EBook Gratuito

APPRENDIMENTO C++

Free unaffiliated eBook created from
Stack Overflow contributors.

#c++

Sommario

Di ... 1

Capitolo 1: Iniziare con C ++ ... 2

Osservazioni ... 2

Versioni ... 2

Examples ... 2

Ciao mondo ... 2

Analisi ... 2

Commenti ... 4

Commenti a riga singola ... 4

Commenti tipo C / stile ... 4

Importanza dei commenti ... 5

Marcatori di commento utilizzati per disabilitare il codice ... 6

Funzione ... 6

Dichiarazione delle funzioni ... 6

Chiamata di funzione ... 7

Definizione della funzione ... 8

Funzione di sovraccarico ... 8

Parametri predefiniti ... 8

Chiamate con funzioni speciali - Operatori ... 9

Visibilità di prototipi e dichiarazioni di funzioni ... 9

Il processo di compilazione C ++ standard ... 11

preprocessore ... 11

Capitolo 2: Algoritmi di libreria standard ... 13

Examples ... 13

std :: for_each ... 13

std :: next_permutation ... 13

std :: accumulano ... 14

std :: trovare ... 16

std :: count ... 17

std :: count_if 18
std :: find_if 19
std :: min_element 21
Uso di std :: nth_element per trovare la mediana (o altri quantili) 22
Capitolo 3: Allineamento 24
introduzione 24
Osservazioni 24
Examples 24
Interrogare l'allineamento di un tipo 24
Controllo dell'allineamento 25
Capitolo 4: Altri comportamenti non definiti in C ++ 26
introduzione 26
Examples 26
Riferendosi a membri non statici negli elenchi di inizializzatori 26
Capitolo 5: Aritmetica in virgola mobile 27
Examples 27
I numeri in virgola mobile sono strani 27
Capitolo 6: Array 29
introduzione 29
Examples 29
Dimensione array: digita sicuro al momento della compilazione 29
Array raw di dimensioni dinamiche 30
Espansione dell'array di dimensioni dinamiche utilizzando std :: vector 31
Una matrice di matrice raw di dimensioni fisse (ovvero una matrice raw 2D) 32
Una matrice di dimensioni dinamiche che utilizza std :: vector per l'archiviazione 33
Inizializzazione di array 35
Capitolo 7: attributi 37
Sintassi 37
Examples 37
[[senza ritorno]] 37
[[sfumare]] 38
[[deprecato]] e [[deprecato ("motivo")]] 39

[[Nodiscard]] 40
[[Maybe_unused]] 40
Capitolo 8: auto 42
Osservazioni 42
Examples 42
Campione automatico di base 42
Modelli auto ed espressioni 43
auto, const e riferimenti 43
Tipo di ritorno finale 44
Lambda generico (C ++ 14) 44
oggetti auto e proxy 45
Capitolo 9: C ++ 11 Modello di memoria 46
Osservazioni 46
Operazioni atomiche 46
Consistenza sequenziale 47
Ordinamento rilassato 47
Rilascio-Acquisisci ordini 47
Rilascio-Consuma ordine 48
Recinzioni 48
Examples 48
Hai bisogno di un modello di memoria 48
Esempio di recinzione 50
Capitolo 10: C incompatibilità 52
introduzione 52
Examples 52
Parole chiave riservate 52
Puntatori debolmente tipizzati 52
goto o cambia 52
Capitolo 11: Campi bit 53
introduzione 53
Osservazioni 53

Examples ... 54
Dichiarazione e uso ... 54
Capitolo 12: Categorie di valore ... 56
Examples ... 56
Significato della categoria di valore ... 56
prvalue ... 56
xValue ... 57
lvalue ... 57
glvalue ... 58
rvalue ... 58
Capitolo 13: Classi / Strutture ... 60
Sintassi ... 60
Osservazioni ... 60
Examples ... 60
Nozioni di base sulla classe ... 60
Specifier di accesso ... 61
Eredità ... 62
Eredità virtuale ... 64
Eredità multipla ... 66
Accesso ai membri della classe ... 67
sfondo ... 68
Ereditarietà privata: limitazione dell'interfaccia di base della classe ... 68
Classi e strutture finali ... 69
Amicizia ... 70
Classi / strutture annidate ... 71
Tipi di membri e alias ... 76
Membri della classe statici ... 79
Funzioni membro non statiche ... 84
Struttura / classe senza nome ... 86
Capitolo 14: Compilazione e costruzione ... 88
introduzione ... 88
Osservazioni ... 88

Examples ... 88
Compilare con GCC ... 88
Collegamento con le librerie: ... 90
Compilazione con Visual C ++ (riga di comando) ... 90
Compilazione con Visual Studio (interfaccia grafica) - Hello World ... 94
Compilando con Clang ... 101
Compilatori online ... 102
Il processo di compilazione C ++ ... 103
Compilazione con codice :: Blocchi (interfaccia grafica) ... 105
Capitolo 15: Comportamento definito dall'implementazione ... 111
Examples ... 111
Char potrebbe essere non firmato o firmato ... 111
Dimensione dei tipi integrali ... 111
Dimensione del char ... 111
Dimensione dei tipi interi con segno e senza segno ... 111
Dimensione di char16_t e char32_t ... 113
Dimensione del bool ... 113
Dimensione di wchar_t ... 114
Modelli di dati ... 114
Numero di bit in un byte ... 115
Valore numerico di un puntatore ... 115
Intervalli di tipi numerici ... 116
Rappresentazione del valore di tipi in virgola mobile ... 117
Overflow durante la conversione da numero intero a numero intero con segno ... 118
Tipo sottostante (e quindi dimensione) di enum ... 118
Capitolo 16: Comportamento indefinito ... 119
introduzione ... 119
Osservazioni ... 119
Examples ... 120
Leggere o scrivere attraverso un puntatore nullo ... 120
Nessuna dichiarazione di reso per una funzione con un tipo di reso non vuoto ... 120

Modifica di una stringa letterale ... 121
Accedere a un indice fuori dai limiti ... 121
Divisione intera per zero ... 122
Overflow intero firmato ... 122
Utilizzando una variabile locale non inizializzata ... 123
Definizioni multiple non identiche (la regola One Definition) ... 124
Associazione errata di allocazione e deallocazione di memoria ... 124
Accedere a un oggetto come il tipo sbagliato ... 125
Overflow a virgola mobile ... 126
Chiamare membri (puri) virtuali da Costruttore o Distruttore ... 126
Eliminazione di un oggetto derivato tramite un puntatore a una classe base che non ha un d ... 127
Accedere a un riferimento ciondolante ... 127
Estendere lo spazio dei nomi `std` o` posix` ... 128
Overflow durante la conversione in o dal tipo a virgola mobile ... 128
Trasmissione statica da base a derivata non valida ... 129
Chiamata di funzione tramite il tipo di puntatore a funzione non corrispondente ... 129
Modifica di un oggetto const ... 129
Accesso a membri inesistenti tramite puntatore al membro ... 130
Conversione da origine a base non valida per i puntatori ai membri ... 131
Aritmetica del puntatore non valida ... 131
Spostamento di un numero non valido di posizioni ... 132
Ritorno da una funzione [[Noreturn]] ... 132
Distruggere un oggetto che è già stato distrutto ... 132
Ricorsione infinita del modello ... 133

Capitolo 17: Comportamento non specificato ... 134

Osservazioni ... 134
Examples ... 134
Ordine di inizializzazione di globals su TU ... 134
Valore di un enum fuori limite ... 135
Cast statico dal valore di bogus void * ... 135
Risultato di alcune reinterpret_cast conversioni ... 135
Risultato di alcuni confronti tra puntatori ... 136
Spazio occupato da un riferimento ... 136

Ordine di valutazione degli argomenti della funzione ... 137
Spostato dallo stato della maggior parte delle classi di libreria standard ... 138
Capitolo 18: Concorrenza con OpenMP ... 140
introduzione ... 140
Osservazioni ... 140
Examples ... 140
OpenMP: sezioni parallele ... 140
OpenMP: sezioni parallele ... 141
OpenMP: Parallel For Loop ... 142
OpenMP: Parallel Gathering / Reduction ... 142
Capitolo 19: Confronti affiancati di classici esempi C ++ risolti tramite C ++ vs C ++ 11 ... 144
Examples ... 144
Looping attraverso un contenitore ... 144
Capitolo 20: Const Correctness ... 146
Sintassi ... 146
Osservazioni ... 146
Examples ... 146
Le basi ... 146
Const Correct Class Design ... 147
Const Correggi i parametri delle funzioni ... 149
Const Correctness come documentazione ... 151
Funzioni membro qualificate CV const : ... 151
Parametri funzione const : ... 153
Capitolo 21: constexpr ... 156
introduzione ... 156
Osservazioni ... 156
Examples ... 156
variabili di constexpr ... 156
funzioni di constexpr ... 158
Statico se dichiarazione ... 160
Capitolo 22: Contenitori C ++ ... 162
introduzione ... 162

Examples ... 162
Diagramma di flusso dei contenitori C ++ ... 162
Capitolo 23: Controllo del flusso ... 165
Osservazioni ... 165
Examples ... 165
Astuccio ... 165
interruttore ... 165
catturare ... 166
predefinito ... 166
Se ... 167
altro ... 167
vai a ... 167
ritorno ... 168
gettare ... 168
provare ... 169
Strutture condizionali: if, if..else ... 170
Istruzioni di salto: pausa, continua, goto, uscita ... 171
Capitolo 24: Conversioni di tipo esplicito ... 175
introduzione ... 175
Sintassi ... 175
Osservazioni ... 175
Examples ... 176
Conversione da base a derivata ... 176
Gettare via la costanza ... 177
Digitare la conversione punitiva ... 177
Conversione tra puntatore e intero ... 178
Conversione tramite costruttore esplicito o funzione di conversione esplicita ... 179
Conversione implicita ... 179
Conversioni Enum ... 180
Derivato per basare la conversione per i puntatori ai membri ... 181
void * a T * ... 181
Casting in stile C ... 182

Capitolo 25: Copia Elision 183
Examples 183
Scopo della copia elisione 183
Copia elisione garantita 184
Valore di ritorno elisione 185
Parametro elisione 186
Valore di ritorno denominato elisione 186
Copia l'inizializzazione elision 187
Capitolo 26: Copia vs Assegnazione 188
Sintassi 188
Parametri 188
Osservazioni 188
Examples 188
Operatore di assegnazione 188
Copia il costruttore 189
Copia Costruttore Assegnazione Vs Costruttore 190
Capitolo 27: Costruire sistemi 192
introduzione 192
Osservazioni 192
Examples 192
Generazione dell'ambiente di compilazione con CMake 192
Compilare con GNU make 193
introduzione 193
Regole di base 193
Build incrementali 194
Documentazione 195
Costruire con SCons 196
Ninja 196
introduzione 196
NMAKE (Utilità di manutenzione programma Microsoft) 197
introduzione 197

Autotools (GNU) ... 197
introduzione ... 197
Capitolo 28: Data e ora usando intestazione ... 198
Examples ... 198
Misurare il tempo usando ... 198
Trova il numero di giorni tra due date ... 198
Capitolo 29: decltype ... 200
introduzione ... 200
Examples ... 200
Esempio di base ... 200
Un altro esempio ... 200
Capitolo 30: Digita la cancellazione ... 202
introduzione ... 202
Examples ... 202
Meccanismo di base ... 202
Cancellazione fino a un tipo normale con vtable manuale ... 203
Un solo spostamento `std :: function` ... 206
Cancellando fino a un buffer contiguo di T ... 208
Digita cancellando il tipo cancellato con std :: any ... 210
Capitolo 31: Digitare parole chiave ... 216
Examples ... 216
classe ... 216
struct ... 217
enum ... 217
unione ... 219
Capitolo 32: eccezioni ... 220
Examples ... 220
Cattura eccezioni ... 220
Rethrow (propagare) l'eccezione ... 221
Funzione Try Blocks In constructor ... 222
Funzione Prova blocco per funzione normale ... 222
Funzione Try Blocks In destructor ... 223

Best practice: lancio per valore, cattura per riferimento const ... 223
Eccezione annidata ... 224
std :: uncaught_exceptions ... 226
Eccezione personalizzata ... 227
Capitolo 33: Enumerazione ... 231
Examples ... 231
Dichiarazione di enumerazione di base ... 231
Enumerazione nelle istruzioni switch ... 232
Iterazione su un enum ... 232
Enumerazioni enunciate ... 233
Dichiarazione anticipata Enum in C ++ 11 ... 234
Capitolo 34: Errori comuni di compilazione / linker (GCC) ... 236
Examples ... 236
errore: '***' non è stato dichiarato in questo ambito ... 236
variabili ... 236
funzioni ... 236
riferimento a `*** 'non definito ... 237
errore fatale: ***: nessun file o directory di questo tipo ... 238
Capitolo 35: Esempi di server client ... 239
Examples ... 239
Ciao TCP Server ... 239
Ciao client TCP ... 242
Capitolo 36: Espressioni regolari ... 244
introduzione ... 244
Sintassi ... 244
Parametri ... 244
Examples ... 245
Esempi di regex_match e regex_search di base ... 245
regex_replace Esempio ... 245
regex_token_iterator Esempio ... 246
regex_iterator Esempio ... 246
Divisione di una corda ... 247

quantificatori ... 247
ancore ... 249
Capitolo 37: File di intestazione ... 250
Osservazioni ... 250
Examples ... 250
Esempio di base ... 250
File sorgenti ... 250
Il processo di compilazione ... 251
Modelli nei file di intestazione ... 252
Capitolo 38: File I / O ... 253
introduzione ... 253
Examples ... 253
Aprire un file ... 253
Lettura da un file ... 254
Scrivere su un file ... 256
Modalità di apertura ... 257
Chiusura di un file ... 258
Flushing a stream ... 259
Lettura di un file ASCII in una stringa std :: ... 259
Lettura di un file in un contenitore ... 260
Leggere una `struct` da un file di testo formattato. ... 261
Copia di un file ... 262
Controllare la fine del file all'interno di una condizione di loop, cattiva pratica? ... 263
Scrittura di file con impostazioni internazionali non standard ... 263
Capitolo 39: Funzione C ++ "call by value" vs. "call by reference" ... 266
introduzione ... 266
Examples ... 266
Chiama per valore ... 266
Capitolo 40: Funzione di sovraccarico ... 268
introduzione ... 268
Osservazioni ... 268
Examples ... 268

Cos'è il sovraccarico di funzione? 268
Tipo di ritorno Sovraccarico funzione 270
Funzione membro qualifica cv Sovraccarico 270
Capitolo 41: Funzioni costanti dei membri della classe 273
Osservazioni 273
Examples 273
funzione membro costante 273
Capitolo 42: Funzioni dei membri virtuali 275
Sintassi 275
Osservazioni 275
Examples 275
Uso dell'override con virtuale in C ++ 11 e versioni successive 275
Funzioni membro virtuale vs non virtuale 276
Funzioni virtuali finali 277
Comportamento delle funzioni virtuali in costruttori e distruttori 278
Pure funzioni virtuali 279
Capitolo 43: Funzioni inline 282
introduzione 282
Sintassi 282
Osservazioni 282
In linea come direttiva di collegamento 282
FAQs 282
Guarda anche 283
Examples 283
Dichiarazione di funzione inline non membro 283
Definizione della funzione inline non membro 283
Funzioni inline dei membri 283
Qual è la funzione di inlining? 284
Capitolo 44: Funzioni membro non statico 285
Sintassi 285
Osservazioni 285

Examples ... 285
Funzioni membro non statiche ... 285
incapsulamento ... 286
Nascondere e importare il nome ... 287
Funzioni dei membri virtuali ... 289
Const Correctness ... 291
Capitolo 45: Funzioni speciali per gli utenti ... 294
Examples ... 294
Distruttori virtuali e protetti ... 294
Sposta e copia impliciti ... 295
Copia e scambia ... 295
Costruttore predefinito ... 297
Distruttore ... 299
Capitolo 46: Futures e promesse ... 302
introduzione ... 302
Examples ... 302
std :: future e std :: promise ... 302
Esempio asincrono rinviato ... 302
std :: packaged_task e std :: future ... 303
std :: future_error e std :: future_errc ... 303
std :: future e std :: async ... 305
Classi di operazioni asincrone ... 306
Capitolo 47: Generazione di numeri casuali ... 307
Osservazioni ... 307
Examples ... 307
Vero generatore di valori casuali ... 307
Generazione di un numero pseudo-casuale ... 308
Utilizzo del generatore per più distribuzioni ... 308
Capitolo 48: Gestione della memoria ... 310
Sintassi ... 310
Osservazioni ... 310
Examples ... 310

Pila ... 310
Archiviazione libera (heap, allocazione dinamica ...) ... 311
Posizionamento nuovo ... 312
Capitolo 49: Gestione delle risorse ... 315
introduzione ... 315
Examples ... 315
L'acquisizione delle risorse è l'inizializzazione ... 315
Mutex e sicurezza del filo ... 316
Capitolo 50: Idolo di Pimpl ... 318
Osservazioni ... 318
Examples ... 318
Idioma Pimpl di base ... 318
Capitolo 51: Implementazione del modello di progettazione in C ++ ... 320
introduzione ... 320
Osservazioni ... 320
Examples ... 320
Modello di osservatore ... 320
Modello adattatore ... 323
Modello di fabbrica ... 325
Builder Pattern with Fluent API ... 326
Passa il costruttore in giro ... 328
Variante di design: oggetto mutabile ... 329
Capitolo 52: Inoltro perfetto ... 330
Osservazioni ... 330
Examples ... 330
Funzioni di fabbrica ... 330
Capitolo 53: Input / output di base in c ++ ... 332
Osservazioni ... 332
Examples ... 332
input dell'utente e output standard ... 332
Capitolo 54: Internazionalizzazione in C ++ ... 334

Osservazioni 334
Examples 334
Comprensione delle caratteristiche della stringa C ++ 334
Capitolo 55: iteratori 336
Examples 336
C Iterators (Puntatori) 336
Breaking It Down 336
Panoramica 337
Gli iteratori sono posizioni 337
Da Iterators ai valori 337
Iteratori non validi 339
Navigazione con Iterators 339
Iterator Concepts 339
Tratti iteratori 340
Iteratori inversi 341
Iterator vettoriale 342
Mappa Iterator 342
Stream Iterators 343
Scrivi il tuo iteratore supportato dal generatore 343
Capitolo 56: Iterazione 345
Examples 345
rompere 345
Continua 345
fare 345
per 345
mentre 346
range-based per loop 346
Capitolo 57: La regola del tre, cinque e zero 347
Examples 347
Regola del Cinque 347
Regola dello zero 348

Regola del tre 349
Protezione di autoassegnazione 351
Capitolo 58: lambda 353
Sintassi 353
Parametri 353
Osservazioni 354
Examples 354
Cos'è un'espressione lambda? 354
Specifica del tipo di reso 357
Cattura in base al valore 358
Acquisizione generalizzata 359
Cattura per riferimento 360
Cattura predefinita 361
Lambda generico 361
Conversione al puntatore della funzione 362
Classe lambda e cattura di questo 363
Portare funzioni lambda a C ++ 03 usando i funtori 365
Lambda ricorsivo 366
Usa la std::function 366
Usando due puntatori intelligenti: 366
Usa un combinatore a Y 367
Utilizzo di lambda per il disimballaggio del pacchetto di parametri inline 368
Capitolo 59: Layout dei tipi di oggetto 370
Osservazioni 370
Examples 370
Tipi di classe 370
Tipi aritmetici 373
Tipi di caratteri stretti 373
Tipi interi 373
Tipi di virgola mobile 373
Array 374
Capitolo 60: letterali 375

introduzione ... 375
Examples ... 375
vero ... 375
falso ... 375
nullptr ... 375
Questo ... 376
Intero letterale ... 376
Capitolo 61: Letterali definiti dall'utente ... 379
Examples ... 379
Valori letterali definiti dall'utente con valori double lunghi ... 379
Valori standard definiti dall'utente per la durata ... 379
Letterali definiti dall'utente standard per le stringhe ... 380
Valori standard definiti dall'utente per complessi ... 380
Valore letterale definito dall'utente self-made per binario ... 381
Capitolo 62: Lo standard ISO C ++ ... 383
introduzione ... 383
Osservazioni ... 383
Examples ... 384
Bozze di lavoro correnti ... 384
C ++ 11 ... 384
Estensioni della lingua ... 384
Caratteristiche generali ... 384
Classi ... 385
Altri tipi ... 385
Modelli ... 385
Concorrenza ... 385
Funzioni linguistiche varie ... 385
Estensioni della libreria ... 386
Generale ... 386
Contenitori e algoritmi ... 386
Concorrenza ... 386
C ++ 14 ... 386

Estensioni della lingua ... 387
Estensioni della libreria ... 387
Deprecato / Rimosso ... 387
C ++ 17 ... 387
Estensioni della lingua ... 387
Estensioni della libreria ... 388
C ++ 03 ... 388
Estensioni della lingua ... 388
C ++ 98 ... 388
Estensioni della lingua (rispetto a C89 / C90) ... 388
Estensioni della libreria ... 389
C ++ 20 ... 389
Estensioni della lingua ... 389
Estensioni della libreria ... 389
Capitolo 63: Loops ... 390
introduzione ... 390
Sintassi ... 390
Osservazioni ... 390
Examples ... 390
Range-Based For ... 390
Per ciclo ... 393
Mentre loop ... 396
Dichiarazione di variabili in condizioni ... 396
Ciclo Do-while ... 397
Istruzioni Loop Control: Break e Continue ... 398
Intervallo: per un sottogruppo ... 399
Capitolo 64: Manipolatori di flusso ... 401
introduzione ... 401
Osservazioni ... 401
Examples ... 402
Manipolatori di flusso ... 403

Manipolatori del flusso di uscita ... 409
Manipolatori del flusso di input ... 410
Capitolo 65: Manipolazione bit ... 412
Osservazioni ... 412
Examples ... 412
Impostazione un po ' ... 412
Manipolazione bit in stile C. ... 412
Utilizzando std :: bitset ... 412
Schiarirsi un po ' ... 412
Manipolazione bit in stile C. ... 412
Utilizzando std :: bitset ... 413
Toggling un po ' ... 413
Manipolazione bit in stile C. ... 413
Utilizzando std :: bitset ... 413
Controllando un po ' ... 413
Manipolazione bit in stile C. ... 413
Utilizzando std :: bitset ... 414
Cambiare l'ennesimo bit in x ... 414
Manipolazione bit in stile C. ... 414
Utilizzando std :: bitset ... 414
Imposta tutti i bit ... 414
Manipolazione bit in stile C. ... 414
Utilizzando std :: bitset ... 414
Rimuovi il bit impostato più a destra ... 414
Manipolazione bit in stile C. ... 414
Set di bit di conteggio ... 415
Controlla se un numero intero è una potenza di 2 ... 416
Applicazione di manipolazione di bit: lettera da piccola a maiuscola ... 416
Capitolo 66: metaprogrammazione ... 418
introduzione ... 418

Osservazioni ... 418
Examples ... 418
Calcolo dei fattori ... 418
Iterazione su un pacchetto di parametri ... 421
Iterazione con std :: integer_sequence ... 422
Invio di tag ... 423
Rileva se l'espressione è valida ... 424
Calcolo della potenza con C ++ 11 (e versioni successive) ... 425
Distinzione manuale dei tipi quando viene dato qualsiasi tipo T ... 426
If-then-else ... 427
Min / Max generico con conteggio di argomenti variabili ... 427
Capitolo 67: Metaprogrammazione aritmitica ... 429
introduzione ... 429
Examples ... 429
Calcolo del potere in O (log n) ... 429
Capitolo 68: Modelli ... 431
introduzione ... 431
Sintassi ... 431
Osservazioni ... 431
Examples ... 433
Modelli di funzione ... 433
Inoltro di argomenti ... 434
Modello di classe base ... 435
Specializzazione dei modelli ... 436
Specializzazione di template parziale ... 436
Valore del parametro del modello predefinito ... 438
Modello alias ... 438
Parametri del modello di modello ... 439
Dichiarare argomenti modello non di tipo con auto ... 440
Empty deleter personalizzato per unique_ptr ... 440
Parametro modello non di tipo ... 440
Strutture dati modello variabile ... 441

Istanziazione esplicita ... 445
Capitolo 69: Modelli di espressione ... 447
Examples ... 447
Modelli di espressioni di base sulle espressioni algebriche element-wise ... 447
File vec.hh: wrapper per std :: vector, utilizzato per mostrare il log quando viene chiama ... 449
File expr.hh: implementazione di modelli di espressione per operazioni basate su elementi ... 450
File main.cc: prova file src ... 454
Un esempio di base che illustra i modelli di espressione ... 456
Capitolo 70: Modello di modello curiosamente ricorrente (CRTP) ... 461
introduzione ... 461
Examples ... 461
Il modello di modello Curiosamente ricorrente (CRTP) ... 461
CRTP per evitare la duplicazione del codice ... 463
Capitolo 71: mutex ... 465
Osservazioni ... 465
È meglio usare std :: shared_mutex di std :: shared_timed_mutex 465
Il codice seguente è l'implementazione di MSVC14.1 di std :: shared_mutex. ... 465
Il codice seguente è l'implementazione di MSVC14.1 di std :: shared_timed_mutex. ... 467
std :: shared_mutex ha elaborato read / write più di 2 volte rispetto a std :: shared_time ... 470
Examples ... 473
std :: unique_lock, std :: shared_lock, std :: lock_guard ... 473
Strategie per le classi di blocco: std :: try_to_lock, std :: adopt_lock, std :: defer_loc ... 474
std :: mutex ... 475
std :: scope_lock (C ++ 17) ... 476
Tipi di mutex ... 476
std :: blocco ... 476
Capitolo 72: Mutex ricorsivo ... 477
Examples ... 477
std :: recursive_mutex ... 477
Capitolo 73: Namespace ... 478
introduzione ... 478

Sintassi ... 478
Osservazioni ... 478
Examples ... 479
Cosa sono gli spazi dei nomi? ... 479
Creare spazi dei nomi ... 480
Estendere spazi dei nomi ... 481
Usando la direttiva ... 481
Ricerca dipendente dall'argomento ... 482
Quando non si verifica ADL ... 482
Spazio dei nomi in linea ... 483
Spazi dei nomi senza nome / anonimi ... 485
Spazi dei nomi nidificati compatti ... 485
Aliasing di un lungo spazio dei nomi ... 486
Ambito di dichiarazione alias ... 486
Alias dello spazio dei nomi ... 487
Capitolo 74: Oggetti callable ... 489
introduzione ... 489
Osservazioni ... 489
Examples ... 489
Puntatori di funzione ... 489
Classi con operatore () (Funzionalità) ... 490
Capitolo 75: Operatori di bit ... 491
Osservazioni ... 491
Examples ... 491
& - AND bit a bit ... 491
| - OR bit a bit ... 492
^ - XOR bit a bit (OR esclusivo) ... 492
~ - bit per bit NOT (complemento unario) ... 494
<< - spostamento a sinistra ... 495
>> - spostamento a destra ... 496
Capitolo 76: Ordinamento ... 497
Osservazioni ... 497

Examples ... 497
Ordinamento di contenitori di sequenza con ordinamento specificato ... 497
Ordinare i contenitori della sequenza con un operatore meno carico ... 497
Ordinare i contenitori della sequenza usando la funzione di confronto ... 498
Ordinamento di contenitori di sequenza usando espressioni lambda (C ++ 11) ... 499
Contenitori di ordinamento e sequenza ... 500
ordinamento con std :: map (ascendente e discendente) ... 501
Ordinamento di array incorporati ... 503
Capitolo 77: Ottimizzazione ... 504
introduzione ... 504
Examples ... 504
Inline Expansion / Inlining ... 504
Ottimizzazione della base vuota ... 504
Capitolo 78: Ottimizzazione in C ++ ... 506
Examples ... 506
Ottimizzazione della classe base vuota ... 506
Introduzione alle prestazioni ... 506
Ottimizzazione eseguendo meno codice ... 507
Rimozione del codice inutile ... 507
Fare codice solo una volta ... 507
Prevenire la ridistribuzione, la copia / lo spostamento inutili ... 508
Usare contenitori efficienti ... 508
Ottimizzazione di piccoli oggetti ... 509
Esempio ... 509
Quando usare? ... 510
Capitolo 79: Pacchetti di parametri ... 512
Examples ... 512
Un modello con un pacchetto di parametri ... 512
Espansione di un pacchetto di parametri ... 512
Capitolo 80: Parola chiave amico ... 513
introduzione ... 513
Examples ... 513

Funzione amico 513
Metodo amico 514
Classe di amici 514
Capitolo 81: parola chiave const 516
Sintassi 516
Osservazioni 516
Examples 516
Const variabili locali 516
Puntatori Const 517
Funzioni membro Const 517
Evitare la duplicazione del codice nei metodi getter const e non-const 517
Capitolo 82: parola chiave mutevole 520
Examples 520
modificatore del membro della classe non statico 520
mutande lambda 520
Capitolo 83: parole 522
introduzione 522
Sintassi 522
Osservazioni 522
Examples 524
asm 524
esplicito 525
noexcept 525
typename 527
taglia di 527
Parole chiave diverse 528
Capitolo 84: Parole chiave di base 533
Examples 533
int 533
bool 533
carbonizzare 533
char16_t 533

char32_t ... 533
galleggiante ... 534
Doppio ... 534
lungo ... 534
corto ... 535
vuoto ... 535
wchar_t ... 535
Capitolo 85: Parole chiave di dichiarazione variabile ... 537
Examples ... 537
const ... 537
decltype ... 537
firmato ... 538
unsigned ... 538
volatile ... 539
Capitolo 86: Piega le espressioni ... 540
Osservazioni ... 540
Examples ... 540
Foldario unario ... 540
Pieghe binarie ... 541
Piegando una virgola ... 541
Capitolo 87: Polimorfismo ... 543
Examples ... 543
Definire classi polimorfiche ... 543
Downcast sicuro ... 544
Polimorfismo e distruttori ... 546
Capitolo 88: precedenza dell'operatore ... 547
Osservazioni ... 547
Examples ... 547
Operatori aritmetici ... 548
Operatori logici AND e OR ... 548
Logico && e || operatori: cortocircuito ... 548
Operatori unari ... 549

Capitolo 89: preprocessore 551
introduzione 551
Osservazioni 551
Examples 551
Includi le guardie 551
Logica condizionale e gestione multipiattaforma 552
Macro 554
Messaggi di errore del preprocessore 558
Macro predefinite 558
X-macro 560
#pragma una volta 562
Operatori preprocessori 562
Capitolo 90: profiling 564
Examples 564
Creazione di profili con gcc e gprof 564
Generazione di diagrammi callgraph con gperf2dot 565
Profilazione dell'uso della CPU con gcc e Google Perf Tools 566
Capitolo 91: puntatori 569
introduzione 569
Sintassi 569
Osservazioni 569
Examples 569
Nozioni di base del puntatore 569
Creazione di una variabile puntatore 569
Prendendo l'indirizzo di un'altra variabile 570
Accedere al contenuto di un puntatore 571
Dereferenziare i puntatori non validi 571
Operazioni di puntamento 572
Puntatore aritmetico 572
Incrementa / Decrementa 572
Addizione / sottrazione 573

Differenziamento del puntatore ... 573
Capitolo 92: Puntatori ai membri ... 575
Sintassi ... 575
Examples ... 575
Puntatori a funzioni membro statiche ... 575
Puntatori alle funzioni membro ... 576
Puntatori alle variabili membro ... 576
Puntatori a variabili membro statiche ... 577
Capitolo 93: Puntatori intelligenti ... 579
Sintassi ... 579
Osservazioni ... 579
Examples ... 579
Condivisione della proprietà (std :: shared_ptr) ... 579
Condivisione con proprietà temporanea (std :: weak_ptr) ... 582
Proprietà univoca (std :: unique_ptr) ... 583
Utilizzare i deletatori personalizzati per creare un wrapper per un'interfaccia C ... 586
Proprietà unica senza spostamento della semantica (auto_ptr) ... 587
Ottenere un shared_ptr riferendosi a questo ... 589
Puntatori di casting std :: shared_ptr ... 589
Scrivere un puntatore intelligente: value_ptr ... 590
Capitolo 94: RAII: l'acquisizione delle risorse è inizializzata ... 593
Osservazioni ... 593
Examples ... 593
Blocco ... 593
Infine / ScopeExit ... 594
ScopeSuccess (c ++ 17) ... 595
ScopeFail (c ++ 17) ... 596
Capitolo 95: Restituzione di diversi valori da una funzione ... 599
introduzione ... 599
Examples ... 599
Utilizzo dei parametri di output ... 599
Utilizzando std :: tupla ... 600

Utilizzando std :: array ... 601
Utilizzando std :: pair ... 601
Usando struct ... 602
Collegamenti strutturati ... 603
Utilizzo di un oggetto oggetto Consumer ... 604
Utilizzando std :: vector ... 605
Utilizzando Output Iterator ... 606
Capitolo 96: Ricerca del nome dipendente dall'argomento ... 607
Examples ... 607
Quali funzioni sono state trovate ... 607
Capitolo 97: Ricorsione in C ++ ... 609
Examples ... 609
Usando la ricorsione della coda e la ricorsione in stile Fibonaci per risolvere la sequenz ... 609
Ricorsione con memoization ... 609
Capitolo 98: Riferimenti ... 611
Examples ... 611
Definire un riferimento ... 611
I riferimenti C ++ sono alias di variabili esistenti ... 611
Capitolo 99: Risoluzione di sovraccarico ... 613
Osservazioni ... 613
Examples ... 613
Corrispondenza esatta ... 613
Categorizzazione dell'argomento al costo del parametro ... 614
Ricerca dei nomi e controllo degli accessi ... 615
Sovraccarico sul riferimento di inoltro ... 615
Passi per la risoluzione del sovraccarico ... 616
Promozioni aritmetiche e conversioni ... 618
Sovraccarico all'interno di una gerarchia di classi ... 619
Sovraccarico di costanza e volatilità ... 620
Capitolo 100: RTTI: informazioni di tipo run-time ... 622
Examples ... 622
Nome di un tipo ... 622

dynamic_cast ... 622
La parola chiave typeid ... 622
Quando usare quale cast in c ++ ... 623
Capitolo 101: Scopes ... 624
Examples ... 624
Semplice ambito di blocco ... 624
Variabili globali ... 624
Capitolo 102: Semaforo ... 626
introduzione ... 626
Examples ... 626
Semaphore C ++ 11 ... 626
Classe di semaforo in azione ... 626
Capitolo 103: Semantica del valore e di riferimento ... 628
Examples ... 628
Copia e supporto per muovere in profondità ... 628
definizioni ... 630
Capitolo 104: Separatori di cifre ... 632
Examples ... 632
Separatore di cifre ... 632
Capitolo 105: SFINAE (Errore di sostituzione non è un errore) ... 633
Examples ... 633
enable_if ... 633
Quando usarlo ... 633
void_t ... 635
finale decltype nei modelli di funzione ... 636
Cos'è SFINAE ... 637
enable_if_all / enable_if_any ... 638
is_detected ... 640
Risoluzione di sovraccarico con un gran numero di opzioni ... 641
Capitolo 106: sindacati ... 643
Osservazioni ... 643

Examples ... 643
Caratteristiche di base dell'Unione ... 643
Uso tipico ... 643
Comportamento indefinito ... 644
Capitolo 107: Singleton Design Pattern ... 645
Osservazioni ... 645
Examples ... 645
Inizializzazione pigra ... 645
sottoclassi ... 646
Singeton sicuro per thread ... 647
Deinitializzazione statica: sicuro singleton ... 647
Capitolo 108: Sovraccarico del modello di funzione ... 649
Osservazioni ... 649
Examples ... 649
Cos'è un sovraccarico del modello di funzione valido? ... 649
Capitolo 109: Sovraccarico dell'operatore ... 651
introduzione ... 651
Osservazioni ... 651
Examples ... 651
Operatori aritmetici ... 651
Operatori unari ... 653
Operatori di confronto ... 654
Operatori di conversione ... 655
Operatore di sottoscrizione di matrice ... 656
Operatore di chiamata di funzione ... 657
Operatore di assegnazione ... 658
Operatore NOT bit a bit ... 658
Operatori di cambio di bit per I / O ... 659
Numeri complessi rivisitati ... 660
Operatori denominati ... 664
Capitolo 110: Specifiche di collegamento ... 667
introduzione ... 667

Sintassi ... 667
Osservazioni ... 667
Examples ... 667
Gestore del segnale per sistema operativo Unix-like ... 667
Creare un'intestazione di libreria C compatibile con C ++ ... 667
Capitolo 111: Specifiers di classe di archiviazione ... 669
introduzione ... 669
Osservazioni ... 669
Examples ... 669
mutevole ... 669
Registrare ... 670
statico ... 670
auto ... 671
extern ... 672
Capitolo 112: Sposta semantica ... 674
Examples ... 674
Spostare la semantica ... 674
Sposta costruttore ... 674
Sposta il compito ... 676
Utilizzo di std :: move per ridurre la complessità da O (n²) a O (n) ... 677
Utilizzare la semantica di movimento sui contenitori ... 680
Riutilizzare un oggetto spostato ... 681
Capitolo 113: static_assert ... 682
Sintassi ... 682
Parametri ... 682
Osservazioni ... 682
Examples ... 682
static_assert ... 682
Capitolo 114: std :: Atomics ... 684
Examples ... 684
tipi atomici ... 684
Capitolo 115: std :: coppia ... 687

Examples ... 687
Creare una coppia e accedere agli elementi ... 687
Confronta gli operatori ... 687
Capitolo 116: std :: forward_list ... 689
introduzione ... 689
Osservazioni ... 689
Examples ... 689
Esempio ... 689
metodi ... 690
Capitolo 117: std :: function: per avvolgere qualsiasi elemento che è callable ... 692
Examples ... 692
Uso semplice ... 692
std :: funzione usata con std :: bind ... 692
std :: function con lambda e std :: bind ... 693
overhead `function` ... 694
Legatura std :: funzione a tipi diversi chiamabili ... 695
Memorizzazione degli argomenti delle funzioni in std :: tuple ... 697
Capitolo 118: std :: integer_sequence ... 699
introduzione ... 699
Examples ... 699
Trasforma una tupla std :: in parametri di funzione ... 699
Creare un pacchetto di parametri costituito da numeri interi ... 700
Trasforma una sequenza di indici in copie di un elemento ... 700
Capitolo 119: std :: iomanip ... 702
Examples ... 702
std :: setw ... 702
std :: setprecision ... 702
std :: setfill ... 703
std :: setiosflags ... 703
Capitolo 120: std :: map ... 706
Osservazioni ... 706
Examples ... 706

Accesso agli elementi ... 706
Inizializzazione di std :: map o std :: multimap ... 707
Eliminazione di elementi ... 708
Inserimento di elementi ... 709
Iterating su std :: map o std :: multimap ... 711
Ricerca in std :: map o in std :: multimap ... 711
Controllo del numero di elementi ... 712
Tipi di mappe ... 712
Mappa normale ... 712
Multi-Map ... 713
Hash-Map (Mappa non ordinata) ... 713
Creazione di std :: map con tipi definiti dall'utente come chiave ... 713
Ordinamento debole rigoroso ... 714
Capitolo 121: std :: matrice ... 715
Parametri ... 715
Osservazioni ... 715
Examples ... 715
Inizializzazione di uno std :: array ... 715
Accesso all'elemento ... 716
Controllo della dimensione della matrice ... 718
Iterazione attraverso la matrice ... 719
Modifica di tutti gli elementi dell'array contemporaneamente ... 719
Capitolo 122: std :: opzionale ... 720
Examples ... 720
introduzione ... 720
Altri approcci a facoltativo ... 720
Opzionale vs puntatore ... 720
Opzionale vs Sentinel ... 720
Opzionale vs std::pair<bool, T> ... 720
Utilizzo degli optionals per rappresentare l'assenza di un valore ... 720
Utilizzo degli optionals per rappresentare l'errore di una funzione ... 721
opzionale come valore di ritorno ... 722

value_or ... 723
Capitolo 123: std :: qualsiasi ... 725
Osservazioni ... 725
Examples ... 725
Utilizzo di base ... 725
Capitolo 124: std :: set e std :: multiset ... 726
introduzione ... 726
Osservazioni ... 726
Examples ... 726
Inserimento di valori in un set ... 726
Inserimento di valori in un multiset ... 727
Modifica l'ordinamento predefinito di un set ... 728
Ordinamento predefinito ... 729
Ordinamento personalizzato ... 729
Lambda sort ... 730
Altre opzioni di ordinamento ... 730
Ricerca di valori in set e multiset ... 730
Eliminazione di valori da un set ... 731
Capitolo 125: std :: string ... 733
introduzione ... 733
Sintassi ... 733
Osservazioni ... 734
Examples ... 734
scissione ... 734
Sostituzione di corde ... 735
Sostituisci per posizione ... 735
Sostituisci le occorrenze di una stringa con un'altra stringa ... 735
Concatenazione ... 736
Accedere a un personaggio ... 737
operatore [] (n) ... 737
a (n) ... 737

davanti() 737
indietro() 738
tokenize 738
Conversione in (const) char * 739
Trovare caratteri (s) in una stringa 740
Taglio di caratteri all'inizio / fine 740
Confronto lessicografico 742
Conversione a std :: wstring 743
Usando la classe std :: string_view 744
In loop attraverso ogni personaggio 745
Conversione in numeri interi / in virgola mobile 745
Conversione tra codifiche di caratteri 746
Verifica se una stringa è un prefisso di un'altra 747
Conversione in std :: string 748
Capitolo 126: std :: variante 750
Osservazioni 750
Examples 750
Base std :: uso variante 750
Crea puntatori pseudo-metodi 751
Costruire un `std :: variant` 752
Capitolo 127: std :: vector 753
introduzione 753
Osservazioni 753
Examples 753
Inizializzazione di un vettore std :: 753
Inserimento di elementi 754
Iterating Over std :: vector 756
Iterating nella direzione di andata 756
Iterazione nella direzione inversa 756
Imporre elementi const 757
Una nota sull'efficienza 758

Accesso agli elementi ... 758
Accesso basato su indice: ... 758
iteratori: ... 761
Usando std :: vector come array C ... 761
Iterator / Pointer Invalidation ... 762
Eliminazione di elementi ... 763
Eliminazione dell'ultimo elemento: ... 763
Eliminazione di tutti gli elementi: ... 763
Eliminazione elemento per indice: ... 763
Eliminazione di tutti gli elementi in un intervallo: ... 764
Eliminazione di elementi in base al valore: ... 764
Eliminazione di elementi in base alla condizione: ... 764
Eliminazione di elementi di lambda, senza creare una funzione di predicato aggiuntiva ... 764
Eliminazione di elementi in base alla condizione da un ciclo: ... 765
Eliminazione di elementi in base alla condizione da un ciclo inverso: ... 765
Trovare un elemento in std :: vector ... 766
Convertire una matrice in std :: vector ... 767
vettore : L'eccezione a così tante, tante regole ... 768
Dimensioni e capacità del vettore ... 769
Vettori concatenanti ... 771
Ridurre la capacità di un vettore ... 772
Utilizzo di un vettore ordinato per la ricerca rapida degli elementi ... 772
Funzioni che restituiscono vettori di grandi dimensioni ... 774
Trova l'elemento massimo e minimo e l'indice rispettivo in un vettore ... 775
Matrici che usano i vettori ... 776
Capitolo 128: Stream C ++ ... 777
Osservazioni ... 777
Examples ... 777
Stream di stringhe ... 777
Leggere un file fino alla fine ... 778

Lettura di un file di testo riga per riga **778**
Linee senza caratteri di spazi bianchi 778
Linee con caratteri di spaziatura 778
Lettura di un file in un buffer in una volta **779**
Copia di flussi **779**
Array **780**
Stampa di collezioni con iostream 780
Stampa di base **780**
Cast di tipo implicito **780**
Generazione e trasformazione **781**
Array **781**
Analisi dei file 782
Analisi dei file nei contenitori STL **782**
Analisi di tabelle di testo eterogenee **782**
Trasformazione **782**
Capitolo 129: Strumenti e tecniche di debug in C ++ per debugging e debug **784**
introduzione 784
Osservazioni 784
Examples 784
Il mio programma C ++ termina con segfault - valgrind 784
Analisi segfault con GDB 786
Codice pulito 787
L'uso di funzioni separate per azioni separate 788
Usando una formattazione / costruzioni coerenti 789
Fai attenzione alle parti importanti del tuo codice 789
Conclusione 789
Analisi statica 789
Avvisi del compilatore 790
Strumenti esterni 790
Altri strumenti 791
Conclusione 791

Safe-stack (Stack corruzioni) ... 791
Quali parti della pila vengono spostate? ... 791
A cosa serve effettivamente? ... 791
Come abilitarlo? ... 792
Conclusione ... 792
Capitolo 130: Strutture dati in C ++ ... 793
Examples ... 793
Implementazione di liste collegate in C ++ ... 793
Capitolo 131: Strutture di sincronizzazione del filo ... 796
introduzione ... 796
Examples ... 796
std :: shared_lock ... 796
std :: call_once, std :: once_flag ... 796
Blocco degli oggetti per un accesso efficiente ... 797
std :: condition_variable_any, std :: cv_status ... 798
Capitolo 132: Tecniche di refactoring ... 799
introduzione ... 799
Examples ... 799
Ristrutturazione a piedi attraverso ... 799
Goto Cleanup ... 801
Capitolo 133: Test unitario in C ++ ... 803
introduzione ... 803
Examples ... 803
Google Test ... 803
Esempio minimo ... 803
Catturare ... 803
Capitolo 134: The This Pointer ... 805
Osservazioni ... 805
Examples ... 805
questo puntatore ... 805
Usando questo puntatore per accedere ai dati dei membri ... 807

Usando questo puntatore per distinguere tra dati e parametri dei membri ... 808
questo Pointer CV-qualificatori ... 809
questo Pointer Ref-Qualificatori ... 812
Capitolo 135: threading ... 814
Sintassi ... 814
Parametri ... 814
Osservazioni ... 814
Examples ... 814
Operazioni di thread ... 814
Passando un riferimento a un thread ... 815
Creare uno std :: thread ... 815
Operazioni sul thread corrente ... 817
Usando std :: async invece di std :: thread ... 819
Chiamata in modo asincrono di una funzione ... 819
Insidie comuni ... 819
Assicurandosi che un thread sia sempre unito ... 819
Riassegnazione degli oggetti thread ... 820
Sincronizzazione di base ... 821
Uso delle variabili di condizione ... 821
Creare un pool di thread semplice ... 823
Memorizzazione locale del thread ... 825
Capitolo 136: Tipi atomici ... 826
Sintassi ... 826
Osservazioni ... 826
Examples ... 826
Accesso multi-thread ... 826
Capitolo 137: Tipi senza nome ... 828
Examples ... 828
Classi senza nome ... 828
Membri anonimi ... 828
Come alias di tipo ... 829
Unione anonima ... 829

Capitolo 138: tipo deduzione 830
Osservazioni 830
Examples 830
Deduzione del parametro Template per costruttori 830
Detrazione del tipo di modello 830
Deduzione del tipo automatico 831
Capitolo 139: Tipo di inferenza 834
introduzione 834
Osservazioni 834
Examples 834
Tipo di dati: automatico 834
Lambda auto 834
Cicli e auto 835
Capitolo 140: Tipo di ritorno Covariance 836
Osservazioni 836
Examples 836
1. Esempio di base senza rendimenti covarianti, mostra perché sono desiderabili 836
2. Versione di risultato covariant dell'esempio di base, controllo di tipo statico 837
3. Risultato puntatore intelligente covariant (pulizia automatica) 838
Capitolo 141: Tipo di ritorno finale 840
Sintassi 840
Osservazioni 840
Examples 840
Evitare di qualificare un nome di tipo nidificato 840
Espressioni Lambda 840
Capitolo 142: Tipo Tratti 842
Osservazioni 842
Examples 842
Tratti di tipo standard 842
costanti 842
funzioni 842

tipi ... 843
Digitare le relazioni con std :: is_same ... 843
Caratteri fondamentali del tipo ... 844
Tipo Proprietà ... 845
Capitolo 143: Typedef e digita alias ... 847
introduzione ... 847
Sintassi ... 847
Examples ... 847
Sintassi typedef di base ... 847
Usi più complessi di typedef ... 847
Dichiarazione di più tipi con typedef ... 848
Dichiarazione alias con "utilizzo" ... 848
Capitolo 144: Una regola di definizione (ODR) ... 850
Examples ... 850
Funzione definita in modo multiplo ... 850
Funzioni inline ... 850
Violazione di ODR tramite risoluzione di sovraccarico ... 852
Capitolo 145: Utilizzando la dichiarazione ... 853
introduzione ... 853
Sintassi ... 853
Osservazioni ... 853
Examples ... 853
Importazione di nomi individuali da uno spazio dei nomi ... 853
Riconoscere i membri di una classe base per evitare l'occultamento del nome ... 853
Costruttori ereditari ... 854
Capitolo 146: Utilizzando std :: unordered_map ... 855
introduzione ... 855
Osservazioni ... 855
Examples ... 855
Dichiarazione e uso ... 855
Alcune funzioni di base ... 855

Capitolo 147: Variabili in linea 857
introduzione 857
Examples 857
Definizione di un membro dati statico nella definizione della classe 857
Titoli di coda 858

Di

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

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

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

Capitolo 1: Iniziare con C ++

Osservazioni

Il programma "Hello World" è un esempio comune che può essere semplicemente utilizzato per verificare la presenza di compilatori e librerie. Usa la libreria standard C ++, con `std::cout` da `<iostream>` , e ha un solo file da compilare, riducendo al minimo la possibilità di un possibile errore dell'utente durante la compilazione.

Il processo per la compilazione di un programma C ++ differisce intrinsecamente tra compilatori e sistemi operativi. L'argomento Compilazione e costruzione contiene i dettagli su come compilare codice C ++ su piattaforme diverse per una varietà di compilatori.

Versioni

Versione	Standard	Data di rilascio
C ++ 98	ISO / IEC 14882: 1998	1998/09/01
C ++ 03	ISO / IEC 14882: 2003	2003/10/16
C ++ 11	ISO / IEC 14882: 2011	2011-09-01
C ++ 14	ISO / IEC 14882: 2014	2014/12/15
C ++ 17	TBD	2017/01/01
C ++ 20	TBD	2020/01/01

Examples

Ciao mondo

Questo programma stampa `Hello World!` al flusso di output standard:

```
#include <iostream>

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

Guardalo dal vivo su Coliru .

Analisi

Esaminiamo in dettaglio ciascuna parte di questo codice:

- `#include <iostream>` è una **direttiva per il preprocessore** che include il contenuto del file di intestazione standard di C ++ `iostream` .

  `iostream` è un **file di intestazione di libreria standard** che contiene le definizioni degli stream di input e output standard. Queste definizioni sono incluse nello spazio dei nomi `std` , spiegato di seguito.

  Gli **stream standard di input / output (I / O)** forniscono ai programmi modalità per ottenere input e output su un sistema esterno, in genere il terminale.

- `int main() { ... }` definisce una nuova funzione denominata `main` . Per convenzione, la funzione `main` viene chiamata all'esecuzione del programma. Ci deve essere una sola funzione `main` in un programma C ++ e deve sempre restituire un numero del tipo `int` .

  Qui, l' `int` è quello che viene chiamato il tipo di ritorno della funzione. Il valore restituito dalla funzione `main` è un **codice di uscita.**

  Per convenzione, un codice di uscita del programma pari a `0` o `EXIT_SUCCESS` viene interpretato come esito positivo da un sistema che esegue il programma. Qualsiasi altro codice di ritorno è associato a un errore.

  Se non è presente alcuna dichiarazione di `return` , la funzione `main` (e quindi il programma stesso) restituisce `0` per impostazione predefinita. In questo esempio, non è necessario scrivere esplicitamente `return 0;` .

  Tutte le altre funzioni, ad eccezione di quelle che restituiscono il tipo `void` , devono restituire esplicitamente un valore in base al tipo restituito, altrimenti non devono restituire del tutto.

- `std::cout << "Hello World!" << std::endl;` stampa "Hello World!" al flusso di output standard:

  - `std` è uno spazio dei nomi e `::` è l' **operatore di risoluzione dell'ambito** che consente di cercare oggetti per nome all'interno di uno spazio dei nomi.

    Ci sono molti spazi dei nomi. Qui, usiamo `::` per mostrare che vogliamo usare `cout` dallo spazio dei nomi `std` . Per ulteriori informazioni, consultare Operatore risoluzione ambito - Documentazione Microsoft .

  - `std::cout` è l'oggetto **standard del flusso di output** , definito in `iostream` , e stampa sullo standard output (`stdout`).

  - `<<` è, *in questo contesto* , l' **operatore di inserimento** del **flusso** , così chiamato perché *inserisce* un oggetto nell'oggetto del *flusso* .

    La libreria standard definisce l'operatore `<<` per eseguire l'inserimento dei dati per determinati tipi di dati in flussi di output. `stream << content` inserisce il `content` nello

stream e restituisce lo stesso flusso ma aggiornato. In questo modo è possibile concatenare inserimenti di stream: `std::cout << "Foo" << " Bar";` stampa "FooBar" sulla console.

- `"Hello World!"` è un **letterale stringa di caratteri** o un "testo letterale". L'operatore di inserimento del flusso per i valori letterali delle stringhe di caratteri è definito nel file `iostream`.

- `std::endl` è uno speciale oggetto **manipolatore di flusso I / O** , anch'esso definito nel file `iostream` . L'inserimento di un manipolatore in un flusso cambia lo stato del flusso.

  Il manipolatore di flusso `std::endl` fa due cose: prima inserisce il carattere di fine riga e poi scarica il buffer del flusso per forzare la visualizzazione del testo sulla console. Ciò garantisce che i dati inseriti nello stream vengano effettivamente visualizzati sulla tua console. (I dati di flusso vengono solitamente memorizzati in un buffer e quindi "svuotati" in batch, a meno che non si imponga immediatamente un lavaggio.)

  Un metodo alternativo che evita il flush è:

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

  dove `\n` è la **sequenza di escape dei caratteri** per il carattere di nuova riga.

- Il punto e virgola (`;`) notifica al compilatore che un'istruzione è terminata. Tutte le istruzioni C ++ e le definizioni di classe richiedono un punto e virgola di fine / fine.

Commenti

Un **commento** è un modo per inserire del testo arbitrario all'interno del codice sorgente senza che il compilatore C ++ lo interpreti con alcun significato funzionale. I commenti sono usati per dare un'idea del design o del metodo di un programma.

Esistono due tipi di commenti in C ++:

Commenti a riga singola

La sequenza double forward-slash `//` contrassegnerà tutto il testo fino a una nuova riga come commento:

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

Commenti tipo C / stile

La sequenza `/*` viene utilizzata per dichiarare l'inizio del blocco di commenti e la sequenza `*/` viene utilizzata per dichiarare la fine del commento. Tutto il testo tra le sequenze di inizio e fine è interpretato come un commento, anche se il testo è altrimenti valido sintassi C ++. Questi sono a volte chiamati commenti in "stile C", poiché questa sintassi dei commenti è ereditata dal linguaggio predecessore di C ++, C:

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

In qualsiasi commento di blocco, puoi scrivere tutto ciò che vuoi. Quando il compilatore incontra il simbolo `*/` , termina il commento di blocco:

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

L'esempio precedente è un codice C ++ (e C) valido. Tuttavia, l'aggiunta di `/*` all'interno di un commento di blocco potrebbe comportare un avviso su alcuni compilatori.

I commenti bloccati possono anche iniziare e terminare *all'interno di* una singola riga. Per esempio:

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

Importanza dei commenti

Come con tutti i linguaggi di programmazione, i commenti offrono numerosi vantaggi:

- Documentazione esplicita del codice per semplificare la lettura / manutenzione
- Spiegazione dello scopo e della funzionalità del codice
- Dettagli sulla storia o il ragionamento dietro il codice
- Inserimento di diritti d'autore / licenze, note di progetto, ringraziamenti speciali, crediti contributori, ecc. Direttamente nel codice sorgente.

Tuttavia, i commenti hanno anche i loro lati negativi:

- Devono essere mantenuti per riflettere eventuali modifiche nel codice
- I commenti eccessivi tendono a rendere il codice *meno* leggibile

La necessità di commenti può essere ridotta scrivendo un codice chiaro e autodocumentante. Un semplice esempio è l'uso di nomi esplicativi per variabili, funzioni e tipi. Factoring attività logicamente correlati in funzioni discrete va di pari passo con questo.

Marcatori di commento utilizzati per disabilitare il codice

Durante lo sviluppo, i commenti possono essere utilizzati anche per disabilitare rapidamente porzioni di codice senza eliminarlo. Questo è spesso utile per scopi di test o di debug, ma non è buono per qualcosa di diverso dalle modifiche temporanee. Questo è spesso definito come "commento".

Allo stesso modo, mantenere le vecchie versioni di un pezzo di codice in un commento a scopo di riferimento è disapprovato, poiché ingombra file pur offrendo poco valore rispetto all'esplorazione della cronologia del codice tramite un sistema di controllo delle versioni.

Funzione

Una **funzione** è un'unità di codice che rappresenta una sequenza di istruzioni.

Le funzioni possono accettare **argomenti** o valori e **restituire** un singolo valore (o non). Per utilizzare una funzione, una **chiamata di funzione** viene utilizzata su valori di argomento e l'uso della chiamata di funzione stessa viene sostituito con il suo valore di ritorno.

Ogni funzione ha una **firma di tipo** - i tipi dei suoi argomenti e il tipo del suo tipo di ritorno.

Le funzioni sono ispirate ai concetti della procedura e della funzione matematica.

- Nota: le funzioni C ++ sono essenzialmente procedure e non seguono la definizione esatta o le regole delle funzioni matematiche.

Le funzioni sono spesso pensate per svolgere un compito specifico. e può essere chiamato da altre parti di un programma. Una funzione deve essere dichiarata e definita prima di essere chiamata altrove in un programma.

- Nota: le definizioni di funzioni popolari possono essere nascoste in altri file inclusi (spesso per comodità e riutilizzo su molti file). Questo è un uso comune dei file di intestazione.

Dichiarazione delle funzioni

Una **dichiarazione di funzione** dichiara l'esistenza di una funzione con il suo nome e la firma del tipo nel compilatore. La sintassi è la seguente:

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

Nell'esempio sopra, la funzione `int add2(int i)` dichiara quanto segue al compilatore:

- Il **tipo di ritorno** è `int` .
- Il **nome** della funzione è `add2` .
- Il **numero di argomenti** per la funzione è 1:
  - Il primo argomento è del tipo `int` .
  - Il primo argomento verrà indicato nel contenuto della funzione con il nome `i` .

Il nome dell'argomento è facoltativo; la dichiarazione per la funzione potrebbe anche essere la seguente:

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

Per la **regola a** una **definizione** , una funzione con una determinata firma di tipo può essere dichiarata o definita solo una volta in un intero codice C ++ visibile al compilatore C ++. In altre parole, le funzioni con una firma di tipo specifica non possono essere ridefinite: devono essere definite una sola volta. Quindi, il seguente non è valido C ++:

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

Se una funzione non restituisce nulla, il suo tipo di ritorno è scritto come `void` . Se non ci sono parametri, l'elenco dei parametri dovrebbe essere vuoto.

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

Chiamata di funzione

Una funzione può essere chiamata dopo che è stata dichiarata. Ad esempio, il seguente programma chiama `add2` con il valore di `2` all'interno della funzione di `main` :

```
#include <iostream>

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

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

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

Qui, `add2(2)` è la sintassi per una chiamata di funzione.

Definizione della funzione

Una *definizione di funzione* * è simile a una dichiarazione, tranne che contiene anche il codice che viene eseguito quando la funzione viene chiamata all'interno del suo corpo.

Un esempio di una definizione di funzione per `add2` potrebbe essere:

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

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

Funzione di sovraccarico

È possibile creare più funzioni con lo stesso nome ma diversi parametri.

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

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

Entrambe le funzioni sono chiamate con lo stesso nome `add2` , ma la funzione effettiva chiamata dipende direttamente dalla quantità e dal tipo dei parametri nella chiamata. Nella maggior parte dei casi, il compilatore C ++ può calcolare quale funzione chiamare. In alcuni casi, il tipo deve essere dichiarato esplicitamente.

Parametri predefiniti

I valori predefiniti per i parametri di funzione possono essere specificati solo nelle dichiarazioni di funzione.

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

In questo esempio, `multiply()` può essere chiamato con uno o due parametri. Se viene fornito un solo parametro, `b` avrà il valore predefinito di 7. Gli argomenti predefiniti devono essere inseriti negli ultimi argomenti della funzione. Per esempio:

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

Chiamate con funzioni speciali - Operatori

Esistono chiamate di funzioni speciali in C ++ che hanno una sintassi diversa da `name_of_function(value1, value2, value3)` . L'esempio più comune è quello degli operatori.

Alcune sequenze di caratteri speciali che verranno ridotte a funzioni chiamate dal compilatore, come ad esempio `!` , `+` , `-` , `*` , `%` e `<<` e molti altri. Questi caratteri speciali sono normalmente associati all'uso non di programmazione o sono usati per l'estetica (ad esempio il carattere `+` è comunemente riconosciuto come simbolo di aggiunta sia nella programmazione C ++ che in matematica elementare).

C ++ gestisce queste sequenze di caratteri con una sintassi speciale; ma, in sostanza, ogni occorrenza di un operatore è ridotta a una chiamata di funzione. Ad esempio, la seguente espressione C ++:

```
3+3
```

è equivalente alla seguente chiamata di funzione:

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

Tutti i nomi delle funzioni dell'operatore iniziano con l' `operator` .

Mentre nell'immediato predecessore di C ++, C, i nomi delle funzioni dell'operatore non possono essere assegnati a significati diversi fornendo definizioni aggiuntive con diversi tipi di firma, in C ++, questo è valido. "Nascondere" le definizioni di funzioni aggiuntive sotto un unico nome di funzione viene definito **overloading dell'operatore** in C ++ ed è una convenzione relativamente comune, ma non universale, in C ++.

Visibilità di prototipi e dichiarazioni di funzioni

In C ++, il codice deve essere dichiarato o definito prima dell'uso. Ad esempio, il seguente produce un errore in fase di compilazione:

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

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

```
{
}
```

Ci sono due modi per risolvere questo problema: inserire la definizione o la dichiarazione di `foo()` prima del suo utilizzo in `main()` . Ecco un esempio:

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

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

Tuttavia è anche possibile "inoltrare-dichiarare" la funzione mettendo solo una dichiarazione "prototipo" prima del suo utilizzo e quindi definendo successivamente il corpo della funzione:

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

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

Il prototipo deve specificare il tipo di ritorno (`void`), il nome della funzione (`foo`) e il tipo di variabile list dell'argomento (`int`), ma i nomi degli argomenti NON sono richiesti .

Un modo comune per integrare questo nell'organizzazione dei file sorgente è creare un file di intestazione contenente tutte le dichiarazioni del prototipo:

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

e quindi fornire la definizione completa altrove:

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

e quindi, una volta compilato, collega il file oggetto corrispondente `foo.o` nel file oggetto compilato dove viene utilizzato nella fase di collegamento, `main.o` :

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

Un errore "simbolo esterno non risolto" si verifica quando la funzione *prototipo* e *chiamata* esistono, ma il *corpo* della funzione non è definito. Questi possono essere più difficili da risolvere in quanto il compilatore non segnalerà l'errore fino alla fase di collegamento finale e non sa a quale riga saltare nel codice per mostrare l'errore.

Il processo di compilazione C ++ standard

Il codice del programma C ++ eseguibile viene solitamente prodotto da un compilatore.

Un **compilatore** è un programma che traduce il codice da un linguaggio di programmazione in un'altra forma che è (più) direttamente eseguibile per un computer. L'uso di un compilatore per tradurre il codice è chiamato **compilazione.**

C ++ eredita la forma del suo processo di compilazione dal suo linguaggio "genitore", C. Di seguito è riportato un elenco che mostra i quattro passaggi principali della compilazione in C ++:

1. Il preprocessore C ++ copia i contenuti di qualsiasi file di intestazione incluso nel file del codice sorgente, genera codice macro e sostituisce le costanti simboliche definite usando #define con i loro valori.
2. Il file di codice sorgente espanso prodotto dal preprocessore C ++ è compilato in linguaggio assembly appropriato per la piattaforma.
3. Il codice assembler generato dal compilatore viene assemblato nel codice oggetto appropriato per la piattaforma.
4. Il file di codice oggetto generato dall'assemblatore è collegato insieme ai file di codice oggetto per tutte le funzioni di libreria utilizzate per produrre un file eseguibile.

- Nota: alcuni codici compilati sono collegati tra loro, ma non per creare un programma finale. Di solito, questo codice "linkato" può anche essere impacchettato in un formato che può essere utilizzato da altri programmi. Questo "pacchetto di codice utilizzabile e confezionato" è ciò che i programmatori C ++ chiamano **libreria.**

Molti compilatori C ++ possono anche unire o disunire alcune parti del processo di compilazione per facilità o per ulteriori analisi. Molti programmatori C ++ useranno strumenti diversi, ma tutti gli strumenti seguiranno generalmente questo processo generalizzato quando saranno coinvolti nella produzione di un programma.

Il link qui sotto estende questa discussione e fornisce una bella grafica per aiutare. [1]: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

preprocessore

Il preprocessore è una parte importante del compilatore.

Modifica il codice sorgente, tagliando alcuni bit, modificandone altri e aggiungendo altre cose.

Nei file di origine, possiamo includere le direttive del preprocessore. Queste direttive indicano al preprocessore di eseguire azioni specifiche. Una direttiva inizia con un # su una nuova riga. Esempio:

```
#define ZERO 0
```

La prima direttiva per il preprocessore che incontrerai è probabilmente la

```
#include <something>
```

direttiva. Quello che fa è prendere tutto di `something` e inserirlo nel tuo file dove era la direttiva. Il programma Hello World inizia con la linea

```
#include <iostream>
```

Questa riga aggiunge le funzioni e gli oggetti che consentono di utilizzare lo standard input e output.

Il linguaggio C, che usa anche il preprocessore, non ha tanti file header come il linguaggio C ++, ma in C ++ puoi usare tutti i file header C.

La prossima importante direttiva è probabilmente la

```
#define something something_else
```

direttiva. Questo dice al preprocessore che mentre va lungo il file, dovrebbe sostituire ogni occorrenza di `something` con `something_else` . Può anche fare cose simili alle funzioni, ma probabilmente conta come C ++ avanzato.

Il `something_else` non è necessario, ma se si definisce `something` come nulla, quindi al di fuori delle direttive del preprocessore, tutte le occorrenze di `something` svaniranno.

Questo in realtà è utile, a causa delle direttive `#if` , `#else` e `#ifdef` . Il formato per questi sarebbe il seguente:

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

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

Queste direttive inseriscono il codice che si trova nel bit true e cancella i bit falsi. questo può essere usato per avere bit di codice che sono inclusi solo su determinati sistemi operativi, senza dover riscrivere l'intero codice.

Leggi Iniziare con C ++ online: https://riptutorial.com/it/cplusplus/topic/206/iniziare-con-c-plusplus

Capitolo 2: Algoritmi di libreria standard

Examples

std :: for_each

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

effetti:

Applica `f` al risultato del dereferenziamento di ogni iteratore nell'intervallo `[first, last)` iniziando dal `first` e procedendo fino `last - 1` .

parametri:

`first, last` - l'intervallo per applicare `f` a.

`f` - oggetto richiamabile che viene applicato al risultato del dereferenziamento di ogni iteratore nell'intervallo `[first, last)` .

Valore di ritorno:

`f` (fino a C ++ 11) e `std::move(f)` (dal C ++ 11).

Complessità:

Applica `f` esattamente l' `last - first` volte.

Esempio:

c ++ 11

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

Applica la funzione data per ogni elemento del vettore `v` stampando questo elemento su `stdout` .

std :: next_permutation

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

effetti:
Setaccia la sequenza di dati dell'intervallo [primo, ultimo] nella successiva permutazione lessicograficamente più alta. Se viene fornito `cmpFun` , la regola di permutazione è personalizzata.

parametri:
`first` : l'inizio dell'intervallo da permutare, inclusivo
`last` - la fine della gamma deve essere permutata, esclusiva

Valore di ritorno:
Restituisce vero se tale permutazione esiste.
Altrimenti l'intervallo viene scambiato con la permutazione lessicograficamente più piccola e restituisce false.

Complessità:
O (n), n è la distanza dal `first last` .

Esempio :

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

stampare tutti i casi di permutazione di 1,2,3 in ordine crescente lessicograficamente.
produzione:

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

std :: accumulano

Definito nell'intestazione `<numeric>`

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

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

effetti:

std :: accumula esegue l'operazione di piega usando la funzione `f` nell'intervallo `[first, last)` iniziando con `init` come valore dell'accumulatore.

In effetti è equivalente a:

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

Nella versione (1) l' `operator+` è usato al posto di `f` , quindi accumulare sul contenitore equivale alla somma degli elementi del contenitore.

parametri:

`first, last` - l'intervallo per applicare `f` a.
`init` - valore iniziale dell'accumulatore.
`f` - funzione di piegatura binaria.

Valore di ritorno:

Valore accumulato di domande `f` .

Complessità:

O (n × k) , dove *n* è la distanza dal `first last` , *O (k)* è la complessità della funzione `f` .

Esempio:

Esempio di somma semplice:

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

Produzione:

```
10
```

Converti cifre in numero:

c ++ 11

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

e più tardi

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

c ++ 11

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

Produzione:

```
123
```

std :: trovare

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

effetti

Trova la prima occorrenza di val all'interno dell'intervallo [primo, ultimo]

parametri

`first` => iteratore che punta all'inizio dell'intervallo `last` => iteratore che punta alla fine dell'intervallo `val` => Il valore da trovare nell'intervallo

Ritorno

Un iteratore che punta al primo elemento all'interno dell'intervallo uguale (==) a val, l'iteratore punta a durare se val non viene trovato.

Esempio

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

using namespace std;

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

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

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

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

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

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


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

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

  return 0;
}
```

Produzione

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

std :: count

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

effetti

Conta il numero di elementi uguali a val

parametri

`first` => iteratore che punta all'inizio della gamma
`last` => iteratore che punta alla fine dell'intervallo
`val` => Verrà conteggiato il verificarsi di questo valore nell'intervallo

Ritorno

Il numero di elementi nell'intervallo uguali (==) a val.

Esempio

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

using namespace std;
```

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

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

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

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

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

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

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

  return 0;
}
```

Produzione

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

std :: count_if

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

effetti

Conta il numero di elementi in un intervallo per il quale una funzione di predicato specificata è vera

parametri

`first` => iteratore che punta all'inizio dell'intervallo `last` => iteratore che punta alla fine dell'intervallo `red` => funzione predicato (restituisce vero o falso)

Ritorno

Il numero di elementi nell'intervallo specificato per il quale la funzione del predicato ha restituito true.

Esempio

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

using namespace std;

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

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

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

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

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

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

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

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

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

  return 0;
}
```

Produzione

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

std :: find_if

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

effetti

Trova il primo elemento in un intervallo per il quale la funzione predicato `pred` restituisce true.

parametri

`first` => iteratore che punta all'inizio dell'intervallo `last` => iteratore che punta alla fine dell'intervallo `pred` => funzione predicato (restituisce vero o falso)

Ritorno

Un iteratore che punta al primo elemento all'interno dell'intervallo, la funzione predicato per cui predice restituisce true. L'iteratore punta a durare se val non viene trovato

Esempio

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


using namespace std;

/*
    define some functions to use as predicates
*/

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

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

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

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


int main()
{

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

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

```
// >= C++11

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

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

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


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



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

  return 0;
}
```

Produzione

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

std :: min_element

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

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

effetti

Trova l'elemento minimo in un intervallo

parametri

`first` - iteratore che punta all'inizio della gamma
`last` - iteratore che punta alla fine dell'intervallo `comp` - un puntatore di funzione o oggetto di funzione che accetta due argomenti e restituisce vero o falso indicando se l'argomento è inferiore all'argomento 2. Questa funzione non dovrebbe modificare gli input

Ritorno

Iteratore sull'elemento minimo dell'intervallo

Complessità

Lineare in uno in meno rispetto al numero di elementi confrontati.

Esempio

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

using namespace std;

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

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

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

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


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

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


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

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

  return 0;
}
```

Produzione

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

Uso di std :: nth_element per trovare la mediana (o altri quantili)

Lo `std::nth_element` algoritmo prende tre iteratori: un iteratore all'inizio, *n* ° posizione, e la fine. Una

volta che la funzione ritorna, il *n* elemento (dell'ordine) sarà il *n* ° elemento più piccolo. (La funzione ha sovraccarichi più elaborati, ad esempio alcuni che assumono funzioni di comparazione, vedi il link sopra per tutte le varianti.)

Nota Questa funzione è molto efficiente - ha una complessità lineare.

Per questo esempio, definiamo la mediana di una sequenza di lunghezza *n* come l'elemento che sarebbe in posizione ⌈n / 2⌉. Ad esempio, la mediana di una sequenza di lunghezza 5 è il terzo elemento più piccolo, e quindi è la mediana di una sequenza di lunghezza 6.

Per utilizzare questa funzione per trovare la mediana, possiamo usare quanto segue. Diciamo che iniziamo con

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

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

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

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

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

Per trovare il *p* esimo quantile , vorremmo cambiare alcune delle linee di cui sopra:

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

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

e cerca il quantile in posizione `pos` .

Leggi Algoritmi di libreria standard online: https://riptutorial.com/it/cplusplus/topic/3177/algoritmi-di-libreria-standard

Capitolo 3: Allineamento

introduzione

Tutti i tipi in C ++ hanno un allineamento. Questa è una restrizione sull'indirizzo di memoria che gli oggetti di quel tipo possono essere creati all'interno. Un indirizzo di memoria è valido per la creazione di un oggetto se la divisione di quell'indirizzo tramite l'allineamento dell'oggetto è un numero intero.

Gli allineamenti di tipo sono sempre una potenza di due (incluso 1).

Osservazioni

Lo standard garantisce quanto segue:

- Il requisito di allineamento di un tipo è un divisore delle sue dimensioni. Ad esempio, una classe con dimensione 16 byte potrebbe avere un allineamento di 1, 2, 4, 8 o 16, ma non 32. (Se i membri di una classe hanno solo 14 byte di dimensione, la classe deve avere un requisito di allineamento di 8, il compilatore inserirà 2 byte di riempimento per rendere la dimensione della classe uguale a 16.)

- Le versioni firmate e non firmate di un tipo intero hanno lo stesso requisito di allineamento.
- Un puntatore a `void` ha lo stesso requisito di allineamento di un puntatore al `char` .
- Le versioni cv-qualificate e cv-non qualificate di un tipo hanno lo stesso requisito di allineamento.

Si noti che mentre l'allineamento esiste in C ++ 03, non è stato fino a C ++ 11 che è diventato possibile interrogare l'allineamento (usando `alignof`) e l'allineamento di controllo (usando `alignas`).

Examples

Interrogare l'allineamento di un tipo

c ++ 11

Il requisito di allineamento di un tipo può essere interrogato utilizzando la parola chiave `alignof` come operatore unario. Il risultato è un'espressione costante di tipo `std::size_t` , *cioè* può essere valutata in fase di compilazione.

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

Uscita possibile

Il requisito di allineamento di int è: 4

Se applicato a un array, restituisce il requisito di allineamento del tipo di elemento. Se applicato a un tipo di riferimento, restituisce il requisito di allineamento del tipo di riferimento. (I riferimenti stessi non hanno allineamento, dal momento che non sono oggetti.)

Controllo dell'allineamento

C ++ 11

La parola chiave `alignas` può essere utilizzata per forzare una variabile, un membro di dati di classe, una dichiarazione o definizione di una classe, o una dichiarazione o una definizione di enum, per avere un particolare allineamento, se supportato. Si presenta in due forme:

- `alignas(x)` , dove `x` è un'espressione costante, dà all'entità l'allineamento `x` , se supportato.
- `alignas(T)` , dove `T` è un tipo, dà all'entità un allineamento uguale al requisito di allineamento di `T` , cioè `alignof(T)` , se supportato.

Se più `alignas` sono applicati alla stessa entità, si applica la più severa.

In questo esempio, è garantito che il buffer `buf` sia allineato in modo appropriato per contenere un oggetto `int` , anche se il suo tipo di elemento è `unsigned char` , che potrebbe avere un requisito di allineamento più debole.

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

`alignas` non può essere usato per dare a un tipo un allineamento più piccolo di quello che avrebbe il tipo senza questa dichiarazione:

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

`alignas` , quando viene fornita un'espressione costante intera, devono avere un allineamento valido. Gli allineamenti validi sono sempre poteri di due e devono essere maggiori di zero. I compilatori sono tenuti a supportare tutti gli allineamenti validi fino all'allineamento del tipo `std::max_align_t` . *Possono* supportano allineamenti grandi di questo, ma il supporto per l'allocazione di memoria per tali oggetti è limitata. Il limite superiore per gli allineamenti dipende dall'implementazione.

C ++ 17 offre un supporto diretto per l' `operator new` per l'allocazione della memoria per i tipi sovradimensionati.

Leggi Allineamento online: https://riptutorial.com/it/cplusplus/topic/9249/allineamento

Capitolo 4: Altri comportamenti non definiti in C ++

introduzione

Altri esempi su come C ++ può andare storto.

Continuazione da comportamento indefinito

Examples

Riferendosi a membri non statici negli elenchi di inizializzatori

Il riferimento a membri non statici negli elenchi di inizializzatori prima che il costruttore abbia iniziato l'esecuzione può provocare un comportamento indefinito. Questo risulta dal momento che non tutti i membri sono costruiti in questo momento. Dalla bozza standard:

> §12.7.1: per un oggetto con un costruttore non banale, facendo riferimento a qualsiasi membro non statico o classe base dell'oggetto prima che il costruttore inizi l'esecuzione, si ottiene un comportamento non definito.

Esempio

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
   int *p;
   X x;
   Y() : p(&x.j) { // undefined, x is not yet constructed
   }
};
```

Leggi Altri comportamenti non definiti in C ++ online:
https://riptutorial.com/it/cplusplus/topic/9885/altri-comportamenti-non-definiti-in-c-plusplus

Capitolo 5: Aritmetica in virgola mobile

Examples

I numeri in virgola mobile sono strani

Il primo errore commesso da quasi ogni programmatore presume che questo codice funzioni come previsto:

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

Il programmatore principiante assume che questo sommerà ogni singolo numero compreso tra `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99` , per ottenere il risultato `199` -la risposta matematicamente corretta.

Accadono due cose che rendono falso questo:

1. Il programma come scritto non conclude mai. `a` mai diventa uguale a `2` e il ciclo non termina mai.
2. Se riscriviamo la logica del loop per verificare `a < 2` , invece, il ciclo termina, ma il totale finisce per essere qualcosa di diverso da `199` . Sulle macchine compatibili con IEEE754, spesso si sommano invece a circa `201` .

La ragione per cui questo accade è che i **numeri in virgola mobile rappresentano approssimazioni dei loro valori assegnati** .

L'esempio classico è il seguente calcolo:

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

Sebbene ciò che noi programmatori vediamo sono tre numeri scritti in base10, ciò che vedono il compilatore (e l'hardware sottostante) sono numeri binari. Dato che `0.1` , `0.2` e `0.3` richiedono una perfetta divisione di `10` - che è abbastanza facile in un sistema di base 10, ma impossibile in un sistema di base 2 - questi numeri devono essere memorizzati in formati imprecisi, simili a come il numero `1/3` deve essere memorizzato nella forma imprecisa `0.333333333333333...` in base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111111011100110011001100110011001100110011001100110011001100110011010; //imperfect
```

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

Leggi Aritmetica in virgola mobile online: https://riptutorial.com/it/cplusplus/topic/5115/aritmetica-in-virgola-mobile

Capitolo 6: Array

introduzione

Le matrici sono elementi dello stesso tipo collocati in posizioni di memoria adiacenti. Gli elementi possono essere referenziati individualmente da un identificativo univoco con un indice aggiunto.

Ciò consente di dichiarare più valori variabili di un tipo specifico e di accedervi singolarmente senza dover dichiarare una variabile per ogni valore.

Examples

Dimensione array: digita sicuro al momento della compilazione.

```
#include <stddef.h>     // size_t, ptrdiff_t

//---------------------------------- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
    -> Size
{ return n; }


//---------------------------------- Usage:

#include <iostream>
using namespace std;
auto main()
    -> int
{
    int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
    Size const  n       = n_items( a );
    int         b[n]    = {};       // An array of the same size as a.

    (void) b;
    cout << "Size = " << n << "\n";
}
```

L'idioma C per la dimensione dell'array, `sizeof(a)/sizeof(a[0])` , accetta un puntatore come argomento e genererà quindi un risultato errato.

Per C ++ 11

usando C ++ 11 puoi fare:

```
std::extent<decltype(MyArray)>::value;
```

Esempio:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Fino a C ++ 17 (in uscita al momento della stesura di questo manuale) C ++ non aveva un linguaggio di base incorporato o un'utilità di libreria standard per ottenere la dimensione di un array, ma questo può essere implementato passando l'array *per riferimento* a un modello di funzione, come sopra riportati. Punto fine ma importante: il parametro size template è un `size_t` , in qualche modo incoerente con il tipo di risultato della funzione `Size` firmato, al fine di adattare il compilatore g ++ che a volte insiste su `size_t` per la corrispondenza dei template.

Con C ++ 17 e successivi si può invece usare `std::size` , che è specializzato per gli array.

Array raw di dimensioni dinamiche

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }

auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';

    delete[] a;
}
```

Un programma che dichiara un array `T a[n];` dove `n` è determinato a run-time, può compilare con alcuni compilatori che supportano *array di lunghezze variadici* C99 (VLA) come estensione del linguaggio. Ma i VLA non sono supportati dal C ++ standard. Questo esempio mostra come allocare manualmente un array di dimensioni dinamiche tramite una `new[]` -expression,

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

... quindi usalo e infine deallocalo tramite una `delete[]` -expression:

```
delete[] a;
```

L'array allocato qui ha valori indeterminati, ma può essere inizializzato a zero semplicemente aggiungendo una parentesi vuota `()` , come questo: `new int[n]()` . Più in generale, per tipo di oggetto arbitrario, esegue un'inizializzazione del *valore* .

Come parte di una funzione giù in una gerarchia di chiamate, questo codice non sarebbe eccezionalmente sicuro, poiché un'eccezione prima dell'espressione `delete[]` (e dopo la `new[]`) causerebbe una perdita di memoria. Un modo per risolvere questo problema è automatizzare la pulizia tramite, ad esempio, un puntatore intelligente `std::unique_ptr` . Ma un modo generalmente migliore per affrontarlo è usare semplicemente un `std::vector` : ecco a cosa serve `std::vector` .

Espansione dell'array di dimensioni dinamiche utilizzando std :: vector.

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` è un modello di libreria standard che fornisce la nozione di una matrice di dimensioni variabili. Si occupa di tutta la gestione della memoria e il buffer è contiguo, quindi un puntatore al buffer (ad esempio `&v[0]` o `v.data()`) può essere passato alle funzioni API che richiedono un array raw. Un `vector` può persino essere espanso in fase di esecuzione, ad esempio tramite la funzione membro `push_back` che aggiunge un elemento.

La complessità della sequenza di *n* operazioni `push_back` , inclusa la copia o lo spostamento coinvolti nelle espansioni di vettori, viene ammortizzata O (*n*). "Ammortizzato": in media.

Internamente questo è solitamente ottenuto dal *raddoppio* del vettore della sua dimensione del buffer, la sua capacità, quando è necessario un buffer più grande. Ad esempio, per un buffer che inizia come dimensione 1, e che viene ripetutamente raddoppiato come necessario per *n* = 17 chiamate `push_back` , ciò comporta operazioni di copia 1 + 2 + 4 + 8 + 16 = 31, che è inferiore a 2 × *n* = 34. E più in generale la somma di questa sequenza non può superare 2 × *n* .

Rispetto all'esempio di matrice raw di dimensioni dinamiche, questo codice basato su `vector` non richiede all'utente di fornire (e conoscere) il numero di elementi in anticipo. Invece il vettore viene semplicemente espanso secondo necessità, per ogni nuovo valore di articolo specificato dall'utente.

Una matrice di matrice raw di dimensioni fisse (ovvero una matrice raw 2D).

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Produzione:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

C ++ non supporta la sintassi speciale per l'indicizzazione di un array multidimensionale. Invece una tale matrice è vista come una matrice di matrici (possibilmente di matrici, e così via), e la normale notazione di indice singolo `[ i ]` è usata per ogni livello. Nell'esempio sopra `m[y]` riferisce alla riga `y` di `m` , dove `y` è un indice a base zero. Allora questa riga può essere indicizzato a sua volta, ad esempio `m[y][x]` , che si riferisce alla `x` th elemento - o colonna - di fila `y` .

Cioè l'ultimo indice varia più velocemente, e nella dichiarazione l'intervallo di questo indice, che qui è il numero di colonne per riga, è l'ultima e "più interna" dimensione specificata.

Poiché il C ++ non fornisce il supporto integrato per gli array di dimensioni dinamiche, oltre all'allocazione dinamica, una matrice di dimensioni dinamiche viene spesso implementata come classe. Quindi la notazione dell'indice matrice matrice raw `m[y][x]` ha un certo costo, sia esponendo l'implementazione (in modo che ad esempio una vista di una matrice trasposta diventa praticamente impossibile) o aggiungendo qualche sovraccarico e leggero inconveniente quando

viene fatto ritornando un oggetto proxy `operator[]` . E così la notazione di indicizzazione per una tale astrazione può e sarà di solito diversa, sia nell'aspetto che nell'ordine degli indici, ad esempio `m(x,y)` o `m.at(x,y)` o `m.item(x,y)` .

Una matrice di dimensioni dinamiche che utilizza std :: vector per l'archiviazione.

Sfortunatamente a partire dal C ++ 14 non esiste una classe di matrice di dimensioni dinamiche nella libreria standard C ++. Classi Matrix che supportano la dimensione dinamica sono comunque disponibili da un certo numero di librerie di 3 rd di partito, tra cui la libreria Boost Matrix (un sub-libreria all'interno della libreria Boost).

Se non vuoi una dipendenza da Boost o da qualche altra libreria, la matrice delle dimensioni dinamiche di un povero in C ++ è proprio come

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

... dove `vector` è `std::vector` . La matrice viene qui creata copiando un vettore di riga *n* volte in cui *n* è il numero di righe, qui 3. Ha il vantaggio di fornire la stessa notazione di indicizzazione `m[y][x]` come per una matrice di matrice raw di dimensioni fisse, ma è un po 'inefficiente perché implica un'allocazione dinamica per ogni riga, ed è un po' pericoloso perché è possibile ridimensionare inavvertitamente una riga.

Un approccio più sicuro ed efficiente consiste nell'utilizzare un singolo vettore come *memoria* per la matrice e mappare il codice del client (*x* , *y*) in un indice corrispondente in quel vettore:

```
// A dynamic size matrix using std::vector for storage.

//---------------------------------------- Machinery:
#include <algorithm>        // std::copy
#include <assert.h>         // assert
#include <initializer_list> // std::initializer_list
#include <vector>           // std::vector
#include <stddef.h>         // ptrdiff_t

namespace my {
    using Size = ptrdiff_t;
    using std::initializer_list;
    using std::vector;

    template< class Item >
    class Matrix
    {
    private:
        vector<Item>    items_;
        Size            n_cols_;

        auto index_for( Size const x, Size const y ) const
            -> Size
        { return y*n_cols_ + x; }

    public:
        auto n_rows() const -> Size { return items_.size()/n_cols_; }
```

```
        auto n_cols() const -> Size { return n_cols_; }

        auto item( Size const x, Size const y )
            -> Item&
        { return items_[index_for(x, y)]; }

        auto item( Size const x, Size const y ) const
            -> Item const&
        { return items_[index_for(x, y)]; }

        Matrix(): n_cols_( 0 ) {}

        Matrix( Size const n_cols, Size const n_rows )
            : items_( n_cols*n_rows )
            , n_cols_( n_cols )
        {}

        Matrix( initializer_list< initializer_list<Item> > const& values )
            : items_()
            , n_cols_( values.size() == 0? 0 : values.begin()->size() )
        {
            for( auto const& row : values )
            {
                assert( Size( row.size() ) == n_cols_ );
                items_.insert( items_.end(), row.begin(), row.end() );
            }
        }
    };
}  // namespace my

//---------------------------------------- Usage:
using my::Matrix;

auto some_matrix()
    -> Matrix<int>
{
    return
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };
}

#include <iostream>
#include <iomanip>
using namespace std;
auto main() -> int
{
    Matrix<int> const m = some_matrix();
    assert( m.n_cols() == 7 );
    assert( m.n_rows() == 3 );
    for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
    {
        for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
        {
            cout << setw( 4 ) << m.item( x, y );        // ← Note: not `m[y][x]`!
        }
        cout << '\n';
    }
}
```

Produzione:

```
  1   2   3   4   5   6   7
  8   9  10  11  12  13  14
 15  16  17  18  19  20  21
```

Il codice sopra riportato non è di livello industriale: è progettato per mostrare i principi di base e soddisfare le esigenze degli studenti che imparano C ++.

Ad esempio, è possibile definire overload `operator()` per semplificare la notazione dell'indicizzazione.

Inizializzazione di array

Un array è solo un blocco di posizioni di memoria sequenziali per un tipo specifico di variabile. Gli array sono allocati allo stesso modo delle variabili normali, ma con parentesi quadre aggiunte al suo nome `[]` che contengono il numero di elementi che si adattano alla memoria dell'array.

Il seguente esempio di un array utilizza la tip `int` , il nome della variabile `arrayOfInts` , e il numero di elementi `[5]` che l'array ha spazio per:

```
int arrayOfInts[5];
```

Un array può essere dichiarato e inizializzato allo stesso tempo in questo modo

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

Quando si inizializza una matrice elencando tutti i suoi membri, non è necessario includere il numero di elementi all'interno delle parentesi quadre. Sarà calcolato automaticamente dal compilatore. Nell'esempio seguente, è 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

È anche possibile inizializzare solo i primi elementi mentre si assegna più spazio. In questo caso, la definizione della lunghezza tra parentesi è obbligatoria. Quanto segue assegnerà un array di lunghezza 5 con inizializzazione parziale, il compilatore inizializza tutti gli elementi rimanenti con il valore standard del tipo di elemento, in questo caso zero.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Le matrici di altri tipi di dati di base possono essere inizializzate allo stesso modo.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize

char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

È anche importante notare che quando si accede agli elementi dell'array, l'indice (o la posizione) dell'elemento dell'array inizia da 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Leggi Array online: https://riptutorial.com/it/cplusplus/topic/3017/array

Capitolo 7: attributi

Sintassi

- [[dettagli]]: Semplice attributo senza argomento
- [[dettagli (argomenti)]]: Attributo con argomenti
- __attribute (dettagli): GCC / Clang / IBM non standard specifici
- __declspec (dettagli): specifico MSVC non standard

Examples

[[senza ritorno]]

C ++ 11

C ++ 11 ha introdotto l'attributo `[[noreturn]]` . Può essere usato per una funzione per indicare che la funzione non ritorna al chiamante eseguendo un'istruzione *return* o raggiungendo la fine se è body (è importante notare che questo non si applica alle funzioni `void` , poiché non tornare al chiamante, semplicemente non restituiscono alcun valore). Tale funzione può terminare chiamando `std::terminate` o `std::exit` o lanciando un'eccezione. Vale anche la pena notare che tale funzione può tornare eseguendo `longjmp` .

Ad esempio, la funzione seguente genera sempre un'eccezione o chiama `std::terminate` , quindi è un buon candidato per `[[noreturn]]` :

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

Questo tipo di funzionalità consente al compilatore di terminare una funzione senza un'istruzione return se sa che il codice non verrà mai eseguito. Qui, poiché la chiamata a `ownAssertFailureHandler` (definita sopra) nel codice seguente non tornerà mai più, il compilatore non ha bisogno di aggiungere il codice sotto quella chiamata:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
    }
```

```
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}
```

È un comportamento non definito se la funzione verrà effettivamente restituita, quindi non è consentito quanto segue:

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
        ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}
```

Nota che `[[noreturn]]` è usato principalmente nelle funzioni void. Tuttavia, questo non è un requisito, consentendo di utilizzare le funzioni nella programmazione generica:

```
template<class InconsistencyHandler>
double fortyTwoDivideBy(int i) {
    if (i == 0)
         i = InconsistencyHandler::correct(i);
    return 42. / i;
}

struct InconsistencyThrower {
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown
inconsistency"s); }
}

struct InconsistencyChangeToOne {
    static int correct(int i) { return 1; }
}

double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);
```

Le seguenti funzioni di libreria standard hanno questo attributo:

- std :: Abort
- std :: uscita
- std :: quick_exit
- std :: inaspettato
- std :: terminare
- std :: rethrow_exception
- std :: throw_with_nested
- std :: :: nested_exception rethrow_nested

[[sfumare]]

C ++ 17

Ogni volta che un `case` termina in un `switch` , il codice del caso successivo verrà eseguito. Quest'ultimo può essere prevenuto usando l'istruzione'break`. Dato che questo cosiddetto

comportamento anticaduta può introdurre bug quando non lo si intende, diversi compilatori e analizzatori statici forniscono un avvertimento su questo.

Da C ++ 17 in poi, è stato introdotto un attributo standard per indicare che l'avviso non è necessario quando il codice è destinato a cadere. I compilatori possono tranquillamente dare avvertimenti quando un caso è terminato senza `break` o `[[fallthrough]]` e ha almeno una dichiarazione.

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

Vedi la proposta per esempi più dettagliati su come `[[fallthrough]]` può essere usato.

[[deprecato]] e [[deprecato ("motivo")]]

C ++ 14

C ++ 14 ha introdotto un modo standard di deprecare le funzioni tramite attributi. `[[deprecated]]` può essere usato per indicare che una funzione è deprecata. `[[deprecated("reason")]]` consente di aggiungere un motivo specifico che può essere mostrato dal compilatore.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the
next release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

Questo attributo può essere applicato a:

- la dichiarazione di una classe
- un nome typedef
- una variabile
- un membro dati non statico
- una funzione
- un'enumerazione
- una specializzazione di modello

(riferimento alla bozza standard di c ++ 14 : 7.6.5 Attributo deprecato)

[[Nodiscard]]

C ++ 17

L'attributo `[[nodiscard]]` può essere utilizzato per indicare che il valore di ritorno di una funzione non deve essere ignorato quando si effettua una chiamata di funzione. Se il valore di ritorno viene ignorato, il compilatore dovrebbe dare un avvertimento su questo. L'attributo può essere aggiunto a:

- Una definizione di funzione
- Un tipo

Aggiungere l'attributo a un tipo ha lo stesso comportamento di aggiungere l'attributo a ogni singola funzione che restituisce questo tipo.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0);                    // Just to make comments clear!
    ++i;                               // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i;                               // i == 2
    onExit([&i]{ --i; });              // BUG: Reducing by 1 directly
                                       //      Compiler warning expected
    std::cout << i << std::end;        // Expected: 2, Real: 1
}
```

Vedi la proposta per esempi più dettagliati su come `[[nodiscard]]` può essere usato.

Nota: i dettagli di implementazione di `Finally` / `onExit` sono omessi nell'esempio, vedi Finally / ScopeExit .

[[Maybe_unused]]

L'attributo `[[maybe_unused]]` viene creato per indicare nel codice che alcune logiche potrebbero non essere utilizzate. Questo, se spesso collegato alle condizioni del preprocessore dove potrebbe essere utilizzato o non può essere utilizzato. Poiché i compilatori possono dare avvertimenti su variabili non utilizzate, questo è un modo per sopprimerle indicando l'intenzione.

Un tipico esempio di variabili che sono necessarie nelle build di debug mentre non necessarie nella produzione sono valori di ritorno che indicano il successo. Nelle build di debug, la condizione dovrebbe essere asserita, sebbene in produzione questi assert siano stati rimossi.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the
map
```

Un esempio più complesso è un diverso tipo di funzioni di aiuto che si trovano in uno spazio dei nomi senza nome. Se queste funzioni non vengono utilizzate durante la compilazione, un

compilatore potrebbe dare un avvertimento su di esse. Idealmente ti piacerebbe proteggerli con gli stessi tag del preprocessore del chiamante, anche se questo potrebbe diventare complesso l'attributo `[[maybe_unused]]` è un'alternativa più `[[maybe_unused]]` .

```
namespace {
    [[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
    // TODO: Reuse this on BSD, MAC ...
    [[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#if OS == "WINDOWS"
      return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
      return createLinuxConfigFilePath(relativePath);
#else
#error "OS is not yet supported"
#endif
}
```

Vedi la proposta per esempi più dettagliati su come `[[maybe_unused]]` può essere usato.

Leggi attributi online: https://riptutorial.com/it/cplusplus/topic/5251/attributi

Capitolo 8: auto

Osservazioni

La parola chiave `auto` è un typename che rappresenta un tipo dedotto automaticamente.

Era già una parola chiave riservata in C ++ 98, ereditata da C. Nelle vecchie versioni di C ++, poteva essere usata per dichiarare esplicitamente che una variabile ha una durata di archiviazione automatica:

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

Quel vecchio significato è ora rimosso.

Examples

Campione automatico di base

La parola chiave `auto` fornisce l'auto-deduzione del tipo di una variabile.

È particolarmente utile quando si ha a che fare con nomi di caratteri lunghi:

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

con loop basati su range :

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

con lambda :

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

per evitare la ripetizione del tipo:

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

per evitare copie sorprendenti e non necessarie:

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

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

Il motivo della copia è che il tipo restituito è in realtà `std::pair<const int,float>` !

Modelli auto ed espressioni

`auto` può anche causare problemi quando entrano in gioco i modelli di espressione:

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

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

Il motivo è che l' `operator*` su `valarray` fornisce un oggetto proxy che fa riferimento al `valarray` come mezzo di valutazione lazy. Usando l' `auto` , stai creando un riferimento ciondolante. Invece di `mult` aveva restituito uno `std::valarray<int>` , quindi il codice avrebbe sicuramente stampato 3.

auto, const e riferimenti

La parola chiave `auto` per sé rappresenta un tipo di valore, simile a `int` o `char` . Può essere modificato con la parola chiave `const` e il simbolo `&` per rappresentare rispettivamente un tipo const o un tipo di riferimento. Questi modificatori possono essere combinati.

In questo esempio, `s` è un tipo di valore (il suo tipo sarà dedotto come `std::string`), quindi ogni iterazione del ciclo `for` *copia* una stringa dal vettore in `s` .

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

Se il corpo del loop modifica `s` (ad esempio chiamando `s.append(" and stuff")`), solo questa copia verrà modificata, non il membro originale delle `strings` .

D'altra parte, se `s` è dichiarato con `auto&` sarà un tipo di riferimento (dedotto per essere `std::string&`), così ad ogni iterazione del ciclo verrà assegnato un *riferimento* ad una stringa nel vettore:

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

Nel corpo di questo ciclo, le modifiche a `s` influenzeranno direttamente l'elemento delle `strings` cui fa riferimento.

Infine, se `s` è dichiarato `const auto&` , sarà un tipo di riferimento const, nel senso che ad ogni iterazione del ciclo verrà assegnato un *riferimento const* a una stringa nel vettore:

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

All'interno del corpo di questo ciclo, `s` non può essere modificato (cioè non può essere chiamato alcun metodo non const).

Quando si utilizza l' `auto` con cicli basati `for` intervalli, è generalmente buona norma usare `const auto&` se il corpo del loop non modificherà la struttura in loop, poiché evita copie non necessarie.

Tipo di ritorno finale

`auto` viene utilizzato nella sintassi per il tipo di ritorno finale:

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

che è equivalente a

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

Molto utile combinato con `decltype` per usare i parametri invece di `std::declval<T>` :

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

Lambda generico (C ++ 14)

C ++ 14

C ++ 14 consente di utilizzare `auto` argomento `auto` in lambda

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

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

Quella lambda è per lo più equivalente a

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

e poi

```
lambda print;

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

oggetti auto e proxy

A volte l' `auto` può comportarsi in modo non proprio come previsto da un programmatore. Il tipo deduce l'espressione, anche quando la deduzione del tipo non è la cosa giusta da fare.

Ad esempio, quando gli oggetti proxy sono utilizzati nel codice:

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

Qui `flag` sarebbe `bool` , ma `std::vector<bool>::reference` , poiché per la specializzazione `bool` del `vector` template l' `operator []` restituisce un oggetto proxy con `operator bool` conversione definito.

Quando `flags.push_back(true)` modifica il contenitore, questo pseudo-riferimento potrebbe finire in sospeso, facendo riferimento a un elemento che non esiste più.

Rende anche possibile la prossima situazione:

```
void foo(bool b);

std::vector<bool> getFlags();

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

Il `vector` viene scartato immediatamente, quindi `flag` è uno pseudo-riferimento a un elemento che è stato scartato. La chiamata a `foo` richiama un comportamento indefinito.

In casi come questo è possibile dichiarare una variabile con `auto` e inizializzarla eseguendo il casting sul tipo che si desidera dedurre:

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

ma a quel punto, semplicemente sostituire l' `auto` con il `bool` ha più senso.

Un altro caso in cui gli oggetti proxy possono causare problemi sono i modelli di espressione . In tal caso, i modelli a volte non sono progettati per durare oltre l'espressione completa corrente per motivi di efficienza e l'utilizzo dell'oggetto proxy sul prossimo causa un comportamento non definito.

Leggi auto online: https://riptutorial.com/it/cplusplus/topic/2421/auto

Capitolo 9: C ++ 11 Modello di memoria

Osservazioni

Diversi thread che tentano di accedere alla stessa posizione di memoria partecipano a una *corsa di dati* se almeno una delle operazioni è una modifica (nota anche come *operazione di memorizzazione*). Queste *razze di dati* causano un *comportamento indefinito* . Per evitarli è necessario impedire a questi thread di eseguire contemporaneamente operazioni in conflitto.

I primitivi di sincronizzazione (mutex, sezione critica e simili) possono proteggere tali accessi. Il modello di memoria introdotto in C ++ 11 definisce due nuovi modi portatili per sincronizzare l'accesso alla memoria in ambiente multi-thread: operazioni atomiche e recinzioni.

Operazioni atomiche

È ora possibile leggere e scrivere in una determinata posizione di memoria mediante l'utilizzo di operazioni di *caricamento atomico* e di *immagazzinamento atomico* . Per comodità questi sono inclusi nella classe template `std::atomic<t>` . Questa classe racchiude un valore di tipo `t` ma questa volta i *carichi* e i *depositi* sull'oggetto sono atomici.

Il modello non è disponibile per tutti i tipi. I tipi disponibili sono specifici dell'implementazione, ma di solito includono la maggior parte (o tutti) i tipi integrali disponibili e i tipi di puntatore. Quindi `std::atomic<unsigned>` e `std::atomic<std::vector<foo> *>` dovrebbero essere disponibili, mentre `std::atomic<std::pair<bool,char>>` molto probabilmente non lo sarà.

