struct my_thread_data D[n];     // can't be initialized
    for (unsigned i = 0; i < n; ++i) {
         D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // reserve space for the ID's of the threads
    thrd_t id[4];
    // launch the threads
    for (unsigned i = 0; i < n; ++i) {
         thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // Wait that all threads have finished, but throw away their
    // return values
    for (unsigned i = 0; i < n; ++i) {
         thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}
```

Initialisation par un thread

Dans la plupart des cas, toutes les données accessibles par plusieurs threads doivent être initialisées avant la création des threads. Cela garantit que tous les threads commencent avec un état clair et qu'aucune *condition de* concurrence ne *se* produit.

Si cela n'est pas possible, `once_flag` et `call_once` peuvent être utilisés

```
#include <threads.h>
#include <stdlib.h>
```

```
// the user data for this example
double const* Big = 0;

// the flag to protect big, must be global and/or static
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
   free((void*)Big);
}

void initBig(void) {
    // assign to temporary with no const qualification
    double* b = malloc(largeNum);
    if (!b) {
       perror("allocation failed for Big");
       exit(EXIT_FAILURE);
    }
    // now initialize and store Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // ensure that the space is freed on exit or quick_exit
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// the user thread function that relies on Big
int myThreadFunc(void* a) {
   call_once(&onceBig, initBig);
   // only use Big from here on
   ...
   return 0;
}
```

`once_flag` est utilisé pour coordonner différents threads qui pourraient vouloir initialiser les mêmes données `Big` . L'appel à `call_once` garantit que

- `initBig` est appelé exactement une fois
- `call_once` bloque jusqu'à ce qu'un tel appel à `initBig` ait été effectué, soit par le même thread, soit par un autre.

Outre l'allocation, une fonction typique d'une fonction appelée une fois est une initialisation dynamique d'une structure de données de contrôle de thread telle que `mtx_t` ou `cnd_t` qui ne peuvent pas être initialisées statiquement, en utilisant `mtx_init` ou `cnd_init` , respectivement.

Lire Threads (natifs) en ligne: https://riptutorial.com/fr/c/topic/4432/threads--natifs-

Chapitre 58: Traitement du signal

Syntaxe

- void (* signal (int sig, void (* func) (int))) (int);

Paramètres

Paramètre	Détails
SIG	Le signal pour mettre le gestionnaire de signal à l'un des `SIGABRT` , `SIGFPE` , `SIGILL` , `SIGTERM` , `SIGINT` , `SIGSEGV` ou à une valeur d'implémentation définie
func	Le gestionnaire de signal, qui est l'un des suivants: `SIG_DFL` , pour le gestionnaire par défaut, `SIG_IGN` pour ignorer le signal ou un pointeur de fonction avec la signature `void foo(int sig);` .

Remarques

L'utilisation de gestionnaires de signaux avec uniquement les garanties du standard C impose diverses limitations à ce qui peut ou ne peut pas être fait dans le gestionnaire de signaux défini par l'utilisateur.

- Si la fonction définie par l'utilisateur est `SIGFPE` lors de la gestion de `SIGSEGV` , `SIGFPE` , `SIGILL` ou de toute autre interruption matérielle définie par l'implémentation, le comportement n'est pas défini par le standard C. C'est parce que l'interface de C ne donne pas les moyens de changer l'état défectueux (par exemple après une division par `0`) et donc, lors du retour du programme, il se trouve exactement dans le même état qu'avant l'interruption matérielle.

- Si la fonction définie par l'utilisateur a été appelée à la suite d'un appel d' `abort` ou de `raise` , le gestionnaire de signaux n'est pas autorisé à appeler à nouveau une `raise` .

- Les signaux peuvent arriver au milieu de toute opération et, par conséquent, l'indivisibilité des opérations ne peut généralement pas être garantie et la gestion du signal ne fonctionne pas bien avec l'optimisation. Par conséquent, toutes les modifications apportées aux données dans un gestionnaire de signaux doivent être des variables.

  - de type `sig_atomic_t` (toutes versions) ou un type atomique sans verrou (depuis C11, optionnel)
  - qui sont qualifiés de `volatile` .

- Les autres fonctions de la bibliothèque standard C ne respecteront généralement pas ces restrictions, car elles peuvent changer les variables dans l'état global du programme. La norme C ne fait que des garanties pour `abort` , `_Exit` (depuis C99), `quick_exit` (depuis C11), le `signal` (pour le même nombre de signaux), et certaines opérations atomiques (depuis

C11).

Le comportement n'est pas défini par la norme C si l'une des règles ci-dessus est violée. Les plates-formes peuvent avoir des extensions spécifiques, mais celles-ci ne sont généralement pas portables au-delà de cette plate-forme.

- Généralement, les systèmes ont leur propre liste de fonctions qui sont *des signaux asynchrones sûrs* , c'est-à-dire des fonctions de la bibliothèque C pouvant être utilisées par un gestionnaire de signaux. Par exemple, souvent, `printf` fait partie de ces fonctions.
- En particulier, le standard C ne définit pas beaucoup l'interaction avec son interface de threads (depuis C11) ou toute autre bibliothèque de thread spécifique à la plate-forme telle que les threads POSIX. De telles plates-formes doivent spécifier l'interaction de telles bibliothèques de threads avec des signaux eux-mêmes.

Examples

Traitement du signal avec “signal ()”

Les numéros de signal peuvent être synchrones (comme `SIGSEGV` - erreur de segmentation) lorsqu'ils sont déclenchés par un dysfonctionnement du programme lui-même ou asynchrone (comme l'attention interactive `SIGINT`) lorsqu'ils sont initiés à l'extérieur du programme, par exemple par une pression sur `Cntrl-C` .

La fonction `signal()` fait partie de la norme ISO C et peut être utilisée pour affecter une fonction à un signal spécifique

```
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
    /* hardware interrupts should not return */
    case SIGSEGV:
    case SIGFPE:
    case SIGILL:
```

C11

```
        /* quick_exit is safe to call */
        quick_exit(EXIT_FAILURE);
```

C11

```
        /* use _Exit in pre-C11 */
        _Exit(EXIT_FAILURE);
```

```
    default:
       /* Reset the signal to the default handler,
          so we will not be called again if things go
          wrong on return. */
      signal(sig, SIG_DFL);
      /* let everybody know that we are finished */
      finished = sig;
      return;
   }
}

int main(void)
{

    /* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access */
    if (signal(SIGSEGV, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGSEGV");
        return EXIT_FAILURE;
    }

    /* Catch the SIGTERM signal, termination request */
    if (signal(SIGTERM, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGTERM");
        return EXIT_FAILURE;
    }

    /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
    signal(SIGINT, SIG_IGN);


    /* Do something that takes some time here, and leaves
       the time to terminate the program from the keyboard. */

    /* Then: */

    if (finished) {
       fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
        return EXIT_FAILURE;
    }


    /* Try to force a segmentation fault, and raise a SIGSEGV */
    {
      char* ptr = 0;
      *ptr = 0;
    }

    /* This should never be executed */
    return EXIT_SUCCESS;
}
```

L'utilisation de `signal()` impose des limitations importantes à ce que vous êtes autorisé à faire dans les gestionnaires de signaux, voir les remarques pour plus d'informations.

POSIX recommande l'utilisation de `sigaction()` au lieu de `signal()` , en raison de son comportement sous-spécifié et des variations importantes de sa mise en œuvre. POSIX définit également beaucoup plus de signaux que la norme ISO C, y compris `SIGUSR1` et `SIGUSR2` , qui peuvent être utilisés librement par le programmeur à toutes fins.

Lire Traitement du signal en ligne: https://riptutorial.com/fr/c/topic/453/traitement-du-signal

Chapitre 59: Type d'aliasing et efficace

Remarques

Les violations des règles d'alias et la violation du type effectif d'un objet sont deux choses différentes et ne doivent pas être confondues.

- *L'aliasing* est la propriété de deux pointeurs `a` et `b` qui se réfèrent au même objet, à savoir que `a == b` .
- Le *type effectif* d'un objet de données est utilisé par C pour déterminer quelles opérations peuvent être effectuées sur cet objet. En particulier, le type effectif est utilisé pour déterminer si deux pointeurs peuvent s'appeler mutuellement.

La création d'alias peut être un problème pour l'optimisation, car la modification de l'objet par le biais d'un pointeur, `a` exemple, peut modifier l'objet visible via l'autre pointeur, `b` . Si votre compilateur C devait supposer que les pointeurs pouvaient toujours s'allier, quel que soit leur type et leur provenance, de nombreuses possibilités d'optimisation seraient perdues et de nombreux programmes seraient plus lents.

Les règles d'aliasing strictes de C font référence aux cas dans lesquels le compilateur *peut supposer* quels objets font (ou non) des pseudonymes. Il y a deux règles à suivre pour les pointeurs de données.

> Sauf indication contraire, deux pointeurs avec le même type de base peuvent être alias.

> Deux pointeurs avec un type de base différent ne peuvent pas créer d'alias, à moins qu'au moins l'un des deux types soit un type de caractère.

Voici le *type de base* signifie que nous mettons de côté des qualifications de type tels que `const` , par exemple , si `a` est `double*` et `b` est `const double*` , le compilateur *doit* généralement présumer qu'un changement de `*a` peut changer `*b` .

Violer la deuxième règle peut avoir des résultats catastrophiques. Ici, violer la règle d'aliasing strict signifie que vous présentez deux pointeurs `a` et `b` de type différent au compilateur qui, en réalité, pointe vers le même objet. Le compilateur peut alors toujours supposer que les deux pointent vers des objets différents, et ne mettra pas à jour son idée de `*b` si vous avez modifié l'objet via `*a` .

Si vous le faites, le comportement de votre programme devient indéfini. Par conséquent, C impose des restrictions assez sévères sur les conversions de pointeurs afin de vous aider à éviter une telle situation.

> À moins que le type de source ou de cible ne soit `void` , toutes les conversions de pointeurs entre pointeurs de type base différent doivent être *explicites* .

Ou en d' autres termes, ils ont besoin d' un *casting,* à moins que vous faites une conversion qui

ajoute juste un qualificatif tel que `const` le type de cible.

Éviter les conversions de pointeurs en général et les conversions en particulier vous protège contre les problèmes d'alias. À moins que vous en ayez vraiment besoin et que ces cas soient très particuliers, vous devriez les éviter comme vous le pouvez.

Examples

Les types de caractères ne sont pas accessibles via des types autres que des caractères.

Si un objet est défini avec une durée statique, de thread ou de stockage automatique et qu'il a un type de caractère, soit: `char` , `unsigned char` , ou `signed char` , il ne sera pas accessible par un type non-caractère. Dans l'exemple ci - dessous un `char` tableau est réinterprété comme le type `int` , et le comportement est indéfini sur chaque déréférencement du `int` pointeur `b` .

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

Ceci n'est pas défini car il viole la règle du "type effectif", aucun objet de données ayant un type efficace ne peut être accédé via un autre type qui n'est pas un type de caractère. Étant donné que l'autre type ici est `int` , ce n'est pas autorisé.

Même si l'alignement et la taille des pointeurs étaient connus pour s'adapter, cela ne dispenserait pas de cette règle, le comportement serait toujours indéfini.

Cela signifie en particulier qu’il n’existe aucun moyen de réserver un objet tampon de type caractère pouvant être utilisé avec des pointeurs de différents types, car vous utiliseriez un tampon reçu par `malloc` ou une fonction similaire.

Une manière correcte d'atteindre le même objectif que dans l'exemple ci-dessus serait d'utiliser une `union` .

```
typedef union bufType bufType;
union bufType {
   char c[sizeof(int[25])];
   int i[25];
};
```

```
int main( void )
{
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;      // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Dans ce cas, l' `union` garantit que le compilateur sait dès le départ que le tampon peut être accédé via différentes vues. Cela a également l'avantage que le tampon a maintenant une "vue" `ai` qui est déjà de type `int` et qu'aucune conversion de pointeur n'est nécessaire.

Type efficace

Le *type effectif* d'un objet de données est la dernière information de type qui lui était associée, le cas échéant.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Observez que pour ce dernier, il n'était pas nécessaire que nous ayons même un pointeur `uint32_t*` vers cet objet. Le fait que nous ayons copié un autre objet `uint32_t` est suffisant.

Violer les règles strictes d'aliasing

Dans le code suivant, supposons pour simplifier que `float` et `uint32_t` aient la même taille.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` et `f` ont un type de base différent, et le compilateur peut donc supposer qu'ils pointent vers des objets différents. Il n'ya pas de possibilité que `*f` ait changé entre les deux initialisations de `a` et `b` , et le compilateur peut donc optimiser le code en quelque chose d'équivalent à

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

C'est-à-dire que la seconde opération de chargement de `*f` peut être complètement optimisée.

Si nous appelons cette fonction "normalement"

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

tout va bien et quelque chose comme

> 4 devrait être égal à 4

est imprimé. Mais si nous trichons et passons le même pointeur, après la conversion,

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

nous violons la règle stricte d'aliasing. Le comportement devient alors indéfini. La sortie pourrait être comme ci-dessus, si le compilateur avait optimisé le second accès, ou quelque chose de complètement différent, et que votre programme se retrouverait dans un état totalement non fiable.

restreindre la qualification

Si nous avons deux arguments de pointeur du même type, le compilateur ne peut faire aucune

supposition et devra toujours supposer que la modification de `*e` peut changer `*f` :

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}

float fval = 4;
float eval = 77;
 fun(&eval, &fval);
```

tout va bien et quelque chose comme

> est 4 égal à 4?

est imprimé. Si nous passons le même pointeur, le programme fera quand même le bon choix et imprimera

> est 4 égal à 22?

Cela peut s'avérer inefficace si nous *connaissons* par des informations externes que `e` et `f` ne pointeront jamais vers le même objet de données. Nous pouvons refléter cette connaissance en ajoutant `restrict` qualificateurs de `restrict` aux paramètres du pointeur:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Le compilateur peut alors toujours supposer que `e` et `f` pointent vers des objets différents.

Changer d'octets

Une fois qu'un objet a un type efficace, vous ne devriez pas tenter de le modifier par un pointeur d'un autre type, à moins que cet autre type est un type de caractère, `char` , `signed char` ou `unsigned char` .

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  uint32_t a = 57;
  // conversion from incompatible types needs a cast !
  unsigned char* ap = (unsigned char*)&a;
  for (size_t i = 0; i < sizeof a; ++i) {
    /* set each byte of a to 42 */
    ap[i] = 42;
  }
  printf("a now has value %" PRIu32 "\n", a);
}
```

Ceci est un programme valide qui imprime

> a maintenant une valeur 707406378

Cela fonctionne parce que:

- L'accès est fait aux octets individuels vus avec le type `unsigned char` afin que chaque modification soit bien définie.
- Les deux vues de l'objet, à travers `a` alias a et through `*ap` , mais `ap` étant un pointeur sur un type de caractère, la règle d'aliasing stricte ne s'applique pas. Le compilateur doit donc supposer que la valeur de `a` peut avoir été modifiée dans la boucle `for` . La valeur modifiée de `a` doit être construite à partir des octets modifiés.
- Le type de `a` , `uint32_t` n'a pas de bits de remplissage. Tous ses bits de la représentation comptent pour la valeur, ici `707406378` , et il ne peut y avoir aucune représentation de piège.