Le operazioni atomiche hanno le seguenti proprietà:

- Tutte le operazioni atomiche possono essere eseguite simultaneamente da più thread senza causare un comportamento indefinito.
- Un *carico atomico* vedrà sia il valore iniziale con cui è stato costruito l'oggetto atomico, sia il valore scritto su di esso tramite un'operazione di *immagazzinamento atomico* .
- *Le memorie atomiche* sullo stesso oggetto atomico vengono ordinate allo stesso modo in tutti i thread. Se un thread ha già visto il valore di un'operazione dell'archivio *atomico* , le successive operazioni di *caricamento atomico* vedranno lo stesso valore o il valore memorizzato dall'operazione successiva *dell'archivio atomico* .
- *Le operazioni atomiche di lettura-modifica-scrittura* consentono al *carico atomico* e *all'archivio atomico* di accedere senza altri *depositi atomici* nel mezzo. Ad esempio si può incrementare atomicamente un contatore da più thread e nessun incremento andrà perso indipendentemente dalla contesa tra i thread.
- Le operazioni atomiche ricevono un parametro facoltativo `std::memory_order` che definisce quali proprietà aggiuntive l'operazione ha rispetto ad altre posizioni di memoria.

std :: memory_order	Senso
`std::memory_order_relaxed`	senza ulteriori restrizioni

std :: memory_order	Senso
`std::memory_order_release` → `std::memory_order_acquire`	se `load-acquire` vede il valore memorizzato da `store-release` quindi memorizza il *sequenziamento prima che* lo `store-release` avvenga prima che i carichi siano sequenziati dopo che il *carico acquisisce*
`std::memory_order_consume`	come `memory_order_acquire` ma solo per carichi dipendenti
`std::memory_order_acq_rel`	combina `load-acquire` e `store-release`
`std::memory_order_seq_cst`	consistenza sequenziale

Questi tag di ordine di memoria consentono tre diverse discipline di ordinamento della memoria: *coerenza sequenziale* , *rilassato* e *acquisizione-acquisizione* con il suo *rilascio-consumo di* fratello.

Consistenza sequenziale

Se non viene specificato un ordine di memoria per un'operazione atomica, l'ordine assume come impostazione predefinita la *coerenza sequenziale* . Questa modalità può anche essere selezionata in modo esplicito taggando l'operazione con `std::memory_order_seq_cst` .

Con questo ordine nessuna operazione di memoria può attraversare l'operazione atomica. Tutte le operazioni di memoria sequenziate prima dell'operazione atomica avvengono prima dell'operazione atomica e l'operazione atomica avviene prima di tutte le operazioni di memoria che vengono sequenziate dopo di essa. Questa modalità è probabilmente la più facile da ragionare, ma porta anche alla peggiore penalizzazione delle prestazioni. Inoltre impedisce tutte le ottimizzazioni del compilatore che potrebbero altrimenti tentare di riordinare le operazioni dopo l'operazione atomica.

Ordinamento rilassato

L'opposto alla *coerenza sequenziale* è l'ordine di memoria *rilassato* . È selezionato con il tag `std::memory_order_relaxed` . L'operazione atomica rilassata non imporrà restrizioni sulle altre operazioni di memoria. L'unico effetto che rimane è che l'operazione è di per sé ancora atomica.

Rilascio-Acquisisci ordini

Un'operazione di *archivio atomico* può essere contrassegnata con `std::memory_order_release` e un'operazione di *caricamento atomico* può essere contrassegnata con `std::memory_order_acquire` . La prima operazione è chiamata *(atomica) store-release* mentre la seconda è chiamata *(atomic) load-acquire* .

Quando *load-acquire* vede il valore scritto da un *store-release* accade quanto segue: tutte le operazioni di memorizzazione sequenziate prima che lo *store-release* diventino visibili a (*accadono prima*) le operazioni di caricamento che sono sequenziate dopo l' *acquisizione* del *carico* .

Le operazioni atomiche di lettura-modifica-scrittura possono anche ricevere il tag cumulativo `std::memory_order_acq_rel` . Questo rende la porzione *carico atomica* dell'operazione un *atomico carico acquisiscono* mentre la porzione *negozio atomico* diventa *deposito release atomico.*

Il compilatore non può spostare le operazioni del negozio dopo un'operazione *di rilascio dello store atomico* . Inoltre, non è consentito spostare le operazioni di carico prima che il *carico atomico acquisisca* (o *carichi-consumi*).

Si noti inoltre che non esiste il *rilascio del carico atomico* o *l'acquisizione del negozio atomico* . Il tentativo di creare tali operazioni li rende operazioni *rilassate* .

Rilascio-Consuma ordine

Questa combinazione è simile a *release-acquire* , ma questa volta il *carico atomico* viene etichettato con `std::memory_order_consume` e diventa *(atomico) carico-consumo* . Questa modalità è la stessa di *Release-Acquisisci* con la sola differenza che tra le operazioni di caricamento in sequenza dopo il *carico-consumano* solo queste, in base al valore caricato dal *carico-consumo,* vengono ordinate.

Recinzioni

Le fence consentono inoltre di ordinare le operazioni di memoria tra i thread. Una recinzione è una barriera di rilascio o un recinto.

Se un recinto di rilascio si verifica prima di un recinto di acquisizione, memorizza i sequenziati prima che il recinto di rilascio sia visibile ai carichi sequenziati dopo il recinto di acquisizione. Per garantire che il recinto di rilascio avvenga prima del recinto di acquisizione, si possono usare altre primitive di sincronizzazione che includono operazioni atomiche rilassate.

Examples

Hai bisogno di un modello di memoria

```
int x, y;
bool ready = false;

void init()
{
  x = 2;
  y = 3;
  ready = true;
}
void use()
{
  if (ready)
    std::cout << x + y;
}
```

Un thread chiama la funzione `init()` mentre un altro thread (o gestore di segnale) chiama la

funzione `use()` . Ci si potrebbe aspettare che la funzione `use()` stampi `5` o non faccia nulla. Questo potrebbe non essere sempre il caso per diversi motivi:

- La CPU può riordinare le scritture che si verificano in `init()` modo che il codice che esegue effettivamente possa apparire come:

```
void init()
{
  ready = true;
  x = 2;
  y = 3;
}
```

- La CPU può riordinare le letture che si verificano in `use()` modo che il codice effettivamente eseguito possa diventare:

```
void use()
{
  int local_x = x;
  int local_y = y;
  if (ready)
    std::cout << local_x + local_y;
}
```

- Un compilatore C ++ ottimizzante può decidere di riordinare il programma in modo simile.

Tale riordino non può cambiare il comportamento di un programma in esecuzione in thread singolo perché un thread non può intercalare le chiamate a `init()` e `use()` . D'altra parte in una impostazione multi-threaded un thread può vedere parte delle scritture eseguite dall'altro thread dove può accadere che `use()` possa vedere `ready==true` e garbage in `x` o `y` o entrambi.

Il modello di memoria C ++ consente al programmatore di specificare quali operazioni di riordino sono consentite e quali no, in modo che anche un programma multi-thread possa comportarsi come previsto. L'esempio sopra può essere riscritto in modo thread-safe come questo:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  ready.store(true, std::memory_order_release);
}
void use()
{
  if (ready.load(std::memory_order_acquire))
    std::cout << x + y;
}
```

Qui `init()` esegue un'operazione di *rilascio dello store atomico* . Questo non solo memorizza il valore `true` in `ready` , ma dice al compilatore che non può spostare questa operazione prima di scrivere operazioni *sequenziate prima di* esso.

La funzione `use()` esegue un'operazione *di acquisizione del carico atomico* . Legge il valore corrente di `ready` e proibisce inoltre al compilatore di posizionare le operazioni di lettura che vengono *sequenziate dopo* che si sono *verificate prima dell'acquisizione del carico atomico* .

Queste operazioni atomiche fanno sì che il compilatore metta tutte le istruzioni hardware necessarie per informare la CPU di astenersi dai riordini indesiderati.

Poiché il *rilascio dell'archivio atomico si* trova nella stessa posizione di memoria *dell'acquisizione del carico atomico* , il modello di memoria stabilisce che se l'operazione di *acquisizione* del *carico* vede il valore scritto dall'operazione *di rilascio* dello *store* , quindi tutte le scritture eseguite da `init()` ' Il thread di s prima di quello *store-release* sarà visibile ai carichi che il thread di `use()` esegue dopo il suo *caricamento-acquisizione* . Questo è se `use()` vede `ready==true` , allora è garantito vedere `x==2 y==3` .

Si noti che il compilatore e la CPU possono ancora scrivere su `y` prima di scrivere in `x` , e allo stesso modo le letture di queste variabili in `use()` possono avvenire in qualsiasi ordine.

Esempio di recinzione

L'esempio sopra può anche essere implementato con recinzioni e operazioni atomiche rilassate:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  atomic_thread_fence(std::memory_order_release);
  ready.store(true, std::memory_order_relaxed);
}
void use()
{
  if (ready.load(std::memory_order_relaxed))
  {
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
  }
}
```

Se l'operazione di caricamento atomica vede il valore scritto dal negozio atomico poi il negozio avviene prima del carico, e così anche le recinzioni: la recinzione rilascio avviene prima della recinzione acquisiscono rendendo le scritture nella `x` ed `y` che precedono la recinzione rilascio di diventare visibile alla dichiarazione `std::cout` che segue la recinzione acquisita.

Un recinto può essere utile se può ridurre il numero complessivo di operazioni di acquisizione, rilascio o altre operazioni di sincronizzazione. Per esempio:

```
void block_and_use()
{
  while (!ready.load(std::memory_order_relaxed))
    ;
```

```
  atomic_thread_fence(std::memory_order_acquire);
  std::cout << x + y;
}
```

La funzione `block_and_use()` gira fino a quando il flag di `ready` viene impostato con l'aiuto del carico atomico rilassato. Quindi viene utilizzata una singola fence per fornire l'ordine di memoria necessario.

Leggi C ++ 11 Modello di memoria online: https://riptutorial.com/it/cplusplus/topic/7975/c-plusplus-11-modello-di-memoria

Capitolo 10: C incompatibilità

introduzione

Questo descrive quale codice C si romperà in un compilatore C ++.

Examples

Parole chiave riservate

Il primo esempio sono le parole chiave che hanno uno scopo speciale in C ++: il seguente è legale in C, ma non in C ++.

```
int class = 5
```

Questi errori sono facili da correggere: basta rinominare la variabile.

Puntatori debolmente tipizzati

In C, i puntatori possono essere espressi su un `void*` , che richiede un cast esplicito in C ++. Quanto segue è illegale in C ++, ma legale in C:

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

L'aggiunta di un cast esplicito rende questo lavoro, ma può causare ulteriori problemi.

goto o cambia

In C ++, non puoi saltare le inizializzazioni con `goto` o `switch` . Quanto segue è valido in C, ma non in C ++:

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

Questi bug possono richiedere una riprogettazione.

Leggi C incompatibilità online: https://riptutorial.com/it/cplusplus/topic/9645/c-incompatibilita

Capitolo 11: Campi bit

introduzione

I campi di bit racchiudono saldamente le strutture C e C ++ per ridurre le dimensioni. Ciò sembra indolore: specificare il numero di bit per i membri e il compilatore esegue il lavoro di bit di co-mingling. La restrizione è l'impossibilità di prendere l'indirizzo di un membro di campo bit, poiché è memorizzato insieme. `sizeof()` è anche non consentito.

Il costo dei campi bit è un accesso più lento, poiché la memoria deve essere recuperata e le operazioni bit a bit vengono applicate per estrarre o modificare i valori dei membri. Queste operazioni si aggiungono anche alla dimensione dell'eseguibile.

Osservazioni

Quanto sono costose le operazioni bit a bit? Supponi una semplice struttura di campo non bit:

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

In qualche secondo codice:

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

Se `sizeof (unsigned) == 4` , allora x viene memorizzato all'inizio della struttura, e y è memorizzato in 4 byte. Il codice di assembly generato può essere simile a:

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

Questo è semplice perché x non è mescolato a y. Ma immagina di ridefinire la struttura con i campi di bit:

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

Sia `x` che `y` saranno allocati a 4 bit, condividendo un singolo byte. La struttura occupa quindi 1 byte, invece di 8. Considerare l'assembly per impostare `y` ora, assumendo che finisca nel nibble superiore:

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

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

Questo può essere un buon compromesso se abbiamo migliaia o milioni di queste strutture, e aiuta a mantenere la memoria nella cache o impedisce lo swapping, o potrebbe gonfiare l'eseguibile per peggiorare questi problemi e rallentare l'elaborazione. Come con tutte le cose, usa il buon senso.

Uso del driver di dispositivo: evitare campi di bit come strategia di implementazione intelligente per i driver di dispositivo. I layout di archiviazione dei campi a bit non sono necessariamente coerenti tra i compilatori, rendendo tali implementazioni non portabili. La lettura-modifica-scrittura per impostare valori non può fare ciò che i dispositivi si aspettano, causando comportamenti imprevisti.

Examples

Dichiarazione e uso

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

Qui, ciascuno di questi due campi occuperà 1 bit in memoria. È specificato da `: 1` espressione dopo i nomi delle variabili. Il tipo base del campo di bit può essere qualsiasi tipo integrale (int a 8 bit int a 64 bit). Si consiglia di utilizzare il tipo `unsigned` , altrimenti potrebbero verificarsi sorprese.

Se sono necessari più bit, sostituire "1" con il numero di bit richiesto. Per esempio:

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

L'intera struttura utilizza solo 22 bit, e con normali impostazioni del compilatore, `sizeof` questa struttura sarebbe 4 byte.

L'utilizzo è piuttosto semplice. Basta dichiarare la variabile e usarla come una struttura ordinaria.

```
Date d;

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

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

```
        "Day:" << d.Day << std::endl;
```

Leggi Campi bit online: https://riptutorial.com/it/cplusplus/topic/2710/campi-bit

Capitolo 12: Categorie di valore

Examples

Significato della categoria di valore

Alle espressioni in C ++ viene assegnata una particolare categoria di valore, in base al risultato di tali espressioni. Le categorie di valori per le espressioni possono influire sulla risoluzione di sovraccarico della funzione C ++.

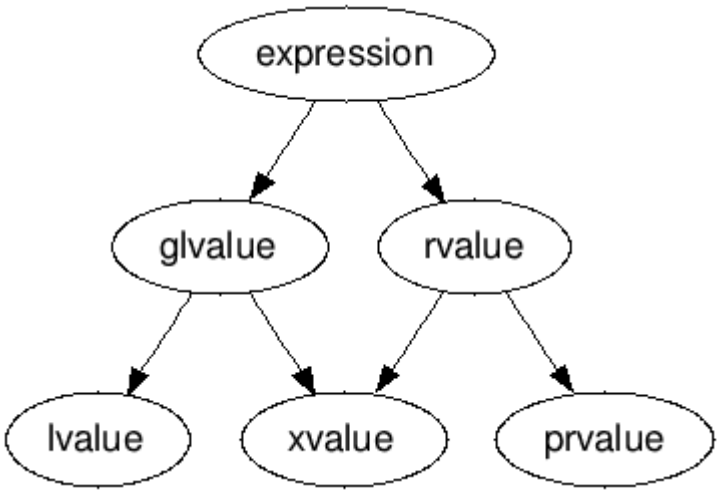
Le categorie valore determinano due proprietà importanti ma separate su un'espressione. Una proprietà è se l'espressione ha identità. Un'espressione ha identità se fa riferimento a un oggetto che ha un nome di variabile. Il nome della variabile potrebbe non essere coinvolto nell'espressione, ma l'oggetto può ancora averne uno.

L'altra proprietà è se è legale spostarsi implicitamente dal valore dell'espressione. O più nello specifico, se l'espressione, se usata come parametro di funzione, si legherà ai tipi di parametri di valore r o meno.

C ++ definisce 3 categorie di valori che rappresentano la combinazione utile di queste proprietà: lvalue (espressioni con identità ma non mobili da), xvalue (espressioni con identità che sono mobili da) e prvalore (espressioni senza identità che sono mobili da). C ++ non ha espressioni che non hanno identità e non possono essere spostate da.

C ++ definisce altre due categorie di valori, ciascuna basata unicamente su una di queste proprietà: glvalue (espressioni con identità) e rvalue (espressioni che possono essere spostate da). Questi agiscono come raggruppamenti utili delle categorie precedenti.

Questo grafico serve come illustrazione:



prvalue

Un'espressione prvalore (pure-rvalue) è un'espressione priva di identità, la cui valutazione viene in

genere utilizzata per inizializzare un oggetto e da cui può essere implicitamente spostato. Questi includono, ma non sono limitati a:

- Espressioni che rappresentano oggetti temporanei, come `std::string("123")` .
- Una espressione di chiamata di funzione che non restituisce un riferimento
- Un letterale (*tranne* un letterale stringa - quelli sono lvalue), tale è `1` , `true` , `0.5f` o `'a'`
- Un'espressione lambda

L'indirizzo incorporato dell'operatore (`&`) non può essere applicato su queste espressioni.

xValue

Un'espressione xvalue (valore eXpiring) è un'espressione che ha identità e rappresenta un oggetto da cui può essere implicitamente spostato. L'idea generale con le espressioni xvalue è che l'oggetto che rappresentano sarà presto distrutto (da qui la parte "eXpiring"), e quindi spostarsi implicitamente da esse va bene.

Dato:

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

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

lvalue

Un'espressione lvalue è un'espressione che ha identità, ma non può essere spostata implicitamente da. Tra queste sono espressioni che consistono in un nome di variabile, nome di funzione, espressioni che sono usi di operatore di deferenziazione ed espressioni che si riferiscono a riferimenti di lvalue.

Il lvalue tipico è semplicemente un nome, ma i lvalue possono venire anche in altri gusti:

```
struct X { ... };

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

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

Inoltre, mentre la maggior parte dei valori letterali (ad es. `4` , `'x'` , ecc.) Sono prvalues, i letterali stringa sono lvalue.

glvalue

Un'espressione glvalue (a "lvalue generalizzato") è qualsiasi espressione che abbia identità, indipendentemente dal fatto che possa essere spostata o meno. Questa categoria include lvalue (espressioni che hanno identità ma non possono essere spostate da) e xvalues (espressioni che hanno identità e possono essere spostate da), ma esclude prvalues (espressioni senza identità).

Se un'espressione ha un *nome* , è un glivalue:

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

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

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

rvalue

Un'espressione rvalue è qualsiasi espressione che può essere implicitamente spostata da, indipendentemente dal fatto che abbia identità.

Più precisamente, le espressioni rvalue possono essere utilizzate come argomento di una funzione che accetta un parametro di tipo `T &&` (dove `T` è il tipo di `expr`). *Solo le* espressioni rvalue possono essere date come argomenti a tali parametri di funzione; se viene utilizzata un'espressione non rvalore, la risoluzione di sovraccarico selezionerà qualsiasi funzione che non utilizza un parametro di riferimento rvalue. E se nessuno esiste, allora si ottiene un errore.

La categoria delle espressioni rvalue include tutte le espressioni xvalue e prvalue e solo quelle espressioni.

La funzione di libreria standard `std::move` esiste per trasformare esplicitamente un'espressione non rvalore in un valore rvalue. Più specificamente, trasforma l'espressione in un valore x, poiché anche se prima era un'espressione di prvalore priva di identità, passandola come parametro a `std::move` , ottiene l'identità (il nome del parametro della funzione) e diventa un valore x.

Considera quanto segue:

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

str = std::string("new value");               //4
std::string &&str_ref = std::move(str);       //5
std::string test3(str_ref);                   //6
```

`std::string` ha un costruttore che accetta un singolo parametro di tipo `std::string&&` ,

comunemente chiamato "costruttore di mosse". Tuttavia, la categoria di valore dell'espressione `str` non è un valore di rvalue (in particolare è un lvalue), quindi non può chiamare quel sovraccarico del costruttore. Invece, chiama `const std::string&` overload, il costruttore di copie.

La linea 3 cambia le cose. Il valore di ritorno di `std::move` è un `T&&` , dove `T` è il tipo base del parametro passato. Quindi `std::move(str)` restituisce `std::string&&` . Una chiamata di funzione il cui valore di ritorno è un riferimento di rvalue è un'espressione di valore (in particolare un valore x), quindi può chiamare il costruttore di movimento di `std::string` . Dopo la riga 3, `str` è stato spostato da (chi è il contenuto non è ora definito).

La riga 4 passa temporaneamente all'operatore di assegnazione di `std::string` . Questo ha un sovraccarico che accetta uno `std::string&&` . L'espressione `std::string("new value")` è un'espressione rvalue (in particolare un valore nominale), quindi può chiamare tale overload. Pertanto, il temporaneo viene spostato in `str` , sostituendo i contenuti indefiniti con contenuti specifici.

La riga 5 crea un riferimento `str_ref` denominato `str_ref` che fa riferimento a `str` . È qui che le categorie di valore si confondono.

Vedi, mentre `str_ref` è un riferimento di rvalue a `std::string` , la categoria di valore dell'espressione `str_ref` *non è un valore rvalore* . È un'espressione lvalue. Sì davvero. Per questo `str_ref` , non è possibile chiamare il costruttore di move di `std::string` con l'espressione `str_ref` . La riga 6 *copia* quindi il valore di `str` in `test3` .

Per spostarlo, dovremmo impiegare di nuovo `std::move` .

Leggi Categorie di valore online: https://riptutorial.com/it/cplusplus/topic/763/categorie-di-valore

Capitolo 13: Classi / Strutture

Sintassi

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

Osservazioni

Si noti che l' ***unica*** differenza tra le parole chiave `struct` e `class` è che, per impostazione predefinita, le variabili membro, le funzioni membro e le classi base di una `struct` sono `public` , mentre in una `class` sono `private` . I programmatori C ++ tendono a chiamarlo classe se ha costruttori e distruttori e la capacità di imporre i propri invarianti; o una struttura se è solo una semplice raccolta di valori, ma il linguaggio C ++ non fa alcuna distinzione.

Examples

Nozioni di base sulla classe

Una *classe* è un tipo definito dall'utente. Una classe viene introdotta con la parola chiave `class` , `struct` o `union` . Nell'uso colloquiale, il termine "classe" di solito si riferisce solo a classi non sindacali.

Una classe è una raccolta di *membri* della *classe* , che possono essere:

- variabili membro (chiamate anche "campi"),
- funzioni membro (chiamate anche "metodi"),
- tipi di membri o typedef (ad esempio "classi nidificate"),
- modelli di membri (di qualsiasi tipo: variabile, funzione, modello di classe o alias)

Le parole chiave `class` e `struct` , chiamate *chiavi di classe* , sono ampiamente intercambiabili, tranne per il fatto che l'identificatore di accesso predefinito per membri e basi è "privato" per una classe dichiarata con la chiave di `class` e "pubblica" per una classe dichiarata con la chiave `struct` o `union` (vedi modificatori di accesso).

Ad esempio, i seguenti frammenti di codice sono identici:

```
struct Vector
{
    int x;
    int y;
    int z;
};
// are equivalent to
```

```
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

Dichiarando una classe è stato aggiunto un nuovo tipo al tuo programma ed è possibile creare un'istanza di oggetti di quella classe

```
Vector my_vector;
```

Si accede ai membri di una classe usando la sintassi del punto.

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

Specifier di accesso

Ci sono tre parole chiave che fungono da **specificatori di accesso** . Questi limitano l'accesso ai membri della classe seguendo l'identificatore, fino a quando un altro specificatore cambia nuovamente il livello di accesso:

Parola chiave	Descrizione
`public`	Tutti hanno accesso
`protected`	Solo la classe stessa, le classi derivate e gli amici hanno accesso
`private`	Solo la classe stessa e gli amici hanno accesso

Quando il tipo viene definito utilizzando la parola chiave `class` , l'identificatore di accesso predefinito è `private` , ma se il tipo è definito utilizzando la parola chiave `struct` , lo specificatore di accesso predefinito è `public` :

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

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

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

Gli specificatori di accesso sono principalmente utilizzati per limitare l'accesso ai campi e ai metodi interni e impongono al programmatore di utilizzare un'interfaccia specifica, ad esempio per forzare l'uso di getter e setter invece di fare riferimento direttamente a una variabile:

```
class MyClass {

public: /* Methods: */

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

private: /* Fields: */

    int m_x;

};
```

L'utilizzo `protected` è utile per consentire a determinate funzionalità del tipo di essere accessibili solo alle classi derivate, ad esempio, nel seguente codice, il metodo `calculateValue()` è accessibile solo alle classi derivanti dalla classe base `Plus2Base` , come ad esempio `FortyTwo` :

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

Si noti che la parola chiave `friend` può essere utilizzata per aggiungere eccezioni di accesso a funzioni o tipi per l'accesso a membri protetti e privati.

Le parole chiave `public` , `protected` e `private` possono anche essere utilizzate per concedere o limitare l'accesso ai sottooggetti di classe base. Vedi l'esempio dell'Eredità .

Eredità

Le classi / le strutture possono avere relazioni di ereditarietà.

Se una classe / struct `B` eredita da una classe / struct `A` , ciò significa che `B` ha come padre `A` Diciamo che `B` è una classe / struct derivata da `A` , e `A` è la classe base / struct.

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

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

Esistono 3 forme di ereditarietà per una classe / struttura:

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

Si noti che l'ereditarietà predefinita è la stessa della visibilità predefinita dei membri: `public` se si utilizza la parola chiave `struct` e `private` per la parola chiave `class` .

È anche possibile che una `class` derivi da una `struct` (o viceversa). In questo caso, l'ereditarietà predefinita è controllata dal figlio, quindi una `struct` che deriva da una `class` verrà predefinita per ereditarietà pubblica e una `class` che deriva da una `struct` avrà un'ereditarietà privata per impostazione predefinita.

eredità `public` :

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

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

eredità `private` :

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

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

eredità `protected` :

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

```
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

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

Si noti che, sebbene l'ereditarietà `protected` sia consentita, l'uso effettivo di esso è raro. Un'istanza di come l'ereditarietà `protected` viene utilizzata nell'applicazione è nella specializzazione parziale della classe base (di solito indicata come "polimorfismo controllato").

Quando OOP era relativamente nuovo, l'eredità (pubblica) veniva spesso detta per modellare una relazione "IS-A". In altre parole, l'ereditarietà pubblica è corretta solo se un'istanza della classe derivata *è anche* un'istanza della classe base.

Questo è stato successivamente ridefinito nel Principio di sostituzione di Liskov : l'ereditarietà pubblica dovrebbe essere utilizzata solo quando / se un'istanza della classe derivata può essere sostituita per un'istanza della classe base in qualsiasi circostanza possibile (e comunque ha senso).

In genere si dice che l'ereditarietà privata incarna un rapporto completamente diverso: "è implementato in termini di" (a volte chiamato una relazione "HAS-A"). Ad esempio, una classe `Stack` può ereditare privatamente da una classe `Vector` . L'ereditarietà privata ha una somiglianza molto maggiore con l'aggregazione che con l'eredità pubblica.

L'ereditarietà protetta non viene quasi mai utilizzata e non esiste un accordo generale sul tipo di relazione che incarna.

Eredità virtuale

Quando si utilizza l'ereditarietà, è possibile specificare la parola chiave `virtual` :

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

Quando la classe `B` ha una base virtuale `A` , significa che `A` **risiederà nella maggior parte della classe derivata** dell'albero di ereditarietà, e quindi che la maggior parte della classe derivata è anche responsabile dell'inizializzazione di quella base virtuale:

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

struct B: virtual A
```

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

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

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

Se cancelliamo un commento `/*A(88)*/` non riceveremo alcun errore dal momento che `C` sta inizializzando la sua base virtuale indiretta `A`

Inoltre, quando creiamo `object` variabili, la maggior parte della classe derivata è `C` , quindi `C` è responsabile della creazione (chiamante costruttore di) `A` e quindi il valore di `A::member` è `88` , non `5` (come sarebbe se fossimo noi creando oggetto di tipo `B`).

È utile quando si risolve il problema dei diamanti .:

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

`B` e `C` ereditano entrambi da `A` , e `D` eredita da `B` e `C` , quindi **ci sono 2 istanze di `A` in `D` !** Ciò si traduce in ambiguità quando si accede ai membri di `A` a `D` , in quanto il compilatore non ha modo di sapere da quale classe si desidera accedere a quel membro (quello che `B` eredita, o quello che è ereditato da `C` ?) .

L'ereditarietà virtuale risolve questo problema: poiché la base virtuale risiede solo nella maggior parte degli oggetti derivati, ci sarà solo un'istanza di `A` in `D`

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

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

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

La rimozione dei commenti risolve l'ambiguità.

Eredità multipla

Oltre all'eredità singola:

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

Puoi anche avere ereditarietà multipla:

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

`C` ora erediterà da `A` e `B` allo stesso tempo.

Nota: questo può portare ad ambiguità se gli stessi nomi sono usati in più `class` ereditate o `struct`. Stai attento!

Ambiguità nell'ereditarietà multipla

L'ereditarietà multipla può essere utile in alcuni casi ma, a volte, tipo strano di incontri problematici durante l'utilizzo dell'ereditarietà multipla.

Ad esempio: due classi base hanno funzioni con lo stesso nome che non sono sovrascritte nella classe derivata e se si scrive codice per accedere a quella funzione utilizzando l'oggetto della classe derivata, il compilatore mostra errore perché, non può determinare quale funzione chiamare. Ecco un codice per questo tipo di ambiguità in ereditarietà multipla.

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

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

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

Ma questo problema può essere risolto usando la funzione di risoluzione dello scope per specificare quale funzione classificare sia base1 o base2:

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

Accesso ai membri della classe

Per accedere alle variabili membro e alle funzioni membro di un oggetto di una classe, il `.` operatore è usato:

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

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

Quando si accede ai membri di una classe tramite un puntatore, viene comunemente utilizzato l'operatore `->` . In alternativa, l'istanza può essere dereferenziata e il `.` operatore utilizzato, anche se questo è meno comune:

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

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

Quando si accede a membri di classi statiche, viene utilizzato l'operatore `::` , ma sul nome della classe anziché un'istanza di esso. In alternativa, è possibile accedere al membro statico da un'istanza o da un puntatore a un'istanza utilizzando `.` o `->` operatore, rispettivamente, con la stessa sintassi dell'accesso a membri non statici.

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

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

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

sfondo

L'operatore `->` è necessario perché l'operatore di accesso membro `.` ha la precedenza sull'operatore di dereferenziazione `*` .

Ci si aspetterebbe che `*pa` avrebbe la dereferenziazione `p` (risultante in un riferimento all'oggetto `p` sta puntando a) e quindi l'accesso al suo membro `a` . Ma in effetti, tenta di accedere al membro `a` di `p` e quindi dereferenziarlo. Ie `*pa` è equivalente a `*(pa)` . Nell'esempio sopra, ciò comporterebbe un errore del compilatore a causa di due fatti: in primo luogo, `p` è un puntatore e non ha un membro `a` . In secondo luogo, `a` è un numero intero e, quindi, non può essere dereferenziato.

La soluzione non comune a questo problema sarebbe quella di controllare esplicitamente la precedenza: `(*p).a`

Invece, l'operatore `->` è quasi sempre usato. È una scorciatoia per il primo dereferenziamento del puntatore e quindi per accedervi. Cioè `(*p).a` è esattamente uguale a `p->a` .

L'operatore `::` è l'operatore dell'ambito, utilizzato nello stesso modo dell'accesso a un membro di uno spazio dei nomi. Questo perché un membro di classe statico è considerato presente nello scope di quella classe, ma non è considerato un membro delle istanze di quella classe. L'uso del normale `.` e `->` è consentito anche per i membri statici, nonostante non siano membri di istanze, per ragioni storiche; questo è utile per scrivere codice generico nei template, poiché il chiamante non deve preoccuparsi se una data funzione membro è statica o non statica.

Ereditarietà privata: limitazione dell'interfaccia di base della classe

L'ereditarietà privata è utile quando è necessario limitare l'interfaccia pubblica della classe:

```
class A {
public:
```

```
    int move();
    int turn();
};

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

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

Questo approccio impedisce in modo efficiente l'accesso ai metodi pubblici A eseguendo il casting sul puntatore A o sul riferimento:

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

Nel caso dell'ereditarietà pubblica questo tipo di casting fornirà l'accesso a tutti i metodi pubblici A nonostante metodi alternativi per prevenirlo in derivati B, come nascondere:

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

o privato utilizzando:

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

quindi per entrambi i casi è possibile:

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

Classi e strutture finali

C ++ 11

Derivare una classe può essere vietato con specificatore `final` . Dichiariamo una classe finale:

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

Ora qualsiasi tentativo di sottoclasse provocherà un errore di compilazione:

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

La classe finale può apparire ovunque nella gerarchia di classi:

```
class A {
};

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

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

Amicizia

La parola chiave `friend` viene utilizzata per fornire ad altre classi e funzioni l'accesso ai membri privati e protetti della classe, anche se sono definiti al di fuori dell'ambito della classe.

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

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

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

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

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

```
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

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

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

Classi / strutture annidate

Una `class` o una `struct` possono anche contenere un'altra definizione di `class` / `struct` all'interno di se stessa, che viene chiamata una "classe nidificata"; in questa situazione, la classe contenente viene chiamata "classe di inclusione". La definizione di classe nidificata è considerata un membro della classe di inclusione, ma è altrimenti separata.

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

Dall'esterno della classe di inclusione, è possibile accedere alle classi nidificate utilizzando l'operatore dell'ambito. Dall'interno della classe di inclusione, tuttavia, le classi nidificate possono essere utilizzate senza qualificatori:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

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

Come con una `class` / `struct` non nidificata, le funzioni membro e le variabili statiche possono essere definite all'interno di una classe nidificata o nello spazio dei nomi che lo racchiude. Tuttavia, non possono essere definiti all'interno della classe che li include, poiché è considerato una classe diversa dalla classe annidata.

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

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

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

};

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

Come con le classi non nidificate, le classi nidificate possono essere inoltrate dichiarate e definite successivamente, purché siano definite prima di essere utilizzate direttamente.

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

    class Inner1 {};

    Inner1 in1;
    Inner2* in2p;

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

class Outer::Inner2 {};

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

C ++ 11

Prima di C ++ 11, le classi nidificate avevano accesso solo ai nomi dei tipi, ai membri `static` e agli enumeratori della classe che li includeva; tutti gli altri membri definiti nella classe allegata erano off-limits.

C ++ 11

A partire da C ++ 11, le classi nidificate e i loro membri vengono trattati come se fossero `friend` della classe che li include e possono accedere a tutti i suoi membri, secondo le consuete regole di accesso; se i membri della classe nidificata richiedono la possibilità di valutare uno o più membri non statici della classe che li include, devono quindi essere passati a un'istanza:

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

        int get_x() {
```

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

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

    int x;
};
```

Viceversa, la classe che acclude *non* viene trattata come un amico della classe nidificata, e quindi non può accedere ai suoi membri privati senza il permesso esplicito.

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

        int x;
    };

    Inner in;

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

Gli amici di una classe annidata non sono automaticamente considerati amici della classe che li accoglie; se hanno bisogno di essere amici della classe che li ospita, questo deve essere dichiarato separatamente. Viceversa, poiché la classe che chiude non è automaticamente considerata un amico della classe annidata, nemmeno gli amici della classe che li accoglie saranno considerati amici della classe annidata.

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

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

        int i;
    };

    int o;
};

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

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

```
}
```

Come con tutti gli altri membri della classe, le classi nidificate possono essere nominate solo al di fuori della classe se hanno accesso pubblico. Tuttavia, è consentito accedervi indipendentemente dal modificatore di accesso, a condizione che non li nominiate esplicitamente.

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

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

// ...

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

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

Puoi anche creare un alias di tipo per una classe nidificata. Se un alias di tipo è contenuto nella classe di inclusione, il tipo nidificato e l'alias di tipo possono avere modificatori di accesso diversi. Se l'alias di tipo è esterno alla classe di inclusione, richiede che la classe nidificata o una sua `typedef` sia pubblica.

```
class Outer {
    class Inner_ {};

  public:
    typedef Inner_ Inner;
};

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

// ...

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

Come con altre classi, le classi nidificate possono derivare o derivare da altre classi.

```
struct Base {};

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

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

Questo può essere utile in situazioni in cui la classe che racchiude è derivata da un'altra classe, consentendo al programmatore di aggiornare la classe nidificata come necessario. Questo può essere combinato con un typedef per fornire un nome coerente per ogni classe nidificata della classe che racchiude:

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

  public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

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

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

// ---

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

  public:
    typedef DerivedInner_ Inner;

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

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

// ...

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

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

Nel caso precedente, sia `BaseOuter` che `DerivedOuter` forniscono rispettivamente il tipo membro `Inner` , come `BaseInner_` e `DerivedInner_` . Ciò consente di derivare i tipi nidificati senza interrompere l'interfaccia della classe che li include e consente di utilizzare il tipo annidato in modo polimorfico.

Tipi di membri e alias

Una `class` o una `struct` può anche definire alias del tipo di membro, che sono alias di tipo contenuti all'interno e trattati come membri della classe stessa.

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

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

Come i membri statici, è possibile accedere a questi typedef utilizzando l'operatore scope, `::` .

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

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

Come con gli alias di tipo normale, a ogni alias del tipo di membro è consentito fare riferimento a qualsiasi tipo definito o aliasato prima, ma non dopo, la sua definizione. Allo stesso modo, un typedef esterno alla definizione della classe può fare riferimento a qualsiasi typedef accessibile all'interno della definizione della classe, a condizione che venga dopo la definizione della classe.

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

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

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

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

Gli alias dei membri possono essere dichiarati con qualsiasi livello di accesso e rispetteranno il modificatore di accesso appropriato.

```
class TypedefAccessLevels {
    typedef int PrvInt;

  protected:
    typedef int ProInt;

  public:
    typedef int PubInt;
};
```

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

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

Questo può essere usato per fornire un livello di astrazione, permettendo al progettista di una classe di cambiare il suo funzionamento interno senza rompere il codice che si basa su di esso.

```
class Something {
    friend class SomeComplexType;

    short s;
    // ...

  public:
    typedef SomeComplexType MyHelper;

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

    // ...
};

// ...

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

In questa situazione, se la classe helper viene modificata da `SomeComplexType` in un altro tipo, è necessario modificare solo la `typedef` e la dichiarazione `friend` ; finché la classe helper fornisce la stessa funzionalità, qualsiasi codice che lo utilizza come `Something::MyHelper` invece di specificarlo per nome funzionerà in genere senza modifiche. In questo modo, riduciamo al minimo la quantità di codice che deve essere modificata quando l'implementazione sottostante viene modificata, in modo tale che il nome del tipo debba essere modificato solo in una posizione.

Questo può anche essere combinato con `decltype` , se lo si desidera.

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

  public:
    typedef decltype(helper) MyHelper;

  private:
    InternalVariable<MyHelper> ivh;

    // ...

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

```
    // ...
};
```

In questa situazione, la modifica dell'implementazione di `SomethingElse::helper` modificherà automaticamente il typedef per noi, a causa di `decltype` . Ciò minimizza il numero di modifiche necessarie quando si desidera cambiare l' `helper` , riducendo al minimo il rischio di errore umano.

Come con tutto, tuttavia, questo può essere preso troppo lontano. Se il typename viene usato solo una o due volte internamente e zero volte esternamente, ad esempio, non è necessario fornire un alias per questo. Se viene usato centinaia o migliaia di volte in un progetto, o se ha un nome abbastanza lungo, può essere utile fornirlo come typedef invece di usarlo sempre in termini assoluti. Bisogna bilanciare la compatibilità e la praticità con la quantità di rumore non necessario creato.

Questo può essere usato anche con classi template, per fornire accesso ai parametri del template dall'esterno della classe.

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

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

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

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

Questo è comunemente usato con i contenitori, che di solito forniscono il loro tipo di elemento, e altri tipi di helper, come alias del tipo di membro. La maggior parte dei contenitori della libreria standard C ++, ad esempio, fornisce i seguenti 12 tipi di helper, insieme a qualsiasi altro tipo speciale di cui potrebbero aver bisogno.

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

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

```
    typedef MyConstIterator<value_type>           const_iterator;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                size_type;
    typedef ptrdiff_t                             difference_type;
};
```

Prima di C ++ 11, era anche comunemente usato per fornire un " `typedef` template" di sorta, in quanto la funzione non era ancora disponibile; questi sono diventati un po 'meno comuni con l'introduzione di modelli alias, ma sono ancora utili in alcune situazioni (e sono combinati con modelli di alias in altre situazioni, che possono essere molto utili per ottenere singoli componenti di un tipo complesso come un puntatore a funzione). Solitamente usano il `type` nome per il loro alias di tipo.

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

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

Questo è stato spesso utilizzato con tipi con più parametri del modello, per fornire un alias che definisce uno o più parametri.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

OneDArray<int, 3>::type     arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type   arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Membri della classe statici

Una classe può anche avere membri `static` , che possono essere variabili o funzioni. Questi sono considerati nella portata della classe, ma non sono trattati come membri normali; hanno una durata di archiviazione statica (esistono dall'inizio del programma fino alla fine), non sono legati a una particolare istanza della classe e esiste solo una copia per l'intera classe.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                         // Non-static member variable.

  public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();      // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
//  (&one.i != &two.i)
//  (&one.i != &three.i)
//  (&two.i != &three.i).
// All three Examples share "num_instances", such that:
//  (&one.num_instances == &two.num_instances)
//  (&one.num_instances == &three.num_instances)
//  (&two.num_instances == &three.num_instances)
```

Le variabili membro statiche non sono considerate definite all'interno della classe, solo dichiarate, e quindi hanno la loro definizione al di fuori della definizione della classe; il programmatore è autorizzato, ma non richiesto, a inizializzare le variabili statiche nella loro definizione. Quando si definiscono le variabili membro, la parola chiave `static` viene omessa.

```
class Example {
    static int num_instances;              // Declaration.

  public:
    static std::string static_str;         // Declaration.

    // ...
};

int         Example::num_instances;         // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.
```

A causa di ciò, le variabili statiche possono essere tipi incompleti (a parte il `void`), a condizione che vengano successivamente definiti come un tipo completo.

```
struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete    i_contain_myself;
    static int             an_array[];
};

struct ForwardDeclared {};
```

```
ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
int             ExIncomplete::an_array[5];
```

Le funzioni membro statiche possono essere definite all'interno o all'esterno della definizione della classe, come con le normali funzioni membro. Come per le variabili membro statiche, la parola chiave `static` viene omessa quando si definiscono funzioni membro statiche al di fuori della definizione della classe.

```
// For Example above, either...
class Example {
    // ...

  public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int  Example::static_func() { return num_instances; }
void Example::set_str(const std::string& str) { static_str = str; }
```

Se una variabile membro statica è dichiarata `const` ma non `volatile` ed è di tipo integrale o di enumerazione, può essere inizializzata alla dichiarazione, all'interno della definizione della classe.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;              // Good.
    static const E ce = VAL;              // Good.
    const static double cd = 5;           // Error.
    static const volatile int cvi = 5;    // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;        // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

C ++ 11

A partire da C ++ 11, le variabili membro statiche dei tipi `LiteralType` (tipi che possono essere costruiti in fase di compilazione, secondo le regole di `constexpr`) possono anche essere dichiarate come `constexpr` ; in tal caso, devono essere inizializzati all'interno della definizione della classe.

```
struct ExConstexpr {
```

```
    constexpr static int ci = 5;                       // Good.
    static constexpr double cd = 5;                    // Good.
    constexpr static int carr[] = { 1, 1, 2 };         // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;                       // Error.
};

constexpr int ExConstexpr::bad_ci = 5;                 // Still an error.
```

Se una variabile membro `const` o `constexpr` è *odr-used* (informalmente, se ha il suo indirizzo preso o è assegnato a un riferimento), allora deve ancora avere una definizione separata, al di fuori della definizione della classe. Questa definizione non può contenere un inizializzatore.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

Poiché i membri statici non sono legati a una determinata istanza, è possibile accedervi utilizzando l'operatore scope, `::` .

```
std::string str = Example::static_str;
```

Possono anche essere accessibili come se fossero membri normali e non statici. Ciò è di importanza storica, ma viene utilizzato meno comunemente dell'operatore di ambito per evitare confusione sul fatto che un membro sia statico o non statico.

```
Example ex;
std::string rts = ex.static_str;
```

I membri della classe possono accedere ai membri statici senza qualificare il loro ambito, come con i membri della classe non statici.

```
class ExTwo {
    static int num_instances;
    int my_num;

  public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

Non possono essere `mutable` , né dovrebbero essere; poiché non sono legati a nessuna istanza data, se un'istanza è o non è const non influisce sui membri statici.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-
ness.
```

I membri statici rispettano i modificatori di accesso, proprio come i membri non statici.

```
class ExAccess {
    static int prv_int;

  protected:
    static int pro_int;

  public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.
```

Poiché non sono legati a una determinata istanza, le funzioni membro statiche non hanno `this` puntatore; a causa di ciò, non possono accedere a variabili membro non statiche a meno che non abbiano passato un'istanza.

```
class ExInstanceRequired {
    int i;

  public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }                          // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};
```

Non avendo `this` puntatore, i loro indirizzi non possono essere archiviati nelle funzioni di puntatori

a membro e vengono invece memorizzati in normali puntatori a funzioni.

```
struct ExPointer {
           void nsfunc() {}
    static void  sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
    f_ptr p_sf = &ExPointer::sfunc; // Good.
```

Non avendo `this` puntatore, non possono essere `const` o `volatile` , né possono avere qualifiche di ref. Inoltre, non possono essere virtuali.

```
struct ExCVQualifiersAndVirtual {
    static void   func()                {} // Good.
    static void  cfunc() const          {} // Error.
    static void  vfunc() volatile       {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void  rfunc() &              {} // Error.
    static void rvfunc() &&             {} // Error.

    virtual static void vsfunc()        {} // Error.
    static virtual void svfunc()        {} // Error.
};
```

Poiché non sono legati a una determinata istanza, le variabili membro statiche vengono effettivamente trattate come variabili globali speciali; vengono creati all'avvio del programma e distrutti quando escono, indipendentemente dal fatto che esistano effettivamente delle istanze della classe. Esiste solo una singola copia di ogni variabile membro statica (a meno che la variabile non sia dichiarata `thread_local` (C ++ 11 o successivo), nel qual caso c'è una copia per thread).

Le variabili membro statiche hanno lo stesso collegamento della classe, indipendentemente dal fatto che la classe abbia un collegamento interno o esterno. Le classi locali e le classi senza nome non possono avere membri statici.

Funzioni membro non statiche

Una classe può avere funzioni membro non statiche , che operano su singole istanze della classe.

```
class CL {
  public:
    void member_function() {}
};
```

Queste funzioni sono chiamate su un'istanza della classe, in questo modo:

```
CL instance;
instance.member_function();
```

Possono essere definiti all'interno o all'esterno della definizione della classe; se definiti all'esterno, vengono specificati come appartenenti all'ambito della classe.

```
struct ST {
    void  defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

Possono essere qualificati CV e / o ref-qualificati , influenzando il modo in cui vedono l'istanza su cui sono chiamati; la funzione vedrà l'istanza come avente i qualificatori di cv specificati, se ce ne sono. La versione che viene chiamata sarà basata sui qualificatori di cv dell'istanza. Se non esiste una versione con gli stessi qualificatori di cv dell'istanza, verrà chiamata una versione più cv se disponibile.

```
struct CVQualifiers {
    void func()                   {} // 1: Instance is non-cv-qualified.
    void func() const             {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers       non_cv_instance;
const CVQualifiers      c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func();      // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only();      // Calls const volatile version.
```

C ++ 11

I qualificatori di ref della funzione membro indicano se la funzione è destinata a essere chiamata su istanze rvalue e utilizzare la stessa sintassi della funzione cv-qualifiers.

```
struct RefQualifiers {
    void func() &  {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func();              // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

I qualificazioni di CV e i qualificazioni di qualificazione possono anche essere combinati, se necessario.

```
struct BothCVAndRef {
    void func() const& {} // Called on normal instances.  Sees instance as const.
    void func() &&     {} // Called on temporary instances.
};
```

Possono anche essere virtuali ; questo è fondamentale per il polimorfismo e consente a una o più

classi figlio di fornire la stessa interfaccia della classe genitore, pur fornendo le proprie funzionalità.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp.func(); // Calls Base::func().
dp.func(); // Calls Derived::func().
```

Per ulteriori informazioni, vedere qui .

Struttura / classe senza nome

La `struct` *senza nome* è consentita (il tipo non ha nome)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

o

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

e più tardi

```
Circle circle;
circle.center.x = 42.f;
```

ma NON `struct` *anonima* (tipo senza nome e oggetto senza nome)

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
```

```
    }; // No member either.
    float radius;
};
```

Nota: alcuni compilatori consentono la `struct` *anonima* come *estensione* .

C ++ 11

- *lamdba* può essere visto come una speciale `struct` *senza nome* .
- `decltype` consente di recuperare il tipo di `struct` *senza nome* :

```
decltype(circle.point) otherPoint;
```

- *l'* istanza `struct` *non* `struct` può essere parametro del metodo template:

```
void print_square_coordinates()
{
    const struct {float x; float y;} points[] = {
        {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
    for (const auto& point : points) {
        std::cout << "{" << point.x << ", " << point.y << "}\n";
    }

    decltype(points[0]) topRightCorner{1, 1};
    auto it = std::find(points, points + 4, topRightCorner);
    std::cout << "top right corner is the "
              << 1 + std::distance(points, it) << "th\n";
}
```

Leggi Classi / Strutture online: https://riptutorial.com/it/cplusplus/topic/508/classi---strutture

Capitolo 14: Compilazione e costruzione

introduzione

I programmi scritti in C ++ devono essere compilati prima di poter essere eseguiti. Esiste una grande varietà di compilatori disponibili in base al tuo sistema operativo.

Osservazioni

La maggior parte dei sistemi operativi vengono forniti senza compilatore e devono essere installati in seguito. Alcune scelte comuni dei compilatori sono:

- GCC, GNU Compiler Collection g ++
- clang: un frontend di famiglia in linguaggio C per LLVM clang ++
- MSVC, Microsoft Visual C ++ (incluso in Visual Studio) visual-c ++
- C ++ Builder, Embarcadero C ++ Builder (incluso in RAD Studio) c ++ builder

Si prega di consultare il manuale del compilatore appropriato, su come compilare un programma C ++.

Un'altra opzione per utilizzare un compilatore specifico con un proprio sistema di generazione specifico, è possibile lasciare che i sistemi di generazione generici configurino il progetto per un compilatore specifico o per quello predefinito installato.

Examples

Compilare con GCC

Supponendo un singolo file sorgente denominato `main.cpp` , il comando per compilare e collegare un eseguibile non ottimizzato è il seguente (Compilare senza ottimizzazione è utile per lo sviluppo iniziale e il debug, sebbene `-Og` sia ufficialmente raccomandato per le nuove versioni di GCC).

```
g++ -o app -Wall main.cpp -O0
```

Per produrre un eseguibile ottimizzato per l'utilizzo in produzione, utilizzare una delle opzioni `-O` (vedere: `-O1` , `-O2` , `-O3` , `-Os` , `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

Se l'opzione -O viene omessa, -O0, che significa nessuna ottimizzazione, viene utilizzata come predefinita (specificando -O senza un numero si risolve in -O1).

In alternativa, utilizzare i flag di ottimizzazione dai gruppi `O` (o più ottimizzazioni sperimentali) direttamente. Il seguente esempio costruisce con l'ottimizzazione `-O2` , più un flag dal livello di ottimizzazione `-O3` :

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

Per produrre un eseguibile ottimizzato specifico per piattaforma (da utilizzare nella produzione sulla macchina con la stessa architettura), utilizzare:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

Uno dei due precedenti produrrà un file binario che può essere eseguito con `.\app.exe` su Windows e `./app` su Linux, Mac OS, ecc.

Il flag `-o` può anche essere saltato. In questo caso, GCC creerà l'output eseguibile predefinito `a.exe` su Windows e `a.out` su sistemi simil-Unix. Per compilare un file senza collegarlo, utilizzare l'opzione `-c` :

```
g++ -o file.o -Wall -c file.cpp
```

Questo produce un file oggetto denominato `file.o` che può essere successivamente collegato ad altri file per produrre un binario:

```
g++ -o app file.o otherfile.o
```

Ulteriori informazioni sulle opzioni di ottimizzazione sono disponibili su gcc.gnu.org . Di particolare nota sono `-Og` (ottimizzazione con enfasi sull'esperienza di debug - raccomandata per il ciclo standard edit-compile-debug) e `-Ofast` (tutte le ottimizzazioni, comprese quelle che ignorano la conformità agli standard rigorosi).

Il flag `-Wall` abilita gli avvisi per molti errori comuni e deve essere sempre utilizzato. Per migliorare la qualità del codice, spesso è anche consigliato utilizzare `-Wextra` e altri flag di avviso che non sono automaticamente abilitati da `-Wall` e `-Wextra` .

Se il codice prevede uno specifico standard C ++, specificare quale standard utilizzare includendo il flag `-std=` . I valori supportati corrispondono all'anno di finalizzazione per ciascuna versione dello standard ISO C ++. A partire da GCC 6.1.0, i valori validi per lo `std=` flag sono `c++98` / `c++03` , `c++11` , `c++14` e `c++17` / `c++1z` . I valori separati da una barra diretta sono equivalenti.

```
g++ -std=c++11 <file>
```

GCC include alcune estensioni specifiche del compilatore che sono disabilitate quando sono in conflitto con uno standard specificato dal flag `-std=` . Per compilare con tutte le estensioni abilitate, è possibile utilizzare il valore `gnu++XX` , dove `XX` è uno degli anni utilizzati dai valori `c++` sopra elencati.

Lo standard predefinito verrà utilizzato se non viene specificato nessuno. Per le versioni di GCC precedenti alla 6.1.0, il valore predefinito è `-std=gnu++03` ; in GCC 6.1.0 e `-std=gnu++14` , il valore predefinito è `-std=gnu++14` .

Si noti che a causa di bug in GCC, il flag `-pthread` deve essere presente alla compilazione e al

collegamento per GCC per supportare la funzionalità di threading standard C ++ introdotta con C ++ 11, come `std::thread` e `std::wait_for` . Omettendola quando si usano le funzioni di threading, non si ottengono avvisi ma risultati non validi su alcune piattaforme.

Collegamento con le librerie:

Utilizzare l'opzione `-l` per passare il nome della libreria:

```
g++ main.cpp -lpcre2-8
#pcre2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

Se la libreria non si trova nel percorso della libreria standard, aggiungere il percorso con l'opzione `-L` :

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Le librerie multiple possono essere collegate tra loro:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

Se una libreria dipende da un'altra, metti la libreria dipendente **prima** della libreria indipendente:

```
g++ main.cpp -lchild-lib -lbase-lib
```

Oppure lascia che il linker determini l'ordinamento stesso tramite `--start-group` e `--end-group` (nota: questo ha un costo significativo delle prestazioni):

```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Compilazione con Visual C ++ (riga di comando)

Per i programmatori provenienti da GCC o Clang per Visual Studio, o per i programmatori più a loro agio con la riga di comando in generale, è possibile utilizzare il compilatore Visual C ++ dalla riga di comando e l'IDE.

Se desideri compilare il codice dalla riga di comando in Visual Studio, devi prima configurare l'ambiente della riga di comando. Questa operazione può essere eseguita aprendo il `Visual Studio Command Prompt` / `Visual Studio Command Prompt Developer Command Prompt` / `Visual Studio Command Prompt Developer Command Prompt x86 Native Tools Command Prompt x64 Native Tools Command Prompt` / `x86 Native Tools Command Prompt x64 Native Tools Command Prompt` o simile (come fornito dalla versione di Visual Studio) oppure, al prompt dei comandi, passando a la sottodirectory `VC` directory di installazione del compilatore (in genere `\Program Files (x86)\Microsoft Visual Studio x\VC` , dove `x` è il numero di versione (come `10.0` per il 2010 o `14.0` per il 2015) ed esegue il file batch `VCVARSALL` con un parametro della riga di comando specificato qui .

Nota che a differenza di GCC, Visual Studio non fornisce un front-end per il linker (`link.exe`)

tramite il compilatore (`cl.exe`), ma fornisce invece il linker come un programma separato, che il compilatore chiama quando esce. `cl.exe` e `link.exe` possono essere usati separatamente con diversi file e opzioni, oppure `cl` può essere detto di passare file e opzioni per `link` se entrambe le attività sono eseguite insieme. Qualsiasi opzione di collegamento specificata per `cl` verrà tradotta in opzioni per il `link` e tutti i file non elaborati da `cl` verranno passati direttamente al `link` . Poiché questa è principalmente una semplice guida alla compilazione con la riga di comando di Visual Studio, gli argomenti per il `link` non verranno descritti in questo momento; se hai bisogno di una lista, guarda qui .

Si noti che gli argomenti da `cl` sono case-sensitive, mentre gli argomenti da `link` non lo sono.

[Si noti che alcuni dei seguenti esempi utilizzano la variabile "directory corrente" della shell di Windows, `%cd%` , quando si specificano i nomi di percorso assoluti. Per chi non ha familiarità con questa variabile, si espande nella directory di lavoro corrente. Dalla riga di comando, sarà la directory in cui ti `cl` quando hai eseguito `cl` , e viene specificato nel prompt dei comandi per impostazione predefinita (se il tuo prompt dei comandi è `C:\src>` , ad esempio, `%cd%` è `C:\src\`).]

Supponendo che un singolo file sorgente denominato `main.cpp` nella cartella corrente, il comando per compilare e collegare un eseguibile non ottimizzato (utile per lo sviluppo iniziale e il debug) è (utilizzare uno dei seguenti):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".

cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

Supponendo un file sorgente aggiuntivo "niam.cpp" nella stessa directory, utilizzare quanto segue:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Puoi anche utilizzare i caratteri jolly, come ci si aspetterebbe:

```
cl main.cpp src\*.cpp
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
//  "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
// Generates executable "main.exe".
```

Per rinominare o riposizionare l'eseguibile, utilizzare uno dei seguenti:

```
cl /o name main.cpp
// Generates executable named "name.exe".
```

```
cl /o folder\ main.cpp
// Generates executable named "main.exe", in folder "%cd%\folder".

cl /o folder\name main.cpp
// Generates executable named "name.exe", in folder "%cd%\folder".

cl /Fename main.cpp
// Same as "/o name".

cl /Fefolder\ main.cpp
// Same as "/o folder\".

cl /Fefolder\name main.cpp
// Same as "/o folder\name".
```

Sia `/o` che `/Fe` passano il loro parametro (chiamiamolo `o-param`) al `link` come `/OUT:o-param` , aggiungendo l'estensione appropriata (generalmente `.exe` o `.dll`) a "nominare" `o-param` come necessario. Mentre sia `/o` che `/Fe` sono a mia conoscenza identiche nella funzionalità, quest'ultimo è preferito per Visual Studio. `/o` è contrassegnato come deprecato e sembra essere fornito principalmente per i programmatori più familiari con GCC o Clang.

Si noti che mentre lo spazio tra `/o` e la cartella e / o il nome specificati è facoltativo, non *può* esserci uno spazio tra `/Fe` e la cartella e / o il nome specificati.

Allo stesso modo, per produrre un eseguibile ottimizzato (per l'uso in produzione), utilizzare:

```
cl /O1 main.cpp
// Optimise for executable size.  Produces small programs, at the possible expense of slower
//  execution.

cl /O2 main.cpp
// Optimise for execution speed.  Produces fast programs, at the possible expense of larger
//  file size.

cl /GL main.cpp other.cpp
// Generates special object files used for whole-program optimisation, which allows CL to
//  take every module (translation unit) into consideration during optimisation.
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during
//  the linking phase to perform additional optimisations.  If linking is not performed at
this
//  time, the generated object files should be linked with "/LTCG".
// Can be used with other CL optimisation options.
```

Infine, per produrre un eseguibile ottimizzato specifico della piattaforma (da utilizzare nella produzione sulla macchina con l'architettura specificata), selezionare il prompt dei comandi appropriato o il parametro `VCVARSALL` per la piattaforma di destinazione. `link` dovrebbe rilevare la piattaforma desiderata dai file oggetto; in caso contrario, utilizzare l' opzione `/MACHINE` per specificare esplicitamente la piattaforma di destinazione.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
cl main.cpp /link /machine:X64
```

Qualsiasi dei precedenti produrrà un eseguibile con il nome specificato da `/o` o `/Fe` , o se nessuno dei due è fornito, con un nome identico al primo file sorgente o oggetto specificato nel compilatore.

```
cl a.cpp b.cpp c.cpp
// Generates "a.exe".

cl d.obj a.cpp q.cpp
// Generates "d.exe".

cl y.lib n.cpp o.obj
// Generates "n.exe".

cl /o yo zp.obj pz.cpp
// Generates "yo.exe".
```

Per compilare un file (s) senza collegamento, utilizzare:

```
cl /c main.cpp
// Generates object file "main.obj".
```

Questo dice a `cl` di uscire senza chiamare il `link` e produce un file oggetto, che può essere successivamente collegato ad altri file per produrre un binario.

```
cl main.obj niam.cpp
// Generates object file "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".

link main.obj niam.obj
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Ci sono anche altri preziosi parametri da riga di comando, che sarebbe molto utile per gli utenti sapere:

```
cl /EHsc main.cpp
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,
//  and `extern "C"` functions will not throw exceptions.
// This is recommended when writing portable, platform-independent code.

cl /clr main.cpp
// "/clr" specifies that the code should be compiled to use the common language runtime,
//  the .NET Framework's virtual machine.
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,
//  and creates an executable that requires .NET to run.

cl /Za main.cpp
// "/Za" specifies that Microsoft extensions should be disabled, and code should be
//  compiled strictly according to ISO C++ specifications.
// This is recommended for guaranteeing portability.

cl /Zi main.cpp
// "/Zi" generates a program database (PDB) file for use when debugging a program, without
//  affecting optimisation specifications, and passes the option "/DEBUG" to LINK.
```

```
cl /LD dll.cpp
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling
those
//  programs.

cl main.cpp /link /LINKER_OPTION
// "/link" passes everything following it directly to LINK, without parsing it in any way.
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

Per chiunque abbia più familiarità con i sistemi * nix e / o GCC / Clang, `cl` , `link` e altri strumenti da riga di comando di Visual Studio possono accettare parametri specificati con un trattino (come `-c`) invece di una barra (come `/c`). Inoltre, Windows riconosce una barra o una barra rovesciata come separatore di percorsi valido, quindi è possibile utilizzare anche i percorsi in stile * nix. Ciò semplifica la conversione di semplici righe di comando del compilatore da `g++` o `clang++` a `cl` , o viceversa, con modifiche minime.

```
g++ -o app src/main.cpp
cl  -o app src/main.cpp
```

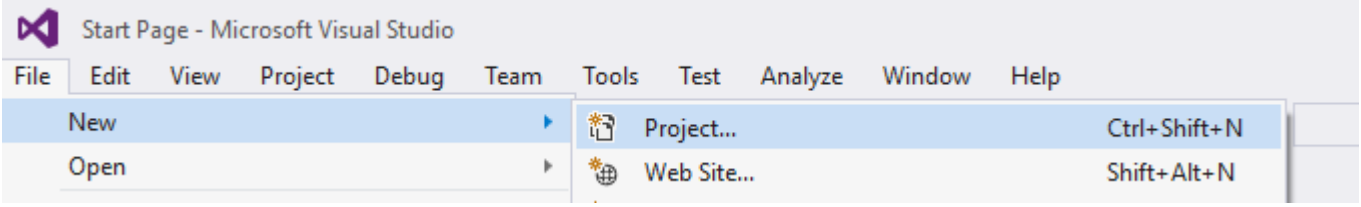
Ovviamente, durante il porting di righe di comando che utilizzano opzioni più complesse di `g++` o `clang++` , è necessario cercare comandi equivalenti nelle documentazioni del compilatore applicabili e / o nei siti di risorse, ma questo rende più facile iniziare le cose con il minimo tempo speso a conoscere nuovi compilatori.

Nel caso in cui ti occorrono caratteristiche linguistiche specifiche per il tuo codice, è necessaria una versione specifica di MSVC. Da Visual C ++ 2015 Update 3 su di esso è possibile scegliere la versione dello standard da compilare tramite il flag `/std` . I valori possibili sono `/std:c++14` e `/std:c++latest` (`/std:c++17` seguirà presto).

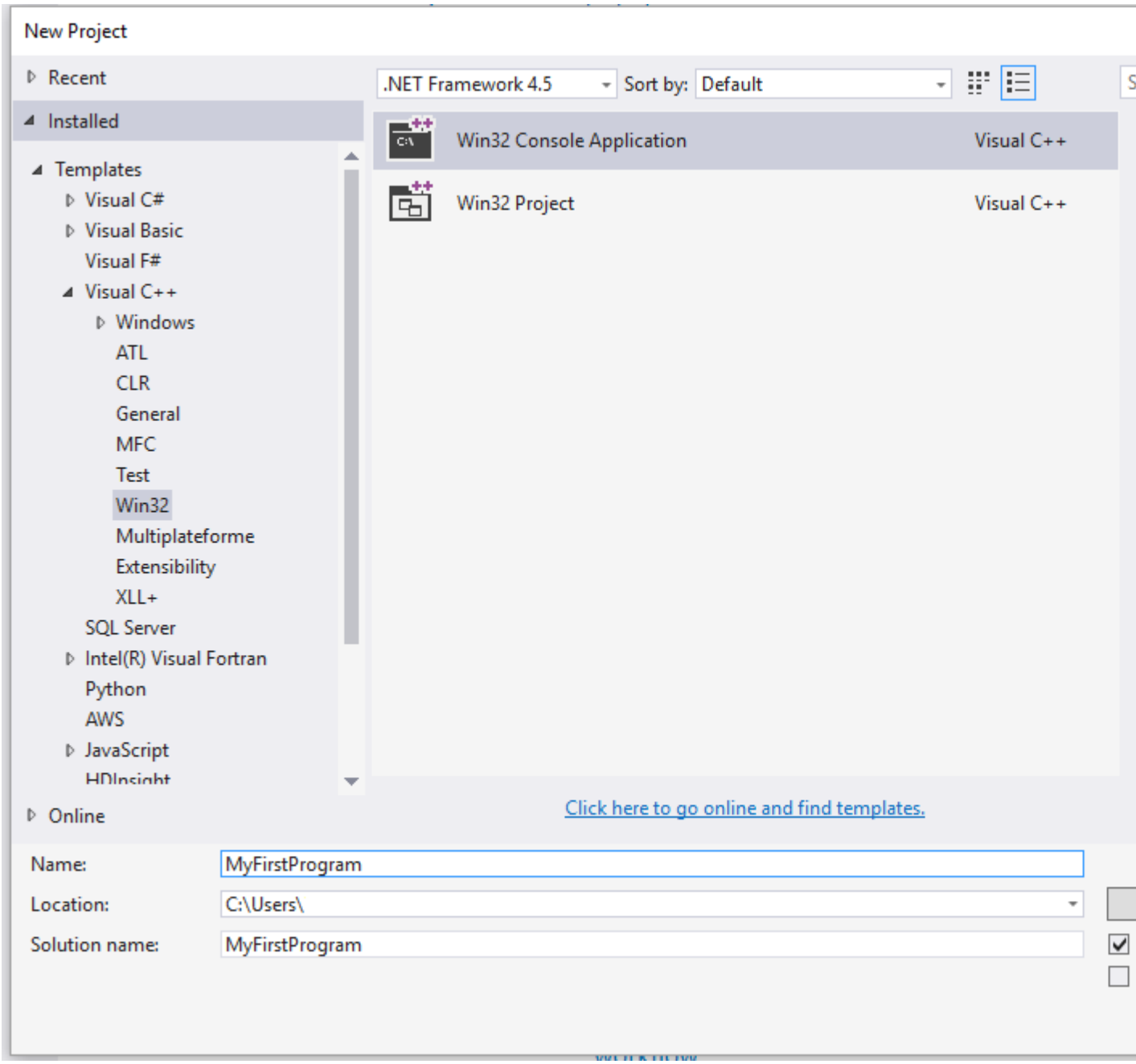
Nota: nelle versioni precedenti di questo compilatore erano disponibili flag di funzionalità specifiche, tuttavia questo era principalmente utilizzato per le anteprime di nuove funzionalità.

Compilazione con Visual Studio (interfaccia grafica) - Hello World

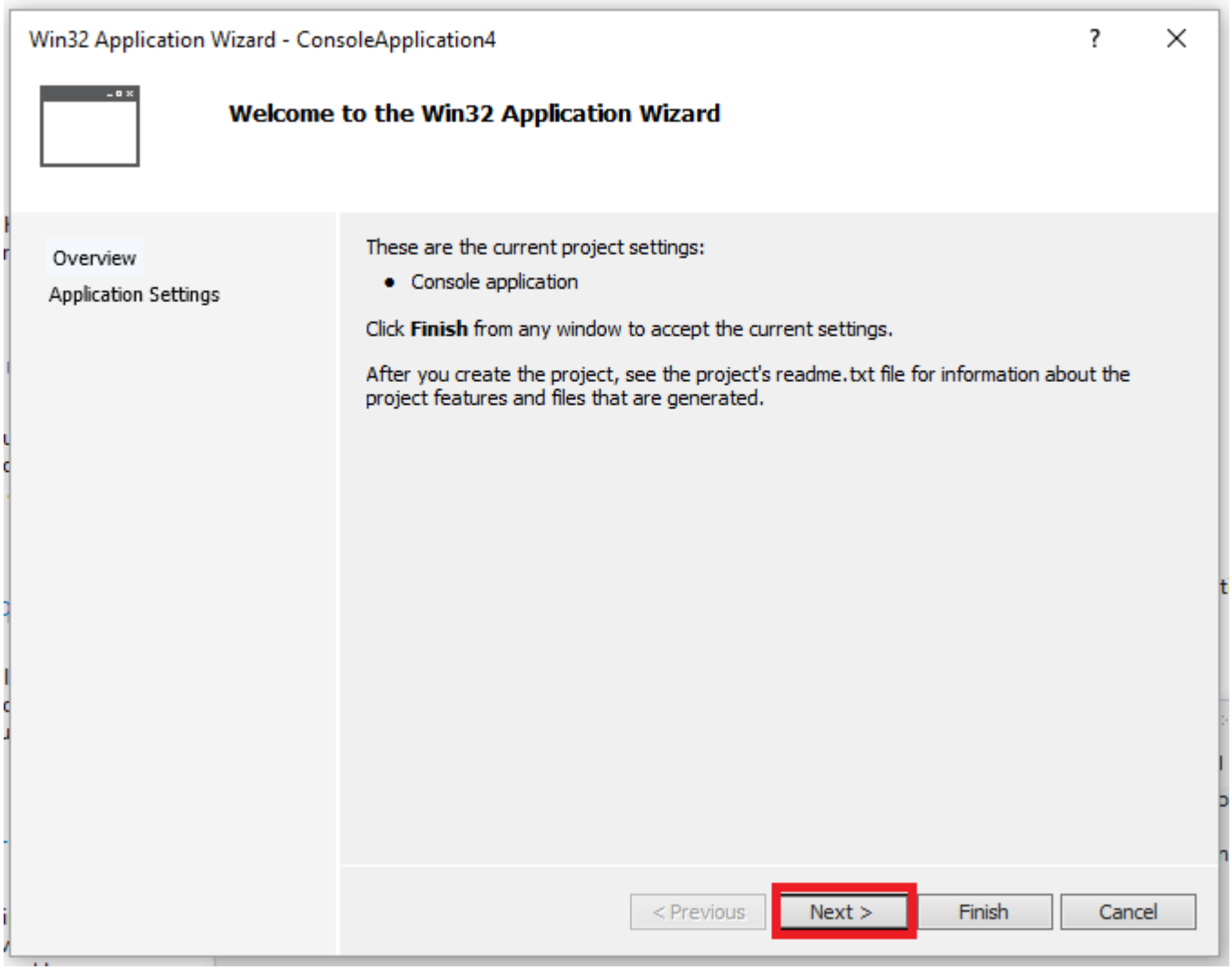
1. Scarica e installa Visual Studio Community 2015
2. Apri la community di Visual Studio
3. Fare clic su File -> Nuovo -> Progetto



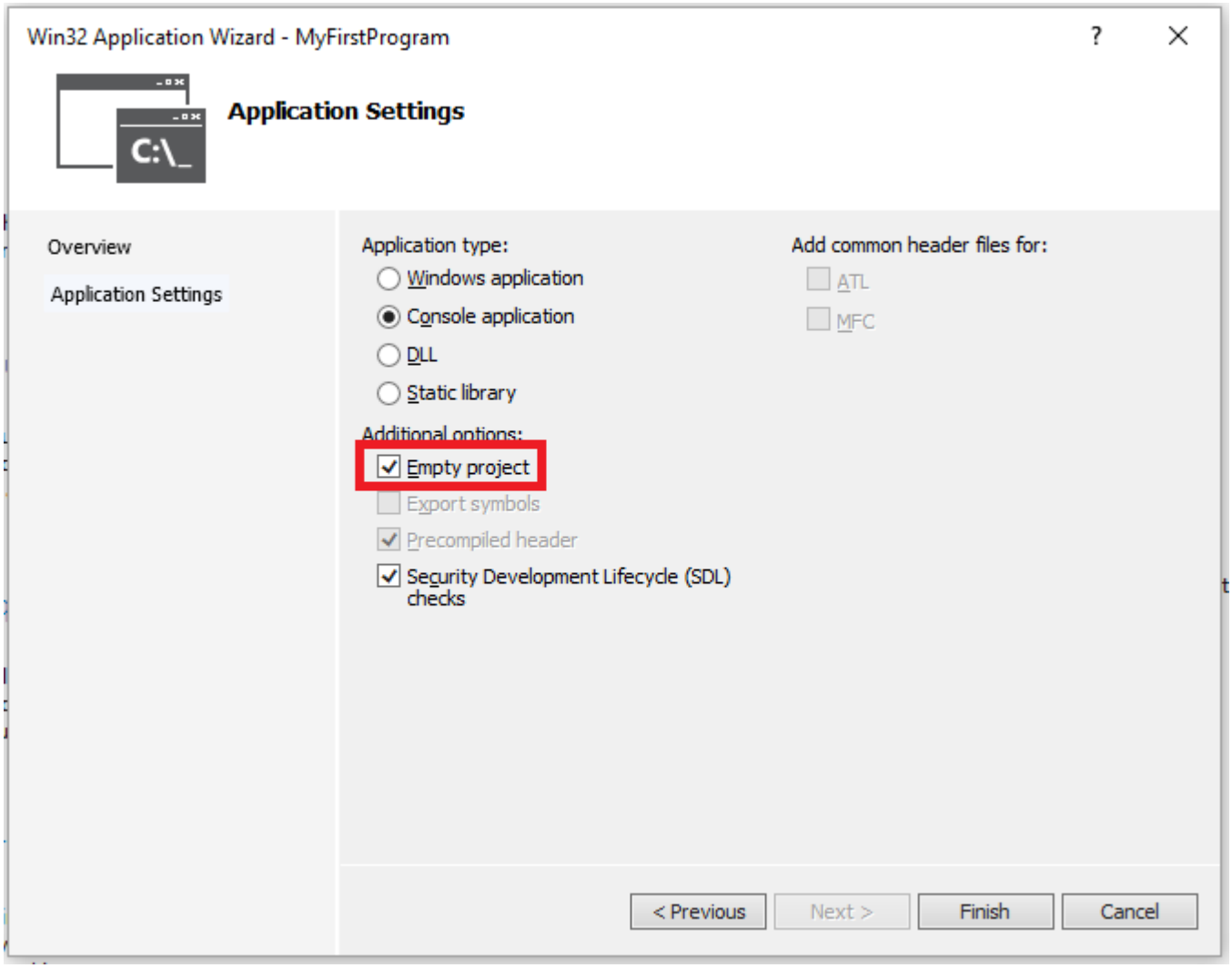
4. Fare clic su Modelli -> Visual C ++ -> Applicazione console Win32 e quindi denominare il progetto **MyFirstProgram** .



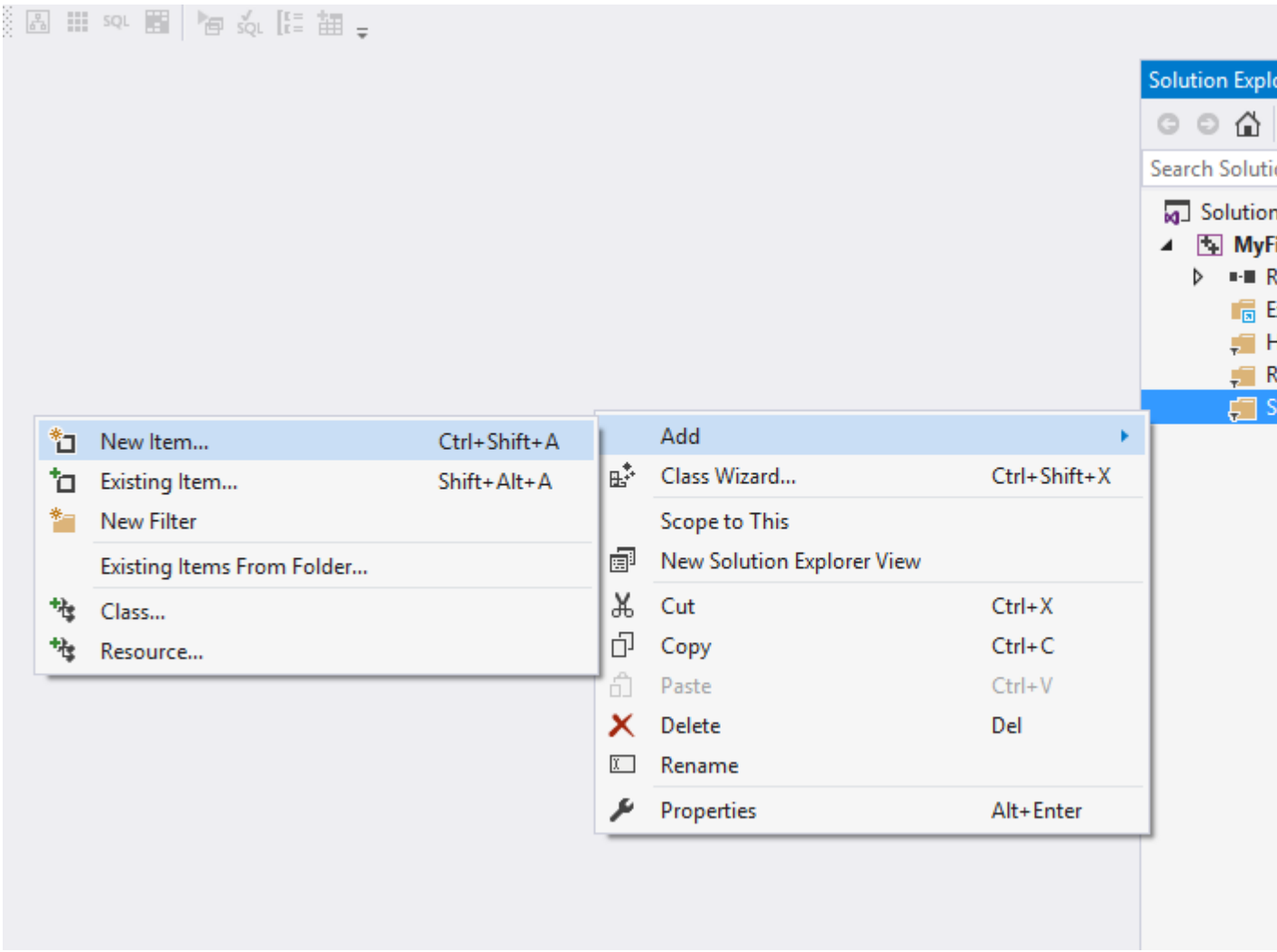
5. Clicca Ok
6. Fare clic su Avanti nella seguente finestra.



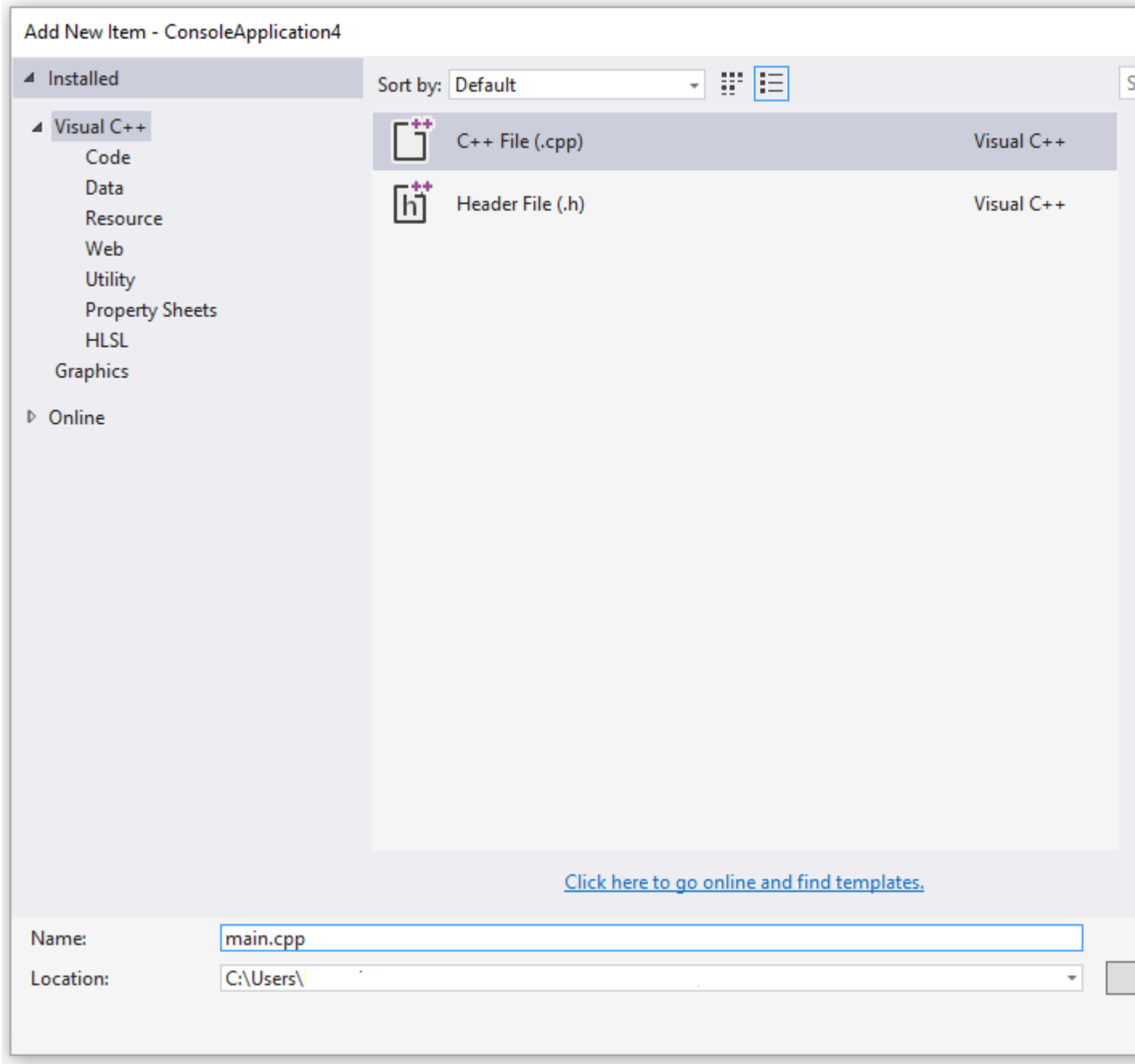
7. Seleziona la casella `Empty project` e quindi fai clic su Fine:



8. Fai clic destro sulla cartella File sorgente quindi -> Aggiungi -> Nuovo elemento:



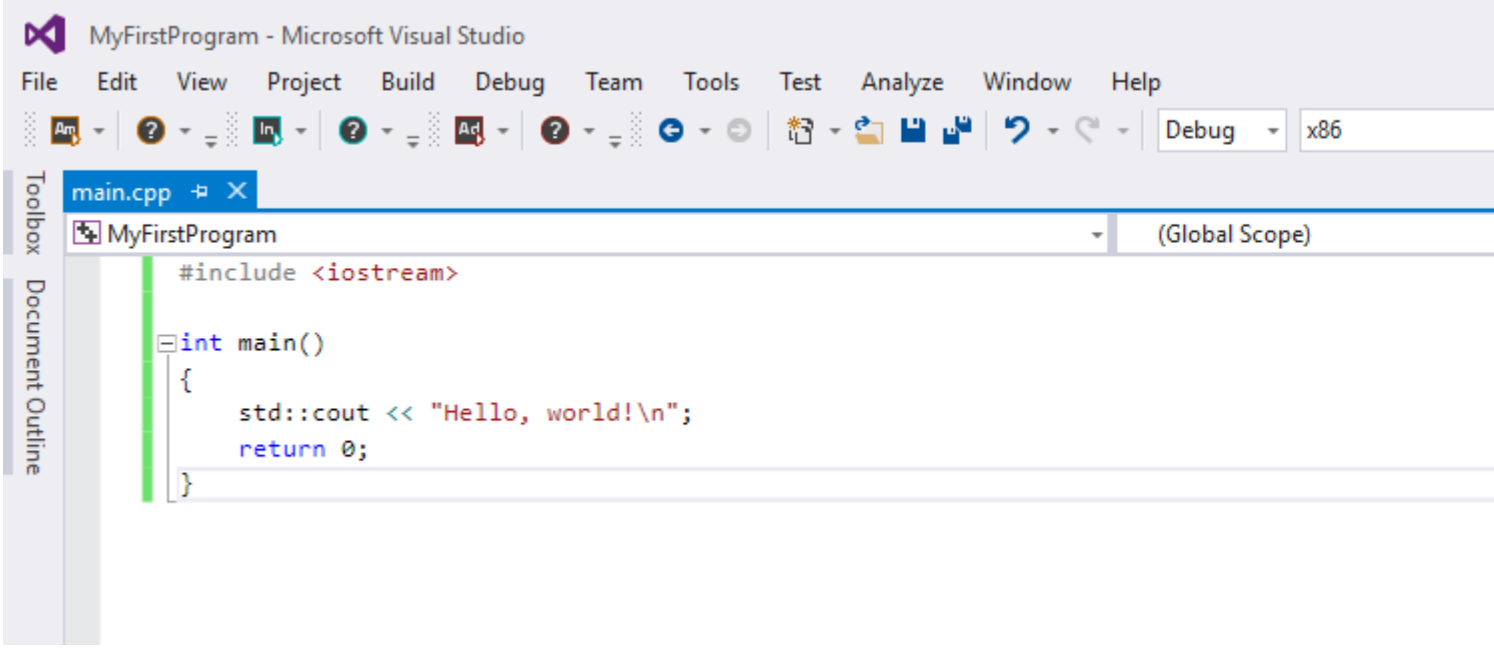
9. Selezionare il file C ++ e denominare il file main.cpp, quindi fare clic su Aggiungi:



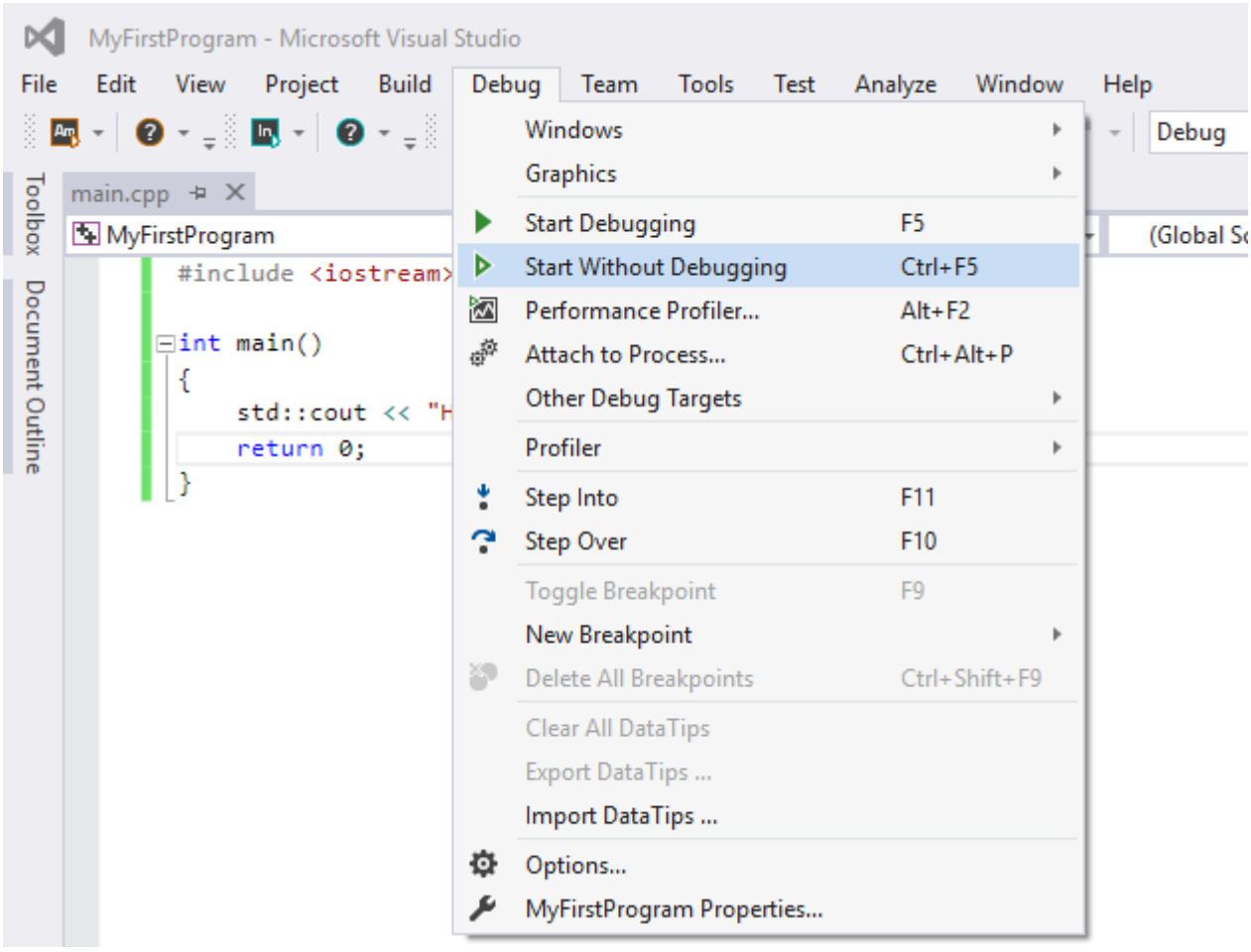
10: Copia e incolla il seguente codice nel nuovo file main.cpp:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

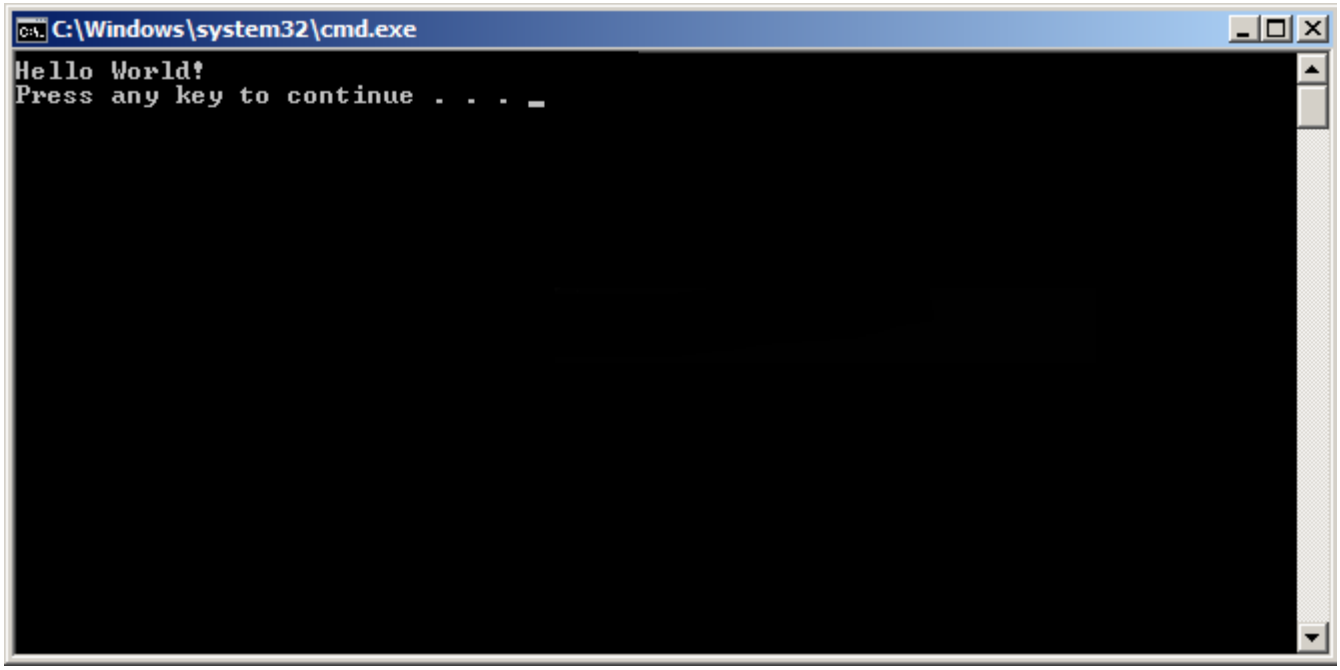
Il tuo ambiente dovrebbe assomigliare a:



11. Fai clic su Debug -> Avvia **senza** eseguire il debug (o premi ctrl + F5):



12. Fatto. Dovresti ottenere il seguente output della console:



Compilando con Clang

Poiché il front-end di Clang è progettato per essere compatibile con GCC, molti programmi che possono essere compilati tramite GCC verranno compilati quando si scambierà `g++` da `clang++` negli script di compilazione. Se non viene fornita la `-std=version` , verrà utilizzato gnu11.

Gli utenti di Windows che sono abituati a MSVC possono scambiare `cl.exe` con `clang-cl.exe` . Per impostazione predefinita, clang tenta di essere compatibile con la versione più alta di MSVC che è stata installata.

Nel caso della compilazione con Visual Studio, clang-cl può essere utilizzato modificando il `Platform toolset` di `Platform toolset` della `Platform toolset` nelle proprietà del progetto.

In entrambi i casi, clang è compatibile solo tramite il suo front-end, sebbene cerchi anche di generare file di oggetti binari compatibili. Gli utenti di clang-cl devono tenere presente che la compatibilità con MSVC non è ancora completa .

Per usare clang o clang-cl, si potrebbe usare l'installazione di default su determinate distribuzioni Linux o quelle in bundle con IDE (come XCode su Mac). Per le altre versioni di questo compilatore o su piattaforme che non hanno questo installato, questo può essere scaricato dalla pagina di download ufficiale .

Se stai usando CMake per costruire il tuo codice, in genere puoi cambiare il compilatore impostando le variabili di ambiente `CC` e `CXX` questo modo:

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

Vedi anche introduzione a Cmake .

Compilatori online

Vari siti Web forniscono accesso online ai compilatori C ++. Le funzionalità del compilatore online variano in modo significativo da un sito all'altro, ma in genere consentono di eseguire le seguenti operazioni:

- Incolla il tuo codice in un modulo Web nel browser.
- Seleziona alcune opzioni del compilatore e compila il codice.
- Raccogli il compilatore e / o l'output del programma.

Il comportamento del sito Web del compilatore online è in genere piuttosto restrittivo poiché consente a chiunque di eseguire compilatori ed eseguire codice arbitrario sul proprio lato server, mentre l'esecuzione ordinariamente remota di codice arbitrario viene considerata come vulnerabilità.

I compilatori online possono essere utili per i seguenti scopi:

- Esegui un piccolo snippet di codice da una macchina che non ha compilatore C ++ (smartphone, tablet, ecc.).
- Assicurati che il codice venga compilato correttamente con diversi compilatori e funzioni allo stesso modo indipendentemente dal compilatore con cui è stato compilato.
- Impara o insegna le basi del C ++.
- Impara le moderne funzionalità C ++ (C ++ 14 e C ++ 17 nel prossimo futuro) quando il compilatore C ++ aggiornato non è disponibile sul computer locale.
- Trova un bug nel tuo compilatore confrontandolo con una vasta serie di altri compilatori. Controlla se un bug del compilatore è stato corretto nelle versioni future, che non sono disponibili sul tuo computer.
- Risolvi i problemi del giudice online.

Quali compilatori online **non** dovrebbero essere utilizzati per:

- Sviluppa applicazioni complete (anche piccole) usando C ++. Solitamente i compilatori online non consentono il collegamento con librerie di terze parti o download di artefatti di build.
- Esegui calcoli intensivi. Le risorse di elaborazione sul lato server sono limitate, quindi qualsiasi programma fornito dall'utente verrà ucciso dopo pochi secondi di esecuzione. Il tempo di esecuzione consentito è di solito sufficiente per il test e l'apprendimento.
- Attacca il server di compilazione stesso o qualsiasi host di terze parti in rete.

Esempi:

> Dichiarazione di non responsabilità: gli autori della documentazione non sono affiliati con le risorse elencate di seguito. I siti Web sono elencati in ordine alfabetico.

- http://codepad.org/ Compilatore online con condivisione del codice. La modifica del codice dopo la compilazione con un avvertimento o errore del codice sorgente non funziona così bene.
- http://coliru.stacked-crooked.com/ Compilatore online per il quale si specifica la riga di

comando. Fornisce entrambi i compilatori GCC e Clang per l'uso.
- http://cpp.sh/ - Compilatore online con supporto per C ++ 14. Non ti permette di modificare la riga di comando del compilatore, ma alcune opzioni sono disponibili tramite i controlli della GUI.
- https://gcc.godbolt.org/ - Fornisce un ampio elenco di versioni del compilatore, architetture e output di disassemblaggio. Molto utile quando è necessario esaminare in che modo viene compilato il codice da diversi compilatori. GCC, Clang, MSVC (`CL`), Intel compiler (`icc`), ELLCC e Zapcc sono presenti, con uno o più di questi compilatori disponibili per ARM, ARMv8 (come ARM64), Atmel AVR, MIPS, MIPS64, MSP430, PowerPC , x86 e x64 architecti. Gli argomenti della riga di comando del compilatore possono essere modificati.
- https://ideone.com/ - Ampiamente utilizzato in rete per illustrare il comportamento del frammento di codice. Fornisce sia GCC e Clang per l'uso, ma non consente di modificare la riga di comando del compilatore.
- http://melpon.org/wandbox - Supporta numerose versioni del compilatore Clang e GNU / GCC.
- http://onlinegdb.com/ - Un IDE estremamente minimalista che include un editor, un compilatore (gcc) e un debugger (gdb).
- http://rextester.com/ - Fornisce compilatori Clang, GCC e Visual Studio per C e C ++ (insieme ai compilatori per altre lingue), con la libreria Boost disponibile per l'uso.
- http://tutorialspoint.com/compile_cpp11_online.php - Una shell UNIX completa con GCC e un esploratore di progetti user-friendly.
- http://webcompiler.cloudapp.net/ - Compilatore di Visual Studio 2015 online, fornito da Microsoft come parte di RiSE4fun.

Il processo di compilazione C ++

Quando sviluppi un programma C ++, il passo successivo è compilare il programma prima di eseguirlo. La compilazione è il processo che converte il programma scritto in un linguaggio leggibile come C, C ++ ecc. In un codice macchina, compreso direttamente dall'unità di elaborazione centrale. Ad esempio, se si ha un file di codice sorgente C ++ denominato prog.cpp e si esegue il comando compile,

```
g++ -Wall -ansi -o prog prog.cpp
```

Ci sono 4 fasi principali coinvolte nella creazione di un file eseguibile dal file sorgente.

1. Il C ++ il preprocessore prende un file di codice sorgente C ++ e tratta le intestazioni (#include), le macro (#define) e altre direttive del preprocessore.

2. Il file di codice sorgente C ++ espanso prodotto dal preprocessore C ++ viene compilato nel linguaggio assembly per la piattaforma.

3. Il codice assembler generato dal compilatore viene assemblato nel codice oggetto per la piattaforma.

4. Il file di codice oggetto prodotto dall'assemblatore è collegato insieme
con i file di codice oggetto per tutte le funzioni di libreria utilizzate per produrre una libreria o

un file eseguibile.

Pre-elaborazione

Il preprocessore gestisce le direttive del preprocessore, come #include e #define. È agnostico della sintassi del C ++, motivo per cui deve essere usato con cautela.

Funziona su un file sorgente C ++ alla volta sostituendo le direttive #include con il contenuto dei rispettivi file (che di solito è solo dichiarazioni), sostituendo le macro (#define) e selezionando porzioni di testo diverse a seconda di #if, direttive #ifdef e #ifndef.

Il preprocessore funziona su un flusso di token di preelaborazione. La sostituzione delle macro è definita come sostituzione di token con altri token (l'operatore ## consente di unire due token quando ha senso).

Dopo tutto ciò, il preprocessore produce un singolo output che è un flusso di token risultante dalle trasformazioni descritte sopra. Aggiunge anche alcuni marcatori speciali che dicono al compilatore da dove proviene ogni riga in modo che possa usare quelli per produrre messaggi di errore sensibili.

Alcuni errori possono essere prodotti in questa fase con un uso intelligente delle direttive #if e #error.

Usando il flag sotto il compilatore, possiamo fermare il processo in fase di pre-elaborazione.

```
g++ -E prog.cpp
```

Compilazione

La fase di compilazione viene eseguita su ciascun output del preprocessore. Il compilatore analizza il codice sorgente C ++ puro (ora senza alcuna direttiva preprocessore) e lo converte in codice assembly. Quindi richiama il back-end sottostante (assemblatore in toolchain) che assembla quel codice in codice macchina producendo un file binario effettivo in qualche formato (ELF, COFF, a.out, ...). Questo file oggetto contiene il codice compilato (in forma binaria) dei simboli definiti nell'input. I simboli nei file oggetto sono indicati per nome.

I file oggetto possono fare riferimento a simboli che non sono definiti. Questo è il caso quando si usa una dichiarazione e non si fornisce una definizione per essa. Il compilatore non si preoccupa di questo, e produrrà felicemente il file oggetto finché il codice sorgente è ben formato.

I compilatori di solito ti permettono di interrompere la compilazione a questo punto. Questo è molto utile perché con esso puoi compilare ogni file del codice sorgente separatamente. Il vantaggio che offre è che non è necessario ricompilare tutto se si modifica solo un singolo file.

I file oggetto prodotti possono essere inseriti in archivi speciali chiamati librerie statiche, per un riutilizzo più semplice in seguito.

È in questa fase che vengono riportati errori di compilazione "regolari", come errori di sintassi o errori di risoluzione del sovraccarico non riusciti.

Per interrompere il processo dopo la fase di compilazione, possiamo usare l'opzione -S:

```
g++ -Wall -ansi -S prog.cpp
```

assemblaggio

L'assemblatore crea codice oggetto. Su un sistema UNIX è possibile visualizzare file con suffisso .o (.OBJ su MSDOS) per indicare i file di codice oggetto. In questa fase l'assemblatore converte i file oggetto dal codice assembly in istruzioni a livello macchina e il file creato è un codice oggetto rilocabile. Quindi, la fase di compilazione genera il programma oggetto rilocabile e questo programma può essere utilizzato in luoghi diversi senza doverlo compilare nuovamente.

Per interrompere il processo dopo la fase di assemblaggio, è possibile utilizzare l'opzione -c:

```
g++ -Wall -ansi -c prog.cpp
```

Collegamento

Il linker è ciò che produce l'output finale della compilazione dai file oggetto prodotti dall'assemblatore. Questo output può essere una libreria condivisa (o dinamica) (e mentre il nome è simile, non hanno molto in comune con le librerie statiche menzionate in precedenza) o un eseguibile.