Lire Type d'aliasing et efficace en ligne: https://riptutorial.com/fr/c/topic/1301/type-d-aliasing-et-efficace

Chapitre 60: Typedef

Introduction

Le mécanisme `typedef` permet la création d'alias pour d'autres types. Il ne crée pas de nouveaux types. Les personnes utilisent souvent `typedef` pour améliorer la portabilité du code, pour donner des alias à des types de structure ou d'union, ou pour créer des alias pour les types de fonctions (ou de pointeurs de fonctions).

Dans la norme C, `typedef` est classé comme une «classe de stockage» par commodité; il se produit de manière syntaxique lorsque des classes de stockage telles que `static` ou `extern` peuvent apparaître.

Syntaxe

- typedef nom_existant nom_alias;

Remarques

Inconvénients de Typedef

`typedef` pourrait conduire à la pollution de l'espace de nommage dans les grands programmes C.

Inconvénients des structures Typedef

De plus, les structures `typedef` sans nom de tag sont une cause majeure de l'imposition inutile de relations d'ordre entre les fichiers d'en-tête.

Considérer:

```
#ifndef FOO_H
#define FOO_H 1

#define FOO_DEF (0xDEADBABE)

struct bar; /* forward declaration, defined in bar.h*/

struct foo {
    struct bar *bar;
};

#endif
```

Avec une telle définition, n'utilisant pas les `typedefs` , il est possible qu'une unité de compilation inclue `foo.h` pour obtenir la définition de `FOO_DEF` . S'il ne tente pas de déréférencer le membre `bar` de la structure `foo` il ne sera pas nécessaire d'inclure le fichier `bar.h`

Typedef vs #define

`#define` est une directive de préprocesseur C qui est également utilisée pour définir les alias pour différents types de données similaires à `typedef` mais avec les différences suivantes:

- `typedef` se limite à donner des noms symboliques aux types uniquement où `#define` peut également être utilisé pour définir des alias pour les valeurs.
- `typedef` interprétation `typedef` est effectuée par le compilateur alors que les instructions `#define` sont traitées par le pré-processeur.
- Notez que `#define cptr char *` suivi de `cptr a, b;` ne fait pas la même chose que `typedef char *cptr;` suivi de `cptr a, b;` . Avec `#define` , `b` est une variable `char` ordinaire, mais c'est aussi un pointeur avec le `typedef` .

Examples

Typedef pour les structures et les syndicats

Vous pouvez donner des noms d'alias à une `struct` :

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

Comparé à la manière traditionnelle de déclarer des structures, les programmeurs n'auraient pas besoin de `struct` chaque fois qu'ils déclarent une instance de cette structure.

Notez que le nom de `Person` (par opposition à `struct Person`) n'est défini qu'au dernier point-virgule. Ainsi, pour les listes liées et les arborescences qui doivent contenir un pointeur vers le même type de structure, vous devez utiliser soit:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

ou:

```
typedef struct Person Person;

struct Person {
    char name[32];
    int age;
    Person *next;
};
```

L'utilisation d'un `typedef` pour un type d' `union` est très similaire.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

Une structure similaire à celle-ci peut être utilisée pour analyser les octets qui constituent une valeur `float` .

Utilisations simples de Typedef

Pour donner des noms courts à un type de données

Au lieu de:

```
long long int foo;
struct mystructure object;
```

on peut utiliser

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

Cela réduit la quantité de frappe nécessaire si le type est utilisé plusieurs fois dans le programme.

Améliorer la portabilité

Les attributs des types de données varient selon les architectures. Par exemple, un `int` peut être un type à 2 octets dans une implémentation et un type à 4 octets dans un autre. Supposons qu'un programme doive utiliser un type de 4 octets pour s'exécuter correctement.

Dans une implémentation, la taille de `int` être de 2 octets et celle de `long` 4 octets. Dans un autre, laissez la taille de `int` être de 4 octets et celle de `long` 8 octets. Si le programme est écrit en utilisant la deuxième implémentation,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

Pour que le programme s'exécute dans la première implémentation, toutes les déclarations `int` devront être modifiées en `long` .

```
/* program now needs long */
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

Pour éviter cela, on peut utiliser `typedef`

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Ensuite, seule l'instruction `typedef` devrait être modifiée à chaque fois, au lieu d'examiner l'ensemble du programme.

C99

L'en-tête `<stdint.h>` et l'en-tête associé `<inttypes.h>` définissent les noms de type standard (en utilisant `typedef`) pour les entiers de différentes tailles. Par exemple, `uint8_t` est un type entier non signé de 8 bits; `int64_t` est un type entier signé de 64 bits. Le type `uintptr_t` est un type entier non signé suffisamment grand pour contenir un pointeur sur un objet. Ces types sont théoriquement facultatifs - mais il est rare qu'ils ne soient pas disponibles. Il existe des variantes comme `uint_least16_t` (le plus petit type entier non signé avec au moins 16 bits) et `int_fast32_t` (le type entier signé le plus rapide avec au moins 32 bits). En outre, `intmax_t` et `uintmax_t` sont les plus grands types d'entiers pris en charge par l'implémentation. Ces types sont obligatoires.

Pour spécifier un usage ou améliorer la lisibilité

Si un ensemble de données a un objectif particulier, on peut utiliser `typedef` pour lui donner un nom significatif. De plus, si la propriété des données change de telle sorte que le type de base doit changer, seule l'instruction `typedef` devra être modifiée, au lieu d'examiner l'ensemble du programme.

Typedef pour les pointeurs de fonction

Nous pouvons utiliser `typedef` pour simplifier l'utilisation des pointeurs de fonctions. Imaginez que nous ayons des fonctions, toutes ayant la même signature, qui utilisent leur argument pour imprimer quelque chose de différentes manières:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n, n);
}
```

Maintenant, nous pouvons utiliser un `typedef` pour créer un type de pointeur de fonction nommé appelé imprimante:

```
typedef void (*printer_t)(int);
```

Cela crée un type, nommé `printer_t` pour un pointeur sur une fonction qui prend un seul argument `int` et ne renvoie rien, ce qui correspond à la signature des fonctions ci-dessus. Pour l'utiliser, nous créons une variable du type créé et lui assignons un pointeur sur l'une des fonctions en question:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Ensuite, appeler la fonction pointée par la variable de pointeur de fonction:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Ainsi, `typedef` permet une syntaxe plus simple lorsqu'il s'agit de pointeurs de fonctions. Cela devient plus évident lorsque des pointeurs de fonction sont utilisés dans des situations plus complexes, telles que des arguments pour des fonctions.

Si vous utilisez une fonction qui prend un pointeur de fonction comme paramètre sans type de pointeur de fonction défini, la définition de fonction serait,

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

Cependant, avec le `typedef` c'est:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

De même, les fonctions peuvent renvoyer des pointeurs de fonction et, à nouveau, l'utilisation d'un `typedef` peut simplifier la syntaxe.

Un exemple classique est la fonction de `signal` de `<signal.h>` . La déclaration pour cela (du standard C) est:

```
void (*signal(int sig, void (*func)(int)))(int);
```

C'est une fonction qui prend deux arguments - un `int` et un pointeur sur une fonction qui prend un `int` comme argument et ne retourne rien - et qui renvoie un pointeur pour fonctionner comme son

deuxième argument.

Si nous avons défini un type `SigCatcher` comme alias pour le pointeur sur le type de fonction:

```
typedef void (*SigCatcher)(int);
```

alors nous pourrions déclarer le `signal()` utilisant:

```
SigCatcher signal(int sig, SigCatcher func);
```

Dans l'ensemble, cela est plus facile à comprendre (même si la norme C n'a pas choisi de définir un type pour effectuer le travail). La fonction `signal` prend deux arguments, un `int` et un `SigCatcher` , et renvoie un `SigCatcher` - où un `SigCatcher` est un pointeur sur une fonction qui prend un argument `int` et ne retourne rien.

Bien que l'utilisation de noms de type `typedef` pour les types de pointeurs vers les fonctions facilite la vie, cela peut également être source de confusion pour les autres utilisateurs qui conserveront votre code ultérieurement. À utiliser avec précaution et avec une documentation appropriée. Voir aussi les pointeurs de fonction .

Lire Typedef en ligne: https://riptutorial.com/fr/c/topic/2681/typedef

Chapitre 61: Types de données

Remarques

- Bien que `char` soit obligatoire pour 1 octet, il n'est **pas** nécessaire que 1 octet soit de 8 bits (souvent appelé également *octet*), même si la plupart des plates-formes informatiques modernes le définissent comme 8 bits. Le nombre de bits par de la mise en œuvre de `char` est fourni par le `CHAR_BIT` macro, défini dans `<limits.h>` . POSIX nécessite 1 octet pour 8 bits.
- Les types entiers à largeur fixe doivent être utilisés de manière éparse, les types intégrés à C sont conçus pour être naturels sur chaque architecture, les types à largeur fixe ne doivent être utilisés que si vous avez explicitement besoin d'un entier spécifique (par exemple pour la mise en réseau).

Examples

Types entiers et constantes

Les entiers signés peuvent être de ces types (l' `int` après `short` ou `long` est facultatif):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */
```

C99

```
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Chacun de ces types d'entiers signés a une version non signée.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

Pour tous les types sauf la `char` la version `signed` est supposée si la partie `signed` ou `unsigned` est omise. Le type `char` constitue un troisième type de caractère, différent du caractère `signed char` et du caractère `unsigned char` et la signature (ou non) dépend de la plate-forme.

Différents types de constantes entières (appelés *littéraux* dans le jargon C) peuvent être écrits dans des bases différentes et de différentes largeurs, en fonction de leur préfixe ou suffixe.

```
/* the following variables are initialized to the same value: */
int d = 42;   /* decimal constant (base10) */
int o = 052;  /* octal constant (base8) */
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Les constantes décimales sont toujours `signed` . Les constantes hexadécimales commencent par `0x` ou `0X` et les constantes octales commencent par `0` . Les deux derniers sont `signed` ou `unsigned` selon que la valeur correspond ou non au type signé.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 65536l; /* l or L represent long int */
```

Sans suffixe, la constante a le premier type qui correspond à sa valeur, c'est-à-dire qu'une constante décimale supérieure à `INT_MAX` est de type `long` si possible, ou `long long` sinon.

Le fichier d'en-tête `<limits.h>` décrit les limites des entiers comme suit. Leurs valeurs définies par la mise en œuvre doivent être égales ou supérieures (valeur absolue) à celles indiquées ci-dessous, avec le même signe.

Macro	Type	Valeur
`CHAR_BIT`	plus petit objet qui n'est pas un champ de bits (octet)	8
`SCHAR_MIN`	`signed char`	-127 / $-(2^{7} - 1)$
`SCHAR_MAX`	`signed char`	+127 / $2^{7} - 1$
`UCHAR_MAX`	`unsigned char`	255/$2^{8} - 1$
`CHAR_MIN`	`char`	voir ci-dessous
`CHAR_MAX`	`char`	voir ci-dessous
`SHRT_MIN`	`short int`	-32767 / $-(2^{15} - 1)$
`SHRT_MAX`	`short int`	+32767 / $2^{15} - 1$
`USHRT_MAX`	`unsigned short int`	65535/$2^{16} - 1$
`INT_MIN`	`int`	-32767 / $-(2^{15} - 1)$
`INT_MAX`	`int`	+32767 / $2^{15} - 1$
`UINT_MAX`	`unsigned int`	65535/$2^{16} - 1$
`LONG_MIN`	`long int`	-2147483647 / $-(2^{31} - 1)$
`LONG_MAX`	`long int`	+2147483647 / $2^{31} - 1$
`ULONG_MAX`	`unsigned long int`	4294967295/$2^{32} - 1$

C99

Macro	Type	Valeur
LLONG_MIN	long long int	-9223372036854775807 / - (2 ^{63} - 1)
LLONG_MAX	long long int	+9223372036854775807 / 2 ^{63} - 1
ULLONG_MAX	unsigned long long int	18446744073709551615/2 ^{64} - 1

Si la valeur d'un objet de type `char` sign- `CHAR_MIN` est utilisée dans une expression, la valeur de `CHAR_MIN` doit être la même que celle de `SCHAR_MIN` et la valeur de `CHAR_MAX` doit être identique à celle de `SCHAR_MAX` . Si la valeur d'un objet de type `char` ne se prolonge pas lorsqu'il est utilisé dans une expression, la valeur de `CHAR_MIN` doit être 0 et la valeur de `CHAR_MAX` doit être identique à celle de `UCHAR_MAX` .

C99

La norme C99 a ajouté un nouvel en-tête, `<stdint.h>` , qui contient les définitions des entiers à largeur fixe. Voir l'exemple entier de largeur fixe pour une explication plus détaillée.

Littéraux de chaîne

Une chaîne littérale dans C est une séquence de caractères, terminée par un littéral zéro.

```
char* str = "hello, world"; /* string literal */

/* string literals can be used to initialize arrays */
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */
char a2[4] = "abc"; /* same as a1 */
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

Les littéraux de chaîne **ne** sont **pas modifiables** (et peuvent en fait être placés dans une mémoire en lecture seule telle que .rodata). Tenter de modifier leurs valeurs entraîne un comportement indéfini.

```
char* s = "foobar";
s[0] = 'F'; /* undefined behaviour */

/* it's good practice to denote string literals as such, by using `const` */
char const* s1 = "foobar";
s1[0] = 'F'; /* compiler error! */
```

Plusieurs littéraux de chaîne sont concaténés au moment de la compilation, ce qui signifie que vous pouvez écrire des constructions comme celles-ci.

C99

```
/* only two narrow or two wide string literals may be concatenated */
char* s = "Hello, " "World";
```

C99

```
/* since C99, more than two can be concatenated */
/* concatenation is implementation defined */
char* s1 = "Hello" ", " "World";

/* common usages are concatenations of format strings */
char* fmt = "%" PRId16; /* PRId16 macro since C99 */
```

Les littéraux de chaîne, identiques aux constantes de caractères, prennent en charge différents jeux de caractères.