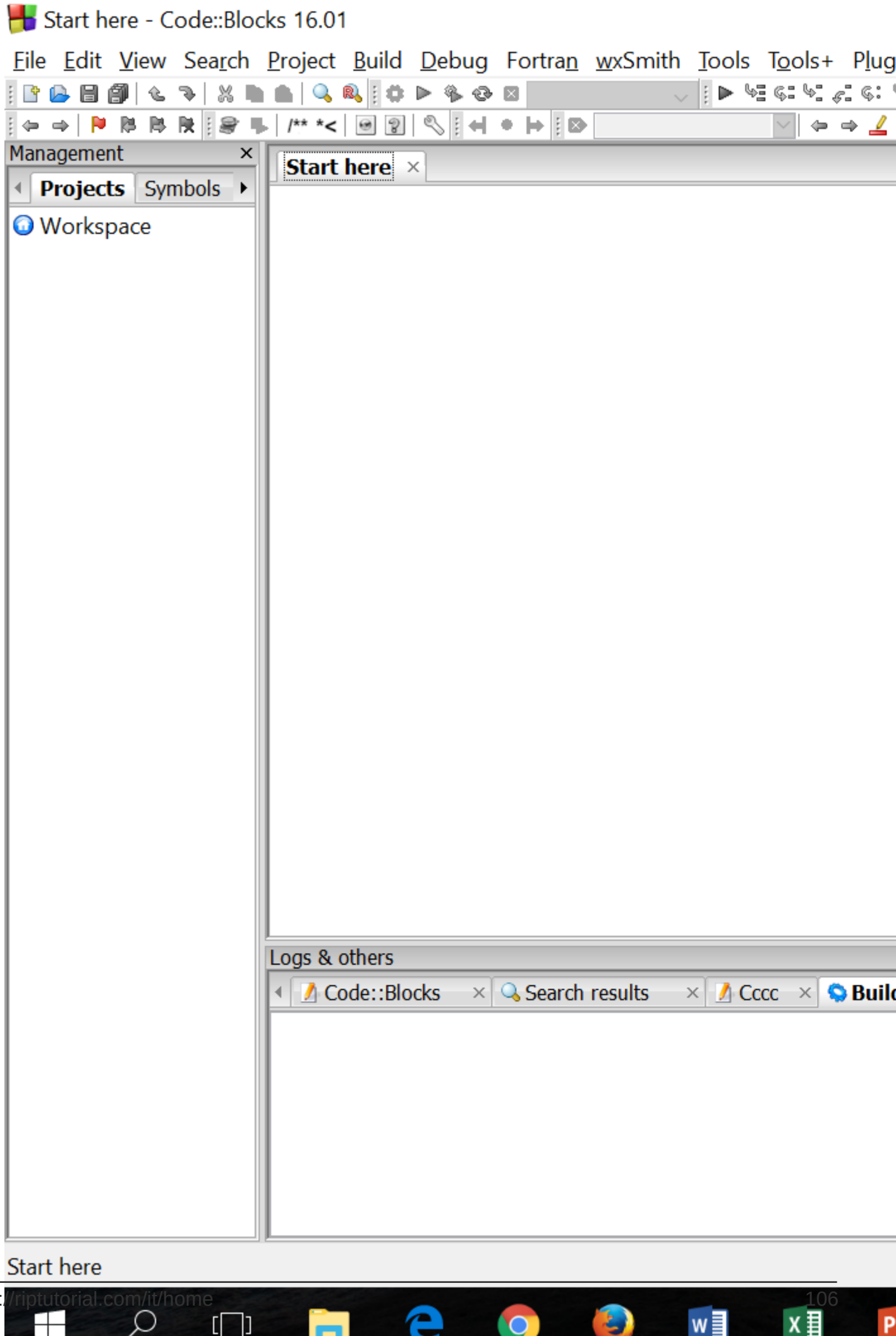
Collega tutti i file oggetto sostituendo i riferimenti a simboli non definiti con gli indirizzi corretti. Ognuno di questi simboli può essere definito in altri file oggetto o nelle librerie. Se sono definiti in librerie diverse dalla libreria standard, è necessario comunicarlo al linker.

In questa fase gli errori più comuni mancano di definizioni o definizioni duplicate. Il primo significa che le definizioni non esistono (cioè non sono scritte), o che i file oggetto o le librerie in cui risiedono non sono stati dati al linker. Quest'ultima è ovvia: lo stesso simbolo è stato definito in due diversi file o librerie di oggetti.

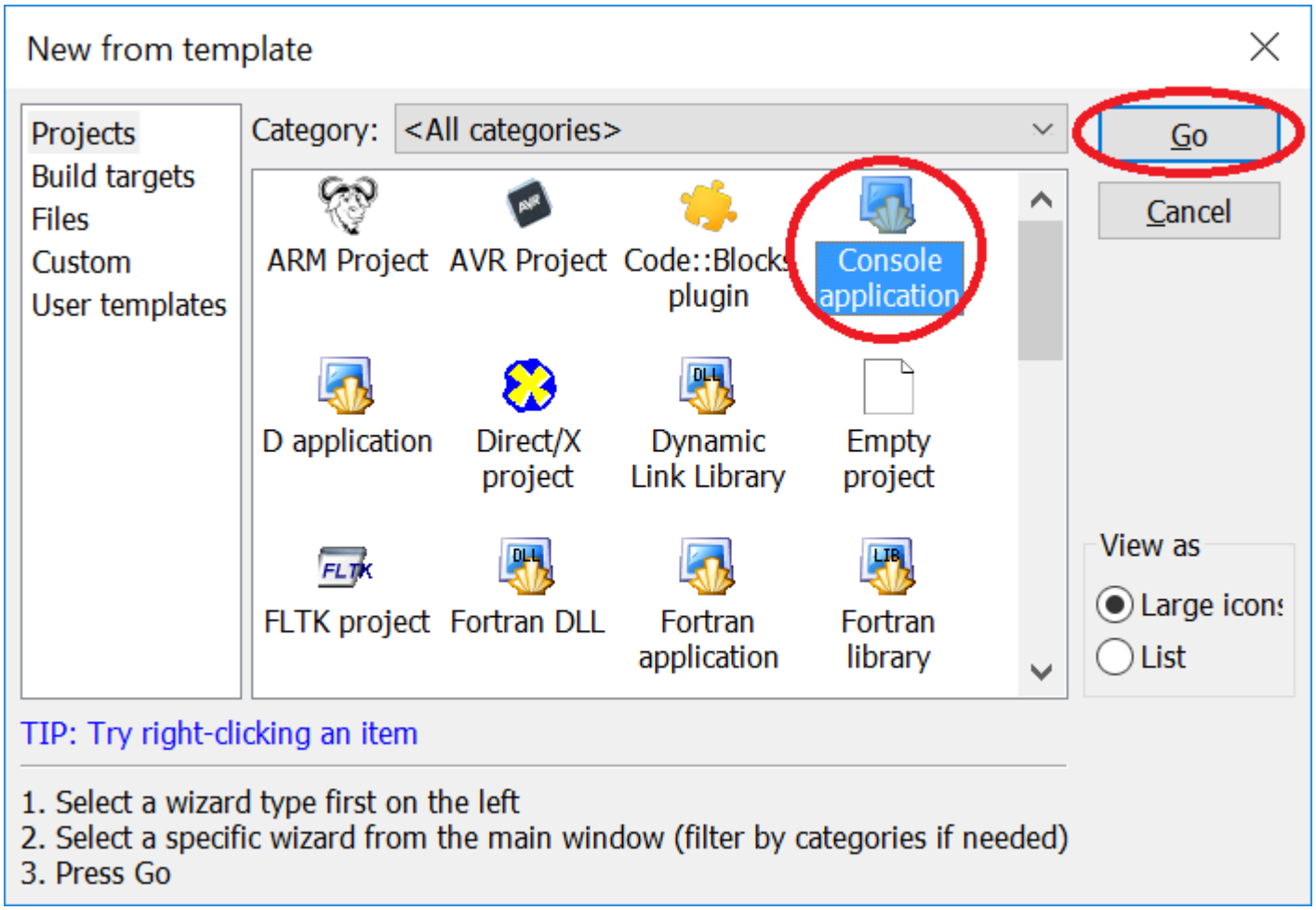
Compilazione con codice :: Blocchi (interfaccia grafica)

1. Scarica e installa il codice :: Blocchi qui . Se sei su Windows, fai attenzione a selezionare un file per il quale il nome contiene `mingw` , gli altri file non installano alcun compilatore.

2. Apri codice :: Blocchi e fai clic su "Crea un nuovo progetto":

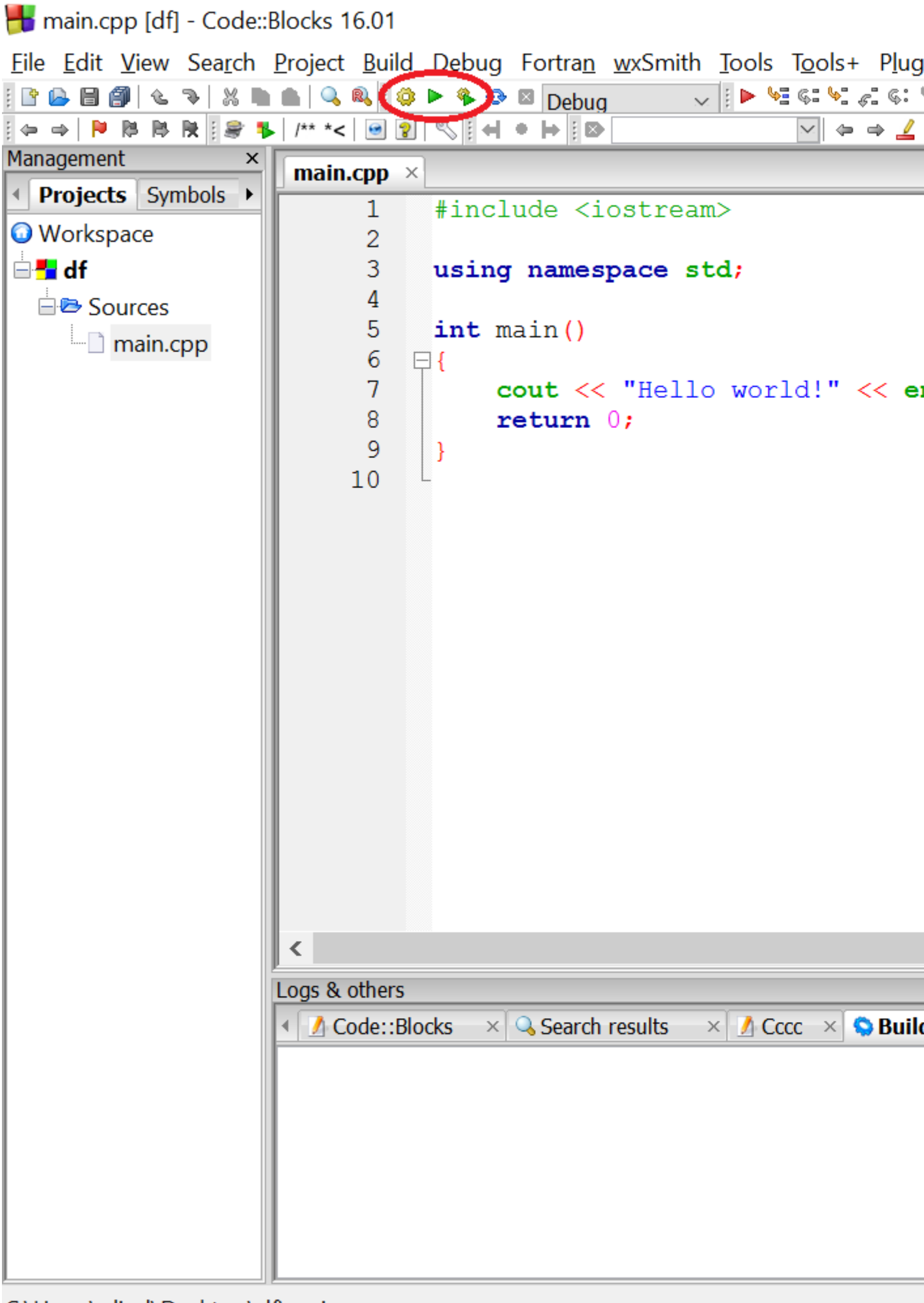

Start here - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Management
Projects
Symbols
Workspace
Start here
Logs & others
Code::Blocks
Search results
Cccc
Start here

3. Seleziona "Applicazione console" e fai clic su "Vai":



4. Fare clic su "Avanti", selezionare "C ++", fare clic su "Avanti", selezionare un nome per il progetto e scegliere una cartella in cui salvarlo, fare clic su "Avanti" e quindi fare clic su "Fine".

5. Ora puoi modificare e compilare il tuo codice. Un codice predefinito che stampa "Hello world!" nella console è già lì. Per compilare e / o eseguire il tuo programma, premi uno dei tre pulsanti di compilazione / esecuzione nella barra degli strumenti:

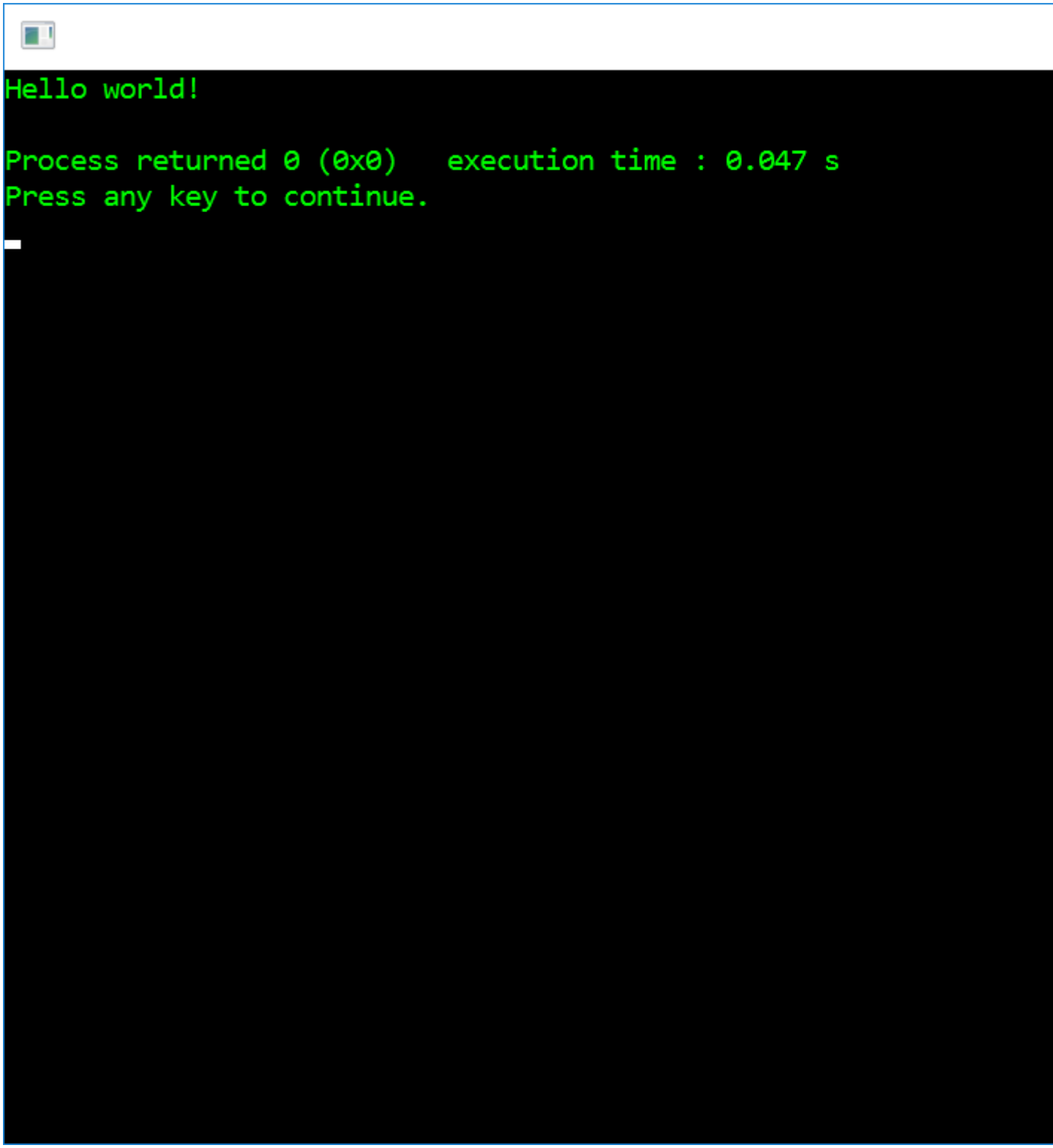

main.cpp [df] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Debug
Management
Projects Symbols
Workspace
df
Sources
main.cpp
main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7 cout << "Hello world!" << en
8 return 0;
9 }
10
Logs & others
Code::Blocks
Search results
Cccc
C:\Users\glind\Desktop\df\main.cpp

Per compilare senza correre, premere , per eseguire senza compilare di nuovo, premere e per compilare e quindi eseguire, premere .

e per compilare e quindi eseguire, premere .

e per compilare e quindi eseguire, premere .

Compilazione ed esecuzione del predefinito "Hello world!" il codice dà il seguente risultato:



Leggi Compilazione e costruzione online:

https://riptutorial.com/it/cplusplus/topic/4708/compilazione-e-costruzione

Capitolo 15: Comportamento definito dall'implementazione

Examples

Char potrebbe essere non firmato o firmato

Lo standard non specifica se `char` deve essere firmato o non firmato. Diversi compilatori lo implementano in modo diverso, o potrebbero consentire di cambiarlo utilizzando un interruttore della riga di comando.

Dimensione dei tipi integrali

I seguenti tipi sono definiti come *tipi interi* :

- `char`
- Tipi interi con segno
- Tipi interi senza segno
- `char16_t` e `char32_t`
- `bool`
- `wchar_t`

Ad eccezione di `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)` , che è diviso tra § 3.9.1.1 [basic.fundamental / 1] e § 5.3.3.1 [expr.sizeof], e `sizeof(bool)` , che è interamente definito dall'implementazione e non ha dimensioni minime, i requisiti minimi di dimensione di questi tipi sono riportati nella sezione § 3.9.1 [basic.fundamental] dello standard, e devono essere dettagliati di seguito.

Dimensione del `char`

Tutte le versioni dello standard C ++ specificano, nel § 5.3.3.1, che `sizeof` rendimenti `1` per `unsigned char` , `signed char` e `char` (è implementazione definita se il `char` tipo è `signed` o `unsigned`).

C ++ 14

`char` è abbastanza grande da rappresentare 256 valori diversi, per essere adatto alla memorizzazione di unità di codice UTF-8.

Dimensione dei tipi interi con segno e senza segno

Lo standard specifica, in § 3.9.1.2, che nella lista dei *tipi interi con* `signed char` *standard* , costituiti

da `signed char` , `short int` , `int` , `long int` e `long long int` , ciascun tipo fornirà almeno lo stesso spazio di archiviazione di quelli precedenti nella lista. Inoltre, come specificato nel § 3.9.1.3, ognuno di questi tipi ha un corrispondente *intero tipo unsigned standard* , `unsigned char` , `unsigned short int` , `unsigned int` , `unsigned long int` , `unsigned long long int` , che ha le stesse dimensioni e allineamento di il suo corrispondente tipo firmato. Inoltre, come specificato al § 3.9.1.1, `char` ha le stesse dimensioni e requisiti di allineamento sia del `signed char` `unsigned char` .

C ++ 11

Prima di C ++ 11, `long long` e `unsigned long long` non erano ufficialmente parte dello standard C ++. Tuttavia, dopo la loro introduzione a C, in C99, molti compilatori supportati per `long long` come un *tipo intero con* `unsigned long long` *esteso* e `unsigned long long` come un *tipo intero senza segno esteso* , con le stesse regole dei tipi C.

Lo standard garantisce quindi che:

```
1 == sizeof(char)  == sizeof(signed char) == sizeof(unsigned char)
  <= sizeof(short) == sizeof(unsigned short)
  <= sizeof(int)   == sizeof(unsigned int)
  <= sizeof(long)  == sizeof(unsigned long)
```

C ++ 11

```
 <= sizeof(long long) == sizeof(unsigned long long)
```

Le dimensioni minime specifiche per ciascun tipo non sono indicate dallo standard. Invece, ogni tipo ha un intervallo minimo di valori che può supportare, che è, come specificato nel § 3.9.1.3, ereditato dallo standard C, in §5.2.4.2.1. La dimensione minima di ciascun tipo può essere approssimativamente dedotta da questo intervallo, determinando il numero minimo di bit richiesti; si noti che per qualsiasi piattaforma data, l'intervallo supportato effettivo di qualsiasi tipo potrebbe essere maggiore del minimo. Nota che per i tipi firmati, gli intervalli corrispondono al complemento a uno, non al complemento a due più comunemente usato; questo per consentire a una gamma più ampia di piattaforme di rispettare lo standard.

genere	Intervallo minimo	Bit minimi richiesti
`signed char`	Da -127 a 127 (- (2 ^{7} - 1) a (2 ^{7} - 1))	8
`unsigned char`	Da 0 a 255 (da 0 a 2 ^{8} - 1)	8
`signed short`	Da -32.767 a 32.767 (- (2 ^{15} - 1) a (2 ^{15} - 1))	16
`unsigned short`	Da 0 a 65.535 (da 0 a 2 ^{16} - 1)	16
`signed int`	Da -32.767 a 32.767 (- (2 ^{15} - 1) a (2 ^{15} - 1))	16
`unsigned int`	Da 0 a 65.535 (da 0 a 2 ^{16} - 1)	16

genere	Intervallo minimo	Bit minimi richiesti
`signed long`	-2.147.483.647 a 2.147.483.647 (- (2 ^{31} - 1) a (2 ^{31} - 1))	32
`unsigned long`	Da 0 a 4.294.967.295 (da 0 a 2 ^{32} - 1)	32

C ++ 11

genere	Intervallo minimo	Bit minimi richiesti
`signed long long`	-9.223.372.036.854.775.807 a 9.223.372.036.854.775.807 (- (2 ^{63} - 1) a (2 ^{63} - 1))	64
`unsigned long long`	Da 0 a 18.446.744.073,709,551,615 (da 0 a 2 ^{64} - 1)	64

Dato che ogni tipo può essere maggiore del suo requisito di dimensioni minime, i tipi possono differire nelle dimensioni tra le implementazioni. L'esempio più notevole di questo è con i modelli di dati LP64 a 64 bit e LLP64, dove i sistemi LLP64 (come ad esempio Windows a 64 bit) hanno 32 bit `int`s e `long` s, e sistemi LP64 (come ad esempio Linux a 64 bit) hanno 32-bit `int` s e 64 bit `long` s. A causa di ciò, non si può presumere che i tipi interi abbiano una larghezza fissa su tutte le piattaforme.

C ++ 11

Se sono richiesti tipi interi con larghezza fissa, utilizzare i tipi dall'intestazione `<cstdint>` , ma si noti che lo standard rende facoltativo per le implementazioni supportare i tipi di larghezza esatta `int8_t` , `int16_t` , `int32_t` , `int64_t` , `intptr_t` , `uint8_t` , `uint16_t` , `uint32_t` , `uint64_t` e `uintptr_t` .

C ++ 11

Dimensione di `char16_t` e `char32_t`

Le dimensioni di `char16_t` e `char32_t` sono definite dall'implementazione, come specificato in § 5.3.3.1, con le disposizioni di cui al § 3.9.1.5:

- `char16_t` è abbastanza grande da rappresentare qualsiasi unità di codice UTF-16 e ha le stesse dimensioni, signnessness e allineamento di `uint_least16_t` ; è quindi necessario avere una dimensione di almeno 16 bit.
- `char32_t` è abbastanza grande da rappresentare qualsiasi unità di codice UTF-32 e ha le stesse dimensioni, signnessness e allineamento di `uint_least32_t` ; è quindi necessario avere una dimensione di almeno 32 bit.

Dimensione del `bool`

La dimensione di `bool` è definita dall'implementazione e può essere o meno `1` .

Dimensione di wchar_t

`wchar_t` , come specificato in § 3.9.1.5, è un tipo distinto, il cui intervallo di valori può rappresentare ogni unità di codice distinta del più grande set di caratteri esteso tra le localizzazioni supportate. Ha le stesse dimensioni, compattezza e allineamento di uno degli altri tipi interi, che è noto come *tipo sottostante* . La dimensione di questo tipo è definita dall'implementazione, come specificato in § 5.3.3.1, e può essere, ad esempio, almeno 8, 16 o 32 bit; se un sistema supporta Unicode, ad esempio, `wchar_t` deve avere almeno 32 bit (un'eccezione a questa regola è Windows, dove `wchar_t` è 16 bit per motivi di compatibilità). È ereditato dallo standard C90, ISO 9899: 1990, § 4.1.5, con una sola riformulazione minima.

A seconda dell'implementazione, la dimensione di `wchar_t` è spesso, ma non sempre, 8, 16 o 32 bit. Gli esempi più comuni di questi sono:

- Nei sistemi Unix e Unix, `wchar_t` è a 32 bit e viene solitamente utilizzato per UTF-32.
- In Windows, `wchar_t` è a 16 bit e viene utilizzato per UTF-16.
- Su un sistema che ha solo il supporto a 8 bit, `wchar_t` è 8 bit.

C ++ 11

Se si desidera il supporto Unicode, si consiglia di utilizzare `char` per UTF-8, `char16_t` per UTF-16 o `char32_t` per UTF-32, invece di utilizzare `wchar_t` .

Modelli di dati

Come accennato in precedenza, le larghezze dei tipi interi possono differire da una piattaforma all'altra. I modelli più comuni sono i seguenti, con dimensioni specificate in bit:

Modello	`int`	`long`	pointer
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

Di questi modelli:

- Windows a 16 bit utilizzato LP32.
- Sistemi a 32 bit * nix (Unix, Linux, Mac OSX e altri sistemi operativi Unix) e Windows utilizzano ILP32.

- Windows a 64 bit utilizza LLP64.
- I sistemi a 64 bit * nix usano LP64.

Si noti, tuttavia, che questi modelli non sono specificamente menzionati nello standard stesso.

Numero di bit in un byte

In C ++, un *byte* è lo spazio occupato da un oggetto `char` . Il numero di bit in un byte è dato da `CHAR_BIT` , che è definito in `climits` e deve essere almeno 8. Mentre la maggior parte dei sistemi moderni ha byte da 8 bit, e POSIX richiede che `CHAR_BIT` sia esattamente 8, ci sono alcuni sistemi dove `CHAR_BIT` è maggiore di 8 cioè un singolo byte può essere composto da 8, 16, 32 o 64 bit.

Valore numerico di un puntatore

Il risultato del lancio di un puntatore su un numero intero utilizzando `reinterpret_cast` è definito dall'implementazione, ma "... è destinato a non sorprendere coloro che conoscono la struttura di indirizzamento della macchina sottostante."

```
int x = 42;
int* p = &x;
long addr = reinterpret_cast<long>(p);
std::cout << addr << "\n"; // prints some numeric address,
                           // probably in the architecture's native address format
```

Allo stesso modo, il puntatore ottenuto dalla conversione da un intero è anche definito dall'implementazione.

Il modo giusto per memorizzare un puntatore come numero intero è usando i tipi `uintptr_t` o `intptr_t` :

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type
#include <stdint.h>

uintptr_t uip;
```

C ++ 11

```
// There is an optional `std::uintptr_t` in C++11
#include <cstdint>

std::uintptr_t uip;
```

C ++ 11 si riferisce a C99 per la definizione `uintptr_t` (C99 standard, 6.3.2.3):

> un tipo intero senza segno con la proprietà che qualsiasi puntatore valido a `void` può essere convertito in questo tipo, quindi convertito di nuovo in puntatore a `void` , e il risultato sarà uguale al puntatore originale.

Mentre, per la maggior parte delle piattaforme moderne, si può assumere uno spazio di indirizzamento piatto e che aritmetica su `uintptr_t` è equivalente all'aritmetica su `char *` , è

completamente possibile per un'implementazione eseguire qualsiasi trasformazione quando si esegue il cast di `void *` su `uintptr_t` finché la trasformazione può essere invertito quando si esegue il ritorno da `uintptr_t` a `void *` .

Aspetti tecnici

- Sui sistemi `intptr_t` XSI (X / Open System Interfaces), sono richiesti i tipi `intptr_t` e `uintptr_t` , altrimenti sono **opzionali** .
- Nell'ambito del significato dello standard C, le funzioni non sono oggetti; non è garantito dallo standard C che `uintptr_t` possa contenere un puntatore a funzione. In ogni caso, la conformità POSIX (2.12.3) richiede che:

  > Tutti i tipi di puntatori a funzione devono avere la stessa rappresentazione del puntatore di tipo a void. La conversione di un puntatore di funzione in void * non deve alterare la rappresentazione. Un valore * vuoto risultante da tale conversione può essere riconvertito al tipo di puntatore a funzione originale, utilizzando un cast esplicito, senza perdita di informazioni.

- C99 §7.18.1:

  > Quando i nomi typedef che differiscono solo in assenza o presenza di u iniziali sono definiti, devono indicare i corrispondenti tipi firmati e non firmati come descritto in 6.2.5; un'implementazione che fornisce uno di questi tipi corrispondenti deve fornire anche l'altro.

  `uintptr_t` potrebbe avere senso se vuoi fare cose ai bit del puntatore che non puoi fare in modo sensato con un numero intero con segno.

Intervalli di tipi numerici

Gli intervalli dei tipi interi sono definiti dall'implementazione. L'intestazione `<limits>` fornisce il `std::numeric_limits<T>` che fornisce i valori minimi e massimi di tutti i tipi fondamentali. I valori soddisfano le garanzie fornite dallo standard C attraverso le `<climits>` e (> = C ++ 11) `<cinttypes>` .

- `std::numeric_limits<signed char>::min()` equivale a `SCHAR_MIN` , che è minore o uguale a -127.
- `std::numeric_limits<signed char>::max()` uguale a `SCHAR_MAX` , che è maggiore o uguale a 127.
- `std::numeric_limits<unsigned char>::max()` equivale a `UCHAR_MAX` , che è maggiore o uguale a 255.
- `std::numeric_limits<short>::min()` equivale a `SHRT_MIN` , che è minore o uguale a -32767.
- `std::numeric_limits<short>::max()` equivale a `SHRT_MAX` , che è maggiore o uguale a 32767.
- `std::numeric_limits<unsigned short>::max()` equivale a `USHRT_MAX` , che è maggiore o uguale a 65535.
- `std::numeric_limits<int>::min()` equivale a `INT_MIN` , che è minore o uguale a -32767.
- `std::numeric_limits<int>::max()` equivale a `INT_MAX` , che è maggiore o uguale a 32767.
- `std::numeric_limits<unsigned int>::max()` equivale a `UINT_MAX` , che è maggiore o uguale a 65535.
- `std::numeric_limits<long>::min()` equivale a `LONG_MIN` , che è minore o uguale a -

2147483647.

- `std::numeric_limits<long>::max()` equivale a `LONG_MAX` , che è maggiore o uguale a 2147483647.
- `std::numeric_limits<unsigned long>::max()` uguale a `ULONG_MAX` , che è maggiore o uguale a 4294967295.

C ++ 11

- `std::numeric_limits<long long>::min()` equivale a `LLONG_MIN` , che è minore o uguale a -9223372036854775807.
- `std::numeric_limits<long long>::max()` equivale a `LLONG_MAX` , che è maggiore o uguale a 9223372036854775807.
- `std::numeric_limits<unsigned long long>::max()` equivale a `ULLONG_MAX` , che è maggiore o uguale a 18446744073709551615.

Per i tipi a virgola mobile `T` , `max()` è il valore finito massimo mentre `min()` è il valore minimo positivo normalizzato. Sono forniti membri aggiuntivi per i tipi a virgola mobile, che sono anche definiti dall'implementazione ma soddisfano determinate garanzie fornite dallo standard C attraverso l'intestazione `<cfloat>` .

- Il membro `digits10` indica il numero di cifre decimali di precisione.
  - `std::numeric_limits<float>::digits10` equivale a `FLT_DIG` , che è almeno 6.
  - `std::numeric_limits<double>::digits10` equivale a `DBL_DIG` , che è almeno 10.
  - `std::numeric_limits<long double>::digits10` equivale a `LDBL_DIG` , che è almeno 10.
- Il membro `min_exponent10` è il minimo E negativo tale che 10 alla potenza E è normale.
  - `std::numeric_limits<float>::min_exponent10` equivale a `FLT_MIN_10_EXP` , che è al massimo -37.
  - `std::numeric_limits<double>::min_exponent10` equivale a `DBL_MIN_10_EXP` , che è al massimo -37. `std::numeric_limits<long double>::min_exponent10` equivale a `LDBL_MIN_10_EXP` , che è al massimo -37.
- Il membro `max_exponent10` è il massimo E tale che 10 alla potenza E è finito.
  - `std::numeric_limits<float>::max_exponent10` equivale a `FLT_MIN_10_EXP` , che è almeno 37.
  - `std::numeric_limits<double>::max_exponent10` equivale a `DBL_MIN_10_EXP` , che è almeno 37.
  - `std::numeric_limits<long double>::max_exponent10` equivale a `LDBL_MIN_10_EXP` , che è almeno 37.
- Se il membro `is_iec559` è true, il tipo è conforme a IEC 559 / IEEE 754 e il suo intervallo è quindi determinato da tale standard.

Rappresentazione del valore di tipi in virgola mobile

Lo standard richiede che il `long double` fornisca almeno la stessa precisione del `double` , che fornisce almeno la stessa precisione del `float` ; e che un `long double` può rappresentare qualsiasi valore che una `double` può rappresentare, mentre un `double` può rappresentare qualsiasi valore che un `float` può rappresentare. I dettagli della rappresentazione sono, tuttavia, definiti dall'implementazione.

Per un punto in virgola mobile di tipo `T` , `std::numeric_limits<T>::radix` specifica la radice utilizzata dalla rappresentazione di `T`

Se `std::numeric_limits<T>::is_iec559` è vero, la rappresentazione di `T` corrisponde a uno dei formati definiti da IEC 559 / IEEE 754.

Overflow durante la conversione da numero intero a numero intero con segno

Quando un intero con segno o senza segno viene convertito in un tipo di intero con segno e il suo valore non è rappresentabile nel tipo di destinazione, il valore prodotto è definito dall'implementazione. Esempio:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and
// the range of unsigned char is 0 to 255
int x = 12345;
signed char sc = x;   // sc has an implementation-defined value
unsigned char uc = x; // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Tipo sottostante (e quindi dimensione) di enum

Se il tipo sottostante non è specificato esplicitamente per un tipo di enumerazione senza ambito, viene determinato in un modo definito dall'implementazione.

```
enum E {
    RED,
    GREEN,
    BLUE,
};
using T = std::underlying_type<E>::type; // implementation-defined
```

Tuttavia, lo standard richiede che il tipo sottostante di un'enumerazione non sia maggiore di `int` meno che sia `int` sia `unsigned int` non siano in grado di rappresentare tutti i valori dell'enumerazione. Pertanto, nel codice precedente, `T` potrebbe essere `int` , `unsigned int` o `short` , ma non `long long` , per fornire alcuni esempi.

Si noti che un enum ha la stessa dimensione (restituita da `sizeof`) come tipo sottostante.

Leggi Comportamento definito dall'implementazione online:
https://riptutorial.com/it/cplusplus/topic/1363/comportamento-definito-dall-implementazione

Capitolo 16: Comportamento indefinito

introduzione

Cos'è il comportamento non definito (UB)? Secondo lo standard ISO C ++ (§1.3.24, N4296), è "un comportamento per il quale questo Standard internazionale non impone alcun requisito".

Ciò significa che quando un programma incontra UB, è permesso di fare tutto ciò che vuole. Questo spesso significa un crash, ma potrebbe semplicemente non fare nulla, far volare via demoni dal demone , o persino *sembrare* che funzioni correttamente!

Inutile dire che dovresti evitare di scrivere codice che invoca UB.

Osservazioni

Se un programma contiene un comportamento non definito, lo standard C ++ non pone alcun vincolo al suo comportamento.

- Potrebbe sembrare che funzioni come previsto dallo sviluppatore, ma potrebbe anche bloccarsi o produrre risultati strani.
- Il comportamento può variare tra le esecuzioni dello stesso programma.
- Qualsiasi parte del programma potrebbe funzionare male, incluse le linee che precedono la linea che contiene un comportamento non definito.
- L'implementazione non è richiesta per documentare il risultato di un comportamento non definito.

Un'implementazione *può* documentare il risultato di un'operazione che produce un comportamento non definito secondo lo standard, ma un programma che dipende da tale comportamento documentato non è portabile.

Perché esiste un comportamento non definito

Intuitivamente, il comportamento non definito è considerato un aspetto negativo in quanto tali errori non possono essere gestiti in modo gentile attraverso, ad esempio, gestori di eccezioni.

Ma lasciare un certo comportamento indefinito è in realtà parte integrante della promessa del C ++ "non si paga per ciò che non si usa". Un comportamento indefinito consente al compilatore di assumere che lo sviluppatore sappia cosa sta facendo e di non introdurre il codice per verificare gli errori evidenziati negli esempi precedenti.

Trovare ed evitare comportamenti indefiniti

Alcuni strumenti possono essere utilizzati per scoprire comportamenti non definiti durante lo sviluppo:

- La maggior parte dei compilatori dispone di flag di avviso per avvisare di alcuni casi di comportamento non definito al momento della compilazione.

- Le versioni più recenti di gcc e clang includono un cosiddetto indicatore "Undefined Behavior Sanitizer" (`-fsanitize=undefined`) che verificherà il comportamento indefinito in fase di esecuzione, a un costo prestazionale.
- `lint` strumenti simili a sfilacciamenti possono eseguire analisi del comportamento indefinite più approfondite.

Comportamento indefinito, non specificato e definito dall'implementazione

Dalla norma C ++ 14 (ISO / IEC 14882: 2014) sezione 1.9 (Esecuzione del programma):

1. Le descrizioni semantiche di questo standard internazionale definiscono una macchina astratta non parametrica parametrizzata. [TAGLIO]

2. Alcuni aspetti e operazioni della macchina astratta sono descritti in questo Standard Internazionale come **definito dall'implementazione** (ad esempio, `sizeof(int)`). Questi costituiscono *i parametri della macchina astratta* . Ogni implementazione deve includere la documentazione che descrive le sue caratteristiche e il comportamento in questi aspetti. [TAGLIO]

3. Alcuni altri aspetti e operazioni della macchina astratta sono descritti in questo Standard Internazionale come **non specificati** (ad esempio, la valutazione delle espressioni in un *nuovo inizializzatore* se la funzione di allocazione non riesce ad allocare memoria). Ove possibile, questo standard internazionale definisce un insieme di comportamenti consentiti. Questi definiscono gli aspetti non deterministici della macchina astratta. Un'istanza della macchina astratta può quindi avere più di una possibile esecuzione per un dato programma e un dato input.

4. Alcune altre operazioni sono descritte in questo standard internazionale come **non definite** (o esempio, l'effetto del tentativo di modificare un oggetto `const`). [*Nota* : questo Standard Internazionale non impone alcun requisito sul comportamento dei programmi che contengono comportamenti non definiti. - *nota finale*]

Examples

Leggere o scrivere attraverso un puntatore nullo

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

Questo **comportamento non** è **definito** , poiché un puntatore nullo non punta a nessun oggetto valido, quindi non vi è alcun oggetto su `*ptr` in cui scrivere.

Sebbene questo spesso causi un errore di segmentazione, non è definito e può succedere di tutto.

Nessuna dichiarazione di reso per una funzione con un tipo di reso non vuoto

Omettere l'istruzione `return` in una funzione che ha un tipo restituito che non è `void` è un **comportamento indefinito** .

```
int function() {
    // Missing return statement
}

int main() {
    function(); //Undefined Behavior
}
```

I compilatori più moderni emettono un avviso in fase di compilazione per questo tipo di comportamento indefinito.

Nota: `main` è l'unica eccezione alla regola. Se `main` non ha un'istruzione `return` , il compilatore inserisce automaticamente `return 0;` per te, quindi può essere tranquillamente tralasciato.

Modifica di una stringa letterale

C ++ 11

```
char *str = "hello world";
str[0] = 'H';
```

`"hello world"` è una stringa letterale, quindi modificarla dà un comportamento indefinito.

L'inizializzazione di `str` nell'esempio precedente è stata formalmente deprecata (programmata per la rimozione da una versione futura dello standard) in C ++ 03. Un certo numero di compilatori prima del 2003 potrebbe emettere un avviso a riguardo (ad esempio una conversione sospetta). Dopo il 2003, i compilatori in genere mettono in guardia su una conversione deprecata.

C ++ 11

L'esempio sopra è illegale e risulta in una diagnostica del compilatore, in C ++ 11 e versioni successive. Un esempio simile può essere costruito per mostrare un comportamento non definito consentendo esplicitamente la conversione del tipo, come ad esempio:

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

Accedere a un indice fuori dai limiti

È un **comportamento indefinito** accedere a un indice che è fuori limite per un array (o contenitore di libreria standard per quello scopo, poiché sono tutti implementati utilizzando un array *raw*):

```
 int array[] = {1, 2, 3, 4, 5};
 array[5] = 0;  // Undefined behavior
```

È *consentito* avere un puntatore che punta alla fine dell'array (in questo caso `array + 5`), non è possibile dereferenziarlo, poiché non è un elemento valido.

```
const int *end = array + 5;  // Pointer to one past the last index
for (int *p = array; p != end; ++p)
  // Do something with `p`
```

In generale, non ti è permesso creare un puntatore fuori limite. Un puntatore deve puntare a un elemento all'interno dell'array o uno dopo la fine.

Divisione intera per zero

```
int x = 5 / 0;    // Undefined behavior
```

La divisione per `0` è matematicamente indefinita e, in quanto tale, ha senso che si tratti di un comportamento non definito.

Però:

```
float x = 5.0f / 0.0f;   // x is +infinity
```

La maggior parte dell'attuazione implementa IEEE-754, che definisce la divisione in virgola mobile per zero per restituire `NaN` (se il numeratore è `0.0f`), `infinity` (se il numeratore è positivo) o `-infinity` (se il numeratore è negativo).

Overflow intero firmato

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

> Se durante la valutazione di un'espressione, il risultato non è definito matematicamente o non rientra nell'intervallo di valori rappresentabili per il suo tipo, il comportamento non è definito.

(C ++ 11 Standard paragrafo 5/4)

Questo è uno dei più cattivi, dato che di solito produce un comportamento riproducibile e non-schiantarsi così gli sviluppatori potrebbero essere tentati di fare molto affidamento sul comportamento osservato.

D'altro canto:

```
unsigned int x = UINT_MAX + 1;

// x is 0
```

è ben definito poiché:

> Gli interi senza segno, dichiarati senza segno, obbediscono alle leggi dell'aritmetico modulo `2^n` dove `n` è il numero di bit nella rappresentazione del valore di quella particolare dimensione del numero intero.

(C ++ 11 paragrafo standard 3.9.1 / 4)

A volte i compilatori possono sfruttare un comportamento indefinito e ottimizzare

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Qui dal momento che un overflow di interi con segno non è definito, il compilatore è libero di presumere che non possa mai accadere e quindi può ottimizzare il blocco "se"

Utilizzando una variabile locale non inizializzata

```
int a;
std::cout << a; // Undefined behavior!
```

Ciò si traduce in un **comportamento indefinito** , perché `a` non è inizializzato.

È spesso, erroneamente, affermato che ciò è dovuto al fatto che il valore è "indeterminato" o "qualsiasi valore si trovasse in quella posizione di memoria precedente". Tuttavia, è l'atto di accedere al valore di `a` nell'esempio precedente che fornisce un comportamento non definito. In pratica, stampare un "valore spazzatura" è un sintomo comune in questo caso, ma questa è solo una possibile forma di comportamento non definito.

Sebbene sia molto improbabile nella pratica (poiché dipende dal supporto hardware specifico), il compilatore potrebbe ugualmente elettrocurare il programmatore durante la compilazione del codice di esempio sopra. Con un tale compilatore e supporto hardware, una simile risposta a comportamenti non definiti aumenterebbe notevolmente la comprensione media del programmatore (vivente) del vero significato di comportamento non definito - il che è che lo standard non pone alcun vincolo sul comportamento risultante.

C ++ 14

L'utilizzo di un valore indeterminato di tipo `unsigned char` non produce un comportamento non definito se il valore viene utilizzato come:

- il secondo o terzo operando dell'operatore condizionale ternario;
- l'operando corretto dell'operatore virgola incorporato;
- l'operando di una conversione in `unsigned char` ;
- l'operando corretto dell'operatore di assegnazione, se l'operando di sinistra è anch'esso di tipo `unsigned char` ;
- l'inizializzatore per un oggetto `unsigned char` ;

o se il valore è scartato. In questi casi, il valore indeterminato si propaga semplicemente al risultato dell'espressione, se applicabile.

Si noti che una variabile `static` è **sempre** inizializzata a zero (se possibile):

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

Definizioni multiple non identiche (la regola One Definition)

Se una classe, enum, funzione inline, modello o membro di un modello ha un collegamento esterno ed è definita in più unità di traduzione, tutte le definizioni devono essere identiche o il comportamento non è definito in base alla One Definition Rule (ODR) .

`foo.h` :

```
class Foo {
  public:
    double x;
  private:
    int y;
};

Foo get_foo();
```

`foo.cpp` :

```
#include "foo.h"
Foo get_foo() { /* implementation */ }
```

`main.cpp` :

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
  public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

Il programma sopra esposto presenta un comportamento indefinito perché contiene due definizioni della classe `::Foo` , che ha un collegamento esterno, in diverse unità di traduzione, ma le due definizioni non sono identiche. A differenza della ridefinizione di una classe all'interno della *stessa* unità di traduzione, non è richiesto che questo problema venga diagnosticato dal compilatore.

Associazione errata di allocazione e deallocazione di memoria

Un oggetto può essere deallocato solo da `delete` se è stato assegnato da un `new` e non è un array. Se l'argomento da `delete` non è stato restituito da `new` o è un array, il comportamento non è definito.

Un oggetto può essere deallocato solo da `delete[]` se è stato assegnato da `new` ed è un array. Se l'argomento per `delete[]` non è stato restituito da `new` o non è un array, il comportamento non è definito.

Se l'argomento `free` non è stato restituito da `malloc` , il comportamento non è definito.

```
int* p1 = new int;
delete p1;      // correct
// delete[] p1; // undefined
// free(p1);    // undefined

int* p2 = new int[10];
delete[] p2;    // correct
// delete p2;   // undefined
// free(p2);    // undefined

int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3);       // correct
// delete p3;   // undefined
// delete[] p3; // undefined
```

Tali problemi possono essere evitati evitando completamente `malloc` e `free` nei programmi C ++, preferendo i puntatori intelligenti della libreria standard su raw `new` e `delete` , e preferendo `std::vector` e `std::string` su raw `new` e `delete[]` .

Accedere a un oggetto come il tipo sbagliato

Nella maggior parte dei casi, è illegale accedere a un oggetto di un tipo come se fosse di un tipo diverso (ignorando i qualificatori di cv). Esempio:

```
float x = 42;
int y = reinterpret_cast<int&>(x);
```

Il risultato è un comportamento indefinito.

Ci sono alcune eccezioni a questa *rigida* regola di *aliasing* :

- È possibile accedere a un oggetto di tipo classe come se fosse di un tipo che è una classe base del tipo effettivo di classe.
- Qualsiasi tipo è accessibile come `char` o `unsigned char` , ma il contrario non è vero: non è possibile accedere a un array di caratteri come se fosse un tipo arbitrario.
- Un tipo intero con segno è accessibile come corrispondente senza segno e *viceversa* .

Una regola correlata è che se una funzione membro non statica viene chiamata su un oggetto che in realtà non ha lo stesso tipo della classe di definizione della funzione o di una classe derivata, si verifica un comportamento non definito. Questo è vero anche se la funzione non accede all'oggetto.

```
struct Base {
};
struct Derived : Base {
    void f() {}
};
struct Unrelated {};
Unrelated u;
Derived& r1 = reinterpret_cast<Derived&>(u); // ok
r1.f();                                      // UB
Base b;
Derived& r2 = reinterpret_cast<Derived&>(b); // ok
r2.f();                                      // UB
```

Overflow a virgola mobile

Se un'operazione aritmetica che produce un tipo a virgola mobile produce un valore che non è compreso nell'intervallo di valori rappresentabili del tipo di risultato, il comportamento non è definito secondo lo standard C ++, ma può essere definito da altri standard a cui la macchina potrebbe conformarsi, come IEEE 754.

```
float x = 1.0;
for (int i = 0; i < 10000; i++) {
    x *= 10.0; // will probably overflow eventually; undefined behavior
}
```

Chiamare membri (puri) virtuali da Costruttore o Distruttore

Lo standard (10.4) afferma:

> Le funzioni membro possono essere chiamate da un costruttore (o distruttore) di una classe astratta; l'effetto di effettuare una chiamata virtuale (10.3) a una pura funzione virtuale direttamente o indirettamente per l'oggetto creato (o distrutto) da tale costruttore (o distruttore) non è definito.

Più in generale, alcune autorità del C ++, ad esempio Scott Meyers, suggeriscono di non chiamare mai funzioni virtuali (anche non pure) da costruttori e dstructors.

Considera il seguente esempio, modificato dal link precedente:

```
class transaction
{
public:
    transaction(){ log_it(); }
    virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
public:
    virtual void log_it() const { /* Do something */ }
};
```

Supponiamo di creare un oggetto `sell_transaction` :

```
sell_transaction s;
```

Questo chiama implicitamente il costruttore di `sell_transaction` , che prima chiama il costruttore della `transaction` . Tuttavia, quando viene chiamato il costruttore della `transaction` , l'oggetto non è ancora del tipo `sell_transaction` , ma solo della `transaction` tipo.

Di conseguenza, la chiamata in `transaction::transaction()` a `log_it` , non farà ciò che potrebbe sembrare la cosa intuitiva, ovvero call `sell_transaction::log_it` .

- Se `log_it` è puro virtuale, come in questo esempio, il comportamento non è definito.
- Se `log_it` non è puro virtuale, verrà chiamato `transaction::log_it` .

Eliminazione di un oggetto derivato tramite un puntatore a una classe base che non ha un distruttore virtuale.

```
class base { };
class derived: public base { };

int main() {
    base* p = new derived();
    delete p; // The is undefined behavior!
}
```

Nella sezione [expr.delete] §5.3.5 / 3 lo standard dice che se viene chiamato `delete` su un oggetto il cui tipo statico non ha un distruttore `virtual` :

> se il tipo statico dell'oggetto da eliminare è diverso dal suo tipo dinamico, il tipo statico deve essere una classe base del tipo dinamico dell'oggetto da eliminare e il tipo statico deve avere un distruttore virtuale o il comportamento non è definito.

Questo è il caso indipendentemente dalla domanda se la classe derivata ha aggiunto membri di dati alla classe base.

Accedere a un riferimento ciondolante

È illegale accedere a un riferimento a un oggetto che è andato fuori portata o è stato altrimenti distrutto. Si dice che tale riferimento *penzoli* poiché non si riferisce più ad un oggetto valido.

```
#include <iostream>
int& getX() {
    int x = 42;
    return x;
}
int main() {
    int& r = getX();
    std::cout << r << "\n";
}
```

In questo esempio, la variabile locale `x` diventa fuori campo quando restituisce `getX` . (Si noti che l' *estensione a vita* non può estendere la durata di una variabile locale oltre l'ambito del blocco in cui

è definita.) Pertanto `r` è un riferimento ciondolante. Questo programma ha un comportamento indefinito, anche se può sembrare che funzioni e stampi `42` in alcuni casi.

Estendere lo spazio dei nomi `std` o` posix`

Lo standard (17.6.4.2.1 / 1) generalmente vieta di estendere lo spazio dei nomi `std` :

> Il comportamento di un programma C ++ non è definito se aggiunge dichiarazioni o definizioni allo spazio dei nomi std o ad uno spazio dei nomi all'interno dello std del namespace, se non diversamente specificato.

Lo stesso vale per `posix` (17.6.4.2.2 / 1):

> Il comportamento di un programma C ++ non è definito se aggiunge dichiarazioni o definizioni allo spazio dei nomi posix o a uno spazio dei nomi all'interno di posizione spazio dei nomi, se non diversamente specificato.

Considera quanto segue:

```
#include <algorithm>

namespace std
{
    int foo(){}
}
```

Niente nello standard proibisce l' `algorithm` (o una delle intestazioni che include) che definisce la stessa definizione, e quindi questo codice violerebbe la regola di una definizione .

Quindi, in generale, questo è proibito. Tuttavia, sono consentite eccezioni specifiche . Forse la cosa più utile è che è possibile aggiungere specializzazioni per i tipi definiti dall'utente. Quindi, per esempio, supponiamo che il tuo codice abbia

```
class foo
{
    // Stuff
};
```

Quindi quanto segue va bene

```
namespace std
{
    template<>
    struct hash<foo>
    {
    public:
        size_t operator()(const foo &f) const;
    };
}
```

Overflow durante la conversione in o dal tipo a virgola mobile

Se, durante la conversione di:

- un tipo intero in un tipo a virgola mobile,
- un tipo a virgola mobile su un tipo intero o
- un tipo a virgola mobile su un tipo a virgola mobile più corto,

il valore di origine è al di fuori dell'intervallo di valori che può essere rappresentato nel tipo di destinazione, il risultato è un comportamento non definito. Esempio:

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Trasmissione statica da base a derivata non valida

Se `static_cast` viene utilizzato per convertire un puntatore (riferimento) in una classe derivata in un puntatore (riferimento) alla classe derivata, ma l'operando non punta (si riferisce) a un oggetto del tipo di classe derivata, il comportamento è indefinito. Vedi la conversione da Base a derivata .

Chiamata di funzione tramite il tipo di puntatore a funzione non corrispondente

Per chiamare una funzione attraverso un puntatore a funzione, il tipo di puntatore della funzione deve corrispondere esattamente al tipo di funzione. Altrimenti, il comportamento non è definito. Esempio:

```
int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // undefined
```

Modifica di un oggetto const

Qualsiasi tentativo di modificare un oggetto `const` risulta in un comportamento non definito. Questo si applica alle variabili `const` , ai membri degli oggetti `const` e ai membri della classe dichiarati `const` . (Tuttavia, un `mutable` membro di un `const` oggetto non è `const` .)

Un tale tentativo può essere fatto tramite `const_cast` :

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

Un compilatore di solito inline il valore di un oggetto `const int` , quindi è possibile che questo codice compili e stampi `123` . I compilatori possono anche posizionare i valori degli oggetti `const` nella memoria di sola lettura, pertanto potrebbe verificarsi un errore di segmentazione. In ogni caso, il comportamento non è definito e il programma potrebbe fare qualsiasi cosa.

Il seguente programma nasconde un errore molto più sottile:

```
#include <iostream>

class Foo* instance;

class Foo {
  public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
  private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}
```

In questo codice, `getFoo` crea un singleton di tipo `const Foo` e il suo membro `m_x` è inizializzato a `123` . Quindi viene chiamato `do_evil` e il valore di `foo.m_x` apparentemente viene modificato in 456. Che cosa è andato storto?

Nonostante il suo nome, `do_evil` non fa nulla di particolarmente malvagio; tutto ciò che fa è chiamare un setter attraverso un `Foo*` . Ma quel puntatore punta a un oggetto `const Foo` anche se `const_cast` non è stato usato. Questo puntatore è stato ottenuto tramite il costruttore di `Foo` . Un oggetto `const` non diventa `const` finché la sua inizializzazione non è completa, quindi `this` ha tipo `Foo*` , non `const Foo*` , all'interno del costruttore.

Pertanto, si verifica un comportamento indefinito anche se non ci sono costrutti chiaramente pericolosi in questo programma.

Accesso a membri inesistenti tramite puntatore al membro

Quando si accede a un membro non statico di un oggetto tramite un puntatore al membro, se l'oggetto non contiene effettivamente il membro indicato dal puntatore, il comportamento non è definito. (Tale puntatore al membro può essere ottenuto tramite `static_cast` .)

```
struct Base { int x; };
struct Derived : Base { int y; };
int Derived::*pdy = &Derived::y;
int Base::*pby = static_cast<int Base::*>(pdy);
```

```
Base* b1 = new Derived;
b1->*pby = 42; // ok; sets y in Derived object to 42
Base* b2 = new Base;
b2->*pby = 42; // undefined; there is no y member in Base
```

Conversione da origine a base non valida per i puntatori ai membri

Quando `static_cast` viene utilizzato per convertire `TD::*` in `TB::*` , il membro a cui fa riferimento deve appartenere a una classe che è una classe base o una classe derivata di `B` Altrimenti il comportamento non è definito. Vedi Conversione da Derivata a base per i puntatori ai membri

Aritmetica del puntatore non valida

I seguenti usi dell'aritmetica del puntatore causano un comportamento non definito:

- Aggiunta o sottrazione di un intero, se il risultato non appartiene allo stesso oggetto matrice dell'operando puntatore. (Qui, si considera che l'elemento uno oltre la fine appartenga ancora alla matrice.)

```
int a[10];
int* p1 = &a[5];
int* p2 = p1 + 4; // ok; p2 points to a[9]
int* p3 = p1 + 5; // ok; p2 points to one past the end of a
int* p4 = p1 + 6; // UB
int* p5 = p1 - 5; // ok; p2 points to a[0]
int* p6 = p1 - 6; // UB
int* p7 = p3 - 5; // ok; p7 points to a[5]
```

- Sottrazione di due puntatori se non appartengono allo stesso oggetto matrice. (Di nuovo, l'elemento uno oltre la fine è considerato appartenere all'array.) L'eccezione è che due puntatori nulli possono essere sottratti, ottenendo 0.

```
int a[10];
int b[10];
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // yields 5
int *p3 = p1 + 2; // ok; p3 points to one past the end of a
int d2 = p3 - p2; // yields 7
int *p4 = &b[0];
int d3 = p4 - p1; // UB
```

- Sottrazione di due puntatori se il risultato supera lo `std::ptrdiff_t` .

- Qualsiasi aritmetica puntatore in cui il tipo di punta dell'operando non corrisponde al tipo dinamico dell'oggetto puntato (ignorando la qualifica cv). Secondo lo standard, "[in] in particolare, un puntatore a una classe base non può essere utilizzato per l'aritmetica del puntatore quando l'array contiene oggetti di un tipo di classe derivata."

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
```

```
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
auto p5 = p4 - p1;          // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter
```

Spostamento di un numero non valido di posizioni

Per l'operatore di spostamento incorporato, l'operando di destra deve essere non negativo e strettamente inferiore alla larghezza di bit dell'operando sinistro promosso. Altrimenti, il comportamento non è definito.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0;  // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                       // int must be at least 16 bits
```

Ritorno da una funzione [[Noreturn]]

C ++ 11

Esempio dallo standard, [dcl.attr.noreturn]:

```
[[ noreturn ]] void f() {
  throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
}
```

Distruggere un oggetto che è già stato distrutto

In questo esempio, un distruttore viene invocato esplicitamente per un oggetto che verrà successivamente distrutto automaticamente.

```
struct S {
    ~S() { std::cout << "destroying S\n"; }
};
int main() {
    S s;
    s.~S();
} // UB: s destroyed a second time here
```

Un problema simile si verifica quando uno `std::unique_ptr<T>` viene creato per puntare a una `T` con durata di archiviazione automatica o statica.

```
void f(std::unique_ptr<S> p);
int main() {
    S s;
    std::unique_ptr<S> p(&s);
    f(std::move(p)); // s destroyed upon return from f
}                    // UB: s destroyed
```

Un altro modo per distruggere un oggetto due volte consiste nel fatto che due `shared_ptr` s gestiscono entrambi l'oggetto senza condividere la proprietà l'uno con l'altro.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);
int main() {
    S* p = new S;
    // I want to pass the same object twice...
    std::shared_ptr<S> sp1(p);
    std::shared_ptr<S> sp2(p);
    f(sp1, sp2);
} // UB: both sp1 and sp2 will destroy s separately
// NB: this is correct:
// std::shared_ptr<S> sp(p);
// f(sp, sp);
```

Ricorsione infinita del modello

Esempio dallo standard, [temp.inst] / 17:

```
template<class T> class X {
    X<T>* p; // OK
    X<T*> a; // implicit generation of X<T> requires
             // the implicit instantiation of X<T*> which requires
             // the implicit instantiation of X<T**> which ...
};
```

Leggi Comportamento indefinito online:
https://riptutorial.com/it/cplusplus/topic/1812/comportamento-indefinito

Capitolo 17: Comportamento non specificato

Osservazioni

Se il comportamento di un costrutto non è specificato, lo standard pone alcuni vincoli sul comportamento, ma lascia una certa libertà all'implementazione, che *non* è necessaria per documentare ciò che accade in una determinata situazione. Contrasta con il comportamento definito dall'implementazione , in cui *è* richiesta l'implementazione per documentare ciò che accade e un comportamento indefinito, in cui qualsiasi cosa può accadere.

Examples

Ordine di inizializzazione di globals su TU

Mentre all'interno di un'unità di traduzione viene specificato l'ordine di inizializzazione delle variabili globali, l'ordine di inizializzazione tra le unità di traduzione non è specificato.

Quindi programma con i seguenti file

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

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

potrebbe produrre come output:

```
foobar
```

o

```
barfoo
```

Ciò potrebbe portare all'ordine di *inizializzazione statica Fiasco* .

Valore di un enum fuori limite

Se un enum di ambito viene convertito in un tipo integrale troppo piccolo per contenere il suo valore, il valore risultante non è specificato. Esempio:

```
enum class E {
    X = 1,
    Y = 1000,
};
// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

Inoltre, se un numero intero viene convertito in un enum e il valore dell'intero è al di fuori dell'intervallo dei valori dell'enumerazione, il valore risultante non è specificato. Esempio:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};
Color c = static_cast<Color>(4);
```

Tuttavia, nel prossimo esempio, il comportamento *non* è *non* specificato, poiché il valore di origine è *compreso nell'intervallo* dell'enumerazione, sebbene non sia uguale a tutti gli enumeratori:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};
Scale s = static_cast<Scale>(3);
```

Qui `s` avrà il valore 3 e sarà disuguale a `ONE` , `TWO` e `FOUR` .

Cast statico dal valore di bogus void *

Se un valore `void*` viene convertito in un puntatore al tipo di oggetto, `T*` , ma non è allineato correttamente per `T` , il valore del puntatore risultante non è specificato. Esempio:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

Il valore di `p3` non è specificato perché `p2` non può puntare a un oggetto di tipo `int` ; il suo valore non è un indirizzo correttamente allineato.

Risultato di alcune reinterpret_cast conversioni

Il risultato di un `reinterpret_cast` da un tipo di puntatore a un altro, o un tipo di riferimento a un altro, non è specificato. Esempio:

```
int f();
auto fp = reinterpret_cast<int(*)(int)>(&f); // fp has unspecified value
```

C ++ 03

Il risultato di un `reinterpret_cast` da un tipo di puntatore a un altro oggetto o un tipo di riferimento a un altro non è specificato. Esempio:

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

Tuttavia, con la maggior parte dei compilatori, questo era equivalente a `static_cast<char*>(static_cast<void*>(&x))` quindi il puntatore risultante `p` puntava al primo byte di `x` . Questo è stato reso il comportamento standard in C ++ 11. Vedi la conversione della punteggiatura di tipo per maggiori dettagli.

Risultato di alcuni confronti tra puntatori

Se due puntatori vengono confrontati utilizzando `<` , `>` , `<=` o `>=` , il risultato non è specificato nei seguenti casi:

- I puntatori indicano diversi array. (Un oggetto non array è considerato un array di dimensioni 1.)

```
int x;
int y;
const bool b1 = &x < &y;             // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];       // true
const bool b3 = &a[0] < &x;          // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                     // note: a+10 points past the end of the array
```

- I puntatori indicano lo stesso oggetto, ma per i membri con controllo di accesso diverso.

```
class A {
  public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
    bool f2() { return &x < &z; } // unspecified
  private:
    int z;
};
```

Spazio occupato da un riferimento

Un riferimento non è un oggetto e, a differenza di un oggetto, non è garantito che occupi alcuni

byte contigui di memoria. Lo standard lascia non specificato se un riferimento richiede alcuna memoria. Un certo numero di caratteristiche del linguaggio cospirano per rendere impossibile esaminare in maniera portabile qualsiasi memoria che il riferimento potrebbe occupare:

- Se `sizeof` viene applicato a un riferimento, restituisce la dimensione del tipo di riferimento, senza fornire informazioni sul fatto che il riferimento occupi qualsiasi spazio di archiviazione.
- Le matrici di riferimenti sono illegali, quindi non è possibile esaminare gli indirizzi di due elementi consecutivi di un ipotetico riferimento di matrici al fine di determinare la dimensione di un riferimento.
- Se viene preso l'indirizzo di un riferimento, il risultato è l'indirizzo del referente, quindi non possiamo ottenere un puntatore al riferimento stesso.
- Se una classe ha un membro di riferimento, il tentativo di estrarre l'indirizzo di quel membro usando `offsetof` produce un comportamento indefinito poiché tale classe non è una classe di layout standard.
- Se una classe ha un membro di riferimento, la classe non è più un layout standard, quindi tenta di accedere a qualsiasi dato utilizzato per memorizzare i risultati del riferimento in un comportamento indefinito o non specificato.

In pratica, in alcuni casi una variabile di riferimento può essere implementata in modo simile a una variabile puntatore e quindi occupa la stessa quantità di memoria di un puntatore, mentre in altri casi un riferimento non può occupare spazio poiché può essere ottimizzato. Ad esempio, in:

```
void f() {
    int x;
    int& r = x;
    // do something with r
}
```

il compilatore è libero di trattare semplicemente `r` come alias per `x` e sostituire tutte le occorrenze di `r` nel resto della funzione `f` con `x` , e non allocare alcuna memoria per contenere `r` .

Ordine di valutazione degli argomenti della funzione

Se una funzione ha più argomenti, non è specificato in quale ordine vengono valutati. Il seguente codice potrebbe stampare `x = 1, y = 2` o `x = 2, y = 1` ma non specificato.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}
int get_val() {
    static int x = 0;
    return ++x;
}
int main() {
    f(get_val(), get_val());
}
```

C ++ 17

In C ++ 17, l'ordine di valutazione degli argomenti delle funzioni rimane non specificato.

Tuttavia, ogni argomento di funzione viene completamente valutato e l'oggetto chiamante è garantito valutato prima che gli argomenti di una funzione siano.

```
struct from_int {
  from_int(int x) { std::cout << "from_int (" << x << ")\n"; }
};
int make_int(int x){ std::cout << "make_int (" << x << ")\n"; return x; }


void foo(from_int a, from_int b) {
}
void bar(from_int a, from_int b) {
}

auto which_func(bool b){
  std::cout << b?"foo":"bar" << "\n";
  return b?foo:bar;
}

int main(int argc, char const*const* argv) {
  which_func( true )( make_int(1), make_int(2) );
}
```

questo deve stampare:

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

o

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

potrebbe *non* stampare la `bar` dopo una qualsiasi delle `make` o `from` , e potrebbe non stampare:

```
bar
make_int(2)
make_int(1)
from_int(2)
from_int(1)
```

o simili. Prima della `bar` stampa C ++ 17 dopo che `make_int` s era legale, come facevano entrambi i `make_int` s prima di fare qualsiasi `from_int` s.

Spostato dallo stato della maggior parte delle classi di libreria standard

C ++ 11

Tutti i contenitori di libreria standard vengono lasciati in uno stato *valido ma non specificato* dopo essere stati spostati da. Ad esempio, nel codice seguente, `v2` conterrà `{1, 2, 3, 4}` dopo lo spostamento, ma non è garantito che `v1` sia vuoto.

```
int main() {
    std::vector<int> v1{1, 2, 3, 4};
    std::vector<int> v2 = std::move(v1);
}
```

Alcune classi hanno uno stato spostato con precisione definito. Il caso più importante è quello di `std::unique_ptr<T>` , che è garantito come null dopo essere stato spostato da.

Leggi Comportamento non specificato online:
https://riptutorial.com/it/cplusplus/topic/4939/comportamento-non-specificato

Capitolo 18: Concorrenza con OpenMP

introduzione

Questo argomento copre le basi della concorrenza in C ++ usando OpenMP. OpenMP è documentato in modo più dettagliato nel tag OpenMP .

Il parallelismo o la concorrenza implica l'esecuzione del codice allo stesso tempo.

Osservazioni

OpenMP non richiede intestazioni o librerie speciali in quanto è una funzionalità incorporata del compilatore. Tuttavia, se si utilizzano funzioni API OpenMP come `omp_get_thread_num()` , sarà necessario includere `omp.h` e la relativa libreria.

Le istruzioni `pragma` OpenMP vengono ignorate quando l'opzione OpenMP non è abilitata durante la compilazione. Si consiglia di fare riferimento all'opzione del compilatore nel manuale del compilatore.

- GCC utilizza `-fopenmp`
- Clang usa `-fopenmp`
- MSVC utilizza `/openmp`

Examples

OpenMP: sezioni parallele

Questo esempio illustra le basi dell'esecuzione di sezioni di codice in parallelo.

Poichè OpenMP è una funzionalità incorporata del compilatore, funziona su qualsiasi compilatore supportato senza includere alcuna libreria. Si consiglia di includere `omp.h` se si desidera utilizzare una delle funzioni API openMP.

Codice di esempio

```
std::cout << "begin ";
//    This pragma statement hints the compiler that the
//    contents within the { } are to be executed in as
//    parallel sections using openMP, the compiler will
//    generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    //    This pragma statement hints the compiler that
    //    this is a section that can be executed in parallel
    //    with other section, a single section will be executed
    //    by a single thread.
    //    Note that it is "section" as opposed to "sections" above
    #pragma omp section
```

```
    {
        std::cout << "hello " << std::endl;
        /** Do something **/
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /** Do something **/
    }
}
//    This line will not be executed until all the
//    sections defined above terminates
std::cout << "end" << std::endl;
```

Uscite

Questo esempio produce 2 possibili uscite e dipende dal sistema operativo e dall'hardware. L'output illustra anche un problema di **condizione di competizione** che si verificherebbe da tale implementazione.

USCITA A	USCITA B
inizio ciao fine del mondo	inizio mondo ciao fine

OpenMP: sezioni parallele

Questo esempio mostra come eseguire blocchi di codice in parallelo

```
std::cout << "begin ";
//    Start of parallel sections
#pragma omp parallel sections
{
    //    Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
//    end of parallel sections
std::cout << "end";
```

Produzione

- inizio ciao mondo per sempre fine

- inizio mondo ciao alla fine per sempre
- inizio ciao per sempre alla fine del mondo
- inizia per sempre ciao fine del mondo

Poiché l'ordine di esecuzione non è garantito, è possibile osservare uno qualsiasi dei risultati sopra riportati.

OpenMP: Parallel For Loop

Questo esempio mostra come dividere un loop in parti uguali ed eseguirli in parallelo.

```
//    Splits element vector into element.size() / Thread Qty
//    and allocate that range for each thread.
#pragma omp parallel for
for    (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

//    Example Allocation (100 element per thread)
//    Thread 1 : 0 ~ 99
//    Thread 2 : 100 ~ 199
//    Thread 2 : 200 ~ 299
//    ...

//    Continue process
//    Only when all threads completed their allocated
//    loop job
...
```

* Prestare particolare attenzione a non modificare le dimensioni del vettore utilizzato in parallelo per i cicli in quanto gli **indici di intervallo assegnati non si aggiornano automaticamente** .

OpenMP: Parallel Gathering / Reduction

Questo esempio illustra un concetto per eseguire la riduzione o la raccolta utilizzando `std::vector` e OpenMP.

Supposto che abbiamo uno scenario in cui vogliamo più thread per aiutarci a generare un sacco di cose, `int` è qui usato per semplicità e può essere sostituito con altri tipi di dati.

Ciò è particolarmente utile quando è necessario unire i risultati degli slave per evitare errori di segmentazione o violazioni di accesso alla memoria e non si desidera utilizzare librerie o librerie contenitore di sincronizzazione personalizzate.

```
//    The Master vector
//    We want a vector of results gathered from slave threads
std::vector<int> Master;

//    Hint the compiler to parallelize this { } of code
//    with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
    //    In this area, you can write any code you want for each
    //    slave thread, in this case a vector to hold each of their results
```

```
    //    We don't have to worry about how many threads were spawn or if we need
    //    to repeat this declaration or not.
    std::vector<int> Slave;

    //    Tell the compiler to use all threads allocated for this parallel region
    //    to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    //    The nowait keyword tells the compiler that the slave threads don't
    //    have to wait for all other slaves to finish this for loop job
    #pragma omp for nowait
    for (size_t i = 0; i < 1000000; ++i
    {
        /* Do something */
        ....
        Slave.push_back(...);
    }

    //    Slaves that finished their part of the job
    //    will perform this thread by thread one at a time
    //    critical section ensures that only 0 or 1 thread performs
    //    the { } at any time
    #pragma omp critical
    {
        //    Merge slave into master
        //    use move iterators instead, avoid copy unless
        //    you want to use it for something else after this section
        Master.insert(Master.end(),
                      std::make_move_iterator(Slave.begin()),
                      std::make_move_iterator(Slave.end()));
    }
}

//    Have fun with Master vector
...
```

Leggi Concorrenza con OpenMP online: https://riptutorial.com/it/cplusplus/topic/8222/concorrenza-con-openmp

Capitolo 19: Confronti affiancati di classici esempi C ++ risolti tramite C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17

Examples

Looping attraverso un contenitore

In C ++, è possibile eseguire il looping di un contenitore di sequenza `c` utilizzando gli indici nel modo seguente:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

Sebbene semplici, tali scritti sono soggetti a errori semantici comuni, come un operatore di confronto errato o una variabile di indicizzazione errata:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
                     ^~~~~~~~~~~~~~^
```

Il ciclo può essere realizzato anche per tutti i contenitori che utilizzano iteratori, con inconvenienti simili:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C ++ 11 ha introdotto la gamma per loop e `auto` parola chiave `auto` , permettendo al codice di diventare:

```
for(auto& x : c) x = 0;
```

Qui gli unici parametri sono il contenitore `c` e una variabile `x` per contenere il valore corrente. Ciò impedisce gli errori di semantica precedentemente indicati.

Secondo lo standard C ++ 11, l'implementazione sottostante è equivalente a:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

In tale implementazione, l'espressione `auto begin = c.begin(), end = c.end();` le forze `begin` e `end` per essere dello stesso tipo, mentre la `end` non viene mai incrementata, né dereferenziata. Quindi il ciclo basato su intervallo funziona solo per contenitori definiti da un iteratore / iteratore di coppie. Lo standard C ++ 17 allenta questo vincolo modificando l'implementazione in:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

Qui `begin` e `end` possono essere di tipi diversi, purché possano essere confrontati per disuguaglianza. Ciò consente di scorrere più contenitori, ad esempio un contenitore definito da un iteratore / sentinella pair.

Leggi Confronti affiancati di classici esempi C ++ risolti tramite C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 online: https://riptutorial.com/it/cplusplus/topic/7134/confronti-affiancati-di-classici-esempi-c-plusplus-risolti-tramite-c-plusplus-vs-c-plusplus-11-vs-c-plusplus-14-vs-c-plusplus-17

Capitolo 20: Const Correctness

Sintassi

- class ClassOne {public: bool non_modifying_member_function () const {/ * ... * /}};
- int ClassTwo :: non_modifying_member_function () const {/ * ... * /}
- void ClassTwo :: modifying_member_function () {/ * ... * /}
- char non_param_modding_func (const ClassOne e uno, const ClassTwo * two) {/ * ... * /}
- float parameter_modifying_function (ClassTwo e uno, ClassOne * two) {/ * ... * /}
- breve ClassThree :: non_modding_non_param_modding_f (const ClassOne &) const {/ * ... * /}

Osservazioni

`const` correctness è uno strumento di risoluzione dei problemi molto utile, in quanto consente al programmatore di determinare rapidamente quali funzioni potrebbero inavvertitamente modificare il codice. Evita anche errori non intenzionali, come quello mostrato in `Const Correct Function Parameters` , dalla compilazione corretta e passando inosservato.

È molto più semplice progettare una classe per la correttezza `const` , piuttosto che aggiungere successivamente la correttezza `const` a una classe preesistente. Se possibile, nel Designer classe che *può* essere `const` corretta in modo che *sia* `const` corretta, per salvare se stessi e gli altri il fastidio di poi modificarlo.

Si noti che questo può anche essere applicato alla correttezza `volatile` se necessario, con le stesse regole della correttezza `const` , ma questo viene usato molto meno spesso.

Refrences:

ISO_CPP

Vendimi per correttezza

Tutorial C ++

Examples

Le basi

`const` correttezza `const` è la pratica della progettazione del codice in modo che solo il codice che ha *bisogno* di modificare un'istanza sia in *grado* di modificare un'istanza (cioè ha accesso in scrittura) e, al contrario, qualsiasi codice che non ha bisogno di modificare un'istanza non è in grado di farlo così (cioè ha solo accesso in lettura). Ciò impedisce che l'istanza venga modificata involontariamente, rendendo il codice meno errorprone e documenta se il codice è destinato a modificare lo stato dell'istanza oppure no. Consente inoltre di trattare le istanze come `const` ogni

volta che non è necessario modificarle o definite come `const` se non devono essere modificate dopo l'inizializzazione, senza perdere alcuna funzionalità.

Questo viene fatto dando funzioni membro `const` CV-qualificazioni , e rendendo parametri puntatore / riferimento `const` , salvo il caso che hanno bisogno scrittura.

```
class ConstCorrectClass {
    int x;

  public:
    int getX() const { return x; } // Function is const: Doesn't modify instance.
    void setX(int i) { x = i; }    // Not const: Modifies instance.
};

// Parameter is const: Doesn't modify parameter.
int const_correct_reader(const ConstCorrectClass& c) {
    return c.getX();
}

// Parameter isn't const: Modifies parameter.
void const_correct_writer(ConstCorrectClass& c) {
    c.setX(42);
}

const ConstCorrectClass invariant; // Instance is const: Can't be modified.
ConstCorrectClass         variant; // Instance isn't const: Can be modified.

// ...

const_correct_reader(invariant); // Good.   Calling non-modifying function on const instance.
const_correct_reader(variant);   // Good.   Calling non-modifying function on modifiable
instance.

const_correct_writer(variant);   // Good.   Calling modifying function on modifiable instance.
const_correct_writer(invariant); // Error.  Calling modifying function on const instance.
```

A causa della natura della correttezza const, questo inizia con le funzioni membro della classe e si avvia verso l'esterno; se provate a chiamare una funzione membro non `const` da un'istanza `const` , o da un'istanza non `const` viene trattata come `const` , il compilatore vi darà un errore riguardo alla perdita dei qualificatori cv.

Const Correct Class Design

In una classe `const` -correct, tutte le funzioni membro che non cambiano lo stato logico hanno `this` cv-qualificato come `const` , che indica che non modificano l'oggetto (a parte i campi `mutable` , che possono essere modificati liberamente anche nelle istanze `const`); se una funzione `const` cv-qualificata restituisce un riferimento, tale riferimento dovrebbe essere anche `const` . Ciò consente loro di essere richiamati su istanze sia costanti che non classificate come cv, in quanto una `const T*` è in grado di legarsi a un `T*` oa un `const T*` . Ciò, a sua volta, consente alle funzioni di dichiarare i propri parametri passati per riferimento come `const` quando non devono essere modificati, senza perdere alcuna funzionalità.

Inoltre, in una classe `const` corretta, tutti i parametri della funzione `const` -by-reference saranno `const` corretti, come discusso in `Const Correct Function Parameters` , in modo che possano essere

modificati solo quando la funzione ha *bisogno* di modificarli esplicitamente.

Per prima cosa, diamo un'occhiata a `this` cv-qualificatore:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField();        // Might modify.  Also exposes member as non-const reference,
                              //  allowing indirect modification.
    void setField(Field& f);  // Modifies.

    void doSomething(int i);  // Might modify.
    void doNothing();         // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; }    // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {            // Modifies.
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {}                  // Doesn't modify.


class ConstCorrectCVQ {
    Field fld;

  public:
    ConstCorrectCVQ(Field& f);     // Modifies.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(Field& f);       // Modifies.

    void doSomething(int i);       // Modifies.
    void doNothing() const;        // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const  {}

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
// getField() and doNothing() can be called on const ConstCorrectCVQ instances.
```

```
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}
```

Possiamo quindi combinare questo con i `Const Correct Function Parameters` , facendo sì che la classe sia completamente `const` -correct.

```
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f);  // Modifies instance.  Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(const Field& f); // Modifies instance.  Doesn't modify parameter.

    void doSomething(int i);       // Modifies.  Doesn't modify parameter (passed by value).
    void doNothing() const;        // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

Questo può anche essere combinato con sovraccarichi sulla base di `const` ness, nel caso in cui vogliamo un comportamento se l'istanza è `const` , e un comportamento diverso se non lo è; un uso comune per questo è costituito da constainer che forniscono accessor che consentono solo modifiche se il contenitore stesso non è `const` .

```
class ConstCorrectContainer {
    int arr[5];

  public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
          int& operator[](size_t index)       { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

Questo è comunemente usato nella libreria standard, con la maggior parte dei contenitori che fornisce sovraccarichi di prendere `const` Ness in considerazione.

Const Correggi i parametri delle funzioni

In una funzione `const` -correct, tutti i parametri passati per riferimento sono contrassegnati come `const` meno che la funzione li modifichi direttamente o indirettamente, impedendo al

programmatore di modificare inavvertitamente qualcosa che non intendevano cambiare. Questo permette la funzione di prendere sia `const` e istanze non-cv qualificato, e, a sua volta, provoca l'istanza di `this` sia di tipo `const T*` quando una funzione membro viene chiamato, dove `T` è il tipo di classe.

```
struct Example {
    void func()       { std::cout << 3 << std::endl; }
    void func() const { std::cout << 5 << std::endl; }
};

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5
```

Mentre gli effetti di questo sono meno immediatamente visibili di quelli di `const` design di classe corretto (in quel `const` funzioni -correct e `const` classi -incorrect causerà errori di compilazione, mentre `const` classi -correct e `const` funzioni -incorrect potranno compilare correttamente), `const` corretta le funzioni colgono un sacco di errori che le funzioni `const` errate lascerebbero scivolare attraverso, come quella qui sotto. [Si noti, tuttavia, che un `const` funzione -incorrect *causerà* errori di compilazione, se superato un `const` esempio quando si aspettava un non- `const` uno.]

```
// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};

    int v_ind = h.get_index();               // Current working index for v.
    int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];

    temp -= h.poll_device();
```

```
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Oops.  Should've been accessing vals.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

// Const correct version.  Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}
```

Const Correctness come documentazione

Una delle cose più utili della correttezza `const` è che serve come un modo di documentare il codice, fornendo certe garanzie al programmatore e agli altri utenti. Tali garanzie vengono applicate dal compilatore causa `const` ness, con una mancanza di `const` Ness a sua volta indicando che il codice non fornisce loro.

Funzioni membro qualificate CV `const` :

- Si può presumere che ogni funzione membro che è `const` abbia intenzione di leggere l'istanza e:
    - Non modifica lo stato logico dell'istanza su cui sono chiamati. Pertanto, non devono modificare alcuna variabile membro dell'istanza su cui sono chiamati, tranne le variabili `mutable` .
    - Non chiama alcuna *altra* funzione che possa modificare qualsiasi variabile membro dell'istanza, tranne le variabili `mutable` .
- Viceversa, si può presumere che qualsiasi funzione membro che non è `const` abbia intenzione di modificare l'istanza e:
    - Può o non può modificare lo stato logico.
    - Può o non può chiamare altre funzioni che modificano lo stato logico.

Questo può essere usato per fare assunzioni sullo stato dell'oggetto dopo che ogni funzione membro è stata chiamata, anche senza vedere la definizione di quella funzione:

```
// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

  public:
    // Constructor clearly changes logical state.  No assumptions necessary.
    ConstMemberFunctions(int v = 0);

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call squared_calc() or bad_func().
    int calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or bad_func().
    int squared_calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or squared_calc().
    void bad_func() const;

    // We can assume this function changes logical state, and may or may not call
    //  calc(), squared_calc(), or bad_func().
    void set_val(int v);
};
```

A causa delle regole `const` , queste ipotesi saranno effettivamente applicate dal compilatore.

```
// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0*/)
  : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}
```

```
// Our assumption was correct.
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}
```

Parametri funzione `const` :

- Qualsiasi funzione con uno o più parametri che sono `const` può avere l'intento di leggere quei parametri, e:
    - Non modifica questi parametri o chiama le funzioni membro che potrebbero modificarli.
    - Non passa quei parametri a nessuna *altra* funzione che li modifichi e / o chiami qualsiasi funzione membro che li modifichi.
- Viceversa, si può presumere che qualsiasi funzione con uno o più parametri che non sono `const` abbia intenzione di modificare tali parametri, e:
    - Può o non può modificare quei parametri, o chiamare qualsiasi funzione membro che li possa modificare.
    - Può o non può passare quei parametri ad altre funzioni che potrebbero modificarli e / o chiamare qualsiasi funzione membro che li modifichi.

Questo può essere usato per fare assunzioni sullo stato dei parametri dopo essere passati a una data funzione, anche senza vedere la definizione di quella funzione.

```
// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
//  to any of these functions.  If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

// We can assume that:
  // l is not modified, and l.set_val() won't be called.
  // l may or may not be passed to const_function_parameter().
  // r is modified, and/or r.set_val() may be called.
  // r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void bad_parameter(const ConstMemberFunctions& c);
```

A causa delle regole `const` , queste ipotesi saranno effettivamente applicate dal compilatore.

```
// function_parameter.cpp

// Our assumption was correct.
```

```
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
//  it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers in c.set_val().
void bad_parameter(const ConstMemberFunctions& c) {
    c.set_val(18);
}
```

Mentre è possibile aggirare la correttezza `const` , e per estensione rompere queste garanzie, questo deve essere fatto intenzionalmente dal programmatore (proprio come interrompere l'incapsulamento con `Machiavelli` , sopra), ed è probabile che causi un comportamento indefinito.

```
class DealBreaker : public ConstMemberFunctions {
  public:
    DealBreaker(int v = 0);

    // A foreboding name, but it's const...
    void no_guarantees() const;
}

DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}

// Our assumption was incorrect.
// const_cast removes const-ness, making the compiler think we know what we're doing.
void DealBreaker::no_guarantees() const {
    const_cast<DealBreaker*>(this)->set_val(823);
}

// ...
```

```
const DealBreaker d(50);
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

Tuttavia, a causa di questo che richiede al programmatore di *dire* in modo molto specifico al compilatore che intendono ignorare `const` ness, e di essere incoerente nei compilatori, è generalmente lecito ritenere che `const` codice corretto si asterrà dal farlo se non diversamente specificato.

Leggi Const Correctness online: https://riptutorial.com/it/cplusplus/topic/7217/const-correctness

Capitolo 21: constexpr

introduzione

`constexpr` è una parola chiave che può essere usata per marcare il valore di una variabile come espressione costante, una funzione come potenzialmente utilizzabile nelle espressioni costanti, o (dal C ++ 17) un'istruzione if se ha solo uno dei suoi rami selezionati per essere compilato.

Osservazioni

La parola chiave `constexpr` stata aggiunta in C ++ 11 ma per alcuni anni da quando è stato pubblicato lo standard C ++ 11, non tutti i principali compilatori lo hanno supportato. nel momento in cui è stato pubblicato lo standard C ++ 11. Al momento della pubblicazione di C ++ 14, tutti i principali compilatori supportano `constexpr` .

Examples

variabili di constexpr

Una variabile dichiarata `constexpr` è implicitamente `const` e il suo valore può essere usato come espressione costante.

Confronto con `#define`

Un `constexpr` è un sostituto sicuro per le espressioni di compilazione basate su `#define` . Con `constexpr` l'espressione valutata in fase di compilazione viene sostituita con il risultato. Per esempio:

C ++ 11

```
int main()
{
   constexpr int N = 10 + 2;
   cout << N;
}
```

produrrà il seguente codice:

```
cout << 12;
```

Una macro in fase di compilazione basata sul pre-processore sarebbe diversa. Tenere conto:

```
#define N 10 + 2

int main()
{
    cout << N;
```

```
}
```

produrrà:

```
cout << 10 + 2;
```

che sarà ovviamente convertito in `cout << 10 + 2;` . Tuttavia, il compilatore dovrebbe fare più lavoro. Inoltre, crea un problema se non utilizzato correttamente.

Ad esempio (con `#define`):

```
cout << N * 2;
```

le forme:

```
cout << 10 + 2 * 2; // 14
```

Ma un `constexpr` pre-valutato `constexpr` correttamente `24` .

Confronto con `const`

Una variabile `const` è una **variabile** che ha bisogno di memoria per la sua memorizzazione. Un `constexpr` non lo fa. Un `constexpr` produce una costante di tempo di compilazione, che non può essere modificata. Si potrebbe obiettare che anche `const` può essere cambiato. Ma considera:

```
int main()
{
   const int size1 = 10;
   const int size2 = abs(10);

   int arr_one[size1];
   int arr_two[size2];
}
```

Con la maggior parte dei compilatori la seconda istruzione fallirà (potrebbe funzionare con GCC, ad esempio). La dimensione di qualsiasi array, come forse saprai, deve essere un'espressione costante (cioè produce un valore in fase di compilazione). Alla seconda variabile `size2` viene assegnato un valore deciso in fase di esecuzione (anche se si sa che è `10` , per il compilatore non è in fase di compilazione).

Ciò significa che una `const` può o non può essere una vera costante in fase di compilazione. Non puoi garantire o far rispettare che un particolare valore `const` sia assolutamente in fase di compilazione. Puoi usare `#define` ma ha le sue insidie.

Quindi usa semplicemente:

C ++ 11

```
int main()
{
```

```
    constexpr int size = 10;

    int arr[size];
}
```

`constexpr` deve valutare un valore in fase di compilazione. Pertanto, non è possibile utilizzare:

C ++ 11

```
constexpr int size = abs(10);
```

A meno che la funzione (`abs`) non stia restituendo un `constexpr` .

Tutti i tipi di base possono essere inizializzati con `constexpr` .

C ++ 11

```
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

È interessante notare che, e in modo conveniente, è possibile utilizzare anche l' `auto` :

C ++ 11

```
constexpr auto domain = ".COM";  // const char * const domain = ".COM"
constexpr auto PI = 3.14;        // constexpr double
```

funzioni di constexpr

Una funzione dichiarata `constexpr` è implicitamente in linea e le chiamate a tale funzione potenzialmente producono espressioni costanti. Ad esempio, la funzione seguente, se chiamata con argomenti con espressioni costanti, produce anche un'espressione costante:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Pertanto, il risultato della chiamata di funzione può essere utilizzato come un limite di matrice o un argomento di modello o per inizializzare una variabile di `constexpr` :

C ++ 11

```
int main()
{
    constexpr int S = Sum(10,20);

    int Array[S];
    int Array2[Sum(20,30)]; // 50 array size, compile time
```

```
}
```

Si noti che se si rimuove `constexpr` dalla specifica del tipo restituito dalla funzione, l'assegnazione a `S` non funzionerà, poiché `S` è una variabile `constexpr` e deve essere assegnata a const di compilazione. Allo stesso modo, la dimensione dell'array non sarà anche un'espressione costante, se la funzione `Sum` non è `constexpr` .

La cosa interessante delle funzioni di `constexpr` è che puoi anche usarlo come funzioni ordinarie:

C ++ 11

```
int a = 20;
auto sum = Sum(a, abs(-20));
```

`Sum` non sarà una funzione di `constexpr` ora, sarà compilata come una funzione ordinaria, assumendo argomenti variabili (non costanti) e restituendo un valore non costante. Non è necessario scrivere due funzioni.

Significa anche che se si tenta di assegnare tale chiamata a una variabile non const, non verrà compilata:

C ++ 11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

La ragione è semplice: a `constexpr` deve essere assegnata una costante in tempo di compilazione. Tuttavia, la suddetta chiamata di funzione rende `Sum` un non- `constexpr` (il valore R è non-const, ma il valore-L si dichiara essere `constexpr`).

La funzione `constexpr` **deve** anche restituire una costante in fase di compilazione. Di seguito non verrà compilato:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;     // ERROR
    return a + b;
}
```

Perché `a1` è una *variabile* non-constexpr e impedisce alla funzione di essere una vera funzione `constexpr` . Rendere più `constexpr` e assegnandogli `a` anche non funziona - dal momento che il valore di `a` (parametro in arrivo) non è ancora ancora noto:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   constexpr int a1 = a;     // ERROR
```

```
   ..
```

Inoltre, di seguito non verrà compilato anche:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   return abs(a) + b; // or abs(a) + abs(b)
}
```

Poiché `abs(a)` non è un'espressione costante (anche `abs(10)` non funzionerà, dato che `abs` non sta restituendo un `constexpr int` !

Che dire di questo?

C ++ 11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

Abbiamo creato la nostra funzione `Abs` che è un `constexpr` , e anche il corpo di `Abs` non infrange nessuna regola. Inoltre, nel sito di chiamata (all'interno di `Sum`), l'espressione `constexpr` un `constexpr` . Quindi, la chiamata a `Sum(-10, 20)` sarà un'espressione costante in fase di compilazione risultante a `30` .

Statico se dichiarazione

C ++ 17

L'istruzione `if constexpr` può essere utilizzata per compilare il codice in modo condizionale. La condizione deve essere un'espressione costante. Il ramo non selezionato viene *scartato.* Un'istruzione scartata all'interno di un modello non viene istanziata. Per esempio:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
  // ... handle p
  if constexpr (sizeof...(rs) > 0)
    g(rs...);  // never instantiated with an empty argument list
}
```

Dichiarazioni Inoltre, variabili e funzioni che vengono ODR-utilizzati solo all'interno scartati non devono essere definiti, e scartati `return` dichiarazioni non sono utilizzati per tipo di funzione di ritorno deduzione.

`if constexpr` è diverso da `#ifdef` . `#ifdef` compila in modo condizionale il codice, ma solo in base a condizioni che possono essere valutate in fase di preelaborazione. Ad esempio, `#ifdef` non può essere utilizzato per compilare in modo condizionale il codice in base al valore di un parametro del modello. D'altra parte, `if constexpr` non può essere usato per scartare il codice sintatticamente non valido, mentre `#ifdef` può `#ifdef` .

```
if constexpr(false) {
    foobar;  // error; foobar has not been declared
    std::vector<int> v("hello, world");  // error; no matching constructor
}
```

Leggi constexpr online: https://riptutorial.com/it/cplusplus/topic/3899/constexpr

Capitolo 22: Contenitori C ++

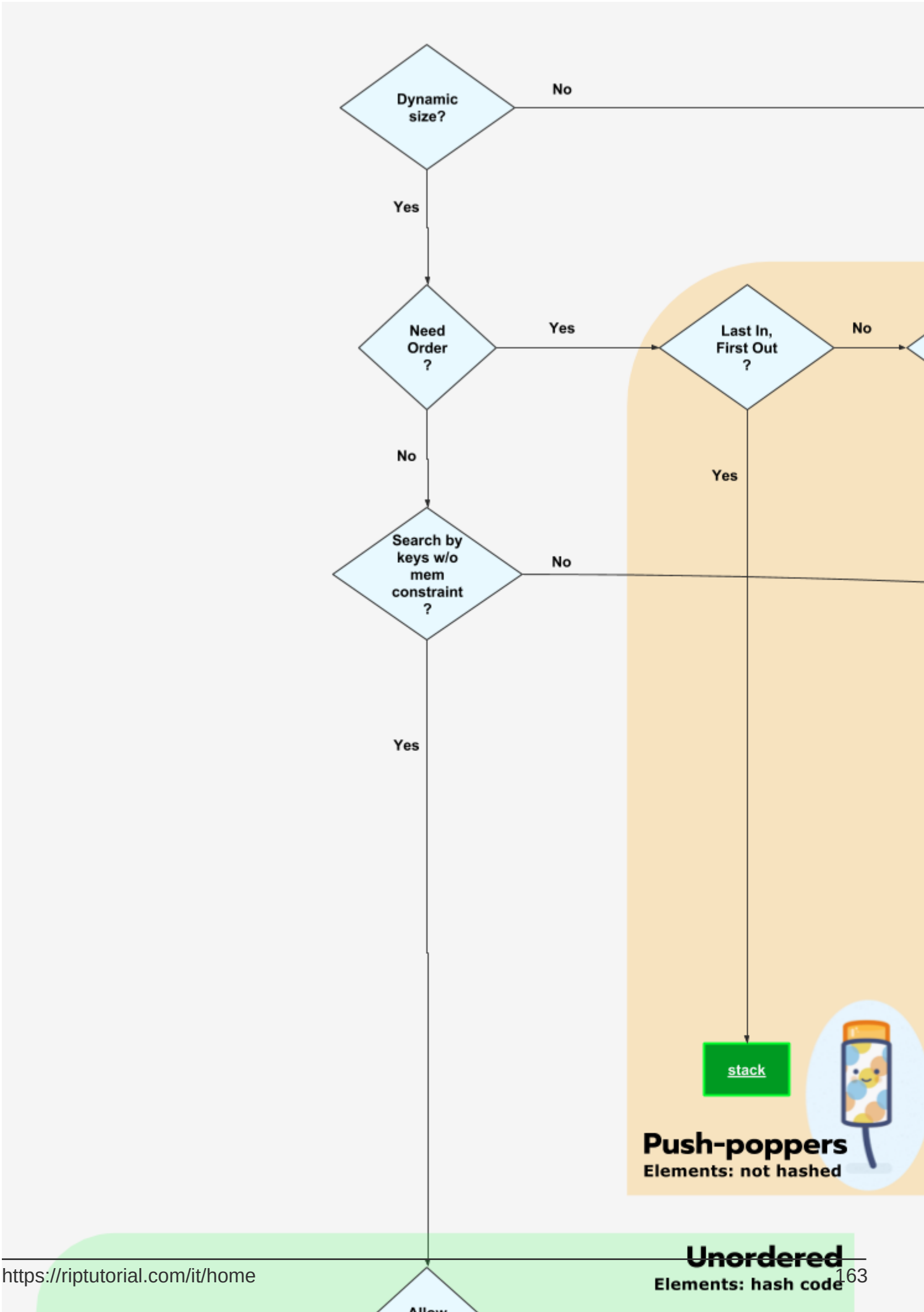
introduzione

I contenitori C ++ memorizzano una raccolta di elementi. I contenitori includono vettori, elenchi, mappe, ecc. Utilizzando i modelli, i contenitori C ++ contengono raccolte di primitive (ad esempio int) o classi personalizzate (ad es. MyClass).

Examples

Diagramma di flusso dei contenitori C ++

Scegliere quale Container C ++ usare può essere complicato, quindi ecco un semplice diagramma di flusso per aiutare a decidere quale Container è adatto per il lavoro.


Dynamic size?
No
Yes
Need Order ?
Yes
Last In, First Out ?
No
Yes
No
Search by keys w/o mem constraint ?
No
Yes
stack
Push-poppers
Elements: not hashed
Unordered
Elements: hash code

. Questo piccolo grafico nel diagramma di flusso è di Megan Hopkins

Leggi Contenitori C ++ online: https://riptutorial.com/it/cplusplus/topic/10848/contenitori-c-plusplus

Capitolo 23: Controllo del flusso

Osservazioni

Controlla l' argomento dei loop per i diversi tipi di loop.

Examples

Astuccio

Introduce un'etichetta case di un'istruzione switch. L'operando deve essere un'espressione costante e deve corrispondere alla condizione dell'interruttore nel tipo. Quando viene eseguita l'istruzione switch, passa all'etichetta case con operando uguale alla condizione, se presente.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

interruttore

Secondo lo standard C ++,

> L'istruzione `switch` fa sì che il controllo venga trasferito a una delle diverse istruzioni a seconda del valore di una condizione.

Lo `switch` parola chiave è seguito da una condizione parentesi e un blocco, che può contenere etichette `case` e un'etichetta `default` facoltativa. Quando viene eseguita l'istruzione switch, il controllo verrà trasferito a un'etichetta `case` con un valore corrispondente a quello della condizione, se presente, o all'etichetta `default` , se presente.

La condizione deve essere un'espressione o una dichiarazione, che ha un numero intero o un tipo di enumerazione o un tipo di classe con una funzione di conversione in un numero intero o un tipo di enumerazione.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
```

```
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

catturare

La parola chiave `catch` introduce un gestore di eccezioni, ovvero un blocco in cui il controllo verrà trasferito quando viene generata un'eccezione di tipo compatibile. La parola chiave `catch` è seguita da una *dichiarazione di eccezione* parentesi, che è simile nella forma a una dichiarazione di parametro di funzione: il nome del parametro può essere omesso e l'ellissi `...` è consentita, che corrisponde a qualsiasi tipo. Il gestore delle eccezioni gestirà l'eccezione solo se la sua dichiarazione è compatibile con il tipo dell'eccezione. Per ulteriori dettagli, vedere la rilevazione di eccezioni .

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

predefinito

In un'istruzione switch, introduce un'etichetta a cui verrà eseguito il salto se il valore della condizione non è uguale a uno dei valori delle etichette del caso.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

C ++ 11

Definisce un costruttore predefinito, un costruttore di copia, un costruttore di movimento, un

distruttore, un operatore di assegnazione copia o un operatore di spostamento di spostamento per avere il suo comportamento predefinito.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

Se

Presenta una dichiarazione if. La parola chiave `if` deve essere seguita da una condizione parentesi, che può essere un'espressione o una dichiarazione. Se la condizione è veritiera, la sottostazione successiva alla condizione verrà eseguita.

```
int x;
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

altro

La prima sottostazione di un'istruzione if può essere seguita dalla parola chiave `else` . La sottostazione dopo la parola chiave `else` verrà eseguita quando la condizione è falsa (ovvero, quando la prima sottostazione non viene eseguita).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

vai a

Salta a un'istruzione etichettata, che deve trovarsi nella funzione corrente.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
    result = true;
  end:
```

```
    release_widget(widget);
    return result;
}
```

ritorno

Restituisce il controllo da una funzione al relativo chiamante.

Se `return` ha un operando, l'operando viene convertito nel tipo di ritorno della funzione e il valore convertito viene restituito al chiamante.

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

Se `return` non ha un operando, la funzione deve avere un tipo di reso `void` . Come caso speciale, una funzione di annullamento del `void` può anche restituire un'espressione se l'espressione ha tipo `void` .

```
void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}
```

Quando restituisce `main` , `std::exit` viene chiamato implicitamente con il valore restituito e il valore viene quindi restituito all'ambiente di esecuzione. (Comunque, il ritorno da `main` distrugge le variabili locali automatiche, mentre chiamare `std::exit` direttamente no.)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

gettare

1. Quando il `throw` verifica in un'espressione con un operando, il suo effetto è di generare un'eccezione , che è una copia dell'operando.

   ```
   void print_asterisks(int count) {
       if (count < 0) {
   ```

```
        throw std::invalid_argument("count cannot be negative!");
    }
    while (count--) { putchar('*'); }
}
```

2. Quando il `throw` verifica in un'espressione senza un operando, il suo effetto consiste nel ripensare all'eccezione corrente . Se non ci sono eccezioni attuali, viene chiamato `std::terminate` .

```
try {
    // something risky
} catch (const std::bad_alloc&) {
    std::cerr << "out of memory" << std::endl;
} catch (...) {
    std::cerr << "unexpected exception" << std::endl;
    // hope the caller knows how to handle this exception
    throw;
}
```

3. Quando il `throw` verifica in un dichiaratore di funzioni, introduce una specifica di eccezione dinamica, che elenca i tipi di eccezioni che la funzione è autorizzata a propagare.

```
// this function might propagate a std::runtime_error,
// but not, say, a std::logic_error
void risky() throw(std::runtime_error);
// this function can't propagate any exceptions
void safe() throw();
```

   Le specifiche delle eccezioni dinamiche sono deprecate dal C ++ 11.

Si noti che i primi due usi del `throw` sopra elencati costituiscono espressioni piuttosto che dichiarazioni. (Il tipo di un'espressione di lancio è `void` .) Ciò rende possibile annidarli all'interno di espressioni, in questo modo:

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

provare

La parola chiave `try` è seguita da un blocco o da un elenco di inizializzatore del costruttore e quindi da un blocco (vedere qui). Il blocco try è seguito da uno o più blocchi catch . Se un'eccezione si propaga dal blocco try, ognuno dei blocchi catch corrispondenti dopo il blocco try ha l'opportunità di gestire l'eccezione, se i tipi corrispondono.

```
std::vector<int> v(N);      // if an exception is thrown here,
                            // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
                           // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
```

```
    // handle bad_alloc exceptions from the try block
}
```

Strutture condizionali: if, if..else

se e altro:

usato per verificare se l'espressione data restituisce vero o falso e agisce in quanto tale:

```
if (condition) statement
```

la condizione può essere qualsiasi espressione C ++ valida che restituisca qualcosa da verificare con verità / falsità, ad esempio:

```
if (true) { /* code here */ }  // evaluate that true is true and execute the code in the
brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

la condizione può essere qualsiasi cosa, una funzione, una variabile o un confronto, ad esempio

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evalute the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any
non zero value will be true,
```

se vogliamo verificare una molteplicità di espressioni, possiamo farlo in due modi:

usando operatori binari :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside
the scope here
if (a || b ) { } //true if a or b is true
```

usando if / ifelse / else :

per un semplice interruttore o se altrimenti

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

per più scelte:

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
```

```
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
}
```

tuttavia, è necessario notare che è necessario utilizzare ' **switch** ' se il codice verifica il valore della stessa variabile

Istruzioni di salto: pausa, continua, goto, uscita.

L'istruzione di rottura:

Usando l'interruzione possiamo lasciare un ciclo anche se la condizione per la sua fine non è soddisfatta. Può essere usato per terminare un ciclo infinito o per costringerlo a terminare prima della sua fine naturale

La sintassi è

```
break;
```

Esempio : spesso utilizziamo i casi di `break` `switch` , vale a dire una volta che un caso in cui lo switch è soddisfatto, viene eseguito il blocco di codice di tale condizione.

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

in questo caso se il caso 1 è soddisfatto allora viene eseguito il blocco 1, quello che vogliamo veramente è solo il blocco1 da elaborare, ma invece, una volta processato il blocco1, i blocchi rimanenti, block2, block3 e blockdefault vengono elaborati anche se solo il caso 1 è stato saturato Per evitare questo, utilizziamo l'interruzione alla fine di ogni blocco come:

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

quindi viene elaborato un solo blocco e il controllo esce dal ciclo di commutazione.

break può anche essere usato in altri loop condizionali e non condizionali come `if` , `while` , `for` etc;

esempio:

```
if(condition1){
   ....
   if(condition2){
    .......
    break;
    }
 ...
}
```

L'istruzione continua:

L'istruzione continue fa in modo che il programma salti il resto del ciclo nell'iterazione corrente come se la fine del blocco di istruzioni fosse stata raggiunta, facendolo saltare alla seguente iterazione.

La sintassi è

```
continue;
```

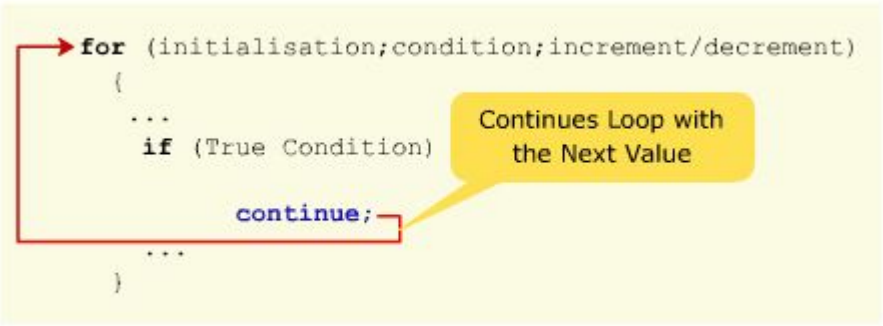
Esempio considerare quanto segue:

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

che produce l'output:

```
 @1
 @3
 @5
 @7
 @9
```

questo codice ogni volta che la condizione `i%2==0` è soddisfatta, `continue` viene elaborata, questo fa sì che il compilatore salti tutto il codice rimanente (stampa @ e i) e viene eseguita l'istruzione di incremento / decremento del ciclo.



L'istruzione goto:

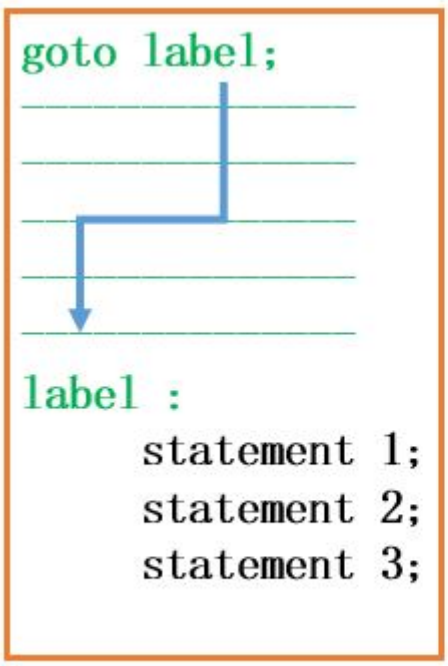
Permette di fare un salto in assoluto verso un altro punto del programma. È necessario utilizzare

questa funzione attentamente poiché la sua esecuzione ignora qualsiasi tipo di limitazione di nidificazione. Il punto di destinazione è identificato da un'etichetta, che viene quindi utilizzata come argomento per l'istruzione goto. Un'etichetta è composta da un identificatore valido seguito da due punti (:)

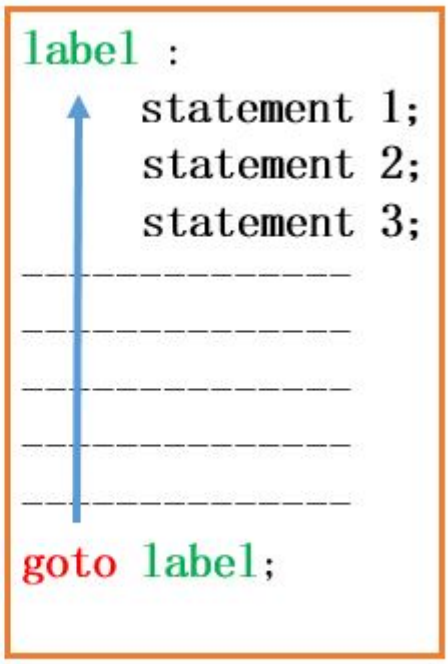
La sintassi è

```
goto label;
..
.
label: statement;
```

Nota: l' *uso dell'istruzione goto è altamente sconsigliato perché rende difficile tracciare il flusso di controllo di un programma, rendendo il programma difficile da capire e difficile da modificare.*



Forward Reference



Backward Reference

Esempio :

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
        goto STEP;
     }

   cout << "value of num : " << num << endl;
   num = num + 1;
 }while( num < 10 );
```

produzione :

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

ogni volta che la condizione `num%2==0` è soddisfatta, goto invia il controllo di esecuzione all'inizio del ciclo `do-while` .

La funzione di uscita:

`exit` è una funzione definita in `cstdlib` . Lo scopo `exit` è terminare il programma in esecuzione con un codice di uscita specifico. Il suo prototipo è:

```
void exit (int exit code);
```

`cstdlib` definisce i codici di uscita standard `EXIT_SUCCESS` e `EXIT_FAILURE` .

Leggi Controllo del flusso online: https://riptutorial.com/it/cplusplus/topic/7837/controllo-del-flusso

Capitolo 24: Conversioni di tipo esplicito

introduzione

Un'espressione può essere *convertita* o *cast esplicita* per digitare `T` usando `dynamic_cast<T>` , `static_cast<T>` , `reinterpret_cast<T>` , o `const_cast<T>` , a seconda del tipo di cast desiderato.

C ++ supporta anche la notazione cast in stile funzione, `T(expr)` e la notazione cast in stile C, `(T)expr` .

Sintassi

- *identificatore di tipo semplice* `(` `)`
- *identificatore di tipo semplice* `(` *elenco di espressioni* `)`
- *semplice-tipo-identificatore braced-init-list*
- *typename-specificifier* `(` `)`
- *typename-specificifier* `(` *expression-list* `)`
- *typename-specifier braced-init-list*
- `dynamic_cast` `<` *id-tipo* `>` `(` *espressione* `)`
- `static_cast` `<` *id-tipo* `>` `(` *espressione* `)`
- `reinterpret_cast` `<` *id-tipo* `>` `(` *espressione* `)`
- `const_cast` `<` *id-tipo* `>` `(` *espressione* `)`
- `(` *type-id* `)` *cast-expression*

Osservazioni

Tutte e sei le annotazioni sul cast hanno una cosa in comune:

- Il casting su un tipo di riferimento lvalue, come in `dynamic_cast<Derived&>(base)` , produce un lvalue. Pertanto, quando si vuole fare qualcosa con lo stesso oggetto ma trattarlo come un tipo diverso, si passa a un tipo di riferimento lvalue.
- La trasmissione a un tipo di riferimento di rvalue, come in `static_cast<string&&>(s)` , produce un valore rvalue.
- Il casting su un tipo non di riferimento, come in `(int)x` , produce un valore, che può essere considerato come una *copia* del valore da trasmettere, ma con un tipo diverso dall'originale.

La parola chiave `reinterpret_cast` è responsabile dell'esecuzione di due diversi tipi di conversioni "non sicure":

- Le conversioni "type punning" , che possono essere utilizzate per accedere alla memoria di un tipo come se fosse di un tipo diverso.
- Conversioni tra tipi interi e tipi di puntatore , in entrambe le direzioni.

La parola chiave `static_cast` può eseguire una varietà di conversioni diverse:

- Base per conversioni derivate
- Qualsiasi conversione che può essere eseguita tramite un'inizializzazione diretta, incluse conversioni e conversioni implicite che chiamano una funzione di costruzione o conversione esplicita. Vedi qui e qui per maggiori dettagli.
- Per `void` , che scarta il valore dell'espressione.

```
// on some compilers, suppresses warning about x being unused
static_cast<void>(x);
```

- Tra i tipi aritmetici e di enumerazione e tra i diversi tipi di enumerazione. Vedi conversioni enum
- Da puntatore a membro della classe derivata, puntatore al membro della classe base. I tipi a cui punta devono corrispondere. Vedi la conversione da derivata a base per i puntatori ai membri
- `void*` a `T*` .

C ++ 11

- Da un lvalue a un valore x, come in `std::move` . Vedi spostare la semantica .

Examples

Conversione da base a derivata

Un puntatore alla classe base può essere convertito in un puntatore alla classe derivata usando `static_cast` . `static_cast` non esegue alcun controllo di runtime e può portare a comportamenti non definiti quando il puntatore non punta effettivamente al tipo desiderato.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1;                          // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
                                           // point to a Derived object
```

Allo stesso modo, un riferimento alla classe base può essere convertito in un riferimento alla classe derivata usando `static_cast` .

```
struct Base {};
struct Derived : Base {};
Derived d;
Base& r1 = d;
Derived& r2 = r1;                          // error; cast required
```

```
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object
```

Se il tipo di origine è polimorfico, `dynamic_cast` può essere utilizzato per eseguire una conversione da base a derivata. Esegue un controllo in fase di esecuzione e l'errore è recuperabile invece di produrre un comportamento non definito. Nel caso del puntatore, un puntatore nullo viene restituito in caso di errore. Nel caso di riferimento, viene generata un'eccezione in caso di errore di tipo `std::bad_cast` (o una classe derivata da `std::bad_cast`).

```
struct Base { virtual ~Base(); }; // Base is polymorphic
struct Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer
```

Gettare via la costanza

Un puntatore a un oggetto const può essere convertito in un puntatore all'oggetto non-const usando la parola chiave `const_cast` . Qui usiamo `const_cast` per chiamare una funzione che non è const-correct. Accetta solo un argomento non-const `char*` anche se non scrive mai attraverso il puntatore:

```
void bad_strlen(char*);
const char* s = "hello, world!";
bad_strlen(s);                    // compile error
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` al tipo di riferimento può essere utilizzato per convertire un valore lovalificato const-value in un valore non const-qualificato.

`const_cast` è pericoloso perché rende impossibile al sistema di tipo C ++ impedirti di tentare di modificare un oggetto const. Ciò comporta un comportamento indefinito.

```
const int x = 123;
int& mutable_x = const_cast<int&>(x);
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Digitare la conversione punitiva

Un puntatore (riferimento) a un tipo di oggetto può essere convertito in un puntatore (riferimento) in qualsiasi altro tipo di oggetto utilizzando `reinterpret_cast` . Questo non chiama costruttori o funzioni di conversione.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                              // maybe this modifies x (see below)
```

C ++ 11

Il risultato di `reinterpret_cast` rappresenta lo stesso indirizzo dell'operando, a condizione che l'indirizzo sia allineato in modo appropriato per il tipo di destinazione. Altrimenti, il risultato non è specificato.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

C ++ 11

Il risultato di `reinterpret_cast` non è specificato, tranne che un puntatore (riferimento) sopravviverà a un round trip dal tipo di origine al tipo di destinazione e viceversa, purché il requisito di allineamento del tipo di destinazione non sia più rigido di quello del tipo di origine.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

Sulla maggior parte delle implementazioni, `reinterpret_cast` non cambia l'indirizzo, ma questo requisito non è stato standardizzato fino a C ++ 11.

`reinterpret_cast` può anche essere utilizzato per convertire da un tipo di membro puntatore a membro di dati a un altro o un tipo di funzione da puntatore a membro a un altro.

L'uso di `reinterpret_cast` è considerato pericoloso perché la lettura o la scrittura attraverso un puntatore o un riferimento ottenuto utilizzando `reinterpret_cast` può attivare un comportamento indefinito quando i tipi di origine e destinazione non sono correlati.

Conversione tra puntatore e intero

Un puntatore oggetto (incluso `void*`) o un puntatore funzione può essere convertito in un tipo intero usando `reinterpret_cast` . Questo verrà compilato solo se il tipo di destinazione è abbastanza lungo. Il risultato è definito dall'implementazione e tipicamente restituisce l'indirizzo numerico del byte in memoria a cui puntano i puntatori del puntatore.

In genere, `long` o `unsigned long` è abbastanza lungo da contenere qualsiasi valore del puntatore, ma questo non è garantito dallo standard.

C ++ 11

Se i tipi `std::intptr_t` e `std::uintptr_t` esistono, sono garantiti per essere abbastanza lunghi da contenere un `void*` (e quindi qualsiasi puntatore al tipo di oggetto). Tuttavia, non è garantito che siano sufficientemente lunghi da contenere un puntatore a funzione.

Allo stesso modo, `reinterpret_cast` può essere utilizzato per convertire un tipo intero in un tipo di puntatore. Anche in questo caso il risultato è definito dall'implementazione, ma è garantito che il valore di un puntatore non venga modificato da un round trip attraverso un tipo intero. Lo standard

non garantisce che il valore zero venga convertito in un puntatore nullo.

```
void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;
register_callback(my_callback,
                  reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Conversione tramite costruttore esplicito o funzione di conversione esplicita

Una conversione che implica il richiamo di una funzione di costruzione o conversione esplicita non può essere eseguita implicitamente. Possiamo richiedere che la conversione venga eseguita esplicitamente utilizzando `static_cast` . Il significato è lo stesso di quello di un'inizializzazione diretta, tranne per il fatto che il risultato è temporaneo.

```
class C {
    std::unique_ptr<int> p;
  public:
    explicit C(int* p) : p(p) {}
};
void f(C c);
void g(int* p) {
    f(p);                 // error: C::C(int*) is explicit
    f(static_cast<C>(p)); // ok
    f(C(p));              // equivalent to previous line
    C c(p); f(c);         // error: C is not copyable
}
```

Conversione implicita

`static_cast` può eseguire qualsiasi conversione implicita. Questo uso di `static_cast` può talvolta essere utile, come nei seguenti esempi:

- Quando si passano argomenti a puntini di sospensione, il tipo di argomento "previsto" non è noto staticamente, quindi non si verificherà alcuna conversione implicita.

  ```
  const double x = 3.14;
  printf("%d\n", static_cast<int>(x)); // prints 3
  // printf("%d\n", x); // undefined behaviour; printf is expecting an int here
  // alternative:
  // const int y = x; printf("%d\n", y);
  ```

  Senza la conversione di tipo esplicita, un `double` oggetto verrebbe passato ai puntini di sospensione e si verificherebbe un comportamento non definito.

- Un operatore di assegnazione classe derivato può chiamare un operatore di assegnazione classe base in questo modo:

  ```
  struct Base { /* ... */ };
  ```

```
struct Derived : Base {
    Derived& operator=(const Derived& other) {
        static_cast<Base&>(*this) = other;
        // alternative:
        // Base& this_base_ref = *this; this_base_ref = other;
    }
};
```

Conversioni Enum

`static_cast` può convertire da un numero intero o un tipo a virgola mobile a un tipo di enumerazione (con ambito o senza ambito) e *viceversa.* Può anche convertire tra tipi di enumerazione.

- La conversione da un tipo di enumerazione senza ambito a un tipo aritmetico è una conversione implicita; è possibile, ma non necessario, usare `static_cast` .

C ++ 11

- Quando un tipo di enumerazione dell'ambito viene convertito in un tipo aritmetico:
    - Se il valore dell'enumerazione può essere rappresentato esattamente nel tipo di destinazione, il risultato è quel valore.
    - Altrimenti, se il tipo di destinazione è un tipo intero, il risultato non è specificato.
    - Altrimenti, se il tipo di destinazione è un tipo a virgola mobile, il risultato è uguale a quello della conversione al tipo sottostante e quindi al tipo a virgola mobile.

  Esempio:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};
Format f = Format::PDF;
int a = f;                          // error
int b = static_cast<int>(f);        // ok; b is 1000
char c = static_cast<char>(f);      // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- Quando un numero intero o un tipo di enumerazione viene convertito in un tipo di enumerazione:
    - Se il valore originale è compreso nell'intervallo dell'enumerazione di destinazione, il risultato è tale valore. Si noti che questo valore potrebbe non essere uguale a tutti gli enumeratori.
    - In caso contrario, il risultato non è specificato (<= C ++ 14) o non definito (> = C ++ 17).

  Esempio:

```
enum Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1;                       // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

C ++ 11

- Quando un tipo a virgola mobile viene convertito in un tipo di enumerazione, il risultato è lo stesso della conversione al tipo sottostante dell'enumerazione e quindi al tipo enum.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

Derivato per basare la conversione per i puntatori ai membri

Un puntatore al membro della classe derivata può essere convertito in un puntatore al membro della classe base utilizzando `static_cast` . I tipi a cui punta devono corrispondere.

Se l'operando è un puntatore nullo al valore del membro, il risultato è anche un puntatore nullo al valore del membro.

Altrimenti, la conversione è valida solo se il membro a cui punta l'operando esiste effettivamente nella classe di destinazione, o se la classe di destinazione è una classe di base o derivata della classe che contiene il membro puntato dall'operando. `static_cast` non verifica la validità. Se la conversione non è valida, il comportamento non è definito.

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1;                                // ok; implicit conversion
int B::*p3 = p2;                                // error
int B::*p4 = static_cast<int B::*>(p2);       // ok; p4 is equal to p1
int A::*p5 = static_cast<int A::*>(p2);       // undefined; p2 points to x, which is a member
                                                // of the unrelated class B
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z
int A::*p8 = static_cast<int A::*>(p6);       // error: types don't match
```

void * a T *

In C ++, `void*` non può essere convertito implicitamente in `T*` dove `T` è un tipo di oggetto. Invece, `static_cast` dovrebbe essere usato per eseguire la conversione esplicitamente. Se l'operando

punta effettivamente su un oggetto `T` , il risultato punta a quell'oggetto. Altrimenti, il risultato non è specificato.

C ++ 11

Anche se l'operando non punta a un oggetto `T` , finché l'operando punta a un byte il cui indirizzo è correttamente allineato per il tipo `T` , il risultato della conversione punta allo stesso byte.

```
// allocating an array of 100 ints, the hard way
int* a = malloc(100*sizeof(*a));                    // error; malloc returns void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok
// int* a = new int[100];                           // no cast needed
// std::vector<int> a(100);                         // better

const char c = '!';
const void* p1 = &c;
const char* p2 = p1;                               // error
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c
const int* p4 = static_cast<const int*>(p1);   // unspecified in C++03;
                                               // possibly unspecified in C++11 if
                                               // alignof(int) > alignof(char)
char* p5 = static_cast<char*>(p1);             // error: casting away constness
```

Casting in stile C.

Il cast di C-Style può essere considerato il cast "Best effort" e viene chiamato così come è l'unico cast che può essere usato in C. La sintassi per questo cast è `(NewType)variable` .

Ogni volta che questo cast viene usato, usa uno dei seguenti cast di c ++ (in ordine):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

Il casting funzionale è molto simile, anche se come alcune restrizioni come risultato della sua sintassi: `NewType(expression)` . Di conseguenza, è possibile trasmettere solo i tipi senza spazi.

È meglio usare il nuovo cast di c ++, perché è più leggibile e può essere facilmente individuato ovunque all'interno di un codice sorgente C ++ e gli errori verranno rilevati in fase di compilazione, invece in fase di esecuzione.

Poiché questo cast può risultare in `reinterpret_cast` non intenzionale, è spesso considerato pericoloso.

Leggi Conversioni di tipo esplicito online: https://riptutorial.com/it/cplusplus/topic/3090/conversioni-di-tipo-esplicito

Capitolo 25: Copia Elision

Examples

Scopo della copia elisione

Ci sono posti nello standard in cui un oggetto viene copiato o spostato per inizializzare un oggetto. Copia elision (a volte chiamata ottimizzazione del valore di ritorno) è un'ottimizzazione per cui, in determinate circostanze specifiche, un compilatore può evitare la copia o lo spostamento anche se lo standard dice che deve accadere.

Considera la seguente funzione:

```
std::string get_string()
{
  return std::string("I am a string.");
}
```

In base al rigoroso testo dello standard, questa funzione inizializzerà una `std::string` temporanea `std::string` , quindi copierà / sposterà ciò nell'oggetto valore di ritorno, quindi distruggerà il temporaneo. Lo standard è molto chiaro che questo è il modo in cui il codice viene interpretato.

Copia elision è una regola che consente a un compilatore C ++ di *ignorare* la creazione del temporaneo e la sua successiva copia / distruzione. Cioè, il compilatore può prendere l'espressione di inizializzazione per il temporaneo e inizializzare direttamente il valore di ritorno della funzione. Ciò ovviamente risparmia le prestazioni.

Tuttavia, ha due effetti visibili sull'utente:

1. Il tipo deve avere il costruttore copia / sposta che sarebbe stato chiamato. Anche se il compilatore elude copia / sposta, il tipo deve essere ancora in grado di essere copiato / spostato.

2. Gli effetti collaterali dei costruttori copia / mossa non sono garantiti nelle circostanze in cui può verificarsi elisione. Considera quanto segue:

C ++ 11

```
struct my_type
{
  my_type() = default;
  my_type(const my_type &) {std::cout <<"Copying\n";}
  my_type(my_type &&) {std::cout <<"Moving\n";}
};

my_type func()
{
  return my_type();
}
```

Cosa chiamerà `func` ? Bene, non stamperà mai "Copia", poiché il temporaneo è un valore rval e `my_type` è un tipo spostabile. Quindi stamperà "Moving"?

Senza la regola di elisione della copia, sarebbe necessario stampare sempre "In movimento". Ma poiché esiste la regola elision di copia, il costruttore di move può o non può essere chiamato; dipende dall'implementazione.

Pertanto, non è possibile dipendere dal richiamo dei costruttori copia / sposta in contesti in cui è possibile elision.

Poiché elision è un'ottimizzazione, il compilatore potrebbe non supportare elision in tutti i casi. E indipendentemente dal fatto che il compilatore elimini un caso particolare o no, il tipo deve ancora supportare l'operazione che viene elisa. Quindi, se una costruzione di copia viene eliminata, il tipo deve ancora avere un costruttore di copia, anche se non verrà chiamato.

Copia elisione garantita

C ++ 17

Normalmente, elision è un'ottimizzazione. Mentre praticamente tutti i compilatori supportano la copia di elision nel più semplice dei casi, l'elisione pone ancora un peso particolare agli utenti. Vale a dire, il tipo di copia / spostamento che sta per essere eliminato *deve* comunque avere l'operazione di copia / spostamento che è stata eli- minata.

Per esempio:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  return std::lock_guard<std::mutex>(a_mutex);
}
```

Questo potrebbe essere utile nei casi in cui `a_mutex` è un mutex che è detenuto privatamente da qualche sistema, tuttavia un utente esterno potrebbe desiderare di avere un blocco con scope.

Anche questo non è legale, perché `std::lock_guard` non può essere copiato o spostato. Anche se praticamente ogni compilatore C ++ elide la copia / mossa, lo standard *richiede* comunque *che* il tipo abbia quell'operazione disponibile.

Fino al C ++ 17.

C ++ 17 impone l'elisione ridefinendo efficacemente il significato stesso di certe espressioni in modo che nessuna copia / spostamento avvenga. Considera il codice sopra.

Sotto la dicitura pre-C ++ 17, tale codice dice di creare un valore temporaneo e quindi di utilizzare il temporaneo per copiare / spostare il valore restituito, ma la copia temporanea può essere eliminata. Sotto la dicitura C ++ 17, ciò non crea affatto un temporaneo.

In C ++ 17, qualsiasi espressione di prvalore , se utilizzata per inizializzare un oggetto dello stesso tipo dell'espressione, non genera un valore temporaneo. L'espressione inizializza direttamente

quell'oggetto. Se si restituisce un valore dello stesso tipo del valore restituito, il tipo non deve necessariamente avere un costruttore copia / sposta. E quindi, con le regole del C ++ 17, il codice sopra può funzionare.

La dicitura C ++ 17 funziona nei casi in cui il tipo del prvalue corrisponde al tipo da inizializzare. Quindi dato `get_lock` sopra, questo non richiederà nemmeno una copia / mossa:

```
std::lock_guard the_lock = get_lock();
```

Poiché il risultato di `get_lock` è un'espressione di prvalue utilizzata per inizializzare un oggetto dello stesso tipo, non si verificherà alcuna copia o spostamento. Quell'espressione non crea mai un temporaneo; è usato per inizializzare direttamente `the_lock` . Non c'è elisione perché non vi è alcuna copia / mossa da elidere.

Il termine "copia elisione garantita" è quindi un termine improprio, ma questo è il nome della funzionalità, come viene proposto per la standardizzazione C ++ 17 . Non garantisce affatto l'elisione; *elimina* la copia / sposta del tutto, ridefinendo il C ++ in modo che non ci sia mai stata una copia / mossa da eliminare.

Questa funzione funziona solo nei casi che coinvolgono un'espressione di valore. In quanto tale, usa le solite regole di elision:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  std::lock_guard<std::mutex> my_lock(a_mutex);
  //Do stuff
  return my_lock;
}
```

Mentre questo è un caso valido per copia elision, le regole C ++ 17 non *eliminano* la copia / spostamento in questo caso. In quanto tale, il tipo deve avere ancora un costruttore di copia / spostamento da utilizzare per inizializzare il valore di ritorno. E poiché `lock_guard` non lo fa, questo è ancora un errore di compilazione. Le implementazioni possono rifiutarsi di eludere le copie quando si passa o si restituisce un oggetto di tipo banalmente copiabile. Questo per consentire di spostare tali oggetti nei registri, che alcune ABI potrebbero imporre nelle loro convenzioni di chiamata.

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

Valore di ritorno elisione

Se si restituisce un'espressione di valore da una funzione e l'espressione di prvalue ha lo stesso tipo del tipo di ritorno della funzione, è possibile elidere la copia dal valore provvisorio di prvalore:

```
std::string func()
{
  return std::string("foo");
}
```

Praticamente tutti i compilatori elideranno la costruzione temporanea in questo caso.

Parametro elisione

Quando si passa un argomento a una funzione e l'argomento è un'espressione di valore del tipo di parametro della funzione e questo tipo non è un riferimento, è possibile elidere la costruzione del prvalue.

```
void func(std::string str) { ... }

func(std::string("foo"));
```

Questo dice di creare una `string` temporanea, quindi spostarla nel parametro funzione `str` . Copia elision permette a questa espressione di creare direttamente l'oggetto in `str` , invece di usare una mossa temporanea +.

Questa è un'utile ottimizzazione per i casi in cui un costruttore è dichiarato `explicit` . Ad esempio, potremmo aver scritto quanto sopra come `func("foo")` , ma solo perché la `string` ha un costruttore implicito che converte da un `const char*` in una `string` . Se quel costruttore fosse `explicit` , saremmo costretti a usare un temporaneo per chiamare il costruttore `explicit` . Copia elision ci salva dal dover fare una copia / mossa inutile.

Valore di ritorno denominato elisione

Se si restituisce un'espressione lvalue da una funzione e questo lvalue:

- rappresenta una variabile automatica locale per quella funzione, che verrà distrutta dopo il `return`
- la variabile automatica non è un parametro di funzione
- e il tipo della variabile è dello stesso tipo del tipo di ritorno della funzione

Se tutti questi sono i casi, è possibile eliminare la copia / spostamento dal valore l:

```
std::string func()
{
  std::string str("foo");
  //Do stuff
  return str;
}
```

Casi più complessi sono eleggibili per elision, ma più è complesso il caso, minore è la probabilità che il compilatore lo eleggi effettivamente:

```
std::string func()
{
```

```
  std::string ret("foo");
  if(some_condition)
  {
    return "bar";
  }
  return ret;
}
```

Il compilatore potrebbe ancora elidere `ret` , ma le probabilità di loro facendo così scendere.

Come notato in precedenza, elision non è consentito per i *parametri di* valore.

```
std::string func(std::string str)
{
  str.assign("foo");
  //Do stuff
  return str; //No elision possible
}
```

Copia l'inizializzazione elision

Se si utilizza un'espressione di valore per copiare l'inizializzazione di una variabile e tale variabile ha lo stesso tipo dell'espressione di valore, la copia può essere eliminata.

```
std::string str = std::string("foo");
```

L'inizializzazione della copia lo trasforma efficacemente in `std::string str("foo");` (ci sono piccole differenze).

Questo funziona anche con i valori di ritorno:

```
std::string func()
{
  return std::string("foo");
}

std::string str = func();
```

Senza copiare elision, ciò provocherebbe 2 chiamate al costruttore di spostamenti di `std::string` . Copia elision consente di chiamare il costruttore di movimento 1 o zero volte e la maggior parte dei compilatori sceglierà quest'ultimo.

Leggi Copia Elision online: https://riptutorial.com/it/cplusplus/topic/2489/copia-elision

Capitolo 26: Copia vs Assegnazione

Sintassi

- **Copia il costruttore**
- MyClass (const MyClass e altro);
- MyClass (MyClass e altro);
- MyClass (volatile const MyClass e altro);
- MyClass (volatile MyClass e altro);
- **Costruttore di assegnazione**
- MyClass & operator = (const MyClass & rhs);
- MyClass & operator = (MyClass & rhs);
- MyClass & operator = (MyClass rhs);
- const MyClass & operator = (const MyClass & rhs);
- const MyClass & operator = (MyClass & rhs);
- const MyClass & operator = (MyClass rhs);
- Operatore MyClass = (const MyClass & rhs);
- Operatore MyClass = (MyClass & rhs);
- Operatore MyClass = (MyClass rhs);

Parametri

RHS	Lato destro dell'uguaglianza per entrambi i costruttori di copia e assegnazione. Ad esempio il costruttore del compito: MyClass operator = (MyClass & rhs);
segnaposto	segnaposto

Osservazioni

Altre buone risorse per ulteriori ricerche:

Qual è la differenza tra l'operatore di assegnazione e il costruttore di copie?

operatore di assegnazione e costruttore di copia C ++

GeeksForGeeks

Articoli C ++

Examples

Operatore di assegnazione

L'Operatore di assegnazione è quando si sostituiscono i dati con un oggetto già esistente (precedentemente inizializzato) con alcuni dati di altri oggetti. Prendiamo questo come esempio:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
            data = rhs.data;
            return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2(42);
   foo = foo2; // Assignment Operator Called
   cout << foo.data << endl; //Prints 42
}
```

Potete vedere qui chiamo l'operatore di assegnazione quando ho già inizializzato l'oggetto `foo` . Poi più tardi assegnerò `foo2` a `foo` . Tutte le modifiche da visualizzare quando si chiama quell'operatore con segno di uguale sono definite nella funzione `operator=` . Puoi vedere un output eseguibile qui: http://cpp.sh/3qtbm

Copia il costruttore

Copia costruttore d'altra parte, è l'esatto opposto del Costruttore di assegnazione. Questa volta, viene utilizzato per inizializzare un oggetto già inesistente (o non precedentemente inizializzato). Ciò significa che copia tutti i dati dall'oggetto a cui lo si sta assegnando, senza in realtà inizializzare l'oggetto su cui si sta copiando. Ora diamo un'occhiata allo stesso codice di prima, ma modifichiamo il costruttore di assegnamenti per essere un costruttore di copie:

```
// Copy Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;
```

```
class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2 = foo; // Copy Constructor called
   cout << foo2.data << endl;
}
```

Puoi vedere qui `Foo foo2 = foo;` nella funzione principale assegno immediatamente l'oggetto prima di inizializzarlo effettivamente, cosa che come detto prima significa che è un costruttore di copie. E notate che non ho bisogno di passare il parametro int per l'oggetto `foo2` dato che ho tirato automaticamente i dati precedenti dall'oggetto foo. Ecco un esempio di output: http://cpp.sh/5iu7

Copia Costruttore Assegnazione Vs Costruttore

Ok, abbiamo esaminato brevemente cosa sono sopra il costruttore di copie e il costruttore di assegnamenti e abbiamo fornito esempi di ciascuno di essi, ora vediamo entrambi nello stesso codice. Questo codice sarà simile come sopra due. Prendiamo questo:

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
```

```
        return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) / Normal Constructor called
   Foo foo2 = foo; //Copy Constructor Called
   cout << foo2.data << endl;

   Foo foo3(42);
   foo3=foo; //Assignment Constructor Called
   cout << foo3.data << endl;
}
```

Produzione:

```
2
2
```

Qui puoi vedere che prima chiamiamo il costruttore di copie eseguendo la riga `Foo foo2 = foo;` . Dal momento che non l'abbiamo inizializzato in precedenza. E poi chiamiamo l'operatore di assegnazione su foo3 poiché era già inizializzato `foo3=foo` ;

Leggi Copia vs Assegnazione online: https://riptutorial.com/it/cplusplus/topic/7158/copia-vs-assegnazione

Capitolo 27: Costruire sistemi

introduzione

C ++, come C, ha una lunga e variegata storia riguardante i flussi di lavoro di compilazione e i processi di compilazione. Oggi, C ++ ha vari sistemi di compilazione popolari che vengono utilizzati per compilare programmi, a volte per piattaforme multiple all'interno di un sistema di generazione. Qui, alcuni sistemi di costruzione saranno rivisti e analizzati.

Osservazioni

Attualmente, non esiste un sistema di creazione universale o dominante per C ++ che sia popolare e multipiattaforma. Tuttavia, esistono diversi sistemi di build principali collegati a piattaforme / progetti principali, il più notevole dei quali è GNU Make con il sistema operativo GNU / Linux e NMAKE con il sistema di progetto Visual C ++ / Visual Studio.

Inoltre, alcuni IDE (Integrated Development Environments) includono anche sistemi di build specializzati da utilizzare specificamente con l'IDE nativo. Alcuni generatori di sistemi di generazione possono generare questi formati nativi di progetti / sistemi di sviluppo IDE, come CMake per Eclipse e Microsoft Visual Studio 2012.

Examples

Generazione dell'ambiente di compilazione con CMake

CMake genera ambienti di compilazione per quasi tutti i compilatori o IDE da una singola definizione di progetto. I seguenti esempi mostreranno come aggiungere un file CMake al codice C ++ multipiattaforma "Hello World" .

I file CMake vengono sempre chiamati "CMakeLists.txt" e dovrebbero già esistere nella directory principale di ogni progetto (e possibilmente anche nelle sottodirectory). Un file CMakeLists.txt di base ha il seguente aspetto:

```
cmake_minimum_required(VERSION 2.4)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

Guardalo dal vivo su Coliru .

Questo file indica a CMake il nome del progetto, quale versione del file aspettarsi e le istruzioni per generare un eseguibile chiamato "HelloWorld" che richiede `main.cpp` .

Generare un ambiente di compilazione per il compilatore / IDE installato dalla riga di comando:

```
> cmake .
```

Costruisci l'applicazione con:

```
> cmake --build .
```

Ciò genera l'ambiente di generazione predefinito per il sistema, a seconda del sistema operativo e degli strumenti installati. Mantieni il codice sorgente pulito da eventuali artefatti di build con l'uso di build "out-of-source":

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake può anche astrarre i comandi di base della piattaforma shell dall'esempio precedente:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake include generatori per numerosi strumenti di sviluppo e IDE comuni. Per generare makefile per il `nmake` Visual Studio :

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Compilare con GNU make

introduzione

GNU Make (styled `make`) è un programma dedicato all'automazione dell'esecuzione dei comandi della shell. GNU Make è un programma specifico che rientra nella famiglia Make. Rendere popolare tra i sistemi operativi simil-Unix e POSIX, compresi quelli derivati dal kernel Linux, Mac OS X e BSD.

GNU Make è particolarmente importante per essere collegato al Progetto GNU, che è collegato al popolare sistema operativo GNU / Linux. GNU Make ha anche versioni compatibili in esecuzione su vari gusti di Windows e Mac OS X. È anche una versione molto stabile con un significato storico che rimane popolare. È per questi motivi che GNU Make viene spesso insegnato insieme a C e C ++.

Regole di base

Per compilare con make, crea un Makefile nella directory del tuo progetto. Il tuo Makefile potrebbe

essere semplice come:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called
without target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
    @$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
    @rm -f $(EXE)
```

> **NOTA: assicurati che le rientranze siano con una linguetta, non con quattro spazi. Altrimenti, riceverai un errore di** `Makefile:10: *** missing separator. Stop.`

Per eseguire ciò dalla riga di comando, effettuare le seguenti operazioni:

```
$ cd ~/Path/to/project
$ make
$ ls
app  main.cpp  Makefile

$ ./app
Hello World!

$ make clean
$ ls
main.cpp  Makefile
```

Build incrementali

Quando inizi ad avere più file, render diventa più utile. Cosa succede se hai modificato **a.cpp** ma non **b.cpp** ? La ricompilazione di **b.cpp** richiederebbe più tempo.

Con la seguente struttura di directory:

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

Questo sarebbe un buon Makefile:

Makefile

```
CXX=g++
CXXFLAGS=-Wall -Wextra -pedantic
EXE=app

SRCS_GLOB=src/*.cpp
SRCS=$(wildcard $(SRCS_GLOB))
OBJS=$(SRCS:.cpp=.o)

all: $(EXE)

$(EXE): $(OBJS)
    @$(CXX) -o $@ $(OBJS)

depend: .depend

.depend: $(SRCS)
    @-rm -f ./.depend
    @$(CXX) $(CXXFLAGS) -MM $^>>./.depend

clean:
    -rm -f $(EXE)
    -rm $(OBJS)
    -rm *~
    -rm .depend

include .depend
```

Di nuovo guarda le schede. Questo nuovo Makefile ti assicura di ricompilare solo i file modificati, riducendo al minimo il tempo di compilazione.

Documentazione

Per ulteriori informazioni, consultare la documentazione ufficiale della Free Software Foundation ,

la documentazione StackOverflow e la risposta elaborata di dmckee su StackOverflow .

Costruire con SCons

È possibile creare il codice C ++ multipiattaforma "Hello World" utilizzando Scons , uno strumento di costruzione di software in lingua Python .

Per prima cosa, crea un file chiamato `SConstruct` (nota che SCons cercherà un file con questo nome esatto per impostazione predefinita). Per ora, il file dovrebbe trovarsi in una directory proprio lungo il tuo `hello.cpp` . Scrivi nel nuovo file la riga

```
Program('hello.cpp')
```

Ora, dal terminale, `scons` . Dovresti vedere qualcosa di simile

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: done building targets.
```

(anche se i dettagli varieranno a seconda del sistema operativo e del compilatore installato).

Le classi `Environment` e `Glob` ti aiuteranno a configurare ulteriormente cosa costruire. Ad esempio, il file di `SConstruct`

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
    )

env.Program('hello', Glob('src/*.cpp'))
```

costruisce l'eseguibile `hello` , usando tutti i file `cpp` in `src` . Il suo `CPPPATH` è `/usr/include/boost` e specifica lo standard C ++ 11.

Ninja

introduzione

Il sistema di costruzione Ninja è descritto dal suo sito web del progetto come "un piccolo sistema di costruzione con un focus sulla velocità". Ninja è progettato per generare i file generati dai generatori di file di sistema e adotta un approccio a basso livello per la creazione di sistemi, in contrasto con i gestori di sistemi di generazione di livello superiore come CMake o Meson.

Ninja è scritto principalmente in C ++ e Python ed è stato creato come alternativa al sistema di

build SCons per il progetto Chromium.

NMAKE (Utilità di manutenzione programma Microsoft)

introduzione

NMAKE è un'utilità da riga di comando sviluppata da Microsoft per essere utilizzata principalmente in combinazione con Microsoft Visual Studio e / o gli strumenti della riga di comando di Visual C ++.

NMAKE è un sistema di build che rientra nel sistema Make family of build, ma ha alcune caratteristiche distinte che si discostano dai programmi Make di Unix, come il supporto della sintassi del percorso del file specifico di Windows (che a sua volta differisce dai percorsi di file in stile Unix).

Autotools (GNU)

introduzione

Gli Autotools sono un gruppo di programmi che creano un sistema di generazione GNU per un determinato pacchetto software. È una suite di strumenti che lavorano insieme per produrre varie risorse di compilazione, come un Makefile (da utilizzare con GNU Make). Pertanto, gli Autotools possono essere considerati un generatore di sistemi di costruzione di fatto.

Alcuni programmi importanti di Autotools includono:

- autoconf
- Automake (da non confondere con `make`)

In generale, Autotools ha lo scopo di generare lo script compatibile con Unix e Makefile per consentire il seguente comando per compilare (oltre ad installare) la maggior parte dei pacchetti (nel caso semplice):

```
./configure && make && make install
```

Come tale, Autotools ha anche una relazione con alcuni gestori di pacchetti, specialmente quelli che sono collegati a sistemi operativi conformi agli standard POSIX.

Leggi Costruire sistemi online: https://riptutorial.com/it/cplusplus/topic/8200/costruire-sistemi

Capitolo 28: Data e ora usando intestazione

Examples

Misurare il tempo usando

Il `system_clock` può essere utilizzato per misurare il tempo trascorso durante una parte dell'esecuzione di un programma.

c ++ 11

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

In questo esempio, `sleep_for` stato utilizzato per rendere attivo il thread attivo per un periodo di tempo misurato in `std::chrono::seconds` , ma il codice tra parentesi potrebbe essere qualsiasi chiamata di funzione che richiede un po 'di tempo per essere eseguita.

Trova il numero di giorni tra due date

Questo esempio mostra come trovare il numero di giorni tra due date. Una data è specificata per anno / mese / giorno del mese e in aggiunta ora / minuto / secondo.

Il programma calcola il numero di giorni in anni dal 2000.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/***
 * Creates a std::tm structure from raw date.
 *
 * \param year (must be 1900 or greater)
 * \param month months since January - [1, 12]
 * \param day day of the month - [1, 31]
 * \param minutes minutes after the hour - [0, 59]
 * \param seconds seconds after the minute - [0, 61](until C++11) / [0, 60] (since C++11)
 *
```

```
 * Based on http://en.cppreference.com/w/cpp/chrono/c/tm
 */
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
    tm_ret.tm_mon = month - 1;
    tm_ret.tm_year = year - 1900;

    return tm_ret;
}

int get_days_in_year(int year) {

    using namespace std;
    using namespace std::chrono;

    // We want results to be in days
    typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

    // Create start time span
    std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
    auto tms = system_clock::from_time_t(std::mktime(&tm_start));

    // Create end time span
        std::tm tm_end =   CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
    auto tme = system_clock::from_time_t(std::mktime(&tm_end));

    // Calculate time duration between those two dates
    auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

    return diff_in_days.count();
}

int main()
{
    for ( int year = 2000; year <= 2016; ++year )
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}
```

Leggi Data e ora usando intestazione online: https://riptutorial.com/it/cplusplus/topic/3936/data-e-ora-usando--chrono--intestazione

Capitolo 29: decltype

introduzione

La parola chiave `decltype` può essere utilizzata per ottenere il tipo di una variabile, una funzione o un'espressione.

Examples

Esempio di base

Questo esempio illustra come è possibile utilizzare questa parola chiave.

```
int a = 10;

// Assume that type of variable 'a' is not known here, or it may
// be changed by programmer (from int to long long, for example).
// Hence we declare another variable, 'b' of the same type using
// decltype keyword.
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

Se, ad esempio, qualcuno cambia, digita "a" per:

```
float a=99.0f;
```

Quindi il tipo di variabile `b` ora diventa automaticamente `float` .

Un altro esempio

Diciamo che abbiamo il vettore:

```
std::vector<int> intVector;
```

E vogliamo dichiarare un iteratore per questo vettore. Un'idea ovvia è usare l' `auto` . Tuttavia, potrebbe essere necessario semplicemente dichiarare una variabile iteratore (e non assegnarla a qualcosa). Vorremmo fare:

```
vector<int>::iterator iter;
```

Tuttavia, con `decltype` diventa facile e meno soggetto a errori (se il tipo di `intVector` cambia).

```
decltype(intVector)::iterator iter;
```

In alternativa:

```
decltype(intVector.begin()) iter;
```

Nel secondo esempio, il tipo di ritorno di `begin` viene utilizzato per determinare il tipo effettivo, che è `vector<int>::iterator` .

Se abbiamo bisogno di un const_iterator, abbiamo solo bisogno di usare `cbegin` :

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

Leggi decltype online: https://riptutorial.com/it/cplusplus/topic/9930/decltype

Capitolo 30: Digita la cancellazione

introduzione

La cancellazione dei tipi è un insieme di tecniche per la creazione di un tipo che può fornire un'interfaccia uniforme a vari tipi sottostanti, nascondendo al contempo le informazioni sul tipo sottostante dal client. `std::function<R(A...)>` , che ha la capacità di contenere oggetti chiamabili di vario tipo, è forse l'esempio più noto di cancellazione di tipo in C ++.

Examples

Meccanismo di base

Digitare la cancellazione è un modo per nascondere il tipo di un oggetto dal codice che lo utilizza, anche se non è derivato da una classe base comune. In tal modo, fornisce un ponte tra i mondi del polimorfismo statico (modelli, sul luogo di utilizzo, il tipo esatto deve essere noto al momento della compilazione, ma non è necessario dichiararlo conforme a un'interfaccia alla definizione) e polimorfismo dinamico (ereditarietà e funzioni virtuali: nel luogo di utilizzo, il tipo esatto non deve essere conosciuto al momento della compilazione, ma deve essere dichiarato conforme a un'interfaccia alla definizione).

Il codice seguente mostra il meccanismo di base della cancellazione dei tipi.

```
#include <ostream>

class Printable
{
public:
  template <typename T>
  Printable(T value) : pValue(new Value<T>(value)) {}
  ~Printable() { delete pValue; }
  void print(std::ostream &os) const { pValue->print(os); }

private:
  Printable(Printable const &)        /* in C++1x: =delete */; // not implemented
  void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
  struct ValueBase
  {
      virtual ~ValueBase() = default;
      virtual void print(std::ostream &) const = 0;
  };
  template <typename T>
  struct Value : ValueBase
  {
      Value(T const &t) : v(t) {}
      virtual void print(std::ostream &os) const { os << v; }
      T v;
  };
  ValueBase *pValue;
};
```

Nel sito di utilizzo, solo la definizione di cui sopra deve essere visibile, proprio come con le classi di base con funzioni virtuali. Per esempio:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Si noti che questo *non* è un modello, ma una funzione normale che deve essere dichiarata solo in un file di intestazione e che può essere definita in un file di implementazione (diversamente dai modelli, la cui definizione deve essere visibile nel luogo di utilizzo).

Alla definizione del tipo concreto, non è necessario conoscere nulla di `Printable` , deve solo essere conforme all'interfaccia, come con i modelli:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
  return os << "MyType {" << mc.i << "}";
}
```

Ora possiamo passare un oggetto di questa classe alla funzione definita sopra:

```
MyType foo = { 42 };
print_value(foo);
```

Cancellazione fino a un tipo normale con vtable manuale

Il C ++ prospera su ciò che è conosciuto come un tipo Regolare (o almeno Pseudo-Regolare).

Un tipo normale è un tipo che può essere costruito e assegnato a e assegnato da tramite copia o sposta, può essere distrutto e può essere paragonato allo stesso modo. Può anche essere costruito senza argomenti. Infine, ha anche il supporto per alcune altre operazioni che sono molto utili in vari algoritmi e contenitori `std` .

Questo è il documento principale , ma in C ++ 11 vorrebbe aggiungere il supporto `std::hash` .

Userò l'approccio manuale vtable per digitare la cancellazione qui.

```
using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
  return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self);
}};
}
struct regular_vtable {
  void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
  void(*move_assign)(void* dest, void* src); // T&=(T&&)
  bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
  bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
```

```
  std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
  std::type_info const&(*type)(); // typeid(T)
  dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
  return {
    [](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
    [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
    [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T
const*>(rhs); },
    [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T
const*>(lhs),*static_cast<T const*>(rhs)); },
    [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
    []()->decltype(auto){ return typeid(T); },
    [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
  };
}
template<class T>
regular_vtable const* get_regular_vtable() noexcept {
  static const regular_vtable vtable=make_regular_vtable<T>();
  return &vtable;
}

struct regular_type {
  using self=regular_type;
  regular_vtable const* vtable = 0;
  dtor_unique_ptr ptr{nullptr, [](void*){}};

  bool empty() const { return !vtable; }

  template<class T, class...Args>
  void emplace( Args&&... args ) {
    ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
    if (ptr)
      vtable = get_regular_vtable<T>();
    else
      vtable = nullptr;
  }
  friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
    if (lhs.vtable != rhs.vtable) return false;
    return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
  }
  bool before(regular_type const& rhs) const {
    auto const& lhs = *this;
    if (!lhs.vtable || !rhs.vtable)
      return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
    if (lhs.vtable != rhs.vtable)
      return lhs.vtable->type().before(rhs.vtable->type());
    return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
  }
  // technically friend bool operator< that calls before is also required

  std::type_info const* type() const {
    if (!vtable) return nullptr;
    return &vtable->type();
  }
  regular_type(regular_type&& o):
    vtable(o.vtable),
    ptr(std::move(o.ptr))
```

```
  {
    o.vtable = nullptr;
  }
  friend void swap(regular_type& lhs, regular_type& rhs){
    std::swap(lhs.ptr, rhs.ptr);
    std::swap(lhs.vtable, rhs.vtable);
  }
  regular_type& operator=(regular_type&& o) {
    if (o.vtable == vtable) {
      vtable->move_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = std::move(o);
    swap(*this, tmp);
    return *this;
  }
  regular_type(regular_type const& o):
    vtable(o.vtable),
    ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr{nullptr, [](void*){}})
  {
    if (!ptr && vtable) vtable = nullptr;
  }
  regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
      vtable->copy_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
  }
  std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
  }
  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
  >
  regular_type(T&& t) {
    emplace<std::decay_t<T>>(std::forward<T>(t));
  }
};
namespace std {
  template<>
  struct hash<regular_type> {
    std::size_t operator()( regular_type const& r )const {
      return r.hash();
    }
  };
  template<>
  struct less<regular_type> {
    bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
      return lhs.before(rhs);
    }
  };
}
```

esempio dal vivo

Un tipo così regolare può essere usato come chiave per una `std::map` o una `std::unordered_map`

che accetta *qualsiasi cosa normale* per una chiave, come:

```
std::map<regular_type, std::any>
```

sarebbe fondamentalmente una mappa da regolare normale, a tutto ciò che è possibile copiare.

Diversamente da `any` , il mio `regular_type` non ottimizza gli oggetti di piccole dimensioni e non supporta il recupero dei dati originali. Ottenere il tipo originale non è difficile.

L'ottimizzazione degli oggetti di piccole dimensioni richiede che noi archiviamo un buffer di archiviazione allineato all'interno del `regular_type` , e `regular_type` attentamente il deleter del `ptr` per distruggere solo l'oggetto e non cancellarlo.

Vorrei iniziare a `make_dtor_unique_ptr` e insegnargli come memorizzare i dati a volte in un buffer e quindi nell'heap se non c'è spazio nel buffer. Potrebbe essere sufficiente.

Un solo spostamento `std :: function`

`std::function` tipo di `std::function` cancella fino a poche operazioni. Una delle cose che richiede è che il valore memorizzato sia copiabile.

Ciò causa problemi in alcuni contesti, come lambdas che memorizza ptr unici. Se si utilizza la `std::function` in un contesto in cui la copia non è importante, come un pool di thread in cui si inviano attività ai thread, questo requisito può aggiungere un sovraccarico.

In particolare, `std::packaged_task<Sig>` è un oggetto callable che è solo move. Puoi archiviare un file `std::packaged_task<R(Args...)>` in un file `std::packaged_task<void(Args...)>` , ma questo è un modo abbastanza pesante e oscuro per creare un solo spostamento callable type-erasure class.

Quindi il `task` . Questo dimostra come potresti scrivere un semplice tipo di `std::function` . Ho omesso il costruttore di copie (che implicherebbe l'aggiunta di un metodo `clone` a `details::task_pimpl<...>`).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
namespace details {
  template<class R, class...Args>
  struct task_pimpl {
    virtual R invoke(Args&&...args) const = 0;
    virtual ~task_pimpl() {};
    virtual const std::type_info& target_type() const = 0;
  };

  // store an F.  invoke(Args&&...) calls the f
  template<class F, class R, class...Args>
  struct task_pimpl_impl:task_pimpl<R,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual R invoke(Args&&...args) const final override {
```

```
      return f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };

  // the void version discards the return value of f:
  template<class F, class...Args>
  struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual void invoke(Args&&...args) const final override {
      f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };
};

template<class R, class...Args>
struct task<R(Args...)> {
  // semi-regular:
  task()=default;
  task(task&&)=default;
  // no copy

private:
  // aliases to make some SFINAE code below less ugly:
  template<class F>
  using call_r = std::result_of_t<F const&(Args...)>;
  template<class F>
  using is_task = std::is_same<std::decay_t<F>, task>;
public:
  // can be constructed from a callable F
  template<class F,
    // that can be invoked with Args... and converted-to-R:
    class= decltype( (R)(std::declval<call_r<F>>()) ),
    // and is not this same type:
    std::enable_if_t<!is_task<F>{}, int>* = nullptr
  >
  task(F&& f):
    m_pImpl( make_pimpl(std::forward<F>(f)) )
  {}

  // the meat: the call operator
  R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
  }
  explicit operator bool() const {
    return (bool)m_pImpl;
  }
  void swap( task& o ) {
    std::swap( m_pImpl, o.m_pImpl );
  }
  template<class F>
  void assign( F&& f ) {
    m_pImpl = make_pimpl(std::forward<F>(f));
  }
```

```
  // Part of the std::function interface:
  const std::type_info& target_type() const {
    if (!*this) return typeid(void);
    return m_pImpl->target_type();
  }
  template< class T >
  T* target() {
    return target_impl<T>();
  }
  template< class T >
  const T* target() const {
    return target_impl<T>();
  }
  // compare with nullptr    :
  friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
  friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
  friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }
  friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
  template<class T>
  using pimpl_t = details::task_pimpl_impl<T, R, Args...>;

  template<class F>
  static auto make_pimpl( F&& f ) {
    using dF=std::decay_t<F>;
    using pImpl_t = pimpl_t<dF>;
    return std::make_unique<pImpl_t>(std::forward<F>(f));
  }
  std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;

  template< class T >
  T* target_impl() const {
    return dynamic_cast<pimpl_t<T>*>(m_pImpl.get());
  }
};
```

Per rendere questa libreria degna, dovresti aggiungere una piccola ottimizzazione del buffer, in modo che non memorizzi tutti i callable nell'heap.

L'aggiunta di SBO richiederebbe un'attività non predefinita `task(task&&)` , alcuni `std::aligned_storage_t` all'interno della classe, un `m_pImpl` `unique_ptr` con un deleter che può essere impostato su destroy-only (e non restituire la memoria `emplace_move_to( void* ) = 0`) e un `emplace_move_to( void* ) = 0` nel `task_pimpl` .

esempio dal vivo del codice precedente (senza SBO).

Cancellando fino a un buffer contiguo di T

Non tutti i tipi di cancellazione includono ereditarietà virtuale, allocazioni, posizionamenti nuovi o persino puntatori di funzioni.

Ciò che rende la cancellazione del tipo cancellato è che descrive un (o più) comportamento (i), e accetta qualsiasi tipo che supporti quel comportamento e lo avvolge. Tutte le informazioni che non sono in quell'insieme di comportamenti sono "dimenticate" o "cancellate".

Un `array_view` prende il suo intervallo o tipo di contenitore in entrata e cancella tutto tranne il fatto

che si tratta di un buffer contiguo di `T`

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} ||
std::is_same< data_t<Src>, std::remove_const_t<T>* >{}>;

template<class T>
struct array_view {
  // the core of the class:
  T* b=nullptr;
  T* e=nullptr;
  T* begin() const { return b; }
  T* end() const { return e; }

  // provide the expected methods of a good contiguous range:
  T* data() const { return begin(); }
  bool empty() const { return begin()==end(); }
  std::size_t size() const { return end()-begin(); }

  T& operator[](std::size_t i)const{ return begin()[i]; }
  T& front()const{ return *begin(); }
  T& back()const{ return *(end()-1); }

  // useful helpers that let you generate other ranges from this one
  // quickly and safely:
  array_view without_front( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin()+i, end()};
  }
  array_view without_back( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin(), end()-i};
  }

  // array_view is plain old data, so default copy:
  array_view(array_view const&)=default;
  // generates a null, empty range:
  array_view()=default;

  // final constructor:
  array_view(T* s, T* f):b(s),e(f) {}
  // start and length is useful in my experience:
  array_view(T* s, std::size_t length):array_view(s, s+length) {}

  // SFINAE constructor that takes any .data() supporting container
  // or other range in one fell swoop:
  template<class Src,
    std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
    std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
  >
  array_view( Src&& src ):
    array_view( src.data(), src.size() )
  {}

  // array constructor:
  template<std::size_t N>
  array_view( T(&arr)[N] ):array_view(arr, N) {}
```

```
  // initializer list, allowing {} based:
  template<class U,
    std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
  >
  array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}
};
```

un `array_view` prende qualsiasi contenitore che supporti `.data()` restituendo un puntatore a `T` e un metodo `.size()` , o un array, e lo cancella come un intervallo di accesso casuale su `T` contigui.

Può prendere uno `std::vector<T>` , uno `std::string<T>` uno `std::array<T, N>` a `T[37]` , un elenco inizializzatore (compresi quelli basati su `{}`), o qualcos'altro si compone che lo supporta (tramite `T* x.data()` e `size_t x.size()`).

In questo caso, i dati che possiamo estrarre dalla cosa che stiamo cancellando, insieme al nostro stato di "non visualizzazione", significa che non dobbiamo allocare memoria o scrivere funzioni dipendenti dal tipo personalizzato.

Esempio dal vivo

Un miglioramento sarebbe usare un terzi `data` e un terzo `size` in un contesto ADL abilitato.

Digita cancellando il tipo cancellato con std :: any

Questo esempio utilizza C ++ 14 e `boost::any` . In C ++ 17 puoi invece scambiare in `std::any` .

La sintassi che otteniamo è:

```
const auto print =
  make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });

super_any<decltype(print)> a = 7;

(a->*print)(std::cout);
```

che è quasi ottimale.

Questo esempio è basato sul lavoro di @dyp e @cpplearner e sul mio.

Per prima cosa utilizziamo un tag per passare tipi:

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

Questa classe di caratteristiche ottiene la firma memorizzata con `any_method` :

Questo crea un tipo di puntatore a funzione e una factory per i detti puntatori di funzione, dato un `any_method` :

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
  template<class T>
  using decorate = std::conditional_t< any_method::is_const, T const, T >;

  using any = decorate<boost::any>;

  using type = R(*)(any&, any_method const*, Args&&...);
  template<class T>
  type operator()( tag_t<T> )const{
    return +[](any& self, any_method const* method, Args&&...args) {
      return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
    };
  }
};
```

`any_method_function::type` è il tipo di puntatore a funzione che memorizzeremo insieme all'istanza. `any_method_function::operator()` prende un `tag_t<T>` e scrive un'istanza personalizzata del `any_method_function::type` che presuppone l' `any&` sta per essere una `T`

Vogliamo essere in grado di digitare-cancellare più di un metodo alla volta. Quindi li raggruppiamo in una tupla e scriviamo un wrapper di supporto per incollare la tupla nello storage statico su base per tipo e mantenere un puntatore su di essi.

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
  return std::make_tuple(
    any_method_function<any_methods>{}(tag<T>)...
  );
}

template<class...methods>
struct any_methods {
private:
  any_method_tuple<methods...> const* vtable = 0;
  template<class T>
  static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
    static const auto table = make_vtable<methods...>(tag<T>);
    return &table;
  }
public:
  any_methods() = default;
  template<class T>
  any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
  any_methods& operator=(any_methods const&)=default;
  template<class T>
  void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }
```

```
  template<class any_method>
  auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
  }
};
```

Potremmo specializzarlo in casi in cui il vtable è piccolo (ad esempio, 1 elemento) e utilizzare puntatori diretti memorizzati in classe in questi casi per l'efficienza.

Ora iniziamo il `super_any` . Io uso `super_any_t` per rendere la dichiarazione di `super_any` un po 'più semplice.

```
template<class...methods>
struct super_any_t;
```

Questo ricerca i metodi che il super supporta per SFINAE e messaggi di errore migliori:

```
template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
    std::integral_constant<bool, std::is_same<M0, method>{}  ||
super_method_applies_helper<super_any_t<Methods...>, method>{}>
{};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} >{};
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};
```

Successivamente creiamo il tipo `any_method` . Un `any_method` è un puntatore pseudo-metodo. Lo creiamo a livello globale e `const` sintassi come:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

o in C ++ 17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Nota che l'uso di un non-lambda può rendere le cose pelose, poiché utilizziamo il tipo per una fase

di ricerca. Questo può essere risolto, ma renderebbe questo esempio più lungo di quanto non sia già. Quindi inizializza sempre un metodo qualsiasi da una lambda, o da un tipo parametarizzato su una lambda.

```
template<class Sig, bool const_method, class F>
struct any_method {
  using signature=Sig;
  enum{is_const=const_method};
private:
  F f;
public:

  template<class Any,
    // SFINAE testing that one of the Anys's matches this type:
    std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
  >
  friend auto operator->*( Any&& self, any_method const& m ) {
    // we don't use the value of the any_method, because each any_method has
    // a unique type (!) and we check that one of the auto*'s in the super_any
    // already has a pointer to us.  We then dispatch to the corresponding
    // any_method_data...

    return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)-
>decltype(auto)
    {
      return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
  }
  any_method( F fin ):f(std::move(fin)) {}

  template<class...Args>
  decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
  }
};
```

Un metodo factory, non necessario in C ++ 17, credo:

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
  return {std::forward<F>(f)};
}
```

Questo è il `any` . E 'sia un `any` , e svolge attorno ad un fascio di puntatori a funzione del tipo di cancellazione che cambia quando il contenuto `any` fa:

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
  using vtable=any_methods<methods...>;
public:
  template<class T,
    std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
  >
  super_any_t( T&& t ):
    boost::any( std::forward<T>(t) )
  {
```

```
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
  }

  boost::any& as_any()&{return *this;}
  boost::any&& as_any()&&{return std::move(*this);}
  boost::any const& as_any()const&{return *this;}
  super_any_t()=default;
  super_any_t(super_any_t&& o):
    boost::any( std::move( o.as_any() ) ),
    vtable(o)
  {}
  super_any_t(super_any_t const& o):
    boost::any( o.as_any() ),
    vtable(o)
  {}
  template<class S,
    std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
  >
  super_any_t( S&& o ):
    boost::any( std::forward<S>(o).as_any() ),
    vtable(o)
  {}
  super_any_t& operator=(super_any_t&&)=default;
  super_any_t& operator=(super_any_t const&)=default;

  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>* =nullptr
  >
  super_any_t& operator=( T&& t ) {
    ((boost::any&)*this) = std::forward<T>(t);
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
    return *this;
  }
};
```

Poiché memorizziamo `any_method` come oggetti `const` , ciò rende la creazione di un `super_any` un po 'più semplice:

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Codice di prova:

```
const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p
<< "\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os
<< p << L"\n"; });

int main()
{
  super_any<decltype(print), decltype(wprint)> a = 7;
  super_any<decltype(print), decltype(wprint)> a2 = 7;

  (a->*print)(std::cout);
  (a->*wprint)(std::wcout);
}
```

esempio dal vivo

Originariamente pubblicato qui in una domanda e risposta SO (e le persone indicate sopra hanno aiutato con l'implementazione).

Leggi Digita la cancellazione online: https://riptutorial.com/it/cplusplus/topic/2872/digita-la-cancellazione

Capitolo 31: Digitare parole chiave

Examples

classe

1. Introduce la definizione di un tipo di classe .

```
class foo {
    int x;
  public:
    int get_x();
    void set_x(int new_x);
};
```

2. Introduce un *identificatore di tipo elaborato,* che specifica che il nome seguente è il nome di un tipo di classe. Se il nome della classe è già stato dichiarato, può essere trovato anche se nascosto da un altro nome. Se il nome della classe non è stato già dichiarato, viene dichiarato in avanti.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
class baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
class foo {
    bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
  public:
    foo();
};
```

3. Introduce un parametro di tipo nella dichiarazione di un modello .

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. Nella dichiarazione di un parametro modello di modello , la `class` parola chiave precede il nome del parametro. Poiché l'argomento per un parametro modello di template può essere solo un modello di classe, l'uso della `class` qui è ridondante. Tuttavia, la grammatica di C ++ lo richiede.

```
template <template <class T> class U>
//                          ^^^^^ "class" used in this sense here;
```

```
//                                  U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Si noti che il senso 2 e il senso 3 possono essere combinati nella stessa dichiarazione. Per esempio:

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

C ++ 11

6. Nella dichiarazione o definizione di enum, dichiara l'enum come enum di ambito .

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

struct

Intercambiabile con la `class` , ad eccezione delle seguenti differenze:

- Se un tipo di classe viene definito utilizzando la `struct` parole chiave, l'accessibilità predefinita di basi e membri è `public` anziché `private` .
- `struct` non può essere utilizzato per dichiarare un parametro di tipo template o un parametro template template; solo la `class` può.

enum

1. Introduce la definizione di un tipo di enumerazione .

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

C ++ 11

In C ++ 11, `enum` può facoltativamente essere seguito da una `class` o da una `struct` per definire un enumerato ambito . Inoltre, enumerazioni sia con scope sia senza ambito possono avere il loro

tipo sottostante esplicitamente specificato da `: T` seguendo il nome enum, dove `T` riferisce ad un tipo intero.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Enumerators in normale `enum` s possono anche essere preceduti dall'operatore portata, anche se sono ancora considerati nell'ambito del `enum` è stato definito in.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Introduce un *identificatore di tipo elaborato,* che specifica che il nome seguente è il nome di un tipo enum dichiarato in precedenza. (Un identificatore di tipo elaborato non può essere usato per inoltrare un tipo di enum.) Un enum può essere chiamato in questo modo anche se nascosto da un altro nome.

   ```
   enum Foo { FOO };
   void Foo() {}
   Foo foo = FOO;      // ill-formed; Foo refers to the function
   enum Foo foo = FOO; // ok; Foo refers to the enum type
   ```

C ++ 11

3. Introduce una *dichiarazione enum opaca,* che dichiara un enum senza definirlo. Può o ridichiarare un enum precedentemente dichiarato, o inoltrare-dichiarare un enum che non è stato precedentemente dichiarato.

   Un enum prima dichiarato come ambito non può in seguito essere dichiarato come senza ambito, o *viceversa.* Tutte le dichiarazioni di enum devono concordare nel tipo sottostante.

   Quando si inoltra una enumerazione senza ambito, il tipo sottostante deve essere esplicitamente specificato, poiché non può essere dedotto fino a quando i valori degli enumeratori non saranno noti.

   ```
   enum class Format; // underlying type is implicitly int
   void f(Format f);
   enum class Format {
       TEXT,
       PDF,
   ```

```
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

unione

1. Introduce la definizione di un tipo di unione .

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

2. Introduce un *identificatore di tipo elaborato,* che specifica che il nome seguente è il nome di un tipo di unione. Se il nome del sindacato è già stato dichiarato, può essere trovato anche se nascosto da un altro nome. Se il nome del sindacato non è stato già dichiarato, viene dichiarato in avanti.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Leggi Digitare parole chiave online: https://riptutorial.com/it/cplusplus/topic/7838/digitare-parole-chiave

Capitolo 32: eccezioni

Examples

Cattura eccezioni

Un blocco `try/catch` viene utilizzato per rilevare le eccezioni. Il codice nella sezione `try` è il codice che può generare un'eccezione e il codice nella clausola `catch` gestisce l'eccezione.

```
#include <iostream>
#include <string>
#include <stdexcept>

int main() {
  std::string str("foo");

  try {
      str.at(10); // access element, may throw std::out_of_range
  } catch (const std::out_of_range& e) {
      // what() is inherited from std::exception and contains an explanatory message
      std::cout << e.what();
  }
}
```

È possibile utilizzare più clausole di `catch` per gestire più tipi di eccezioni. Se sono presenti più clausole `catch` , il meccanismo di gestione delle eccezioni tenta di farli corrispondere **in ordine** di aspetto nel codice:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Le classi di eccezioni derivate da una classe base comune possono essere catturate con una singola clausola `catch` per la classe base comune. L'esempio sopra può sostituire le due clausole `catch` per `std::length_error` e `std::out_of_range` con una singola clausola per `std:exception` :

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Poiché le clausole di `catch` vengono provate in ordine, assicurarsi di scrivere prima clausole di cattura più specifiche, altrimenti il codice di gestione delle eccezioni potrebbe non essere mai chiamato:

```
try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
    /* This block of code will never execute, because std::runtime_error inherits
       from std::exception, and all exceptions of type std::exception were already
       caught by the previous catch clause. */
}
```

Un'altra possibilità è il gestore catch-all, che catturerà qualsiasi oggetto lanciato:

```
try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}
```

Rethrow (propagare) l'eccezione

A volte si vuole fare qualcosa con l'eccezione che si cattura (come scrivere per registrare o stampare un avviso) e lasciarlo gonfiare verso l'ambito superiore per essere gestito. Per fare ciò, puoi ripensare a qualsiasi eccezione che ricevi:

```
try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}
```

Usando il `throw;` senza argomenti restituirà l'eccezione attualmente rilevata.

C ++ 11

Per rilanciare uno `std::exception_ptr` gestito, la libreria standard C ++ ha la funzione `rethrow_exception` che può essere utilizzata includendo l'intestazione `<exception>` nel programma.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
```

```
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

Funzione Try Blocks In constructor

L'unico modo per rilevare l'eccezione nell'elenco di inizializzazione:

```
struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};
```

Funzione Prova blocco per funzione normale

```
void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}
```

Che è equivalente a

```
void function_with_try_block()
{
    try
    {
```

```
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}
```

Si noti che per costruttori e distruttori, il comportamento è diverso in quanto il blocco catch re-getta comunque un'eccezione (quella catturata se non c'è nessun altro lancio nel corpo del blocco catch).

La funzione `main` può avere una funzione try block come qualsiasi altra funzione, ma il blocco try della funzione `main` non rileverà eccezioni che si verificano durante la costruzione di una variabile statica non locale o la distruzione di qualsiasi variabile statica. Invece, `std::terminate` viene chiamato.

Funzione Try Blocks In destructor

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Nota che, sebbene sia possibile, bisogna stare molto attenti con il lancio dal distruttore, come se un distruttore chiamato durante lo sbobinamento dello stack lanci un'eccezione, `std::terminate` viene chiamato.

Best practice: lancio per valore, cattura per riferimento const

In generale, è considerata una buona pratica lanciare per valore (piuttosto che per puntatore), ma catturare per riferimento (const).

```
try {
    // throw new std::runtime_error("Error!");   // Don't do this!
    // This creates an exception object
    // on the heap and would require you to catch the
    // pointer and manage the memory yourself. This can
    // cause memory leaks!

    throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
```

Una delle ragioni per cui la cattura per riferimento è una buona pratica è che elimina la necessità di ricostruire l'oggetto quando viene passato al blocco catch (o quando si propagano attraverso altri blocchi catch). La cattura per riferimento consente inoltre di gestire le eccezioni in modo polimorfico ed evitare l'affettamento degli oggetti. Tuttavia, se stai rilanciando un'eccezione (come `throw e;` vedi l'esempio sotto), puoi ancora ottenere l'affettamento degli oggetti perché il `throw e;` l'istruzione fa una copia dell'eccezione come qualunque sia il tipo dichiarato:

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                     // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}
```

Se sei sicuro di non fare nulla per modificare l'eccezione (come aggiungere informazioni o modificare il messaggio), l'acquisizione tramite riferimento const consente al compilatore di effettuare ottimizzazioni e migliorare le prestazioni. Ma questo può ancora causare lo splicing dell'oggetto (come mostrato nell'esempio sopra).

Avviso: fare attenzione a non generare eccezioni involontarie nei blocchi `catch` , in particolare in relazione all'allocazione di memoria o risorse aggiuntive. Ad esempio, la costruzione di `logic_error` , `runtime_error` o delle loro sottoclassi potrebbe generare `bad_alloc` causa della memoria che si esaurisce durante la copia della stringa di eccezione, i flussi di I / O potrebbero generare durante la registrazione con rispettive maschere di eccezione impostate, ecc.

Eccezione annidata

C ++ 11

Durante la gestione delle eccezioni esiste un caso d'uso comune quando si cattura un'eccezione generica da una funzione di basso livello (come un errore del filesystem o di trasferimento dei dati) e si genera un'eccezione più specifica di alto livello che indica che alcune operazioni di alto livello

potrebbero non essere eseguito (come non essere in grado di pubblicare una foto sul Web). Ciò consente alla gestione delle eccezioni di reagire a problemi specifici con operazioni di alto livello e consente inoltre, avendo solo un messaggio di errore, che il programmatore trovi un posto nell'applicazione in cui si è verificata un'eccezione. Il lato negativo di questa soluzione è che l'eccezione del callstack viene troncata e l'eccezione originale viene persa. Ciò impone agli sviluppatori di includere manualmente il testo dell'eccezione originale in uno appena creato.

Le eccezioni nidificate mirano a risolvere il problema allegando un'eccezione di basso livello, che descrive la causa, a un'eccezione di alto livello, che descrive cosa significa in questo caso particolare.

`std::nested_exception` consente di nidificare le eccezioni grazie a `std::throw_with_nested` :

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unkown exception\n";
    }
}

void print_current_exception_with_nested(int level =  0)
{
    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
        //Empty // End recursion
    }
}
```

```
// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch(...) {
        print_current_exception_with_nested();
    }
}
```

Uscita possibile:

```
exception: run() failed
 MyException: Couldn't open nonexistent.file
  exception: basic_ios::clear
```

Se lavori solo con le eccezioni ereditate da `std::exception` , il codice può anche essere semplificato.

std :: uncaught_exceptions

c ++ 17

C ++ 17 introduce `int std::uncaught_exceptions()` (per sostituire il `bool std::uncaught_exception()` limitato `bool std::uncaught_exception()`) per sapere quante eccezioni sono attualmente non rilevate. Ciò consente a una classe di determinare se viene distrutta durante lo svolgimento di una pila o meno.

```
#include <exception>
#include <string>
#include <iostream>

// Apply change on destruction:
// Rollback in case of exception (failure)
// Else Commit (success)
```

```
class Transaction
{
public:
    Transaction(const std::string& s) : message(s) {}
    Transaction(const Transaction&) = delete;
    Transaction& operator =(const Transaction&) = delete;
    void Commit() { std::cout << message << ": Commit\n"; }
    void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

private:
    std::string message;
    int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
public:
    ~Foo() {
        try {
            Transaction transaction("In ~Foo"); // Commit,
                                                // even if there is an uncaught exception
            //...
        } catch (const std::exception& e) {
            std::cerr << "exception/~Foo:" << e.what() << std::endl;
        }
    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
        Foo foo; // ~Foo commit its transaction.
        //...
        throw std::runtime_error("Error");
    } catch (const std::exception& e) {
        std::cerr << "exception/main:" << e.what() << std::endl;
    }
}
```

Produzione:

```
In ~Foo: Commit
In main: Rollback
exception/main:Error
```

Eccezione personalizzata

Non dovresti lanciare valori grezzi come eccezioni, usa invece una delle classi di eccezioni standard o creane una tua.

Avere la propria classe di eccezione ereditata da `std::exception` è un buon modo per farlo. Ecco una classe di eccezioni personalizzata che eredita direttamente da `std::exception` :

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset
    std::string error_message;      ///< Error message

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
        {}

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     *  @return A pointer to a const char*. The underlying memory
     *  is in possession of the Except object. Callers must
     *  not attempt to free the memory.
     */
    virtual const char* what() const throw () {
       return error_message.c_str();
    }

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Un esempio di presa del tiro:

```
try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
             <<"\nError number: "<<e.getErrorNumber()
             <<"\nError offset: "<<e.getErrorOffset();
}
```

Dato che non stai solo lanciando un messaggio di errore stupido, anche altri valori che rappresentano esattamente l'errore, la gestione degli errori diventa molto più efficiente e significativa.

Esiste una classe di eccezioni che consente di gestire bene i messaggi di errore:
`std::runtime_error`

Puoi ereditare anche da questa classe:

```
#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
        {
            error_number = err_num;
            error_offset = err_off;

        }

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
```

```
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Nota che non ho sovrascritto la funzione `what()` dalla classe base (`std::runtime_error`), cioè useremo la versione della classe base di `what()` . Puoi sovrascriverlo se hai ulteriori impegni.

Leggi eccezioni online: https://riptutorial.com/it/cplusplus/topic/1354/eccezioni

Capitolo 33: Enumerazione

Examples

Dichiarazione di enumerazione di base

Le enumerazioni standard consentono agli utenti di dichiarare un nome utile per un insieme di numeri interi. I nomi sono indicati collettivamente come enumeratori. Un'enumerazione e i relativi enumeratori associati sono definiti come segue:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

Un'enumerazione è un *tipo* , distinto da tutti gli altri tipi. In questo caso, il nome di questo tipo è `myEnum` . Ci si aspetta che oggetti di questo tipo assumano il valore di un enumeratore all'interno dell'enumerazione.

Gli enumeratori dichiarati all'interno dell'enumerazione sono valori costanti del tipo dell'enumerazione. Sebbene gli enumeratori siano dichiarati all'interno del tipo, l'operatore scope `::` non è necessario per accedere al nome. Quindi il nome del primo enumeratore è `enumName1` .

C ++ 11

L'operatore dell'ambito può essere facoltativamente utilizzato per accedere a un enumeratore all'interno di un'enumerazione. Quindi `enumName1` può anche essere scritto `myEnum::enumName1` .

Agli enumeratori vengono assegnati valori interi che iniziano da 0 e aumentano di 1 per ciascun enumeratore in un'enumerazione. Quindi, nel caso precedente, `enumName1` ha il valore 0, mentre `enumName2` ha il valore 1.

Agli enumeratori può anche essere assegnato un valore specifico dall'utente; questo valore deve essere un'espressione costante integrale. Gli enumeratori i cui valori non sono esplicitamente forniti avranno il loro valore impostato sul valore dell'enumeratore precedente + 1.

```
enum myEnum
{
    enumName1 = 1, // value will be 1
    enumName2 = 2, // value will be 2
    enumName3,     // value will be 3, previous value + 1
    enumName4 = 7, // value will be 7
    enumName5,     // value will be 8
    enumName6 = 5, // value will be 5, legal to go backwards
    enumName7 = 3, // value will be 3, legal to reuse numbers
    enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Enumerazione nelle istruzioni switch

Un uso comune per gli enumeratori è per le istruzioni switch e quindi vengono comunemente visualizzati nelle macchine a stati. Infatti, una caratteristica utile delle istruzioni switch con enumerazioni è che se non è inclusa alcuna istruzione predefinita per lo switch e non tutti i valori dell'enumerazione sono stati utilizzati, il compilatore emetterà un avviso.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
       ...
    case middle:
       ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

Iterazione su un enum

Non esiste un built-in per iterare sull'enumerazione.

Ma ci sono diversi modi

- per `enum` con solo valori consecutivi:

  ```
  enum E {
      Begin,
      E1 = Begin,
      E2,
      // ..
      En,
      End
  };

  for (E e = E::Begin; e != E::End; ++e) {
      // Do job with e
  }
  ```

C ++ 11

con la `enum class` , l' `operator ++` deve essere implementato:

```
E& operator ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
    return e;
```

```
}
```

- usando un contenitore come `std::vector`

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*..*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();
```

e poi

```
for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
    E e = *it;
    // Do job with e;
}
```

C ++ 11

- o `std::initializer_list` e una sintassi più semplice:

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*..*/ En};
```

e poi

```
for (auto e : all_E) {
    // Do job with e
}
```

Enumerazioni enunciate

C ++ 11 introduce quelli che sono noti come *enumerati* . Queste sono enumerazioni i cui membri devono essere qualificati con `enumname::membername` . Le enumerazioni scopate vengono dichiarate usando la sintassi della `enum class` . Ad esempio, per memorizzare i colori in un arcobaleno:

```
enum class rainbow {
```

```
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

Per accedere a un colore specifico:

```
rainbow r = rainbow::INDIGO;
```

`enum class` non possono essere convertite implicitamente in `int` s senza un cast. Quindi `int x = rainbow::RED` non è valido.

Le enumerazioni scopate consentono inoltre di specificare il *tipo sottostante* , che è il tipo utilizzato per rappresentare un membro. Di default è `int` . In un gioco Tic-Tac-Toe, puoi memorizzare il pezzo come

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
};
```

Come puoi notare, `enum` s può avere una virgola finale dopo l'ultimo membro.

Dichiarazione anticipata Enum in C ++ 11

Enumerazioni scopate:

```
...
enum class Status; // Forward declaration
Status doWork(); // Use the forward declaration
...
enum class Status { Invalid, Success, Fail };
Status doWork() // Full declaration required for implementation
{
    return Status::Success;
}
```

Enumerazioni senza ambito:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

Un esempio dettagliato di file multipli può essere trovato qui: Esempio di mercante di frutta cieca

Leggi Enumerazione online: https://riptutorial.com/it/cplusplus/topic/2796/enumerazione

Capitolo 34: Errori comuni di compilazione / linker (GCC)

Examples

errore: '***' non è stato dichiarato in questo ambito

Questo errore si verifica se viene utilizzato un oggetto sconosciuto.

variabili

Non compilare:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Difficoltà:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }

    return 0;
}
```

funzioni

> La maggior parte delle volte questo errore si verifica se l'intestazione necessaria non è inclusa (ad esempio, usando `std::cout` senza `#include <iostream>`)

Non compilare:

```
#include <iostream>
```

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

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Difficoltà:

```
#include <iostream>

void doCompile(); // forward declare the function

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

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

O:

```
#include <iostream>

void doCompile() // define the function before using it
{
    std::cout << "No!" << std::endl;
}

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

    return 0;
}
```

Nota: il compilatore interpreta il codice dall'alto verso il basso (semplificazione). Tutto deve essere almeno dichiarato (o definito) prima dell'uso.

riferimento a `*** 'non definito

Questo errore del linker si verifica se il linker non riesce a trovare un simbolo utilizzato. Il più delle volte, ciò accade se una libreria usata non è collegata.

qmake:

```
LIBS += nameOfLib
```

cmake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

chiamata g ++:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

Si potrebbe anche dimenticare di compilare e collegare tutti i file `.cpp` usati (functionsModule.cpp definisce la funzione necessaria):

```
g++  -o binName main.o functionsModule.o
```

errore fatale: ***: nessun file o directory di questo tipo

Il compilatore non riesce a trovare un file (un file sorgente usa `#include "someFile.hpp"`).

qmake:

```
INCLUDEPATH += dir/Of/File
```

cmake:

```
include_directories(dir/Of/File)
```

chiamata g ++:

```
g++ -o main main.cpp -Idir/Of/File
```

Leggi Errori comuni di compilazione / linker (GCC) online:
https://riptutorial.com/it/cplusplus/topic/4256/errori-comuni-di-compilazione---linker--gcc-

Capitolo 35: Esempi di server client

Examples

Ciao TCP Server

Permettetemi di iniziare dicendo che dovreste prima visitare la Guida di Beej alla Programmazione di rete e dargli una lettura veloce, che spiega la maggior parte di questa roba in modo un po 'più verboso. Qui creeremo un semplice server TCP che dirà "Hello World" a tutte le connessioni in arrivo e quindi le chiuderà. Un'altra cosa da notare è che il server comunicherà ai client in modo iterativo, il che significa un client alla volta. Assicurati di controllare le pagine man pertinenti in quanto potrebbero contenere informazioni preziose su ciascuna struttura di chiamata e socket.

Eseguiremo il server con una porta, quindi prenderemo anche un argomento per il numero di porta. Iniziamo con il codice -

```
#include <cstring>    // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h>    // close()


int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
        std::cerr << "Run program as 'program <port>'\n";
        return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8;  // number of connections allowed on the incoming queue


    addrinfo hints, *res, *p;    // we need 2 pointers, res to hold and p to iterate over
    memset(&hints, 0, sizeof(hints));

    // for more explanation, man socket
    hints.ai_family   = AF_UNSPEC;    // don't specify which IP version to use yet
    hints.ai_socktype = SOCK_STREAM;  // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
    hints.ai_flags    = AI_PASSIVE;


    // man getaddrinfo
    int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
```

```
        return -2;
    }

    std::cout << "Detecting addresses" << std::endl;

    unsigned int numOfAddr = 0;
    char ipStr[INET6_ADDRSTRLEN];    // ipv6 length makes sure both ipv4/6 addresses can be
stored in this variable


    // Now since getaddrinfo() has given us a list of addresses
    // we're going to iterate over them and ask user to choose one
    // address for program to bind to
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        std::string ipVer;

        // if address is ipv4 address
        if (p->ai_family == AF_INET) {
            ipVer             = "IPv4";
            sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
            addr              = &(ipv4->sin_addr);
            ++numOfAddr;
        }

        // if address is ipv6 address
        else {
            ipVer              = "IPv6";
            sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
            addr               = &(ipv6->sin6_addr);
            ++numOfAddr;
        }

        // convert IPv4 and IPv6 addresses from binary to text form
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " << ipStr
                  << std::endl;
    }

    // if no addresses found :(
    if (!numOfAddr) {
        std::cerr << "Found no host address to use\n";
        return -3;
    }

    // ask user to choose an address
    std::cout << "Enter the number of host address to bind with: ";
    unsigned int choice = 0;
    bool madeChoice     = false;
    do {
        std::cin >> choice;
        if (choice > (numOfAddr + 1) || choice < 1) {
            madeChoice = false;
            std::cout << "Wrong choice, try again!" << std::endl;
        } else
            madeChoice = true;
    } while (!madeChoice);



    p = res;
```

```
    // let's create a new socket, socketFD is returned as descriptor
    // man socket for more information
    // these calls usually return -1 as result of some error
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        freeaddrinfo(res);
        return -4;
    }


    // Let's bind address to our socket we've just created
    int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
    if (bindR == -1) {
        std::cerr << "Error while binding socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -5;
    }


    // finally start listening for connections on our socket
    int listenR = listen(sockFD, backLog);
    if (listenR == -1) {
        std::cerr << "Error while Listening on socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -6;
    }


    // structure large enough to hold client's address
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);


    const std::string response = "Hello World";


    // a fresh infinite loop to communicate with incoming connections
    // this will take client connections one at a time
    // in further examples, we're going to use fork() call for each client connection
    while (1) {

        // accept call will give us a new socket descriptor
        int newFD
          = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
        if (newFD == -1) {
            std::cerr << "Error while Accepting on socket\n";
            continue;
        }

        // send call sends the data you specify as second param and it's length as 3rd param,
also returns how many bytes were actually sent
        auto bytes_sent = send(newFD, response.data(), response.length(), 0);
        close(newFD);
```

```
    }

    close(sockFD);
    freeaddrinfo(res);

    return 0;
}
```

Il seguente programma funziona come -

```
Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1
```

Ciao client TCP

Questo programma è complementare al programma Hello TCP Server, è possibile eseguirne uno per verificare la validità reciproca. Il flusso del programma è abbastanza comune con Hello TCP server, quindi assicurati di dare un'occhiata anche a questo.

Ecco il codice -

```
#include <cstring>
#include <iostream>
#include <string>

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>


int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }


    auto &ipAddress = argv[1];
    auto &portNum   = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }
```

```
    if (p == NULL) {
        std::cerr << "No addresses found\n";
        return -3;
    }

    // socket() call creates a new socket and returns it's descriptor
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        return -4;
    }

    // Note: there is no bind() call as there was in Hello TCP Server
    // why? well you could call it though it's not necessary
    // because client doesn't necessarily has to have a fixed port number
    // so next call will bind it to a random available port number

    // connect() call tries to establish a TCP connection to the specified server
    int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
    if (connectR == -1) {
        close(sockFD);
        std::cerr << "Error while connecting socket\n";
        return -5;
    }

    std::string reply(15, ' ');

    // recv() call tries to get the response from server
    // BUT there's a catch here, the response might take multiple calls
    // to recv() before it is completely received
    // will be demonstrated in another example to keep this minimal
    auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
    if (bytes_recv == -1) {
        std::cerr << "Error while receiving bytes\n";
        return -6;
    }

    std::cout << "\nClient recieved: " << reply << std::endl;
    close(sockFD);
    freeaddrinfo(p);

    return 0;
}
```

Leggi Esempi di server client online: https://riptutorial.com/it/cplusplus/topic/7177/esempi-di-server-client

Capitolo 36: Espressioni regolari

introduzione

Le espressioni regolari (a volte chiamate regex o espressioni regolari) sono una sintassi testuale che rappresenta i modelli che possono essere confrontati nelle stringhe operate.

Le espressioni regolari, introdotte in C ++ 11 , possono opzionalmente supportare un array di restituzione di stringhe corrispondenti o un'altra sintassi testuale che definisce come sostituire schemi abbinati nelle stringhe su cui si opera.

Sintassi

- regex_match // Restituisce se l'intera sequenza di caratteri è stata abbinata dalla regex, eventualmente acquisendo in un oggetto match
- regex_search // Indica se una parte della sequenza di caratteri è stata abbinata alla regex, eventualmente acquisendo in un oggetto match
- regex_replace // Restituisce la sequenza di caratteri di input modificata da un'espressione regolare tramite una stringa di formato di sostituzione
- regex_token_iterator // Inizializzato con una sequenza di caratteri definita da iteratori, una lista di indici di cattura da iterare e una regex. Dereferencing restituisce la corrispondenza indicizzata al momento della regex. Incrementando le mosse al successivo indice di cattura o se attualmente nell'ultimo indice, reimposta l'indice e incide l'occorrenza successiva di una corrispondenza regolare nell'esecuzione dei caratteri
- regex_iterator // Inizializzato con una sequenza di caratteri definita da iteratori e una regex. Il dereferenziamento restituisce la parte della sequenza di caratteri che corrisponde attualmente all'intera espressione regolare. Incrementando trova la prossima occorrenza di una corrispondenza regolare nell'esecuzione dei caratteri

Parametri

Firma	Descrizione
`bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)`	**`BidirectionalIterator`** è qualsiasi iteratore di caratteri che fornisce operatori di incremento e decremento **`smatch`** può essere `cmatch` o qualsiasi altra variante di `match_results` che accetta il tipo di `BidirectionalIterator` l'argomento `smatch` può essere omesso se i risultati della regex non sono necessari. **Restituisce** se `re` combacia con l'intero carattere sequenza definita dal `first` e `last`
`bool regex_match(const string& str, smatch& sm, const regex re&,`	**`string`** può essere un `const char*` o una `string`

Firma	Descrizione
`regex_constraints::match_flag_type flags)`	L-Value, *le funzioni che accettano una* `string` *R-Value sono eliminate esplicitamente* **`smatch`** può essere `cmatch` o qualsiasi altra variante di `match_results` che accetta il tipo di `str` l'argomento `smatch` può essere omesso se i risultati del l'espressione regolare non sono necessari **Indica se** `re` corrisponde l'intera sequenza di caratteri definito da `str`

Examples

Esempi di regex_match e regex_search di base

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use
regular expressions.\""s;
smatch sm;

cout << input << endl;

// If input ends in a quotation that contains a word that begins with "reg" and another word
begining with "ex" then capture the preceeding portion of input
if (regex_match(input, sm, regex("(.*)\".*\\breg.*\\bex.*\"\\s*$"))) {
    const auto capture = sm[1].str();

    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a
problem, think\n"

    // Search our capture for "a problem" or "# problems"
    if(regex_search(capture, sm, regex("(a|d+)\\s+problems?"))) {
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);

        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1
problem\n"
        cout << "Now they have " << count + 1 << " problems.\n"; // Ouputs: "Now they have 2
problems\n"
    }
}
```

Esempio dal vivo

regex_replace Esempio

Questo codice contiene vari stili di controvento e li converte in uno stile True True :

```
const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n    foo();\n}\nif
(allman)\n{\n\tfoo();\n}\nif (horstmann)\n{\tfoo();\n}\nif (pico)\n{\tfoo(); }\nif
(whitesmiths)\n\t{\n\tfoo();\n\t}\n"s;

cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+?;)\\s*\\}?\\s*"), "$1
{\n\t$2\n}\n") << endl;
```

Esempio dal vivo

regex_token_iterator Esempio

Un `std::regex_token_iterator` fornisce un formidabile strumento per estrarre elementi da un file di valori separati da virgola . Oltre ai vantaggi dell'iterazione, questo iteratore è anche in grado di catturare virgole sfuggite dove altri metodi combattono:

```
const auto input = "please split,this,csv, ,line,\\,\n"s;
const regex re{ "((?:[^\\\\,]|\\\\.)+)(?:,|$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\n"));
```

Esempio dal vivo

Un clamore notevole con gli iteratori di espressioni regolari è che l'argomento di `regex` regolare deve essere un valore L. Un valore R non funzionerà .

regex_iterator Esempio

Quando l'elaborazione delle acquisizioni deve essere eseguita in modo iterativo, `regex_iterator` è una buona scelta. `regex_iterator` un `regex_iterator` restituisce un `match_result` . Questo è ottimo per le catture condizionali o catture che hanno interdipendenza. Diciamo che vogliamo tokenize del codice C ++. Dato:

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

Possiamo tokenize questa stringa: `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s` con un `regex_iterator` come questo:

```
vector<TOKENS> tokens;
const regex re{ "\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\)?)\\s*(?:(\\+)|(-)|(\\*)|(/)|(=))" };

for_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto&
i) {
    if(i[1].length() > 0) {
        tokens.push_back(OPEN_PARENTHESIS);
    }

    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);
```

```
    if(i[3].length() > 0) {
        tokens.push_back(CLOSE_PARENTHESIS);
    }

    auto it = next(cbegin(i), 4);

    for(int result = ADDITION; it != cend(i); ++result, ++it) {
        if (it->length() > 0U) {
            tokens.push_back(static_cast<TOKENS>(result));
            break;
        }
    }
});

match_results<string::const_reverse_iterator> sm;

if(regex_search(crbegin(input), crend(input), sm, regex{ tokens.back() == SUBTRACTION ?
"^\\s*\\d+\\s*-\\s*(-?)" : "^\\s*\\d+\\s*(-?)" })) {
    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);
}
```

Esempio dal vivo

Un notcha notabile con gli iteratori regex è che l'argomento `regex` regolare deve essere un valore L, un valore R non funzionerà: Visual Studio regex_iterator Bug?

Divisione di una corda

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}
```

```
split("Some  string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

quantificatori

Diciamo che ci viene data una `const string input` come numero di telefono per essere convalidato. Potremmo iniziare richiedendo un input numerico con **zero o più quantificatori** : `regex_match(input, regex("\\d*"))` o **uno o più quantificatori** : `regex_match(input, regex("\\d+"))` Ma entrambi sono davvero in difetto se l' `input` contiene una stringa numerica non valida come: "123" Usiamo un **n o più quantificatori** per assicurarci di ottenere almeno 7 cifre:

```
regex_match(input, regex("\\d{7,}"))
```

Ciò garantirà che otterremo almeno un numero di telefono di cifre, ma l' `input` potrebbe contenere anche una stringa numerica troppo lunga come "123456789012". Quindi lascia andare con un **quantificatore tra n e m in** modo che l' `input` sia di almeno 7 cifre ma non più di 11:

```
regex_match(input, regex("\\d{7,11}"));
```

Questo ci avvicina, ma le stringhe numeriche illegali che sono nell'intervallo [7, 11] sono ancora accettate, come: "123456789" Quindi rendiamo opzionale il codice paese con un **quantificatore pigro** :

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

È importante notare che il **quantificatore pigro** corrisponde al *minor numero possibile di caratteri* , quindi l'unico modo in cui questo carattere verrà confrontato è se ci sono già 10 caratteri che sono stati abbinati da `\d{7,10}` . (Per abbinare avidamente il primo personaggio avremmo dovuto fare: `\d{0,1}` .) Il **quantificatore pigro** può essere aggiunto a qualsiasi altro quantificatore.

Ora, come potremmo rendere facoltativo il prefisso *e* accettare solo un prefisso internazionale se fosse presente il prefisso?

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

In questa regex finale, il `\d{7}` *richiede* 7 cifre. Queste 7 cifre sono opzionalmente precedute da 3 o 4 cifre.

Si noti che non abbiamo aggiunto il **quantificatore pigro** : ~~`\d{3,4}?\d{7}`~~ , il `\d{3,4}?` avrebbe dovuto corrispondere a 3 o 4 caratteri, preferendo 3. Invece faremo la partita del gruppo non catturante al massimo una volta, preferendo non corrispondere. Causando una mancata corrispondenza se l' `input` non includeva il prefisso come: "1234567".

In conclusione dell'argomento quantificatore, vorrei menzionare l'altro quantificatore che puoi usare, il **quantificatore possessivo** . *O* il **quantificatore pigro** o **quantificatore possessivo** può essere aggiunto a qualsiasi quantificatore. L'unica funzione del **quantificatore possessivo** è quella di assistere il motore regex dicendolo, prendendo avidamente questi caratteri *e non li abbandonare mai, anche se causa il regex fallire* . Questo per esempio non ha molto senso: `regex_match(input, regex("\\d{3,4}+\\d{7}))` Perché un `input` come:" 1234567890 "non verrebbe abbinato come `\d{3,4}+` corrisponderà sempre a 4 caratteri, anche se la corrispondenza 3 avrebbe permesso al regex di avere successo.
Il **quantificatore possessivo** viene utilizzato al meglio *quando il token quantificato limita il numero di caratteri accoppiabili* . Per esempio:

```
regex_match(input, regex("(?:.*\\d{3,4}+){3}"))
```

Può essere utilizzato per corrispondere se l' `input` contiene uno dei seguenti elementi:

123 456 7890
123-456-7890
(123)456-7890
(123) 456 - 7890

Ma quando questa regex brilla davvero è quando l' `input` contiene un input *illegale* :

12345 - 67890

Senza il **quantificatore possessivo** il motore regex deve tornare indietro e testare *ogni combinazione di* `.*` *e di 3 o 4 caratteri* per vedere se può trovare una combinazione abbinabile. Con il **quantificatore possessivo** le regex inizia dove il **quantificatore** 2 ° **possessivo** lasciato, il carattere '0', e il motore di regex cerca di regolare il `.*` Per consentire `\d{3,4}` per abbinare; quando non può la regex appena fallisce, nessun monitoraggio posteriore è fatto per vedere se precedente `.*` Regolazione avrebbe potuto consentire una corrispondenza.

ancore

C ++ fornisce solo 4 ancore:

- `^` che asserisce l'inizio della stringa
- `$` che asserisce la fine della stringa
- `\b` che asserisce un carattere `\W` o l'inizio o la fine della stringa
- `\B` che asserisce un carattere `\w`

Diciamo per esempio che vogliamo catturare un numero *con il* suo segno:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {

    do {
        cout << sm[1] << endl;
        input = sm.suffix().str();
    } while(regex_search(input, sm, regex{ "(?:^\\W|\\b\\W)([+-]?\\d+)" }));
}
```

Esempio dal vivo

Una nota importante qui è che l'ancora non consuma alcun personaggio.

Leggi Espressioni regolari online: https://riptutorial.com/it/cplusplus/topic/1681/espressioni-regolari

Capitolo 37: File di intestazione

Osservazioni

In C ++, come in C, il compilatore C ++ e il processo di compilazione fanno uso del preprocessore C. Come specificato dal manuale del preprocessore GNU C, un file di intestazione è definito come segue:

> Un file di intestazione è un file contenente dichiarazioni C e definizioni di macro (vedi Macro) da condividere tra diversi file di origine. Si richiede l'uso di un file di intestazione nel programma includendolo, con la direttiva di pre-elaborazione C '#include'.
>
> I file di intestazione hanno due scopi.
>
> - I file di intestazione di sistema dichiarano le interfacce a parti del sistema operativo. Li includi nel tuo programma per fornire le definizioni e le dichiarazioni necessarie per richiamare le chiamate e le librerie di sistema.
> - I tuoi file di intestazione contengono dichiarazioni per le interfacce tra i file di origine del tuo programma. Ogni volta che si dispone di un gruppo di dichiarazioni correlate e definizioni macro tutte o la maggior parte delle quali sono necessarie in diversi file sorgente, è una buona idea creare un file di intestazione per loro.

Tuttavia, per il preprocessore stesso, un file di intestazione non è diverso da un file sorgente.

Lo schema di organizzazione del file di intestazione / sorgente è semplicemente una convenzione consolidata e standard impostata da vari progetti software al fine di fornire una separazione tra interfaccia e implementazione.

Sebbene non sia formalmente applicato dallo stesso standard C ++, è altamente raccomandato seguire la convenzione di file header / sorgente e, in pratica, è già quasi onnipresente.

Si noti che i file di intestazione possono essere sostituiti come una convenzione della struttura del file di progetto dalla funzione imminente dei moduli, che è ancora da considerare per l'inclusione in un futuro standard C ++ al momento della scrittura (ad esempio C ++ 20).

Examples

Esempio di base

L'esempio seguente conterrà un blocco di codice che deve essere suddiviso in diversi file di origine, come indicato da `// filename` commenti del `// filename` .