```
/* normal string literal, of type char[] */
char* s1 = "abc";

/* wide character string literal, of type wchar_t[] */
wchar_t* s2 = L"abc";
```

C11

```
/* UTF-8 string literal, of type char[] */
char* s3 = u8"abc";

/* 16-bit wide string literal, of type char16_t[] */
char16_t* s4 = u"abc";

/* 32-bit wide string literal, of type char32_t[] */
char32_t* s5 = U"abc";
```

Types entiers de largeur fixe (depuis C99)

C99

L'en-tête `<stdint.h>` fournit plusieurs définitions de type entier à largeur fixe. Ces types sont *facultatifs* et ne sont fournis que si la plate-forme a un type entier de la largeur correspondante et si le type signé correspondant a une représentation à deux complément des valeurs négatives.

Reportez-vous à la section Remarques pour obtenir des conseils d'utilisation sur les types de largeur fixe.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255;  /* exactly 8-bits wide */

int64_t i64 = -65  /* exactly 64 bit in two's complement representation */
```

Constantes à virgule flottante

Le langage C dispose de trois types réels de virgule flottante, `float` , `double` et `long double` .

```
float f = 0.314f;        /* suffix f or F denotes type float */
double d = 0.314;        /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */
```

```
/* the different parts of a floating point definition are optional */
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

L'en-tête `<float.h>` définit différentes limites pour les opérations en virgule flottante.

L'arithmétique en virgule flottante est définie par l'implémentation. Cependant, la plupart des plates-formes modernes (arm, x86, x86_64, MIPS) utilisent les opérations à virgule flottante IEEE 754 .

C possède également trois types de virgule flottante complexes optionnels dérivés de ce qui précède.

Interprétation des déclarations

Une particularité syntaxique distinctive de C est que les déclarations reflètent l'utilisation de l'objet déclaré tel qu'il serait dans une expression normale.

L'ensemble d'opérateurs suivant avec une priorité et une associativité identiques est réutilisé dans les déclarants, à savoir:

- l'opérateur unaire `*` "dereference" qui désigne un pointeur;
- l'opérateur binary `[]` "array subscription" qui désigne un tableau;
- l'opérateur (1 + n) -ary `()` "fonction call" qui dénote une fonction;
- le `()` regroupant les parenthèses qui remplacent la priorité et l'associativité des autres opérateurs listés.

Les trois opérateurs ci-dessus ont la priorité et l'associativité suivantes:

Opérateur	Priorité relative	Associativité
`[]` (abonnement tableau)	1	De gauche à droite
`()` (appel de fonction)	1	De gauche à droite
`*` (déréférence)	2	De droite à gauche

Lors de l'interprétation des déclarations, il faut partir de l'identifiant extérieur et appliquer les opérateurs adjacents dans le bon ordre, comme indiqué dans le tableau ci-dessus. Chaque application d'un opérateur peut être remplacée par les mots anglais suivants:

Expression	Interprétation
`thing[X]`	un tableau de taille `X` de ...
`thing(t1, t2, t3)`	une fonction prenant `t1` , `t2` , `t3` et retournant ...

Expression	Interprétation
`*thing`	un pointeur sur ...

Il s'ensuit que le début de l'interprétation en anglais commence toujours par l'identifiant et se termine par le type qui se trouve à gauche de la déclaration.

Exemples

```
char *names[20];
```

`[]` a priorité sur `*` , donc l'interprétation est la suivante: `names` est un tableau de taille 20 d'un pointeur sur `char` .

```
char (*place)[10];
```

Dans le cas de l' utilisation de parenthèses pour modifier la priorité, le `*` est appliqué en premier: `place` est un pointeur sur un tableau de taille 10 de `char` .

```
int fn(long, short);
```

Il n'y a pas de priorité à se soucier ici: `fn` est une fonction qui prend du `long` , `short` et qui renvoie `int` .

```
int *fn(void);
```

Le `()` est appliqué en premier: `fn` est une fonction qui prend un `void` et renvoie un pointeur sur `int` .

```
int (*fp)(void);
```

Substitution de la priorité de `()` : `fp` est un pointeur vers une fonction prenant `void` et le retour `int` .

```
int arr[5][8];
```

Les tableaux multidimensionnels ne font pas exception à la règle; les opérateurs `[]` sont appliqués dans l'ordre de gauche à droite en fonction de l'associativité de la table: `arr` est un tableau de taille 5 d'un tableau de taille 8 de `int` .

```
int **ptr;
```

Les deux opérateurs de déréférencement ont la même priorité, de sorte que l'associativité prend effet. Les opérateurs sont appliqués dans l'ordre de droite à gauche: `ptr` est un pointeur sur un pointeur vers un `int` .

Plusieurs déclarations

La virgule peut être utilisée comme séparateur (* pas * agissant comme l'opérateur de virgule) afin de délimiter plusieurs déclarations au sein d'une même déclaration. La déclaration suivante contient cinq déclarations:

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

Les objets déclarés dans l'exemple ci-dessus sont les suivants:

- `fn` : une fonction `void` et renvoyant `int` ;
- `ptr` : un pointeur sur un `int` ;
- `fp` : un pointeur sur une fonction prenant `int` et retournant `int` ;
- `arr` : un tableau de taille 10 d'un tableau de taille 20 de `int` ;
- `num` : `int` .

Interprétation alternative

Comme les déclarations sont utilisées en miroir, une déclaration peut également être interprétée en fonction des opérateurs pouvant être appliqués sur l'objet et du type résultant final de cette expression. Le type qui se trouve à gauche est le résultat final obtenu après application de tous les opérateurs.

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

Lire Types de données en ligne: https://riptutorial.com/fr/c/topic/309/types-de-donnees

Chapitre 62: Valgrind

Syntaxe

- Valgrind *nom-programme optionnel-arguments* < *entrée de test*

Remarques

Valgrind est un outil de débogage qui peut être utilisé pour diagnostiquer les erreurs concernant la gestion de la mémoire dans les programmes C. Valgrind peut être utilisé pour détecter des erreurs telles que l'utilisation incorrecte du pointeur, y compris l'écriture ou la lecture au-delà de l'espace alloué, ou un appel non valide à `free()` . Il peut également être utilisé pour améliorer les applications grâce à des fonctions de profilage de la mémoire.

Pour plus d'informations, voir http://valgrind.org .

Examples

Courir Valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

Cela exécutera votre programme et produira un rapport sur les allocations et les dé-allocations qu'il a effectuées. Il vous avertira également des erreurs courantes telles que l'utilisation de mémoire non initialisée, le déréférencement de pointeurs vers des endroits étranges, la suppression de la fin des blocs alloués à l'aide de malloc ou la libération de blocs.

Ajouter des drapeaux

Vous pouvez également activer d'autres tests, tels que:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

Voir valgrind --help pour plus d'informations sur les nombreuses options, ou consultez la documentation à l' adresse http://valgrind.org/ pour des informations détaillées sur la signification de la sortie.

Octets perdus - Oublier de libérer

Voici un programme qui appelle malloc mais pas gratuit:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
```

```
{
    char *s;

    s = malloc(26); // the culprint

    return 0;
}
```

Sans arguments supplémentaires, valgrind ne cherchera pas cette erreur.