File sorgenti

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
  return global_value; // return 42;
}
```

I file di intestazione vengono quindi inclusi da altri file di origine che desiderano utilizzare la funzionalità definita dall'interfaccia dell'intestazione, ma non richiedono la conoscenza della sua implementazione (riducendo quindi l'accoppiamento del codice). Il seguente programma utilizza l'intestazione `my_function.h` come sopra definito:

```
// main.cpp

#include <iostream>       // A C++ Standard Library header.
#include "my_function.h"  // A personal header

int main(int argc, char** argv) {
  std::cout << my_function() << std::endl;
  return 0;
}
```

Il processo di compilazione

Poiché i file di intestazione sono spesso parte di un flusso di lavoro del processo di compilazione, un tipico processo di compilazione che utilizza la convenzione di file di intestazione / origine di solito eseguirà quanto segue.

Supponendo che il file di intestazione e il codice sorgente siano già nella stessa directory, un programmatore eseguirà i seguenti comandi:

```
g++ -c my_function.cpp       # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o   # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

In alternativa, se si desidera compilare `main.cpp` in un file oggetto per primo e quindi collegare insieme solo i file oggetto come passo finale:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Modelli nei file di intestazione

I modelli richiedono la generazione del codice in fase di compilazione: una funzione basata su modelli, ad esempio, verrà trasformata in più funzioni distinte una volta parametrizzata una funzione basata su un modello mediante l'uso nel codice sorgente.

Ciò significa che la funzione template, la funzione membro e le definizioni di classe non possono essere delegate a un file di codice sorgente separato, poiché qualsiasi codice che utilizzerà un costrutto basato su modelli richiede la conoscenza della sua definizione per generare in genere un codice derivato.

Pertanto, il codice basato su modelli, se inserito nelle intestazioni, deve contenere anche la sua definizione. Un esempio di questo è qui sotto:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
  T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                        // for C++11 or later
  return type_point;
}
```

Leggi File di intestazione online: https://riptutorial.com/it/cplusplus/topic/7211/file-di-intestazione

Capitolo 38: File I / O

introduzione

L'I / O del file C ++ viene eseguito tramite *flussi* . Le astrazioni chiave sono:

`std::istream` per leggere il testo.

`std::ostream` per scrivere testo.

`std::streambuf` per leggere o scrivere caratteri.

L'input formattato utilizza l' `operator>>` .

L'output formattato utilizza l' `operator<<` .

Gli stream usano `std::locale` , ad es. Per i dettagli sulla formattazione e per la traduzione tra codifiche esterne e codifica interna.

Altro sugli stream: <iostream> Library

Examples

Aprire un file

L'apertura di un file viene eseguita allo stesso modo per tutti e 3 i flussi di file (`ifstream` , `ofstream` e `fstream`).

Puoi aprire il file direttamente nel costruttore:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.

std::fstream iofs("foo.txt");  // fstream:  Opens file "foo.txt" for reading and writing.
```

In alternativa, è possibile utilizzare la funzione membro `open()` del flusso di file `open()` :

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream:  Opens file "bar.txt" for reading and writing.
```

Si dovrebbe **sempre** verificare se un file è stato aperto correttamente (anche durante la scrittura). Gli errori possono includere: il file non esiste, il file non ha i diritti di accesso corretti, il file è già in

uso, si sono verificati errori del disco, unità disconnessa ... Il controllo può essere eseguito come segue:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt");  // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

Quando il percorso del file contiene barre retroverse (ad esempio, su sistema Windows) dovresti eseguire correttamente l'escape:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:\\folder\\foo.txt"); // using escaped backslashes
```

C ++ 11

o usa letterale crudo:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs(R"(c:\folder\foo.txt)"); // using raw literal
```

o usare invece le barre in avanti:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

C ++ 11

Se si desidera aprire il file con caratteri non ASCII nel percorso su Windows al momento, è possibile utilizzare un argomento del percorso dei caratteri ampio **non standard** :

```
// Open the file 'пример\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\foo.txt)"); // using wide characters with raw literal
```

Lettura da un file

Esistono diversi modi per leggere i dati da un file.

Se si conosce come vengono formattati i dati, è possibile utilizzare l'operatore di estrazione del flusso (`>>`). Supponiamo che tu abbia un file chiamato *foo.txt* che contiene i seguenti dati:

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

Quindi è possibile utilizzare il seguente codice per leggere i dati dal file:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

L'operatore di estrazione del flusso `>>` estrae ogni carattere e si arresta se trova un carattere che non può essere memorizzato o se è un carattere speciale:

- Per i tipi di stringa, l'operatore si ferma in uno spazio bianco ( ) o su una nuova riga (`\n`).
- Per i numeri, l'operatore si ferma con un carattere non numerico.

Ciò significa che anche la seguente versione del file *foo.txt* verrà letta con successo dal codice precedente:

```
John
Doe 25
4 6 1987


Jane
Doe
15 5
24
1976
```

L'operatore di estrazione del flusso `>>` restituisce sempre lo stream a esso assegnato. Pertanto, più operatori possono essere concatenati insieme per leggere i dati consecutivamente. Tuttavia, uno stream può anche essere usato come espressione booleana (come mostrato nel ciclo `while` nel codice precedente). Questo perché le classi di flusso hanno un operatore di conversione per il tipo `bool` . Questo operatore `bool()` restituirà `true` fintanto che lo stream non ha errori. Se un flusso entra in uno stato di errore (ad esempio, poiché non è possibile estrarre altri dati), l'operatore `bool()` restituirà `false` . Pertanto, il ciclo `while` nel codice precedente verrà chiuso dopo che il file di input è stato letto fino alla fine.

Se desideri leggere un intero file come una stringa, puoi usare il seguente codice:

```
// Opens 'foo.txt'.
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());

// Sets position to the start of the file.
```

```
is.seekg(0, std::ios::beg);

// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
  std::istreambuf_iterator<char>());
```

Questo codice riserva spazio per la `string` al fine di ridurre le allocazioni di memoria non necessarie.

Se si desidera leggere un file riga per riga, è possibile utilizzare la funzione `getline()` :

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

Se vuoi leggere un numero fisso di caratteri, puoi usare la funzione membro del flusso `read()` :

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

Dopo aver eseguito un comando di lettura, è necessario verificare sempre se è stato impostato il flag di stato di errore `failbit` , in quanto indica se l'operazione ha avuto esito negativo o meno. Questo può essere fatto chiamando la funzione membro `fail()` del file stream `fail()` :

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

Scrivere su un file

Esistono diversi modi per scrivere in un file. Il modo più semplice è utilizzare un flusso di file di output (`ofstream`) insieme all'operatore di inserimento del flusso (`<<`):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Invece di `<<` , è anche possibile utilizzare la funzione membro `write()` del flusso del file di output `write()` :

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";
```

```
    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

Dopo aver scritto su uno stream, è necessario controllare sempre se è stato impostato lo stato di errore flag `badbit` , poiché indica se l'operazione ha avuto esito negativo o meno. Questo può essere fatto chiamando la funzione membro del flusso del file di output `bad()` :

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Modalità di apertura

Quando si crea un flusso di file, è possibile specificare una modalità di apertura. Una modalità di apertura è fondamentalmente un'impostazione per controllare come il flusso apre il file.

(Tutte le modalità possono essere trovate nel namespace `std::ios` .)

Una modalità di apertura può essere fornita come secondo parametro per il costruttore di un flusso di file o per la sua funzione membro `open()` :

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

Si noti che è necessario impostare `ios::in` o `ios::out` se si desidera impostare altri flag in quanto non sono impostati in modo implicito dai membri iostream sebbene abbiano un valore predefinito corretto.

Se non si specifica una modalità di apertura, vengono utilizzate le seguenti modalità predefinite:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` e `out`

Le modalità di apertura dei file che puoi specificare in base alla progettazione sono:

Modalità	Senso	Per	Descrizione
`app`	aggiungere	Produzione	Aggiunge i dati alla fine del file.
`binary`	binario	Input Output	L'input e l'output sono fatti in binario.
`in`	ingresso	Ingresso	Apre il file per la lettura.
`out`	produzione	Produzione	Apre il file per la scrittura.
`trunc`	troncare	Input Output	Rimuove il contenuto del file all'apertura.

Modalità	Senso	Per	Descrizione
`ate`	alla fine	Ingresso	Va alla fine del file quando si apre.

Nota: l' impostazione della modalità `binary` consente di leggere / scrivere i dati esattamente così come sono; non impostandolo consente la traduzione del carattere `'\n'` nuova riga in / da una sequenza di fine riga specifica della piattaforma.

Ad esempio su Windows la sequenza di fine riga è CRLF (`"\r\n"`).
Scrivi: `"\n"` => `"\r\n"`
Leggi: `"\r\n"` => `"\n"`

Chiusura di un file

Raramente chiudere un file è raramente necessario in C ++, in quanto un flusso di file chiuderà automaticamente il suo file associato nel suo distruttore. Tuttavia, si dovrebbe cercare di limitare la durata di un oggetto flusso di file, in modo che non mantenga l'handle del file aperto più a lungo del necessario. Ad esempio, ciò può essere fatto inserendo tutte le operazioni sui file in un proprio ambito (`{}`):

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
}  // The ofstream will go out of scope here.
   // Its destructor will take care of closing the file properly.
```

Chiamare `close()` esplicitamente è necessario solo se si desidera riutilizzare lo stesso oggetto `fstream` secondo momento, ma non si vuole mantenere il file aperto tra:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();

// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();

// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;
```

```
// Close the file "foo.txt" once again.
output.close();
```

Flushing a stream

I flussi di file sono bufferizzati per impostazione predefinita, come lo sono molti altri tipi di flussi. Ciò significa che le scritture sul flusso potrebbero non modificare immediatamente il file sottostante. In oder per forzare immediatamente tutte le scritture bufferizzate, è possibile *svuotare* il flusso. Puoi farlo direttamente invocando il metodo `flush()` o tramite il manipolatore di flusso `std::flush` :

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

C'è un manipolatore di flusso `std::endl` che combina la scrittura di una nuova riga con lo streaming del flusso:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

Il buffering può migliorare le prestazioni della scrittura su uno stream. Pertanto, le applicazioni che richiedono molta scrittura dovrebbero evitare il risciacquo inutilmente. Contrariamente, se l'I / O viene eseguito di rado, le applicazioni dovrebbero prendere in considerazione il flushing frequentemente per evitare che i dati si blocchino nell'oggetto stream.

Lettura di un file ASCII in una stringa std ::

```
std::ifstream f("file.txt");

if (f)
{
  std::stringstream buffer;
  buffer << f.rdbuf();
  f.close();

  // The content of "file.txt" is available in the string `buffer.str()`
}
```

Il metodo `rdbuf()` restituisce un puntatore a uno `streambuf` che può essere inserito nel `buffer` tramite la funzione membro `stringstream::operator<<` .

Un'altra possibilità (resa popolare da Effective STL di Scott Meyers) è:

```
std::ifstream f("file.txt");
```

```
if (f)
{
  std::string str((std::istreambuf_iterator<char>(f)),
                  std::istreambuf_iterator<char>());

  // Operations on `str`...
}
```

Questo è bello perché richiede poco codice (e consente di leggere un file direttamente in qualsiasi contenitore STL, non solo stringhe), ma può essere lento per i file di grandi dimensioni.

NOTA : le parentesi extra attorno al primo argomento del costruttore di stringhe sono essenziali per prevenire il problema di *analisi più irritante* .

Ultimo, ma non per importanza:

```
std::ifstream f("file.txt");

if (f)
{
  f.seekg(0, std::ios::end);
  const auto size = f.tellg();

  std::string str(size, ' ');
  f.seekg(0);
  f.read(&str[0], size);
  f.close();

  // Operations on `str`...
}
```

che è probabilmente l'opzione più veloce (tra le tre proposte).

Lettura di un file in un contenitore

Nell'esempio seguente usiamo `std::string` e `operator>>` per leggere gli elementi dal file.

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v;

    std::string s;
    while(file >> s) // keep reading until we run out
    {
        v.push_back(s);
    }
```

Nell'esempio sopra stiamo semplicemente iterando attraverso il file leggendo un "oggetto" alla volta usando l' `operator>>` . Questo stesso effetto può essere ottenuto usando lo `std::istream_iterator` che è un iteratore di input che legge un "elemento" alla volta dallo stream. Inoltre, la maggior parte dei contenitori può essere costruita utilizzando due iteratori in modo da semplificare il codice sopra riportato per:

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v(std::istream_iterator<std::string>{file},
                                std::istream_iterator<std::string>{});
```

Possiamo estendere questo per leggere qualsiasi tipo di oggetto che ci piace semplicemente specificando l'oggetto che vogliamo leggere come parametro template per lo `std::istream_iterator`. Quindi possiamo semplicemente estendere quanto sopra per leggere le righe (piuttosto che le parole) in questo modo:

```
// Unfortunately there is  no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};


    std::ifstream file("file3.txt");

    // Read the lines of a file into a container.
    std::vector<std::string>  v(std::istream_iterator<Line>{file},
                                std::istream_iterator<Line>{});
```

Leggere una `struct` da un file di testo formattato.

C ++ 11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
```

```
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "   age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

file4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Produzione:

```
name: Wogger Wabbit
 age: 2 years
height: 6.2lbs

name: Bilbo Baggins
 age: 111 years
height: 81.3lbs

name: Mary Poppins
 age: 29 years
height: 154.8lbs
```

Copia di un file

```
std::ifstream  src("source_filename", std::ios::binary);
std::ofstream  dst("dest_filename",   std::ios::binary);
dst << src.rdbuf();
```

C ++ 17

Con C ++ 17 il modo standard per copiare un file include l'intestazione `<filesystem>` e l'uso di `copy_file` :

```
std::fileystem::copy_file("source_filename", "dest_filename");
```

La libreria del filesystem era originariamente sviluppata come `boost.filesystem` e infine fusa con ISO C ++ a partire da C ++ 17.

Controllare la fine del file all'interno di una condizione di loop, cattiva pratica?

`eof` restituisce `true` solo **dopo aver** letto la fine del file. NON indica che la prossima lettura sarà la fine del flusso.

```
while (!f.eof())
{
  // Everything is OK

  f >> buffer;

  // What if *only* now the eof / fail bit is set?

  /* Use `buffer` */
}
```

Potresti scrivere correttamente:

```
while (!f.eof())
{
  f >> buffer >> std::ws;

  if (f.fail())
    break;

  /* Use `buffer` */
}
```

ma

```
while (f >> buffer)
{
  /* Use `buffer` */
}
```

è più semplice e meno soggetto a errori.

Ulteriori riferimenti:

- `std::ws` : elimina gli spazi bianchi iniziali da un flusso di input
- `std::basic_ios::fail` : restituisce `true` se si è verificato un errore nel flusso associato

Scrittura di file con impostazioni internazionali non standard

Se è necessario scrivere un file utilizzando impostazioni internazionali diverse per l'impostazione predefinita, è possibile utilizzare `std::locale` e `std::basic_ios::imbue()` per farlo per un flusso di file specifico:

Guida per l'uso:

- Devi sempre applicare un locale a uno stream prima di aprire il file.
- Una volta che il flusso è stato imbevuto non è necessario modificare le impostazioni internazionali.

Motivi per le restrizioni: l' importazione di un flusso di file con una locale ha un comportamento non definito se le impostazioni internazionali correnti non sono indipendenti dallo stato o non puntano all'inizio del file.

I flussi UTF-8 (e altri) non sono indipendenti dallo stato. Anche un flusso di file con una locale UTF-8 può provare a leggere l'indicatore BOM dal file quando viene aperto; quindi basta aprire il file per leggere i caratteri dal file e non sarà all'inizio.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
  std::cout << "User-preferred locale setting is "
            << std::locale("").name().c_str() << std::endl;

  // Write a floating-point value using the user's preferred locale.
  std::ofstream ofs1;
  ofs1.imbue(std::locale(""));
  ofs1.open("file1.txt");
  ofs1 << 78123.456 << std::endl;

  // Use a specific locale (names are system-dependent)
  std::ofstream ofs2;
  ofs2.imbue(std::locale("en_US.UTF-8"));
  ofs2.open("file2.txt");
  ofs2 << 78123.456 << std::endl;

  // Switch to the classic "C" locale
  std::ofstream ofs3;
  ofs3.imbue(std::locale::classic());
  ofs3.open("file3.txt");
  ofs3 << 78123.456 << std::endl;
}
```

Passare in modo esplicito alla classica "C" locale è utile se il tuo programma usa una diversa localizzazione predefinita e vuoi garantire uno standard fisso per leggere e scrivere file. Con una locale preferita "C", scrive l'esempio

```
78,123.456
78,123.456
78123.456
```

Se, ad esempio, la locale preferita è il tedesco e quindi utilizza un formato numerico diverso, l'esempio scrive

```
78 123,456
```

```
78,123.456
78123.456
```

(notare la virgola decimale nella prima riga).

Leggi File I / O online: https://riptutorial.com/it/cplusplus/topic/496/file-i---o

Capitolo 39: Funzione C ++ "call by value" vs. "call by reference"

introduzione

Lo scopo di questa sezione è di spiegare le differenze di teoria e implementazione per ciò che accade con i parametri di una funzione al momento della chiamata.

Nel dettaglio i parametri possono essere visti come variabili prima della chiamata di funzione e all'interno della funzione, dove il comportamento visibile e l'accessibilità a queste variabili differiscono con il metodo utilizzato per passarle sopra.

Inoltre, la riusabilità delle variabili e dei rispettivi valori dopo la chiamata alla funzione è spiegata da questo argomento.

Examples

Chiama per valore

Dopo aver chiamato una funzione, ci sono nuovi elementi creati nello stack del programma. Questi includono alcune informazioni sulla funzione e anche lo spazio (posizioni di memoria) per i parametri e il valore di ritorno.

Quando si consegna un parametro a una funzione, il valore della variabile utilizzata (o letterale) viene copiato nella posizione di memoria del parametro della funzione. Ciò implica che ora ci sono due posizioni di memoria con lo stesso valore. All'interno della funzione lavoriamo solo sulla posizione della memoria dei parametri.

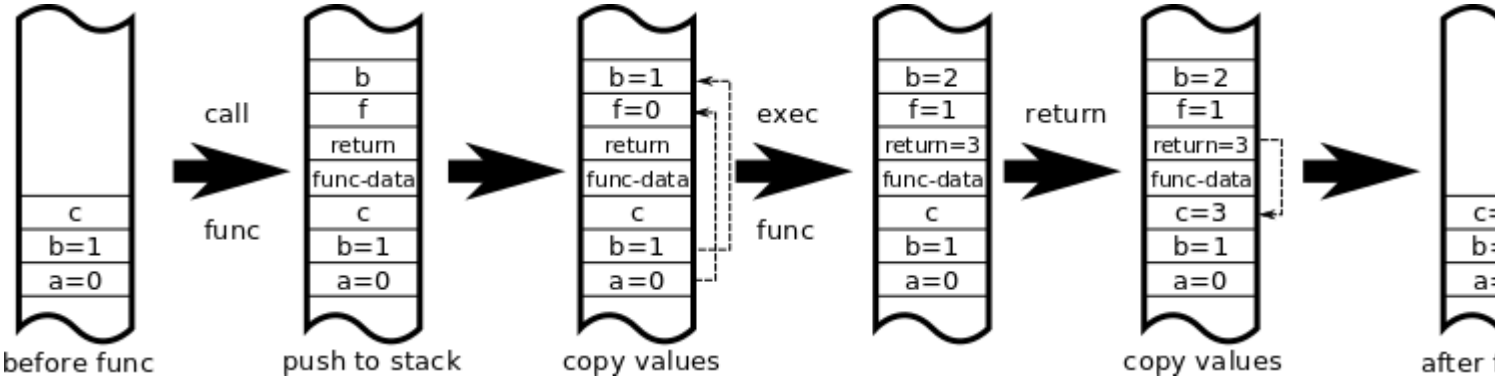
Dopo aver lasciato la funzione, la memoria sullo stack dei programmi viene spuntata (rimossa) che cancella tutti i dati della chiamata di funzione, inclusa la posizione di memoria dei parametri utilizzati all'interno. Pertanto, i valori modificati all'interno della funzione non influiscono sui valori delle variabili esterne.

```
int func(int f, int b) {
  //new variables are created and values from the outside copied
  //f has a value of 0
  //inner_b has a value of 1
  f = 1;
  //f has a value of 1
  b = 2;
  //inner_b has a value of 2
  return f+b;
}

int main(void) {
  int a = 0;
  int b = 1; //outer_b
```

```
  int c;

  c = func(a,b);
  //the return value is copied to c

  //a has a value of 0
  //outer_b has a value of 1   <--- outer_b and inner_b are different variables
  //c has a value of 3
}
```

In questo codice creiamo le variabili all'interno della funzione principale. Questi vengono assegnati valori. Dopo aver chiamato le funzioni, vengono create due nuove variabili: `f` e `inner_b` dove `b` condivide il nome con la variabile esterna e non condivide la posizione di memoria. Il comportamento di `a<->f` e `b<->b` è identico.

Il seguente grafico simboleggia cosa sta accadendo nello stack e perché non ci sono cambiamenti in varibale `b` . Il grafico non è completamente preciso ma enfatizza l'esempio.



Si chiama "call by value" perché non si consegnano le variabili ma solo i valori di queste variabili.

Leggi Funzione C ++ "call by value" vs. "call by reference" online: https://riptutorial.com/it/cplusplus/topic/10669/funzione-c-plusplus--call-by-value--vs---call-by-reference-

Capitolo 40: Funzione di sovraccarico

introduzione

Vedi anche argomento separato su Risoluzione overload

Osservazioni

Le ambiguità possono verificarsi quando un tipo può essere convertito implicitamente in più di un tipo e non esiste una funzione di corrispondenza per quel tipo specifico.

Per esempio:

```
void foo(double, double);
void foo(long, long);

//Call foo with 2 ints
foo(1, 2); //Function call is ambiguous - int can be converted into a double/long at the same
time
```

Examples

Cos'è il sovraccarico di funzione?

L'overloading delle funzioni sta avendo più funzioni dichiarate nello stesso ambito con lo stesso identico nome nello stesso punto (noto come *scope*) che differisce solo nella loro *firma* , vale a dire gli argomenti che accettano.

Supponiamo che stiate scrivendo una serie di funzioni per le capacità di stampa generalizzate, iniziando con `std::string` :

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

Funziona bene, ma supponiamo che tu voglia una funzione che accetta anche un `int` e stampa anche quello. Potresti scrivere:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

Ma poiché le due funzioni accettano parametri diversi, puoi semplicemente scrivere:

```
void print(int num)
```

```
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Ora avete 2 funzioni, entrambe denominate `print` , ma con diverse firme. Uno accetta `std::string` , l'altro un `int` . Ora puoi chiamarli senza preoccuparti di nomi diversi:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Invece di:

```
print("Hello world!");
print_int(1337);
```

Quando hai sovraccaricato le funzioni, il compilatore deduce quale delle funzioni chiamare dai parametri che le hai forniti. Bisogna fare attenzione quando si scrivono sovraccarichi di funzione. Ad esempio, con conversioni di tipo implicite:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Ora non è immediatamente chiaro quale sovraccarico di `print` viene chiamato quando scrivi:

```
print(5);
```

E potrebbe essere necessario fornire al compilatore alcuni indizi, come:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

È necessario prestare attenzione quando si scrivono sovraccarichi che accettano parametri facoltativi:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Perché non c'è modo per il compilatore di dire se una chiamata come `print(17)` è pensata per la prima o la seconda funzione a causa del secondo parametro opzionale, questo non riuscirà a compilare.

Tipo di ritorno Sovraccarico funzione

Si noti che non è possibile sovraccaricare una funzione in base al suo tipo di ritorno. Per esempio:

```
// WRONG CODE
std::string getValue()
{
  return "hello";
}

int getValue()
{
  return 0;
}

int x = getValue();
```

Ciò causerà un errore di compilazione in quanto il compilatore non sarà in grado di determinare quale versione di `getValue` chiamare, anche se il tipo restituito è assegnato a un `int` .

Funzione membro qualifica cv Sovraccarico

Le funzioni all'interno di una classe possono essere sovraccaricate per quando sono accessibili tramite un riferimento qualificato cv a quella classe; questo è più comunemente usato per il sovraccarico di `const` , ma può essere usato anche per sovraccaricare `volatile` e `const volatile` . Questo perché tutte le funzioni membro non statiche prendono `this` come parametro nascosto, al quale vengono applicati i qualificatori cv. Questo è più comunemente usato per sovraccaricare `const` , ma può anche essere usato per `volatile` e `const volatile` .

Questo è necessario perché una funzione membro può essere chiamata solo se è almeno classificata come cv come l'istanza su cui è chiamata. Mentre un non `const` istanza può chiamare sia `const` e non `const` membri, un `const` istanza può chiamare solo `const` membri. Ciò consente a una funzione di avere un comportamento diverso a seconda dei cv-qualificatori dell'istanza chiamante e consente al programmatore di disabilitare le funzioni per un qualificatore cv indesiderato non fornendo una versione con quel qualificatore (i).

Una classe con un metodo di `print` base potrebbe essere sovraccaricata `const` modo:

```
#include <iostream>

class Integer
{
    public:
        Integer(int i_): i{i_}{}

        void print()
        {
            std::cout << "int: " << i << std::endl;
```

```
        }

        void print() const
        {
            std::cout << "const int: " << i << std::endl;
        }

    protected:
        int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

Questo è un principio chiave della correttezza `const` : contrassegnando le funzioni membro come `const` , è possibile che vengano chiamate su istanze `const` , che a loro volta consentono alle funzioni di assumere istanze come riferimenti / riferimenti `const` se non è necessario modificarle. Ciò consente al codice di specificare se modifica lo stato prendendo parametri non modificati come parametri `const` e modificati senza qualificatori di cv, rendendo il codice più sicuro e più leggibile.

```
class ConstCorrect
{
  public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." <<
std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Error.  Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Good.  Can only be called from non-const instance.
}
```

Un uso comune di questo è dichiarare accessors come `const` , e mutatori come non- `const` .

Nessun membro della classe può essere modificato all'interno di una funzione membro `const` . Se c'è un membro che devi veramente modificare, come il blocco di `std::mutex` , puoi dichiararlo come `mutable` :

```
class Integer
{
    public:
        Integer(int i_): i{i_}{}

        int get() const
        {
            std::lock_guard<std::mutex> lock{mut};
            return i;
        }

        void set(int i_)
        {
            std::lock_guard<std::mutex> lock{mut};
            i = i_;
        }

    protected:
        int i;
        mutable std::mutex mut;
};
```

Leggi Funzione di sovraccarico online: https://riptutorial.com/it/cplusplus/topic/510/funzione-di-sovraccarico

Capitolo 41: Funzioni costanti dei membri della classe

Osservazioni

Cosa significa "funzioni membro const" di una classe. La semplice definizione sembra essere quella, una funzione membro const non può cambiare l'oggetto. Ma cosa significa "non può cambiare" significa davvero qui. Significa semplicemente che non puoi svolgere un compito per i membri dei dati della classe.

Tuttavia, puoi eseguire altre operazioni indirette come inserire una voce in una mappa come mostrato nell'esempio. Permettendo che questo possa sembrare che questa funzione const stia modificando l'oggetto (sì, lo fa in un certo senso), ma è permesso.

Quindi, il vero significato è che una funzione membro const non può eseguire un assegnamento per le variabili dei dati di classe. Ma può fare altre cose come spiegato nell'esempio.

Examples

funzione membro costante

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;             // This works? Yes it does.
        delete mapOfStrings;                      // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
};
```

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

    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Leggi Funzioni costanti dei membri della classe online:
https://riptutorial.com/it/cplusplus/topic/7120/funzioni-costanti-dei-membri-della-classe

Capitolo 42: Funzioni dei membri virtuali

Sintassi

- virtual void f ();
- virtual void g () = 0;
- // C ++ 11 o successivo:
    - virtual void h () override;
    - void i () override;
    - virtual void j () final;
    - void k () final;

Osservazioni

- Solo le funzioni membro non statiche e non modello possono essere `virtual` .
- Se si utilizza C ++ 11 o versioni successive, si consiglia di utilizzare l' `override` quando si esegue l'override di una funzione membro virtuale da una classe base.
- Le classi di base polimorfiche hanno spesso distruttori virtuali per consentire l'eliminazione di un oggetto derivato tramite un puntatore alla classe base . Se il distruttore non era virtuale, tale operazione porta a un comportamento non definito [expr.delete] §5.3.5 / 3 .

Examples

Uso dell'override con virtuale in C ++ 11 e versioni successive

L' `override` specificatore ha un significato speciale in C ++ 11 in poi, se aggiunto alla fine della firma della funzione. Questo significa che una funzione è

- Override della funzione presente nella classe base e
- La funzione di classe Base è `virtual`

Non vi è alcun significato di `run time` di questo identificatore poiché è inteso principalmente come indicazione per i compilatori

L'esempio seguente dimostrerà il cambiamento nel comportamento con il nostro senza utilizzare l'override.

Senza intervento `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
```

```
};

struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

Con `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Nota che l' `override` non è una parola chiave, ma un identificatore speciale che può comparire solo nelle firme delle funzioni. In tutti gli altri contesti, l' `override` può ancora essere utilizzato come identificatore:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2;  // Compilation error: keywords can't be used as identifiers.
}
```

Funzioni membro virtuale vs non virtuale

Con funzioni membro virtuali:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
```

```
}
```

Senza funzioni membro virtuali:

```
#include <iostream>

struct X {
   void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
   void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Funzioni virtuali finali

C ++ 11 ha introdotto `final` identificatore `final` che vieta l'override del metodo se comparso nella firma del metodo:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

Lo specificatore `final` può essere utilizzato solo con la funzione membro virtuale e non può essere

applicato a funzioni membro non virtuali

Come `final` , c'è anche un 'override' del chiamante che impedisce la sovrascrittura delle funzioni `virtual` nella classe derivata.

Gli specifiers `override` e `final` possono essere combinati insieme per ottenere l'effetto desiderato:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Comportamento delle funzioni virtuali in costruttori e distruttori

Il comportamento delle funzioni virtuali nei costruttori e nei distruttori spesso confonde quando viene rilevato per la prima volta.

```
#include <iostream>
using namespace std;

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", "  << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }

};

int main() {
     derived d;
}
```

Produzione:

Quando chiamato dal costruttore di base, viene chiamato base :: v ().
Quando chiamato dal costruttore derivato, viene chiamato derivato :: v ().
Quando viene chiamato dal derivato derivato, viene chiamato derivato :: v ().
Quando viene chiamato dal distruttore di base, viene chiamato base :: v ().

Il ragionamento dietro questo è che la classe derivata può definire membri aggiuntivi che non sono

ancora inizializzati (nel caso del costruttore) o già distrutti (nel caso del distruttore) e chiamare le sue funzioni membro non sarebbe sicuro. Pertanto durante la costruzione e la distruzione di oggetti C ++, il tipo *dinamico* di `*this` è considerato come la classe del costruttore o del distruttore e non una classe più derivata.

Esempio:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override   { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Pure funzioni virtuali

Possiamo anche specificare che una funzione `virtual` è *pura virtuale* (astratta), aggiungendo `= 0` alla dichiarazione. Le classi con una o più funzioni virtuali pure sono considerate astratte e non possono essere istanziate; solo le classi derivate che definiscono o ereditano le definizioni per tutte le funzioni virtuali pure possono essere istanziate.

```
struct Abstract {
    virtual void f() = 0;
};

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Anche se una funzione è specificata come pura virtualità, può essere fornita un'implementazione predefinita. Nonostante ciò, la funzione sarà ancora considerata astratta e le classi derivate dovranno definirla prima che possano essere istanziate. In questo caso, la versione della funzione derivata della classe è anche autorizzata a chiamare la versione della classe base.

```
struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

Ci sono un paio di motivi per cui potremmo voler fare questo:

- Se vogliamo creare una classe che non può essere istanziata da sola, ma non impedisce l'istanziazione delle sue classi derivate, possiamo dichiarare il distruttore come virtuale puro. Essendo il distruttore, deve essere definito comunque, se vogliamo essere in grado di deallocare l'istanza. E poiché il distruttore è probabilmente già virtuale per prevenire perdite di memoria durante l'uso polimorfico , non si verificherà un colpo di prestazioni non necessario dal dichiarare un'altra funzione `virtual` . Questo può essere utile quando si creano interfacce.

  ```
  struct Interface {
      virtual ~Interface() = 0;
  };
  Interface::~Interface() = default;

  struct Implementation : Interface {};
  // ~Implementation() is automatically defined by the compiler if not explicitly
  //  specified, meeting the "must be defined before instantiation" requirement.
  ```

- Se la maggior parte o tutte le implementazioni della pura funzione virtuale conterranno il codice duplicato, tale codice può invece essere spostato nella versione della classe base, rendendo più semplice il mantenimento del codice.

  ```
  class SharedBase {
      State my_state;
      std::unique_ptr<Helper> my_helper;
      // ...

    public:
      virtual void config(const Context& cont) = 0;
      // ...
  };
  /* virtual */ void SharedBase::config(const Context& cont) {
      my_helper = new Helper(my_state, cont.relevant_field);
      do_this();
      and_that();
  }

  class OneImplementation : public SharedBase {
      int i;
      // ...

    public:
      void config(const Context& cont) override;
      // ...
  };
  ```

```
void OneImplementation::config(const Context& cont) /* override */ {
    my_state = { cont.some_field, cont.another_field, i };
    SharedBase::config(cont);
    my_unique_setup();
};

// And so on, for other classes derived from SharedBase.
```

Leggi Funzioni dei membri virtuali online: https://riptutorial.com/it/cplusplus/topic/1752/funzioni-dei-membri-virtuali

Capitolo 43: Funzioni inline

introduzione

Una funzione definita con lo specificatore in `inline` è una funzione inline. Una funzione inline può essere definita in modo multiplo senza violare la regola One Definition e può quindi essere definita in un'intestazione con linkage esterno. Dichiarare una funzione suggerisce inline al compilatore che la funzione debba essere sottolineata durante la generazione del codice, ma non fornisce una garanzia.

Sintassi

- **`inline`** *function_declaration*
- *funzione_definizione in* **`inline`**
- *class {function_definition};*

Osservazioni

Di solito se il codice generato per una funzione è *sufficientemente* piccolo, allora è un buon candidato essere in linea. Perchè così? Se una funzione è di grandi dimensioni ed è racchiusa in un ciclo, per tutte le chiamate effettuate, il codice della funzione di grandi dimensioni verrebbe duplicato, con conseguente aumento delle dimensioni del file binario generato. Ma quanto è piccolo?

Mentre le funzioni inline sembrano essere un ottimo modo per evitare il sovraccarico della funzione, è necessario notare che non tutte le funzioni contrassegnate in `inline` sono inline. In altre parole, quando si dice in `inline` , è solo un suggerimento per il compilatore, non un ordine: il compilatore non è obbligato a incorporare la funzione, è libero di ignorarlo - molti di loro lo fanno. I compilatori moderni sono più bravi a rendere tali ottimizzazioni che questa parola chiave è ormai una traccia del passato, quando questo suggerimento della funzione di inlining da parte del programmatore è stato preso sul serio dai compilatori. Anche le funzioni non marcate in `inline` sono delineate dal compilatore quando ne vede i benefici.

In linea come direttiva di collegamento

L'uso più pratico di `inline` nel C ++ moderno deriva dal suo utilizzo come direttiva di collegamento. Quando si *definisce* , non dichiarando, una funzione in un'intestazione che verrà inclusa in più fonti, ciascuna unità di traduzione avrà la propria copia di questa funzione che porta a una violazione ODR (One Definition Rule); questa regola dice approssimativamente che può esserci solo una definizione di una funzione, una variabile, ecc. Per aggirare questa violazione, contrassegnando la definizione della funzione `inline` implicitamente rende il collegamento della funzione interno.

FAQs

Quando dovrei scrivere la parola chiave "inline" per una funzione / metodo in C ++?

Solo quando si desidera definire la funzione in un'intestazione. Più esattamente solo quando la definizione della funzione può essere visualizzata in più unità di compilazione. È una buona idea definire funzioni piccole (come in un liner) nel file di intestazione in quanto fornisce al compilatore maggiori informazioni su cui lavorare mentre ottimizza il codice. Aumenta anche il tempo di compilazione.

Quando non dovrei scrivere la parola chiave 'inline' per una funzione / metodo in C ++?

Non aggiungere in `inline` quando pensi che il tuo codice verrà eseguito più velocemente se il compilatore lo indicherà.

Quando il compilatore non saprà quando eseguire una funzione / metodo in linea?

Generalmente, il compilatore sarà in grado di farlo meglio di te. Tuttavia, il compilatore non ha l'opzione per il codice inline se non ha la definizione della funzione. Nel codice ottimizzato di solito tutti i metodi privati sono in linea se lo chiedi o no.

Guarda anche

- Quando dovrei scrivere la parola chiave "in linea" per una funzione / metodo?
- C'è ancora un uso per inline?

Examples

Dichiarazione di funzione inline non membro

```
inline int add(int x, int y);
```

Definizione della funzione inline non membro

```
inline int add(int x, int y)
{
    return x + y;
}
```

Funzioni inline dei membri

```
// header (.hpp)
struct A
{
```

```
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

Qual è la funzione di inlining?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

Nel codice precedente, quando `add` è in linea, il codice risultante diventerebbe qualcosa di simile

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

La funzione inline non si vede da nessuna parte, il suo corpo viene *inserito* nel corpo del chiamante. Era `add` non è stato inline, una funzione sarebbe chiamato. L'overhead di chiamare una funzione, come la creazione di un nuovo stack frame , la copia di argomenti, la creazione di variabili locali, il salto (perdita della località di riferimento e là da errori di cache), ecc., Devono essere sostenuti.

Leggi Funzioni inline online: https://riptutorial.com/it/cplusplus/topic/7150/funzioni-inline

Capitolo 44: Funzioni membro non statico

Sintassi

- // Chiamando:
    - variable.member_function ();
    - variable_pointer-> member_function ();
- // Definizione:
    - ret_type class_name :: member_function () cv-qualifiers {
        - corpo;
    - }
- // Prototipo:
    - classe class_name {
        - virt-specificatore ret_type member_function () cv-qualifiers virt-specificatore-seq;
        - // virt-specificatore: "virtuale", se applicabile.
        - // cv-qualifiers: "const" e / o "volatile", se applicabile.
        - // virt-specifier-seq: "override" e / o "final", se applicabile.
    - }

Osservazioni

Un non `static` funzione membro è una `class` / `struct` / `union` funzione membro, che è chiamato un caso particolare, e opera su detta istanza. A differenza delle funzioni membro `static` , non può essere chiamato senza specificare un'istanza.

Per informazioni su classi, strutture e sindacati, consultare l'argomento principale .

Examples

Funzioni membro non statiche

Una `class` o una `struct` possono avere funzioni membro e variabili membro. Queste funzioni hanno una sintassi per lo più simile alle funzioni standalone e possono essere definite all'interno o all'esterno della definizione della classe; se definito al di fuori della definizione della classe, il nome della funzione è preceduto dal nome della classe e dall'operatore scope (`::` :).

```
class CL {
  public:
    void  definedInside() {}
    void definedOutside();
};
```

```
void CL::definedOutside() {}
```

Queste funzioni sono richiamate su un'istanza (o un riferimento a un'istanza) della classe con l'operatore punto (`.`) O un puntatore a un'istanza con l'operatore freccia (`->`) e ogni chiamata è legata all'istanza la funzione è stato chiamato; quando una funzione membro viene chiamata su un'istanza, ha accesso a tutti i campi dell'istanza (tramite il puntatore `this`), ma può accedere solo ai campi di altre istanze se tali istanze vengono fornite come parametri.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

  private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int  i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

Queste funzioni possono accedere alle variabili membro e / o alle altre funzioni membro, indipendentemente dalla variabile o dai modificatori di accesso della funzione. Possono anche essere scritti fuori ordine, accedere alle variabili membro e / o chiamare le funzioni membro dichiarate prima di loro, poiché l'intera definizione di classe deve essere analizzata prima che il compilatore possa iniziare a compilare una classe.

```
class Access {
  public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
  protected:
    int j;
    int get_i() const { return i; }
  private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

incapsulamento

Un uso comune delle funzioni membro è per l'incapsulamento, utilizzando un *accessore* (comunemente noto come getter) e un *mutatore* (comunemente noto come setter) invece di accedere direttamente ai campi.

```
class Encapsulator {
```

```
    int encapsulated;

  public:
    int  get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e)  { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

All'interno della classe, `encapsulated` può essere liberamente accessibile da qualsiasi funzione membro non statico; al di fuori della classe, l'accesso ad esso è regolato dalle funzioni membro, usando `get_encapsulated()` per leggerlo e `set_encapsulated()` per modificarlo. Ciò impedisce modifiche non intenzionali alla variabile, poiché vengono utilizzate funzioni separate per leggerlo e scriverlo. [Ci sono molte discussioni sul fatto che getter e setter forniscano o interrompano l'incapsulamento, con buoni argomenti per entrambe le affermazioni; un dibattito così acceso non rientra negli scopi di questo esempio.]

Nascondere e importare il nome

Quando una classe base fornisce un insieme di funzioni sovraccariche e una classe derivata aggiunge un altro sovraccarico all'insieme, questo nasconde tutti i sovraccarichi forniti dalla classe base.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1);    // Output:  int
hb.f(true); // Output:  bool
hb.f(s);    // Output:  std::string;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  float
hd.f(true); // Output:  float
hd.f(s);    // Error: Can't convert from std::string to float.
```

Ciò è dovuto al nome delle regole di risoluzione: durante la ricerca del nome, una volta trovato il nome corretto, smettiamo di cercare, anche se chiaramente non abbiamo trovato la *versione* corretta dell'entità con quel nome (come con `hd.f(s)`); a causa di ciò, sovraccaricare la funzione nella classe derivata impedisce la ricerca del nome dalla scoperta degli overload nella classe

base. Per evitare ciò, è possibile utilizzare una dichiarazione di utilizzo per "importare" i nomi dalla classe base nella classe derivata, in modo che siano disponibili durante la ricerca del nome.

```
struct HidingDerived : HiddenBase {
     // All members named HiddenBase::f shall be considered members of HidingDerived for
lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  int
hd.f(true); // Output:  bool
hd.f(s);    // Output:  std::string
```

Se una classe derivata importa nomi con una use-declaration, ma dichiara anche funzioni con la stessa firma delle funzioni nella classe base, le funzioni della classe base saranno silenziate o nascoste in modo silenzioso.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()    {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

È inoltre possibile utilizzare una dichiarazione di utilizzo per modificare i modificatori di accesso, a condizione che l'entità importata fosse `public` o `protected` nella classe base.

```
struct ProMem {
  protected:
    void func() {}
};

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.
```

Allo stesso modo, se vogliamo chiamare esplicitamente una funzione membro da una classe specifica nella gerarchia dell'ereditarietà, possiamo qualificare il nome della funzione quando chiamiamo la funzione, specificando quella classe per nome.

```
struct One {
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.
```

Funzioni dei membri virtuali

Le funzioni membro possono anche essere dichiarate `virtual` . In questo caso, se richiamati su un puntatore o su un riferimento a un'istanza, non saranno accessibili direttamente; piuttosto, cercheranno la funzione nella tabella delle funzioni virtuali (un elenco di funzioni da puntatore a membro per le funzioni virtuali, più comunemente noto come `vtable` o `vftable`) e la useranno per chiamare la versione appropriata per la dinamica dell'istanza (effettivo) tipo. Se la funzione viene chiamata direttamente, da una variabile di una classe, non viene eseguita alcuna ricerca.

```
struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...

Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.
```

```
b.func();   // Output:  In Base.
d.func();   // Output:  In Derived.

pb->func(); // Output:  In Base.
pd->func(); // Output:  In Derived.

rb.func();  // Output:  In Base.
rd.func();  // Output:  In Derived.

slicer(b);  // Output:  In Base.
slicer(d);  // Output:  In Base.
```

Si noti che mentre `pd` è `Base*` , e `rd` è una `Base&` , chiamando `func()` su una delle due chiamate `Derived::func()` invece di `Base::func()` ; questo perché il `vtable` per `Derived` aggiorna la voce `Base::func()` per puntare invece a `Derived::func()` . Al contrario, si noti come passare un'istanza a `slicer()` comporta sempre il richiamo di `Base::func()` , anche quando l'istanza passata è `Derived` ; ciò è dovuto a qualcosa noto come *slicing dei dati* , in cui il passaggio di un'istanza `Derived` in un parametro `Base` per valore rende la parte dell'istanza `Derived` che non è un'istanza di `Base` inaccessibile.

Quando una funzione membro è definita come virtuale, tutte le funzioni membro membro derivate con la stessa firma lo sovrascrivono, indipendentemente dal fatto che la funzione di override sia specificata come `virtual` o meno. Questo può rendere le classi derivate più difficili da analizzare per i programmatori, tuttavia, poiché non vi è alcuna indicazione su quale funzione sia / sono `virtual` .

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                //  You'd have to check B to know that, though.
};
```

Si noti, tuttavia, che una funzione derivata sovrascrive una funzione di base solo se le loro firme corrispondono; anche se una funzione derivata è dichiarata esplicitamente `virtual` , creerà invece una nuova funzione virtuale se le firme non corrispondono.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

C ++ 11

A partire da C ++ 11, l'intento di sovrascrivere può essere reso esplicito con la `override` parola chiave sensibile al contesto. Questo dice al compilatore che il programmatore si aspetta che sovrascriva una funzione di classe base, che fa sì che il compilatore ometta un errore se *non*

sovrascrive nulla.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

Ciò ha anche il vantaggio di dire ai programmatori che la funzione è sia virtuale, sia anche dichiarata in almeno una classe base, che può rendere più semplice l'analisi di classi complesse.

Quando una funzione è dichiarata `virtual` e definita al di fuori della definizione della classe, lo specificatore `virtual` deve essere incluso nella dichiarazione della funzione e non ripetuto nella definizione.

C ++ 11

Questo vale anche per l' `override` .

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

Se una classe base sovraccarica una funzione `virtual` , solo gli overload specificati come `virtual` saranno virtuali.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

Per ulteriori informazioni, consultare l'argomento pertinente .

Const Correctness

Uno degli usi principali di `this` cv-qualificatore è la *correttezza* `const` . Questa è la pratica di garantire che solo gli accessi che *devono* modificare un oggetto siano in *grado* di modificare

l'oggetto e che qualsiasi funzione (membro o non membro) che non ha bisogno di modificare un oggetto non abbia accesso in scrittura a tale oggetto (direttamente o indirettamente). Ciò impedisce modifiche involontarie, rendendo il codice meno errorprone. Inoltre, consente a qualsiasi funzione che non ha bisogno di modificare lo stato di essere in grado di prendere un oggetto `const` o non `const` , senza la necessità di riscrivere o sovraccaricare la funzione.

`const` correttezza `const` , per sua natura, inizia dal basso verso l'alto: qualsiasi funzione membro della classe che non ha bisogno di cambiare stato è dichiarata come `const` , in modo che possa essere richiamata su istanze `const` . Questo, a sua volta, consente ai parametri passati per riferimento di essere dichiarati `const` quando non hanno bisogno di essere modificati, il che consente alle funzioni di prendere gli oggetti `const` o non `const` senza lamentarsi, e la `const` -ness può propagarsi all'esterno in questo maniera. A causa di ciò, i getter sono frequentemente `const` , come lo sono tutte le altre funzioni che non hanno bisogno di modificare lo stato logico.

```
class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(const Field& f) : fld(f) {}     // Modifies.

    const Field& get_field()       { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; }    // Modifies.

    void do_something(int i) {                     // Modifies.
        fld.insert_value(i);
    }
    void do_nothing()        { }                   // Doesn't modify; should be const.
};

class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f) : fld(f) {}       // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }    // Not const: Modifies.

    void do_something(int i) {                     // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const  { }                   // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
  // Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
  // Error: Same as above.
// Oops.

const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
```

```
but_i_can.do_nothing();          // Good.
```

Come illustrato dai commenti su `ConstIncorrect` e `ConstCorrect` , anche le funzioni di cv-qualifying servono da documentazione. Se una classe è `const` corretta, qualsiasi funzione che non è `const` può essere assunta in modo sicuro per cambiare stato, e qualsiasi funzione che è `const` può essere considerata come sicura per non cambiare stato.

Leggi Funzioni membro non statico online: https://riptutorial.com/it/cplusplus/topic/5661/funzioni-membro-non-statico

Capitolo 45: Funzioni speciali per gli utenti

Examples

Distruttori virtuali e protetti

Una classe progettata per essere ereditata da una classe base viene chiamata. Bisogna fare attenzione con le funzioni speciali dei membri di tale classe.

Una classe progettata per essere utilizzata polimorficamente in fase di esecuzione (tramite un puntatore alla classe base) dovrebbe dichiarare il distruttore `virtual` . Ciò consente di distruggere correttamente le parti derivate dell'oggetto, anche quando l'oggetto viene distrutto attraverso un puntatore alla classe base.

```
class Base {
public:
    virtual ~Base() = default;

private:
    //    data members etc.
};

class Derived : public Base { //  models Is-A relationship
public:
    //    some methods

private:
    //    more data members
};

//    virtual destructor in Base ensures that derived destructors
//    are also called when the object is destroyed
std::unique_ptr<Base> base = std::make_unique<Derived>();
base = nullptr;  //    safe, doesn't leak Derived's members
```

Se la classe non ha bisogno di essere polimorfica, ma deve comunque consentire che la sua interfaccia sia ereditata, utilizzare un distruttore non virtuale `protected` .

```
class NonPolymorphicBase {
public:
    //    some methods

protected:
    ~NonPolymorphicBase() = default; //    note: non-virtual

private:
    //    etc.
};
```

Una tale classe non può mai essere distrutta attraverso un puntatore, evitando perdite silenziose dovute alla divisione.

Questa tecnica si applica in particolare alle classi progettate per essere classi base `private` . Tale classe potrebbe essere utilizzata per incapsulare alcuni dettagli di implementazione comuni, fornendo al tempo stesso metodi `virtual` come punti di personalizzazione. Questo tipo di classe non dovrebbe mai essere utilizzato in modo polimorfico e un distruttore `protected` aiuta a documentare questo requisito direttamente nel codice.

Infine, alcune classi potrebbero richiedere che non vengano *mai* utilizzate come classe base. In questo caso, la classe può essere contrassegnata come `final` . In questo caso, un normale distruttore pubblico non virtuale va bene.

```
class FinalClass final {  //    marked final here
public:
    ~FinalClass() = default;

private:
    //    etc.
};
```

Sposta e copia impliciti

Ricordare che la dichiarazione di un distruttore impedisce al compilatore di generare costruttori di movimento impliciti e spostare gli operatori di assegnazione. Se dichiari un distruttore, ricorda di aggiungere anche le definizioni appropriate per le operazioni di spostamento.

Inoltre, la dichiarazione delle operazioni di spostamento sopprimerà la generazione delle operazioni di copia, quindi anche queste dovrebbero essere aggiunte (se agli oggetti di questa classe è richiesto di avere semantica della copia).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Copia e scambia

Se stai scrivendo una classe che gestisce le risorse, devi implementare tutte le funzioni dei membri speciali (vedi Regola di tre / cinque / zero). L'approccio più diretto alla scrittura del costruttore di copie e dell'operatore di assegnazione sarebbe:

```
person(const person &other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
```

```
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

Ma questo approccio ha alcuni problemi. Fallisce la forte garanzia di eccezione - se i `new[]` lanci `new[]` , abbiamo già eliminato le risorse possedute da `this` e non possiamo recuperare. Stiamo duplicando gran parte della logica della costruzione delle copie nell'assegnazione delle copie. E dobbiamo ricordare il controllo dell'auto-assegnazione, che di solito aggiunge solo un sovraccarico all'operazione di copia, ma è ancora critico.

Per soddisfare la forte garanzia di eccezione ed evitare la duplicazione del codice (doppio quindi con il successivo operatore di assegnazione del movimento), possiamo usare l'idioma copy-and-swap:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

Perché funziona? Considera cosa succede quando abbiamo

```
person p1 = ...;
person p2 = ...;
p1 = p2;
```

Per prima cosa, copia-costruisci `rhs` da `p2` (che non abbiamo dovuto duplicare qui). Se l'operazione viene lanciata, non facciamo nulla in `operator=` e `p1` rimane intatto. Successivamente, scambiamo i membri tra `*this` e `rhs` , e quindi `rhs` esce dall'ambito. Quando `operator=` , pulisce implicitamente le risorse originali di `this` (tramite il distruttore, che non abbiamo dovuto duplicare).

Anche l'autoassegnazione funziona - è meno efficiente con copy-and-swap (comporta un'allocazione e una deallocazione aggiuntive), ma se questo è lo scenario improbabile, non rallentiamo il tipico caso d'uso per renderlo conto.

C ++ 11

La formulazione sopra funziona come-è già per l'assegnazione del movimento.

```
p1 = std::move(p2);
```

Qui, spostiamo-costruisci `rhs` da `p2` , e tutto il resto è altrettanto valido. Se una classe è mobile ma non è copiabile, non è necessario cancellare l'assegnazione della copia, poiché questo operatore di assegnazione sarà semplicemente mal formato a causa del costruttore di copie cancellato.

Costruttore predefinito

Un *costruttore predefinito* è un tipo di costruttore che non richiede parametri quando viene chiamato. Prende il nome dal tipo che costruisce ed è una funzione membro di esso (come lo sono tutti i costruttori).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Un altro modo per soddisfare il requisito "nessun parametro" è che lo sviluppatore fornisca valori predefiniti per tutti i parametri:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};


D d; // calls constructor of D with the provided default values for the parameters
```

In alcune circostanze (cioè, lo sviluppatore non fornisce costruttori e non ci sono altre condizioni di squalifica), il compilatore fornisce implicitamente un costruttore predefinito vuoto:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Avere qualche altro tipo di costruttore è una delle condizioni di squalifica menzionate in precedenza:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};

C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

c ++ 11

Per evitare la creazione implicita di un costruttore predefinito, una tecnica comune è dichiararla come `private` (senza definizione). L'intenzione è quella di causare un errore di compilazione quando qualcuno tenta di utilizzare il costruttore (questo risulta in un errore di *Access to private* o di linker, a seconda del compilatore).

Per essere sicuri che un costruttore predefinito (funzionalmente simile a quello implicito) sia definito, uno sviluppatore potrebbe scriverne uno vuoto in modo esplicito.

c ++ 11

In C ++ 11, uno sviluppatore può anche usare la parola chiave `delete` per impedire al compilatore di fornire un costruttore predefinito.

```
class C{
    int i;
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

Inoltre, uno sviluppatore potrebbe anche essere esplicito sul volere che il compilatore fornisca un costruttore predefinito.

```
class C{
    int i;
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;
```

```
    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

c ++ 14

È possibile determinare se un tipo ha un costruttore predefinito (o è un tipo primitivo) utilizzando `std::is_default_constructible` da `<type_traits>` :

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

c ++ 11

In C ++ 11, è ancora possibile utilizzare la versione non-functor di `std::is_default_constructible` :

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Distruttore

Un *distruttore* è una funzione senza argomenti chiamata quando un oggetto definito dall'utente sta per essere distrutto. Prende il nome dal tipo che distrugge con un prefisso `~` .

```
class C{
    int* is;
    string s;
public:
    C()
    : is( new int[10] ){
    }

    ~C(){  // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
public:
    C_child(){}
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
```

```
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch;  // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

Nella maggior parte dei casi (cioè, un utente non fornisce alcun distruttore e non ci sono altre condizioni di squalifica), il compilatore fornisce implicitamente un distruttore predefinito:

```
class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
```

c ++ 11

In C ++ 11, uno sviluppatore può sovrascrivere questo comportamento impedendo al compilatore di fornire un distruttore predefinito.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
    C c1;
} // Compile ERROR: C has no destructor
```

Inoltre, uno sviluppatore potrebbe anche essere esplicito sul fatto che il compilatore debba fornire un distruttore predefinito.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

c ++ 11

È possibile determinare se un tipo ha un distruttore (o è un tipo primitivo) utilizzando `std::is_destructible` da `<type_traits>` :

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Leggi Funzioni speciali per gli utenti online: https://riptutorial.com/it/cplusplus/topic/1476/funzioni-speciali-per-gli-utenti

Capitolo 46: Futures e promesse

introduzione

Promesse e Futures vengono utilizzati per traghettare un singolo oggetto da un thread all'altro.

Un oggetto `std::promise` è impostato dal thread che genera il risultato.

Un oggetto `std::future` può essere utilizzato per recuperare un valore, per verificare se un valore è disponibile o per interrompere l'esecuzione fino a quando il valore non è disponibile.

Examples

std :: future e std :: promise

L'esempio seguente imposta una promessa per essere utilizzata da un altro thread:

```
    {
        auto promise = std::promise<std::string>();

        auto producer = std::thread([&]
        {
            promise.set_value("Hello World");
        });

        auto future = promise.get_future();

        auto consumer = std::thread([&]
        {
            std::cout << future.get();
        });

        producer.join();
        consumer.join();
}
```

Esempio asincrono rinviato

Questo codice implementa una versione di `std::async` , ma si comporta come se `async` venisse sempre chiamato con il criterio di avvio `deferred` . Questa funzione inoltre non ha lo speciale comportamento `future async` ; il `future` restituito può essere distrutto senza mai acquisire il suo valore.

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    using result_type = decltype(func());

    auto promise = std::promise<result_type>();
    auto future  = promise.get_future();
```

```
    std::thread(std::bind([=](std::promise<result_type>& promise)
    {
        try
        {
            promise.set_value(func());
            // Note: Will not work with std::promise<void>. Needs some meta-template
programming which is out of scope for this example.
        }
        catch(...)
        {
            promise.set_exception(std::current_exception());
        }
    }, std::move(promise))).detach();

    return future;
}
```

std :: packaged_task e std :: future

`std::packaged_task` raggruppa una funzione e la promessa associata per il suo tipo di ritorno:

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task   = std::packaged_task<decltype(func())()>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}
```

Il thread inizia a correre immediatamente. Possiamo staccarlo o unirlo alla fine dello scope. Quando termina la chiamata alla funzione std :: thread, il risultato è pronto.

Si noti che questo è leggermente diverso da `std::async` dove lo `std::future` restituito quando viene distrutto in realtà **bloccherà** fino al termine del thread.

std :: future_error e std :: future_errc

Se non vengono soddisfatti i vincoli per std :: promise e std :: future non viene generata un'eccezione di tipo std :: future_error.

Il membro del codice di errore nell'eccezione è di tipo std :: future_errc e i valori sono i seguenti, insieme ad alcuni casi di test:

```
enum class future_errc {
    broken_promise             = /* the task is no longer shared */,
    future_already_retrieved   = /* the answer was already retrieved */,
    promise_already_satisfied  = /* the answer was stored already */,
    no_state                   = /* access to a promise in non-shared state */
};
```

Promessa non attiva:

```
int test()
{
    std::promise<int> pr;
    return 0; // returns ok
}
```

Promessa attiva, non utilizzata:

```
  int test()
    {
        std::promise<int> pr;
        auto fut = pr.get_future(); //blocks indefinitely!
        return 0;
    }
```

Doppio recupero:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

    try{
        auto fut2 = pr.get_future();    //   second attempt to get future
        return 0;
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The future has already been retrieved
from the promise or packaged_task."
        return -1;
    }
    return fut2.get();
}
```

Impostazione di std :: promise value due volte:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10);  // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The state of the promise has already been
set."
        return -1;
    }
    return fut.get();
}
```

std :: future e std :: async

Nel seguente esempio di naie parallelo sort sort, `std::async` viene utilizzato per avviare più attività parallele merge_sort. `std::future` è usato per aspettare i risultati e sincronizzarli:

```
#include <iostream>
using namespace std;


void merge(int low,int mid,int high, vector<int>&num)
{
    vector<int> copy(num.size());
    int h,i,j,k;
    h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
            copy[i]=num[h];
            h++;
        }
        else
        {
            copy[i]=num[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    else
    {
        for(k=h;k<=mid;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}


void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low<high)
    {
        mid = low + (high-low)/2;
        auto future1    =  std::async(std::launch::deferred,[&]()
```

```
                                        {
                                          merge_sort(low,mid,num);
                                        });
        auto future2    =  std::async(std::launch::deferred, [&]()
                                         {
                                            merge_sort(mid+1,high,num) ;
                                         });

        future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}
```

Nota: nell'esempio `std::async` viene avviato con policy `std::launch_deferred` . Questo per evitare che un nuovo thread venga creato in ogni chiamata. Nel caso del nostro esempio, le chiamate a `std::async` sono fatte fuori ordine, si sincronizzano alle chiamate per `std::future::get()` .

`std::launch_async` forza la `std::launch_async` un nuovo thread in ogni chiamata.

La politica di default è `std::launch::deferred| std::launch::async` , ovvero l'implementazione determina la politica per la creazione di nuovi thread.

Classi di operazioni asincrone

- std :: async: esegue un'operazione asincrona.
- std :: future: fornisce l'accesso al risultato di un'operazione asincrona.
- std :: promise: pacchetti il risultato di un'operazione asincrona.
- std :: packaged_task: raggruppa una funzione e la promessa associata per il suo tipo di ritorno.

Leggi Futures e promesse online: https://riptutorial.com/it/cplusplus/topic/9840/futures-e-promesse

Capitolo 47: Generazione di numeri casuali

Osservazioni

La generazione di numeri casuali in C ++ è fornita dall'intestazione `<random>` . Questa intestazione definisce dispositivi casuali, generatori pseudo-casuali e distribuzioni.

I dispositivi casuali restituiscono numeri casuali forniti dal sistema operativo. Dovrebbero essere utilizzati per l'inizializzazione di generatori pseudo casuali o direttamente per scopi crittografici.

I generatori pseudo-casuali restituiscono numeri pseudo-casuali interi in base al seme iniziale. L'intervallo di numeri pseudo-casuali si estende in genere su tutti i valori di un tipo senza segno. Tutti i generatori pseudo-casuali nella libreria standard restituiranno gli stessi numeri per lo stesso seme iniziale per tutte le piattaforme.

Le distribuzioni consumano numeri casuali da generatori pseudo casuali o dispositivi casuali e producono numeri casuali con distribuzione necessaria. Le distribuzioni non sono indipendenti dalla piattaforma e possono produrre numeri diversi per gli stessi generatori con le stesse sementi iniziali su piattaforme diverse.

Examples

Vero generatore di valori casuali

Per generare valori casuali veri che possono essere usati per la crittografia, `std::random_device` deve essere usato come generatore.

```
#include <iostream>
#include <random>

int main()
{
   std::random_device crypto_random_generator;
   std::uniform_int_distribution<int> int_distribution(0,9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(crypto_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i < 10; i++) {
       std::cout << actual_distribution[i] << " ";
   }

   return 0;
}
```

`std::random_device` è usato allo stesso modo di un generatore di valori pseudo casuali.

Tuttavia, `std::random_device` **può essere implementato in termini di un motore di numeri pseudo-casuali definito dall'implementazione** se una sorgente non deterministica (ad esempio un dispositivo hardware) non è disponibile all'implementazione.

La rilevazione di tali implementazioni dovrebbe essere possibile tramite la funzione membro `entropy` (che restituisce zero quando il generatore è completamente deterministico), ma molte librerie popolari (sia libstdc ++ di GCC che libc ++ di LLVM) restituiscono sempre zero, anche quando usano casualità esterna di alta qualità .

Generazione di un numero pseudo-casuale

Un generatore di numeri pseudo casuali genera valori che possono essere indovinati sulla base di valori generati in precedenza. In altre parole: è deterministico. Non utilizzare un generatore di numeri pseudo casuali in situazioni in cui è richiesto un numero casuale reale.

```
#include <iostream>
#include <random>

int main()
{
   std::default_random_engine pseudo_random_generator;
   std::uniform_int_distribution<int> int_distribution(0, 9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(pseudo_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i <= 9; i++) {
       std::cout << actual_distribution[i] << " ";
   }

   return 0;
}
```

Questo codice crea un generatore di numeri casuali e una distribuzione che genera numeri interi nell'intervallo [0,9] con uguale probabilità. Quindi conta quante volte ogni risultato è stato generato.

Il parametro template di `std::uniform_int_distribution<T>` specifica il tipo di intero che dovrebbe essere generato. Usa `std::uniform_real_distribution<T>` per generare float o double.

Utilizzo del generatore per più distribuzioni

Il generatore di numeri casuali può (e dovrebbe) essere utilizzato per più distribuzioni.

```
#include <iostream>
#include <random>

int main()
{
```

```
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);
    std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
    std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

    std::cout << int_distribution(pseudo_random_generator) << std::endl;
    std::cout << float_distribution(pseudo_random_generator) << std::endl;
    std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

    return 0;
}
```

In questo esempio, viene definito un solo generatore. Successivamente viene utilizzato per generare un valore casuale in tre diverse distribuzioni. La distribuzione `rigged_dice` genererà un valore compreso tra 0 e 5, ma quasi sempre genera un `5` , perché la possibilità di generare un `5` è `100 / 105` .

Leggi Generazione di numeri casuali online:
https://riptutorial.com/it/cplusplus/topic/1541/generazione-di-numeri-casuali

Capitolo 48: Gestione della memoria

Sintassi

- : :(*opt*) nuovo (*espressione-elenco*) (*opt*) *nuovo-tipo-id nuovo-inizializzatore* (*opt*)
- : :(*opt*) nuovo (*espressione-elenco*) (*opt*) (*type-id*) *new-initializer* (*opt*)
- : :(*opt*) elimina *cast-expression*
- : :(*opt*) delete [] *cast-expression*
- std :: unique_ptr < *id-tipo* > var_name (nuovo *tipo-id* (*opt*)); // C ++ 11
- std :: shared_ptr < *id-tipo* > var_name (nuovo *tipo-id* (*opt*)); // C ++ 11
- std :: shared_ptr < *id-tipo* > var_name = std :: make_shared < *id-tipo* > (*opt*); // C ++ 11
- std :: unique_ptr < *id-tipo* > var_name = std :: make_unique < *id-tipo* > (*opt*); // C ++ 14

Osservazioni

A leading `::` forza l'operatore new o delete a cercare in ambito globale, ignorando qualsiasi operatore new o delete sovraccarico specifico della classe.

Gli argomenti facoltativi che seguono la `new` parola chiave vengono in genere utilizzati per chiamare il posizionamento nuovo , ma possono anche essere utilizzati per passare ulteriori informazioni all'allocatore, ad esempio un tag che richiede che la memoria venga allocata da un pool selezionato.

Il tipo allocato viene in genere specificato esplicitamente, *ad esempio,* `new Foo` , ma può anche essere scritto come `auto` (dal C ++ 11) o `decltype(auto)` (dal C ++ 14) per dedurlo dall'inizializzatore.

L'inizializzazione dell'oggetto assegnato avviene secondo le stesse regole dell'inizializzazione delle variabili locali. In particolare, l'oggetto verrà inizializzato di default se l'inizializzatore è omesso e, quando si assegna dinamicamente un tipo scalare o un array di tipo scalare, non si garantisce che la memoria venga azzerata.

Un oggetto array creato da una *nuova espressione* deve essere distrutto usando `delete[]` , indipendentemente dal fatto che la *nuova espressione sia* stata scritta con `[]` o meno. Per esempio:

```
using IA = int[4];
int* pIA = new IA;
delete[] pIA;  // right
// delete pIA;  // wrong
```

Examples

Pila

Lo stack è una piccola regione di memoria in cui vengono inseriti valori temporanei durante l'esecuzione. L'allocazione dei dati nello stack è molto veloce rispetto all'allocazione dell'heap, poiché tutta la memoria è già stata assegnata per questo scopo.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

Lo stack è chiamato perché le catene di chiamate di funzione avranno la memoria temporanea "impilata" l'una sull'altra, ognuna utilizzando una piccola sezione separata di memoria.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

I dati memorizzati nello stack sono validi solo finché l'ambito che ha allocato la variabile è ancora attivo.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

Archiviazione libera (heap, allocazione dinamica ...)

Il termine **'heap'** è un termine di calcolo generale che indica un'area di memoria da cui le parti possono essere allocate e deallocate indipendentemente dalla memoria fornita dallo **stack** .

In C++ lo *standard si* riferisce a quest'area come al **negozio gratuito** che è considerato un termine più accurato.

Le aree di memoria allocate dall'archivio **gratuito** potrebbero vivere più a lungo dell'ambito originale in cui è stato allocato. Anche i dati troppo grandi per essere memorizzati nello stack possono essere allocati dal **Free Store** .

La memoria grezza può essere allocata e deallocata dalle parole chiave *new* ed *delete* .

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;              // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;          // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;       // Setting the pointer to nullptr after delete is often considered good
practice
```

È anche possibile allocare array di dimensioni fisse con *nuovi* e *cancella* , con una sintassi leggermente diversa. L'allocazione delle matrici non è compatibile con l'allocazione di non matrice e il mixaggio dei due porterà alla corruzione dell'heap. L'allocazione di una matrice alloca anche la memoria per tenere traccia della dimensione della matrice per l'eliminazione successiva in un modo definito dall'implementazione.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

Quando si utilizza *new* e *delete* invece *malloc* e *free* , il costruttore e il distruttore verranno eseguiti (simile agli oggetti basati sullo stack). Questo è il motivo per cui *new* e *delete* sono preferiti su *malloc* e *gratuiti* .

```
struct ComplexType {
    int a = 0;

    ComplexType() { std::cout << "Ctor" << std::endl; }
    ~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
//Calls the destructor for ComplexType() and deletes memory for a Complextype at pC
delete foo;
```

C ++ 11

Da C ++ 11 in poi, si consiglia l'uso di puntatori intelligenti per indicare la proprietà.

C ++ 14

C ++ 14 ha aggiunto `std::make_unique` , cambiando la raccomandazione per favorire `std::make_unique` o `std::make_shared` invece di usare naked *new* e *delete* .

Posizionamento nuovo

Ci sono situazioni in cui non vogliamo fare affidamento su Free Store per allocare memoria e vogliamo usare allocazioni di memoria personalizzate usando `new` .

Per queste situazioni possiamo usare `Placement New` , dove possiamo dire all'operatore 'new' di allocare memoria da una locazione di memoria pre-allocata

Per esempio

```
int a4byteInteger;

char *a4byteChar = new (&a4byteInteger) char[4];
```

In questo esempio, la memoria puntata da `a4byteChar` è di 4 byte allocata a 'stack' tramite la variabile intera `a4byteInteger` .

Il vantaggio di questo tipo di allocazione di memoria è il fatto che i programmatori controllano l'allocazione. Nell'esempio sopra, poiché `a4byteInteger` è allocato sullo stack, non è necessario effettuare una chiamata esplicita a "delete a4byteChar`.

Lo stesso comportamento può essere raggiunto anche per la memoria allocata dinamica. Per esempio

```
int *a8byteDynamicInteger = new int[2];

char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

In questo caso, il puntatore di memoria di `a8byteChar` si `a8byteChar` alla memoria dinamica allocata da `a8byteDynamicInteger` . In questo caso, tuttavia, è necessario chiamare esplicitamente `delete a8byteDynamicInteger` per rilasciare la memoria

Un altro esempio per la classe C ++

```
struct ComplexType {
    int a;

    ComplexType() : a(0) {}
    ~ComplexType() {}
};

int main() {
    char* dynArray = new char[256];

    //Calls ComplexType's constructor to initialize memory as a ComplexType
    new((void*)dynArray) ComplexType();

    //Clean up memory once we're done
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();
    delete[] dynArray;

    //Stack memory can also be used with placement new
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11

    new((void*)localArray) ComplexType();
```

```
    //Only need to call the destructor for stack memory
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();

    return 0;
}
```

Leggi Gestione della memoria online: https://riptutorial.com/it/cplusplus/topic/2873/gestione-della-memoria

Capitolo 49: Gestione delle risorse

introduzione

Una delle cose più difficili da fare in C e C ++ è la gestione delle risorse. Fortunatamente, in C ++, abbiamo molti modi per progettare la gestione delle risorse nei nostri programmi. Questo articolo spera di spiegare alcuni degli idiomi e dei metodi usati per gestire le risorse allocate.

Examples

L'acquisizione delle risorse è l'inizializzazione

L'acquisizione delle risorse è inizializzazione (RAII) è un idioma comune nella gestione delle risorse. Nel caso della memoria dinamica, utilizza puntatori intelligenti per realizzare la gestione delle risorse. Quando si utilizza RAII, una risorsa acquisita viene immediatamente assegnata a un puntatore intelligente oa un gestore risorse equivalente. La risorsa è accessibile solo attraverso questo gestore, quindi il gestore può tenere traccia di varie operazioni. Ad esempio, `std::auto_ptr` libera automaticamente la sua risorsa corrispondente quando non rientra nell'ambito o viene cancellata in altro modo.

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    {
        auto_ptr ap(new int(5)); // dynamic memory is the resource
        cout << *ap << endl; // prints 5
    } // auto_ptr is destroyed, its resource is automatically freed
}
```

C ++ 11

`std::auto_ptr` problema principale di `std::auto_ptr` è che non può essere copiato senza trasferire la proprietà:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
    cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

A causa di questa strana semantica della copia, `std::auto_ptr` non può essere usato in contenitori,

tra le altre cose. La ragione è che impedisce di cancellare la memoria due volte: se ci sono due `auto_ptrs` con la proprietà della stessa risorsa, entrambi provano a liberarlo quando vengono distrutti. Liberare una risorsa già liberata può generalmente causare problemi, quindi è importante prevenirlo. Tuttavia, `std::shared_ptr` ha un metodo per evitare ciò mentre non trasferisce la proprietà durante la copia:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr sp2;
    {
        shared_ptr sp1(new int(5)); // give ownership to sp1
        cout << *sp1 << endl; // prints 5
        sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
        cout << *sp1 << endl; // prints 5
        cout << *sp2 << endl; // prints 5
    } // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
    cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed
```

Mutex e sicurezza del filo

Possono verificarsi problemi quando più thread tentano di accedere a una risorsa. Per un semplice esempio, supponiamo di avere un thread che aggiunge uno a una variabile. Lo fa leggendo prima la variabile, aggiungendone una, quindi memorizzandola di nuovo. Supponiamo di inizializzare questa variabile su 1, quindi creare due istanze di questo thread. Dopo che entrambi i thread sono terminati, l'intuizione suggerisce che questa variabile abbia un valore di 3. Tuttavia, la tabella seguente illustra cosa potrebbe andare storto:

	Discussione 1	Thread 2
Ora Passo 1	Leggi 1 dalla variabile	
Ora Passo 2		Leggi 1 dalla variabile
Ora Passo 3	Aggiungi 1 più 1 per ottenere 2	
Fase 4		Aggiungi 1 più 1 per ottenere 2
Ora Passo 5	Memorizza 2 in variabile	
Ora Passo 6		Memorizza 2 in variabile

Come puoi vedere, al termine dell'operazione, 2 è nella variabile, invece di 3. Il motivo è che il Thread 2 leggeva la variabile prima che il thread 1 venisse fatto aggiornandolo. La soluzione? Mutex.

Un mutex (portmanteau di **mutual ex** clusione) è un oggetto di gestione delle risorse progettato per risolvere questo tipo di problema. Quando un thread vuole accedere a una risorsa,

"acquisisce" il mutex della risorsa. Una volta fatto l'accesso alla risorsa, il thread "rilascia" il mutex. Mentre il mutex viene acquisito, tutte le chiamate per acquisire il mutex non ritorneranno fino a quando il mutex non verrà rilasciato. Per capire meglio, pensate a un mutex come linea di attesa al supermercato: i fili si allineano cercando di acquisire il mutex e poi aspettando che i thread che li precedono finiscano, quindi usando la risorsa, quindi uscendo da linea rilasciando il mutex. Ci sarebbe un pandemonio completo se tutti provassero ad accedere alla risorsa contemporaneamente.

C ++ 11

`std::mutex` è l'implementazione di un mutex di C ++ 11.

```
#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish

    cout << var << endl; // prints 3
}
```

Leggi Gestione delle risorse online: https://riptutorial.com/it/cplusplus/topic/8336/gestione-delle-risorse

Capitolo 50: Idolo di Pimpl

Osservazioni

Il **linguaggio Pimpl (p** ointer a **IMPL** ementation, talvolta indicato come *opaco tecnica puntatore* o *Cheshire Cat),* riduce i tempi di compilazione di una classe spostando tutti i membri di dati privati in una struttura definita nel file cpp.

La classe possiede un puntatore all'implementazione. In questo modo, può essere inoltrato, in modo che il file di intestazione non abbia bisogno di `#include` classi utilizzate nelle variabili dei membri privati.

Quando si utilizza l'idioma pimpl, la modifica di un membro dati privato non richiede la ricompilazione di classi che dipendono da esso.

Examples

Idioma Pimpl di base

C ++ 11

Nel file di intestazione:

```
// widget.h

#include <memory>  // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    public:
        Widget();
        ~Widget();
        void DoSomething();

    private:
        // the pImpl idiom is named after the typical variable name used
        // ie, pImpl:
        struct Impl;                    // forward declaration
        std::experimental::propagate_const<std::unique_ptr< Impl >> pImpl;  // ptr to actual
implementation
};
```

Nel file di implementazione:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // no need to include this header inside widget.h
```

```
struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
    pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}
```

Il `pImpl` contiene lo stato del `Widget` (o alcuni / la maggior parte di esso). Invece della descrizione `Widget` dello stato esposto nel file di intestazione, può essere esposto solo all'interno dell'implementazione.

`pImpl` sta per "pointer to implementation". L'implementazione "reale" di `Widget` è in `pImpl` .

Pericolo: nota che per far funzionare questa `unique_ptr` con `unique_ptr` , `~Widget()` deve essere implementato in un punto in un file dove `Impl` è completamente visibile. È possibile `=default` lì, ma se `=default` dove `Impl` non è definito, il programma può facilmente diventare mal formato, non è richiesta alcuna diagnostica.

Leggi Idolo di Pimpl online: https://riptutorial.com/it/cplusplus/topic/2143/idolo-di-pimpl

Capitolo 51: Implementazione del modello di progettazione in C ++

introduzione

In questa pagina, puoi trovare esempi di come i modelli di design sono implementati in C ++. Per i dettagli su questi modelli, è possibile consultare la documentazione dei modelli di progettazione .

Osservazioni

Un modello di progettazione è una soluzione generale riutilizzabile per un problema comune in un dato contesto nella progettazione del software.

Examples

Modello di osservatore

L'intento di Observer Pattern è di definire una dipendenza uno-a-molti tra gli oggetti in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente.

Il soggetto e gli osservatori definiscono la relazione uno-a-molti. Gli osservatori dipendono dal soggetto in modo tale che quando lo stato del soggetto cambia, gli osservatori vengono avvisati. A seconda della notifica, gli osservatori possono anche essere aggiornati con nuovi valori.

Ecco l'esempio del libro "Design Patterns" di Gamma.

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

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
     virtual ~Subject() = default;
     void Attach(Observer& o) { observers.push_back(&o); }
     void Detach(Observer& o)
     {
         observers.erase(std::remove(observers.begin(), observers.end(), &o));
     }
```

```
     void Notify()
     {
         for (auto* o : observers) {
             o->Update(*this);
         }
     }
private:
     std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:

    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
        this->minute = minute;
        this->second = second;

        Notify();
    }

    int GetHour() const { return hour; }
    int GetMinute() const { return minute; }
    int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
     explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~DigitalClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }

     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Digital time is " << hour << ":"
                   << minute << ":"
                   << second << std::endl;
     }

private:
     ClockTimer& subject;
};

class AnalogClock: public Observer
```

```
{
public:
     explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~AnalogClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }
     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Analog time is " << hour << ":"
                   << minute << ":"
                   << second << std::endl;
     }
private:
     ClockTimer& subject;
};

int main()
{
    ClockTimer timer;

    DigitalClock digitalClock(timer);
    AnalogClock analogClock(timer);

    timer.SetTime(14, 41, 36);
}
```

Produzione:

```
Digital time is 14:41:36
Analog time is 14:41:36
```

Ecco il riepilogo del modello:

1. Gli oggetti (oggetto `DigitalClock` o `AnalogClock`) utilizzano le interfacce oggetto (`Attach()` o `Detach()`) per iscriversi (registrarsi) come osservatori o annullare l'iscrizione (rimuovere) stessi dall'essere osservatori (`subject.Attach(*this); subject.Detach(*this);`

2. Ogni soggetto può avere molti osservatori (osservatori `vector<Observer*> observers;`).

3. Tutti gli osservatori devono implementare l'interfaccia Observer. Questa interfaccia ha solo un metodo, `Update()` , che viene chiamato quando lo stato del soggetto cambia (`Update(Subject &)`)

4. Oltre ai metodi `Attach()` e `Detach()` , il soggetto concreto implementa un metodo `Notify()` che viene utilizzato per aggiornare tutti gli osservatori correnti ogni volta che si verificano cambiamenti di stato. Ma in questo caso, tutti sono fatti nella classe genitore, `Subject` (`Subject::Attach (Observer&)` , `void Subject::Detach(Observer&)` e `void Subject::Notify()` .

5. L'oggetto Concrete può anche avere metodi per impostare e ottenere il suo stato.

6. Gli osservatori concreti possono essere qualsiasi classe che implementa l'interfaccia Observer. Ogni osservatore si iscrive (registra) con un soggetto concreto per ricevere l'aggiornamento (`subject.Attach(*this);`).

7. I due oggetti di Observer Pattern sono **accoppiati liberamente** , possono interagire ma con poca conoscenza l'uno dell'altro.

Variazione:

Segnale e slot

I segnali e gli slot sono un costrutto linguistico introdotto in Qt, che semplifica l'implementazione del pattern Observer evitando il codice boilerplate. Il concetto è che i controlli (noti anche come widget) possono inviare segnali contenenti informazioni sugli eventi che possono essere ricevuti da altri controlli utilizzando funzioni speciali conosciute come slot. Lo slot di Qt deve essere un membro della classe dichiarato come tale. Il sistema segnale / slot si adatta perfettamente al modo in cui sono progettate le interfacce utente grafiche. Analogamente, il sistema segnale / slot può essere utilizzato per la notifica di eventi I / O asincroni (inclusi socket, pipe, dispositivi seriali, ecc.) O per associare eventi di timeout a istanze e metodi o funzioni appropriati. Non è necessario scrivere codice di registrazione / cancellazione / chiamata, poiché il Meta Object Compiler (MOC) di Qt genera automaticamente l'infrastruttura necessaria.

Il linguaggio C # supporta anche un costrutto simile sebbene con una terminologia e sintassi diverse: gli eventi svolgono il ruolo di segnali e i delegati sono gli slot. Inoltre, un delegato può essere una variabile locale, proprio come un puntatore a funzione, mentre uno slot in Qt deve essere un membro di classe dichiarato come tale.

Modello adattatore

Converti l'interfaccia di una classe in un'altra interfaccia che i clienti si aspettano. Adapter (o Wrapper) consente alle classi di funzionare insieme che non potrebbero altrimenti a causa di interfacce incompatibili. La motivazione del modello Adapter è che possiamo riutilizzare il software esistente se possiamo modificare l'interfaccia.

1. Il modello dell'adattatore si basa sulla composizione dell'oggetto.

2. Operazione di chiamate client su oggetto Adapter.

3. Adattatore chiama Adaptee per eseguire l'operazione.

4. In AWL, stack adattato dal vettore: quando stack esegue push (), il vettore sottostante vector vector: push_back ().

Esempio:

```
#include <iostream>

// Desired interface (Target)
```

```
class Rectangle
{
  public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
  public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
  private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
         std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};

int main()
{
  int x = 20, y = 50, w = 300, h = 200;
  Rectangle *r = new RectangleAdapter(x,y,w,h);
  r->draw();
}

//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)
```

Riassunto del codice:

1. Il cliente pensa di parlare con un `Rectangle`
2. L'obiettivo è la classe `Rectangle` . Questo è ciò su cui il client invoca il metodo.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
```

3. Si noti che la classe dell'adattatore utilizza l'ereditarietà multipla.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}
```

4. Adapter `RectangleAdapter` consente a `LegacyRectangle` rispondere alla richiesta (`draw()` su un `Rectangle`) ereditando le classi BOTH.

5. La classe `LegacyRectangle` non ha gli stessi metodi (`draw()`) come `Rectangle` , ma `Adapter(RectangleAdapter)` può prendere le chiamate del metodo `Rectangle` e girare e richiamare il metodo su `LegacyRectangle` , `oldDraw()` .

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

Il modello di progettazione **dell'adattatore** traduce l'interfaccia per una classe in un'interfaccia compatibile ma diversa. Quindi, questo è simile al pattern **proxy** in quanto è un wrapper a componente singolo. Ma l'interfaccia per la classe dell'adattatore e la classe originale potrebbe essere diversa.

Come abbiamo visto nell'esempio sopra, questo modello di **adattatore** è utile per esporre un'interfaccia diversa per un'API esistente per consentirne il funzionamento con altro codice. Inoltre, utilizzando il modello di adattatore, possiamo prendere interfacce eterogenee e trasformarle per fornire API coerenti.

Il bridge pattern ha una struttura simile ad un object adapter, ma Bridge ha un intento diverso: è pensato per **separare** un'interfaccia dalla sua implementazione in modo che possano essere variate facilmente e indipendentemente. Un **adattatore ha lo** scopo di **cambiare l'interfaccia** di un oggetto **esistente** .

Modello di fabbrica

Il modello di fabbrica disaccoppia la creazione dell'oggetto e consente la creazione per nome utilizzando un'interfaccia comune:

```
class Animal{
```

```
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string  getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};


class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string  getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string&   name )
    {
      if ( name == "bear" )
        return std::make_shared<Bear>();
      if ( name == "cat" )
        return std::shared_ptr<Cat>();

    return nullptr;
    }


};
```

Builder Pattern with Fluent API

The Builder Pattern disaccoppia la creazione dell'oggetto dall'oggetto stesso. L'idea principale dietro è che **un oggetto non deve essere responsabile della propria creazione** .
L'assemblaggio corretto e valido di un oggetto complesso può essere un'attività complessa di per sé, quindi questa attività può essere delegata a un'altra classe.

Ispirato da Email Builder in C # , ho deciso di creare una versione C ++ qui. Un oggetto di posta elettronica non è necessariamente un *oggetto molto complesso* , ma può dimostrare il modello.

```
#include <iostream>
#include <sstream>
```

```
#include <string>

using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
  public:
    friend class EmailBuilder;  // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
               << "\nto: " << m_to
               << "\nsubject: " << m_subject
               << "\nbody: " << m_body;
        return stream.str();
    }

  private:
    Email() = default; // restrict construction to builder

    string m_from;
    string m_to;
    string m_subject;
    string m_body;
};

class EmailBuilder
{
  public:
    EmailBuilder& from(const string &from) {
        m_email.m_from = from;
        return *this;
    }

    EmailBuilder& to(const string &to) {
        m_email.m_to = to;
        return *this;
    }

    EmailBuilder& subject(const string &subject) {
        m_email.m_subject = subject;
        return *this;
    }

    EmailBuilder& body(const string &body) {
        m_email.m_body = body;
        return *this;
    }

    operator Email&&() {
        return std::move(m_email); // notice the move
    }

  private:
    Email m_email;
```

```
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}


int main()
{
    Email mail = Email::make().from("me@mail.com")
                              .to("you@mail.com")
                              .subject("C++ builders")
                              .body("I like this API, don't you?");

    cout << mail << endl;
}
```

Per le versioni precedenti di C ++, si può semplicemente ignorare l'operazione `std::move` e rimuovere il && dall'operatore di conversione (anche se ciò creerà una copia temporanea).

Il costruttore termina il proprio lavoro quando rilascia l'email creata `operator Email&&()` . In questo esempio, il builder è un oggetto temporaneo e restituisce l'e-mail prima di essere distrutto. Puoi anche utilizzare un'operazione esplicita come `Email EmailBuilder::build() {...}` posto dell'operatore di conversione.

Passa il costruttore in giro

Un'ottima caratteristica di Builder Pattern è la possibilità di **utilizzare diversi attori per costruire un oggetto insieme.** Questo viene fatto passando il builder agli altri attori che daranno un po 'più di informazioni all'oggetto costruito. Questo è particolarmente potente quando si costruisce una sorta di query, aggiungendo filtri e altre specifiche.

```
void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
           .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
           .body("And the body. Someone else knows the addresses.");
}

int main()
{
    EmailBuilder builder;
```

```
    add_addresses(builder);
    compose_mail(builder);

    Email mail = builder;
    cout << mail << endl;
}
```

Variante di design: oggetto mutabile

È possibile modificare il design di questo modello in base alle proprie esigenze. Darò una variante.

Nell'esempio fornito l'oggetto Email è immutabile, cioè le sue proprietà non possono essere modificate perché non c'è accesso ad esse. Questa era una caratteristica desiderata. Se è necessario modificare l'oggetto dopo la sua creazione, è necessario fornire alcuni setter ad esso. Dal momento che questi setter verrebbero duplicati nel builder, potresti considerare di fare tutto in una classe (non è più necessaria alcuna classe di builder). Tuttavia, considererei la necessità di rendere l'oggetto costruito mutevole in primo luogo.

Leggi Implementazione del modello di progettazione in C ++ online: https://riptutorial.com/it/cplusplus/topic/4335/implementazione-del-modello-di-progettazione-in-c-plusplus

Capitolo 52: Inoltro perfetto

Osservazioni

L'inoltro perfetto richiede l' *inoltro dei riferimenti* per preservare i qualificatori di riferimento degli argomenti. Tali riferimenti appaiono solo in un *contesto dedotto* . Questo è:

```
template<class T>
void f(T&& x) // x is a forwarding reference, because T is deduced from a call to f()
{
    g(std::forward<T>(x)); // g() will receive an lvalue or an rvalue, depending on x
}
```

Quanto segue non implica un inoltro perfetto, perché `T` non è dedotto dalla chiamata del costruttore:

```
template<class T>
struct a
{
    a(T&& x); // x is a rvalue reference, not a forwarding reference
};
```

C ++ 17

C ++ 17 consentirà la deduzione degli argomenti del modello di classe. Il costruttore di "a" nell'esempio precedente diventerà un utente di un riferimento di inoltro

```
a example1(1);
  // same as a<int> example1(1);

int x = 1;
a example2(x);
  // same as a<int&> example2(x);
```

Examples

Funzioni di fabbrica

Supponiamo di voler scrivere una funzione factory che accetta un elenco arbitrario di argomenti e passa quegli argomenti non modificati ad un'altra funzione. Un esempio di tale funzione è `make_unique` , che viene utilizzato per costruire in modo sicuro una nuova istanza di `T` e restituire un `unique_ptr<T>` che possiede l'istanza.

Le regole linguistiche relative ai modelli variadici e ai riferimenti rvalue ci consentono di scrivere una funzione del genere.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
```

```
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

L'uso di ellissi `...` indica un pacchetto di parametri, che rappresenta un numero arbitrario di tipi. Il compilatore espande questo pacchetto di parametri al numero corretto di argomenti nel sito di chiamata. Questi argomenti vengono quindi passati al costruttore di `T` usando `std::forward`. Questa funzione è necessaria per conservare i qualificatori di ref degli argomenti.

```
struct foo
{
    foo() {}
    foo(const foo&) {}                    // copy constructor
    foo(foo&&) {}                         // copy constructor
    foo(int, int, int) {}
};

foo f;
auto p1 = make_unique<foo>(f);            // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Leggi Inoltro perfetto online: https://riptutorial.com/it/cplusplus/topic/1750/inoltro-perfetto

Capitolo 53: Input / output di base in c ++

Osservazioni

La libreria standard `<iostream>` definisce pochi flussi per input e output:

```
|stream | description                       |
|-------|-----------------------------------|
|cin    | standard input stream             |
|cout   | standard output stream            |
|cerr   | standard error (output) stream    |
|clog   | standard logging (output) stream  |
```

Su quattro flussi menzionati sopra, `cin` viene sostanzialmente utilizzato per l'input dell'utente e altri tre vengono utilizzati per l'output dei dati. In generale o nella maggior parte degli ambienti di codifica `cin` (*ingresso console* o *ingresso* standard) è tastiera e `cout` (*uscita console* o uscita standard) è monitor.

```
cin >> value

cin   - input stream
'>>'  - extraction operator
value - variable (destination)
```

`cin` qui estrae l'input immesso dall'utente e fornisce un valore variabile. Il valore viene estratto solo dopo che l'utente preme il tasto ENTER.

```
cout << "Enter a value: "

cout              - output stream
'<<'              - insertion operator
"Enter a value: " - string to be displayed
```

`cout` qui prende la stringa da visualizzare e la inserisce sullo standard output o sul monitor

Tutti e quattro i flussi si trovano nello spazio dei nomi standard `std` quindi è necessario stampare `std::stream` affinché lo stream `stream` lo usi.

C'è anche un manipolatore `std::endl` nel codice. Può essere utilizzato solo con flussi di output. Inserisce il carattere di fine riga `'\n'` nel flusso e lo svuota. Provoca immediatamente la produzione.

Examples

input dell'utente e output standard

```
#include <iostream>
```

```
int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Leggi Input / output di base in c ++ online: https://riptutorial.com/it/cplusplus/topic/10683/input---output-di-base-in-c-plusplus

Capitolo 54: Internazionalizzazione in C ++

Osservazioni

Il linguaggio C ++ non impone alcun set di caratteri, alcuni compilatori possono ***supportare*** l'uso di UTF-8 o anche UTF-16. Tuttavia non vi è la certezza che verrà fornito qualcosa oltre i semplici caratteri ANSI / ASCII.

Quindi tutto il supporto linguistico internazionale è definito dall'implementazione, dipende dalla piattaforma, dal sistema operativo e dal compilatore che si sta utilizzando.

Diverse librerie di terze parti (come la International Unicode Committee Library) che possono essere utilizzate per estendere il supporto internazionale della piattaforma.

Examples

Comprensione delle caratteristiche della stringa C ++

```
#include <iostream>
#include <string>

int main()
{
    const char * C_String = "This is a line of text w";
    const char * C_Problem_String = "This is a line of text ";
  std::string Std_String("This is a second line of text w");
    std::string Std_Problem_String("This is a second line of ex ");

  std::cout << "String Length: " << Std_String.length() << '\n';
    std::cout << "String Length: " << Std_Problem_String.length() << '\n';

    std::cout << "CString Length: " << strlen(C_String) << '\n';
    std::cout << "CString Length: " << strlen(C_Problem_String) << '\n';
    return 0;
}
```

A seconda della piattaforma (Windows, OSX, ecc.) E del compilatore (GCC, MSVC, ecc.), Questo programma **potrebbe non riuscire a compilare, visualizzare valori diversi o visualizzare gli stessi valori** .

Esempio di output sotto il compilatore Microsoft MSVC:

> Lunghezza della stringa: 31
> Lunghezza della stringa: 31
> Lunghezza CString: 24
> Lunghezza CString: 24

Ciò dimostra che sotto MSVC ogni carattere esteso utilizzato è considerato un singolo "carattere" e questa piattaforma supporta pienamente le lingue internazionalizzate.

Va notato tuttavia che questo comportamento è inusuale, questi caratteri internazionali sono memorizzati internamente come Unicode e quindi sono in realtà lunghi diversi byte. ***Ciò potrebbe causare errori imprevisti***

Sotto il compilatore GNC / GCC l'output del programma è:

> Lunghezza della stringa: 31
> Lunghezza della corda: 36
> Lunghezza CString: 24
> Lunghezza CString: 26

Questo esempio dimostra che mentre il compilatore GCC utilizzato su questa piattaforma (Linux) supporta questi caratteri estesi, utilizza anche (*correttamente)* diversi byte per memorizzare un singolo carattere.
In questo caso è possibile utilizzare caratteri Unicode, ma il programmatore deve prestare molta attenzione nel ricordare che la lunghezza di una "stringa" in questo scenario è la ***lunghezza in byte*** , non la ***lunghezza in caratteri leggibili*** .

Queste differenze sono dovute al modo in cui i linguaggi internazionali vengono gestiti su base per piattaforma e, cosa più importante, che le stringhe C e C ++ utilizzate in questo esempio possono essere considerate ***una matrice di byte*** , in modo che (per questo utilizzo) ***il linguaggio C ++ consideri un carattere (char) per essere un singolo byte*** .

Leggi Internazionalizzazione in C ++ online:
https://riptutorial.com/it/cplusplus/topic/5270/internazionalizzazione-in-c-plusplus

Capitolo 55: iteratori

Examples

C Iterators (Puntatori)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

Questo codice produrrebbe i numeri da 1 a 5, uno su ogni riga come questa:

1
2
3
4
5

Breaking It Down

```
const int array[] = { 1, 2, 3, 4, 5 };
```

Questa linea crea un nuovo array intero con 5 valori. Gli array C sono solo indicatori della memoria in cui ogni valore è memorizzato insieme in un blocco contiguo.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

Queste linee creano due puntatori. Il primo puntatore riceve il valore del puntatore dell'array, che è l'indirizzo del primo elemento dell'array. L'operatore `sizeof` quando usato su un array C restituisce

la dimensione dell'array in byte. Diviso per la dimensione di un elemento, questo dà il numero di elementi nella matrice. Possiamo usarlo per trovare l'indirizzo del blocco *dopo* l'array.

```
for (const int* i = first; i < afterLast; ++i) {
```

Qui creiamo un puntatore che utilizzeremo come un iteratore. E 'inizializzato con l'indirizzo del primo elemento che vogliamo iterare, e noi continueremo a iterare finché `i` è minore `afterLast` , il che significa che il tempo che `i` sta puntando a un indirizzo all'interno `array` .

```
    std::cout << *i << std::endl;
```

Infine, all'interno del loop possiamo accedere al valore che il nostro iteratore `i` sta indicando attraverso il dereferenziamento. Qui l'operatore dereferenzia `*` restituisce il valore all'indirizzo in `i` .

Panoramica

Gli iteratori sono posizioni

Gli iteratori sono un mezzo per navigare e operare su una sequenza di elementi e sono un'estensione generalizzata dei puntatori. Concettualmente è importante ricordare che gli iteratori sono posizioni, non elementi. Ad esempio, prendi la seguente sequenza:

```
A B C
```

La sequenza contiene tre elementi e quattro posizioni

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Gli elementi sono cose all'interno di una sequenza. Le posizioni sono luoghi in cui operazioni significative possono accadere alla sequenza. Ad esempio, uno si inserisce in una posizione, *prima* o *dopo l'* elemento A, non in un elemento. Anche la cancellazione di un elemento (`erase(A)`) viene effettuata individuando innanzitutto la sua posizione, quindi eliminandola.

Da Iterators ai valori

Per convertire da una posizione a un valore, un iteratore è *dereferenziato* :

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

Si può pensare a un iteratore come al dereferenziamento del valore a cui si riferisce nella sequenza. Ciò è particolarmente utile per capire perché non si dovrebbe mai dereferenziare l'iteratore di `end()` in una sequenza:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           +-- An iterator here has no value. Do not dereference it!
  +-------------- An iterator here dereferences to the value A.
```

In tutte le sequenze e i contenitori trovati nella libreria standard C ++, `begin()` restituirà un iteratore alla prima posizione e `end()` restituirà un iteratore a una passata l'ultima posizione (*non* l'ultima posizione!). Di conseguenza, i nomi di questi iteratori negli algoritmi sono spesso etichettati `first` e `last` :

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           |
  +- first    +- last
```

È anche possibile ottenere un iteratore per *qualsiasi sequenza* , perché anche una sequenza vuota contiene almeno una posizione:

```
+---+
|   |
+---+
```

In una sequenza vuota, `begin()` e `end()` saranno la stessa posizione, e *nessuno dei due* può essere dereferenziato:

```
+---+
|   |
+---+
  ↑
  |
  +- empty_sequence.begin()
  |
  +- empty_sequence.end()
```

La visualizzazione alternativa degli iteratori è che essi segnano le posizioni *tra gli* elementi:

```
+---+---+---+
| A | B | C |
+---+---+---+
↑   ^   ^   ↑
|           |
+- first    +- last
```

e la dereferenziazione di un iteratore restituisce un riferimento all'elemento che viene dopo l'iteratore. Alcune situazioni in cui questa visualizzazione è particolarmente utile sono:

- `insert` operazioni inserirà elementi nella posizione indicata dall'iteratore,
- `erase` operazioni restituirà un iteratore corrispondente alla stessa posizione di quello passato,

- un iteratore e il suo iteratore inverso corrispondente si trovano nella stessa posizione tra gli elementi

Iteratori non validi

Un iteratore viene *invalidato* se (ad esempio nel corso di un'operazione) la sua posizione non fa più parte di una sequenza. Un iteratore non valido non può essere cancellato fino a quando non è stato riassegnato a una posizione valida. Per esempio:

```
std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid
```

I numerosi algoritmi e le funzioni dei membri della sequenza nella libreria standard C ++ hanno regole che governano quando gli iteratori sono invalidati. Ogni algoritmo è diverso nel modo in cui tratta (e invalida) gli iteratori.

Navigazione con Iterators

Come sappiamo, gli iteratori sono per le sequenze di navigazione. Per fare ciò, un iteratore deve migrare la sua posizione per tutta la sequenza. Gli iteratori possono avanzare nella sequenza e alcuni possono avanzare all'indietro:

```
auto first = my_vector.begin();
++first;                                            // advance the iterator 1 position
std::advance(first, 1);                             // advance the iterator 1 position
first = std::next(first);                           // returns iterator to the next element
std::advance(first, -1);                            // advance the iterator 1 position
backwards
first = std::next(first, 20);                       // returns iterator to the element 20
position forward
first = std::prev(first, 5);                        // returns iterator to the element 5
position backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two
iterators.
```

Nota, il secondo argomento di std :: distance dovrebbe essere raggiungibile dal primo (o, in altre parole, `first` dovrebbe essere minore o uguale del `second`).

Anche se è possibile eseguire operatori aritmetici con iteratori, non tutte le operazioni sono definite per tutti i tipi di iteratori. `a = b + 3;` funzionerebbe per Iterator di accesso casuale, ma non funzionerebbe per gli iteratori di andata o bidirezionali, che possono ancora essere fatti avanzare di 3 posizioni con qualcosa come `b = a; ++b; ++b; ++b;` . Pertanto, si consiglia di utilizzare funzioni speciali nel caso in cui non si è sicuri di quale sia il tipo di iteratore (ad esempio, in una funzione modello che accetta iteratore).

Iterator Concepts

Lo standard C ++ descrive diversi concetti di iteratore. Questi sono raggruppati in base a come si comportano nelle sequenze a cui si riferiscono. Se si conosce il concetto di un *modello* iteratore (si comporta come), si può essere certi del comportamento di tale iteratore *indipendentemente dalla sequenza a cui appartiene* . Sono spesso descritti in ordine dal più piccolo al meno restrittivo (perché il prossimo concetto di iteratore è un passo migliore del suo predecessore):

- Input Iterators: può essere dereferenziato *solo una volta* per posizione. Può solo avanzare e solo una posizione alla volta.
- Iteratori di inoltro: un iteratore di input che può essere dereferenziato qualsiasi numero di volte.
- Iteratori bidirezionali: un iteratore diretto che può anche avanzare *all'indietro di* una posizione alla volta.
- Iteratori di accesso casuale: un iteratore bidirezionale che può avanzare avanti o indietro di un numero qualsiasi di posizioni alla volta.
- Iteratori contigui (dal C ++ 17): un iteratore di accesso casuale che garantisce che i dati sottostanti siano contigui in memoria.

Gli algoritmi possono variare a seconda del concetto modellato dagli iteratori a cui sono assegnati. Ad esempio, sebbene `random_shuffle` possa essere implementato per gli iteratori di inoltro, potrebbe essere fornita una variante più efficiente che richiede iteratori di accesso casuale.

Tratti iteratori

I tratti iteratori forniscono un'interfaccia uniforme alle proprietà degli iteratori. Permettono di recuperare valore, differenza, puntatore, tipi di riferimento e anche categoria di iteratore:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val)  {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

La categoria di iteratore può essere utilizzata per specializzare gli algoritmi:

```
template<class BidirIt>
void test(BidirIt a, std::bidirectional_iterator_tag)  {
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag)  {
    std::cout << "Forward iterator is used" << std::endl;
```

```
}

template<class Iter>
void test(Iter a)  {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Le categorie di iteratori sono fondamentalmente concetti di iteratori, ad eccezione degli Iterator contigui che non hanno il proprio tag, poiché è stato trovato che interrompe il codice.