Mais si nous `--leak-check=yes` ou `--tool=memcheck` , cela se plaint et affiche les lignes responsables de ces fuites de mémoire si le programme a été compilé en mode debug:

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4776==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776==    by 0x80483F8: main (missing_free.c:9)
==4776==
```

Si le programme n'est pas compilé en mode débogage (par exemple avec l'indicateur `-g` dans GCC), il nous indiquera toujours où la fuite s'est produite en termes de la fonction concernée, mais pas les lignes.

Cela nous permet de revenir en arrière et de voir quel bloc a été alloué dans cette ligne et d'essayer de suivre pour voir pourquoi il n'a pas été libéré.

Erreurs les plus fréquentes rencontrées lors de l'utilisation de Valgrind

Valgrind vous fournit les *lignes sur lesquelles l'erreur s'est produite* à la fin de chaque ligne au format `(file.c:line_no)` . Les erreurs dans valgrind sont résumées de la manière suivante:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Les erreurs les plus courantes incluent:

1. **Erreurs de lecture / écriture illégales**

```
==8451== Invalid read of size 2
==8451==    at 0x4E7381D: getenv (getenv.c:84)
==8451==    by 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451==    by 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451==    by 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451==    by 0x40059C: main (valg.c:10)
==8451==  Address 0x700000007 is not stack'd, malloc'd or (recently) free'd
```

Cela se produit lorsque le code commence à accéder à la mémoire qui n'appartient pas au programme. La taille de la mémoire utilisée vous donne également une indication de la variable utilisée.

2. **Utilisation de variables non initialisées**

```
==8795== 1 errors in context 5 of 8:
==8795== Conditional jump or move depends on uninitialised value(s)
==8795==    at 0x4E881AF: vfprintf (vfprintf.c:1631)
==8795==    by 0x4E8F898: printf (printf.c:33)
==8795==    by 0x400548: main (valg.c:7)
```

Selon l'erreur, à la ligne 7 du `valg.c` `main` de `valg.c` , l'appel à `printf()` transmettait une variable non initialisée à `printf` .

3. **Libération illégale de la mémoire**

```
==8954== Invalid free() / delete / delete[] / realloc()
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x4005A8: main (valg.c:10)
==8954==  Address 0x5203040 is 0 bytes inside a block of size 240 free'd
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40059C: main (valg.c:9)
==8954==  Block was alloc'd at
==8954==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40058C: main (valg.c:7)
```

Selon valgrind, le code libérait la mémoire illégalement (une seconde fois) à la *ligne 10* de `valg.c` , alors qu'il était déjà libéré à la *ligne 9* et que le bloc lui-même était affecté à la *ligne 7* .

Lire Valgrind en ligne: https://riptutorial.com/fr/c/topic/2674/valgrind

Chapitre 63: X-macros

Introduction

Les macros X sont une technique basée sur un préprocesseur pour minimiser le code répétitif et maintenir les correspondances données / code. Plusieurs expansions de macros distinctes basées sur un ensemble commun de données sont prises en charge en représentant l'ensemble des extensions via une macro maître unique, avec le texte de remplacement de cette macro consistant en une séquence d'expansion d'une macro interne, une pour chaque donnée. La macro interne est traditionnellement nommée `X()` , d'où le nom de la technique.

Remarques

L'utilisateur d'une macro principale de style X-macro doit fournir sa propre définition pour la macro `X()` interne et, dans son périmètre, développer la macro principale. Les références de macro internes du maître sont donc étendues en fonction de la définition de l'utilisateur de `X()` . De cette façon, la quantité de code répétitif du code source dans le fichier source peut être réduite (apparaissant une seule fois dans le texte de remplacement de `X()`), comme le préfèrent les adeptes de la philosophie «Ne pas répéter».

De plus, en redéfinissant `X()` et en développant la macro principale une ou plusieurs fois de plus, les macros X peuvent faciliter la gestion des données et du code correspondants - une extension de la macro déclare les données (par exemple, éléments de tableau ou enum). les autres extensions produisent le code correspondant.

Bien que le nom "X-macro" provienne du nom traditionnel de la macro interne, la technique ne dépend pas de ce nom particulier. Tout nom de macro valide peut être utilisé à sa place.

Les critiques incluent

- les fichiers sources qui reposent sur les macros X sont plus difficiles à lire;
- comme toutes les macros, les macros X sont strictement textuelles - elles n'offrent pas de sécurité intrinsèque; et
- Les macros X permettent la *génération de code* . En comparaison avec les alternatives basées sur les fonctions d'appel, les macros X agrandissent efficacement le code.

Une bonne explication des macros X peut être trouvée dans l'article de Randy Meyers [X-Macros] dans Dr. Dobbs (http://www.drdobbs.com/the-new-cx-macros/184401387) .

Examples

Utilisation triviale des macros X pour printfs

```
/* define a list of preprocessor tokens on which to call X */
#define X_123 X(1) X(2) X(3)
```

```
/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

Cet exemple entraînera le préprocesseur générant le code suivant:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Valeur Enum et Identifiant

```
/* declare items of the enum */
#define FOREACH \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

Ensuite, vous pouvez utiliser la valeur énumérée dans votre code et imprimer facilement son identifiant en utilisant:

```
printf("%s\n", enum2string(MyEnum_item2));
```

Extension: Donne la macro X comme argument

L'approche X-macro peut être généralisée en faisant du nom de la macro "X" un argument de la macro maître. Cela présente l'avantage d'éviter les collisions de noms de macro et de permettre l'utilisation d'une macro à usage général comme macro "X".

Comme toujours avec les macros X, la macro principale représente une liste d'éléments dont la signification est spécifique à cette macro. Dans cette variante, une telle macro peut être définie comme suit:

```
/* declare list of items */
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

On pourrait alors générer du code pour imprimer les noms d'élément comme suit:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

Cela étend à ce code:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

Contrairement aux macros X standard, où le nom "X" est une caractéristique intégrée de la macro maître, avec ce style, il peut s'avérer inutile, voire indésirable, de `PRINTSTRING` ultérieurement la macro utilisée comme argument (`PRINTSTRING` dans cet exemple).

Génération de code

Les macros X peuvent être utilisées pour la génération de code, en écrivant du code répétitif: parcourez une liste pour effectuer certaines tâches ou pour déclarer un ensemble de constantes, d'objets ou de fonctions.

Ici, nous utilisons des macros X pour déclarer un enum contenant 4 commandes et une carte de leurs noms sous forme de chaînes

Ensuite, nous pouvons imprimer les valeurs de chaîne de l’enum.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
  COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
  COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP
```

```
/* the following prints "Quit\n": */
printf("%s\n", commandNames[cmdQuit]());
```

De même, nous pouvons générer une table de saut pour appeler les fonctions par la valeur enum.

Cela nécessite que toutes les fonctions aient la même signature. S'ils ne prennent aucun argument et renvoient un int, nous placerions ceci dans un en-tête avec la définition enum:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table  */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

Tous les éléments suivants peuvent être dans des unités de compilation différentes en supposant que la partie ci-dessus est incluse en tant qu'en-tête:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
  COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) {/* code performing open command */}
int doCmdClose(void) {/* code performing close command */}
int doCmdSave(void) {/* code performing save command */}
int doCmdQuit(void) {/* code performing quit command */}
```

Un exemple de cette technique utilisée dans le code réel est la distribution de commandes GPU dans Chrome .