Iteratori inversi

Se vogliamo scorrere a ritroso attraverso un elenco o un vettore, possiamo usare un `reverse_iterator` . Un iteratore inverso è costituito da un iteratore bidirezionale o di accesso casuale che mantiene come membro a cui è possibile accedere tramite `base()` .

Per eseguire iterazioni all'indietro, utilizzare `rbegin()` e `rend()` come gli iteratori rispettivamente per la fine della raccolta e l'inizio della raccolta.

Ad esempio, per iterare all'indietro utilizzare:

```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

Un iteratore inverso può essere convertito in un iteratore in avanti tramite la funzione membro di `base()` . La relazione è che l'iteratore inverso fa riferimento a un elemento dopo l'iteratore di `base()` :

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators

 +---+---+---+---+---+---+---+
 |   | 1 | 2 | 3 | 4 | 5 |   |
 +---+---+---+---+---+---+---+
   ↑   ↑               ↑   ↑
   |   |               |   |
rend() |         rbegin()  end()
       |                   rbegin().base()
     begin()
     rend().base()
```

Nella visualizzazione in cui gli iteratori segnano le posizioni tra gli elementi, la relazione è più semplice:

```
  +---+---+---+---+---+
  | 1 | 2 | 3 | 4 | 5 |
  +---+---+---+---+---+
  ↑                   ↑
```

```
  |                   |
  |                 end()
  |                 rbegin()
begin()             rbegin().base()
rend()
rend().base()
```

Iterator vettoriale

`begin` restituisce un `iterator` al primo elemento nel contenitore della sequenza.

`end` restituisce un `iterator` al primo elemento dopo la fine.

Se l'oggetto vettoriale è `const` , sia `begin` che `end` restituiscono un `const_iterator` . Se vuoi che venga restituito un `const_iterator` anche se il tuo vettore non è `const` , puoi usare `cbegin` e `cend` .

Esempio:

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

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };  //intialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Produzione:

1 2 3 4 5

Mappa Iterator

Un iteratore del primo elemento nel contenitore.

Se un oggetto mappa è const-qualificato, la funzione restituisce un `const_iterator` . In caso contrario, restituisce un `iterator` .

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;

// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Produzione:

a => 200
b => 100
c => 300

Stream Iterators

Gli iteratori di streaming sono utili quando è necessario leggere una sequenza o stampare dati formattati da un contenitore:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2     3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));

// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimeted by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

Il programma di esempio stamperà `1 -- 2 -- 3 -- 4 --` sullo standard output.

Scrivi il tuo iteratore supportato dal generatore

Un modello comune in altri linguaggi sta avendo una funzione che produce un "flusso" di oggetti, ed essendo in grado di usare il codice loop per passarci sopra.

Possiamo modellarlo in C ++ come

```
template<class T>
struct generator_iterator {
  using difference_type=std::ptrdiff_t;
  using value_type=T;
  using pointer=T*;
  using reference=T;
  using iterator_category=std::input_iterator_tag;
  std::optional<T> state;
  std::function< std::optional<T>() > operation;
  // we store the current element in "state" if we have one:
  T operator*() const {
    return *state;
  }
  // to advance, we invoke our operation.  If it returns a nullopt
  // we have reached the end:
  generator_iterator& operator++() {
    state = operation();
    return *this;
  }
  generator_iterator operator++(int) {
```

```
    auto r = *this;
    ++(*this);
    return r;
  }
  // generator iterators are only equal if they are both in the "end" state:
  friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
    if (!lhs.state && !rhs.state) return true;
    return false;
  }
  friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
    return !(lhs==rhs);
  }
  // We implicitly construct from a std::function with the right signature:
  generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
  {
    if (operation)
      state = operation();
  }
  // default all special member functions:
  generator_iterator( generator_iterator && ) =default;
  generator_iterator( generator_iterator const& ) =default;
  generator_iterator& operator=( generator_iterator && ) =default;
  generator_iterator& operator=( generator_iterator const& ) =default;
  generator_iterator() =default;
};
```

esempio dal vivo

Archiviamo l'elemento generato in anticipo in modo da poter rilevare più facilmente se siamo già alla fine.

Poiché la funzione di un iteratore del generatore finale non viene mai utilizzata, è possibile creare un intervallo di iteratori di generatore copiando solo la `std::function` una volta. Un iteratore generatore predefinito è paragonabile a se stesso ea tutti gli altri iteratori del generatore finale.

Leggi iteratori online: https://riptutorial.com/it/cplusplus/topic/473/iteratori

Capitolo 56: Iterazione

Examples

rompere

Salta dal ciclo di chiusura o dall'istruzione `switch` più vicina.

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

Continua

Salta alla fine del più piccolo circuito chiuso.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

fare

Introduce un ciclo di do-while .

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

per

Introduce un ciclo for o, in C ++ 11 e versioni successive, un ciclo for-based .

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
```

```
    putchar('*');
}
```

mentre

Introduce un ciclo while .

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

range-based per loop

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Leggi Iterazione online: https://riptutorial.com/it/cplusplus/topic/7841/iterazione

Capitolo 57: La regola del tre, cinque e zero

Examples

Regola del Cinque

C ++ 11

C ++ 11 introduce due nuove funzioni membro speciali: il costruttore di movimento e l'operatore di assegnazione del movimento. Per tutti gli stessi motivi per cui vuoi seguire la Regola dei Tre in C ++ 03, di solito vuoi seguire la Regola dei Cinque in C ++ 11: Se una classe richiede UNA delle cinque funzioni membro speciali, e se muove la semantica sono desiderati, quindi molto probabilmente richiede TUTTI i CINQUE di loro.

Nota, tuttavia, che non seguire la Regola dei Cinque non viene considerato un errore, ma un'opportunità di ottimizzazione persa, purché la Regola del Tre sia ancora seguita. Se nessun costruttore di movimento o operatore di assegnazione movimento è disponibile quando il compilatore normalmente ne usa uno, utilizzerà semantica di copia, se possibile, risultando in un'operazione meno efficiente a causa di operazioni di copia non necessarie. Se non si desidera spostare la semantica per una classe, non è necessario dichiarare un costruttore di movimenti o un operatore di assegnazione.

Lo stesso esempio della regola del tre:

```
class Person
{
    char* name;
    int age;

public:
    // Destructor
    ~Person() { delete [] name; }

    // Implement Copy Semantics
    Person(Person const& other)
        : name(new char[std::strlen(other.name) + 1])
        , age(other.age)
    {
        std::strcpy(name, other.name);
    }

    Person &operator=(Person const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Person copy(other);
        swap(*this, copy);
        return *this;
    }

    // Implement Move Semantics
    // Note: It is usually best to mark move operators as noexcept
    //       This allows certain optimizations in the standard library
```

```
    //       when the class is used in a container.

    Person(Person&& that) noexcept
        : name(nullptr)               // Set the state so we know it is undefined
        , age(0)
    {
        swap(*this, that);
    }

    Person& operator=(Person&& that) noexcept
    {
        swap(*this, that);
        return *this;
    }

    friend void swap(Person& lhs, Person& rhs) noexcept
    {
        std::swap(lhs.name, rhs.name);
        std::swap(lhs.age, rhs.age);
    }
};
```

In alternativa, sia l'operatore di assegnazione di copia che di spostamento può essere sostituito con un singolo operatore di assegnazione, che prende un'istanza in base al valore anziché al riferimento o al riferimento di rvalue per facilitare l'uso dell'idioma copy-and-swap.

```
Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}
```

Estendere dalla Regola dei Tre alla Regola dei Cinque è importante per i motivi di prestazione, ma nella maggior parte dei casi non è strettamente necessario. L'aggiunta del costruttore di copia e dell'operatore di assegnazione garantisce che lo spostamento del tipo non crei memoria (la costruzione delle mosse ricadrà semplicemente sulla copia in quel caso), ma eseguirà copie che il chiamante probabilmente non ha previsto.

Regola dello zero

C ++ 11

Possiamo combinare i principi della Rule of Five e RAII per ottenere un'interfaccia molto più snella: la regola dello zero: qualsiasi risorsa che deve essere gestita dovrebbe essere del suo tipo. Quel tipo dovrebbe seguire la Regola del Cinque, ma tutti gli utenti di quella risorsa non hanno bisogno di scrivere *nessuna* delle cinque funzioni membro speciali e possono semplicemente `default` tutte.

Utilizzando la classe `Person` introdotta nell'esempio della regola del tre , possiamo creare un oggetto di gestione delle risorse per le `cstrings` :

```
class cstring {
private:
```

```
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

E una volta che questo è separato, la nostra classe `Person` diventa molto più semplice:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

I membri speciali in `Person` non hanno nemmeno bisogno di essere dichiarati esplicitamente; il compilatore verrà predefinito o cancellato in modo appropriato, in base al contenuto di `Person` . Pertanto, il seguente è anche un esempio della regola zero.

```
struct Person {
    cstring name;
    int arg;
};
```

Se la `cstring` dovesse essere un tipo di solo spostamento, con un operatore di costruzione / assegnazione della copia di `delete` d, allora anche `Person` sposterebbe automaticamente.

Il termine regola zero è stato introdotto da R. Martinho Fernandes

Regola del tre

C ++ 03

La Regola dei tre stabilisce che se un tipo ha bisogno di avere un costruttore di copia definito dall'utente, un operatore di assegnazione copia o un distruttore, allora deve avere *tutti e tre* .

Il motivo della regola è che una classe che ha bisogno di uno dei tre gestisce alcune risorse (handle di file, memoria allocata dinamicamente, ecc.) E tutte e tre sono necessarie per gestire coerentemente tale risorsa. Le funzioni di copia riguardano il modo in cui la risorsa viene copiata tra gli oggetti e il distruttore distrugge la risorsa, in accordo con i principi RAII .

Considera un tipo che gestisce una risorsa stringa:

```
class Person
{
    char* name;
    int age;

public:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
       std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};
```

Poiché il `name` stato assegnato nel costruttore, il distruttore lo rilascia per evitare perdite di memoria. Ma cosa succede se un tale oggetto viene copiato?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

Innanzitutto, `p1` sarà costruito. Quindi `p2` verrà copiato da `p1` . Tuttavia, il costruttore di copie generato da C ++ copierà ogni componente del tipo così com'è. Il che significa che `p1.name` e `p2.name` puntano entrambi alla **stessa** stringa.

Quando `main` estremità, saranno chiamati distruttori. Il primo distruttore di `p2` sarà chiamato; cancellerà la stringa. Quindi verrà chiamato il distruttore di `p1` . Tuttavia, la stringa è *già stata eliminata* . Chiamare la `delete` sulla memoria che era già stata eliminata produce un comportamento indefinito.

Per evitare ciò, è necessario fornire un costruttore di copia adatto. Un approccio consiste nell'implementare un sistema di conteggio di riferimento, in cui diverse istanze di `Person` condividono gli stessi dati di stringa. Ogni volta che viene eseguita una copia, il conteggio dei riferimenti condivisi viene incrementato. Il distruttore decrementa quindi il conteggio dei riferimenti, rilasciando la memoria solo se il conteggio è zero.

Oppure potremmo implementare la semantica del valore e il comportamento di copia profonda :

```
Person(Person const& other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

Person &operator=(Person const& other)
```

```
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);            //  assume swap() exchanges contents of *this and copy
    return *this;
}
```

L'implementazione dell'operatore di assegnazione delle copie è complicata dalla necessità di rilasciare un buffer esistente. La tecnica di copia e swap crea un oggetto temporaneo che contiene un nuovo buffer. Scambiare il contenuto di `*this` e `copy` dà la proprietà alla `copy` del buffer originale. La distruzione della `copy` , al ripristino della funzione, rilascia il buffer precedentemente di proprietà di `*this` .

Protezione di autoassegnazione

Quando si scrive un operatore di assegnazione copia, è *molto* importante che sia in grado di lavorare in caso di autoassegnazione. Cioè, deve permettere questo:

```
SomeType t = ...;
t = t;
```

L'autoassegnazione di solito non avviene in modo così ovvio. In genere avviene tramite un percorso tortuoso attraverso vari sistemi di codice, in cui la posizione dell'assegnazione ha semplicemente due puntatori o riferimenti `Person` e non ha idea di essere lo stesso oggetto.

Qualsiasi operatore di assegnazione copia che scrivi deve essere in grado di tenerne conto.

Il modo tipico per farlo è quello di avvolgere tutta la logica di assegnazione in una condizione come questa:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Nota: è importante pensare all'assegnazione di sé e assicurarsi che il codice si comporti correttamente quando ciò accade. Tuttavia, l'autoassegnazione è un evento molto raro e l'ottimizzazione per evitare che possa effettivamente pessimizzare il caso normale. Dal momento che il caso normale è molto più comune, pessimizzare l'autoassegnazione può ridurre l'efficienza del codice (quindi fai attenzione ad usarlo).

Ad esempio, la normale tecnica per implementare l'operatore di assegnazione è l' `copy and swap idiom` . La normale implementazione di questa tecnica non si preoccupa di testare l'autoassegnazione (anche se l'autoassegnazione è costosa perché viene fatta una copia). La ragione è che la pessimizzazione del caso normale ha dimostrato di essere molto più costosa (come accade più spesso).

Spostare gli operatori di assegnazione deve anche essere protetto contro l'autoassegnazione. Tuttavia, la logica di molti di questi operatori è basata su `std::swap` , che può gestire lo scambio da / verso la stessa memoria. Quindi, se la tua logica di assegnazione del movimento non è altro che una serie di operazioni di scambio, allora non hai bisogno di una protezione per l'assegnazione automatica.

Se questo non è il tuo caso, *devi* prendere misure simili come sopra.

Leggi La regola del tre, cinque e zero online: https://riptutorial.com/it/cplusplus/topic/1206/la-regola-del-tre--cinque-e-zero

Capitolo 58: lambda

Sintassi

- [*default-capture* , *capture-list*] (*argomento-elenco*) *attributi* mutable *throw-specification* -> *return-type* { *lambda-body* } // Order of lambda specificatori e attributi.
- [*capture-list*] (*argument-list*) { *lambda-body* } // Definizione lambda comune.
- [=] (*argument-list*) { *lambda-body* } // Cattura tutte le variabili locali necessarie per valore.
- [&] (*argument-list*) { *lambda-body* } // Cattura tutte le variabili locali necessarie per riferimento.
- [*capture-list*] { *lambda-body* } // L'elenco degli argomenti e gli specificatori possono essere omessi.

Parametri

Parametro	Dettagli
default-capture	Specifica come vengono catturate tutte le variabili non elencate. Può essere `=` (acquisizione per valore) o `&` (acquisizione per riferimento). Se omesso, le variabili non elencate sono inaccessibili all'interno del *corpo lambda* . L' *acquisizione di default* deve precedere la *lista di cattura* .
capture-list	Specifica in che modo le variabili locali sono rese accessibili all'interno del *corpo lambda* . Le variabili senza prefisso vengono catturate dal valore. Le variabili con prefisso `&` vengono acquisite per riferimento. All'interno di un metodo di classe, `this` può essere usato per rendere tutti i suoi membri accessibili per riferimento. Le variabili non elencate sono inaccessibili, a meno che l'elenco non sia preceduto da *un'acquisizione predefinita* .
argument-list	Specifica gli argomenti della funzione lambda.
mutevole	*(facoltativo)* Normalmente le variabili acquisite dal valore sono `const` . La specifica di `mutable` li rende non-const. Le modifiche a tali variabili vengono mantenute tra le chiamate.
gettare-specifica	*(facoltativo)* Specifica il comportamento di lancio delle eccezioni della funzione lambda. Ad esempio: `noexcept` o `throw(std::exception)` .
attributi	*(opzionale)* Qualsiasi attributo per la funzione lambda. Ad esempio, se il *corpo lambda* genera sempre un'eccezione, allora `[[noreturn]]` può essere usato.
-> tipo di ritorno	*(opzionale)* Specifica il tipo di ritorno della funzione lambda. Obbligatorio quando il tipo di ritorno non può essere determinato dal compilatore.
lambda-	Un blocco di codice contenente l'implementazione della funzione lambda.

Parametro	Dettagli
body	

Osservazioni

C ++ 17 (la bozza attuale) introduce `constexpr` , fondamentalmente lambda che può essere valutato in fase di compilazione. Un lambda è automaticamente `constexpr` se soddisfa i requisiti di `constexpr` , ma è anche possibile specificarlo usando la parola chiave `constexpr` :

```
//Explicitly define this lambdas as constexpr
[]() constexpr {
    //Do stuff
}
```

Examples

Cos'è un'espressione lambda?

Un'espressione lambda fornisce un modo conciso per creare oggetti funzione semplici. Un'espressione lambda è un prvalore il cui oggetto risultato è chiamato oggetto chiusura , che si comporta come un oggetto funzione.

Il nome "espressione lambda" deriva dal lambda calcolo , che è un formalismo matematico inventato negli anni '30 dalla Chiesa di Alonzo per indagare sulle questioni relative alla logica e alla computabilità. Il calcolo Lambda ha costituito la base di LISP , un linguaggio di programmazione funzionale. Rispetto al lambda calcolo e al LISP, le espressioni lambda in C ++ condividono le proprietà di essere senza nome e di catturare variabili dal contesto circostante, ma non hanno la capacità di operare e restituire funzioni.

Un'espressione lambda viene spesso utilizzata come argomento per le funzioni che accettano un oggetto chiamabile. Ciò può essere più semplice della creazione di una funzione con nome, che verrebbe utilizzata solo quando passata come argomento. In tali casi, le espressioni lambda sono generalmente preferite perché consentono di definire gli oggetti funzione in linea.

Un lambda consiste tipicamente di tre parti: una lista di cattura `[]` , un elenco di parametri facoltativo `()` e un corpo `{}` , che possono essere tutti vuoti:

```
[](){}                // An empty lambda, which does and returns nothing
```

Elenco di acquisizione

`[]` è la **lista di cattura** . Per impostazione predefinita, non è possibile accedere alle variabili dello scope che racchiude un lambda. *Catturare* una variabile lo rende accessibile all'interno della lambda, sia come copia che come riferimento . Le variabili catturate diventano una parte del lambda; al contrario degli argomenti della funzione, non devono essere passati quando si chiama

il lambda.

```
int a = 0;                          // Define an integer variable
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                    //      Note: It is the responsibility of the programmer
                                    //      to ensure that a is not destroyed before the
                                    //      lambda is called.
auto b = f();                       // Call the lambda function. a is taken from the capture list
and not passed here.
```

Lista dei parametri

`()` è la **lista dei parametri** , che è quasi la stessa di quella delle normali funzioni. Se il lambda non accetta argomenti, queste parentesi possono essere omesse (eccetto se è necessario dichiarare il `mutable` lambda). Questi due lambda sono equivalenti:

```
auto call_foo  = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C ++ 14

L'elenco dei parametri può utilizzare il tipo di segnaposto `auto` posto dei tipi effettivi. In questo modo, questo argomento si comporta come un parametro di modello di un modello di funzione. I seguenti lambda sono equivalenti quando si desidera ordinare un vettore in codice generico:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
{ return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Corpo della funzione

`{}` è il **corpo** , che è lo stesso delle normali funzioni.

Chiamando un lambda

L'oggetto risultato di un'espressione lambda è una chiusura , che può essere chiamata usando l' `operator()` (come con altri oggetti funzione):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

Tipo di reso

Per impostazione predefinita, viene dedotto il tipo di ritorno di un'espressione lambda.

```
[](){ return true; };
```

In questo caso il tipo di `bool` è `bool` .

Puoi anche specificare manualmente il tipo di ritorno usando la seguente sintassi:

```
[]() -> bool { return true; };
```

Mutevole Lambda

Gli oggetti catturati dal valore nel lambda sono di default immutabili. Questo perché l' `operator()` dell'oggetto di chiusura generato è `const` per impostazione predefinita.

```
auto func = [c = 0](){++c; std::cout << c;};  // fails to compile because ++c
                                              // tries to mutate the state of
                                              // the lambda.
```

La modifica può essere consentita usando la parola chiave `mutable` , che rende non- `const` l' `operator()` dell'oggetto più vicino `operator()` :

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

Se usato insieme al tipo restituito, il `mutable` viene prima di esso.

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

Un esempio per illustrare l'utilità di lambda

Prima del C ++ 11:

C ++ 11

```
// Generic functor used for comparison
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

Dal momento che C ++ 11:

C ++ 11

```
// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value <
threshold; });
```

Specifica del tipo di reso

Per lambdas con una sola istruzione return, o più istruzioni return le cui espressioni sono dello stesso tipo, il compilatore può dedurre il tipo restituito:

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}
```

Per lambda con più dichiarazioni di ritorno di tipi *diversi* , il compilatore non può dedurre il tipo di reso:

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

In questo caso devi specificare esplicitamente il tipo di reso:

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

Le regole per questo corrispondono alle regole per `auto` deduzione del tipo `auto` . Lambdas senza tipi di ritorno specificati in modo esplicito non restituisce mai riferimenti, quindi se si desidera un tipo di riferimento deve essere specificato esplicitamente:

```
auto copy = [](X& x) { return x; };       // 'copy' returns an X, so copies its input
auto ref  = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Cattura in base al valore

Se si specifica il nome della variabile nella lista di cattura, lambda la catturerà per valore. Ciò significa che il tipo di chiusura generato per il lambda memorizza una copia della variabile. Ciò richiede anche che il tipo della variabile sia *copy-constructible* :

```
int a = 0;

[a]() {
    return a;   // Ok, 'a' is captured by value
};
```

C ++ 14

```
auto p = std::unique_ptr<T>(...);

[p]() {         // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

Da C ++ 14 in poi, è possibile inizializzare le variabili sul posto. Ciò consente di spostare solo i tipi da catturare nella lambda.

C ++ 14

```
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Anche se un lambda cattura le variabili in base al valore quando sono date dal loro nome, tali variabili non possono essere modificate all'interno del corpo lambda di default. Questo perché il tipo di chiusura inserisce il corpo lambda in una dichiarazione di `operator() const` .

Il `const` applica agli accessi alle variabili membro del tipo di chiusura e alle variabili catturate che sono membri della chiusura (tutte le apparenze in contrario):

```
int a = 0;

[a]() {
    a = 2;      // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

Per rimuovere il `const` , devi specificare la parola chiave `mutable` sul lambda:

```
int a = 0;

[a]() mutable {
    a = 2;      // OK, 'a' can be modified
```

```
    return a;
};
```

Poiché `a` stato catturato dal valore, qualsiasi modifica eseguita chiamando lambda non influirà su `a` . Il valore di `a` stato copiato nella lambda quando è stato costruito, in modo copia del lambda di `a` separato dall'esterno `a` variabile.

```
int a = 5 ;
auto plus5Val = [a] (void) { return a + 5 ; } ;
auto plus5Ref = [&a] (void) {return a + 5 ; } ;

a = 7 ;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref() ;
// The result will be "7, value 10, reference 12"
```

Acquisizione generalizzata

C ++ 14

Lambda può catturare espressioni, piuttosto che semplici variabili. Ciò consente a lambda di memorizzare i tipi di spostamento:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
  p->SomeFunc();
};
```

Ciò sposta la variabile `p` esterna nella variabile di cattura lambda, anche chiamata `p` . `lamb` ora possiede la memoria allocata da `make_unique` . Poiché la chiusura contiene un tipo che non può essere copiato, ciò significa che l' `lamb` è esso stesso non copiabile. Ma può essere spostato:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

Ora `lamb_move` possiede la memoria.

Nota che `std::function<>` richiede che i valori memorizzati siano copiabili. Puoi scrivere la tua `std::function` richiede solo lo spostamento , oppure puoi semplicemente inserire il lambda in un wrapper `shared_ptr` :

```
auto shared_lambda = [](auto&& f){
  return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
  (auto&&...args)->decltype(auto) {
    return (*spf)(decltype(args)(args)...);
  };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

prende il nostro lambda di sola mossa e carica il suo stato in un puntatore condiviso, quindi restituisce un lambda che *può* essere copiato e quindi memorizzato in una `std::function` o simile.

L'acquisizione generalizzata utilizza `auto` deduzione del tipo `auto` per il tipo di variabile. Dichiarerà queste acquisizioni come valori per impostazione predefinita, ma possono anche essere riferimenti:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
  v += add; //Modifies `a`
};

lamb(20); //`a` becomes 20.
```

Generalizzare l'acquisizione non ha bisogno di catturare una variabile esterna. Può catturare un'espressione arbitraria:

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

Questo è utile per dare a lambda valori arbitrari che possono contenere e potenzialmente modificare, senza doverli dichiarare esternamente al lambda. Naturalmente, ciò è utile solo se non si intende accedere a tali variabili dopo che lambda ha completato il proprio lavoro.

Cattura per riferimento

Se precedi il nome di una variabile locale con un `&` , allora la variabile verrà catturata per riferimento. Concettualmente, ciò significa che il tipo di chiusura di lambda avrà una variabile di riferimento, inizializzata come riferimento alla variabile corrispondente al di fuori dell'ambito del lambda. Qualsiasi utilizzo della variabile nel corpo lambda farà riferimento alla variabile originale:

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
assert(a == 1);
```

La parola chiave `mutable` non è necessaria, perché `a` stessa non è `const` .

Naturalmente, catturare per riferimento significa che il lambda **non deve** sfuggire allo scopo delle variabili che cattura. Quindi puoi chiamare funzioni che svolgono una funzione, ma non devi chiamare una funzione che *memorizzerà* il lambda oltre la portata dei tuoi riferimenti. E tu non devi

restituire il lambda.

Cattura predefinita

Per impostazione predefinita, non è possibile accedere alle variabili locali che non sono esplicitamente specificate nell'elenco di cattura dall'interno del corpo lambda. Tuttavia, è possibile acquisire implicitamente variabili nominate dal corpo lambda:

```
int a = 1;
int b = 2;

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

L'acquisizione esplicita può ancora essere eseguita insieme all'acquisizione implicita di default. La definizione di acquisizione esplicita sovrascriverà l'acquisizione predefinita:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Lambda generico

c ++ 14

Le funzioni Lambda possono assumere argomenti di tipo arbitrario. Ciò consente a un lambda di essere più generico:

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

Questo è implementato in C ++ facendo in modo che l' `operator()` del tipo di chiusura `operator()` sovraccarichi una funzione modello. Il seguente tipo ha un comportamento equivalente alla chiusura lambda sopra:

```
struct _unique_lambda_type
{
  template<typename T>
  auto operator() (T x) const {return x + x;}
};
```

Non tutti i parametri in un lambda generico devono essere generici:

```
[](auto x, int y) {return x + y;}
```

Qui, `x` è dedotto in base al primo argomento di funzione, mentre `y` sarà sempre `int` .

I lambda generici possono prendere argomenti come riferimento, usando le solite regole per `auto` e `&` . Se un parametro generico è preso come `auto&&` , questo è un riferimento di *inoltro* all'argomento passato e non un riferimento di *rvalue* :

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Le funzioni Lambda possono essere variadiche e inoltrare perfettamente i loro argomenti:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

o:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

che funziona solo "correttamente" con variabili di tipo `auto&&` .

Un motivo valido per utilizzare lambda generico è per la sintassi in visita.

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
  std::cout << e;
});
```

Qui stiamo visitando in modo polimorfico; ma in altri contesti, i nomi del tipo che stiamo passando non sono interessanti:

```
mutex_wrapped<std::ostream&> os = std::cout;
os.write([&](auto&& os){
  os << "hello world\n";
});
```

Ripetendo il tipo di `std::ostream&` is noise qui; sarebbe come dover menzionare il tipo di variabile ogni volta che la usi. Qui stiamo creando un visitatore, ma non uno polimorfico; `auto` è usato per lo stesso motivo per cui si può usare `auto` in un ciclo `for(:)` .

Conversione al puntatore della funzione

Se la lista di cattura di un lambda è vuota, allora lambda ha una conversione implicita in un puntatore a funzione che prende gli stessi argomenti e restituisce lo stesso tipo di ritorno:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};
```

```
using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

Tale conversione può anche essere applicata utilizzando un operatore unario più:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

La chiamata a questo puntatore di funzione si comporta esattamente come invocando `operator()` sul lambda. Questo puntatore a funzione non è in alcun modo dipendente dall'esistenza della chiusura lambda sorgente. Potrebbe quindi sopravvivere alla chiusura lambda.

Questa funzione è utile principalmente per l'utilizzo di lambda con API che trattano i puntatori di funzione, piuttosto che gli oggetti funzione C ++.

C ++ 14

La conversione a un puntatore di funzione è anche possibile per lambda generico con una lista di cattura vuota. Se necessario, verrà utilizzata la deduzione degli argomenti del modello per selezionare la specializzazione corretta.

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };
using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter;  // deduces int, int
// note however that the following is ambiguous
// func_ptr sorter_func2 = +sorter;
```

Classe lambda e cattura di questo

Un'espressione lambda valutata in una funzione membro della classe è implicitamente un amico di quella classe:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };

        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
};
```

Tale lambda non è solo un amico di quella classe, ha lo stesso accesso della classe dichiarata all'interno.

Lambda possono catturare `this` puntatore che rappresenta l'istanza oggetto la funzione esterno era chiamato. Questo viene fatto aggiungendo `this` alla lista di cattura:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

Quando `this` viene catturato, il lambda può usare i nomi dei membri della sua classe contenente come se fosse nella sua classe contenente. Quindi un implicito `this->` è applicato a tali membri.

Si noti che `this` viene catturato dal valore, ma non dal valore del tipo. Viene catturato dal valore di `this` , che è un *puntatore* . In quanto tale, il lambda non *possiede* `this` . Se il lambda out vive la vita dell'oggetto che lo ha creato, il lambda può diventare non valido.

Questo significa anche che la lambda può modificare `this` senza essere dichiarato `mutable` . È il puntatore che è `const` , non l'oggetto puntato. Cioè, a meno che la funzione membro esterno fosse di per sé una funzione `const` .

Inoltre, tieni presente che le clausole di acquisizione predefinite, sia `[=]` che `[&]` , cattureranno *anche* `this` implicitamente. E entrambi lo catturano per il valore del puntatore. In effetti, si tratta di un errore di specificare `this` nella lista di cattura quando viene dato un valore predefinito.

C ++ 17

Lambdas può catturare una copia di `this` oggetto, creata nel momento in cui viene creata la lambda. Questo viene fatto aggiungendo `*this` alla lista di cattura:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}
```

```
    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};
```

Portare funzioni lambda a C ++ 03 usando i funtori

Le funzioni Lambda in C ++ sono zucchero sintattico che fornisce una sintassi molto concisa per scrivere i funtori . Come tale, è possibile ottenere funzionalità equivalenti in C ++ 03 (anche se molto più prolisso) convertendo la funzione lambda in un functor:

```
// Some dummy types:
struct T1 {int dummy;};
struct T2 {int dummy;};
struct R {int dummy;};

// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
  // Use auto because the type of the lambda is unknown.
  auto lambda = [val, &ref](int arg1, int arg2) -> R {
    /* lambda-body */
    return R();
  };
  return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
  // Capture list.
  T1 val;
  T2& ref;

public:
  // Constructor
  inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

  // Functor body
  R operator()(int arg1, int arg2) const {
    /* lambda-body */
    return R();
  }
};

// Equivalent to use_lambda, but uses a functor (valid C++03).
R use_functor(T1 val, T2 ref) {
  Functor functor(val, ref);
  return functor(12, 27);
}

// Make this a self-contained example.
```

```
int main() {
  T1 t1;
  T2 t2;
  use_functor(t1,t2);
  use_lambda(t1,t2);
  return 0;
}
```

Se la funzione lambda è `mutable` rendi il call-operator del funtore non const, cioè:

```
R operator()(int arg1, int arg2) /*non-const*/ {
  /* lambda-body */
  return R();
}
```

Lambda ricorsivo

Diciamo che desideriamo scrivere Ecdid's `gcd()` come un lambda. Come una funzione, è:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Ma un lambda non può essere ricorsivo, non ha modo di invocare se stesso. Un lambda ha nome e utilizzando `this` all'interno del corpo di un lambda si riferisce a un catturato `this` (assumendo che il lambda viene creato nel corpo di una funzione membro, altrimenti è un errore). Quindi come risolviamo questo problema?

Usa la `std::function`

Possiamo avere un lambda per catturare un riferimento a una `std::function` non ancora costruita:

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

Funziona, ma dovrebbe essere usato con parsimonia. È lento (stiamo usando la cancellazione di tipo ora invece di una chiamata di funzione diretta), è fragile (copiare `gcd` o restituire `gcd` si interromperà poiché lambda si riferisce all'oggetto originale) e non funzionerà con lambdas generici.

Usando due puntatori intelligenti:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
*gcd_self = std::make_unique<std::function<int(int, int)>>(
  [gcd_self](int a, int b){
    return b == 0 ? a : (**gcd_self)(b, a%b);
  };
};
```

Ciò aggiunge un sacco di riferimenti indiretti (che sono generali), ma può essere copiato / restituito e tutte le copie condividono lo stato. Ti consente di restituire la lambda, ed è altrimenti meno fragile della soluzione di cui sopra.

Usa un combinatore a Y

Con l'aiuto di una breve struttura di utilità, possiamo risolvere tutti questi problemi:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // we pass ourselves to f, then the arguments.
        // the lambda should take the first argument as `auto&& recurse` or similar.
        return f(*this, std::forward<Args>(args)...);
    }
};
// helper function that deduces the type of the lambda:
template <class F>
y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
    return {std::forward<F>(f)};
}
// (Be aware that in C++17 we can do better than a `make_` function)
```

possiamo implementare il nostro `gcd` come:

```
auto gcd = make_y_combinator(
  [](auto&& gcd, int a, int b){
    return b == 0 ? a : gcd(b, a%b);
  }
);
```

`y_combinator` è un concetto `y_combinator` dal calcolo lambda che ti consente di avere una ricorsione senza essere in grado di `y_combinator` fino a quando non sei definito. Questo è esattamente il problema dei lambda.

Crei un lambda che prende "recurse" come primo argomento. Quando si desidera recurse, si passano gli argomenti per recurse.

`y_combinator` restituisce quindi un oggetto funzione che chiama tale funzione con i suoi argomenti, ma con un oggetto " `y_combinator` " adatto (ovvero lo `y_combinator` stesso) come primo argomento. `y_combinator` il resto degli argomenti che chiamate `y_combinator` con il lambda.

In breve:

```
auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
  // write body that processes some arguments
  // when you want to recurse, call recurse(some other arguments)
});
```

e hai ricorsione in una lambda senza restrizioni gravi o sovraccarico significativo.

Utilizzo di lambda per il disimballaggio del pacchetto di parametri inline

C ++ 14

Tradizionalmente, il disimballaggio del pacchetto di parametri richiede la scrittura di una funzione di supporto per ogni volta che si desidera eseguirla.

In questo esempio di giocattolo:

```
template<std::size_t...Is>
void print_indexes( std::index_sequence<Is...> ) {
  using discard=int[];
  (void)discard{0,((void)(
    std::cout << Is << '\n' // here Is is a compile-time constant.
  ),0)...};
}
template<std::size_t I>
void print_indexes_upto() {
  return print_indexes( std::make_index_sequence<I>{} );
}
```

`print_indexes_upto` vuole creare e decomprimere un pacchetto di parametri degli indici. Per fare ciò, deve chiamare una funzione di supporto. Ogni volta che si desidera decomprimere un pacchetto di parametri creato, si finisce per dover creare una funzione di supporto personalizzata per farlo.

Questo può essere evitato con lambda.

È possibile decomprimere i pacchetti di parametri in un set di invocazioni di un lambda, come questo:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    using discard=int[];
    (void)discard{0,(void(
      f( index<Is> )
    ),0)...};
  };
}

template<std::size_t N>
auto index_over(index_t<N> = {}) {
  return index_over( std::make_index_sequence<N>{} );
}
```

C ++ 17

`index_over()` con espressioni fold può essere semplificato per:

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    ((void)(f(index<Is>)), ...);
  };
}
```

Dopo averlo fatto, puoi usare questo per sostituire dover decomprimere manualmente i pacchetti di parametri con un secondo sovraccarico in altro codice, permettendoti di decomprimere i pacchetti di parametri "in linea":

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
  using T = std::remove_reference_t<Tup>;
  using std::tuple_size;
  auto from_zero_to_N = index_over< tuple_size<T>{} >();

  from_zero_to_N(
    [&](auto i){
      using std::get;
      f( get<i>( std::forward<Tup>(tup) ) );
    }
  );
}
```

L' `auto i` passato al lambda da `index_over` è uno `std::integral_constant<std::size_t, ???>` . Questo ha una conversione di `constexpr` in `std::size_t` che non dipende dallo stato di `this` , quindi possiamo usarlo come costante in fase di compilazione, come quando lo passiamo a `std::get<i>` sopra.

Per tornare all'esempio del giocattolo in alto, riscrivilo come:

```
template<std::size_t I>
void print_indexes_upto() {
  index_over(index<I>)([](auto i){
    std::cout << i << '\n'; // here i is a compile-time constant
  });
}
```

che è molto più breve e mantiene la logica nel codice che la usa.

Esempio dal vivo con cui giocare.

Leggi lambda online: https://riptutorial.com/it/cplusplus/topic/572/lambda

Capitolo 59: Layout dei tipi di oggetto

Osservazioni

Vedi anche Dimensione dei tipi interi .

Examples

Tipi di classe

Per "classe", intendiamo un tipo che è stato definito usando la parola chiave `class` o `struct` (ma non `enum struct` o `enum class`).

- Anche una classe vuota occupa ancora almeno un byte di spazio; consisterà quindi esclusivamente di imbottitura. Ciò garantisce che se `p` punta a un oggetto di una classe vuota, allora `p + 1` è un indirizzo distinto e punta a un oggetto distinto. Tuttavia, è possibile che una classe vuota abbia una dimensione pari a 0 quando viene utilizzata come classe base. Vedi l' ottimizzazione di base vuota .

```
class Empty_1 {};                               // sizeof(Empty_1)       == 1
class Empty_2 {};                               // sizeof(Empty_2)       == 1
class Derived : Empty_1 {};                     // sizeof(Derived)       == 1
class DoubleDerived : Empty_1, Empty_2 {};      // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; };                    // sizeof(Holder)        == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder)  == 2
class DerivedHolder : Empty_1 { Empty_1 e; };   // sizeof(DerivedHolder) == 2
```

- La rappresentazione dell'oggetto di un tipo di classe contiene le rappresentazioni di oggetti della classe base e dei tipi di membri non statici. Pertanto, ad esempio, nella seguente classe:

```
struct S {
    int x;
    char* y;
};
```

  c'è una sequenza consecutiva di `sizeof(int)` byte all'interno di un oggetto `S` , chiamato *subobject,* che contiene il valore di `x` , e un altro subobject con `sizeof(char*)` byte che contiene il valore di `y` . I due non possono essere intercalati.

- Se un tipo di classe ha membri e / o classi base con tipi `t1, t2,...tN` , la dimensione deve essere almeno `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` dati i punti precedenti . Tuttavia, a seconda dei requisiti di allineamento dei membri e delle classi di base, il compilatore può essere costretto a inserire il padding tra i sottooggetti, o all'inizio o alla fine dell'oggetto completo.

```
struct AnInt      { int i; };
```

```
  // sizeof(AnInt)        == sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(AnInt)        == 4 (4).
struct TwoInts    { int i, j; };
  // sizeof(TwoInts)      >= 2 * sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(TwoInts)      == 8 (4 + 4).
struct IntAndChar { int i; char c; };
  // sizeof(IntAndChar)   >= sizeof(int) + sizeof(char)
  // Assuming a typical 32- or 64-bit system, sizeof(IntAndChar)   == 8 (4 + 1 +
padding).
struct AnIntDerived : AnInt { long long l; };
  // sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
  // Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding +
8).
```

- Se il riempimento è inserito in un oggetto a causa dei requisiti di allineamento, la dimensione sarà maggiore della somma delle dimensioni dei membri e delle classi base. Con l'allineamento `n` byte, la dimensione sarà in genere il multiplo più piccolo di `n` che è più grande della dimensione di tutti i membri e le classi base. Ogni membro `memN` verrà tipicamente collocato in un indirizzo che è un multiplo di `alignof(memN)` , e `n` sarà tipicamente il più grande `alignof` di tutti gli `alignof` dei membri. A causa di ciò, se un membro con un `alignof` più `alignof` è seguito da un membro con un `alignof` più `alignof` , esiste la possibilità che quest'ultimo non sia allineato correttamente se posto immediatamente dopo il primo. In questo caso, il riempimento (noto anche come *membro di allineamento*) verrà posizionato tra i due membri, in modo che quest'ultimo possa avere l'allineamento desiderato. Viceversa, se un membro con un `alignof` più `alignof` è seguito da un membro con un `alignof` più `alignof` , di solito non sarà necessario alcun riempimento. Questo processo è anche noto come "imballaggio".
  A causa delle classi che in genere condividono l' `alignof` del loro membro con l' `alignof` più grande, le classi saranno in genere allineate con l' `alignof` del tipo più grande incorporato che contengono direttamente o indirettamente.

```
// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
  // sizeof(Char)                == 1 (sizeof(char))
struct Int  { int i; };
  // sizeof(Int)                 == 4 (sizeof(int))
struct CharInt { char c; int i; };
  // sizeof(CharInt)             == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
  // sizeof(ShortIntCharInt)     == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                    3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
  // sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                    1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
  // sizeof(ShortCharShortInt)   == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
  //                                    2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
  // sizeof(IntLLInt)            == 16 (4 (int) + 8 (long long) + 4 (int))
  // If packing isn't explicitly specified, most compilers will pack this as
  //   8-byte alignment, such that:
  // sizeof(IntLLInt)            == 24 (4 (int) + 4 (padding) + 8 (long long) +
  //                                    4 (int) + 4 (padding))
```

```
// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};
  // ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
  //                                             alignof(int) >= alignof(short)
  // sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
  //                                      2 (short) + 4 (int))
  // Note that t is placed at alignment of 2, not 4.  alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};
  // Large_1 has 4-byte alignment.
    // alignof(ShortIntCharInt) == alignof(int) == 4
    // alignof(b) == 1
    // Therefore, alignof(Large_1) == 4.
  // sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
  //                        16 (ShortIntCharInt))
struct Large_2 {
    IntLLInt illi;
    float f;
    IntLLInt jmmj;
};
  // Large_2 has 8-byte alignment.
    // alignof(IntLLInt) == alignof(long long) == 8
    // alignof(float) == 4
    // Therefore, alignof(Large_2) == 8.
  // sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))
```

C ++ 11

- Se l'allineamento rigoroso viene forzato con gli `alignas` , verrà utilizzato il riempimento per forzare il tipo a soddisfare l'allineamento specificato, anche quando altrimenti sarebbe più piccolo. Ad esempio, con la definizione seguente, `Chars<5>` avrà tre (o forse più) byte di riempimento inseriti alla fine in modo che la sua dimensione totale sia 8. Non è possibile per una classe con un allineamento di 4 avere una dimensione di 5 perché sarebbe impossibile creare una matrice di quella classe, quindi la dimensione deve essere "arrotondata" a un multiplo di 4 inserendo i byte di riempimento.

```
// This type shall always be aligned to a multiple of 4.  Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");
```

- Se due membri non statici di una classe hanno lo stesso identificatore di accesso , allora

quello che viene dopo in ordine di dichiarazione è garantito per venire successivamente nella rappresentazione dell'oggetto. Ma se due membri non statici hanno identificatori di accesso diversi, il loro ordine relativo all'interno dell'oggetto non è specificato.
- Non è specificato l'ordine in cui gli oggetti secondari della classe base vengono visualizzati all'interno di un oggetto, sia che si verifichino consecutivamente, sia se compaiono prima, dopo o tra sottooggetti membri.

Tipi aritmetici

Tipi di caratteri stretti

Il tipo di `unsigned char` utilizza tutti i bit per rappresentare un numero binario. Pertanto, ad esempio, se il `unsigned char` è lungo 8 bit, i 256 pattern di bit possibili di un oggetto `char` rappresentano i 256 diversi valori {0, 1, ..., 255}. Il numero 42 è garantito per essere rappresentato dal modello di bit `00101010` .

Il tipo di `signed char` non ha bit di riempimento, *cioè,* se il `signed char` è lungo 8 bit, allora ha 8 bit di capacità per rappresentare un numero.

Nota che queste garanzie non si applicano a tipi diversi dai tipi di caratteri stretti.

Tipi interi

I tipi interi senza segno usano un sistema binario puro, ma possono contenere bit di riempimento. Ad esempio, è possibile (anche se improbabile) che l' `unsigned int` abbia una lunghezza di 64 bit, ma sia in grado di memorizzare interi compresi tra 0 e 2^{32} - 1, inclusi. Gli altri 32 bit sarebbero i bit di riempimento, che non dovrebbero essere scritti direttamente.

I tipi interi con segno usano un sistema binario con un bit di segno ed eventualmente dei bit di riempimento. I valori che appartengono all'intervallo comune di un tipo di intero con segno e il tipo di intero senza segno corrispondente hanno la stessa rappresentazione. Ad esempio, se il modello di bit `0001010010101011` di un oggetto `unsigned short` rappresenta il valore `5291` , quindi rappresenta anche il valore `5291` quando viene interpretato come oggetto `short` .

È definito dall'implementazione se viene utilizzato il complemento a due, il complemento a un altro o la rappresentazione a livello di segno, poiché tutti e tre i sistemi soddisfano i requisiti del paragrafo precedente.

Tipi di virgola mobile

La rappresentazione del valore dei tipi a virgola mobile è definita dall'implementazione. Più comunemente, i tipi `float` e `double` sono conformi a IEEE 754 e sono lunghi 32 e 64 bit (quindi, ad esempio, `float` avrebbe 23 bit di precisione che seguiranno 8 bit esponenziali e 1 bit di segno). Tuttavia, lo standard non garantisce nulla. I tipi a virgola mobile hanno spesso "rappresentazioni di

trap", che causano errori quando vengono utilizzati nei calcoli.

Array

Un tipo di array non ha padding tra i suoi elementi. Pertanto, una matrice con elemento di tipo `T` è solo una sequenza di oggetti `T` disposti in memoria, nell'ordine.

Una matrice multidimensionale è una matrice di matrici e la suddetta si applica in modo ricorsivo. Ad esempio, se abbiamo la dichiarazione

```
int a[5][3];
```

quindi `a` è una matrice di 5 matrici di 3 `int` s. Pertanto, `a[0]` , che consiste dei tre elementi `a[0][0]` , `a[0][1]` , `a[0][2]` , è disposto in memoria prima di `a[1]` , che consiste di `a[1][0]` , `a[1][1]` e `a[1][2]` . Questo è chiamato ordine *principale di riga* .

Leggi Layout dei tipi di oggetto online: https://riptutorial.com/it/cplusplus/topic/9329/layout-dei-tipi-di-oggetto

Capitolo 60: letterali

introduzione

Tradizionalmente, un letterale è un'espressione che denota una costante il cui tipo e valore sono evidenti dalla sua ortografia. Ad esempio, `42` è un valore letterale, mentre `x` non lo è dato che uno deve vedere la sua dichiarazione per conoscerne il tipo e leggere le righe di codice precedenti per conoscerne il valore.

Tuttavia, C ++ 11 ha aggiunto anche valori letterali definiti dall'utente , che non sono letterali nel senso tradizionale, ma possono essere utilizzati come una scorciatoia per le chiamate di funzione.

Examples

vero

Una parola chiave che indica uno dei due possibili valori di tipo `bool` .

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

falso

Una parola chiave che indica uno dei due possibili valori di tipo `bool` .

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

nullptr

C ++ 11

Una parola chiave che denota una costante del puntatore nullo. Può essere convertito in qualsiasi puntatore o tipo puntatore-membro, ottenendo un puntatore nullo del tipo risultante.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Si noti che `nullptr` non è esso stesso un puntatore. Il tipo di `nullptr` è un tipo fondamentale noto come `std::nullptr_t` .

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

Questo

All'interno di una funzione di membro di una classe, la parola chiave `this` è un puntatore all'istanza della classe in cui è stata chiamata la funzione. `this` non può essere usato in una funzione membro statica.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

Il tipo di `this` dipende dalla qualifica cv della funzione membro: se `X::f` è `const` , il tipo di `this` all'interno di `f` è `const X*` , quindi `this` non può essere usato per modificare membri di dati non statici all'interno di un funzione membro `const` . Allo stesso modo, `this` eredita la qualifica `volatile` dalla funzione in cui appare.

C ++ 11

`this` può anche essere utilizzato in un *inizializzatore controvento o uguale* per un membro dati non statico.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` è un valore, quindi non può essere assegnato a.

Intero letterale

Un intero letterale è un'espressione primaria della forma

- decimale letterale

È una cifra decimale diversa da zero (1, 2, 3, 4, 5, 6, 7, 8, 9), seguita da zero o più cifre decimali (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- ottale letterale

È la cifra zero (0) seguita da zero o più cifre ottali (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- hex-letterale

È la sequenza di caratteri 0x o la sequenza di caratteri 0X seguita da una o più cifre esadecimali (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C , d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- binario-letterale (dal C ++ 14)

È la sequenza di caratteri 0b o la sequenza di caratteri 0B seguita da una o più cifre binarie (0, 1)

```
int b = 0b101010; // C++14
```

Il suffisso intero, se fornito, può contenere uno o entrambi i seguenti elementi (se sono forniti entrambi, possono apparire in qualsiasi ordine:

- suffisso senza segno (il carattere u o il carattere U)

```
unsigned int u_1 = 42u;
```

- lungo-suffisso (il carattere l del carattere L) o il suffisso lungo-lungo (la sequenza di caratteri ll o la sequenza di caratteri LL) (dal C ++ 11)

Anche le seguenti variabili sono inizializzate allo stesso valore:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Gli appunti

Le lettere nei valori letterali interi non fanno distinzione tra maiuscole e minuscole: 0xDeAdBaBeU e 0XdeadBABEu rappresentano lo stesso numero (un'eccezione è il suffisso long-long, che è ll o LL, mai lL o Ll)

Non ci sono letterali interi negativi. Espressioni come -1 applicano l'operatore unario meno al valore rappresentato dal valore letterale, che può implicare conversioni di tipo implicito.

In C precedenti a C99 (ma non in C ++), i valori decimali non troncati che non si adattano a int

lungo possono avere il tipo unsigned long int.

Quando usate in un'espressione di controllo di #if o #elif, tutte le costanti di interi con segno agiscono come se avessero tipo std :: intmax_t e tutte le costanti di interi non firmati agiscono come se avessero tipo std :: uintmax_t.

Leggi letterali online: https://riptutorial.com/it/cplusplus/topic/7836/letterali

Capitolo 61: Letterali definiti dall'utente

Examples

Valori letterali definiti dall'utente con valori double lunghi

```
#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}
```

L'output di questo programma è il seguente:

```
3 km = 3000 m
3 mi = 4828.03 m
```

Valori standard definiti dall'utente per la durata

C ++ 14

I seguenti valori letterali dell'utente di durata sono dichiarati nello `namespace` dei `namespace std::literals::chrono_literals` , dove sia `literals` che `chrono_literals` sono spazi dei nomi in linea . L'accesso a questi operatori può essere ottenuto con l' `using namespace std::literals` , `using namespace std::chrono_literals` e `using namespace std::literals::chrono_literals` .

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
    std::chrono::minutes t5 = 88min;
    auto t6 = 2 * 0.5h;
```

```
    auto total = t1 + t2 + t3 + t4 + t5 + t6;

    std::cout.precision(13);
    std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
              << " hours" << std::endl; // 2 hours
}
```

Letterali definiti dall'utente standard per le stringhe

C ++ 14

I seguenti valori letterali dell'utente di stringa sono dichiarati nello `namespace` dei `namespace` `std::literals::string_literals` , dove sia `literals` che `string_literals` sono spazi dei nomi in linea . L'accesso a questi operatori può essere acquisito `using namespace std::literals` , `using namespace std::string_literals` e `using namespace std::literals::string_literals` .

```
#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;

    std::cout << s << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
    std::cout << utf16conv.to_bytes(s16) << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
    std::cout << utf32conv.to_bytes(s32) << std::endl;

    std::wcout << ws << std::endl;
}
```

Nota:

La stringa letterale può contenere `\0`

```
std::string s1 = "foo\0\0bar";  // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle
```

Valori standard definiti dall'utente per complessi

C ++ 14

Quelli che seguono i letterali utente complessi sono dichiarati nello `namespace` dei `namespace std::literals::complex_literals` , dove sia `literals` che `complex_literals` sono spazi dei nomi in linea . L'accesso a questi operatori può essere ottenuto con l' `using namespace std::literals` , `using namespace std::complex_literals` e `using namespace std::literals::complex_literals` .

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;        // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;      // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl;   // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}
```

Valore letterale definito dall'utente self-made per binario

Nonostante tu possa scrivere un numero binario in C ++ 14 come:

```
int number =0b0001'0101; // ==21
```

ecco un esempio famoso con un'implementazione fatta da sé per i numeri binari:

Nota: l'intero programma di espansione dei modelli è in esecuzione in fase di compilazione.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" ) ;
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value  } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };


// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value ; }

// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value ; }

#include <iostream>

int main()
{
    std::cout  << 10101_B << ", " << 011011000111_b  << '\n' ; // prints 21, 1735
}
```

Leggi Letterali definiti dall'utente online: https://riptutorial.com/it/cplusplus/topic/2745/letterali-definiti-dall-utente

Capitolo 62: Lo standard ISO C ++

introduzione

Nel 1998, c'è stata una prima pubblicazione del C ++ standardizzato, un linguaggio standardizzato internamente. Da quel momento, C ++ si è evoluto risultando in diversi dialetti di C ++. In questa pagina, puoi trovare una panoramica di tutti i diversi standard e le loro modifiche rispetto alla versione precedente. I dettagli su come utilizzare queste funzionalità sono descritti su pagine più specializzate.

Osservazioni

Quando viene citato C ++, spesso viene fatto riferimento allo "Standard". Ma qual è esattamente quello standard?

C ++ ha una lunga storia. Iniziato come un piccolo progetto da Bjarne Stroustrup all'interno di Bell Labs, all'inizio degli anni '90 era diventato piuttosto popolare. Molte aziende stavano creando compilatori C ++ in modo che gli utenti potessero eseguire i propri compilatori C ++ su una vasta gamma di computer. Ma per facilitare questo, tutti questi compilatori concorrenti dovrebbero condividere una singola definizione della lingua.

A quel punto, il linguaggio C era stato standardizzato con successo. Ciò significa che è stata scritta una descrizione formale della lingua. Questo è stato presentato all'americana National Standards Institute (ANSI), che ha aperto il documento per la revisione e successivamente lo ha pubblicato nel 1989. Un anno dopo, l'Organizzazione internazionale per gli standard (poiché avrebbe sigle diverse in diverse lingue, ha scelto una forma , ISO, derivato dall'isos greco, che significa uguale) ha adottato lo standard americano come standard internazionale.

Per C ++, è stato chiaro fin dall'inizio che c'era un interesse internazionale. È stato avviato un gruppo di lavoro all'interno di ISO (noto come WG21, all'interno del sottocomitato 22). Questo gruppo di lavoro ha redatto un primo standard intorno al 1995. Ma come sappiamo i programmatori, non c'è nulla di più pericoloso per una consegna pianificata rispetto alle funzionalità dell'ultimo minuto, e ciò è accaduto anche al C ++. Nel 1995, una nuova fantastica libreria denominata STL emerse, e le persone che lavoravano nel WG21 decisero di aggiungere una versione ridotta allo standard di bozza del C ++. Naturalmente, questo ha fatto perdere le scadenze e solo 3 anni dopo il documento è diventato definitivo. ISO è un'organizzazione molto formale, quindi lo standard C ++ è stato battezzato con il nome non molto commerciabile di ISO / IEC 14882. Poiché gli standard possono essere aggiornati, questa versione esatta è diventata nota come 14882: 1998.

E infatti c'era una richiesta per aggiornare lo standard. Lo standard è un documento molto spesso, che mira a descrivere esattamente come i compilatori C ++ dovrebbero funzionare. Anche una leggera ambiguità può valere la pena di essere risolta, così nel 2003 è stato rilasciato un aggiornamento come 14882: 2003. Tuttavia, questo non ha aggiunto alcuna caratteristica al C ++; le nuove funzionalità sono state programmate per il secondo aggiornamento.

Informalmente, questo secondo aggiornamento era noto come C ++ 0x, perché non era noto se questo avrebbe richiesto fino al 2008 o al 2009. Bene, anche questa versione ha avuto un leggero ritardo, motivo per cui è diventato 14882: 2011.

Fortunatamente, il WG21 ha deciso di non permettere che accadesse di nuovo. C ++ 11 è stato ben accolto e lasciato ad un rinnovato interesse per C ++. Quindi, per mantenere lo slancio, il terzo aggiornamento è passato dalla pianificazione alla pubblicazione in 3 anni, fino a diventare 14882: 2014.

Anche il lavoro non si è fermato qui. È stato proposto lo standard C ++ 17 e il lavoro per C ++ 20 è stato avviato.

Examples

Bozze di lavoro correnti

Tutti gli standard ISO pubblicati sono disponibili per la vendita dallo store ISO (http://www.iso.org). Le bozze di lavoro degli standard C ++ sono pubblicamente disponibili gratuitamente.

Le diverse versioni dello standard:

- Prossimi (a volte indicati come C ++ 20 o C ++ 2a): bozza di lavoro corrente (versione HTML)
- Proposta (talvolta denominata C ++ 17 o C ++ 1z): bozza di lavoro N4659 di marzo 2017 .
- C ++ 14 (A volte indicato come C ++ 1y): bozza di lavoro di novembre 2014 N4296
- C ++ 11 (a volte indicato come C ++ 0x): bozza di lavoro N3242 di febbraio 2011
- C ++ 03
- C ++ 98

C ++ 11

Lo standard C ++ 11 è un'estensione importante dello standard C ++. Di seguito è possibile trovare una panoramica delle modifiche così come sono state raggruppate nelle FAQ di isocpp con collegamenti a documentazione più dettagliata.

Estensioni della lingua

Caratteristiche generali

- auto
- decltype
- Intervallo per la dichiarazione
- Elenchi di inizializzatori
- Sintassi e semantica di inizializzazione uniforme
- Rvalue i riferimenti e sposta la semantica

- lambda
- nox per impedire la propagazione delle eccezioni
- constexpr
- nullptr - un letterale del puntatore nullo
- Copiare e rilanciare le eccezioni
- Spazi dei nomi incorporati
- Letterali definiti dall'utente

Classi

- = predefinito e = cancella
- Controllo dello spostamento e della copia predefiniti
- Delegare i costruttori
- Inizializzatori dei membri in classe
- Costruttori ereditati
- Comandi di sostituzione: override
- Comandi di sostituzione: finale
- Operatori di conversione espliciti

Altri tipi

- classe enum
- lungo lungo - un numero intero più lungo
- Tipi interi estesi
- Sindacati generalizzati
- POD generalizzati

Modelli

- Modelli esterni
- Alias modello
- Modelli Variadic
- Tipi locali come argomenti del modello

Concorrenza

- Modello di memoria di concorrenza
- Inizializzazione e distruzione dinamiche con concurrency
- Memorizzazione locale del thread

Funzioni linguistiche varie

- Qual è il valore di __cplusplus per C ++ 11?
- Sintassi di tipo ritorno su suffisso

- Prevenire il restringimento
- Parentesi ad angolo retto
- static_assert assertion in fase di compilazione
- Letterali stringa grezzi
- attributi
- Allineamento
- Funzionalità C99

Estensioni della libreria

Generale

- unique_ptr
- shared_ptr
- weak_ptr
- Garbage collection ABI
- tuple
- Digita i tratti
- funzione e legatura
- Espressioni regolari
- Utilità temporali
- Generazione di numeri casuali
- Allocatori individuati

Contenitori e algoritmi

- Miglioramenti degli algoritmi
- Miglioramenti del contenitore
- contenitori non ordinati_ *
- std :: matrice
- forward_list

Concorrenza

- discussioni
- Esclusione reciproca
- serrature
- Variabili di condizione
- Atomics
- Futures e promesse
- async
- Abbandonare un processo

C ++ 14

Lo standard C ++ 14 viene spesso definito come un bugfix per C ++ 11. Contiene solo un elenco limitato di modifiche di cui la maggior parte sono estensioni alle nuove funzionalità in C ++ 11. Di seguito è possibile trovare una panoramica delle modifiche così come sono state raggruppate nelle FAQ di isocpp con collegamenti a documentazione più dettagliata.

Estensioni della lingua

- Letterali binari
- Deduzione del tipo di reso generalizzato
- decltype (auto)
- Cattura di lambda generalizzata
- Lambda generico
- Modelli variabili
- `constexpr` esteso
- L'attributo `[[deprecated]]`
- Separatori di cifre

Estensioni della libreria

- Blocco condiviso
- Valori letterali definiti dall'utente per `std::` types
- `std::make_unique`
- Tipo trasformazione `_t` alias
- Indirizzamento di tuple per tipo (ad es. `get<string>(t)`)
- Operatori operativi trasparenti (es. `greater<>(x)`)
- `std::quoted`

Deprecato / Rimosso

- `std::gets` stato deprecato in C ++ 11 e rimosso da C ++ 14
- `std::random_shuffle` è deprecato

C ++ 17

Lo standard C ++ 17 è completo ed è stato proposto per la standardizzazione. Nei compilatori con supporto sperimentale per queste funzionalità, viene solitamente indicato come C ++ 1z.

Estensioni della lingua

- Piega le espressioni
- dichiarando argomenti modello non di tipo con `auto`
- Copia elisione garantita

- Deduzione del parametro Template per costruttori
- Attacchi strutturati
- Spazi dei nomi nidificati compatti
- Nuovi attributi: `[[fallthrough]]` , `[[nodiscard]]` , `[[maybe_unused]]`
- Messaggio predefinito per `static_assert`
- Inizializzatori in `if` e `switch`
- Variabili in linea
- `if constexpr`
- Garanzie di valutazione dell'ordine di espressione
- Allocazione dinamica della memoria per dati sovra allineati

Estensioni della libreria

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` ed `extract()` per contenitori associativi
- Una libreria di file system con l'intestazione `<filesystem>` .
- Versioni parallele della maggior parte degli algoritmi standard (nell'intestazione `<algorithm>`).
- Aggiunta di funzioni matematiche speciali nell'intestazione `<cmath>` .
- Spostamento dei nodi tra la mappa <>, unordered_map <>, set <> e unordered_set <>

C ++ 03

Lo standard C ++ 03 riguarda principalmente i report sui difetti dello standard C ++ 98. Oltre a questi difetti, aggiunge solo una nuova funzionalità.

Estensioni della lingua

- Initalizzazione del valore

C ++ 98

C ++ 98 è la prima versione standardizzata di C ++. Poiché è stato sviluppato come un'estensione di C, vengono aggiunte molte delle funzionalità che separano il C ++ da C.

Estensioni della lingua (rispetto a C89 / C90)

- Classi, Classi derivate, funzioni membro virtuali, funzioni membro const
- Sovraccarico delle funzioni, sovraccarico dell'operatore
- Commenti a riga singola (È stato introdotto in C-languague con lo standard C99)
- Riferimenti
- nuovo e cancella
- tipo booleano (è stato introdotto nel C-languague con lo standard C99)

- modelli
- spazi dei nomi
- eccezioni
- cast specifici

Estensioni della libreria

- La libreria di modelli standard

C ++ 20

C ++ 20 è il prossimo standard di C ++, attualmente in fase di sviluppo, basato sullo standard C ++ 17. Il suo progresso può essere monitorato sul sito Web ISO cpp ufficiale .

Le seguenti caratteristiche sono semplicemente ciò che è stato accettato per la prossima versione dello standard C ++, mirato per il 2020.

Estensioni della lingua

Nessuna estensione di lingua è stata accettata per ora.

Estensioni della libreria

Per il momento non sono state accettate estensioni di libreria.

Leggi Lo standard ISO C ++ online: https://riptutorial.com/it/cplusplus/topic/2742/lo-standard-iso-c-plusplus

Capitolo 63: Loops

introduzione

Un'istruzione loop esegue ripetutamente un gruppo di istruzioni finché non viene soddisfatta una condizione. Esistono 3 tipi di loop primitivi in C ++: for, while, and do ... while.

Sintassi

- while (*condition*) *statement* ;
- fare una *dichiarazione* mentre (*espressione*);
- for (*for-init-statement* ; *condition* ; *expression*) *statement* ;
- per i *(per-gamma-dichiarazione: for-gamma-inizializzatore) istruzione;*
- rompere ;
- Continua ;

Osservazioni

`algorithm` chiamate agli `algorithm` sono generalmente preferibili ai cicli scritti a mano.

Se si desidera qualcosa che un algoritmo già fa (o qualcosa di molto simile), la chiamata all'algoritmo è più chiara, spesso più efficiente e meno soggetta a errori.

Se hai bisogno di un ciclo che faccia qualcosa di abbastanza semplice (ma richiederebbe un groviglio confuso di raccoglitori e adattatori se stai usando un algoritmo), scrivi semplicemente il ciclo.

Examples

Range-Based For

C ++ 11

`for` cicli `for` possono essere utilizzati per scorrere gli elementi di un intervallo basato su iteratore, senza utilizzare un indice numerico o accedere direttamente agli iteratori:

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

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

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

Questo itererà su ogni elemento in `v` , con `val` ottiene il valore dell'elemento corrente. La seguente

dichiarazione:

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

è equivalente a:

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

C ++ 17

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

Questa modifica è stata introdotta per il supporto pianificato di Ranges TS in C ++ 20.

In questo caso, il nostro ciclo è equivalente a:

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

Si noti che `auto val` dichiara un tipo di valore, che sarà una copia di un valore memorizzato nell'intervallo (lo stiamo inizializzando in copia dall'iteratore mentre procediamo). Se i valori memorizzati nell'intervallo sono costosi da copiare, è possibile utilizzare `const auto &val` . Inoltre, non è necessario utilizzare l' `auto` ; è possibile utilizzare un nome di tipo appropriato, purché sia implicitamente convertibile dal tipo di valore dell'intervallo.

Se hai bisogno di accedere all'iteratore, il range-based non può aiutarti (non senza alcuno sforzo, almeno).

Se desideri fare riferimento, puoi farlo:

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

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

È possibile eseguire l'iterazione sul riferimento `const` se si dispone di un contenitore `const` :

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

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

Uno userebbe i riferimenti di inoltro quando l'iteratore di sequenza restituisce un oggetto proxy ed è necessario operare su quell'oggetto in modo non `const` . Nota: molto probabilmente confonderà i lettori del tuo codice.

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

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

Il tipo "range" fornito alla gamma-base `for` può essere uno dei seguenti:

- Matrici di lingue:

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

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

  Si noti che l'assegnazione di un array dinamico non conta:

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

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

- Qualsiasi tipo che ha funzioni membro `begin()` e `end()` , che restituiscono gli iteratori agli elementi del tipo. I contenitori di libreria standard sono idonei, ma è possibile utilizzare anche i tipi definiti dall'utente:

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

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

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

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

- Qualsiasi tipo che ha funzioni di `begin(type)` e `end(type)` non membro che possono essere trovate tramite ricerca dipendente dall'argomento, in base al `type` . Questo è utile per creare un tipo di intervallo senza dover modificare il tipo di classe stesso:

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

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

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

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

Per ciclo

Un ciclo `for` esegue istruzioni nel `loop body` del `loop body` , mentre la `condition` del ciclo è vera. Prima che l' `initialization statement` del ciclo venga eseguita esattamente una volta. Dopo ogni ciclo, viene eseguita la parte di `iteration execution` .

Un ciclo `for` è definito come segue:

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

Spiegazione delle dichiarazioni del segnaposto:

- `initialization statement` : questa istruzione viene eseguita una sola volta, all'inizio del ciclo `for` . È possibile inserire una dichiarazione di più variabili di un tipo, ad esempio `int i = 0, a = 2, b = 3` . Queste variabili sono valide solo nell'ambito del ciclo. Le variabili definite prima del ciclo con lo stesso nome sono nascoste durante l'esecuzione del ciclo.
- `condition` : questa istruzione viene valutata prima di ogni esecuzione del *corpo del ciclo* e interrompe il ciclo se restituisce `false` .
- `iteration execution` : questa istruzione viene eseguita dopo il *corpo* del ciclo, prima della valutazione della *condizione* successiva, a meno che il ciclo `for` venga interrotto nel *corpo* (per `break` , `goto` , `return` o eccezione generata). È possibile immettere più istruzioni nella parte di `iteration execution` , ad esempio `a++, b+=10, c=b+a` .

L'equivalente approssimativo di un ciclo `for` , riscritto come un ciclo `while` è:

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

Il caso più comune per l'utilizzo di un ciclo `for` consiste nell'eseguire le istruzioni un numero specifico di volte. Ad esempio, considera quanto segue:

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

Un ciclo valido è anche:

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

Un esempio di nascondere le variabili dichiarate prima di un ciclo è:

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

Ma se vuoi usare la variabile già dichiarata e non nasconderla, ometti la parte della dichiarazione:

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

Gli appunti:

- Le istruzioni di inizializzazione e di incremento possono eseguire operazioni non correlate all'istruzione di condizione, o nulla del tutto - se si desidera farlo. Tuttavia, per motivi di leggibilità, è consigliabile eseguire solo operazioni direttamente pertinenti al ciclo.
- Una variabile dichiarata nell'istruzione di inizializzazione è visibile solo all'interno dell'ambito del ciclo `for` e viene rilasciata al termine del ciclo.
- Non dimenticare che la variabile che è stata dichiarata nella dichiarazione di `initialization statement` può essere modificata durante il ciclo, così come la variabile verificata nella `condition` .

Esempio di un ciclo che conta da 0 a 10:

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

Spiegazione dei frammenti di codice:

- `int counter = 0` inizializza il `counter` variabile su 0. (Questa variabile può essere utilizzata solo all'interno del ciclo `for` .)
- `counter <= 10` è una condizione booleana che controlla se il `counter` è minore o uguale a 10. Se è `true` , il ciclo viene eseguito. Se è `false` , il ciclo termina.
- `++counter` è un'operazione di incremento che incrementa il valore del `counter` di 1 prima del successivo controllo di condizione.

Lasciando tutte le istruzioni vuote, puoi creare un ciclo infinito:

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

L'equivalente del ciclo `while` di cui sopra è:

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

Tuttavia, un ciclo infinito può ancora essere lasciato usando le istruzioni `break` , `goto` o `return` o lanciando un'eccezione.

Il prossimo esempio comune di iterare su tutti gli elementi di una raccolta STL (es. Un `vector`) senza usare l'intestazione `<algorithm>` è:

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

Mentre loop

Un ciclo `while` esegue le istruzioni ripetutamente finché la condizione data non diventa `false` . Questa istruzione di controllo viene utilizzata quando non è noto, in anticipo, quante volte deve essere eseguito un blocco di codice.

Ad esempio, per stampare tutti i numeri da 0 a 9, è possibile utilizzare il seguente codice:

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

C ++ 17

Da notare che dal C ++ 17, le prime 2 affermazioni possono essere combinate

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

Per creare un ciclo infinito, è possibile utilizzare il seguente costrutto:

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

C'è un'altra variante dei cicli `while` , ovvero il `do...while` costrutto. Vedere l' esempio del ciclo di prova per ulteriori informazioni.

Dichiarazione di variabili in condizioni

Nella condizione dei cicli `for` e `while` , è anche consentito dichiarare un oggetto. Questo oggetto verrà considerato in ambito fino alla fine del ciclo e verrà mantenuto per ogni iterazione del ciclo:

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

for (auto& a : some_container) {
    a.do_something();
}
// a is no longer in scope.

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

Tuttavia, non è permesso fare lo stesso con un ciclo `do...while` ; invece, dichiarare la variabile prima del ciclo e (facoltativamente) racchiudere sia la variabile che il ciclo all'interno di un ambito locale se si desidera che la variabile vada fuori campo dopo la fine del ciclo:

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

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

Questo perché la porzione di *istruzione* di un ciclo `do...while` (il corpo del ciclo) viene valutata prima che venga raggiunta la porzione di *espressione* (il `while`) e, quindi, qualsiasi dichiarazione *nell'espressione* non sarà visibile durante la prima iterazione del ciclo continuo.

Ciclo Do-while

Un ciclo di *do-while* è molto simile ad un ciclo *while* , tranne per il fatto che la condizione viene verificata alla fine di ogni ciclo, non all'inizio. Il ciclo è quindi garantito per l'esecuzione almeno una volta.

Il codice seguente stamperà `0` , poiché la condizione verrà valutata su `false` alla fine della prima iterazione:

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

Nota: non dimenticare il punto e virgola alla fine del `while(condition);` , che è necessario nel costrutto *do-while* .

A differenza del ciclo *do-while* , il seguente non stampa nulla, perché la condizione restituisce `false` all'inizio della prima iterazione:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Nota: è possibile uscire da un ciclo *while* senza che la condizione diventi falsa utilizzando

un'istruzione `break` , `goto` o `return` .

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

Un semplice ciclo *do-while* viene utilizzato occasionalmente anche per scrivere macro che richiedono il proprio ambito (nel qual caso il punto e virgola finale viene omesso dalla definizione della macro e deve essere fornito dall'utente):

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

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

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

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

Istruzioni Loop Control: Break e Continue

Le istruzioni di controllo del ciclo vengono utilizzate per modificare il flusso di esecuzione dalla sua sequenza normale. Quando l'esecuzione lascia un ambito, tutti gli oggetti automatici creati in tale ambito vengono distrutti. Le `break` e le `continue` sono istruzioni di controllo del ciclo.

L'istruzione `break` termina un ciclo senza ulteriori considerazioni.

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

Il codice sopra verrà stampato:

```
1
2
3
```

L'istruzione `continue` non esce immediatamente dal ciclo, ma salta il resto del corpo del ciclo e passa all'inizio del ciclo (compreso il controllo della condizione).

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

Il codice sopra verrà stampato:

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

Poiché tali cambiamenti del flusso di controllo a volte sono difficili da comprendere per gli esseri umani, l' `break` e il `continue` vengono usati con parsimonia. L'implementazione più semplice di solito è più facile da leggere e capire. Ad esempio, il primo ciclo `for` con l' `break` sopra potrebbe essere riscritto come:

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

Il secondo esempio con `continue` potrebbe essere riscritto come:

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

Intervallo: per un sottogruppo

Utilizzando i loop di base-intervallo, è possibile eseguire il loop su una sotto-parte di un determinato contenitore o altro intervallo generando un oggetto proxy che si qualifica per loop basati su intervalli.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
  Iterator b;
  Sentinel e;
  Iterator begin() const { return b; }
  Sentinel end() const { return e; }
  bool empty() const { return begin()==end(); }
  range_t without_front( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {std::next(b, count), e};
```

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

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

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

ora possiamo fare:

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

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

e stampare

```
2
3
4
```

Tenere presente che gli oggetti intermedi generati nella parte `for(:range_expression)` del ciclo `for` saranno scaduti dal momento `for` inizia il ciclo `for` .

Leggi Loops online: https://riptutorial.com/it/cplusplus/topic/589/loops

Capitolo 64: Manipolatori di flusso

introduzione

I manipolatori sono speciali funzioni di supporto che aiutano a controllare i flussi di input e output usando l' `operator >>` o l' `operator <<` .

Tutti possono essere inclusi da `#include <iomanip>` .

Osservazioni

I manipolatori possono essere utilizzati in altro modo. Per esempio:

1. `os.width(n);` uguale a `os << std::setw(n);`
   `is.width(n);` è uguale a `is >> std::setw(n);`

2. `os.precision(n);` equivale a `os << std::setprecision(n);`
   `is.precision(n);` è uguale a `is >> std::setprecision(n);`

3. `os.setfill(c);` uguale a `os << std::setfill(c);`

4. `str >> std::setbase(base);` o `str << std::setbase(base);` uguale a

```
str.setf(base ==  8 ? std::ios_base::oct :
            base == 10 ? std::ios_base::dec :
                base == 16 ? std::ios_base::hex :
                     std::ios_base::fmtflags(0),
         std::ios_base::basefield);
```

5. `os.setf(std::ios_base::flag);` equivale a `os << std::flag;`
   `is.setf(std::ios_base::flag);` è uguale a `is >> std::flag;`
   `os.unsetf(std::ios_base::flag);` equivale a `os << std::no ## flag;`
   `is.unsetf(std::ios_base::flag);` è uguale a `is >> std::no ## flag;`
   (dove ## - è *operatore di concatenazione*)
   per la `flag` successiva s: **`boolalpha`** , **`showbase`** , **`showpoint`** , **`showpos`** , **`skipws`** , **`uppercase`** .

6. **`std::ios_base::basefield`** .
   Per `flag` s: **`dec`** , **`hex`** e **`oct`** :

- `os.setf(std::ios_base::flag, std::ios_base::basefield);` equivale a `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::basefield);` è uguale a `is >> std::flag;`
  (1)
- `str.unsetf(std::ios_base::flag, std::ios_base::basefield);` equivale a
  `str.setf(std::ios_base::fmtflags(0), std::ios_base::basefield);`
  (2)

7. **`std::ios_base::adjustfield`** .
   Per le `flag` : **`left`** , **`right`** e **`internal`** :

- `os.setf(std::ios_base::flag, std::ios_base::adjustfield);` equivale a `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::adjustfield);` è uguale a `is >> std::flag;`
  (1)
- `str.unsetf(std::ios_base::flag, std::ios_base::adjustfield);` equivale a
  `str.setf(std::ios_base::fmtflags(0), std::ios_base::adjustfield);`
  (2)

(1) Se il flag del campo corrispondente precedentemente impostato è già stato `unsetf` da `unsetf` .
(2) Se il `flag` è impostato.

8. **`std::ios_base::floatfield`** .

- `os.setf(std::ios_base::flag, std::ios_base::floatfield);` equivale a `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::floatfield);` è uguale a `is >> std::flag;`
  per `flag` s: **`fixed`** e **`scientific`** .
- `os.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` equivale a `os <<`
  `std::defaultfloat;`
  `is.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` è uguale a `is >>`
  `std::defaultfloat;`

9. `str.setf(std::ios_base::fmtflags(0), std::ios_base::flag);` equivale a
   `str.unsetf(std::ios_base::flag)`
   per `flag` s: **`basefield`** , campo di **`adjustfield`** , campo **`floatfield`** .

10. `os.setf(mask)` uguale a `os << setiosflags(mask);`
    `is.setf(mask)` uguale a `is >> setiosflags(mask);`
    `os.unsetf(mask)` uguale a `os << resetiosflags(mask);`
    `is.unsetf(mask)` uguale a `is >> resetiosflags(mask);`
    Per quasi tutte le `mask` di tipo `std::ios_base::fmtflags` .

Examples

Manipolatori di flusso

`std::boolalpha` e `std::noboolalpha` - `std::noboolalpha` rappresentazione testuale e numerica dei booleani.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` e `std::noshowbase` - controllano se viene usato il prefisso che indica la base numerica.

`std::dec` (decimale), `std::hex` (esadecimale) e `std::oct` (ottale) - sono usati per cambiare base per interi.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29  '\n';
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

I valori predefiniti sono `std::ios_base::noshowbase` e `std::ios_base::dec` .

Se vuoi vedere di più su `std::istringstream` controlla l'intestazione < sstream >.

`std::uppercase` `std::nouppercase` e `std::nouppercase` - controllano se i caratteri maiuscoli vengono utilizzati nell'output in virgola mobile e in esadecimale. Non ha alcun effetto sui flussi di input.

```
std::cout << std::hex << std::showbase
              << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
              << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

L'impostazione predefinita è `std::nouppercase` .

`std::setw(n)` - cambia la larghezza del prossimo campo di input / output esattamente a `n` .

La proprietà width `n` sta resettando a `0` quando vengono chiamate alcune funzioni (la lista completa è qui).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

L'impostazione predefinita è `std::setw(0)` .

`std::left` , `std::right` e `std::internal` - modifica la posizione di default dei caratteri di riempimento impostando `std::ios_base::adjustfield` a `std::ios_base::left` , `std::ios_base::right` e `std::ios_base::internal` corrispondente. `std::left` e `std::right` applicano a qualsiasi output, `std::internal` - per interi, floating point e monetari. Non ha alcun effetto sui flussi di input.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87**********
// hex: 41*************
//   $: $3.67**********
// usd: USD *3.67******
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
```

```
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: -**********9.87
// hex: *************41
//   $: $3.67**********
// usd: USD *******3.67
// usd: USD        3.67

std::cout << std::right << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: **********-9.87
// hex: *************41
//   $: **********$3.67
// usd: ******USD *3.67
// usd:       USD  3.67
```

L'impostazione predefinita è `std::left` .

`std::fixed` , `std::scientific` , `std::hexfloat` [C ++ 11] e `std::defaultfloat` [C ++ 11] - cambia la formattazione per l'input / output a virgola mobile.

`std::fixed` imposta lo `std::ios_base::floatfield` a `std::ios_base::fixed` ,
`std::scientific` - to `std::ios_base::scientific` ,
`std::hexfloat` - a `std::ios_base::fixed | std::ios_base::scientific` e
`std::defaultfloat` - to `std::ios_base::fmtflags(0)` .

`fmtflags`

```
#include <sstream>
...

std::cout << '\n'
          << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
          << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
          << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'
          << "The number 0.07 in default:    " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
// The number 0.01 in scientific: 7.000000e-02
```

```
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308
```

Il valore predefinito è `std::ios_base::fmtflags(0)` .

C'è un **bug** su alcuni compilatori che causa

```
double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0
```

`std::showpoint` e `std::noshowpoint` - controllano se il punto decimale è sempre incluso nella rappresentazione in virgola mobile. Non ha alcun effetto sui flussi di input.

```
std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
          << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7
```

L'impostazione predefinita è `std::showpoint` .

`std::showpos` e `std::noshowpos` - visualizzazione del controllo del segno `+` nell'output *non negativo* . Non ha alcun effetto sui flussi di input.

```
std::cout << "With showpos: " << std::showpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
          << "Without showpos: " << std::noshowpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17
```

Predefinito se `std::noshowpos` .

`std::unitbuf` , `std::nounitbuf` - controlla il flusso di output di flussaggio dopo ogni operazione. Non ha alcun effetto sul flusso di input. `std::unitbuf` provoca il flushing.

`std::setbase(base)` - imposta la base numerica del flusso.

`std::setbase(8)` equivale a impostare `std::ios_base::basefield` a `std::ios_base::oct` ,
`std::setbase(16)` - a `std::ios_base::hex` ,
`std::setbase(10)` - a `std::ios_base::dec` .

Se la `base` è diversa da `8` , `10` o `16` allora `std::ios_base::basefield` è impostato su `std::ios_base::fmtflags(0)` . Significa output decimale e input dipendente dal prefisso.

Come predefinito, `std::ios_base::basefield` è `std::ios_base::dec` quindi di default `std::setbase(10)` .

`std::setprecision(n)` - cambia la precisione in virgola mobile.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
const long double pi = std::acos(-1.L);

std::cout << '\n'
          << "default precision (6):   pi: " << pi << '\n'
          << "                        10pi: " << 10 * pi << '\n'
          << "std::setprecision(4):  10pi: " << std::setprecision(4) << 10 * pi << '\n'
          << "                    10000pi: " << 10000 * pi << '\n'
          << "std::fixed:         10000pi: " << std::fixed << 10000 * pi << std::defaultfloat
<< '\n'
          << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
          << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi <<
'\n'
          << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi <<
'\n'
          << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                         10pi: 31.4159
// std::setprecision(4):  10pi: 31.42
//                     10000pi: 3.142e+04
// std::fixed:         10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi:
3.1415926535897932385128089594061862044327426701784133911132812 5
// max+1 radix precision:   pi:
3.1415926535897932385128089594061862044327426701784133911132812 5
// significant digits prec: pi: 3.14159265358979324
```

L'impostazione predefinita è `std::setprecision(6)` .

`std::setiosflags(mask)` e `std::resetiosflags(mask)` - imposta e cancella i flag specificati nella `mask` di `std::ios_base::fmtflags` type.

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
```

```
std::cout << "Parsing \"10 010\" with std::oct gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                              std::ios_base::uppercase |
                              std::ios_base::showbase) << 42 << '\n';
// Output: OX2A
```

`std::skipws` e `std::noskipws` : il controllo saltato degli spazi bianchi `std::noskipws` dalle funzioni di input formattate. Non ha alcun effetto sui flussi di output.

```
#include <sstream>
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default  behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';
// Output: Default  behavior:  c1 = a  c2 = b  c3 = c
// noskipws behavior:  c1 = a  c2 =    c3 = b
```

Il valore predefinito è `std::ios_base::skipws` .

`std::quoted(s[, delim[, escape]])` [C ++ 14] - inserisce o estrae stringhe tra virgolette con spazi incorporati.

`s` - la stringa da inserire o estrarre.
`delim` : il carattere da utilizzare come delimitatore `"` , per impostazione predefinita.
`escape` - il carattere da utilizzare come carattere di escape, `\` per impostazione predefinita.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in     [" << in << "]\n"
          << "stored as   [" << ss.str() << "]\n";

ss >> std::quoted(out);
```

```
std::cout << "written out [" << out << "]\n";
// Output:
// read in     [String with spaces, and embedded "quotes" too]
// stored as   ["String with spaces, and embedded \"quotes\" too"]
// written out [String with spaces, and embedded "quotes" too]
```

Per maggiori informazioni vedi il link sopra.

Manipolatori del flusso di uscita

`std::ends` - inserisce un carattere null `'\0'` nel flusso di output. Più formalmente la dichiarazione di questo manipolatore è simile

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

e questo manipolatore posiziona il carattere chiamando `os.put(charT())` quando viene usato in un'espressione
`os << std::ends;`

`std::endl` e `std::flush` sia il flusso di output in `out` chiamando `out.flush()` . Provoca immediatamente la produzione. Ma `std::endl` inserisce il simbolo di fine riga `'\n'` prima dello svuotamento.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
// Output: First line.
// Second line. Still second line.
```

`std::setfill(c)` - cambia il carattere di riempimento in `c` . Spesso usato con `std::setw` .

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:         79
// setfill('#'): ########79
```

`std::put_money(mon[, intl])` [C ++ 11]. In un'espressione `out << std::put_money(mon, intl)` , converte il valore monetario `mon` (di `long double` o `std::basic_string` type) nella sua rappresentazione di carattere come specificato dal facet `std::money_put` della locale attualmente imbevuta `out` . Usa le stringhe di valuta internazionali se `intl` è `true` , usa altrimenti i simboli di valuta.

```
long double money = 123.45;
```

```
// or std::string money = "123.45";

std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD  1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ￥123 or JPY  123
```

`std::put_time(tmb, fmt)` [C ++ 11] - formatta ed emette un valore di data / ora per `std::tm` secondo il formato specificato `fmt` .

`tmb` - puntatore alla struttura del tempo del calendario `const std::tm*` come ottenuto da `localtime()` o `gmtime()` .
`fmt` - puntatore a una stringa `const CharT*` null-terminata `const CharT*` specifica il formato di conversione.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

Per maggiori informazioni vedi il link sopra.

Manipolatori del flusso di input

`std::ws` - consuma spazi bianchi iniziali nel flusso di input. È diverso da `std::skipws` .

```
#include <sstream>
...

std::string str;
std::istringstream("  \v\n\r\t    Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C ++ 11]. In un'espressione `in >> std::get_money(mon, intl)` analizza

l'input di carattere come valore monetario, come specificato dal `std::money_get` della locale attualmente `std::money_get in` , e memorizza il valore in `mon` (di `long double` o `std::basic_string`). Manipulator prevede le stringhe di valuta internazionali *richieste* se `intl` è `true` , si aspetta altrimenti simboli di valuta *facoltativi* .

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C ++ 11] - analizza un valore di data / ora memorizzato in `tmb` del formato specificato `fmt` .

`tmb` - puntatore valido all'oggetto `const std::tm*` cui verrà archiviato il risultato.
`fmt` - puntatore a una stringa di stringa null-terminata `const CharT*` specifica il formato di conversione.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

Per maggiori informazioni vedi il link sopra.

Leggi Manipolatori di flusso online: https://riptutorial.com/it/cplusplus/topic/10699/manipolatori-di-flusso

Capitolo 65: Manipolazione bit

Osservazioni

Per usare `std::bitset` dovrai includere l' intestazione `<bitset>` .

```
#include <bitset>
```

`std::bitset` sovraccarica tutte le funzioni dell'operatore per consentire lo stesso utilizzo della gestione in stile c dei bitset.

Riferimenti

- Bit Twiding Hack

Examples

Impostazione un po '

Manipolazione bit in stile C.

Un bit può essere impostato usando l'operatore OR bit a bit (`|`).

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

Utilizzando std :: bitset

`set(x)` o `set(x,true)` - imposta il bit in posizione `x` su `1` .

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

Schiarirsi un po '

Manipolazione bit in stile C.

Un bit può essere cancellato usando l'operatore AND bit per bit (`&`).

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

Utilizzando std :: bitset

`reset(x)` o `set(x,false)` - cancella il bit in posizione `x` .

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

Toggling un po '

Manipolazione bit in stile C.

Un bit può essere commutato usando l'operatore XOR (`^`).

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

Utilizzando std :: bitset

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

Controllando un po '

Manipolazione bit in stile C.

Il valore del bit può essere ottenuto spostando il numero a `x` volte di destra e quindi eseguendo il bit AND (`&`) su di esso:

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

L'operazione di spostamento a destra può essere implementata come uno spostamento aritmetico (firmato) o uno spostamento logico (senza segno). Se il `number` nel `number` dell'espressione `number >> x` ha un tipo firmato e un valore negativo, il valore risultante è definito dall'implementazione.

Se abbiamo bisogno del valore di quel bit direttamente sul posto, potremmo invece spostare la maschera:

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

Entrambi possono essere usati come condizionali, poiché tutti i valori diversi da zero sono considerati veri.

Utilizzando std :: bitset

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

Cambiare l'ennesimo bit in x

Manipolazione bit in stile C.

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

Utilizzando std :: bitset

`set(n,val)` - imposta il bit `n` sul valore `val` .

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

Imposta tutti i bit

Manipolazione bit in stile C.

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

(Vedi qui per una spiegazione del perché questo funziona ed è in realtà l'approccio migliore).

Utilizzando std :: bitset

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

Rimuovi il bit impostato più a destra

Manipolazione bit in stile C.

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

Spiegazione

- se `n` è zero, abbiamo `0 & 0xFF..FF` che è zero
- else `n` può essere scritto `0bxxxxxx10..00` e `n - 1` è `0bxxxxxx011..11` , quindi `n & (n - 1)` è `0bxxxxxx000..00` .

Set di bit di conteggio

Il conteggio della popolazione di un bitstring è spesso necessario in crittografia e altre applicazioni e il problema è stato ampiamente studiato.

Il modo ingenuo richiede un'iterazione per bit:

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

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

Un bel trucco (basato su Rimuovi bit più a destra) è:

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

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

Passa attraverso tante iterazioni quanti sono i bit impostati, quindi è buono quando si prevede che il `value` abbia pochi bit diversi da zero.

Il metodo fu proposto per la prima volta da Peter Wegner (in CACM 3/322 - 1960) ed è ben noto poiché compare in *C Programming Language* di Brian W. Kernighan e Dennis M. Ritchie.

Ciò richiede 12 operazioni aritmetiche, una delle quali è una multicrizione:

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

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

Questo tipo di implementazione ha il miglior comportamento nel caso peggiore (vedi il peso di Hamming per ulteriori dettagli).

Molte CPU hanno un'istruzione specifica (come il `popcnt` di x86) e il compilatore può offrire una funzione specifica (**non standard**) incorporata. Ad esempio con g ++ c'è:

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

Controlla se un numero intero è una potenza di 2

Il trucco `n & (n - 1)` (vedi Rimuovi il bit più a destra) è anche utile per determinare se un intero è una potenza di 2:

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

Si noti che senza la prima parte del controllo (`n &&`), `0` viene erroneamente considerato una potenza di 2.

Applicazione di manipolazione di bit: lettera da piccola a maiuscola

Una delle numerose applicazioni di manipolazione di bit è la conversione di una lettera da piccola a maiuscola o viceversa scegliendo una **maschera** e un'operazione **bit** appropriata. Ad esempio, la **lettera** ha questa rappresentazione binaria `01(1)00001` mentre la sua controparte capitale ha `01(0)00001` . Si differenziano unicamente tra parentesi nel bit. In questo caso, la conversione di **una** lettera da piccola a maiuscola è fondamentalmente l'impostazione del bit tra parentesi a uno. Per fare ciò, facciamo quanto segue:

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

Il codice per convertire una lettera in una lettera è

```
#include <cstdio>

int main()
{
```

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

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

    return 0;
}
```

Il risultato è

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

Leggi Manipolazione bit online: https://riptutorial.com/it/cplusplus/topic/3016/manipolazione-bit

Capitolo 66: metaprogrammazione

introduzione

In C ++ Metaprogramming si riferisce all'uso di macro o modelli per generare codice in fase di compilazione.

In generale, le macro sono disapprovate in questo ruolo e i modelli sono preferiti, sebbene non siano così generici.

La metaprogrammazione dei modelli fa spesso uso di calcoli in fase di compilazione, tramite modelli o funzioni di `constexpr` , per raggiungere i propri obiettivi di generazione del codice, tuttavia i calcoli in fase di compilazione non sono di per sé metaprogrammazione.

Osservazioni

Metaprogrammazione (o più specificamente, Metaprogrammazione di modelli) è la pratica dell'uso di modelli per creare costanti, funzioni o strutture di dati in fase di compilazione. Ciò consente di eseguire calcoli una volta al momento della compilazione anziché in ogni periodo di esecuzione.

Examples

Calcolo dei fattori

I fattoriali possono essere calcolati in fase di compilazione utilizzando tecniche di metaprogrammazione del modello.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` è una struttura, ma nella metaprogrammazione del modello viene considerata una metafunzione del modello. Per convenzione, i metafunzionamenti dei template vengono valutati controllando un particolare membro, `::type` per i metafunzioni che danno come risultato tipi: `::value` per metafuntion che generano valori.

Nel codice sopra, valutiamo la metafunzione `factorial` istanziando il modello con i parametri che vogliamo passare, e usando il `::value` per ottenere il risultato della valutazione.

La metafunzione stessa si basa sull'istanza ricorsiva della stessa metafunzione con valori più piccoli. La specializzazione `factorial<0>` rappresenta la condizione di chiusura. La metaprogrammazione dei modelli ha la maggior parte delle restrizioni di un linguaggio di programmazione funzionale , quindi la ricorsione è il costrutto di "looping" primario.

Poiché i metafuncoli del modello vengono eseguiti in fase di compilazione, i risultati possono essere utilizzati in contesti che richiedono valori in fase di compilazione. Per esempio:

```
int my_array[factorial<5>::value];
```

Gli array automatici devono avere una dimensione definita in fase di compilazione. E il risultato di una metafunzione è una costante in fase di compilazione, quindi può essere utilizzata qui.

Limitazione : la maggior parte dei compilatori non consente la profondità di ricorsione oltre un limite. Ad esempio, il compilatore `g++` per impostazione predefinita limita la ricorsione a 256 livelli. Nel caso di `g++` , il programmatore può impostare la profondità della ricorsione usando l' `-ftemplate-depth-X` .

C ++ 11

Dal momento che C ++ 11, il modello `std::integral_constant` può essere utilizzato per questo tipo di calcolo del modello:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
  std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
  std::integral_constant<long long, 1> {};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

Inoltre, `constexpr` funzioni di `constexpr` diventano un'alternativa più pulita.

```
#include <iostream>

constexpr long long factorial(long long n)
```

```
{
  return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
  char test[factorial(3)];
  std::cout << factorial(7) << '\n';
}
```

Il corpo di `factorial()` è scritto come una singola istruzione perché in C ++ 11 `constexpr` funzioni di `constexpr` possono usare solo un sottoinsieme piuttosto limitato della lingua.

C ++ 14

Dal momento che C ++ 14, molte restrizioni per `constexpr` funzioni di `constexpr` sono state eliminate e ora possono essere scritte in modo molto più pratico:

```
constexpr long long factorial(long long n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

O anche:

```
constexpr long long factorial(int n)
{
  long long result = 1;
  for (int i = 1; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

C ++ 17

Dal momento che c ++ 17 si può usare espressione di piega per calcolare fattoriale:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
   std::cout << factorial<int, 5>::value << std::endl;
}
```

Iterazione su un pacchetto di parametri

Spesso, è necessario eseguire un'operazione su ogni elemento di un pacchetto di parametri modello variadic. Ci sono molti modi per farlo e le soluzioni diventano più facili da leggere e scrivere con C ++ 17. Supponiamo di voler semplicemente stampare ogni elemento in un pacchetto. La soluzione più semplice è recitare:

C ++ 11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

Potremmo invece utilizzare il trucco di espansione, per eseguire tutto lo streaming in una singola funzione. Questo ha il vantaggio di non aver bisogno di un secondo sovraccarico, ma ha lo svantaggio di una lettura inferiore a quella stellare:

C ++ 11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Per una spiegazione di come funziona, vedi l'eccellente risposta di TC .

C ++ 17

Con C ++ 17, otteniamo due potenti nuovi strumenti nel nostro arsenale per risolvere questo problema. Il primo è un'espressione di piegatura:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

E il secondo è `if constexpr` , che ci consente di scrivere la nostra soluzione ricorsiva originale in una singola funzione:

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;
```

```
    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Iterazione con std :: integer_sequence

Dal momento che C ++ 14, lo standard fornisce il modello di classe

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

e una metafunzione generatrice per questo:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

Anche se questo è standard in C ++ 14, questo può essere implementato usando gli strumenti C ++ 11.

Possiamo usare questo strumento per chiamare una funzione con una `std::tuple` di argomenti (standardizzata in C ++ 17 come `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
        std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}


// this will print 3
int f(int, char, double);

auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Invio di tag

Un modo semplice per selezionare le funzioni al momento della compilazione è di inviare una funzione a una coppia di funzioni sovraccariche che accettano un tag come un argomento (di solito l'ultimo). Ad esempio, per implementare `std::advance()` , possiamo effettuare la spedizione sulla categoria iteratore:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }
}

template <class Iter, class Distance>
void advance(Iter& it, Distance n) {
    details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
}
```

Gli argomenti `std::XY_iterator_tag` dei `details::advance` sovraccaricati `details::advance` funzioni `details::advance` sono parametri di funzione non utilizzati. L'effettiva implementazione non ha importanza (in realtà è completamente vuota). Il loro unico scopo è quello di consentire al compilatore di selezionare un sovraccarico in base a quali `details::advance` classe tag `details::advance` viene chiamato con.

In questo esempio, `advance` utilizza la metafunzione `iterator_traits<T>::iterator_category` che restituisce una delle classi `iterator_tag` , a seconda del tipo effettivo di `Iter` . Un oggetto predefinito di `iterator_category<Iter>::type` consente quindi al compilatore di selezionare uno dei diversi overload dei `details::advance` . (È probabile che questo parametro di funzione sia completamente ottimizzato, poiché è un oggetto costruito di default di una `struct` vuota e mai usato).

L'invio di tag può fornire codice molto più facile da leggere rispetto agli equivalenti che utilizzano SFINAE e `enable_if` .

Nota: mentre C ++ 17 `if constexpr` può semplificare l'implementazione di `advance` in particolare,

non è adatto per implementazioni aperte a differenza di dispatching di tag.

Rileva se l'espressione è valida

È possibile rilevare se un operatore o una funzione possono essere chiamati su un tipo. Per verificare se una classe ha un sovraccarico di `std::hash` , si può fare questo:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};
```

C ++ 17

Da C ++ 17, `std::void_t` può essere usato per semplificare questo tipo di costrutto

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) > >
    : std::true_type
{};
```

dove `std::void_t` è definito come:

```
template< class... > using void_t = void;
```

Per rilevare se un operatore, come l' `operator<` è definito, la sintassi è quasi la stessa:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

Questi possono essere usati per usare una `std::unordered_map<T>` se `T` ha un sovraccarico per `std::hash` , ma altrimenti tenta di usare una `std::map<T>` :

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K,V>>;
```

Calcolo della potenza con C ++ 11 (e versioni successive)

Con C ++ 11 e calcoli superiori in fase di compilazione può essere molto più facile. Ad esempio, il calcolo della potenza di un determinato numero in fase di compilazione seguirà:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

La parola chiave `constexpr` è responsabile del calcolo della funzione nel tempo di compilazione, quindi e solo dopo, quando tutti i requisiti per questo verranno soddisfatti (per ulteriori informazioni consultare il riferimento alla parola chiave constexpr), ad esempio tutti gli argomenti devono essere noti al momento della compilazione.

Nota: in C ++ 11 la funzione `constexpr` deve comporre solo da un'istruzione return.

Vantaggi: confrontando questo metodo con il metodo standard di calcolo del tempo di compilazione, questo metodo è utile anche per i calcoli del runtime. Significa che se gli argomenti della funzione non sono noti al momento della compilazione (ad esempio, il valore e la potenza sono dati come input dall'utente), la funzione viene eseguita in un tempo di compilazione, quindi non è necessario duplicare un codice (come sarebbe forzato in standard più vecchi di C ++).

Per esempio

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                                // as both arguments are known at compilation time
                                // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3);  // runtime calculated,
                                      // because value is known only at runtime.
}
```

C ++ 17

Un altro modo per calcolare la potenza in fase di compilazione può utilizzare l'espressione della piega come segue:

```
#include <iostream>
#include <utility>
```

```
template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};

int main() {
   std::cout << power<int, 4, 2>::value << std::endl;
}
```

Distinzione manuale dei tipi quando viene dato qualsiasi tipo T

Quando si implementa SFINAE utilizzando `std::enable_if` , è spesso utile avere accesso ai modelli di supporto che determinano se un determinato tipo `T` corrisponde a un insieme di criteri.

Per aiutarci, lo standard fornisce già due tipi analogici a `true` e `false` che sono `std::true_type` e `std::false_type` .

L'esempio seguente mostra come rilevare se un tipo `T` è un puntatore oppure no, il modello `is_pointer` il comportamento `std::is_pointer` standard `std::is_pointer` :

```
template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }
```

Esistono tre passaggi nel codice precedente (a volte ne occorrono solo due):

1. La prima dichiarazione di `is_pointer_` è il *caso predefinito* ed eredita da `std::false_type` . Il caso *predefinito* dovrebbe sempre ereditare da `std::false_type` poiché è analogo a una "condizione `false` ".

2. La seconda dichiarazione specializza il modello `is_pointer_` per il puntatore `T*` senza preoccuparsi di cosa sia realmente `T` Questa versione eredita da `std::true_type` .

3. La terza dichiarazione (quella reale) rimuove semplicemente tutte le informazioni non necessarie da `T` (in questo caso rimuoviamo `volatile` qualificatori `const` e `volatile`) e quindi ricade su una delle due dichiarazioni precedenti.

Poiché `is_pointer<T>` è una classe, per accedere al suo valore è necessario:

- Use `::value` , ad esempio `is_pointer<int>::value` - `value` è un membro di classe statico di tipo `bool` ereditato da `std::true_type` o `std::false_type` ;
- Costruisci un oggetto di questo tipo, ad esempio `is_pointer<int>{}` - Funziona perché `std::is_pointer` eredita il suo costruttore predefinito da `std::true_type` o `std::false_type` (che

hanno costruttori `constexpr`) e sia `std::true_type` che `std::false_type` hanno `constexpr` operatori di conversione a `bool` .

È buona abitudine fornire "template helper helper" che consentono di accedere direttamente al valore:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

C ++ 17

In C ++ 17 e versioni successive, la maggior parte dei modelli di helper fornisce già una versione `_v` , ad esempio:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

If-then-else

C ++ 11

Il tipo `std::conditional` nell'intestazione della libreria standard `<type_traits>` può selezionare un tipo o l'altro, in base a un valore booleano in fase di compilazione:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

Questa struttura contiene un puntatore a `T` se `T` è maggiore della dimensione di un puntatore, o `T` stesso se è minore o uguale alla dimensione di un puntatore. Pertanto `sizeof(ValueOrPointer)` sarà sempre <= `sizeof(void*)` .

Min / Max generico con conteggio di argomenti variabili

C ++ 11

È possibile scrivere una funzione generica (ad esempio `min`) che accetta vari tipi numerici e un numero arbitrario di argomenti mediante meta-template template. Questa funzione dichiara un `min` per due argomenti e in modo ricorsivo per ulteriori.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
```

```
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Leggi metaprogrammazione online:
https://riptutorial.com/it/cplusplus/topic/462/metaprogrammazione

Capitolo 67: Metaprogrammazione aritmitica

introduzione

Questi sono esempi di utilizzo della metaprogrammazione del modello C ++ nell'elaborazione di operazioni aritmetiche in fase di compilazione.

Examples

Calcolo del potere in O (log n)

Questo esempio mostra un modo efficiente di calcolare il potere usando la metaprogrammazione del modello.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
   static const int value = 1;
   static_assert(base != 0, "power<0, 0> is not allowed");
};


template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Esempio di utilizzo:

```
std::cout << power<2, 9>::value;
```

C ++ 14

Questo gestisce anche gli esponenti negativi:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs
% 2>::intermediateValue;
```

```
    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) :
intermediateValue;

};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};


template <int base>
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};


int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

Leggi Metaprogrammazione aritmitica online:
https://riptutorial.com/it/cplusplus/topic/10907/metaprogrammazione-aritmitica

Capitolo 68: Modelli

introduzione

Classi, funzioni e variabili (dal C ++ 14) possono essere modellate. Un modello è un pezzo di codice con alcuni parametri gratuiti che diventeranno una classe, una funzione o una variabile concreta quando tutti i parametri sono specificati. I parametri possono essere tipi, valori o modelli stessi. Un modello ben noto è `std::vector` , che diventa un tipo di contenitore concreto quando viene specificato il tipo di elemento, *ad esempio,* `std::vector<int>` .

Sintassi

- *dichiarazione* template < *template-parameter-list* >
- export template < *template-parameter-list* > *declaration* / * fino a C ++ 11 * /
- modello <> *dichiarazione*
- *dichiarazione del* modello
- *dichiarazione* modello esterno / * dal C ++ 11 * /
- template < *template-parameter-list* > class ... (*opt*) *identifier* (*opt*)
- template < *template-parameter-list* > *identificatore di* classe (*opt*) = *id-expression*
- template < *template-parameter-list* > typename ... (*opt*) *identifier* (*opt*) / * dal C ++ 17 * /
- template < *template-parameter-list* > typename *identifier* (*opt*) = *id-expression* / * dal C ++ 17 * /
- *espressione postfissa* . *id-espressione del* modello
- *postfix-expression* -> modello *id-expression*
- `template` *nested-name-specifier simple-template-id* `::`

Osservazioni

Il `template` parole è una parola chiave con cinque significati diversi nel linguaggio C ++, a seconda del contesto.

1. Quando viene seguito da un elenco di parametri del modello racchiuso tra `<>` , dichiara un modello come un modello di classe , un modello di funzione o una specializzazione parziale di un modello esistente.

```
template <class T>
void increment(T& x) { ++x; }
```

2. Quando seguito da un `<>` *vuoto* , dichiara una specializzazione esplicita (completa) .

```
template <class T>
void print(T x);

template <> // <-- keyword used in this sense here
void print(const char* s) {
```

```
    // output the content of the string
    printf("%s\n", s);
}
```

3. Quando seguito da una dichiarazione senza `<>` , forma una dichiarazione o una definizione di istanziazione esplicita .

```
template <class T>
std::set<T> make_singleton(T x) { return std::set<T>(x); }

template std::set<int> make_singleton(int x); // <-- keyword used in this sense here
```

4. All'interno di un elenco di parametri del modello, introduce un parametro del modello di modello .

```
template <class T, template <class U> class Alloc>
//                 ^^^^^^^^ keyword used in this sense here
class List {
    struct Node {
        T value;
        Node* next;
    };
    Alloc<Node> allocator;
    Node* allocate_node() {
        return allocator.allocate(sizeof(T));
    }
    // ...
};
```

5. Dopo l'operatore di risoluzione dell'ambito `::` e gli operatori di accesso dei membri della classe `.` e `->` , specifica che il seguente nome è un modello.

```
struct Allocator {
    template <class T>
    T* allocate();
};

template <class T, class Alloc>
class List {
    struct Node {
        T value;
        Node* next;
    }
    Alloc allocator;
    Node* allocate_node() {
        // return allocator.allocate<Node>();       // error: < and > are interpreted as
                                                    // comparison operators
        return allocator.template allocate<Node>(); // ok; allocate is a template
        //              ^^^^^^^^ keyword used in this sense here
    }
};
```

Prima del C ++ 11, un modello poteva essere dichiarato con la parola chiave `export` , trasformandolo in un modello *esportato* . Non è necessario che la definizione di un modello esportato sia presente in ogni unità di traduzione in cui il modello viene istanziato. Ad esempio,

avrebbe dovuto funzionare quanto segue:

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
export template <class T> T identity(T x);
#endif
```

`foo.cpp` :

```
#include "foo.h"
template <class T> T identity(T x) { return x; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    const int x = identity(42); // x is 42
}
```

A causa della difficoltà di implementazione, la parola chiave `export` non è stata supportata dalla maggior parte dei compilatori principali. È stato rimosso in C ++ 11; ora è illegale usare la parola chiave `export` . Invece, di solito è necessario definire i template nelle intestazioni (a differenza delle funzioni non template, che di solito *non sono* definite nelle intestazioni). Vedi Perché i modelli possono essere implementati solo nel file di intestazione?

Examples

Modelli di funzione

I modelli possono anche essere applicati alle funzioni (così come le strutture più tradizionali) con lo stesso effetto.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

Questo può quindi essere usato allo stesso modo dei modelli di struttura.

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

In entrambi i casi l'argomento template viene utilizzato per sostituire i tipi dei parametri; il risultato funziona proprio come una normale funzione C ++ (se i parametri non corrispondono al tipo di template il compilatore applica le conversioni standard).

Un'ulteriore proprietà delle funzioni del modello (a differenza delle classi template) è che il compilatore può dedurre i parametri del modello in base ai parametri passati alla funzione.

```
printSum(4, 5);     // Both parameters are int.
                    // This allows the compiler deduce that the type
                    // T is also int.

printSum(5.0, 4);   // In this case the parameters are two different types.
                    // The compiler is unable to deduce the type of T
                    // because there are contradictions. As a result
                    // this is a compile time error.
```

Questa funzione ci consente di semplificare il codice quando combiniamo strutture e funzioni del modello. C'è un modello comune nella libreria standard che ci consente di creare una `template structure X` usando una funzione helper `make_X()` .

```
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1      first;
    T2      second;
};
// 2) A make function that has a parameter type for
//    each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

In che modo questo aiuta?

```
auto val1 = MyPair<int, float>{5, 8.7};     // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7);            // Create object using the types of the paramters.
                                            // In this code both val1 and val2 are the same
                                            // type.
```

Nota: questo non è progettato per abbreviare il codice. Questo è progettato per rendere il codice più robusto. Consente di modificare i tipi modificando il codice in un singolo posto piuttosto che in più posizioni.

Inoltro di argomenti

Il modello può accettare sia lvalue che rvalue riferimenti usando il *riferimento di inoltro* :

```
template <typename T>
void f(T &&t);
```

In questo caso, il vero tipo di `t` sarà dedotto a seconda del contesto:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

Nel primo caso, il tipo `T` è dedotto come *riferimento a* `X` (`X&`), e il tipo di `t` è il *riferimento di lvalue a* `X` , mentre nel secondo caso il tipo di `T` è dedotto come `X` e il tipo di `t` come *riferimento di valore a* `X` (`X&&`).

Nota: vale la pena notare che nel primo caso `decltype(t)` è uguale a `T` , ma non nel secondo.

Per poter inoltrare perfettamente `t` ad un'altra funzione, che si tratti di un riferimento lvalue o rvalue, si deve usare `std::forward` :

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

I riferimenti di inoltro possono essere utilizzati con modelli variadici:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Nota: i riferimenti di inoltro possono essere utilizzati solo per i parametri del modello, ad esempio, nel seguente codice, `v` è un riferimento di rvalue, non un riferimento di inoltro:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Modello di classe base

L'idea di base di un modello di classe è che il parametro template venga sostituito da un tipo in fase di compilazione. Il risultato è che la stessa classe può essere riutilizzata per più tipi. L'utente specifica quale tipo verrà utilizzato quando viene dichiarata una variabile della classe. Tre esempi di questo sono mostrati in `main()` :

```
#include <iostream>
using std::cout;

template <typename T>         // A simple class to hold one number of any type
class Number {
public:
    void setNum(T n);         // Sets the class field to the given number
    T plus1() const;          // returns class field's "follower"
private:
    T num;                    // Class field
```

```
};

template <typename T>          // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T>          // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt;         // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n";     // Prints 2

    Number<double> aDouble;    // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n";    // Prints 4.14159

    Number<float> aFloat;      // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n";      // Prints 2.4

    return 0;  // Successful completion
}
```

Specializzazione dei modelli

È possibile definire l'implementazione per istanze specifiche di una classe / metodo template.

Ad esempio se hai:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

Puoi quindi scrivere:

```
template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Quindi un utente che scrive `sqrt(4.0)` otterrà l'implementazione generica mentre `sqrt(4)` otterrà l'implementazione specializzata.

Specializzazione di template parziale

Al contrario di una specializzazione del modello parziale, la specializzazione del modello parziale consente di introdurre template con alcuni degli argomenti del modello esistente. La specializzazione del modello parziale è disponibile solo per classi / strutture template:

```
// Common case:
template<typename T, typename U>
struct S {
```

```
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) {// Do something}
};
```

Come mostrato sopra, le specializzazioni di template parziali possono introdurre insiemi di dati e membri di funzioni completamente diversi.

Quando viene creata un'istanza di un modello parzialmente specializzato, viene selezionata la specializzazione più adatta. Ad esempio, definiamo un modello e due specializzazioni parziali:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Ora le seguenti chiamate:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

stamperà

```
General case
T = int
T = int, U = double
```

I modelli di funzioni possono essere completamente specializzati:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
```

```
}

// OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1,2);    // Prints "Two ints: 1 2"
}

// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Valore del parametro del modello predefinito

Proprio come nel caso degli argomenti della funzione, i parametri del modello possono avere i loro valori predefiniti. Tutti i parametri del modello con un valore predefinito devono essere dichiarati alla fine dell'elenco dei parametri del modello. L'idea di base è che i parametri del modello con valore predefinito possono essere omessi durante l'istanziazione del modello.

Semplice esempio di utilizzo del valore predefinito del parametro del modello:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;

    /* Print the length of a.arr: 5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* Last parameter is omitted, N = 10 */
    my_array<int> b;

    /* Print the length of a.arr: 10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

Modello alias

C ++ 11

Esempio di base:

```
template<typename T> using pointer = T*;
```

Questa definizione rende il `pointer<T>` un alias di `T*` . Per esempio:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

I modelli alias non possono essere specializzati. Tuttavia, tale funzionalità può essere ottenuta indirettamente facendo in modo che si riferiscano a un tipo annidato in una struttura:

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

template<typename T>
 struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

Parametri del modello di modello

A volte vorremmo passare al modello un tipo di modello senza fissarne i valori. Questo è ciò per cui vengono creati i parametri del modello di modello. Esempi di parametri template template molto semplici:

```
template <class T>
struct Tag1 { };

template <class T>
struct Tag2 { };

template <template <class> class Tag>
struct IntTag {
   typedef Tag<int> type;
};

int main() {
   IntTag<Tag1>::type t;
}
```

C ++ 11

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

template <class T, template <class...> class C, class U>
C<T> cast_all(const C<U> &c) {
   C<T> result(c.begin(), c.end());
   return result;
}

int main() {
   std::vector<float> vf = {1.2, 2.6, 3.7};
   auto vi = cast_all<int>(vf);
   for(auto &&i: vi) {
      std::cout << i << std::endl;
   }
}
```

Dichiarare argomenti modello non di tipo con auto

Prima di C ++ 17, quando scrivevo un parametro non-type del modello, dovevi prima specificare il suo tipo. Quindi un modello comune è diventato scrivere qualcosa come:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

Ma per espressioni complicate, l'uso di qualcosa di simile comporta la scrittura di `decltype(expr), expr` durante l'istanziazione di modelli. La soluzione è semplificare questo idioma e consentire semplicemente l' `auto` :

C ++ 17

```
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};

using five = integral_constant<5>;
```

Empty deleter personalizzato per unique_ptr

Un buon esempio motivante può venire dal tentativo di combinare l'ottimizzazione di base vuota con un deleter personalizzato per `unique_ptr` . Differenti deletatori dell'API C hanno diversi tipi di rendimento, ma non ci interessa: vogliamo solo che qualcosa funzioni per qualsiasi funzione:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) const {
        DeleteFn(ptr);
    }
};

template <T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

E ora puoi semplicemente utilizzare qualsiasi puntatore a funzione che possa assumere un argomento di tipo `T` come parametro non di tipo template, indipendentemente dal tipo restituito, e ricavarne un overhead no-size `unique_ptr` :

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Parametro modello non di tipo

A parte i tipi come parametro template, è possibile dichiarare valori di espressioni costanti che soddisfano uno dei seguenti criteri:

- tipo integrale o di enumerazione,
- puntatore all'oggetto o puntatore alla funzione,
- lvalue riferimento a oggetto o lvalue riferimento alla funzione,
- puntatore al membro,
- `std::nullptr_t` .

Come tutti i parametri del modello, i parametri del modello non di tipo possono essere specificati, impostati in modo predefinito o derivati in modo implicito tramite Detrazione argomento modello.

Esempio di utilizzo dei parametri del modello non di tipo:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size])  // Pass array by reference. Requires.
{                                        // an exact size. We allow all sizes
    return size;                         // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int  anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

Esempio di specifica esplicita dei parametri del modello di tipo e non di tipo:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

I parametri del modello non di tipo sono uno dei modi per ottenere la ricorrenza del modello e consentono di eseguire Metaprogramming .

Strutture dati modello variabile

C ++ 14

È spesso utile definire classi o strutture che hanno un numero variabile e un tipo di membri di dati definiti in fase di compilazione. L'esempio canonico è `std::tuple` , ma a volte è necessario definire le proprie strutture personalizzate. Ecco un esempio che definisce la struttura usando compounding (piuttosto che ereditarietà come `std::tuple` . Inizia con la definizione generale (vuota), che funge anche da base per la terminazione della recrusione nella specializzazione successiva:

```
template<typename ... T>
struct DataStructure {};
```

Questo ci permette già di definire una struttura vuota, `DataStructure<> data` , anche se non è ancora molto utile.

Segue la specializzazione del caso ricorsivo:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

Ora è sufficiente per noi creare strutture dati arbitrarie, come `DataStructure<int, float, std::string> data(1, 2.1, "hello")` .

Quindi cosa sta succedendo? Innanzitutto, si noti che questa è una specializzazione il cui requisito è che esista almeno un parametro di modello variadico (vale a dire `T` sopra), mentre non si preoccupa del trucco specifico del pacchetto `Rest` . Sapere che `T` esiste consente, per `first` , la definizione del suo membro dei dati. Il resto dei dati è ricorsivamente impacchettato come `DataStructure<Rest ... > rest` . Il costruttore avvia entrambi i membri, inclusa una chiamata di costruttore ricorsiva al membro del `rest` .

Per capirlo meglio, possiamo lavorare su un esempio: supponiamo di avere una dichiarazione `DataStructure<int, float> data` . La dichiarazione corrisponde per prima cosa alla specializzazione, fornendo una struttura con i membri dei dati di `DataStructure<float> rest int first` e `DataStructure<float> rest` . Il `rest` definizione corrisponde ancora una volta questa specializzazione, creando un proprio `float first` e `DataStructure<> rest` membri. Infine, quest'ultimo `rest` corrisponde alla definizione del caso base, producendo una struttura vuota.

Puoi visualizzarlo come segue:

```
DataStructure<int, float>
   -> int first
   -> DataStructure<float> rest
         -> float first
         -> DataStructure<> rest
              -> (empty)
```

Ora abbiamo la struttura dei dati, ma non è ancora molto utile dato che non possiamo accedere facilmente ai singoli elementi di dati (ad esempio per accedere all'ultimo membro di `DataStructure<int, float, std::string> data` dovremmo utilizzare i `data.rest.rest.first` , che non è esattamente user-friendly). Quindi aggiungiamo un metodo `get` (necessario solo nella specializzazione in quanto la struttura del caso base non ha dati da `get`):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
    ...
};
```

Come potete vedere, questa funzione del membro `get` è a sua volta `data.get<1>()` su modelli - questa volta sull'indice del membro necessario (quindi l'utilizzo può essere `data.get<1>()` come `data.get<1>()` , simile a `std::tuple`). Il lavoro effettivo è svolto da una funzione statica in una classe helper, `GetHelper` . La ragione per cui non possiamo definire la funzionalità richiesta direttamente `DataStructure` 's `get` è perché (come vedremo tra breve vedi) avremmo bisogno di specializzarsi su `idx` - ma non è possibile specializzare una funzione di membro template, senza che si specializza la classe contenente modello. Nota anche l'uso di un `auto` stile C ++ 14 qui rende le nostre vite molto più semplici, altrimenti avremmo bisogno di un'espressione piuttosto complicata per il tipo di ritorno.

Quindi alla classe helper. Questa volta avremo bisogno di una dichiarazione in avanti vuota e di due specializzazioni. Prima la dichiarazione:

```
template<size_t idx, typename T>
struct GetHelper;
```

Ora il caso base (quando `idx==0`). In questo caso, restituiamo il `first` membro:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

Nel caso ricorsivo, decrementiamo `idx` e invochiamo `GetHelper` per il `rest` membro:

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

Per lavorare con un esempio, supponiamo di avere `DataStructure<int, float> data` e abbiamo bisogno di `data.get<1>()` . Questo richiama `GetHelper<1, DataStructure<int, float>>::get(data)` (la seconda specializzazione), che a sua volta richiama `GetHelper<0,`

`DataStructure<float>>::get(data.rest)` , che alla fine restituisce (dalla 1a specializzazione come ora `idx` è 0) `data.rest.first` .

Quindi è così! Ecco l'intero codice di funzionamento, con alcuni esempi di utilizzo nella funzione `main` :

```
#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;
```

```
    return 0;
}
```

Istanziazione esplicita

Una definizione di istanziazione esplicita crea e dichiara una classe, una funzione o una variabile concreta da un modello, senza utilizzarlo ancora. Un'istanza esplicita può essere referenziata da altre unità di traduzione. Questo può essere usato per evitare di definire un modello in un file di intestazione, se verrà istanziato solo con un insieme finito di argomenti. Per esempio:

```
// print_string.h
template <class T>
void print_string(const T* str);

// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Poiché `print_string<char>` e `print_string<wchar_t>` sono esplicitamente istanziati in `print_string.cpp` , il linker sarà in grado di trovarli anche se il modello `print_string` non è definito nell'intestazione. Se queste dichiarazioni di istanziazione esplicite non fossero presenti, si verificherebbe probabilmente un errore del linker. Vedi Perché i modelli possono essere implementati solo nel file di intestazione?

C ++ 11

Se una definizione di istanziazione esplicita è preceduta dalla `extern` parola chiave , diventa una *dichiarazione* di istanza esplicita, invece. La presenza di una dichiarazione di istanziazione esplicita per una data specializzazione impedisce l'istanza implicita della specializzazione data all'interno dell'unità di traduzione corrente. Invece, un riferimento a quella specializzazione che altrimenti causerebbe un'istanza implicita può riferirsi a una definizione di istanziazione esplicita nella stessa o in un'altra TU.

`foo.h`

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
#endif
```

`foo.cpp`

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

`main.cpp`

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42);   // instantiates foo<int> here;
               // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
               // uses instantiation of foo<double> in foo.cpp instead
}
```

Leggi Modelli online: https://riptutorial.com/it/cplusplus/topic/460/modelli

Capitolo 69: Modelli di espressione

Examples

Modelli di espressioni di base sulle espressioni algebriche element-wise

Introduzione e motivazione

I modelli di espressione (indicati come **ET** in seguito) sono una potente tecnica di meta-programmazione del modello, utilizzata per accelerare i calcoli di espressioni talvolta piuttosto costose. È ampiamente utilizzato in diversi domini, ad esempio nell'implementazione di librerie di algebra lineare.

Per questo esempio, considera il contesto dei calcoli algebrici lineari. Più specificamente, calcoli che riguardano solo **operazioni element-wise** . Questo tipo di calcoli sono le applicazioni di base degli **ET** e rappresentano una buona introduzione al modo in cui gli **ET** lavorano internamente.

Diamo un'occhiata a un esempio motivante. Considera il calcolo dell'espressione:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

Qui per semplicità, presumo che la classe `Vector` e l'operazione + (vector plus: operazione elemento-saggio plus) e operazione * (qui significa prodotto interno vettoriale: anche operazione elemento-saggio) siano entrambi correttamente implementati, come come dovrebbero essere, matematicamente.

In un'implementazione convenzionale senza utilizzare **ET** (o altre tecniche simili), per ottenere il `result` finale sono necessarie **almeno cinque** costruzioni di istanze `Vector` :

1. Tre istanze corrispondenti a `vec_1` , `vec_2` e `vec_3` .
2. Un'istanza `Vector` temporanea `_tmp` , che rappresenta il risultato di `_tmp = vec_2*vec_3;` .
3. Infine con l'uso corretto **dell'ottimizzazione del valore** di **ritorno** , la costruzione del `result` finale in `result = vec_1 + _tmp;` .

L'implementazione tramite **ET** può **eliminare la creazione di** `Vector _tmp` **temporaneo** in 2, lasciando così solo **quattro** costruzioni di istanze `Vector` . Più interessante, considera la seguente espressione che è più complessa:

```
Vector result = vec_1 + (vec_2*vec3 + vec_1)*(vec_2 + vec_3*vec_1);
```

Ci saranno anche quattro costruzioni di istanze `Vector` in totale: `vec_1, vec_2, vec_3` e `result` . In

altre parole, in questo esempio, in **cui sono coinvolte solo operazioni basate su elementi** , è garantito che **non verranno creati oggetti temporanei da calcoli intermedi** .

Come funzionano gli ET

In sostanza, gli **ET per i calcoli algebrici** sono costituiti da due elementi costitutivi:

1. **Pure algebraic expressions** (**PAE**): sono proxy / astrazioni di espressioni algebriche. Un algebrico puro non esegue calcoli effettivi, sono semplicemente astrazioni / modellizzazione del flusso di lavoro di calcolo. Un PAE può essere un modello **dell'ingresso o dell'output di qualsiasi espressione algebrica** . Le istanze di **PAE** sono spesso considerate economiche da copiare.
2. **Valutazioni pigre** : quali sono l'implementazione di calcoli reali. Nell'esempio seguente, vedremo che per le espressioni che coinvolgono solo operazioni element-wise, le valutazioni lazy possono implementare calcoli effettivi all'interno dell'operazione di accesso indicizzato sul risultato finale, creando così uno schema di valutazione on-demand: non viene eseguito un calcolo solo se si accede / chiede il risultato finale.

Quindi, in particolare come implementiamo gli **ET** in questo esempio? Passeggiamo ora.

Considera sempre il seguente frammento di codice:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

L'espressione per calcolare il risultato può essere scomposta ulteriormente in due sottoespressioni:

1. Un vettore più un'espressione (indicato come **plus_expr**)
2. Un'espressione di prodotto interno vettoriale (indicata come **innerprod_expr**).

Cosa fanno gli **ET** è il seguente:

- Invece di calcolare immediatamente ogni sottoespressione, gli **ET** modellano prima l'intera espressione usando una struttura grafica. Ogni nodo nel grafico rappresenta un **PAE** . La connessione di bordo dei nodi rappresenta il flusso di calcolo effettivo. Quindi per l'espressione sopra, otteniamo il seguente grafico:

```
       result = plus_expr( vec_1, innerprod_expr(vec_2, vec_3) )
          /   \
         /     \
        /       \
       /   innerprod_expr( vec_2, vec_3 )
      /          /  \
     /          /    \
    /          /      \
```

```
vec_1     vec_2    vec_3
```

- Il calcolo finale viene implementato **osservando la gerarchia del grafo** : poiché qui si tratta **solo di** operazioni **element-wise** , il calcolo di ogni valore indicizzato nel `result` **può essere fatto indipendentemente** : la valutazione finale del `result` può essere **spostata** pigramente su un **elemento- saggia valutazione** di ogni elemento di `result` . In altre parole, poiché il calcolo di un elemento di `result` , `elem_res` , può essere espresso utilizzando gli elementi corrispondenti in `vec_1` (`elem_1`), `vec_2` (`elem_2`) e `vec_3` (`elem_3`) come:

```
elem_res = elem_1 + elem_2*elem_3;
```

non è quindi necessario creare un `Vector` temporaneo per memorizzare il risultato del prodotto interno intermedio: **l'intero calcolo per un elemento può essere eseguito del tutto e codificato all'interno dell'operazione di accesso indicizzato** .

Ecco i codici di esempio in azione.

File vec.hh: wrapper per std :: vector, utilizzato per mostrare il log quando viene chiamata una costruzione.

```
#ifndef EXPR_VEC
# define EXPR_VEC

# include <vector>
# include <cassert>
# include <utility>
# include <iostream>
# include <algorithm>
# include <functional>

///
/// This is a wrapper for std::vector. It's only purpose is to print out a log when a
/// vector constructions in called.
/// It wraps the indexed access operator [] and the size() method, which are
/// important for later ETs implementation.
///

// std::vector wrapper.
template<typename ScalarType> class Vector
{
public:
  explicit Vector() { std::cout << "ctor called.\n"; };
  explicit Vector(int size): _vec(size) { std::cout << "ctor called.\n"; };
  explicit Vector(const std::vector<ScalarType> &vec): _vec(vec)
  { std::cout << "ctor called.\n"; };

  Vector(const Vector<ScalarType> & vec): _vec{vec()}
  { std::cout << "copy ctor called.\n"; };
  Vector(Vector<ScalarType> && vec): _vec(std::move(vec()))
  { std::cout << "move ctor called.\n"; };
```

```
  Vector<ScalarType> & operator=(const Vector<ScalarType> &) = default;
  Vector<ScalarType> & operator=(Vector<ScalarType> &&) = default;

  decltype(auto) operator[](int indx) { return _vec[indx]; }
  decltype(auto) operator[](int indx) const { return _vec[indx]; }

  decltype(auto) operator()() & { return (_vec); };
  decltype(auto) operator()() const & { return (_vec); };
  Vector<ScalarType> && operator()() && { return std::move(*this); }

  int size() const { return _vec.size(); }

private:
  std::vector<ScalarType> _vec;
};

///
/// These are conventional overloads of operator + (the vector plus operation)
/// and operator * (the vector inner product operation) without using the expression
/// templates. They are later used for bench-marking purpose.
///

// + (vector plus) operator.
template<typename ScalarType>
auto operator+(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops plus -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::plus<>());
  return Vector<ScalarType>(std::move(_vec));
}

// * (vector inner product) operator.
template<typename ScalarType>
auto operator*(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops multiplies -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::multiplies<>());
  return Vector<ScalarType>(std::move(_vec));
}

#endif //!EXPR_VEC
```

File expr.hh: implementazione di modelli di espressione per operazioni basate su elementi (vector plus e vector inner

product)

Scopriamolo in sezioni.

1. Sezione 1 implementa una classe base per tutte le espressioni. Impiega il **modello di modello Curiosamente ricorrente** (CRTP).
2. La Sezione 2 implementa il primo **PAE** : un **terminale** , che è solo un wrapper (riferimento const) di una struttura di dati di input contenente un valore di input reale per il calcolo.
3. La sezione 3 implementa il secondo **PAE** : **binary_operation** , che è un modello di classe utilizzato in seguito per vector_plus e vector_innerprod. È parametrizzato dal **tipo di operazione** , **dal PAE sul lato sinistro** e **dal PAE sul lato destro** . Il calcolo reale è codificato nell'operatore di accesso indicizzato.
4. La sezione 4 definisce le operazioni vector_plus e vector_innerprod come operazioni basate **sull'elemento** . Sovraccarica anche l'operatore + e * per **PAE** s: in modo che queste due operazioni restituiscano anche **PAE** .

```
#ifndef EXPR_EXPR
# define EXPR_EXPR


/// Fwd declaration.
template<typename> class Vector;

namespace expr
{


/// ----------------------------------------
///
/// Section 1.
///
/// The first section is a base class template for all kinds of expression. It
/// employs the Curiously Recurring Template Pattern, which enables its instantiation
/// to any kind of expression structure inheriting from it.
///
/// ----------------------------------------


  /// Base class for all expressions.
  template<typename Expr> class expr_base
  {
  public:
    const Expr& self() const { return static_cast<const Expr&>(*this); }
    Expr& self() { return static_cast<Expr&>(*this); }

  protected:
    explicit expr_base() {};
    int size() const { return self().size_impl(); }
    auto operator[](int indx) const { return self().at_impl(indx); }
    auto operator()() const { return self()(); };
  };


/// ----------------------------------------
```

```
///
/// The following section 2 & 3 are abstractions of pure algebraic expressions (PAE).
/// Any PAE can be converted to a real object instance using operator(): it is in
/// this conversion process, where the real computations are done.

///
/// Section 2. Terminal
///
/// A terminal is an abstraction wrapping a const reference to the Vector data
/// structure. It inherits from expr_base, therefore providing a unified interface
/// wrapping a Vector into a PAE.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator.
///
/// It might no be necessary for user defined data structures to have a terminal
/// wrapper, since user defined structure can inherit expr_base, therefore eliminates
/// the need to provide such terminal wrapper.
///
/// ----------------------------------------


  /// Generic wrapper for underlying data structure.
  template<typename DataType> class terminal: expr_base<terminal<DataType>>
  {
  public:
    using base_type = expr_base<terminal<DataType>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit terminal(const DataType &val): _val(val) {}
    int size_impl() const { return _val.size(); };
    auto at_impl(int indx) const { return _val[indx]; };
    decltype(auto) operator()() const { return (_val); }

  private:
    const DataType &_val;
  };


/// ----------------------------------------
///
/// Section 3. Binary operation expression.
///
/// This is a PAE abstraction of any binary expression. Similarly it inherits from
/// expr_base.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator. Each call to the at_impl() method is
/// a element wise computation.
///
/// ----------------------------------------


  /// Generic wrapper for binary operations (that are element-wise).
  template<typename Ops, typename lExpr, typename rExpr>
  class binary_ops: public expr_base<binary_ops<Ops,lExpr,rExpr>>
  {
  public:
    using base_type = expr_base<binary_ops<Ops,lExpr,rExpr>>;
```

```
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit binary_ops(const Ops &ops, const lExpr &lxpr, const rExpr &rxpr)
      : _ops(ops), _lxpr(lxpr), _rxpr(rxpr) {};
    int size_impl() const { return _lxpr.size(); };

    /// This does the element-wise computation for index indx.
    auto at_impl(int indx) const { return _ops(_lxpr[indx], _rxpr[indx]); };

    /// Conversion from arbitrary expr to concrete data type. It evaluates
    /// element-wise computations for all indices.
    template<typename DataType> operator DataType()
    {
      DataType _vec(size());
      for(int _ind = 0; _ind < _vec.size(); ++_ind)
        _vec[_ind] = (*this)[_ind];
      return _vec;
    }

  private: /// Ops and expr are assumed cheap to copy.
    Ops   _ops;
    lExpr _lxpr;
    rExpr _rxpr;
  };


/// ----------------------------------------
/// Section 4.
///
/// The following two structs defines algebraic operations on PAEs: here only vector
/// plus and vector inner product are implemented.
///
/// First, some element-wise operations are defined : in other words, vec_plus and
/// vec_prod acts on elements in Vectors, but not whole Vectors.
///
/// Then, operator + & * are overloaded on PAEs, such that: + & * operations on PAEs
/// also return PAEs.
///
/// ----------------------------------------


  /// Element-wise plus operation.
  struct vec_plus_t
  {
    constexpr explicit vec_plus_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs+rhs; }
  };

  /// Element-wise inner product operation.
  struct vec_prod_t
  {
    constexpr explicit vec_prod_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs*rhs; }
  };
```

```
  /// Constant plus and inner product operator objects.
  constexpr vec_plus_t vec_plus{};
  constexpr vec_prod_t vec_prod{};

  /// Plus operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator+(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_plus_t,lExpr,rExpr>(vec_plus,lhs,rhs); }

  /// Inner prod operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator*(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_prod_t,lExpr,rExpr>(vec_prod,lhs,rhs); }

} //!expr


#endif //!EXPR_EXPR
```

File main.cc: prova file src

```
# include <chrono>
# include <iomanip>
# include <iostream>
# include "vec.hh"
# include "expr.hh"
# include "boost/core/demangle.hpp"


int main()
{
  using dtype = float;
  constexpr int size = 5e7;

  std::vector<dtype> _vec1(size);
  std::vector<dtype> _vec2(size);
  std::vector<dtype> _vec3(size);

  // ... Initialize vectors' contents.

  Vector<dtype> vec1(std::move(_vec1));
  Vector<dtype> vec2(std::move(_vec2));
  Vector<dtype> vec3(std::move(_vec3));

  unsigned long start_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "\nNo-ETs evaluation starts.\n";

  Vector<dtype> result_no_ets = vec1 + (vec2*vec3);

  unsigned long stop_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << std::setprecision(6) << std::fixed
            << "No-ETs. Time eclapses: " << (stop_ms_no_ets-start_ms_no_ets)/1000.0
            << " s.\n" << std::endl;
```

```
  unsigned long start_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "Evaluation using ETs starts.\n";

  expr::terminal<Vector<dtype>> vec4(vec1);
  expr::terminal<Vector<dtype>> vec5(vec2);
  expr::terminal<Vector<dtype>> vec6(vec3);

  Vector<dtype> result_ets = (vec4 + vec5*vec6);

  unsigned long stop_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << std::setprecision(6) << std::fixed
            << "With ETs. Time eclapses: " << (stop_ms_ets-start_ms_ets)/1000.0
            << " s.\n" << std::endl;

  auto ets_ret_type = (vec4 + vec5*vec6);
  std::cout << "\nETs result's type:\n";
  std::cout << boost::core::demangle( typeid(decltype(ets_ret_type)).name() ) << '\n';

  return 0;
}
```

Ecco un output possibile quando compilato con `-O3 -std=c++14` utilizzando GCC 5.3:

```
ctor called.
ctor called.
ctor called.

No-ETs evaluation starts.
ctor called.
ctor called.
No-ETs. Time eclapses: 0.571000 s.

Evaluation using ETs starts.
ctor called.
With ETs. Time eclapses: 0.164000 s.


ETs result's type:
expr::binary_ops<expr::vec_plus_t, expr::terminal<Vector<float> >,
expr::binary_ops<expr::vec_prod_t, expr::terminal<Vector<float> >,
expr::terminal<Vector<float> > > >
```

Le osservazioni sono:

- **In questo caso l'** utilizzo degli **ET** raggiunge un incremento delle prestazioni piuttosto significativo (> 3x).
- La creazione dell'oggetto vettoriale temporaneo è stata eliminata. Come nel caso degli **ET** , Ctor è chiamato solo una volta.
- Boost :: demangle è stato usato per visualizzare il tipo di ritorno ET prima della conversione: ha chiaramente costruito esattamente lo stesso grafico di espressione mostrato sopra.

Restituzioni e avvertimenti

- Un ovvio svantaggio degli **ET** è la curva di apprendimento, la complessità dell'implementazione e la difficoltà di manutenzione del codice. Nell'esempio sopra in cui vengono prese in considerazione solo le operazioni basate sull'elemento, l'implementazione contiene già enormi quantità di piastre, e tanto meno nel mondo reale, dove espressioni algebriche più complesse si verificano in ogni calcolo e l'indipendenza dal punto di vista dell'elemento non è più valida (ad esempio la moltiplicazione della matrice), la difficoltà sarà esponenziale.

- Un altro avvertimento sull'uso degli **ET** è che giocano bene con la parola chiave `auto` . Come accennato in precedenza, i **PAE** sono essenzialmente proxy: e i proxy in pratica non funzionano bene con l' `auto` . Considera il seguente esempio:

```
auto result = ...;                // Some expensive expression:
                                  // auto returns the expr graph,
                                  // NOT the computed value.
for(auto i = 0; i < 100; ++i)
    ScalrType value = result* ... // Some other expensive computations using result.
```

Qui **in ogni iterazione del ciclo for, il risultato verrà rivalutato** , poiché il grafico dell'espressione anziché il valore calcolato viene passato al ciclo for.

Librerie esistenti che implementano **ET**

- **boost :: proto** è una potente libreria che ti permette di definire le tue regole e grammatiche per le tue espressioni ed eseguirle usando **ET** .
- **Eigen** è una libreria per l'algebra lineare che implementa vari calcoli algebrici in modo efficiente usando **ET** .

Un esempio di base che illustra i modelli di espressione

Un modello di espressione è una tecnica di ottimizzazione in fase di compilazione utilizzata principalmente nel calcolo scientifico. Lo scopo principale è evitare temporanei inutili e ottimizzare i calcoli del loop usando un singolo passaggio (in genere quando si eseguono operazioni su aggregati numerici). Modelli di espressione sono stati inizialmente concepiti per aggirare le inefficienze naive sovraccarico operatore nell'attuazione numerici `Array` o `Matrix` tipi. Una terminologia equivalente per i modelli di espressione è stata introdotta da Bjarne Stroustrup, che li chiama "operazioni fuse" nell'ultima versione del suo libro, "Il linguaggio di programmazione C ++".

Prima di entrare effettivamente nei modelli di espressione, dovresti capire perché ne hai bisogno in primo luogo. Per illustrare questo, considera la semplice classe Matrix fornita di seguito:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;
```

```
    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW>
operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

Data la definizione della classe precedente, ora puoi scrivere espressioni Matrix come:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
        c(x, y) = 3.0;
    }
}

Matrix<double, cols, rows> d = a + b + c;  // d(x, y) = 6
```

Come illustrato sopra, essere in grado di sovraccaricare l' `operator+()` fornisce una notazione che riproduce la notazione matematica naturale per le matrici.

Sfortunatamente, l'implementazione precedente è anche altamente inefficiente rispetto a una versione equivalente "realizzata a mano".

Per capire perché, devi considerare cosa succede quando scrivi un'espressione come `Matrix d = a + b + c` . Questo infatti si espande in `((a + b) + c)` o `operator+(operator+(a, b), c)` . In altre parole, il ciclo all'interno `operator+()` viene eseguito due volte, mentre potrebbe essere facilmente eseguito in un singolo passaggio. Ciò comporta anche la creazione di 2 provini, che peggiorano ulteriormente le prestazioni. In sostanza, aggiungendo la flessibilità per usare una notazione vicina

alla sua controparte matematica, hai anche reso la classe `Matrix` altamente inefficiente.

Ad esempio, senza il sovraccarico dell'operatore, è possibile implementare una sommatoria Matrix molto più efficiente utilizzando un singolo passaggio:

```
template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}
```

L'esempio precedente tuttavia ha i suoi svantaggi perché crea un'interfaccia molto più complessa per la classe Matrix (dovresti considerare metodi come `Matrix::add2()` , `Matrix::AddMultiply()` e così via).

Facciamo un passo indietro e vediamo come possiamo adattare l'overloading dell'operatore per eseguire in modo più efficiente

Il problema deriva dal fatto che l'espressione `Matrix d = a + b + c` viene valutata troppo "appassionatamente" prima di aver avuto l'opportunità di costruire l'intero albero delle espressioni. In altre parole, ciò che si vuole veramente ottenere è di valutare `a + b + c` in un solo passaggio e solo quando effettivamente si ha bisogno di assegnare l'espressione risultante a `d` .

Questa è l'idea alla base dei modelli di espressione: invece di avere `operator+()` valuta immediatamente il risultato dell'aggiunta di due istanze Matrix, restituirà un *"modello di espressione"* per la valutazione futura una volta che l'intero albero di espressioni è stato creato.

Ad esempio, ecco una possibile implementazione per un modello di espressione corrispondente alla somma di 2 tipi:

```
template <typename LHS, typename RHS>
class MatrixSum
{
public:
    using value_type = typename LHS::value_type;

    MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

    value_type operator() (int x, int y) const  {
        return lhs(x, y) + rhs(x, y);
    }
private:
    const LHS& lhs;
    const RHS& rhs;
};
```

Ed ecco la versione aggiornata di `operator+()`

```
template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}
```

Come si può vedere, l' `operator+()` non restituisce più una "valutazione entusiasta" del risultato dell'aggiunta di 2 istanze Matrix (che sarebbe un'altra istanza Matrix), ma invece di un modello di espressione che rappresenta l'operazione di aggiunta. Il punto più importante da tenere a mente è che l'espressione non è stata ancora valutata. Mantiene semplicemente i riferimenti ai suoi operandi.

In effetti, nulla ti impedisce di `MatrixSum<>` un'istanza del modello di espressione `MatrixSum<>` come segue:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

Tuttavia, in una fase successiva, quando è effettivamente necessario il risultato della somma, valutare l'espressione `d = a + b` come segue:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}
```

Come si può vedere, un altro vantaggio di utilizzare un modello di espressione, è che si è praticamente riuscito a valutare la somma di `a` e `b` e assegnarlo a `d` in un unico passaggio.

Inoltre, nulla ti impedisce di combinare più modelli di espressioni. Ad esempio, `a + b + c` risulterebbe nel seguente modello di espressione:

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

E anche qui puoi valutare il risultato finale con un solo passaggio:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}
```

Infine, l'ultimo pezzo del puzzle è quello di collegare effettivamente il modello di espressione alla classe `Matrix` . Ciò si ottiene essenzialmente fornendo un'implementazione per `Matrix::operator=()` , che prende il modello di espressione come argomento e lo valuta in un passaggio, come fatto "manualmente" prima:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
```

```
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

    template <typename E>
    Matrix<T, COL, ROW>& operator=(const E& expression) {
        for (std::size_t y = 0; y != rows(); ++y) {
            for (std::size_t x = 0; x != cols(); ++x) {
                values[y * COL + x] = expression(x, y);
            }
        }
        return *this;
    }

private:
    std::vector<T> values;
};
```

Leggi Modelli di espressione online: https://riptutorial.com/it/cplusplus/topic/3404/modelli-di-espressione

Capitolo 70: Modello di modello curiosamente ricorrente (CRTP)

introduzione

Un modello in cui una classe eredita da un modello di classe con se stesso come uno dei suoi parametri del modello. Il CRTP viene solitamente utilizzato per fornire il *polimorfismo statico* in C ++.

Examples

Il modello di modello Curiosamente ricorrente (CRTP)

CRTP è un'alternativa potente e statica alle funzioni virtuali e all'ereditarietà tradizionale che può essere utilizzata per fornire proprietà dei tipi in fase di compilazione. Funziona avendo un modello di classe base che prende, come uno dei suoi parametri template, la classe derivata. Ciò le permette di svolgere legalmente uno `static_cast` della sua `this` puntatore alla classe derivata.

Ovviamente, ciò significa anche che una classe CRTP deve *sempre* essere utilizzata come classe base di un'altra classe. E la classe derivata deve passare alla classe base.

C ++ 14

Supponiamo che tu abbia un set di contenitori che supportano tutte le funzioni `begin()` e `end()` . I requisiti della libreria standard per i contenitori richiedono più funzionalità. Possiamo progettare una classe base CRTP che fornisce tale funzionalità, basata esclusivamente su `begin()` e `end()` :

```
#include <iterator>
template <typename Sub>
class Container {
  private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

  public:
    decltype(auto) front() {
      return *self().begin();
    }

    decltype(auto) back() {
      return *std::prev(self().end());
    }

    decltype(auto) size() const {
      return std::distance(self().begin(), self().end());
    }

    decltype(auto) operator[](std::size_t i) {
```

```
      return *std::next(self().begin(), i);
    }
};
```

La classe precedente fornisce le funzioni `front()` , `back()` , `size()` e `operator[]` per qualsiasi sottoclasse che fornisce `begin()` e `end()` . Un esempio di sottoclasse è un semplice array allocato dinamicamente:

```
#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
  public:
    using Base = Container<DynArray<T>>;

    DynArray(std::size_t size)
      : size_{size},
      data_{std::make_unique<T[]>(size_)}
    { }

    T* begin() { return data_.get(); }
    const T* begin() const { return data_.get(); }
    T* end() { return data_.get() + size_; }
    const T* end() const { return data_.get() + size_; }

  private:
    std::size_t size_;
    std::unique_ptr<T[]> data_;
};
```

Gli utenti della classe `DynArray` possono utilizzare facilmente le interfacce fornite dalla classe base CRTP come segue:

```
DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);
```

Utilità: questo modello evita in particolare le chiamate alle funzioni virtuali in fase di esecuzione che si verificano attraversano la gerarchia dell'ereditarietà e si basano semplicemente su valori statici:

```
DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls
```

L'unico cast statico all'interno della funzione `begin()` nella classe base `Container<DynArray<int>>` consente al compilatore di ottimizzare drasticamente il codice e nessuna ricerca di tabelle virtuali avviene al momento dell'esecuzione.

Limitazioni: Poiché la classe base è basata su modelli e diversa per due diversi `DynArray` , non è possibile archiviare i puntatori alle loro classi base in un array omogeneo di tipo come generalmente si potrebbe fare con l'ereditarietà normale in cui la classe base non dipende dal

derivato genere:

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

A* a = new B;
```

CRTP per evitare la duplicazione del codice

L'esempio in Visitor Pattern fornisce un caso d'uso convincente per CRTP:

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

struct Square : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};
```

Ogni tipo di bambino di `IShape` deve implementare la stessa funzione allo stesso modo. Questo è un sacco di digitazione extra. Invece, possiamo introdurre un nuovo tipo nella gerarchia che fa questo per noi:

```
template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // visit with our exact type
        visitor.visit(*static_cast<Derived const*>(this));
    }
};
```

E ora, ogni forma deve semplicemente ereditare dall'accettore:

```
struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double radius;
};
```

```
struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength)
{}
    Point topLeft;
    double sideLength;
};
```

Nessun codice duplicato necessario.

Leggi Modello di modello curiosamente ricorrente (CRTP) online:
https://riptutorial.com/it/cplusplus/topic/9269/modello-di-modello-curiosamente-ricorrente--crtp-

Capitolo 71: mutex

Osservazioni

È meglio usare *std :: shared_mutex* di *std :: shared_timed_mutex* .

La differenza di prestazioni è più del doppio.

Se vuoi usare RWLock, troverai che ci sono due opzioni.
È std :: shared_mutex e shared_timed_mutex.
potresti pensare che std :: shared_timed_mutex sia solo la versione 'std :: shared_mutex + time method'.

Ma l'implementazione è completamente diversa.

Il codice seguente è l'implementazione di MSVC14.1 di std :: shared_mutex.

```
class shared_mutex
{
public:
typedef _Smtx_t * native_handle_type;

shared_mutex() _NOEXCEPT
    : _Myhandle(0)
    {    // default construct
    }

~shared_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock() _NOEXCEPT
    {    // lock exclusive
    _Smtx_lock_exclusive(&_Myhandle);
    }

bool try_lock() _NOEXCEPT
    {    // try to lock exclusive
    return (_Smtx_try_lock_exclusive(&_Myhandle) != 0);
    }

void unlock() _NOEXCEPT
    {    // unlock exclusive
    _Smtx_unlock_exclusive(&_Myhandle);
    }

void lock_shared() _NOEXCEPT
```

```
    {    // lock non-exclusive
    _Smtx_lock_shared(&_Myhandle);
    }

bool try_lock_shared() _NOEXCEPT
    {    // try to lock non-exclusive
    return (_Smtx_try_lock_shared(&_Myhandle) != 0);
    }

void unlock_shared() _NOEXCEPT
    {    // unlock non-exclusive
    _Smtx_unlock_shared(&_Myhandle);
    }

native_handle_type native_handle() _NOEXCEPT
    {    // get native handle
    return (&_Myhandle);
    }

shared_mutex(const shared_mutex&) = delete;
shared_mutex& operator=(const shared_mutex&) = delete;
private:
    _Smtx_t _Myhandle;
};

void __cdecl _Smtx_lock_exclusive(_Smtx_t * smtx)
{    /* lock shared mutex exclusively */
AcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_lock_shared(_Smtx_t * smtx)
{    /* lock shared mutex non-exclusively */
AcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}

int __cdecl _Smtx_try_lock_exclusive(_Smtx_t * smtx)
{    /* try to lock shared mutex exclusively */
return (TryAcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx)));
}

int __cdecl _Smtx_try_lock_shared(_Smtx_t * smtx)
{    /* try to lock shared mutex non-exclusively */
return (TryAcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx)));
}

void __cdecl _Smtx_unlock_exclusive(_Smtx_t * smtx)
{    /* unlock exclusive shared mutex */
ReleaseSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_unlock_shared(_Smtx_t * smtx)
{    /* unlock non-exclusive shared mutex */
ReleaseSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}
```

Puoi vedere che std :: shared_mutex è implementato in Windows Slim Reader / Write Locks (https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa904937(v=vs.85).aspx)

Ora esaminiamo l'implementazione di std :: shared_timed_mutex.

Il codice seguente è l'implementazione di MSVC14.1 di std :: shared_timed_mutex.

```
class shared_timed_mutex
{
typedef unsigned int _Read_cnt_t;
static constexpr _Read_cnt_t _Max_readers = _Read_cnt_t(-1);
public:
shared_timed_mutex() _NOEXCEPT
    : _Mymtx(), _Read_queue(), _Write_queue(),
        _Readers(0), _Writing(false)
    {    // default construct
    }

~shared_timed_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock()
    {    // lock exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing)
        _Write_queue.wait(_Lock);
    _Writing = true;
    while (0 < _Readers)
        _Read_queue.wait(_Lock);    // wait for writing, no readers
    }

bool try_lock()
    {    // try to lock exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || 0 < _Readers)
        return (false);
    else
        {    // set writing, no readers
        _Writing = true;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock for duration
    return (try_lock_until(chrono::steady_clock::now() + _Rel_time));
    }

template<class _Clock,
    class _Duration>
    bool try_lock_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock until time point
    auto _Not_writing = [this] { return (!_Writing); };
    auto _Zero_readers = [this] { return (_Readers == 0); };
    unique_lock<mutex> _Lock(_Mymtx);

    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Not_writing))
        return (false);
```

```
    _Writing = true;

    if (!_Read_queue.wait_until(_Lock, _Abs_time, _Zero_readers))
        {    // timeout, leave writing state
        _Writing = false;
        _Lock.unlock();    // unlock before notifying, for efficiency
        _Write_queue.notify_all();
        return (false);
        }

    return (true);
    }

void unlock()
    {    // unlock exclusive
        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);

        _Writing = false;
        }

    _Write_queue.notify_all();
    }

void lock_shared()
    {    // lock non-exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing || _Readers == _Max_readers)
        _Write_queue.wait(_Lock);
    ++_Readers;
    }

bool try_lock_shared()
    {    // try to lock non-exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || _Readers == _Max_readers)
        return (false);
    else
        {    // count another reader
        ++_Readers;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_shared_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock non-exclusive for relative time
    return (try_lock_shared_until(_Rel_time
        + chrono::steady_clock::now()));
    }

template<class _Time>
    bool _Try_lock_shared_until(_Time _Abs_time)
    {    // try to lock non-exclusive until absolute time
    auto _Can_acquire = [this] {
        return (!_Writing && _Readers < _Max_readers); };

    unique_lock<mutex> _Lock(_Mymtx);
```

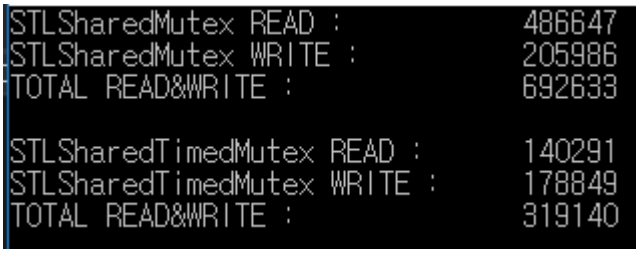
```
    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Can_acquire))
        return (false);

    ++_Readers;
    return (true);
    }

template<class _Clock,
    class _Duration>
    bool try_lock_shared_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

bool try_lock_shared_until(const xtime *_Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

void unlock_shared()
    {    // unlock non-exclusive
    _Read_cnt_t _Local_readers;
    bool _Local_writing;

        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);
        --_Readers;
        _Local_readers = _Readers;
        _Local_writing = _Writing;
        }

    if (_Local_writing && _Local_readers == 0)
        _Read_queue.notify_one();
    else if (!_Local_writing && _Local_readers == _Max_readers - 1)
        _Write_queue.notify_all();
    }

shared_timed_mutex(const shared_timed_mutex&) = delete;
shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
private:
mutex _Mymtx;
condition_variable _Read_queue, _Write_queue;
_Read_cnt_t _Readers;
bool _Writing;
};

class stl_condition_variable_win7 final : public stl_condition_variable_interface
{
public:
    stl_condition_variable_win7()
    {
        __crtInitializeConditionVariable(&m_condition_variable);
    }

    ~stl_condition_variable_win7() = delete;
    stl_condition_variable_win7(const stl_condition_variable_win7&) = delete;
    stl_condition_variable_win7& operator=(const stl_condition_variable_win7&) = delete;

    virtual void destroy() override {}
```

```
    virtual void wait(stl_critical_section_interface *lock) override
    {
        if (!stl_condition_variable_win7::wait_for(lock, INFINITE))
            std::terminate();
    }

    virtual bool wait_for(stl_critical_section_interface *lock, unsigned int timeout) override
    {
        return __crtSleepConditionVariableSRW(&m_condition_variable,
static_cast<stl_critical_section_win7 *>(lock)->native_handle(), timeout, 0) != 0;
    }

    virtual void notify_one() override
    {
        __crtWakeConditionVariable(&m_condition_variable);
    }

    virtual void notify_all() override
    {
        __crtWakeAllConditionVariable(&m_condition_variable);
    }

private:
    CONDITION_VARIABLE m_condition_variable;
};
```

Puoi vedere che std :: shared_timed_mutex è implementato in std :: condition_value.

Questa è un'enorme differenza.

Quindi controlliamo le prestazioni di due di loro.



Questo è il risultato del test di lettura / scrittura per 1000 millisecondi.

std :: shared_mutex ha elaborato read / write più di 2 volte rispetto a std :: shared_timed_mutex.

In questo esempio, il rapporto di lettura / scrittura è lo stesso, ma la frequenza di lettura è più frequente della velocità di scrittura in tempo reale.
Pertanto, la differenza di prestazioni può essere maggiore.

il codice qui sotto è il codice in questo esempio.

```
void useSTLSharedMutex()
{
    std::shared_mutex shared_mtx_lock;
```

```
    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;

                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;
```

```
    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedMutex READ :           " << readProcessedCnt << std::endl;
    std::cout << "STLSharedMutex WRITE :          " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}

void useSTLSharedTimedMutex()
{
    std::shared_timed_mutex shared_mtx_lock;

    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;
```

```
                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;

    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedTimedMutex READ :      " << readProcessedCnt << std::endl;
    std::cout << "STLSharedTimedMutex WRITE :     " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}
```

Examples

std :: unique_lock, std :: shared_lock, std :: lock_guard

Utilizzato per l'acquisizione in stile RAII di blocchi di prova, serrature temporizzate di prova e serrature ricorsive.

`std::unique_lock` consente la proprietà esclusiva dei mutex.

`std::shared_lock` consente la proprietà condivisa dei mutex. Diversi thread possono contenere `std::shared_locks` su uno `std::shared_mutex` . Disponibile da C ++ 14.

`std::lock_guard` è un'alternativa leggera a `std::unique_lock` e `std::shared_lock` .

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it =  _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
```

```
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string,std::string>  _phonebook;
};
```

Strategie per le classi di blocco: std :: try_to_lock, std :: adopt_lock, std :: defer_lock

Quando si crea uno std :: unique_lock, ci sono tre diverse strategie di blocco tra cui scegliere: `std::try_to_lock` , `std::defer_lock` e `std::adopt_lock`

1. `std::try_to_lock` consente di provare un blocco senza bloccare:

```
{
    std::atomic_int temp {0};
    std::mutex _mutex;

    std::thread t( [&](){

        while( temp!= -1){
            std::this_thread::sleep_for(std::chrono::seconds(5));
            std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

            if(lock.owns_lock()){
                //do something
                temp=0;
            }
        }
    });

    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if(lock.owns_lock()){
            if (temp < INT_MAX){
                ++temp;
            }
            std::cout << temp << std::endl;
        }
    }
}
```

2. `std::defer_lock` consente di creare una struttura di blocco senza acquisire il blocco. Quando si blocca più di un mutex, esiste una finestra di opportunità per un deadlock se due chiamanti di funzione cercano di acquisire i blocchi contemporaneamente:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
```

```
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    lock1.lock()
    lock2.lock(); // deadlock here
    std::cout << "Locked! << std::endl;
    //...
}
```

Con il seguente codice, qualunque cosa accada nella funzione, i blocchi vengono acquisiti e rilasciati nell'ordine appropriato:

```
   {
       std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
       std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
       std::lock(lock1,lock2); // no deadlock possible
       std::cout << "Locked! << std::endl;
       //...

   }
```

3. `std::adopt_lock` non tenta di bloccarsi una seconda volta se il thread chiamante attualmente possiede il lock.

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
    std::cout << "Locked! << std::endl;
    //...
}
```

Qualcosa da tenere a mente è che std :: adopt_lock non è un sostituto per l'uso mutex ricorsivo. Quando il blocco esce dall'ambito, il mutex viene **rilasciato** .

std :: mutex

std :: mutex è una struttura di sincronizzazione semplice e non ricorsiva che viene utilizzata per proteggere i dati a cui si accede da più thread.

```
    std::atomic_int temp{0};
    std::mutex _mutex;

    std::thread t( [&](){

                      while( temp!= -1){
                          std::this_thread::sleep_for(std::chrono::seconds(5));
                          std::unique_lock<std::mutex> lock( _mutex);

                              temp=0;
                      }
                  });


    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
```

```
        if ( temp < INT_MAX )
            temp++;
        cout << temp << endl;

    }
```

std :: scope_lock (C ++ 17)

`std::scoped_lock` fornisce semantica di stile RAII per possedere un altro mutex, combinato con gli algoritmi di lock lock usati da `std::lock` . Quando `std::scoped_lock` viene distrutto, i mutex vengono rilasciati nell'ordine inverso rispetto al quale sono stati acquisiti.

```
{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}
```

Tipi di mutex

C ++ 1x offre una selezione di classi mutex:

- std :: mutex - offre funzionalità di blocco semplici.
- std :: timed_mutex - offre la funzionalità try_to_lock
- std :: recursive_mutex - consente il blocco ricorsivo con lo stesso thread.
- std :: shared_mutex, std :: shared_timed_mutex - offre funzionalità di blocco condivise e univoche.

std :: blocco

`std::lock` utilizza algoritmi di deadlock avoidance per bloccare uno o più mutex. Se durante una chiamata viene generata un'eccezione per bloccare più oggetti, `std::lock` sblocca gli oggetti bloccati correttamente prima di rilanciare l'eccezione.

```
std::lock(_mutex1, _mutex2);
```

Leggi mutex online: https://riptutorial.com/it/cplusplus/topic/9895/mutex

Capitolo 72: Mutex ricorsivo

Examples

std :: recursive_mutex

Il mutex ricorsivo consente allo stesso thread di bloccare ricorsivamente una risorsa - fino a un limite non specificato.

Ci sono pochissime giustificazioni in termini reali per questo. Alcune implementazioni complesse potrebbero dover chiamare una copia sovraccaricata di una funzione senza rilasciare il blocco.

```
    std::atomic_int temp{0};
    std::recursive_mutex _mutex;

    //launch_deferred launches asynchronous tasks on the same thread id

    auto future1 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;

                    std::this_thread::sleep_for(std::chrono::seconds(3));
                    std::unique_lock<std::recursive_mutex> lock( _mutex);
                    temp=0;

                });

    auto future2 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;
                    while ( true )
                    {
                        std::this_thread::sleep_for(std::chrono::milliseconds(1));
                        std::unique_lock<std::recursive_mutex> lock( _mutex,
std::try_to_lock);
                        if ( temp < INT_MAX )
                             temp++;

                        cout << temp << endl;

                    }
                });
    future1.get();
    future2.get();
```

Leggi Mutex ricorsivo online: https://riptutorial.com/it/cplusplus/topic/9929/mutex-ricorsivo

Capitolo 73: Namespace

introduzione

Utilizzato per prevenire conflitti di nomi quando si utilizzano più librerie, uno spazio dei nomi è un prefisso dichiarativo per funzioni, classi, tipi, ecc.

Sintassi

- *identificatore di* spazio dei nomi (*opt*) { *declaration-seq* }
- *identificatore di* spazio dei nomi in linea (*opt*) { *declaration-seq* } / * dal C ++ 11 * /
- inline (*opt*) namespace *attributo- identificatore -seq identificatore* (*opt*) { *declaration-seq* } / * dal C ++ 17 * /
- namespace *enclosing-namespace-specificifier* :: *identificatore* { *declaration-seq* } / * dal C ++ 17 * /
- *identificatore dello* spazio dei nomi = *specificatore del namespace qualificato* ;
- usando namespace *nested-name-specifier* (*opt*) *nome-spazio-nome* ;
- *attribute-specificatore-seq* utilizzando namespace *nested-name-specificatore* (*opt*) *nome-spazio-nome* ; / * dal C ++ 11 * /

Osservazioni

Lo `namespace` parole chiave ha tre significati diversi a seconda del contesto:

1. Se seguito da un nome facoltativo e da una sequenza di dichiarazioni racchiuse tra parentesi, definisce un nuovo spazio dei nomi o estende uno spazio dei nomi esistente con tali dichiarazioni. Se il nome è omesso, lo spazio dei nomi è uno spazio dei nomi senza nome .

2. Quando seguito da un nome e un segno di uguale, dichiara un alias di namespace .

3. Quando è preceduto `using` e seguito da un nome di spazio dei nomi, forma una *direttiva using* , che consente di trovare nomi nello spazio dei nomi dato da una ricerca di nome non qualificata (ma non ridichiara quei nomi nell'ambito corrente). Una *direttiva using* non può verificarsi nell'ambito della classe.

`using namespace std;` è scoraggiato Perché? Perché lo `namespace std` è enorme! Ciò significa che c'è un'alta probabilità che i nomi entrino in collisione:

```
//Really bad!
using namespace std;

//Calculates p^e and outputs it to std::cout
void pow(double p, double e) { /*...*/ }

//Calls pow
```

```
pow(5.0, 2.0); //Error! There is already a pow function in namespace std with the same
signature,
               //so the call is ambiguous
```

Examples

Cosa sono gli spazi dei nomi?

Uno spazio dei nomi C ++ è una raccolta di entità C ++ (funzioni, classi, variabili), i cui nomi sono preceduti dal nome del namespace. Quando si scrive codice all'interno di uno spazio dei nomi, le entità denominate appartenenti a tale spazio dei nomi non devono essere precedute dal nome del namespace, ma le entità esterne devono utilizzare il nome completo. Il nome completo ha il formato `<namespace>::<entity>` . Esempio:

```
namespace Example
{
  const int test = 5;

  const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

I namespace sono utili per raggruppare insieme le definizioni correlate. Prendi l'analogia di un centro commerciale. Generalmente un centro commerciale è suddiviso in diversi negozi, ogni negozio vende articoli da una specifica categoria. Un negozio potrebbe vendere prodotti elettronici, mentre un altro negozio potrebbe vendere scarpe. Queste separazioni logiche nei tipi di negozi aiutano gli acquirenti a trovare gli articoli che stanno cercando. I namespace aiutano i programmatori c ++, come gli acquirenti, a trovare le funzioni, le classi e le variabili che stanno cercando organizzandole in modo logico. Esempio:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
```

```
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

C'è un singolo spazio dei nomi predefinito, che è lo spazio dei nomi globale che non ha nome, ma può essere indicato da `::` . Esempio:

```
void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar();   // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}
```

Creare spazi dei nomi

Creare un namespace è davvero semplice:

```
//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}
```

Per chiamare la `bar` , devi prima specificare lo spazio dei nomi, seguito dall'operatore di risoluzione dell'ambito `::`

```
Foo::bar();
```

È consentito creare uno spazio dei nomi in un altro, ad esempio:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
}
```

C ++ 17

Il codice sopra riportato potrebbe essere semplificato come segue:

```
namespace A::B::C
{
    void bar() {}
}
```

Estendere spazi dei nomi

Un'utile caratteristica dello `namespace` dei `namespace` è che è possibile espanderli (aggiungere membri ad esso).

```
namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}
```

Usando la direttiva

La parola chiave 'using' ha tre sapori. Combinato con la parola chiave 'namespace' scrivi una 'direttiva using':

Se non vuoi scrivere `Foo::` davanti a tutte le cose nel namespace `Foo` , puoi usare `using namespace Foo;` per importare ogni singola cosa da `Foo` .

```
namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

È anche possibile importare entità selezionate in uno spazio dei nomi anziché nell'intero spazio dei nomi:

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

Una parola di cautela: l' `using namespace` nei file di intestazione è visto come uno stile sbagliato nella maggior parte dei casi. Se ciò è fatto, lo spazio dei nomi viene importato in *ogni* file che

include l'intestazione. Poiché non c'è modo di "non `using` " uno spazio dei nomi, questo può portare all'inquinamento dello spazio dei nomi (simboli più o inaspettati nello spazio dei nomi globale) o, peggio, ai conflitti. Vedi questo esempio per un'illustrazione del problema:

```
/***** foo.h *****/
namespace Foo
{
    class C;
}

/***** bar.h *****/
namespace Bar
{
    class C;
}

/***** baz.h *****/
#include "foo.h"
using namespace Foo;

/***** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

Una *direttiva using* non può verificarsi nell'ambito della classe.

Ricerca dipendente dall'argomento

Quando si chiama una funzione senza un qualificatore di namespace esplicito, il compilatore può scegliere di chiamare una funzione all'interno di uno spazio dei nomi se uno dei tipi di parametro di quella funzione è anche in quello spazio dei nomi. Questo è chiamato "Argument Dipendent Lookup", o ADL:

```
namespace Test
{
  int call(int i);

  class SomeClass {...};

  int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); //Succeeds
```

`call` fallisce perché nessuno dei suoi tipi di parametro proviene dallo spazio dei nomi `Test` . `call_too` funziona perché `SomeClass` è un membro di `Test` e pertanto si qualifica per le regole ADL.

Quando non si verifica ADL

ADL non si verifica se la normale ricerca non qualificata trova un membro della classe, una funzione che è stata dichiarata nell'ambito di un blocco o qualcosa che non è di tipo funzione. Per esempio:

```
void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});       // error: ADL is disabled and ::foo() doesn't take any arguments
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}
```

Spazio dei nomi in linea

C ++ 11

`inline namespace` include il contenuto dello `inline namespace` include, quindi

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

è per lo più equivalente a

```
namespace Outer
{

    namespace Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

ma l'elemento di `Outer::Inner::` e quelli associati a `Outer::` sono identici.

Quindi seguire è equivalente

```
Outer::foo();
Outer::Inner::foo();
```

L'alternativa `using namespace Inner;` non sarebbe equivalente per alcune parti difficili come la specializzazione template:

Per

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- Lo spazio dei nomi in linea consente la specializzazione di `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      inline namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
  }
  ```

- Mentre l' `using namespace` non consente la specializzazione di `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
      using namespace Inner;
      // Specialization of `Outer::foo` is not possible
      // it should be `Outer::Inner::foo`.
  }
  ```

Lo spazio dei nomi in linea è un modo per consentire a più versioni di convivere e l'impostazione predefinita a quella in `inline`

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }

}
```

E con l'uso

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Spazi dei nomi senza nome / anonimi

È possibile utilizzare uno spazio dei nomi senza nome per garantire che i nomi abbiano un collegamento interno (è possibile fare riferimento solo all'unità di traduzione corrente). Tale spazio dei nomi è definito allo stesso modo di qualsiasi altro spazio dei nomi, ma senza il nome:

```
namespace {
    int foo = 42;
}
```

`foo` è visibile solo nell'unità di traduzione in cui appare.

Si consiglia di non utilizzare mai spazi dei nomi non denominati nei file di intestazione in quanto fornisce una versione del contenuto per ogni unità di traduzione in cui è inclusa. Ciò è particolarmente importante se si definiscono globali non costanti.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

Spazi dei nomi nidificati compatti

C ++ 17

```
namespace a {
  namespace b {
    template<class T>
    struct qualifies : std::false_type {};
  }
}

namespace other {
  struct bob {};
}

namespace a::b {
  template<>
  struct qualifies<::other::bob> : std::true_type {};
}
```

È possibile inserire sia l' `a` e `b` spazi dei nomi in un unico passaggio con `namespace a::b` a partire dal C ++ 17.

Aliasing di un lungo spazio dei nomi

Questo di solito è usato per rinominare o accorciare lunghi riferimenti di namespace come riferimenti a componenti di una libreria.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;


//    Both Type declarations are equivalent
boost::multiprecision::Number X   //    Writing the full namespace path, longer
Name1::Number Y                   //    using the name alias, shorter
```

Ambito di dichiarazione alias

La Dichiarazione alias è influenzata dalle precedenti dichiarazioni *usando*

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}


using namespace boost;
```

```
//   Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

Tuttavia, è più facile confondersi su quale spazio dei nomi stai facendo aliasing quando hai qualcosa del genere:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;

//    Not recommended as
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Alias dello spazio dei nomi

Ad uno spazio dei nomi può essere assegnato un alias (*ovvero* un altro nome per lo stesso spazio dei nomi) utilizzando la sintassi dello `namespace` dei `namespace` *identificatore* `=` . È possibile accedere ai membri dello spazio dei nomi con alias qualificandoli con il nome dell'alias. Nell'esempio seguente, lo spazio `AReallyLongName::AnotherReallyLongName` nomi nidificato `AReallyLongName::AnotherReallyLongName` è scomodo da digitare, quindi la funzione `qux` locale dichiara un alias `N` È possibile accedere ai membri di tale spazio dei nomi semplicemente utilizzando `N::` .

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
```

```
    N::baz(N::foo(), N::bar());
}
```

Leggi Namespace online: https://riptutorial.com/it/cplusplus/topic/495/namespace

Capitolo 74: Oggetti callable

introduzione

Gli oggetti callable sono la raccolta di tutte le strutture C ++ che possono essere utilizzate come una funzione. In pratica, questo è tutto ciò che è possibile passare alla funzione STL di C ++ 17 invoke () o che può essere utilizzata nel costruttore di std :: function, questo include: Puntatori di funzione, Classi con operatore (), Classi con implicito conversioni, riferimenti a funzioni, puntatori a funzioni membro, puntatori a dati membro, lambda. Gli oggetti callable sono usati in molti algoritmi STL come predicato.

Osservazioni

Un discorso molto utile di Stephan T. Lavavej (<functional>: What's New, And Proper Use) (Slides) porta alla base di questa documentazione.

Examples

Puntatori di funzione

I puntatori di funzione sono il modo più semplice per passare le funzioni in giro, che può anche essere usato in C. (Per ulteriori dettagli, consultare la documentazione C).

Ai fini degli oggetti chiamabili, un puntatore a funzione può essere definito come:

```
typedef returnType(*name)(arguments);                        // All
using name = returnType(*)(arguments);                       // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>;      // <= C++14
```

Se dovessimo usare un puntatore a funzione per scrivere il nostro ordinamento vettoriale, sembrerebbe:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function
```

```
struct GreaterThanInt {
   static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member
function
```

In alternativa, avremmo potuto richiamare il puntatore alla funzione in uno dei seguenti modi:

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

Classi con operatore () (Funzionalità)

Ogni classe che sovraccarica l' `operator()` può essere utilizzata come oggetto funzione. Queste classi possono essere scritte a mano (spesso denominate funtori) o generate automaticamente dal compilatore scrivendo Lambdas da C ++ 11 in poi.

```
struct Person {
    std::string name;
    unsigned int age;
};

// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
         return person.name == _name;
    }
private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...
```

Poiché i funtori hanno una propria identità, non possono essere inseriti in un typedef e questi devono essere accettati tramite argomenti del template. La definizione di `std::find_if` può essere simile a:

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
     for (Iterator i = begin, i != end, ++i)
         if (predicate(*i))
             return i;
     return end;
}
```

Da C ++ 17 in poi, la chiamata del predicato può essere eseguita con invoke:
`std::invoke(predicate, *i) .`

Leggi Oggetti callable online: https://riptutorial.com/it/cplusplus/topic/6073/oggetti-callable

Capitolo 75: Operatori di bit

Osservazioni

Le operazioni di spostamento dei bit non sono trasferibili su tutte le architetture dei processori, i processori diversi possono avere diverse larghezze di bit. In altre parole, se hai scritto

```
int a = ~0;
int b = a << 1;
```

Questo valore sarebbe diverso su una macchina a 64 bit rispetto a una macchina a 32 bit o su un processore basato su x86 su un processore basato su PIC.

L'endianità non ha bisogno di essere presa in considerazione per le operazioni bit-wise stesse, cioè lo spostamento a destra (`>>`) sposterà i bit verso il bit meno significativo e uno XOR eseguirà un esclusivo o sui bit. L'endianità deve essere presa in considerazione solo con i dati stessi, ovvero se l'endianità è una preoccupazione per l'applicazione, è una preoccupazione indipendentemente dalle operazioni bit-saggio.

Examples

& - AND bit a bit

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Produzione

```
a = 6, b = 10, c = 2
```

Perché

Un po 'saggio `AND` opera a livello di bit e utilizza la seguente tabella di verità booleana:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

Quando il valore binario per `a` (`0110`) e il valore binario per `b` (`1010`) sono `AND` 'ed insieme otteniamo il valore binario di `0010` :

```
int a = 0 1 1 0
int b = 1 0 1 0 &
        ---------
int c = 0 0 1 0
```

Il bit Wise AND non modifica il valore dei valori originali se non è specificamente assegnato all'utilizzo dell'operatore composto di assegnazione bit saggio `&=` :

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

| - OR bit a bit

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Produzione

```
a = 5, b = 12, c = 13
```

Perché

Un po 'saggio `OR` opera a livello di bit e usa la seguente tabella di verità booleana:

```
true OR true = true
true OR false = true
false OR false = false
```

Quando il valore binario di `a` (`0101`) e il valore binario di `b` (`1100`) sono `OR` "ed insieme otteniamo il valore binario di `1101` :

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

L'OR bit-saggio non modifica il valore dei valori originali se non è specificamente assegnato all'utilizzo dell'operatore composto assegnazione bit saggio `|=` :

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

^ - XOR bit a bit (OR esclusivo)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Produzione

```
a = 5, b = 9, c = 12
```

Perché

Un po 'saggio `XOR` (esclusivo o) opera a livello di bit e utilizza la seguente tabella di verità booleana:

```
true OR true = false
true OR false = true
false OR false = false
```

Si noti che con un'operazione XOR `true OR true = false` dove, come con le operazioni `true AND/OR true = true` , quindi la natura esclusiva dell'operazione XOR.

Usando questo, quando il valore binario di `a` (`0101`) e il valore binario di `b` (`1001`) sono `XOR` 'ed insieme otteniamo il valore binario di `1100` :

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

Il bit XOR bit non modifica il valore dei valori originali se non è specificamente assegnato all'utilizzo dell'operatore composto assegnazione bit saggio `^=` :

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

Il bit XOR può essere utilizzato in molti modi ed è spesso utilizzato nelle operazioni di bit mask per la crittografia e la compressione.

Nota: il seguente esempio viene spesso mostrato come esempio di un bel trucco. Ma non dovrebbe essere usato nel codice di produzione (ci sono modi migliori per `std::swap()` per ottenere lo stesso risultato).

Puoi anche utilizzare un'operazione XOR per scambiare due variabili senza un temporaneo:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

Per la produzione di questo è necessario aggiungere un assegno per assicurarsi che possa essere utilizzato.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
```

```
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

Quindi anche se sembra un bel trucco in isolamento non è utile nel codice reale. xor non è un'operazione logica di base, ma una combinazione di altre: a ^ c = ~ (a & c) & (a | c)

anche nel 2015 le variabili dei compilatori possono essere assegnate come binarie:

```
int cn=0b0111;
```

~ - bit per bit NOT (complemento unario)

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)

std::cout << "a = " << static_cast<int>(a) <<
             ", b = " << static_cast<int>(b) << std::endl;
```

Produzione

```
a = 234, b = 21
```

Perché

Un po 'saggio `NOT` (complemento unario) opera a livello di bit e semplicemente capovolge ogni bit. Se è un `1` , è cambiato in uno `0` , se è uno `0` , è cambiato in un `1` . Il bit saggio NON ha lo stesso effetto di XOR un valore rispetto al valore massimo per un tipo specifico:

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)
unsigned char c = a ^ ~0;
```

Il bit bit NOT può anche essere un modo conveniente per verificare il valore massimo per un tipo specifico integrale:

```
unsigned int i = ~0;
unsigned char c = ~0;

std::cout << "max uint = " << i << std::endl <<
             "max uchar = " << static_cast<short>(c) << std::endl;
```

Il bit-saggio NOT non cambia il valore del valore originale e non ha un operatore di assegnazione composto, quindi non puoi fare `a ~= 10` per esempio.

Il *bit saggio* NON (`~`) non deve essere confuso con il *logico* NOT (`!`); dove un po 'saggio NON

capovolgerà ogni bit, un NOT logico userà l'intero valore per eseguire la sua operazione, in altre parole `(!1) != (~1)`

<< - spostamento a sinistra

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Produzione

```
a = 1, b = 2
```

Perché

Il bit shift sinistro sposterà i bit del valore della mano sinistra (`a`) il numero specificato a destra (`1`), in pratica riempendo i bit meno significativi con 0, quindi spostando il valore di `5` (binario `0000 0101`) a sinistra 4 volte (es. `5 << 4`) produrrà il valore di `80` (binario `0101 0000`). È possibile notare che lo spostamento di un valore a sinistra 1 volta è uguale a moltiplicare il valore per 2, ad esempio:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

Ma va notato che l'operazione di spostamento a sinistra sposta *tutti i* bit a sinistra, incluso il bit di segno, ad esempio:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Uscita possibile: `a = 2147483647, b = -2`

Mentre alcuni compilatori produrranno risultati che sembrano prevedibili, è opportuno notare che se si lascia un numero con segno maiuscolo in modo che il bit del segno sia interessato, il risultato **non** è **definito** . È anche **indefinito** se il numero di bit che desideri spostare è un numero negativo o è maggiore del numero di bit che il tipo a sinistra può contenere, ad esempio:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

Il cambio bit left shift non cambia il valore dei valori originali a meno che non sia specificamente assegnato all'utilizzo dell'operatore composto assegnazione bit saggio `<<=` :

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

>> - spostamento a destra

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Produzione

```
a = 2, b = 1
```

Perché

Lo spostamento bit destro corretto sposterà i bit del valore della mano sinistra (`a`) il numero specificato a destra (`1`); va notato che mentre l'operazione di un passaggio a destra è standard, ciò che accade ai bit di uno spostamento a destra su un numero *negativo con* segno è *definito dall'implementazione* e quindi non può essere garantito che sia portatile, ad esempio:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

È anche indefinito se il numero di bit che desideri spostare è un numero negativo, ad esempio:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

Il bit right shift non cambia il valore dei valori originali a meno che non sia specificatamente assegnato all'operatore di assegnazione bit saggio bit `>>=` :

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Leggi Operatori di bit online: https://riptutorial.com/it/cplusplus/topic/2572/operatori-di-bit

Capitolo 76: Ordinamento

Osservazioni

La famiglia di funzioni `std::sort` si trova nella libreria degli `algorithm` .

Examples

Ordinamento di contenitori di sequenza con ordinamento specificato

Se i valori in un contenitore hanno già alcuni operatori già sovraccaricati, `std::sort` può essere utilizzato con i funtori specializzati per ordinare in ordine ascendente o discendente:

C ++ 11

```
#include <vector>
#include <algorithm>
#include <functional>

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

//sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

//sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

//Or just:
std::sort(v.rbegin(), v.rend());
```

C ++ 14

In C ++ 14, non è necessario fornire l'argomento modello per gli oggetti funzione di confronto e lasciare che l'oggetto si deduca in base a ciò che viene passato in:

```
std::sort(v.begin(), v.end(), std::less<>());     // ascending order
std::sort(v.begin(), v.end(), std::greater<>());  // descending order
```

Ordinare i contenitori della sequenza con un operatore meno carico

Se non viene passata alcuna funzione di ordinamento, `std::sort` ordinerà gli elementi chiamando l' `operator<` su coppie di elementi, che devono restituire un tipo contestualmente convertibile in `bool` (o semplicemente `bool`). I tipi di base (interi, float, puntatori, ecc.) Sono già stati creati negli operatori di confronto.

Possiamo sovraccaricare questo operatore per far funzionare la chiamata di `sort` predefinita su tipi

definiti dall'utente.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    // Use variable to provide total order operator less
    //`this` always represents the left-hand side of the compare.
    bool operator<(const Base &b) const {
        return this->variable < b.variable;
    }

    int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using operator<(const Base &b) function
    std::sort(vector.begin(), vector.end());
    std::sort(deque.begin(), deque.end());
    // List must be sorted differently due to its design
    list.sort();

    return 0;
}
```

Ordinare i contenitori della sequenza usando la funzione di confronto

```
// Include sequence containers
#include <vector>
#include <deque>
```

```
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
    std::sort(vector.begin(), vector.end(), compare);
    std::sort(deque.begin(), deque.end(), compare);
    list.sort(compare);

    return 0;
}
```

Ordinamento di contenitori di sequenza usando espressioni lambda (C ++ 11)

C ++ 11

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>

// Include sorting algorithm
#include <algorithm>
```

```
class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};


int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector <Base> vector = {a, b};
    std::deque <Base> deque = {a, b};
    std::list <Base> list = {a, b};
    std::array <Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
      [](const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [](const Base &a, const Base &b) {
                    return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}
```

Contenitori di ordinamento e sequenza

`std::sort` , trovato `algorithm` intestazione della libreria standard, è un algoritmo di libreria standard per l'ordinamento di un intervallo di valori, definito da una coppia di iteratori. `std::sort` prende come ultimo parametro un funtore usato per confrontare due valori; questo è il modo in cui determina l'ordine. Nota che `std::sort` non è stabile .

La funzione di confronto *deve* imporre un ordinamento rigoroso e debole sugli elementi. Un semplice confronto inferiore a (o maggiore di) sarà sufficiente.

Un contenitore con iteratori ad accesso casuale può essere ordinato usando l'algoritmo `std::sort` :

C ++ 11

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

```
std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` richiede che i suoi iteratori siano iteratori ad accesso casuale. I contenitori di sequenza `std::list` e `std::forward_list` (che richiedono C ++ 11) non forniscono iteratori ad accesso casuale, quindi non possono essere usati con `std::sort` . Tuttavia, essi hanno `sort` funzioni membro che implementano un algoritmo di ordinamento che funziona con i propri tipi di iteratori.

C ++ 11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

Le funzioni di `sort` membri `sort` sempre l'intero elenco, in modo che non possano ordinare un sotto-intervallo di elementi. Tuttavia, dal momento che `list` e `forward_list` hanno operazioni di splicing rapido, è possibile estrarre gli elementi da ordinare dall'elenco, ordinarli, quindi reinserirli dove erano abbastanza efficienti in questo modo:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

ordinamento con std :: map (ascendente e discendente)

Questo esempio ordina gli elementi in ordine **crescente** di una **chiave** utilizzando una mappa. Puoi usare qualsiasi tipo, inclusa la classe, invece di `std::string` , nell'esempio seguente.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1,      "Earth"));
    sorted_map.insert(std::make_pair(0.532,  "Mars"));
    sorted_map.insert(std::make_pair(10.97,  "Jupiter"));
```

```
    sorted_map.insert(std::make_pair(9.14,   "Saturn"));
    sorted_map.insert(std::make_pair(3.981,  "Uranus"));
    sorted_map.insert(std::make_pair(3.865,  "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius)" << '\n';
    }
}
```

Produzione:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

Se sono possibili voci con chiavi uguali, utilizzare `multimap` anziché `map` (come nell'esempio seguente).

Per ordinare gli elementi in modo **discendente** , dichiarare la mappa con un corretto functor di comparazione (`std::greater<>`):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6,   "bug"));
    sorted_map.insert(std::make_pair(4,   "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2,   "chicken"));
    sorted_map.insert(std::make_pair(0,   "fish"));
    sorted_map.insert(std::make_pair(4,   "horse"));
    sorted_map.insert(std::make_pair(8,   "spider"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (has " << entry.first << " legs)" << '\n';
    }
}
```

Produzione

```
centipede (has 100 legs)
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
horse (has 4 legs)
```

```
chicken (has 2 legs)
fish (has 0 legs)
```

Ordinamento di array incorporati

L' `sort` algoritmo ordina una sequenza definita da due iteratori. Questo è sufficiente per ordinare un array integrato (noto anche come stile c).

C ++ 11

```
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

Prima di C ++ 11, la fine dell'array doveva essere "calcolata" usando la dimensione dell'array:

C ++ 11

```
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Leggi Ordinamento online: https://riptutorial.com/it/cplusplus/topic/1675/ordinamento

Capitolo 77: Ottimizzazione

introduzione

Durante la compilazione, il compilatore spesso modificherà il programma per aumentare le prestazioni. Questo è permesso dalla regola as-if , che consente tutte le trasformazioni che non cambiano il comportamento osservabile.

Examples

Inline Expansion / Inlining

L'espansione in linea (nota anche come inlining) è l'ottimizzazione del compilatore che sostituisce una chiamata a una funzione con il corpo di quella funzione. Ciò risparmia la funzione di overhead, ma a costo di spazio, poiché la funzione può essere duplicata più volte.

```
// source:

int process(int value)
{
    return 2 * value;
}

int foo(int a)
{
    return process(a);
}

// program, after inlining:

int foo(int a)
{
    return 2 * a; // the body of process() is copied into foo()
}
```

Inlining è più comunemente fatto per piccole funzioni, dove la funzione chiamata overhead è significativa rispetto alla dimensione del corpo della funzione.

Ottimizzazione della base vuota

La dimensione di qualsiasi oggetto o membro subobject è richiesta per essere almeno 1 anche se il tipo è un tipo di `class` vuoto (cioè una `class` o una `struct` che non ha membri di dati non statici), in modo da essere in grado di garantire che gli indirizzi di oggetti distinti dello stesso tipo sono sempre distinti.

Tuttavia, `class` sottooggetti di `class` base non sono così limitati e possono essere completamente ottimizzati dal layout dell'oggetto:

```
#include <cassert>
```

```
struct Base {}; // empty class

struct Derived1 : Base {
    int i;
};

int main() {
    // the size of any object of empty class type is at least 1
    assert(sizeof(Base) == 1);

    // empty base optimization applies
    assert(sizeof(Derived1) == sizeof(int));
}
```

L'ottimizzazione di base vuota viene comunemente utilizzata dalle classi di libreria standard compatibili con allocatore (`std::vector` , `std::function` , `std::shared_ptr` , ecc.) Per evitare di occupare spazio di archiviazione aggiuntivo per il relativo membro allocatore se l'allocatore è stateless. Ciò si ottiene memorizzando uno dei membri di dati richiesti (ad esempio, `begin` , `end` o puntatore di `capacity` per il `vector`).

Riferimento: cppreference

Leggi Ottimizzazione online: https://riptutorial.com/it/cplusplus/topic/9767/ottimizzazione

Capitolo 78: Ottimizzazione in C ++

Examples

Ottimizzazione della classe base vuota

Un oggetto non può occupare meno di 1 byte, in quanto i membri di un array di questo tipo avrebbero lo stesso indirizzo. Quindi `sizeof(T)>=1` sempre valido. È anche vero che una classe derivata non può essere più piccola di *nessuna delle* sue classi base. Tuttavia, quando la classe base è vuota, la sua dimensione non è necessariamente aggiunta alla classe derivata:

```
class Base {};

class Derived : public Base
{
public:
    int i;
};
```

In questo caso, non è necessario allocare un byte per `Base` in `Derived` per avere un indirizzo distinto per tipo per oggetto. Se viene eseguita l'ottimizzazione della classe base vuota (e non è richiesto alcun riempimento), quindi `sizeof(Derived) == sizeof(int)` , cioè, non viene eseguita alcuna allocazione aggiuntiva per la base vuota. Questo è possibile anche con più classi di base (in C ++, le basi multiple non possono avere lo stesso tipo, quindi non ci sono problemi da ciò).

Si noti che questo può essere eseguito solo se il primo membro di `Derived` differisce nel tipo da una qualsiasi delle classi di base. Questo include qualsiasi base comune diretta o indiretta. Se è lo stesso tipo di una delle basi (o c'è una base comune), è necessario allocare almeno un singolo byte per garantire che non ci siano due oggetti distinti dello stesso tipo con lo stesso indirizzo.

Introduzione alle prestazioni

C e C ++ sono noti come linguaggi ad alte prestazioni, in gran parte a causa della notevole quantità di personalizzazione del codice, che consente all'utente di specificare le prestazioni in base alla scelta della struttura.

Quando si ottimizza, è importante fare un benchmark del codice rilevante e capire completamente come verrà utilizzato il codice.

Gli errori di ottimizzazione comuni includono:

- **Ottimizzazione prematura: il** codice complesso può *peggiorare* dopo l'ottimizzazione, sprecando tempo e fatica. La prima priorità dovrebbe essere quella di scrivere codice *corretto* e *gestibile* , piuttosto che codice ottimizzato.
- **Ottimizzazione per il caso d'uso sbagliato: l'** aggiunta di spese generali per l'1% potrebbe non valere il rallentamento per l'altro 99%
- **Micro-ottimizzazione: i** compilatori lo fanno in modo molto efficiente e la micro-

ottimizzazione può anche danneggiare la capacità dei compilatori di ottimizzare ulteriormente il codice

Gli obiettivi di ottimizzazione tipici sono:

- Per fare meno lavoro
- Utilizzare algoritmi / strutture più efficienti
- Per fare un uso migliore dell'hardware

Il codice ottimizzato può avere effetti collaterali negativi, tra cui:

- Maggiore utilizzo della memoria
- Il codice complesso è difficile da leggere o mantenere
- API compromessa e progettazione del codice

Ottimizzazione eseguendo meno codice

L'approccio più diretto all'ottimizzazione consiste nell'eseguire meno codice. Questo approccio di solito dà una velocità fissa senza modificare la complessità temporale del codice.

Anche se questo approccio ti dà una chiara accelerazione, questo darà solo notevoli miglioramenti quando il codice è chiamato molto.

Rimozione del codice inutile

```
void func(const A *a); // Some random function

// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());

// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

C ++ 14

Da C ++ 14, i compilatori possono ottimizzare questo codice per rimuovere l'allocazione e la deallocazione di corrispondenza.

Fare codice solo una volta

```
std::map<std::string, std::unique_ptr<A>> lookup;
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a thirth time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}
```

```
// Within this function, we will have the same noticeable effect as the slow variant while
going at double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

Un approccio simile a questa ottimizzazione può essere utilizzato per implementare una versione stabile di `unique`

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
already in or not
        // This prevents an insertion, which will traverse through the map for every unique
element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Prevenire la ridistribuzione, la copia / lo spostamento inutili

Nell'esempio precedente, abbiamo già impedito le ricerche in std :: set, tuttavia il file `std::vector` contiene ancora un algoritmo in crescita, nel quale dovrà riallocare la sua memoria. Questo può essere evitato prenotando per la giusta dimensione.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the
vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
generated code
    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See example above
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Usare contenitori efficienti

L'ottimizzazione utilizzando le giuste strutture di dati al momento giusto può modificare la complessità temporale del codice.

```
// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Utilizzando un contenitore che utilizza un'implementazione diversa per la memorizzazione dei suoi elementi (contenitore hash anziché albero), possiamo trasformare la nostra implementazione in complessità N. Come effetto collaterale, chiameremo l'operatore di confronto per std :: string less, poiché deve essere chiamato solo quando la stringa inserita deve finire nello stesso bucket.

```
// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Ottimizzazione di piccoli oggetti

L'ottimizzazione degli oggetti piccoli è una tecnica che viene utilizzata all'interno di strutture di dati di basso livello, ad esempio la `std::string` (talvolta denominata Ottimizzazione stringa corta / piccola). È pensato per utilizzare lo spazio di stack come buffer invece di una memoria allocata nel caso in cui il contenuto sia abbastanza piccolo da adattarsi allo spazio riservato.

Aggiungendo overhead di memoria extra e calcoli extra, si tenta di evitare una costosa allocazione dell'heap. I vantaggi di questa tecnica dipendono dall'utilizzo e possono anche danneggiare le prestazioni se utilizzate in modo errato.

Esempio

Un modo molto ingenuo per implementare una stringa con questa ottimizzazione sarebbe il seguente:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};                       ///< Remember if we allocated memory
    char *_buffer{nullptr};                         ///< Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE]= {'\0'};   ///< Stack space used for SMALL OBJECT
OPTIMIZATION

public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
       : _isAllocated(rhs._isAllocated)
       , _buffer(rhs._buffer)
       , _smallBuffer(rhs._smallBuffer) //< Not needed if allocated
    {
        if (_isAllocated)
        {
           // Prevent double deletion of the memory
           rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
    }
    // Other methods, including other constructors, copy constructor,
    // assignment operators have been omitted for readability
};
```

Come puoi vedere nel codice qui sopra, è stata aggiunta una maggiore complessità per evitare alcune operazioni `new` e di `delete` . Inoltre, la classe ha un'impronta di memoria più ampia che potrebbe non essere utilizzata, tranne in un paio di casi.

Spesso si tenta di codificare il valore bool `_isAllocated` , all'interno del pointer `_buffer` con la manipolazione del bit per ridurre la dimensione di una singola istanza (intel 64 bit: potrebbe ridurre le dimensioni di 8 byte). Un'ottimizzazione che è possibile solo quando è noto quali sono le regole di allineamento della piattaforma.

Quando usare?

Poiché questa ottimizzazione aggiunge molta complessità, non è consigliabile utilizzare questa ottimizzazione su ogni singola classe. Si incontrerà spesso in strutture di dati di basso livello di uso comune. Nelle comuni implementazioni di `standard library` C ++ 11 si possono trovare usi in `std::basic_string<>` e `std::function<>` .

Poiché questa ottimizzazione impedisce solo allocazioni di memoria quando i dati memorizzati sono più piccoli del buffer, darà benefici solo se la classe viene spesso utilizzata con dati di piccole dimensioni.

Un ultimo svantaggio di questa ottimizzazione è che è necessario uno sforzo supplementare quando si sposta il buffer, rendendo l'operazione di spostamento più costosa rispetto a quando il buffer non verrebbe utilizzato. Questo è particolarmente vero quando il buffer contiene un tipo non POD.

Leggi Ottimizzazione in C ++ online: https://riptutorial.com/it/cplusplus/topic/4474/ottimizzazione-in-c-plusplus

Capitolo 79: Pacchetti di parametri

Examples

Un modello con un pacchetto di parametri

```
template<class ... Types> struct Tuple {};
```

Un pacchetto di parametri è un parametro del modello che accetta zero o più argomenti del modello. Se un modello ha almeno un pacchetto di parametri è un *modello variadico* .

Espansione di un pacchetto di parametri

Il `parameter_pack ...` viene espanso in un elenco di sostituzioni separate da virgola di `parameter_pack` con ciascuno dei suoi parametri

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
  std::cout << first_argument << "\n";
  variadic_printer(other_arguments...); // Parameter pack expansion
}
```

Il codice sopra richiamato con `variadic_printer(1, 2, 3, "hello");` stampe

```
1
2
3
hello
```

Leggi Pacchetti di parametri online: https://riptutorial.com/it/cplusplus/topic/7668/pacchetti-di-parametri

Capitolo 80: Parola chiave amico

introduzione

Classi ben progettate incapsulano le loro funzionalità, nascondendo la loro implementazione e fornendo un'interfaccia pulita e documentata. Ciò consente la riprogettazione o la modifica finché l'interfaccia rimane invariata.

In uno scenario più complesso, potrebbero essere necessarie più classi che si basano sui dettagli dell'implementazione reciproca. Le classi e le funzioni di amici consentono a questi peer di accedere ai dettagli degli altri, senza compromettere l'incapsulamento e l'occultamento delle informazioni dell'interfaccia documentata.

Examples

Funzione amico

Una classe o una struttura può dichiarare qualsiasi funzione che sia amica. Se una funzione è amica di una classe, può accedere a tutti i suoi membri protetti e privati:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

I modificatori di accesso non alterano la semantica degli amici. Le dichiarazioni pubbliche, protette e private di un amico sono equivalenti.

Le dichiarazioni di amici non sono ereditate. Ad esempio, se eseguiamo la sottoclasse di `PrivateHolder` :

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

e prova ad accedere ai suoi membri, otterremo quanto segue:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
    std::cout << pd.derived_private_value << std::endl;
}
```

Nota che la funzione membro `PrivateHolderDerived` non può accedere a `PrivateHolder::private_value` , mentre la funzione friend può farlo.

Metodo amico

I metodi possono essere dichiarati come amici così come le funzioni:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Classe di amici

Un'intera classe può essere dichiarata come amica. La dichiarazione di classe dell'amico indica che qualsiasi membro dell'amico può accedere ai membri privati e protetti della classe dichiarante:

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};
```

```
class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

La dichiarazione della classe di amici non è riflessiva. Se le classi hanno bisogno di un accesso privato in entrambe le direzioni, entrambe hanno bisogno di dichiarazioni di amici.

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};
```

Leggi Parola chiave amico online: https://riptutorial.com/it/cplusplus/topic/3275/parola-chiave-amico

Capitolo 81: parola chiave const

Sintassi

- const Type myVariable = initial; // Dichiara una variabile const; non può essere cambiato
- const Type & myReference = myVariable; // Dichiara un riferimento a una variabile const
- const Type * myPointer = & myVariable; // Dichiara un puntatore-a-const. Il puntatore può cambiare, ma il membro dati sottostante non può essere modificato tramite il puntatore
- Digita * const myPointer = & myVariable; // Dichiara un puntatore const. Il puntatore non può essere riassegnato per puntare a qualcos'altro, ma il membro dati sottostante può essere modificato
- const Type * const myPointer = & myVariable; // Dichiara un const pointer-to-const.

Osservazioni

Una variabile contrassegnati come `const` non può essere modificata [1]. Il tentativo di chiamare qualsiasi operazione non const su di esso comporterà un errore del compilatore.

1: Beh, può essere cambiato tramite `const_cast` , ma non dovresti quasi mai usarlo

Examples

Const variabili locali

Dichiarazione e uso.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Legame di riferimenti e puntatori

```
int &b = a;       // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;      // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;           // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1            // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;           // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Puntatori Const

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;

//Error: Cannot assign to const pointer
pC = &b;
```

Funzioni membro Const

Le funzioni membro di una classe possono essere dichiarate `const` , che dice al compilatore e ai futuri lettori che questa funzione non modificherà l'oggetto:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

In una funzione membro `const` , `this` puntatore è effettivamente un `const MyClass *` invece di un `MyClass *` . Ciò significa che non è possibile modificare alcuna variabile membro all'interno della funzione; il compilatore emetterà un avvertimento. Quindi `setMyInt` non può essere dichiarato `const` .

Dovresti quasi sempre contrassegnare le funzioni membro come `const` quando possibile. Solo le funzioni membro `const` possono essere chiamate su `const MyClass` .

`static` metodi `static` non possono essere dichiarati come `const` . Questo perché un metodo statico appartiene a una classe e non è chiamato su oggetto; quindi non può mai modificare le variabili interne dell'oggetto. Quindi dichiarare `static` metodi `static` come `const` sarebbe ridondante.

Evitare la duplicazione del codice nei metodi getter const e non-const.

Nei metodi C ++ che differiscono solo dal `const` qualificatore possono essere sovraccaricati. A

volte potrebbe essere necessario disporre di due versioni di getter che restituiscono un riferimento ad un membro.

Lascia che `Foo` sia una classe, che ha due metodi che eseguono operazioni identiche e restituisce un riferimento a un oggetto di tipo `Bar` :

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

L'unica differenza tra loro è che un metodo è non-const e restituisce un riferimento non-const (che può essere utilizzato per modificare l'oggetto) e il secondo è const e restituisce const riferimento.

Per evitare la duplicazione del codice, c'è la tentazione di chiamare un metodo da un altro. Tuttavia, non possiamo chiamare il metodo non-const da quello const. Ma possiamo chiamare il metodo const da quello non const. Ciò richiederà di usare 'const_cast' per rimuovere il qualificatore const.

La soluzione è:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
        return foo;
    }
};
```

Nel codice qui sopra, chiamiamo la versione const di `GetBar` dalla `GetBar` non-const `GetBar` in tipo const: `const_cast<const Foo*>(this)` . Siccome chiamiamo il metodo const da non-const, l'oggetto stesso è non-const, e il cast via è const.

Esaminare il seguente esempio più completo:

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Leggi parola chiave const online: https://riptutorial.com/it/cplusplus/topic/2386/parola-chiave-const

Capitolo 82: parola chiave mutevole

Examples

modificatore del membro della classe non statico

`mutable` modificatore `mutable` in questo contesto è usato per indicare che un campo dati di un oggetto const può essere modificato senza influenzare lo stato visibile esternamente dell'oggetto.

Se stai pensando di memorizzare nella cache un risultato di costosi calcoli, dovresti probabilmente usare questa parola chiave.

Se si dispone di un campo di dati di blocco (ad esempio, `std::unique_lock`) che è bloccato e sbloccato all'interno di un metodo const, questa parola chiave è anche ciò che è possibile utilizzare.

Non si dovrebbe usare questa parola chiave per interrompere la costanza logica di un oggetto.

Esempio con la memorizzazione nella cache:

```
class pi_calculator {
 public:
     double get_pi() const {
         if (pi_calculated) {
             return pi;
         } else {
             double new_pi = 0;
             for (int i = 0; i < 1000000000; ++i) {
                 // some calculation to refine new_pi
             }
             // note: if pi and pi_calculated were not mutable, we would get an error from a
compiler
             // because in a const method we can not change a non-mutable field
             pi = new_pi;
             pi_calculated = true;
             return pi;
         }
     }
 private:
     mutable bool pi_calculated = false;
     mutable double pi = 0;
};
```

mutande lambda

Per impostazione predefinita, l' `operator()` implicito `operator()` di un lambda è `const` . Ciò non consente di eseguire operazioni non `const` sul lambda. Per consentire la modifica dei membri, una lambda può essere contrassegnata come `mutable` , il che rende l' `operator()` implicito `operator()` non `const` :

```
int a = 0;

auto bad_counter = [a] {
    return a++;   // error: operator() is const
                  // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;  // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Leggi parola chiave mutevole online: https://riptutorial.com/it/cplusplus/topic/2705/parola-chiave-mutevole

Capitolo 83: parole

introduzione

Le parole chiave hanno un significato definito dallo standard C ++ e non possono essere utilizzate come identificatori. È illegale ridefinire le parole chiave utilizzando il preprocessore in qualsiasi unità di traduzione che include un'intestazione di libreria standard. Tuttavia, le parole chiave perdono il loro significato speciale all'interno degli attributi.