Lire X-macros en ligne: https://riptutorial.com/fr/c/topic/628/x-macros

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage C	4444, Abhineet, Alejandro Caro, alk, Ankush, ArturFH, Bahm, bevenson, bfd, Blacksilver, blatinox, bta, chqrlie, Community, Dair, Dan Fairaizl, Daniel Jour, Daniel Margosian, David G., David Grayson, Donald Duck, Dov Benyomin Sohacheski, Ed Cottrell, employee of the month, EOF, EsmaeelE, Frosty The DopeMan, Iskar Jarak, Jens Gustedt, John Slegers, JonasCz, Jonathan Leffler, Juan T, juleslasne, Kusalananda, Leandros, LiHRaM, Lundin, Malick, Mark Yisri, MC93, MoultoB, msohng, Myst, Narox Nox, Neal, Nemanja Boric, Nicolas Verlet, OiciTrap, P.P., PSN, Rakitić, RamenChef, Roland Illig, Ryan Hilbert, Shoe , Shog9, skrtbhtngr, sohnryang, stackptr, syb0rg, techydesigner, tlhIngan, Toby, vasili111, Vin, Vraj Pandya
2	- classification et conversion des personnages	Alejandro Caro, Jonathan Leffler, Roland Illig, Toby
3	Affirmation	2501, AShelly, Blagovest Buyukliev, bta, eush77, greatwolf, J Wu, Jens Gustedt, Jonathan Leffler, Jossi, jxh, Leandros, Malcolm McLean, Ryan Haining, stackptr, syb0rg, Tim Post, Toby
4	Arguments de ligne de commande	4386427, A B, alk, drov, dvhh, Jonathan Leffler, Malcolm McLean, Shog9, syb0rg, Toby, Woodrow Barlow, Yotam Salmon
5	Arguments variables	2501, Blacksilver, eush77, Jean-Baptiste Yunès, Jonathan Leffler, Leandros, mirabilos, syb0rg, Toby
6	Assemblage en ligne	bevenson, EsmaeelE, Jonathan Leffler
7	Atomique	Jens Gustedt
8	Booléen	alk, Bob__, Braden Best, Chrono Kitsune, dhein, Insane, Jens Gustedt, Magisch, Mateusz Piotrowski, Peter, Toby
9	Champs de bits	alk, EvilTeach, Fantastic Mr Fox, haccks, Ishay Peled, Jens Gustedt, John Odom, Jonathan Leffler, Lundin, madD7, Paul Hutchinson, RamenChef, Rishikesh Raje, Toby, vkgade
10	Classes de stockage	alk, Blagovest Buyukliev, Chrono Kitsune, greatwolf, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, L.V.Rao,

		madD7, Neui, Nitinkumar Ambekar, P.P., Toby, tversteeg, vuko_zrno
11	commentaires	Ankush, Chandrahas Aroori, Jonathan Leffler, Toby
12	Communication interprocessus (IPC)	cLDSEED, EsmaeelE, Jonathan Leffler, Toby
13	Compilation	alk, Amani Kilumanga, bevenson, Blacksilver, Firas Moalla, haccks, Ishay Peled, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, Jossi, jxh, MC93, MikeCAT, nathanielng, P.P., Qrchack, R. Joiny, syb0rg, Toby, tofro, Turtle, Vraj Pandya, Алексей Неудачин
14	Comportement défini par la mise en œuvre	Jens Gustedt, John Bollinger, P.P.
15	Comportement non défini	2501, Abhineet, Aleksi Torhamo, alk, Antti Haapala, Armali, Ben Steffan, blatinox, bta, BurnsBA, caf, Christoph, Cody Gray, Community, cshu, DaBler, Daniel Jour, DarkDust, FedeWar, Firas Moalla, Giorgi Moniava, gsamaras, haccks, hmijail, honk, Jacob H, Jean-Baptiste Yunès, Jens Gustedt, John, John Bollinger, Jonathan Leffler, Kamiccolo, Leandros, Lundin, Magisch, Mark Yisri, Martin, MikeCAT, Nemanja Boric, P.P., Peter, Roland Illig, TimF, Toby, tversteeg, user45891, Vasfed, void
16	Contraintes	Armali, Toby, Vality
17	Conversions implicites et explicites	alk, Firas Moalla, Jens Gustedt, Jeremy Thien, kdopen, Lundin, Toby
18	Cordes	4386427, alk, Amani Kilumanga, Andrey Markeev, bevenson, catalogue_number, Chris Sprague, Chrono Kitsune, Cody Gray, Damien, Daniel, depperm, dylanweber, FedeWar, Firas Moalla, haccks, Ishay Peled, jasoninnn, Jean-Baptiste Yunès, Jens Gustedt, John Bollinger, Jonathan Leffler, Leandros, Malcolm McLean, mantal, MikeCAT, P.P., Purag, Roland Illig, stackptr, still_learning, syb0rg, Toby, vasili111, Waqas Bukhary, Wolf, Wyzard, Алексей Неудачин
19	Créer et inclure des fichiers d'en-tête	4444, Jonathan Leffler, patrick96, Sirsireesh Kodali
20	Déclaration vs définition	Ashish Ahuja, foxtrot9, Kerrek SB, Toby
21	Déclarations	alk, AnArrayOfFunctions, Blacksilver, Firas Moalla, J Wu, Jens

		Gustedt, Jonathan Leffler, Jonathon Reinhart
22	Effets secondaires	EsmaeelE, Jonathan Leffler, L.V.Rao, madD7, RamenChef, Sirsireesh Kodali, Toby
23	Énoncés d'itération / boucles: pour, pendant et après	alk, GoodDeeds, Jens Gustedt, jxh, L.V.Rao, Malcolm McLean, Nagaraj, RamenChef, reshad, Toby
24	Entrée / sortie formatée	alk, fluter, Jonathan Leffler, Jossi, lardenn, MikeCAT, polarysekt , StardustGogeta
25	Énumérations	Alejandro Caro, alk, jasoninnn, Jens Gustedt, Jonathan Leffler, OznOg, Toby
26	Fichiers et flux d'E / S	alk, bevenson, EWoodward, haccks, iRove, Jean Vitor, Jens Gustedt, Jonathan Leffler, Jossi, Leandros, Malcolm McLean, Pedro Henrique A. Oliveira, RamenChef, reshad, Snaipe, stackptr, syb0rg, tkk, Toby, tversteeg, William Pursell
27	Génération de nombres aléatoires	dylanweber, ganchito55, haccks, hexwab, Jonathan Leffler, Leandros, Malcolm McLean, MikeCAT, Toby
28	Gestion de la mémoire	4386427, alk, Anderson Giacomolli, Andrey Markeev, Ankush, Antti Haapala, Cullub, Daksh Gupta, dhein, dkrmr, doppelheathen, dvhh, elslooo, EOF, EsmaeelE, Firas Moalla, fluter, gdc, greatwolf, honk, Jens Gustedt, Jonathan Leffler, juleslasne, Luiz Berti, madD7, Malcolm McLean, Mark Yisri, Matthieu, Neui, P.P., Paul Campbell, Paul V, reflective_mind, Seth, Srikar, stackptr, syb0rg, Tamarous, tbodt, the sudhakar, Toby, tofro, Vivek S, vuko_zrno, Wyzard
29	Idiomes de programmation C courants et pratiques de développeur	Chandrahas Aroori, Jonathan Leffler, Nityesh Agarwal, Shubham Agrawal
30	Initialisation	Jonathan Leffler, Liju Thomas, P.P.
31	Inlining	Alex, EsmaeelE, Jens Gustedt, Jonathan Leffler, Toby
32	Instructions de saut	alk, Jens Gustedt, Jonathan Leffler, lordjohncena, Malcolm McLean, Sourav Ghosh, syb0rg, Toby
33	La gestion des erreurs	Jens Gustedt, stackptr
34	Les opérateurs	202_accepted, 3442, alk, Amani Kilumanga, Andrea Corbelli, Bakhtiar Hasan, BenG, blatinox, cpplearner, Damien, Dariusz, EsmaeelE, Faisal Mudhir, Fantastic Mr Fox, Firas Moalla,

		gsamaras, hrs, Iwillnotexist Idonotexist, Jens Gustedt, Jonathan Leffler, kdopen, Ken Y-N, L.V.Rao, Leandros, LostAvatar, Magisch, MikeCAT, noamgot, P.P., Paul92, Peter, stackptr, Toby, Will, Wolf, Yu Hao
35	Les syndicats	Jossi, RamenChef, Toby, Vality
36	Listes liées	4386427, alk, Andrea Biondo, bevenson, iRove, Jonathan Leffler, Jossi, Leandros, Mateusz Piotrowski, Ryan, Toby
37	Littéraux composés	alk, haccks, Jens Gustedt, Kerrek SB
38	Littéraux pour les nombres, les caractères et les chaînes	Jens Gustedt, Jonathan Leffler, Klas Lindbäck, Neui, Paul92, Toby
39	Mathématiques standard	Alejandro Caro, alk, Blagovest Buyukliev, immerhart, Jonathan Leffler, manav m-n, Toby
40	Multithreading	Parham Alvani, Toby
41	Paramètres de fonction	2501, Alejandro Caro, alk, Chrono Kitsune, ganesh kumar, George Stocker, Jens Gustedt, Jonathan Leffler, Leandros, MikeCAT, Minar Ashiq Tishan, P.P., RamenChef, Richard Chambers, someoneigna, syb0rg, Toby
42	Passer des tableaux 2D à des fonctions	deamentiaemundi, Malcolm McLean, Shrinivas Patgar, Toby
43	Pièges communs	abacles, Accepted Answer, alk, bevenson, Bjorn A., Chrono Kitsune, clearlight, Community, Dmitry Grigoryev, Dreamer, Dunno, FedeWar, Fred Barclay, Gavin Higham, Giorgi Moniava, hlovdal, Ishay Peled, Jeremy, John Hascall, Jonathan Leffler, Ken Y-N, Leandros, Lord Farquaad, MikeCAT, P.P., Roland Illig , rxantos, Sourav Ghosh, stackptr, Tamarous, techEmbedded, Toby, Waqas Bukhary
44	Pointeurs	0xEDD1E, alk, Altece, Amani Kilumanga, Andrey Markeev, Ankush, Antti Haapala, Ashish Ahuja, Bjorn A., bruno, bta, chqrlie, Courtney Pattison, Dair, Daniel Porteous, David G., dhein, dkrmr, Don't You Worry Child, e.jahandar, elslooo, EOF, erebos, Faisal Mudhir, Fantastic Mr Fox, FedeWar, Firas Moalla , fluter, foxtrot9, Gavin Higham, gdc, Giorgi Moniava, gsamaras, haccks, haltode, Harry Johnston, Hemant Kumar, honk, Jens Gustedt, Jonathan Leffler, Jonnathan Soares, Josh de Kock, jpX , L.V.Rao, LaneL, Leandros, Luiz Berti, Malcolm McLean, Matthieu, Michael Fitzpatrick, MikeCAT, Neui, Nitinkumar Ambekar, OiciTrap, P.P., Pbd, Peter, RamenChef, raymai97,

		Rohan, Sergey, Shahbaz, signal, slugonamission, solomonope, someoneigna, Spidey, Srikar, stackptr, syb0rg, tbodt, the sudhakar, thndrwrks, Toby, Vality, vijay kant sharma, Vivek S, Wyzard, xhienne, Алексей Неудачин
45	Pointeurs de fonction	Alejandro Caro, alk, David Refaeli, Filip Allberg, hlovdal, John Burger, Leandros, Malcolm McLean, P.P., Srikar, stackptr, Toby
46	Points de séquence	2501, Armali, bta, Community, haccks, Jens Gustedt, John Bode, Toby
47	Portée de l'identifiant	embedded_guy, Firas Moalla, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler
48	Préprocesseur et Macros	Alex Garcia, alk, bevenson, bwoebi, Dariusz, DrPrItay, Erlend Graff, EsmaeelE, EvilTeach, fastlearner, Firas Moalla, gman, hashdefine, hlovdal, javac, Jens Gustedt, Jonathan Leffler, Justin, Leandros, luser droog, Madhusoodan P, Maniero, mnoronha, Nitinkumar Ambekar, P.P., Paul J. Lucas, Peter, Richard Chambers, Robert Baldyga, stackptr, Toby, v7d8dpo4
49	Qualificatifs de type	alk, Blagovest Buyukliev, Jens Gustedt, Jesferman, madD7, tversteeg
50	Relevés de sélection	alk, bevenson, Blagovest Buyukliev, Faisal Mudhir, GoodDeeds, gsamaras, jxh, L.V.Rao, lordjohncena, MikeCAT, NeoR, noamgot, OznOg, P.P., Toby, tofro
51	Sélection générique	2501, Jens Gustedt, Sun Qingyao
52	Séquence de caractères multi-caractères	Jonathan Leffler, PassionInfinite, Toby
53	Structs	alk, Chrono Kitsune, Damien, Elazar, EsmaeelE, Faisal Mudhir, Firas Moalla, gmug, jasoninnn, Jens Gustedt, Jonathan Leffler, Jossi, kamoroso94, Madhusoodan P, OznOg, Paul Kramme, PhotometricStereo, RamenChef, Toby, Vality
54	Structure rembourrage et emballage	EsmaeelE, Jarrod Dixon, Jedi, Jesferman, Jonathan Leffler, Liju Thomas, MayeulC, tilz0R
55	Tableaux	2501, alk, AnArrayOfFunctions, AShelly, cdrini, cSmout, Dariusz , Elazar, Eli Sadoff, Firas Moalla, Guy, Iskar Jarak, Jasmin Solanki, Jens Gustedt, John Bollinger, Jonathan Leffler, L.V.Rao, Leandros, Liju Thomas, lordjohncena, Magisch, mhk, OznOg, Ray, Ryan Haining, Ryan Hilbert, stackptr, Toby, Waqas Bukhary

56	Test des frameworks	Community, EsmaeelE, Jonathan Leffler, lordjohncena, Toby, user2314737, vuko_zrno
57	Threads (natifs)	alk, Jens Gustedt, P.P.
58	Traitement du signal	3442, alk, Dariusz, Jens Gustedt, Leandros, mirabilos
59	Type d'aliasing et efficace	2501, 4386427, Jens Gustedt
60	Typedef	Buser, Chandrahas Aroori, GoodDeeds, Jonathan Leffler, mame98, PhotometricStereo, Stephen Leppik, Toby
61	Types de données	2501, alk, Blagovest Buyukliev, Firas Moalla, Jens Gustedt, Keith Thompson, Ken Y-N, Leandros, P.P., Peter, WMios
62	Valgrind	abacles, alk, Ankush, Chandrahas Aroori, Devansh Tandon, drov, Firas Moalla, J F, Jonathan Leffler, vasili111
63	X-macros	Cimbali, Jens Gustedt, John Bollinger, Leandros, MD XF, mpromonet, poolie, RamenChef, technosaurus, templatetypedef, Toby