Sintassi

- asm (*string letterale*);
- noexcept (*espressione*) // significato 1
- noxcept (*espressione costante*) // significato 2
- noexcept // significato 2
- dimensione *dell'espressione unaria*
- sizeof (*type-id*)
- sizeof ... (*identificatore*) // dal C ++ 11
- typename *nested-name- identificatore identificatore* // significato 1
- typename template *nested-name-specifier* (*opt*) *simple-template-id* // che significa 1
- *identificatore* typename (*opt*) // significato 2
- typename ... *identificatore* (*opt*) // significato 2; dal C ++ 11
- typename *identifier* (*opt*) = *type-id* // che significa 2
- template < *template-parameter-list* > typename ... (*opt*) *identifier* (*opt*) // significato 3
- template < *template-parameter-list* > typename *identifier* (*opt*) = *id-expression* // significato 3

Osservazioni

L'elenco completo delle parole chiave è il seguente:

- `alignas` (dal C ++ 11)
- `alignof` (dal C ++ 11)
- `asm`
- `auto` : dal C ++ 11 , prima del C ++ 11
- `bool`
- `break`
- `case`
- `catch`
- `char`
- `char16_t` (dal C ++ 11)
- `char32_t` (dal C ++ 11)
- `class`
- `const`
- `constexpr` (dal C ++ 11)
- `const_cast`

- `continue`
- `decltype` (dal C ++ 11)
- `default`
- `delete` per gestione memoria , per funzioni (dal C ++ 11)
- `do`
- `double`
- `dynamic_cast`
- `else`
- `enum`
- `explicit`
- `export`
- `extern` come `extern` di dichiarazione , nella specifica del collegamento , per i modelli
- `false`
- `float`
- `for`
- `friend`
- `goto`
- `if`
- `inline` per funzioni , per namespace (dal C ++ 11), per variabili (dal C ++ 17)
- `int`
- `long`
- `mutable`
- `namespace`
- `new`
- `noexcept` (dal C ++ 11)
- `nullptr` (dal C ++ 11)
- `operator`
- `private`
- `protected`
- `public`
- `register`
- `reinterpret_cast`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `static_assert` (dal C ++ 11)
- `static_cast`
- `struct`
- `switch`
- `template`
- `this`
- `thread_local` (dal C ++ 11)
- `throw`
- `true`
- `try`
- `typedef`
- `typeid`
- `typename`
- `union`
- `unsigned`
- `using` per redeclare un nome , per alias uno spazio dei nomi , per alias un tipo
- `virtual` per funzioni , per classi base

- `void`
- `volatile`
- `wchar_t`
- `while`

I token `final` e `override` non sono parole chiave. Possono essere usati come identificatori e hanno un significato speciale solo in determinati contesti.

I token `and` , `and_eq` , `bitand` , `bitor` , `compl` , `not` , `not_eq` , `or` , `or_eq` , `xor` e `xor_eq` sono ortografie alternative di `&&` , `&=` , `&` , `|` , `~` `!` , `!=` , `||` , `|=` , `^` , e `^=` , rispettivamente. Lo standard non li tratta come parole chiave, ma sono parole chiave a tutti gli effetti, dal momento che è impossibile ridefinirli o usarli per indicare qualcosa di diverso dagli operatori che rappresentano.

I seguenti argomenti contengono spiegazioni dettagliate di molte delle parole chiave in C ++, che servono a scopi fondamentali come la denominazione di tipi di base o il controllo del flusso di esecuzione.

- Parole chiave di base
- Controllo del flusso
- Iterazione
- Parole chiave letterali
- Digitare parole chiave
- Parole chiave di dichiarazione variabile
- Classi / Strutture
- Specifiers di classe di archiviazione

Examples

asm

La parola chiave `asm` accetta un singolo operando, che deve essere una stringa letterale. Ha un significato definito dall'implementazione, ma in genere viene passato all'assemblatore dell'implementazione, con l'output dell'assembler incorporato nell'unità di traduzione.

L'istruzione `asm` è una *definizione* , non *un'espressione* , quindi può apparire sia nell'ambito del blocco che dello spazio dei nomi (incluso l'ambito globale). Tuttavia, poiché l'assembly inline non può essere vincolato dalle regole del linguaggio C ++, `asm` potrebbe non apparire all'interno di una funzione `constexpr` .

Esempio:

```
[[noreturn]] void halt_system() {
    asm("hlt");
```

```
}
```

esplicito

1. Quando viene applicato a un costruttore di argomento singolo, impedisce che venga utilizzato il costruttore per eseguire conversioni implicite.

```
class MyVector {
  public:
    explicit MyVector(uint64_t size);
};
MyVector v1(100);  // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 is uint64_t
int len2 = 100;
MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
```

   Dal momento che C ++ 11 ha introdotto gli elenchi di inizializzatori, in C ++ 11 e versioni successive, l' `explicit` può essere applicato a un costruttore con un numero qualsiasi di argomenti, con lo stesso significato del caso a argomento singolo.

```
struct S {
    explicit S(int x, int y);
};
S f() {
    return {12, 34};  // ill-formed
    return S{12, 34}; // ok
}
```

C ++ 11

2. Se applicato a una funzione di conversione, impedisce che la funzione di conversione venga utilizzata per eseguire conversioni implicite.

```
class C {
    const int x;
  public:
    C(int x) : x(x) {}
    explicit operator int() { return x; }
};
C c(42);
int x = c;                    // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion
```

noexcept

C ++ 11

1. Un operatore unario che determina se la valutazione del proprio operando può propagare un'eccezione. Si noti che i corpi delle funzioni chiamate non vengono esaminati, quindi `noexcept` può produrre falsi negativi. L'operando non viene valutato.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n';   // prints 1
}
```

In questo esempio, anche se `bar()` non può mai generare un'eccezione, `noexcept(bar())` è ancora falso perché il fatto che `bar()` non possa propagare un'eccezione non è stato esplicitamente specificato.

2. Quando si dichiara una funzione, specifica se la funzione può propagare o meno un'eccezione. Da solo, dichiara che la funzione non può propagare un'eccezione. Con un argomento tra parentesi, dichiara che la funzione può o non può propagare un'eccezione a seconda del valore di verità dell'argomento.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

In questo esempio, abbiamo dichiarato che `f4` , `f5` e `f6` non possono propagare eccezioni. (Anche se un'eccezione può essere generata durante l'esecuzione di `f6` , viene catturata e non è consentita la propagazione fuori dalla funzione.) Abbiamo dichiarato che `f2` può propagare un'eccezione. Quando l' `noexcept` è omesso, è equivalente a `noexcept(false)` , quindi abbiamo implicitamente dichiarato che `f1` e `f3` possono propagare eccezioni, anche se non possono essere generate eccezioni durante l'esecuzione di `f3` .

C ++ 17

Indipendentemente dal fatto che una funzione non `noexcept` fa parte del tipo di funzione: cioè nell'esempio precedente, `f1` , `f2` e `f3` hanno tipi diversi da `f4` , `f5` e `f6` . Pertanto, `noexcept` è anche significativo nei puntatori di funzione, negli argomenti del modello e così via.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;          // ok; types match
void (*p4)() = &g2;          // ok; implicit conversion
```

typename

1. Quando seguito da un nome qualificato, `typename` specifica che si tratta del nome di un tipo. Questo è spesso richiesto nei template, in particolare, quando lo specificatore del nome annidato è un tipo dipendente diverso dall'istanza corrente. In questo esempio, `std::decay<T>` dipende dal parametro di modello `T` , quindi per denominare il tipo di `type` nidificato, è necessario prefissare l'intero nome qualificato con `typename` . Per ulteriori informazioni, vedere Dove e perché devo inserire le parole chiave "template" e "typename"?

```
template <class T>
auto decay_copy(T&& r) -> typename std::decay<T>::type;
```

2. Introduce un parametro di tipo nella dichiarazione di un modello . In questo contesto, è intercambiabile con la `class` .

```
template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

C ++ 17

3. `typename` può anche essere usato quando si dichiara un parametro template template , che precede il nome del parametro, proprio come la `class` .

```
template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

taglia di

Un operatore unario che produce la dimensione in byte del suo operando, che può essere un'espressione o un tipo. Se l'operando è un'espressione, non viene valutato. La dimensione è un'espressione costante di tipo `std::size_t` .

Se l'operando è un tipo, deve essere tra parentesi.

- È illegale applicare `sizeof` a un tipo di funzione.
- È illegale applicare `sizeof` a un tipo incompleto, incluso `void` .
- Se sizeof è applicato a un tipo di riferimento `T&` oa `T&&` , è equivalente a `sizeof(T)` .
- Quando `sizeof` viene applicato a un tipo di classe, restituisce il numero di byte in un oggetto completo di quel tipo, inclusi eventuali byte di riempimento nel mezzo o alla fine. Pertanto, un'espressione `sizeof` non può mai avere un valore pari a 0. Vedere il layout dei tipi di oggetto per ulteriori dettagli.
- I `char` , `signed char` e `unsigned char` hanno una dimensione pari a 1. Al contrario, un byte è definito come la quantità di memoria richiesta per memorizzare un oggetto `char` . Non significa necessariamente 8 bit, poiché alcuni sistemi hanno oggetti `char` più lunghi di 8 bit.

Se *expr* è un'espressione, `sizeof( expr )` è equivalente a `sizeof(T)` dove `T` è il tipo di *expr.*

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

C ++ 11

L'operatore `sizeof...` produce il numero di elementi in un pacchetto di parametri.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Parole chiave diverse

vuoto C ++

1. Se utilizzato come tipo restituito dalla funzione, la parola chiave void specifica che la funzione non restituisce un valore. Se utilizzato per la lista dei parametri di una funzione, void specifica che la funzione non ha parametri. Se utilizzato nella dichiarazione di un puntatore, void specifica che il puntatore è "universale".
2. Se il tipo di puntatore è void *, il puntatore può puntare a qualsiasi variabile che non è dichiarata con la parola chiave const o volatile. Un puntatore void non può essere dereferenziato a meno che non venga lanciato su un altro tipo. Un puntatore vuoto può essere convertito in qualsiasi altro tipo di puntatore dati.
3. Un puntatore vuoto può puntare a una funzione, ma non a un membro della classe in C ++.

```
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
pv = &i;
   // Cast optional in C required in C++
pint = (int *)pv;
```

C ++ volatile

1. Un qualificatore di tipo che è possibile utilizzare per dichiarare che un oggetto può essere modificato nel programma dall'hardware.

```
volatile declarator ;
```

C ++ virtuale

1. La parola chiave virtuale dichiara una funzione virtuale o una classe base virtuale.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

parametri

1. **specificatori di testo** Specifica il tipo di ritorno della funzione membro virtuale.
2. **membro-funzione-dichiaratore** Dichiara una funzione membro.
3. **access-specificatore** Definisce il livello di accesso alla classe base, pubblica, protetta o privata. Può apparire prima o dopo la parola chiave virtuale.
4. **nome-classe-base** Identifica un tipo di classe precedentemente dichiarato

   questo puntatore

1. Questo puntatore è un puntatore accessibile solo all'interno delle funzioni membro non statiche di una classe, di una struct o di un tipo di unione. Punta all'oggetto per cui viene chiamata la funzione membro. Le funzioni membro statiche non hanno questo puntatore.

```
this->member-identifier
```

Un puntatore di questo oggetto non fa parte dell'oggetto stesso; non si riflette nel risultato di una dichiarazione sizeof sull'oggetto. Invece, quando una funzione membro non statico viene chiamata per un oggetto, l'indirizzo dell'oggetto viene passato dal compilatore come argomento nascosto alla funzione. Ad esempio, la seguente chiamata di funzione:

```
myDate.setMonth( 3 );

can be interpreted this way:


setMonth( &myDate, 3 );

The object's address is available from within the member function as the this pointer. Most
uses of this are implicit. It is legal, though unnecessary, to explicitly use this when
referring to members of the class. For example:


void Date::setMonth( int mn )
{
   month = mn;             // These three statements
   this->month = mn;       // are equivalent
   (*this).month = mn;
}

The expression *this is commonly used to return the current object from a member function:


return *this;

The this pointer is also used to guard against self-reference:
```

```
if (&Object != this) {
// do not execute in cases of self-reference
```

provare, lanciare e prendere istruzioni (C ++)

1. Per implementare la gestione delle eccezioni in C ++, usate try, throw e catch espressioni.
2. Innanzitutto, utilizzare un blocco try per racchiudere una o più istruzioni che potrebbero generare un'eccezione.
3. Un'espressione di lancio indica che una condizione eccezionale, spesso un errore, si è verificata in un blocco try. È possibile utilizzare un oggetto di qualsiasi tipo come l'operando di un'espressione di lancio. In genere, questo oggetto viene utilizzato per comunicare informazioni sull'errore. Nella maggior parte dei casi, si consiglia di utilizzare la classe std :: exception o una delle classi derivate definite nella libreria standard. Se uno di questi non è appropriato, si consiglia di derivare la propria classe di eccezione da std :: exception.
4. Per gestire le eccezioni che possono essere lanciate, implementare uno o più blocchi catch immediatamente dopo un blocco try. Ogni blocco di cattura specifica il tipo di eccezione che può gestire.

```
    MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
      throw myDataFormatException("Format error");
   // ...
}
```

Il codice dopo la clausola try è la sezione di codice protetta. L'espressione di tiro lancia - cioè, solleva - un'eccezione. Il blocco di codice dopo la clausola catch è il gestore di eccezioni. Questo è il gestore che cattura l'eccezione generata se i tipi nelle espressioni throw e catch sono compatibili.

```
    try {
```

```
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions - dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

amico (C ++)

1. In alcune circostanze, è più conveniente concedere l'accesso a livello di membro a funzioni che non sono membri di una classe o di tutti i membri in una classe separata. Solo l'implementatore della classe può dichiarare chi sono i suoi amici. Una funzione o una classe non possono dichiararsi amici di qualsiasi classe. In una definizione di classe, utilizzare la parola chiave friend e il nome di una funzione non membro o un'altra classe per concederne l'accesso ai membri privati e protetti della classe. In una definizione di modello, un parametro di tipo può essere dichiarato come amico.

2. Se dichiari una funzione amico che non è stata dichiarata in precedenza, tale funzione viene esportata nell'ambito di protezione non classificatore.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
   friend int ForwardDeclared::IsAFriend();// C2039 error expected
};
```

funzioni amico

1. Una funzione amico è una funzione che non è un membro di una classe ma ha accesso ai membri privati e protetti della classe. Le funzioni amico non sono considerate membri della classe; sono normali funzioni esterne a cui sono assegnati privilegi speciali di accesso.

2. Gli amici non fanno parte dell'ambito della classe e non vengono chiamati utilizzando gli operatori di selezione membri (. E ->) a meno che non siano membri di un'altra classe.

3. Una funzione amico è dichiarata dalla classe che sta concedendo l'accesso. La dichiarazione amico può essere posizionata ovunque nella dichiarazione della classe. Non è influenzato dalle parole chiave di controllo degli accessi.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }
```

```
private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
    // Output: 0
           1
}
```

I membri della classe come amici

```
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Leggi parole online: https://riptutorial.com/it/cplusplus/topic/4891/parole

Capitolo 84: Parole chiave di base

Examples

int

Denota un tipo intero con segno con "la dimensione naturale suggerita dall'architettura dell'ambiente di esecuzione", il cui intervallo include almeno da -32767 a +32767, incluso.

```
int x = 2;
int y = 3;
int z = x + y;
```

Può essere combinato con `unsigned` , `short` , `long` e `long long` (qv) per fornire altri tipi interi.

bool

Un tipo intero il cui valore può essere `true` o `false` .

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

carbonizzare

Un tipo intero che è "abbastanza grande da memorizzare qualsiasi membro del set di caratteri di base dell'implementazione". È definito dall'implementazione se `char` è firmato (e ha un intervallo di almeno da -127 a +127, incluso) o senza segno (e ha un intervallo di almeno da 0 a 255, inclusi).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

char16_t

C ++ 11

Un tipo intero senza segno con le stesse dimensioni e allineamento di `uint_least16_t` , che è quindi abbastanza grande da contenere un'unità di codice UTF-16.

```
const char16_t message[] = u"□□□□□\n";            // Chinese for "hello, world\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\n"; // prints 7
```

char32_t

C ++ 11

Un tipo intero senza segno con le stesse dimensioni e allineamento di `uint_least32_t` , che è quindi abbastanza grande da contenere un'unità di codice UTF-32.

```
const char32_t full_house[] = U"□□□□□";               // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\n"; // prints 6
```

galleggiante

Un tipo a virgola mobile. Ha l'intervallo più stretto tra i tre tipi di virgola mobile in C ++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

Doppio

Un tipo a virgola mobile. La sua gamma include quella del `float` . Se combinato con `long` , denota il tipo `long double` floating point, la cui gamma include quella del `double` .

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

lungo

Denota un tipo intero con segno che è lungo almeno quanto `int` , e il cui intervallo include almeno da -2147483647 a +2147483647, compreso (vale a dire - (2 ^ 31 - 1) su + (2 ^ 31 - 1)). Questo tipo può anche essere scritto come `long int` .

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

La combinazione `long double` indica un tipo a virgola mobile, che ha la gamma più ampia tra i tre tipi a virgola mobile.

```
long double area(long double radius) {
    const long double pi = 3.14159265358979323851;
    return pi*radius*radius;
}
```

C ++ 11

Quando il `long` specific si verifica due volte, come in `long long` , denota un tipo intero con segno che è lungo almeno quanto a `long` , e il cui intervallo include almeno -9223372036854775807 a +9223372036854775807, incluso (cioè, - (2 ^ 63 - 1) a + (2 ^ 63 - 1)).

```
// support files up to 2 TiB
```

```
const long long max_file_size = 2LL << 40;
```

corto

Denota un tipo di intero con `char` che è almeno lungo come `char` e il cui intervallo include almeno da -32767 a +32767, incluso. Questo tipo può anche essere scritto come `short int` .

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

vuoto

Un tipo incompleto; non è possibile che un oggetto abbia il `void` tipo, né ci sono matrici di `void` o riferimenti al `void` . Viene utilizzato come tipo di restituzione di funzioni che non restituiscono nulla.

Inoltre, una funzione può essere dichiarata ridondante con un singolo parametro di tipo `void` ; questo è equivalente a dichiarare una funzione senza parametri (es. `int main()` e `int main(void)` dichiarano la stessa funzione). Questa sintassi è consentita per compatibilità con C (dove le dichiarazioni di funzione hanno un significato diverso rispetto a C ++).

Il tipo `void*` ("pointer to `void` ") ha la proprietà che ogni puntatore a oggetti può essere convertito ad esso e viceversa e generare lo stesso puntatore. Questa caratteristica rende il tipo `void*` adatto a certi tipi di interfacce di cancellazione del tipo (non sicure dal punto di vista del tipo), ad esempio per contesti generici in API in stile C (ad esempio `qsort` , `pthread_create`).

Qualsiasi espressione può essere convertita in un'espressione di tipo `void` ; questa è chiamata *espressione di valore scartato* :

```
static_cast<void>(std::printf("Hello, %s!\n", name));  // discard return value
```

Questo può essere utile per segnalare esplicitamente che il valore di un'espressione non è di interesse e che l'espressione deve essere valutata solo per i suoi effetti collaterali.

wchar_t

Un numero intero abbastanza grande da rappresentare tutti i caratteri del set di caratteri estesi supportato più grande, noto anche come set di caratteri ampi. (Non è portabile presupporre che `wchar_t` usi una particolare codifica, come UTF-16.)

Viene normalmente utilizzato quando è necessario memorizzare caratteri su ASCII 255, in quanto ha una dimensione maggiore rispetto al tipo di carattere `char` .

```
const wchar_t message_ahmaric[] = L"ሰላም ልዑል\n"; //Ahmaric for "hello, world\n"
const wchar_t message_chinese[] = L"□□□□□\n";// Chinese for "hello, world\n"
const wchar_t message_hebrew[]  = L"שלום עולם\n"; //Hebrew for "hello, world\n"
const wchar_t message_russian[] = L"Привет мир\n";  //Russian for "hello, world\n"
const wchar_t message_tamil[]   = L"ஹலோ உலகம்\n"; //Tamil for "hello, world\n"
```

Leggi Parole chiave di base online: https://riptutorial.com/it/cplusplus/topic/7839/parole-chiave-di-base

Capitolo 85: Parole chiave di dichiarazione variabile

Examples

const

Un identificatore di tipo; quando viene applicato a un tipo, produce la versione const-qualified del tipo. Vedere la parola chiave const per i dettagli sul significato di `const` .

```
const int x = 123;
x = 456;    // error
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

decltype

C ++ 11

Rende il tipo del suo operando, che non viene valutato.

- Se l'operando `e` è un nome senza parentesi aggiuntive, `decltype(e)` è il *tipo dichiarato* di `e` .

  ```
  int x = 42;
  std::vector<decltype(x)> v(100, x); // v is a vector<int>
  ```

- Se l'operando `e` è un membro della classe di accesso senza parentesi aggiuntivi, quindi `decltype(e)` è il *tipo dichiarato* degli Stati accesso.

  ```
  struct S {
      int x = 42;
  };
  const S s;
  decltype(s.x) y; // y has type int, even though s.x is const
  ```

- In tutti gli altri casi, `decltype(e)` restituisce sia il tipo che la categoria di valore dell'espressione `e` , come segue:

  - Se `e` è un lvalue di tipo `T` , quindi `decltype(e)` è `T&` .
  - Se `e` è un xvalue di tipo `T` , `decltype(e)` è `T&&` .
  - Se `e` è un valore di tipo `T` , allora `decltype(e)` è `T`

Questo include il caso con parentesi estranee.

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
decltype((x)) c = x;   // c has type int&, since x is an lvalue
```

C ++ 14

La forma speciale `decltype(auto)` deduce il tipo di variabile dal suo inizializzatore o il tipo di ritorno di una funzione dalle dichiarazioni di `return` nella sua definizione, usando le regole di deduzione di tipo di `decltype` piuttosto che quelle di `auto` .

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

firmato

Una parola chiave che fa parte di determinati nomi di tipi interi.

- Se usato da solo, `int` è implicito, così che `signed` , `signed int` e `int` sono dello stesso tipo.
- Quando combinato con `char` , restituisce il tipo `signed char` , che è un tipo diverso da `char` , anche se `char` è anche firmato. `signed char` ha un intervallo che include almeno da -127 a +127 inclusi.
- Se combinato con `short` , `long` o `long long` , è ridondante, poiché questi tipi sono già firmati.
- `signed` non può essere combinato con `bool` , `wchar_t` , `char16_t` , o `char32_t` .

Esempio:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

unsigned

Un identificatore di tipo che richiede la versione senza segno di un tipo intero.

- Se usato da solo, `int` è implicito, quindi `unsigned` è lo stesso tipo di `unsigned int` .
- Il `unsigned char` è diverso dal tipo `char` , anche se `char` non è firmato. Può contenere numeri interi fino ad almeno 255.
- `unsigned` può anche essere combinato con `short` , `long` o `long long` . Non può essere combinato con `bool` , `wchar_t` , `char16_t` o `char32_t` .

Esempio:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

volatile

Un qualificatore di tipo; quando applicato a un tipo, produce la versione qualificata volatile del tipo. La qualifica volatile ha lo stesso ruolo della qualifica `const` nel sistema di tipi, ma la `volatile` non impedisce la modifica degli oggetti; al contrario, obbliga il compilatore a trattare tutti gli accessi a tali oggetti come effetti collaterali.

Nell'esempio seguente, se `memory_mapped_port` non fosse volatile, il compilatore potrebbe ottimizzare la funzione in modo che esegua solo la scrittura finale, che sarebbe errata se `sizeof(int)` è maggiore di 1. La qualifica `volatile` lo costringe a trattare tutte le `sizeof(int)` scrive come diversi effetti collaterali e quindi li esegue tutti (nell'ordine).

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Leggi Parole chiave di dichiarazione variabile online:
https://riptutorial.com/it/cplusplus/topic/7840/parole-chiave-di-dichiarazione-variabile

Capitolo 86: Piega le espressioni

Osservazioni

Le espressioni di piegatura sono supportate per i seguenti operatori

+	-	*	/	%	\	&	|	<<	>>		
+ =	- =	* =	/ =	% =	\ =	& =	| =	<< =	>> =	=	
==	! =	<	>	<=	> =	&&	||	,	. *	-> *	

Quando si piega su una sequenza vuota, un'espressione di piegatura è mal formata, ad eccezione dei seguenti tre operatori:

Operatore	Valore quando il pacchetto di parametri è vuoto
&&	vero
||	falso
,	void ()

Examples

Foldario unario

Le piegature unarie vengono utilizzate per *piegare* i pacchetti di parametri su un operatore specifico. Ci sono 2 tipi di pieghe unari:

- Unary **Left** Fold `(... op pack)` che si espande come segue:

```
((Pack1 op Pack2) op ...) op PackN
```

- Unario **Right** Fold `(pack op ...)` che si espande come segue:

```
Pack1 op (... (Pack(N-1) op PackN))
```

Ecco un esempio

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold
```

```
    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); //  6
```

Pieghe binarie

Le pieghe binarie sono sostanzialmente pieghe unarie , con un argomento in più.

Esistono 2 tipi di pieghe binarie:

- Binary **Left** Fold - `(value op ... op pack)` - Si espande come segue:

  ```
  (((Value op Pack1) op Pack2) op ...) op PackN
  ```

- Binary **Right** Fold `(pack op ... op value)` - Si espande come segue:

  ```
  Pack1 op (... op (Pack(N-1) op (PackN op Value)))
  ```

Ecco un esempio:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}

int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Piegando una virgola

È un'operazione comune dover eseguire una particolare funzione su ciascun elemento in un pacchetto di parametri. Con C ++ 11, il meglio che possiamo fare è:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Ma con un'espressione fold, quanto sopra semplifica bene a:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
```

```
}
```

Non è richiesta alcuna caldaia criptica.

Leggi Piega le espressioni online: https://riptutorial.com/it/cplusplus/topic/2676/piega-le-espressioni

Capitolo 87: Polimorfismo

Examples

Definire classi polimorfiche

L'esempio tipico è una classe di forma astratta, che può quindi essere derivata in quadrati, cerchi e altre forme concrete.

La classe genitore:

Iniziamo con la classe polimorfica:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

Come leggere questa definizione?

- È possibile definire il comportamento polimorfico mediante funzioni membro introdotte con la parola chiave `virtual` . Qui `get_surface()` e `describe_object()` saranno ovviamente implementati in modo diverso per un quadrato che per un cerchio. Quando la funzione viene invocata su un oggetto, la funzione corrispondente alla classe reale dell'oggetto sarà determinata in fase di esecuzione.
- Non ha senso definire `get_surface()` per una forma astratta. Questo è il motivo per cui la funzione è seguita da `= 0` . Ciò significa che la funzione è *pura funzione virtuale* .
- Una classe polimorfica dovrebbe sempre definire un distruttore virtuale.
- È possibile definire funzioni membro non virtuali. Quando queste funzioni saranno invocate per un oggetto, la funzione verrà scelta in base alla classe utilizzata in fase di compilazione. Qui `get_double_surface()` è definito in questo modo.
- Una classe che contiene almeno una funzione virtuale pura è una classe astratta. Le classi astratte non possono essere istanziate. Potresti avere solo puntatori o riferimenti di un tipo di classe astratta.

Classi derivate

Una volta definita una classe di base polimorfica, è possibile derivarla. Per esempio:

```
class Square : public Shape {
    Point top_left;
```

```
    double side_length;
public:
    Square (const Point& top_left, double side)
       : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
                  << " with a length of " << side_length << std::endl;
    }
};
```

Alcune spiegazioni:

- È possibile definire o sovrascrivere qualsiasi funzione virtuale della classe genitore. Il fatto che una funzione fosse virtuale nella classe genitore lo rende virtuale nella classe derivata. Non c'è bisogno di dire al compilatore di nuovo la parola chiave `virtual` . Tuttavia, si consiglia di aggiungere la `override` della parola chiave alla fine della dichiarazione della funzione, al fine di prevenire piccoli bug causati da variazioni non notate nella firma della funzione.
- Se tutte le pure funzioni virtuali della classe genitrice sono definite, puoi istanziare oggetti per questa classe, altrimenti diventerà anche una classe astratta.
- Non sei obbligato a scavalcare tutte le funzioni virtuali. Puoi mantenere la versione del genitore se è adatta alle tue necessità.

Esempio di istanziazione

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr;  // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Downcast sicuro

Supponiamo di avere un puntatore a un oggetto di una classe polimorfica:

```
Shape *ps;                        // see example on defining a polymorphic class
ps =  get_a_new_random_shape();   // if you don't have such a function yet, you
                                  // could just write ps = new Square(0.0,0.0, 5);
```

un downcast sarebbe quello di lanciare da una `Shape` polimorfica generale a una delle sue forme derivate e più specifiche come `Square` o `Circle` .

Perché downcast?

Nella maggior parte dei casi, non è necessario sapere quale sia il tipo reale dell'oggetto, poiché le funzioni virtuali consentono di manipolare il tuo oggetto indipendentemente dal suo tipo:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

Se non hai bisogno di alcun downcast, il tuo design sarebbe perfetto.

Tuttavia, a volte potrebbe essere necessario downcast. Un tipico esempio è quando si desidera richiamare una funzione non virtuale che esiste solo per la classe figlio.

Prendi in considerazione per esempio i cerchi. Solo i cerchi hanno un diametro. Quindi la classe verrebbe definita come:

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
    Circle (const Point& center, double radius)
       : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

La funzione membro `get_diameter()` esiste solo per le cerchie. Non è stato definito per un oggetto `Shape` :

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

Come abbattere?

Se sapessi per certo che `ps` indica una cerchia puoi optare per un `static_cast` :

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

Questo farà il trucco. Ma è molto rischioso: se `ps` sembra essere diverso da un `Circle` il comportamento del tuo codice sarà indefinito.

Quindi, piuttosto che giocare alla roulette russa, dovresti usare tranquillamente un `dynamic_cast` . Questo è specifico per le classi polimorfiche:

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();   // OUCH !!! Compilation error
```

```
    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Nota che `dynamic_cast` non è possibile su una classe che non è polimorfica. Avresti bisogno di almeno una funzione virtuale nella classe o dei suoi genitori per poterla usare.

Polimorfismo e distruttori

Se si intende utilizzare una classe in modo polimorfico, con le istanze derivate memorizzate come puntatori / riferimenti di base, il distruttore della sua classe base deve essere `virtual` o `protected` . Nel primo caso, questo causerà la distruzione dell'oggetto per controllare il `vtable` , chiamando automaticamente il distruttore corretto in base al tipo dinamico. In quest'ultimo caso, la distruzione dell'oggetto tramite un puntatore / riferimento di classe base è disabilitata e l'oggetto può essere eliminato solo se trattato esplicitamente come tipo effettivo.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

struct ProtectedDestructor {
  protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Entrambe queste pratiche garantiscono che il distruttore della classe derivata verrà sempre chiamato su istanze di classi derivate, impedendo perdite di memoria.

Leggi Polimorfismo online: https://riptutorial.com/it/cplusplus/topic/1717/polimorfismo

Capitolo 88: precedenza dell'operatore

Osservazioni

Gli operatori sono elencati dall'alto in basso, in ordine decrescente. Gli operatori con lo stesso numero hanno la stessa priorità e la stessa associatività.

1. `::`
2. Gli operatori postfix: `[] () T(...) . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`
3. Gli operatori del prefisso unario: `++ -- * & + - ! ~ sizeof new delete delete[]` ; la notazione cast in stile C, `(T)...` ; (C ++ 11 e successivi) `sizeof... alignof noexcept`
4. `.*` e `->*`
5. `*` , `/` , e `%` , operatori aritmetici binari
6. `+` e `-` , operatori aritmetici binari
7. `<<` e `>>`
8. `<` , `>` , `<=` , `>=`
9. `==` e `!=`
10. `&` , l'operatore AND bit a bit
11. `^`
12. `|`
13. `&&`
14. `||`
15. `?:` (operatore condizionale ternario)
16. `=` , `*=` , `/=` , `%=` , `+=` , `-=` , `>>=` , `<<=` , `&=` , `^=` , `|=`
17. `throw`
18. `,` (operatore virgola)

L'assegnazione, l'assegnazione composta e gli operatori condizionali ternari sono giusti-associativi. Tutti gli altri operatori binari sono associati a sinistra.

Le regole per l'operatore condizionale ternario sono un po 'più complicate di quanto possano esprimere le regole di precedenza semplici.

- Un operando si lega meno strettamente ad un `?` alla sua sinistra o a `:` a destra rispetto a qualsiasi altro operatore. In effetti, il secondo operando dell'operatore condizionale viene analizzato come se fosse tra parentesi. Questo consente un'espressione come `a ? b , c : d` per essere sintatticamente valido.
- Un operando si lega più strettamente ad un `?` alla sua destra che a un operatore di assegnazione o di `throw` alla sua sinistra, quindi `a = b ? c : d` è equivalente a `a = (b ? c : d)` e `throw a ? b : c` è equivalente al `throw (a ? b : c)` .
- Un operando si lega più strettamente a un operatore incaricato alla sua destra che a `:` a sinistra, quindi `a ? b : c = d` è equivalente a `a ? b : (c = d)` .

Examples

Operatori aritmetici

Gli operatori aritmetici in C ++ hanno la stessa precedenza in matematica:

Moltiplicazione e divisione hanno lasciato l'associatività (il che significa che saranno valutati da sinistra a destra) e hanno una precedenza più alta di addizioni e sottrazioni, che hanno anche lasciato associatività.

Possiamo anche forzare la precedenza dell'espressione usando parentesi `()`. Proprio come faresti nella matematica normale.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Operatori logici AND e OR

Questi operatori hanno la solita precedenza in C ++: AND before OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

Questo codice è equivalente al seguente:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

L'aggiunta della parentesi non modifica il comportamento, tuttavia facilita la lettura. Aggiungendo queste parentesi, non c'è confusione sull'intento dello scrittore.

Logico && e || operatori: cortocircuito

&& ha la precedenza su ||, questo significa che vengono poste parentesi per valutare cosa sarebbe valutato insieme.

c ++ utilizza la valutazione di cortocircuito in && e || non fare esecuzioni inutili.
Se il lato sinistro di || ritorna vero il lato destro non ha più bisogno di essere valutato.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
}
```

Operatori unari

Gli operatori unari agiscono sull'oggetto su cui sono chiamati e hanno un'alta precedenza. (Vedi

Note)

Quando viene usato postfix, l'azione si verifica solo dopo aver valutato l'intera operazione, portando ad alcune aritmetiche interessanti:

```
int a = 1;
++a;             // result: 2
a--;             // result: 1
int minusa=-a;  // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Leggi precedenza dell'operatore online: https://riptutorial.com/it/cplusplus/topic/3895/precedenza-dell-operatore

Capitolo 89: preprocessore

introduzione

Il preprocessore C è un semplice parser / replacer di testo che viene eseguito prima della compilazione effettiva del codice. Utilizzato per estendere e facilitare l'uso del linguaggio C (e successivo C ++), può essere utilizzato per:

un. **Compresi altri file che** utilizzano `#include`

b. **Definisci una macro di sostituzione del testo** usando `#define`

c. **Compilazione condizionale** usando `#if #ifdef`

d. **Logica specifica per piattaforma / compilatore** (come estensione della compilazione condizionale)

Osservazioni

Le istruzioni di preprocessore vengono eseguite prima che i file di origine vengano consegnati al compilatore. Sono capaci di una logica condizionale di livello molto basso. Poiché i costrutti del preprocessore (ad es. Macro simili a oggetti) non vengono digitati come normali funzioni (la fase di pre-elaborazione avviene prima della compilazione), il compilatore non può imporre controlli di tipo, pertanto dovrebbero essere usati con attenzione.

Examples

Includi le guardie

Un file di intestazione può essere incluso da altri file di intestazione. Un file sorgente (unità di compilazione) che include più intestazioni può quindi, indirettamente, includere alcune intestazioni più di una volta. Se un tale file di intestazione che è incluso più di una volta contiene definizioni, il compilatore (dopo la pre-elaborazione) rileva una violazione della regola di una definizione (ad esempio §3.2 dello standard C ++ 2003) e pertanto genera una diagnostica e la compilazione non riesce.

L'inclusione multipla è impedita usando "include guardie", che a volte sono anche conosciute come guardie di intestazione o macro guardie. Questi sono implementati usando il preprocessore `#define` , `#ifndef` , direttive `#endif` .

per esempio

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED
```

```
class Foo    //  a class definition
{
};

#endif
```

Il vantaggio principale dell'utilizzo di protezioni include è che funzioneranno con tutti i compilatori e i preprocessori conformi agli standard.

Tuttavia, le guardie includono anche alcuni problemi per gli sviluppatori, in quanto è necessario garantire che le macro siano uniche all'interno di tutte le intestazioni utilizzate in un progetto. Nello specifico, se due (o più) intestazioni utilizzano `FOO_H_INCLUDED` come guardia di inclusione, la prima di quelle intestazioni incluse in una unità di compilazione impedirà efficacemente che gli altri vengano inclusi. Particolari sfide vengono introdotte se un progetto utilizza un numero di librerie di terze parti con i file di intestazione che capita di utilizzare guardie in comune.

È inoltre necessario assicurarsi che le macro utilizzate in protezioni non siano in conflitto con altre macro definite nei file di intestazione.

La maggior parte delle implementazioni C ++ supporta anche la direttiva `#pragma once` che garantisce che il file venga incluso solo una volta all'interno di una singola compilation. Questa è una direttiva standard di *fatto* , ma non fa parte di alcuno standard ISO C ++. Per esempio:

```
// Foo.h
#pragma once

class Foo
{
};
```

Mentre `#pragma once` evita alcuni problemi associati alle guardie, un `#pragma` - per definizione negli standard - è intrinsecamente un hook specifico del compilatore, e verrà ignorato silenziosamente dai compilatori che non lo supportano. I progetti che utilizzano `#pragma once` sono più difficili da trasferire ai compilatori che non lo supportano.

Un certo numero di linee guida di codifica e standard di garanzia per C ++ in particolare scoraggiano qualsiasi uso del preprocessore diverso da `#include` file di intestazione o allo scopo di inserire protezioni incluse nelle intestazioni.

Logica condizionale e gestione multipiattaforma

In breve, la logica di pre-elaborazione condizionale riguarda la possibilità di rendere la logica del codice disponibile o non disponibile per la compilazione utilizzando le definizioni di macro.

Tre casi d'uso importanti sono:

- diversi **profili di app** (ad es. debug, release, test, ottimizzati) che possono essere candidati alla stessa app (ad esempio con registrazione aggiuntiva).
- **cross-platform compila** - codice di base singola, piattaforme di compilazione multiple.
- utilizzando una base di codice comune per più **versioni dell'applicazione** (es. **versioni**

Basic, Premium e Pro di un software) - con caratteristiche leggermente diverse.

Esempio a: un approccio multipiattaforma per la rimozione dei file (illustrativo):

```
#ifdef _WIN32
#include <windows.h> // and other windows system files
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
  return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
  return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
  //TODO: check if NSAPI has a more specific function with permission dialog
  return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Macro come `_WIN32` , `__APPLE__` o `__unix__` sono normalmente predefinite dalle corrispondenti implementazioni.

Esempio b: abilitazione della registrazione aggiuntiva per una build di debug:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "--------BEGIN-DUMP---------------\n"
              << AppState::Instance()->Settings().ToString() << "\n"
#if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
              << ListToString(AppState::UndoStack()->GetActionNames())
              << AppState::Instance()->CrntDocument().Name()
              << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
#endif
              << "--------END-DUMP---------------\n"
}
```

Esempio c: abilitare una funzione premium in una build di prodotto separata (nota: questo è illustrativo, spesso è un'idea migliore consentire di sbloccare una funzionalità senza la necessità di reinstallare un'applicazione)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium
users. Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
}
```

Alcuni trucchi comuni:

Definizione dei simboli al momento del richiamo:

Il preprocessore può essere richiamato con simboli predefiniti (con inizializzazione opzionale). Ad esempio questo comando (`gcc -E` esegue solo il preprocessore)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

elabora Sample.cpp nello stesso modo in cui sarebbe se `#define OPTIMISE_FOR_OS_X` e `#define TESTING_MODE 1` venissero aggiunti all'inizio di Sample.cpp.

Garantire una macro è definita:

Se una macro non è definita e il suo valore viene confrontato o verificato, il preprocessore assume quasi sempre in silenzio il valore di `0` . Ci sono alcuni modi per lavorare con questo. Un approccio è quello di assumere che le impostazioni predefinite siano rappresentate come 0 e che qualsiasi modifica (ad esempio al profilo di build dell'app) debba essere esplicitamente eseguita (ad es. ENABLE_EXTRA_DEBUGGING = 0 per impostazione predefinita, set -DENABLE_EXTRA_DEBUGGING = 1 per eseguire l'override). Un altro approccio è rendere esplicite tutte le definizioni e le impostazioni predefinite. Questo può essere ottenuto utilizzando una combinazione di direttive `#ifndef` e `#error` :

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#    error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#    if ( 1 == ENABLE_EXTRA_DEBUGGING )
  //code
#    endif
#endif
```

Macro

Le macro sono suddivise in due gruppi principali: macro simili a oggetti e macro simili a funzioni. Le macro vengono considerate come sostituzione di token all'inizio del processo di compilazione. Ciò significa che sezioni di codice grandi (o ripetute) possono essere astratte in una macro di preprocessore.

```
// This is an object-like macro
#define    PI         3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesnt know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define    AREA(r)    (PI*(r)*(r))

// They can be used like this:
double pi_macro   = PI;
double area_macro = AREA(4.6);
```

La libreria Qt utilizza questa tecnica per creare un sistema meta-oggetto facendo in modo che l'utente dichiari la macro Q_OBJECT all'inizio della classe definita dall'utente che estende QObject.

I nomi macro sono in genere scritti in maiuscolo, per renderli più facili da distinguere dal codice normale. Questo non è un requisito, ma è semplicemente considerato di buon stile da molti programmatori.

Quando viene rilevata una macro simile a un oggetto, viene espansa come una semplice operazione di copia-incolla, con il nome della macro sostituito con la sua definizione. Quando viene rilevata una macro simile a una funzione, vengono espansi sia il nome che i relativi parametri.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
// Compiler sees:
double area = (3.14159265358979*(5)*(5))
```

A causa di ciò, i parametri macro simili a funzioni sono spesso racchiusi tra parentesi, come in `AREA()` sopra. Questo serve a prevenire eventuali bug che possono verificarsi durante l'espansione della macro, in particolare i bug causati da un singolo parametro macro composto da più valori effettivi.

```
#define BAD_AREA(r) PI * r * r

double bad_area = BAD_AREA(5 + 1.6);
// Compiler sees:
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;

double good_area = AREA(5 + 1.6);
// Compiler sees:
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

Si noti inoltre che, a causa di questa semplice espansione, è necessario prestare attenzione con i parametri passati ai macro, per prevenire effetti collaterali imprevisti. Se il parametro viene modificato durante la valutazione, verrà modificato ogni volta che viene utilizzato nella macro espansa, che di solito non è ciò che vogliamo. Questo è vero anche se la macro racchiude i parametri tra parentesi per impedire all'espansione di rompere qualsiasi cosa.

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

Inoltre, le macro non forniscono alcuna sicurezza del tipo, con conseguenti errori di difficile comprensione relativi alla mancata corrispondenza dei tipi.

Poiché i programmatori normalmente terminano le righe con un punto e virgola, i macro che si intendono utilizzare come linee indipendenti sono spesso progettati per "inghiottire" un punto e virgola; questo impedisce che eventuali bug non intenzionali siano causati da un punto e virgola in più.

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

In questo esempio, il doppio punto e virgola accidentale interrompe il blocco `if...else` , impedendo al compilatore di far corrispondere il `else` al `if` . Per evitare ciò, il punto e virgola viene omesso dalla definizione della macro, che lo farà "inghiottire" il punto e virgola immediatamente dopo ogni suo utilizzo.

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
    std::cout << "Hooray!  I work again!" << std::endl;
```

Lasciare il punto e virgola finale consente inoltre di utilizzare la macro senza terminare l'istruzione corrente, il che può essere utile.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)

// ...

some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Normalmente, una definizione di macro termina alla fine della riga. Se una macro deve coprire più righe, tuttavia, una barra rovesciata può essere utilizzata alla fine di una riga per indicare ciò. Questo backslash deve essere l'ultimo carattere della riga, che indica al preprocessore che la riga seguente deve essere concatenata sulla riga corrente, trattandoli come una singola riga. Questo può essere usato più volte di seguito.

```
#define TEXT "I \
am \
many \
lines."

// ...

std::cout << TEXT << std::endl; // Output:   I am many lines.
```

Ciò è particolarmente utile in macro di funzioni complesse, che potrebbero dover coprire più righe.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

Nel caso di macro più simili alle funzioni, può essere utile assegnare loro il proprio ambito per evitare possibili conflitti di nomi o causare la distruzione di oggetti alla fine della macro, in modo simile a una funzione effettiva. Un idioma comune per questo è *do mentre 0* , dove la macro è racchiusa in un blocco *do-while* . Generalmente questo blocco *non viene* seguito con un punto e virgola, consentendogli di ingoiare un punto e virgola.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

Esistono anche macro variadiche; analogamente alle funzioni variadiche, questi prendono un numero variabile di argomenti e quindi li espandono tutti al posto di un parametro speciale "Varargs", `__VA_ARGS__` .

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(printf, "%d", 8);
// Compiler sees:
printf("%d", 8);
```

Nota che durante l'espansione, `__VA_ARGS__` può essere posizionato ovunque nella definizione e verrà espanso correttamente.

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)

VARIADIC2(some_func, 3, 8, 6, 9);
// Compiler sees:
some_func(8, 6, 9, 3);
```

Nel caso di un parametro variadico a zero argomenti, diversi compilatori gestiranno diversamente la virgola finale. Alcuni compilatori, come Visual Studio, ingoiano silenziosamente la virgola senza

alcuna sintassi speciale. Altri compilatori, come GCC, richiedono di posizionare `##` immediatamente prima di `__VA_ARGS__` . A causa di ciò, è saggio definire condizionalmente le macro variadiche quando la portabilità è un problema.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if       COMPILER == "VS"
    #define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif     COMPILER == "GCC"
    #define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Messaggi di errore del preprocessore

Gli errori di compilazione possono essere generati usando il preprocessore. Ciò è utile per una serie di motivi, tra cui alcuni, che includono la notifica a un utente se si trovano su una piattaforma non supportata o su un compilatore non supportato.

es. Errore di restituzione se la versione di gcc è 3.0.0 o precedente.

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

es. Errore di restituzione in caso di compilazione su un computer Apple.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

Macro predefinite

Le macro predefinite sono quelle definite dal compilatore (diversamente da quelle definite dall'utente nel file sorgente). Queste macro non devono essere ridefinite o indefinite dall'utente.

Le seguenti macro sono predefinite dallo standard C ++:

- `__LINE__` contiene il numero di riga della linea su cui è utilizzata questa macro e può essere modificato dalla direttiva `#line` .
- `__FILE__` contiene il nome file del file in cui questa macro è utilizzata e può essere modificata dalla direttiva `#line` .
- `__DATE__` contiene la data (nel formato `"Mmm dd yyyy"`) della compilazione del file, dove *Mmm* è formattato come se fosse ottenuto da una chiamata a `std::asctime()` .
- `__TIME__` contiene il tempo (nel formato `"hh:mm:ss"`) della compilazione del file.
- `__cplusplus` è definito da compilatori C ++ (conformi) durante la compilazione di file C ++. Il suo valore è la versione standard che il compilatore è **pienamente** conforme, ad esempio `199711L` per C ++ 98 e C ++ 03, `201103L` per C ++ 11 e `201402L` per C ++ 14 standard.

c ++ 11

- `__STDC_HOSTED__` è definito su `1` se l'implementazione è *ospitata* o `0` se è *indipendente* .

c ++ 17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` contiene un letterale `size_t` , che è l'allineamento utilizzato per una `operator new` chiamata all'operatore di allineamento non consapevole.

Inoltre, le seguenti macro possono essere predefinite dalle implementazioni e possono o non possono essere presenti:

- `__STDC__` ha un significato dipendente dall'implementazione e viene generalmente definito solo quando si compila un file come C, per indicare la piena conformità allo standard C. (O mai, se il compilatore decide di non supportare questa macro.)

c ++ 11

- `__STDC_VERSION__` ha un significato dipendente dall'implementazione, e il suo valore è solitamente la versione C, analogamente a quanto `__cplusplus` sia la versione C ++. (O non è nemmeno definito, se il compilatore decide di non supportare questa macro.)
- `__STDC_MB_MIGHT_NEQ_WC__` è definito a `1` , se i valori della codifica ristretta del set di caratteri di base potrebbero non essere uguali ai valori delle loro controparti ampie (ad esempio se `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `__STDC_ISO_10646__` è definito se `wchar_t` è codificato come Unicode e si espande in una costante intera nel formato `yyyymmL` , indicando l'ultima revisione Unicode supportata.
- `__STDCPP_STRICT_POINTER_SAFETY__` è definito su `1` , se l'implementazione ha *una sicurezza puntatore rigorosa* (altrimenti ha *una sicurezza di puntatore rilassata*)
- `__STDCPP_THREADS__` è definito su `1` , se il programma può avere più di un thread di esecuzione (applicabile *all'implementazione indipendente* - le *implementazioni ospitate* possono sempre avere più di un thread)

Vale anche la pena ricordare `__func__` , che non è una macro, ma una variabile locale predefinita. Contiene il nome della funzione in cui è utilizzato, come array di caratteri statici in un formato definito dall'implementazione.

Oltre a quelle macro predefinite standard, i compilatori possono avere il proprio set di macro predefinite. Si deve fare riferimento alla documentazione del compilatore per apprenderli. Per esempio:

- gcc
- Microsoft Visual C ++
- fragore
- Compilatore Intel C ++

Alcune delle macro sono solo per richiedere il supporto di alcune funzionalità:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
   // C library header declarations here
}
#endif
```

Altri sono molto utili per il debug:

c ++ 11

```
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
              << " in function " << __func__ << "()"
              << " in file " << __FILE__
              << std::endl;
}
```

E altri per il controllo della versione banale:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
                  << "v 1.1\n" // I have to remember to update this manually
                  << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
                  << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

X-macro

Una tecnica idiomatica per generare strutture di codice ripetitive in fase di compilazione.

Una X-macro consiste di due parti: la lista e l'esecuzione della lista.

Esempio:

```
#define LIST \
    X(dog)   \
    X(cat)   \
    X(racoon)

// class Animal {
//  public:
//    void say();
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}
```

che è espanso dal preprocessore nel seguente:

```
Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}
```

Man mano che le liste diventano più grandi (diciamo, più di 100 elementi), questa tecnica aiuta ad evitare un eccesso di copia-incolla.

Fonte: https://en.wikipedia.org/wiki/X_Macro

Vedi anche: X-macros

Se la definizione di una `X` seamingly irrilevante prima di usare `LIST` non è di tuo gradimento, puoi anche passare un nome di macro come argomento:

```
#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)
```

Ora, si specifica esplicitamente quale macro deve essere utilizzata quando si espande l'elenco, ad es

```
#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)
```

Se ogni invocazione della `MACRO` dovesse assumere parametri aggiuntivi - costante rispetto all'elenco, è possibile utilizzare macro variadic

```
//a walkaround for Visual studio
#define EXPAND(x) x

#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(racoon, __VA_ARGS__))
```

Il primo argomento è fornito dal `LIST` , mentre il resto è fornito dall'utente nel richiamo `LIST` . Per esempio:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

si espanderà a

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

#pragma una volta

La maggior parte, ma non tutte, le implementazioni C ++ supportano la direttiva `#pragma once` che garantisce che il file venga incluso solo una volta all'interno di una singola compilation. Non fa parte di alcuno standard ISO C ++. Per esempio:

```
// Foo.h
#pragma once

class Foo
{
};
```

Mentre `#pragma once` evita alcuni problemi associati alle guardie , un `#pragma` - per definizione negli standard - è intrinsecamente un hook specifico del compilatore, e verrà ignorato silenziosamente dai compilatori che non lo supportano. I progetti che utilizzano `#pragma once` devono essere modificati per essere conformi agli standard.

Con alcuni compilatori, in particolare quelli che utilizzano intestazioni precompilate , `#pragma once` può comportare una notevole accelerazione del processo di compilazione. Allo stesso modo, alcuni preprocessori ottengono l'accelerazione della compilazione rintracciando quali intestazioni hanno impiegato le guardie. Il vantaggio netto, quando sono impiegati entrambi `#pragma once` e include guardie, dipende dall'implementazione e può essere un aumento o una diminuzione dei tempi di compilazione.

`#pragma once` combinato con le protezioni incluse, era il layout consigliato per i file di intestazione quando si scrivevano applicazioni basate su MFC su Windows, ed era generato dalla `add class` Visual Studio, `add dialog` , `add windows` procedure guidate di `add windows` . Quindi è molto comune trovarli combinati in C ++ Windows Candidati.

Operatori preprocessori

`#` operatore o operatore di stringa viene utilizzato per convertire un parametro Macro in una stringa letterale. Può essere utilizzato solo con le macro con argomenti.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")

PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor""\n");
```

Il compilatore concatena due stringhe e l'argomento finale `printf()` sarà una stringa letterale con carattere di fine riga alla fine.

Il preprocessore ignorerà gli spazi prima o dopo l'argomento macro. Quindi sotto la dichiarazione di stampa ci darà lo stesso risultato.

```
PRINT(   This line will be converted to string by preprocessor );
```

Se il parametro della stringa letterale richiede una sequenza di escape come prima di una doppia citazione (), verrà automaticamente inserito dal preprocessore.

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor""\n");
```

`##` operatore o operatore di Token che incolla è usato per concatenare due parametri o token di una Macro.

```
// preprocessor will combine the variable and the x
#define PRINT(x) printf("variable" #x " = %d", variable##x)

int variableY = 15;
PRINT(Y);
//compiler sees
printf("variable""Y"" = %d", variableY);
```

e l'output finale sarà

```
variableY = 15
```

Leggi preprocessore online: https://riptutorial.com/it/cplusplus/topic/1098/preprocessore

Capitolo 90: profiling

Examples

Creazione di profili con gcc e gprof

Il profilo GNU gprof, gprof , ti consente di profilare il tuo codice. Per usarlo, è necessario eseguire le seguenti operazioni:

1. Costruisci l'applicazione con le impostazioni per generare informazioni di profilazione
2. Genera informazioni di profilo eseguendo l'applicazione creata
3. Visualizza le informazioni di creazione generate con gprof

Per costruire l'applicazione con le impostazioni per generare informazioni di profilazione, aggiungiamo il flag `-pg` . Quindi, per esempio, potremmo usare

```
$ gcc -pg *.cpp -o app
```

o

```
$ gcc -O2 -pg *.cpp -o app
```

e così via.

Una volta creata l'applicazione, ad esempio l' `app` , eseguirla come al solito:

```
$ ./app
```

Questo dovrebbe produrre un file chiamato `gmon.out` .

Per vedere i risultati del profilo, ora esegui

```
$ gprof app gmon.out
```

(nota che forniamo sia l'applicazione che l'output generato).

Ovviamente puoi anche pipe o reindirizzare:

```
$ gprof app gmon.out | less
```

e così via.

Il risultato dell'ultimo comando dovrebbe essere una tabella, le cui righe sono le funzioni e le cui colonne indicano il numero di chiamate, il tempo totale speso, il tempo trascorso in autonomia

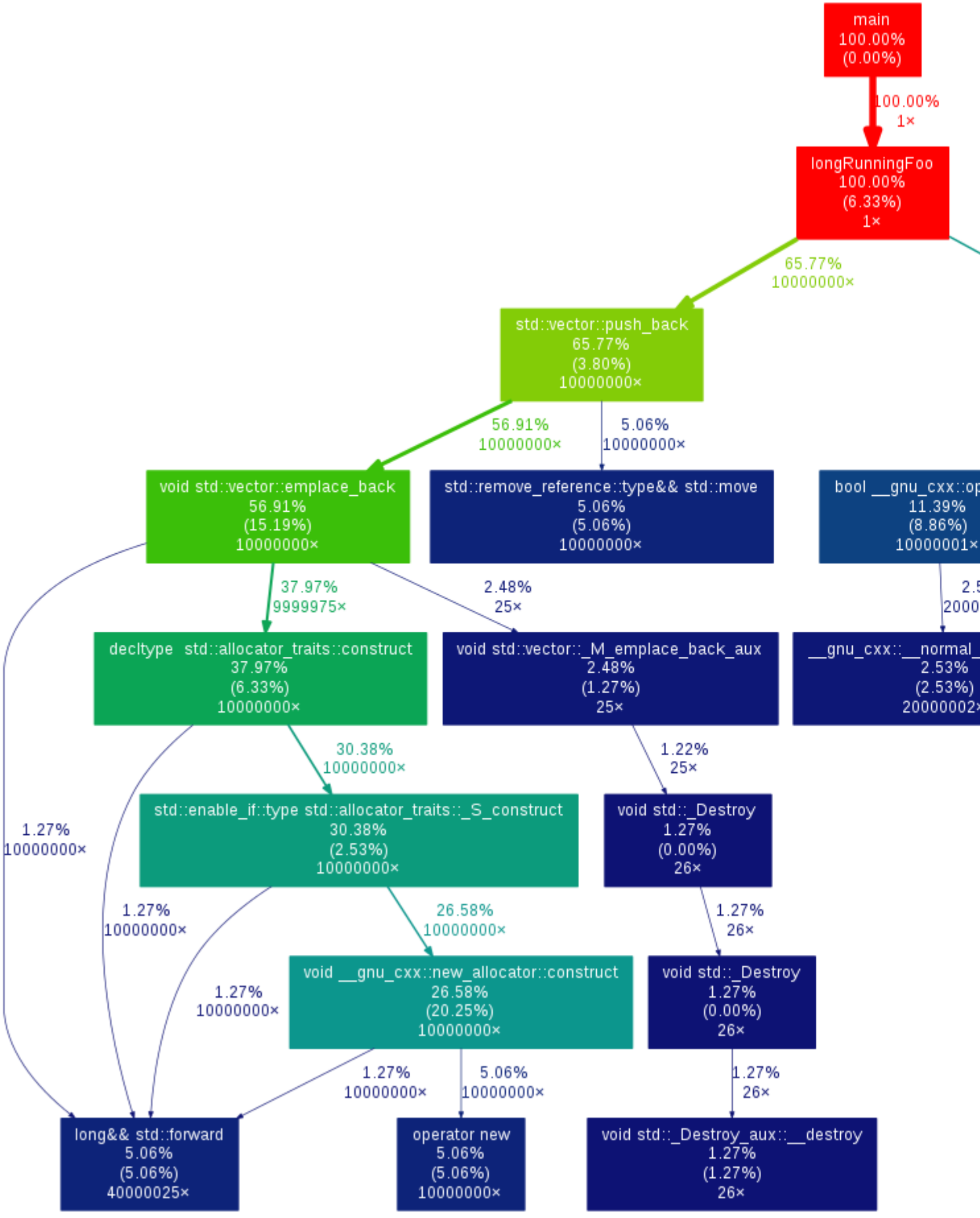
(ovvero il tempo trascorso nella funzione escludendo le chiamate ai bambini).

Generazione di diagrammi callgraph con gperf2dot

Per applicazioni più complesse, i profili di esecuzione flat possono essere difficili da seguire. Questo è il motivo per cui molti strumenti di profilazione generano anche una qualche forma di informazioni callgraph con annotazioni.

gperf2dot converte l'output di testo di molti profiler (Linux perf, callgrind, oprofile, ecc.) in un diagramma di callgraph. Puoi usarlo eseguendo il tuo profiler (esempio per `gprof`):

```
# compile with profiling flags
g++ *.cpp -pg
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



Profilazione dell'uso della CPU con gcc e Google Perf Tools

Google Perf Tools fornisce anche un profilo della CPU, con un'interfaccia leggermente più amichevole. Per usarlo:

1. Installa Google Perf Tools
2. Compila il tuo codice come al solito
3. Aggiungi la libreria profiler `libprofiler` al percorso di caricamento della libreria in fase di runtime
4. Usa `pprof` per generare un profilo di esecuzione piatto o un diagramma del callgraph

Per esempio:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000
./main
```

dove:

- `CPUPROFILE` indica il file di output per i dati di profilazione
- `CPUPROFILE_FREQUENCY` indica la frequenza di campionamento del profiler;

Usa `pprof` per post-elaborare i dati di profilazione.

È possibile generare un profilo di chiamata flat come testo:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text --lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
      22  32.8%  32.8%       67 100.0% longRunningFoo ??:0
      20  29.9%  62.7%       20  29.9% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
       4   6.0%  68.7%        4   6.0% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
       3   4.5%  73.1%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:388
       3   4.5%  77.6%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:401
       2   3.0%  80.6%        2   3.0% __munmap /build/eglibc-3GlaMS/eglibc-
2.19/misc/../sysdeps/unix/syscall-template.S:81
       2   3.0%  83.6%       12  17.9% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:298
       2   3.0%  86.6%        2   3.0% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:385
       2   3.0%  89.6%        2   3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:26
       1   1.5%  91.0%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
       1   1.5%  92.5%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
       1   1.5%  94.0%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:293
       1   1.5%  95.5%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
```

```
2.19/stdlib/random.c:296
       1   1.5%  97.0%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:371
       1   1.5%  98.5%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:381
       1   1.5% 100.0%        1   1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:28
       0   0.0% 100.0%       67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-
2.19/csu/libc-start.c:287
       0   0.0% 100.0%       67 100.0% _start ??:0
       0   0.0% 100.0%       67 100.0% main ??:0
       0   0.0% 100.0%       14  20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:27
       0   0.0% 100.0%       27  40.3% std::vector::_M_emplace_back_aux ??:0
```

... oppure puoi generare un callgraph annotato in un pdf con:

```
pprof --pdf ./main main.prof > out.pdf
```

Leggi profiling online: https://riptutorial.com/it/cplusplus/topic/5347/profiling

Capitolo 91: puntatori

introduzione

Un puntatore è un indirizzo che fa riferimento a una posizione in memoria. Sono comunemente usati per consentire a funzioni o strutture di dati di conoscere e modificare la memoria senza dover copiare la memoria a cui si fa riferimento. I puntatori sono utilizzabili sia con tipi primitivi (built-in) che definiti dall'utente.

I puntatori utilizzano "dereferenzia" `*` , "indirizzo di" `&` , e "freccia" `->` operatori. Gli operatori '*' e '->' sono usati per accedere alla memoria puntata, e l'operatore `&` viene usato per ottenere un indirizzo in memoria.

Sintassi

- <Tipo di dati> * <Nome variabile>;
- <Tipo di dati> * <Nome variabile> = & <Nome variabile dello stesso tipo di dati>;
- <Tipo di dati> * <Nome variabile> = <Valore dello stesso tipo di dati>;
- int * foo; // Un puntatore che punta a un valore intero
- int * bar = & myIntVar;
- long * bar [2];
- long * bar [] = {& myLongVar1, & myLongVar2}; // Uguale a: long * bar [2]

Osservazioni

Essere consapevoli dei problemi quando si dichiarano più puntatori sulla stessa linea.

```
int* a, b, c; //Only a is a pointer, the others are regular ints.

int* a, *b, *c; //These are three pointers!

int *foo[2]; //Both *foo[0] and *foo[1] are pointers.
```

Examples

Nozioni di base del puntatore

C ++ 11

Nota: in tutti i seguenti casi, si presume l'esistenza di `nullptr` costante C ++ 11. Per le versioni precedenti, sostituire `nullptr` con `NULL` , la costante utilizzata per riprodurre un ruolo simile.

Creazione di una variabile puntatore

Una variabile puntatore può essere creata usando la sintassi specifica `*` , ad es `int *pointer_to_int;` .
Quando una variabile è di *tipo puntatore* (`int *`), contiene solo un indirizzo di memoria. L'indirizzo di memoria è la posizione in cui sono memorizzati i dati del *tipo sottostante* (`int`).

La differenza è evidente quando si confronta la dimensione di una variabile con la dimensione di un puntatore allo stesso tipo:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Prendendo l'indirizzo di un'altra variabile

I puntatori possono essere assegnati tra loro come normali variabili; in questo caso, è l' **indirizzo di memoria** che viene copiato da un puntatore a un altro, **non i dati effettivi a** cui punta un puntatore.
Inoltre, possono assumere il valore `nullptr` che rappresenta una posizione di memoria nulla. Un puntatore uguale a `nullptr` contiene una posizione di memoria non valida e quindi non fa riferimento a dati validi.

È possibile ottenere l'indirizzo di memoria di una variabile di un determinato tipo mediante il prefisso della variabile con l' *indirizzo* dell'operatore `&` . Il valore restituito da `&` è un puntatore al tipo sottostante che contiene l'indirizzo di memoria della variabile (che è un dato valido **finché la variabile non esce dall'ambito**).

```
// Copy `p_bar0` into `p_bar_1`.
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;
```

```
// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

In contrasto con i riferimenti:

- l'assegnazione di due puntatori non sovrascrive la memoria a cui fa riferimento il puntatore assegnato;
- i puntatori possono essere nulli.
- l' *indirizzo* dell'operatore è richiesto esplicitamente.

Accedere al contenuto di un puntatore

Come richiedere un indirizzo `&` , oltre all'accesso al contenuto, è necessario l'utilizzo *dell'operatore di dereferenza* `*` come prefisso. Quando un puntatore viene dereferenziato, diventa una variabile del tipo sottostante (in realtà, un riferimento ad esso). Può quindi essere letto e modificato, se non `const` .

```
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

La combinazione di `*` e l'operatore `.` è abbreviato da `->` :

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " <<  p_bar0->foo1  << std::endl; // Prints 5
```

Dereferenziare i puntatori non validi

Quando si dereferenzia un puntatore, è necessario assicurarsi che punti a dati validi. Dereferenziare un puntatore non valido (o un puntatore nullo) può portare alla violazione di accesso alla memoria, o leggere o scrivere dati inutili.

```
big_struct *never_do_this() {
   // This is a local variable. Outside `never_do_this` it doesn't exist.
   big_struct retval;
   retval.foo1 = 11;
   // This returns the address of `retval`.
   return &retval;
   // `retval` is destroyed and any code using the value returned
   // by `never_do_this` has a pointer to a memory location that
   // contains garbage data (or is inaccessible).
}
```

In tale scenario, `g++` e `clang++` rilasciano correttamente gli avvertimenti:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-
Wreturn-stack-address]
(Gcc)   warning: address of local variable ‘retval’ returned [-Wreturn-local-addr]
```

Quindi, bisogna fare attenzione quando i puntatori sono argomenti di funzioni, in quanto potrebbero essere nulli:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Operazioni di puntamento

Ci sono due operatori per i puntatori: Indirizzo-di operatore (&): restituisce l'indirizzo di memoria del suo operando. Operatore Contents-of (Dereference) (*): restituisce il valore della variabile che si trova all'indirizzo specificato dal suo operatore.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr
```

L'asterisco (*) viene utilizzato nel dichiarare un puntatore con lo scopo semplice di indicare che si tratta di un puntatore. Non confondere questo con l'operatore di **dereferenziazione** , che viene utilizzato per ottenere il valore situato all'indirizzo specificato. Sono semplicemente due cose diverse rappresentate con lo stesso segno.

Puntatore aritmetico

Incrementa / Decrementa

Un puntatore può essere incrementato o decrementato (prefisso e postfix). L'incremento di un puntatore fa avanzare il valore del puntatore all'elemento nell'array di un elemento oltre l'elemento correntemente puntato. Il decremento di un puntatore lo sposta sull'elemento precedente dell'array.

L'aritmetica del puntatore non è consentita se il tipo a cui punta il puntatore non è completo. `void` è sempre un tipo incompleto.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011  in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

Se viene incrementato un puntatore all'elemento end, il puntatore punta a un elemento oltre la fine dell'array. Tale puntatore non può essere dereferenziato, ma può essere decrementato.

L'incremento di un puntatore all'elemento one-past-end nell'array o il decremento di un puntatore al primo elemento di un array produce un comportamento non definito.

Un puntatore a un oggetto non matrice può essere trattato, ai fini dell'aritmetica del puntatore, come se fosse una matrice di dimensione 1.

Addizione / sottrazione

I valori interi possono essere aggiunti ai puntatori; agiscono come incrementi, ma per un numero specifico anziché per 1. I valori interi possono essere sottratti dai puntatori, agendo come decremento del puntatore. Come con l'incremento / decremento, il puntatore deve puntare a un tipo completo.

```
char* str = new char[10];  // str = 0x010
str += 2;                  // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10];    // arr = 0x100
arr += 2;                  // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) ==
4.
```

Differenziamento del puntatore

La differenza tra due puntatori allo stesso tipo può essere calcolata. I due puntatori devono essere all'interno dello stesso oggetto matrice; risultati del comportamento altrimenti indefiniti.

Dato due puntatori `P` e `Q` nello stesso array, se `P` è l' `i` -esimo elemento dell'array, e `Q` è il `j` -esimo elemento, allora `P - Q` è `i - j` . Il tipo del risultato è `std::ptrdiff_t` , da `<cstddef>` .

```
char* start = new char[10];  // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Leggi puntatori online: https://riptutorial.com/it/cplusplus/topic/3056/puntatori

Capitolo 92: Puntatori ai membri

Sintassi

- Supponendo una classe chiamata Class ...
    - scrivi * ptr = & Class :: member; // Indica solo membri statici
    - digitare Class :: * ptr = & Class :: member; // Indica i membri della classe non statici
- Per i puntatori ai membri della classe non statici, date le seguenti due definizioni:
    - Istanza di classe;
    - Classe * p = & istanza;
- Puntatori alle variabili dei membri della classe
    - ptr = & Class :: i; // Punta alla variabile i all'interno di ogni classe
    - istanza. * ptr = 1; // Accedi all'istanza i
    - p -> * ptr = 1; // Accedi a p i
- Puntatori alle funzioni dei membri della classe
    - ptr = & Class :: F; // Punta a funzionare 'F' in ogni classe
    - (. Esempio * ptr) (5); // Chiama l'istanza F
    - (P -> * PTR) (6); // Chiama p F

Examples

Puntatori a funzioni membro statiche

Una funzione membro `static` è proprio come una normale funzione C / C ++, tranne con scope:

- È all'interno di una `class` , quindi ha bisogno del suo nome decorato con il nome della classe;
- Ha accessibilità, con `public` , `protected` o `private` .

Quindi, se hai accesso alla funzione membro `static` e la decori correttamente, puoi indicare la funzione come qualsiasi normale funzione al di fuori di una `class` :

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
    static int Static(int i) { return 3*i; }
}; // Class
```

```
int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;          // Point to one function
    fn(3);               // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);               // Call it
 } // main()
```

Puntatori alle funzioni membro

Per accedere a una funzione membro di una classe, è necessario disporre di un "handle" per l'istanza particolare, come l'istanza stessa o un puntatore o un riferimento ad essa. Data un'istanza di classe, puoi puntare a vari dei suoi membri con un puntatore-membro, SE si ottiene la sintassi corretta! Ovviamente, il puntatore deve essere dichiarato dello stesso tipo di quello a cui si sta puntando ...

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
    fn = &Class::B;   // fn now points to B within any Class
    (p->*fn)(6);      // Pass 6 to c's (via p) function B (via fn)
} // main()
```

A differenza dei puntatori alle variabili membro (nell'esempio precedente), l'associazione tra l'istanza di classe e il puntatore membro deve essere strettamente associata a parentesi, che sembra un po 'strana (come se il `.*` E `->*` non siano strani abbastanza!)

Puntatori alle variabili membro

Per accedere a un membro di una `class` , è necessario disporre di un "handle" per l'istanza particolare, come l'istanza stessa o un puntatore o un riferimento ad esso. Data un'istanza di `class` , puoi puntare a vari dei suoi membri con un puntatore-membro, SE si ottiene la sintassi corretta! Ovviamente, il puntatore deve essere dichiarato dello stesso tipo di quello a cui si sta puntando ...

```
class Class {
public:
    int x, y, z;
    char m, n, o;
```

```
}; // Class

int x;  // Global variable

int main() {
    Class c;        // Need a Class instance to play with
    Class *p = &c;  // Need a Class pointer to play with

    int *p_i;       // Pointer to an int

    p_i = &x;       // Now pointing to x
    p_i = &c.x;     // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_c_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_c_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

La sintassi del puntatore-membro richiede alcuni elementi sintattici aggiuntivi:

- Per definire il tipo di puntatore, è necessario menzionare il tipo di base, oltre al fatto che si trova all'interno di una classe: `int Class::*ptr;` .
- Se si dispone di una classe o di riferimento e si desidera utilizzare con un membro puntatore-a-, è necessario utilizzare il `.*` Operatore (simile al `.` Operatore).
- Se si ha un puntatore a una classe e si desidera utilizzarlo con un puntatore a membro, è necessario utilizzare l'operatore `->*` (simile all'operatore `->`).

Puntatori a variabili membro statiche

Una variabile membro `static` è proprio come una normale variabile C / C ++, tranne con scope:

- È all'interno di una `class` , quindi ha bisogno del suo nome decorato con il nome della classe;
- Ha accessibilità, con `public` , `protected` o `private` .

Quindi, se hai accesso alla variabile membro `static` e la decori correttamente, puoi puntare alla variabile come qualsiasi variabile normale al di fuori di una `class` :

```
class Class {
public:
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)

int j = 2;   // Just another global variable

int main() {
    int k = 3; // Local variable
```

```
    int *p;

    p = &k;   // Point to k
    *p = 2;   // Modify it
    p = &j;   // Point to j
    *p = 3;   // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;   // Modify it
} // main()
```

Leggi Puntatori ai membri online: https://riptutorial.com/it/cplusplus/topic/2130/puntatori-ai-membri

Capitolo 93: Puntatori intelligenti

Sintassi

- `std::shared_ptr<ClassType> variableName = std::make_shared<ClassType>(arg1, arg2, ...);`
- `std::shared_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`
- `std::unique_ptr<ClassType> variableName = std::make_unique<ClassType>(arg1, arg2, ...);` // C ++ 14
- `std::unique_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`

Osservazioni

Il C ++ non è un linguaggio gestito dalla memoria. La memoria allocata dinamicamente (cioè gli oggetti creati con `new`) sarà "trapelata" se non è esplicitamente deallocata (con `delete`). È responsabilità del programmatore assicurarsi che la memoria allocata dinamicamente venga liberata prima di scartare l'ultimo puntatore a quell'oggetto.

I puntatori intelligenti possono essere utilizzati per gestire automaticamente l'ambito della memoria allocata dinamicamente (ovvero quando l'ultimo riferimento del puntatore esce dall'ambito viene cancellato).

Puntatori intelligenti sono preferiti su puntatori "grezzi" nella maggior parte dei casi. Rendono esplicita la semantica della proprietà della memoria allocata dinamicamente, comunicando nel suo nome se un oggetto è destinato a essere condiviso o di proprietà esclusiva.

Usa `#include <memory>` per poter usare puntatori intelligenti.

Examples

Condivisione della proprietà (std :: shared_ptr)

Il modello di classe `std::shared_ptr` definisce un puntatore condiviso che è in grado di condividere la proprietà di un oggetto con altri puntatori condivisi. Questo contrasta con `std::unique_ptr` che rappresenta la proprietà esclusiva.

Il comportamento di condivisione è implementato attraverso una tecnica nota come conteggio dei riferimenti, in cui il numero di puntatori condivisi che puntano all'oggetto è memorizzato accanto a esso. Quando questo conteggio raggiunge lo zero, attraverso la distruzione o la riassegnazione dell'ultima istanza di `std::shared_ptr` , l'oggetto viene automaticamente distrutto.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Per creare più puntatori intelligenti che condividono lo stesso oggetto, dobbiamo creare un altro `shared_ptr` che alias il primo puntatore condiviso. Ecco due modi per farlo:

```
std::shared_ptr<Foo> secondShared(firstShared);  // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared;                      // 2nd way: Assigning
```

Uno dei due modi sopra rende `secondShared` un puntatore condiviso che condivide la proprietà della nostra istanza di `Foo` con `firstShared` .

Il puntatore intelligente funziona esattamente come un puntatore raw. Questo significa che puoi usare `*` per dereferenziarli. Anche l'operatore regolare `->` funziona:

```
secondShared->test(); // Calls Foo::test()
```

Infine, quando l'ultimo alias `shared_ptr` esce dall'ambito, viene chiamato il distruttore della nostra istanza di `Foo` .

Avvertenza: la costruzione di un file `shared_ptr` può generare un'eccezione `bad_alloc` quando vengono `bad_alloc` dati aggiuntivi per la semantica della proprietà condivisa. Se il costruttore riceve un puntatore regolare, assume di possedere l'oggetto puntato e chiama il deleter se viene lanciata un'eccezione. Ciò significa che `shared_ptr<T>(new T(args))` non perderà un oggetto `T` se l'allocazione di `shared_ptr<T>` fallisce. Tuttavia, è consigliabile utilizzare `make_shared<T>(args)` o `allocate_shared<T>(alloc, args)` , che consentono all'implementazione di ottimizzare l'allocazione della memoria.

Allocazione di array ([]) utilizzando shared_ptr

C ++ 11 C ++ 17

Sfortunatamente, non esiste un modo diretto per allocare gli array usando `make_shared<>` .

È possibile creare array per `shared_ptr<>` usando `new` e `std::default_delete` .

Ad esempio, per allocare una matrice di 10 numeri interi, possiamo scrivere il codice come

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

La specifica di `std::default_delete` è obbligatoria qui per assicurarti che la memoria allocata sia correttamente ripulita usando `delete[]` .

Se conosciamo le dimensioni al momento della compilazione, possiamo farlo in questo modo:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
  std::shared_ptr<T> operator()const{
    auto r = std::make_shared<std::array<T,N>>();
    if (!r) return {};
    return {r.data(), r};
  }
```

```
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }
```

quindi `make_shared_array<int[10]>` restituisce un `shared_ptr<int>` punta a 10 interi di default costruiti.

C ++ 17

Con C ++ 17, `shared_ptr` ottenuto un supporto speciale per i tipi di array. Non è più necessario specificare esplicitamente l'array-deleter e il puntatore condiviso può essere de-referenziato usando l'operatore `[]` dell'indice di array:

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

I puntatori condivisi possono puntare a un sottooggetto dell'oggetto che possiede:

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Sia `p2` che `p1` possiedono l'oggetto di tipo `Foo` , ma `p2` punta al suo membro `int x` . Ciò significa che se `p1` esce dallo scope o viene riassegnato, l'oggetto `Foo` sottostante sarà ancora vivo, assicurando che `p2` non penzoli.

Importante: un `shared_ptr` conosce solo se stesso e tutti gli altri `shared_ptr` creati con il costruttore alias. Non conosce altri puntatori, inclusi tutti gli altri `shared_ptr` creati con un riferimento alla stessa istanza di `Foo` :

```
Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                 // deleted already!!
```

Trasferimento di proprietà di shared_ptr

Per impostazione predefinita, `shared_ptr` incrementa il conteggio dei riferimenti e non trasferisce la proprietà. Tuttavia, può essere fatto per trasferire la proprietà usando `std::move` :

```
shared_ptr<int> up = make_shared<int>();
// Transferring the ownership
shared_ptr<int> up2 = move(up);
```

```
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1
```

Condivisione con proprietà temporanea (std :: weak_ptr)

Le istanze di `std::weak_ptr` possono puntare a oggetti di proprietà di istanze di `std::shared_ptr` mentre diventano essi stessi solo proprietari temporanei. Ciò significa che i puntatori deboli non alterano il conteggio dei riferimenti dell'oggetto e quindi non impediscono la cancellazione di un oggetto se tutti i puntatori condivisi dell'oggetto sono riassegnati o distrutti.

Nel seguente esempio vengono utilizzate istanze di `std::weak_ptr` modo che la distruzione di un oggetto ad albero non sia inibita:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

Quando i nodi figli vengono aggiunti ai figli del nodo radice, il loro `parent` membro `std::weak_ptr` viene impostato sul nodo radice. Il membro `parent` viene dichiarato come un puntatore debole anziché un puntatore condiviso, in modo che il conteggio dei riferimenti del nodo radice non venga incrementato. Quando il nodo root viene ripristinato alla fine di `main()` , la root viene distrutta. Dal momento che i restanti solo `std::shared_ptr` riferimenti ai nodi secondari erano contenuti nella collezione del radice `children` , tutti i figli vengono successivamente distrutti pure.

A causa dei dettagli di implementazione del blocco di controllo, la memoria allocata shared_ptr potrebbe non essere rilasciata fino a `shared_ptr` contatore di riferimento `weak_ptr` e il `weak_ptr` riferimento `weak_ptr` entrambi raggiungono lo zero.

```
#include <memory>
int main()
{
```

```
    {
         std::weak_ptr<int> wk;
         {
             // std::make_shared is optimized by allocating only once
             // while std::shared_ptr<int>(new int(42)) allocates twice.
             // Drawback of std::make_shared is that control block is tied to our integer
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // sh memory should be released at this point...
         }
         // ... but wk is still alive and needs access to control block
     }
     // now memory is released (sh and wk)
}
```

Poiché `std::weak_ptr` non mantiene `std::weak_ptr` suo oggetto di riferimento, l'accesso diretto ai dati tramite uno `std::weak_ptr` non è possibile. Fornisce invece una funzione membro `lock()` che tenta di recuperare un oggetto `std::shared_ptr` sull'oggetto di riferimento:

```
#include <cassert>
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         std::shared_ptr<int> sp;
         {
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // calling lock will create a shared_ptr to the object referenced by wk
             sp = wk.lock();
             // sh will be destroyed after this point, but sp is still alive
         }
         // sp still keeps the data alive.
         // At this point we could even call lock() again
         // to retrieve another shared_ptr to the same data from wk
         assert(*sp == 42);
         assert(!wk.expired());
         // resetting sp will delete the data,
         // as it is currently the last shared_ptr with ownership
         sp.reset();
         // attempting to lock wk now will return an empty shared_ptr,
         // as the data has already been deleted
         sp = wk.lock();
         assert(!sp);
         assert(wk.expired());
     }
}
```

Proprietà univoca (std :: unique_ptr)

C ++ 11

Un `std::unique_ptr` è un modello di classe che gestisce la durata di un oggetto archiviato in modo dinamico. A differenza di `std::shared_ptr` , l'oggetto dinamico è di proprietà di una *sola istanza* di `std::unique_ptr` in qualsiasi momento,

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Nota: `std::unique_ptr` è disponibile da C ++ 11 e `std::make_unique` da C ++ 14.)

Solo la variabile `ptr` contiene un puntatore a un `int` allocato dinamicamente. Quando un puntatore univoco che possiede un oggetto esce dall'ambito, l'oggetto di proprietà viene eliminato, ovvero viene chiamato il suo distruttore se l'oggetto è di tipo classe e viene rilasciata la memoria per quell'oggetto.

Per usare `std::unique_ptr` e `std::make_unique` con tipi di array, usa le loro specializzazioni di array:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

È possibile accedere a `std::unique_ptr` proprio come un puntatore raw, perché sovraccarica quegli operatori.

È possibile trasferire la proprietà dei contenuti di un puntatore intelligente a un altro puntatore utilizzando `std::move` , che farà in modo che il puntatore smart originale punti a `nullptr` .

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);

int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

Passando `unique_ptr` a funzioni come parametro:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

Restituzione di `unique_ptr` dalle funzioni. Questo è il metodo preferito di scrittura C ++ 11 per le funzioni di fabbrica, in quanto trasmette chiaramente la semantica della proprietà del `unique_ptr` : il chiamante possiede il `unique_ptr` risultante ed è responsabile di esso.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Confronta questo a:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

C ++ 14

Il modello di classe `make_unique` viene fornito dal C ++ 14. È facile aggiungerlo manualmente al codice C ++ 11:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

C ++ 11

A differenza del puntatore intelligente *stupido* (`std::auto_ptr`), `unique_ptr` può anche essere istanziato con l'allocazione vettoriale (*non con* `std::vector`). Gli esempi precedenti riguardavano allocazioni *scalari* . Ad esempio, per disporre di un array intero allocato dinamicamente per 10 elementi, si specificherà `int[]` come tipo di modello (e non solo `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Quale può essere semplificato con:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Ora, usi `arr_ptr` come se fosse un array:

```
arr_ptr[2] =  10; // Modify third element
```

Non devi preoccuparti della disallocazione. Questa versione specializzata del modello chiama appropriatamente costruttori e distruttori. L'uso della versione vettoriale di `unique_ptr` o di un `vector` stesso è una scelta personale.

Nelle versioni precedenti al C ++ 11, `std::auto_ptr` era disponibile. A differenza di `unique_ptr` , è consentito copiare `auto_ptr` s, in base al quale il sorgente `ptr` perderà la proprietà del puntatore contenuto e il target lo riceverà.

Utilizzare i deletatori personalizzati per creare un wrapper per un'interfaccia C

Molte interfacce C come SDL2 hanno le proprie funzioni di cancellazione. Ciò significa che non è possibile utilizzare direttamente i puntatori intelligenti:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Invece, è necessario definire il proprio deleter. Gli esempi qui usano la struttura `SDL_Surface` che dovrebbe essere liberata usando la funzione `SDL_FreeSurface()` , ma dovrebbero essere adattabili a molte altre interfacce C.

Il deleter deve essere richiamabile con un argomento pointer e, quindi, può essere ad esempio un semplice puntatore a funzione:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Anche qualsiasi altro oggetto callable funzionerà, ad esempio una classe con un `operator()` :

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};

std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
                                                         // as the deleter is value-
initialized
```

Questo non solo ti offre una gestione della memoria automatica sicura, zero overhead (se usi `unique_ptr`), ma ottieni anche un'eccezionale sicurezza.

Si noti che il deleter è parte del tipo per `unique_ptr` e l'implementazione può utilizzare l' ottimizzazione di base vuota per evitare qualsiasi modifica delle dimensioni per i deletatori personalizzati vuoti. Quindi mentre `std::unique_ptr<SDL_Surface, SurfaceDeleter>` e `std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)>` risolvono lo stesso problema in modo simile, il primo tipo è ancora solo la dimensione di un puntatore mentre il quest'ultimo tipo deve contenere *due* puntatori: sia la `SDL_Surface*` che il puntatore della funzione! Quando si dispone di funzioni di deleter personalizzate, è preferibile racchiudere la funzione in un tipo vuoto.

Nei casi in cui il conteggio dei riferimenti è importante, si potrebbe usare un `shared_ptr` invece di un `unique_ptr` . `shared_ptr` memorizza sempre un deleter, questo cancella il tipo del deleter, che potrebbe essere utile nelle API. Gli svantaggi dell'utilizzo di `shared_ptr` su `unique_ptr` includono un costo di memoria più elevato per l'archiviazione del deleter e un costo delle prestazioni per il mantenimento del conteggio dei riferimenti.

```
// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

C ++ 17

Con `template auto` , possiamo rendere ancora più semplice il wrapping dei nostri delet personalizzati:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Con il quale l'esempio sopra è semplicemente:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Qui, lo scopo `auto` è quello di gestire tutte le funzioni libere, sia che restituiscano `void` (ad es. `SDL_FreeSurface`) o meno (es. `fclose`).

Proprietà unica senza spostamento della semantica (auto_ptr)

C ++ 11

NOTA: `std::auto_ptr` è stato deprecato in C ++ 11 e verrà rimosso in C ++ 17. Dovresti usarlo solo se sei costretto a usare C ++ 03 o prima e sei disposto a stare attento. Si consiglia di passare a unique_ptr in combinazione con `std::move` per sostituire il comportamento di `std::auto_ptr` .

Prima di avere `std::unique_ptr` , prima di spostare la semantica, avevamo `std::auto_ptr` . `std::auto_ptr` fornisce una proprietà esclusiva ma trasferisce la proprietà sulla copia.

Come con tutti i puntatori intelligenti, `std::auto_ptr` pulisce automaticamente le risorse (vedi RAII):

```
{
    std::auto_ptr<int> p(new int(42));
    std::cout << *p;
} // p is deleted here, no memory leaked
```

ma consente solo un proprietario:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
  // px is now empty
```

Questo permette di usare std :: auto_ptr per mantenere la proprietà esplicita e unica a rischio di perdere la proprietà non intenzionale:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
       // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

Il trasferimento di proprietà è avvenuto nel costruttore "copia". Il costruttore di copie di `auto_ptr` e l'operatore di assegnazione copia eseguono i loro operandi con riferimento non `const` modo che possano essere modificati. Un esempio di implementazione potrebbe essere:

```
template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }

    /* other functions ... */
};
```

Questo interrompe la semantica della copia, che richiede che la copia di un oggetto ti lasci con due versioni equivalenti. Per qualsiasi tipo di copia, `T` , dovrei essere in grado di scrivere:

```
T a = ...;
T b(a);
assert(b == a);
```

Ma per `auto_ptr` , questo non è il caso. Di conseguenza, non è sicuro inserire `auto_ptr` nei contenitori.

Ottenere un shared_ptr riferendosi a questo

`enable_shared_from_this` ti permette di ottenere un'istanza `shared_ptr` valida a `this` .

Derivando la tua classe dal modello di classe `enable_shared_from_this` , erediti un metodo `shared_from_this` che restituisce un'istanza `shared_ptr` a `this` .

Si noti che l'oggetto deve essere creato come `shared_ptr` al primo posto:

```
#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Nota (2) non è possibile chiamare `enable_shared_from_this` all'interno del costruttore.

```
#include <memory> // enable_shared_from_this

class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

Se si utilizza `shared_from_this()` su un oggetto non di proprietà di `shared_ptr` , come un oggetto automatico locale o un oggetto globale, il comportamento non è definito. Dal momento che C ++ 17 getta invece `std::bad_alloc` .

Usare `shared_from_this()` da un costruttore è equivalente a usarlo su un oggetto non di proprietà di un `shared_ptr` , perché gli oggetti sono posseduti da `shared_ptr` dopo il ritorno del costruttore.

Puntatori di casting std :: shared_ptr

Non è possibile utilizzare direttamente `static_cast` , `const_cast` , `dynamic_cast` e `reinterpret_cast` su `std::shared_ptr` per recuperare un puntatore che condivide la proprietà con il puntatore passato come argomento. Devono invece essere utilizzate le funzioni `std::static_pointer_cast` ,

`std::const_pointer_cast` , `std::dynamic_pointer_cast` e `std::reinterpret_pointer_cast` :

```
struct Base { virtual ~Base() noexcept {}; };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Si noti che `std::reinterpret_pointer_cast` non è disponibile in C ++ 11 e C ++ 14, poiché è stato proposto solo da N3920 e adottato in Library Fundamentals TS a febbraio 2014 . Tuttavia, può essere implementato come segue:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Scrivere un puntatore intelligente: value_ptr

Un `value_ptr` è un puntatore intelligente che si comporta come un valore. Quando viene copiato, ne copia il contenuto. Quando viene creato, crea il suo contenuto.

```
// Like std::default_delete:
template<class T>
struct default_copier {
  // a copier must handle a null T const* in and return null:
  T* operator()(T const* tin)const {
    if (!tin) return nullptr;
    return new T(*tin);
  }
  void operator()(void* dest, T const* tin)const {
    if (!tin) return;
    return new(dest) T(*tin);
  }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
// the value pointer type itself:
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
  class Base=std::unique_ptr<T, Deleter>
>
struct value_ptr:Base, private Copier {
  using copier_type=Copier;
  // also typedefs from unique_ptr

  using Base::Base;

  value_ptr( T const& t ):
    Base( std::make_unique<T>(t) ),
    Copier()
  {}
  value_ptr( T && t ):
    Base( std::make_unique<T>(std::move(t)) ),
    Copier()
```

```
{}
// almost-never-empty:
    value_ptr():
  Base( std::make_unique<T>() ),
  Copier()
{}
value_ptr( empty_ptr_t ) {}

value_ptr( Base b, Copier c={} ):
  Base(std::move(b)),
  Copier(std::move(c))
{}

Copier const& get_copier() const {
  return *this;
}

value_ptr clone() const {
  return {
    Base(
      get_copier()(this->get()),
      this->get_deleter()
    ),
    get_copier()
  };
}
value_ptr(value_ptr&&)=default;
value_ptr& operator=(value_ptr&&)=default;

value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
value_ptr& operator=(value_ptr const&o) {
  if (o && *this) {
    // if we are both non-null, assign contents:
    **this = *o;
  } else {
    // otherwise, assign a clone (which could itself be null):
    *this = o.clone();
  }
  return *this;
}
value_ptr& operator=( T const& t ) {
  if (*this) {
    **this = t;
  } else {
    *this = value_ptr(t);
  }
  return *this;
}
value_ptr& operator=( T && t ) {
  if (*this) {
    **this = std::move(t);
  } else {
    *this = value_ptr(std::move(t));
  }
  return *this;
}
T& get() { return **this; }
T const& get() const { return **this; }
T* get_pointer() {
  if (!*this) return nullptr;
  return std::addressof(get());
```

```
  }
  T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  // operator-> from unique_ptr
};
template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
  return {std::make_unique<T>(std::forward<Args>(args)...)};
}
```

Questo valore value_ptr è vuoto solo se lo costruisci con `empty_ptr_t` o se ti sposti da esso. Espone il fatto che è un `unique_ptr` , quindi `explicit operator bool() const` lavora su di esso. `.get()` è stato modificato per restituire un riferimento (poiché non è quasi mai vuoto) e `.get_pointer()` restituisce invece un puntatore.

Questo puntatore intelligente può essere utile per i casi `pImpl` , dove vogliamo semantica del valore ma non vogliamo nemmeno esporre il contenuto di `pImpl` al di fuori del file di implementazione.

Con una `Copier` non predefinita, può persino gestire classi di base virtuali che sanno come generare istanze delle loro derivate e trasformarle in tipi di valore.

Leggi Puntatori intelligenti online: https://riptutorial.com/it/cplusplus/topic/509/puntatori-intelligenti

Capitolo 94: RAII: l'acquisizione delle risorse è inizializzata

Osservazioni

RAII è l'acronimo di **R** esource **A** cquisition **I** s **I** nitialization. Anche a volte indicato come SBRM (Scope-Based Resource Management) o RRID (Resource Release Is Destruction), RAII è un idioma utilizzato per legare le risorse alla durata dell'oggetto. In C ++, il distruttore per un oggetto viene sempre eseguito quando un oggetto esce dall'ambito di applicazione - possiamo trarne vantaggio per collegare la pulizia delle risorse alla distruzione dell'oggetto.

Ogni volta che è necessario acquisire alcune risorse (ad es. Un blocco, un handle di file, un buffer allocato) che sarà necessario rilasciare, si dovrebbe prendere in considerazione l'utilizzo di un oggetto per gestire tale gestione delle risorse. Lo sbobinamento dello stack avverrà indipendentemente dall'eccezione o dall'uscita dell'ambito iniziale, quindi l'oggetto gestore risorse ripulirà la risorsa per te senza dover considerare attentamente tutti i possibili percorsi di codice corrente e futuro.

Vale la pena notare che RAII non completamente libera lo sviluppatore di pensare alla durata delle risorse. Un caso è, ovviamente, una chiamata crash o exit (), che impedirà il richiamo dei distruttori. Dal momento che il SO pulirà il processo, le risorse locali come la memoria dopo la fine di un processo, questo non è un problema nella maggior parte dei casi. Tuttavia con le risorse di sistema (ad es. Named pipe, lock file, shared memory) hai ancora bisogno di strutture per gestire il caso in cui un processo non si è ripulito da solo, cioè all'avvio test se il file di lock è lì, se lo è, verificare il processo con il pid effettivamente esiste, quindi agire di conseguenza.

Un'altra situazione è quando un processo unix chiama una funzione della famiglia exec, cioè dopo un fork-exec per creare un nuovo processo. Qui, il processo figlio avrà una copia completa della memoria dei genitori (inclusi gli oggetti RAII) ma una volta che è stato chiamato exec, nessuno dei distruttori sarà chiamato in quel processo. D'altra parte, se un processo è biforcato e nessuno dei processi chiama exec, tutte le risorse vengono ripulite in entrambi i processi. Questo è corretto solo per tutte le risorse effettivamente duplicate nel fork, ma con le risorse di sistema, entrambi i processi avranno solo un riferimento alla risorsa (cioè il percorso di un file di blocco) e cercheranno entrambi di rilasciarlo individualmente, causando potenzialmente l'altro processo fallisce.

Examples

Blocco

Cattivo bloccaggio:

```
std::mutex mtx;
```

```
void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock();   // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock();       // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock();       // Must also force unlock in the presence of
        throw;              // exceptions and allow the exception to continue.
    }
}
```

Questo è il modo sbagliato per implementare il blocco e lo sblocco del mutex. Per garantire il corretto rilascio del mutex con `unlock()` necessario che il programmatore si assicuri che tutti i flussi risultanti dall'uscita della funzione risultino in una chiamata a `unlock()` . Come mostrato sopra, si tratta di processi fragili in quanto richiede a qualsiasi manutentore di continuare a seguire il modello manualmente.

Usando una classe appropriatamente predisposta per implementare RAII, il problema è banale:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx);   // constructor locks.
                                           // destructor unlocks. destructor call
                                           // guaranteed by language.
    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

`lock_guard` è un modello di classe estremamente semplice che chiama semplicemente `lock()` sul suo argomento nel suo costruttore, mantiene un riferimento all'argomento e chiama `unlock()` sull'argomento nel suo distruttore. Cioè, quando `lock_guard` esce dallo scope, il `mutex` è garantito per essere sbloccato. Non importa se il motivo per cui è andato fuori ambito è un'eccezione o un ritorno anticipato - tutti i casi sono gestiti; indipendentemente dal flusso di controllo, abbiamo garantito che sbloccheremo correttamente.

Infine / ScopeExit

Per i casi in cui non vogliamo scrivere classi speciali per gestire alcune risorse, possiamo scrivere una classe generica:

```
template<typename Function>
```

```
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;
private:
    Function f;
};
// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)};
}
```

E il suo utilizzo di esempio

```
void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&](){ v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}
```

Nota (1): alcune discussioni sulla definizione del distruttore devono essere considerate per gestire l'eccezione:

- `~Finally() noexcept { f(); }` : `std::terminate` viene chiamato in caso di eccezione
- `~Finally() noexcept(noexcept(f())) { f(); }` : terminate () viene chiamato solo in caso di eccezione durante lo srotolamento dello stack.
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */} }` No `std::terminate` chiamato, ma non possiamo gestire l'errore (anche per lo stacco non stack).

ScopeSuccess (c ++ 17)

C ++ 17

Grazie a `int std::uncaught_exceptions()` , possiamo implementare azioni che vengono eseguite solo in caso di successo (nessuna eccezione generata nell'ambito). In precedenza `bool std::uncaught_exception()` consente solo di rilevare se è in esecuzione **uno** srotolamento dello stack.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
```

```
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Produzione:

```
Success 1
```

ScopeFail (c ++ 17)

C ++ 17

Grazie a `int std::uncaught_exceptions()` , possiamo implementare un'azione che viene eseguita solo in caso di errore (eccezione generata nell'ambito). In precedenza `bool`

`std::uncaught_exception()` consente solo di rilevare se è in esecuzione **uno** srotolamento dello stack.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Produzione:

```
Failure 2
```

Leggi RAII: l'acquisizione delle risorse è inizializzata online:
https://riptutorial.com/it/cplusplus/topic/1320/raii--l-acquisizione-delle-risorse-e-inizializzata

Capitolo 95: Restituzione di diversi valori da una funzione

introduzione

Esistono molte situazioni in cui è utile restituire diversi valori da una funzione: ad esempio, se si desidera immettere un articolo e restituire il prezzo e il numero in magazzino, questa funzionalità potrebbe essere utile. Ci sono molti modi per farlo in C ++ e la maggior parte coinvolge l'STL. Tuttavia, se si desidera evitare l'STL per qualche motivo, ci sono ancora diversi modi per farlo, comprese le `structs/classes` e `arrays` .

Examples

Utilizzo dei parametri di output

I parametri possono essere utilizzati per restituire uno o più valori; questi parametri devono essere puntatori o riferimenti non `const` .

Riferimenti:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
    e = a * b;
    f = a / b;
}
```

puntatori:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {
    *c = a + b;
    *d = a - b;
    *e = a * b;
    *f = a / b;
}
```

Alcune librerie o framework usano un `#define` "OUT" vuoto per rendere abbondantemente ovvio quali parametri sono parametri di output nella firma della funzione. Questo non ha impatto funzionale e verrà compilato, ma rende la firma della funzione un po 'più chiara;

```
#define OUT

void calculate(int a, int b, OUT int& c) {
    c = a + b;
}
```

Utilizzando std :: tupla

C ++ 11

Il tipo `std::tuple` può raggruppare qualsiasi numero di valori, potenzialmente inclusi valori di tipi diversi, in un singolo oggetto di ritorno:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
   return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

In C ++ 17, è possibile utilizzare un elenco di inizializzazione rinforzato:

C ++ 17

```
std::tuple<int, int, int, int> foo(int a, int b)    {
    return {a + b, a - b, a * b, a / b};
}
```

Il recupero dei valori dalla `tuple` restituita può essere complicato, richiedendo l'uso della funzione `std::get` template:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Se i tipi possono essere dichiarati prima che la funzione ritorni, allora `std::tie` può essere utilizzato per decomprimere una `tuple` in variabili esistenti:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

Se uno dei valori restituiti non è necessario, è possibile utilizzare `std::ignore` :

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

C ++ 17

I collegamenti strutturati possono essere utilizzati per evitare `std::tie` :

```
auto [add, sub, mul, div] = foo(5,12);
```

Se vuoi restituire una tupla di riferimenti lvalue invece di una tupla di valori, usa `std::tie` al posto di `std::make_tuple` .

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
  if (b<a)
    return std::tie(b,a);
```

```
  else
    return std::tie(a,b);
}
```

che permette

```
void increase_least(int& a, int& b) {
  std::get<0>(minmax(a,b))++;
}
```

In alcuni rari casi utilizzerai `std::forward_as_tuple` invece di `std::tie` ; fai attenzione se lo fai, poiché i provvisori potrebbero non durare abbastanza a lungo per essere consumati.

Utilizzando std :: array

C ++ 11

Il contenitore `std::array` può unire un numero fisso di valori di ritorno. Questo numero deve essere noto in fase di compilazione e tutti i valori restituiti devono essere dello stesso tipo:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

Questo sostituisce gli array in stile c della `int bar[4]` form `int bar[4]` . Il vantaggio è che ora è possibile utilizzare varie funzioni di `c++` std. Fornisce anche utili funzioni membro come `at` cui è una funzione di accesso sicuro membro con controllo vincolato e `size` che consente di restituire la dimensione della matrice senza calcolo.

Utilizzando std :: pair

Il modello struct `std::pair` può raggruppare *esattamente* due valori di ritorno, di due tipi:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

Con C ++ 11 o `std::make_pair` successive, è possibile utilizzare un elenco di inizializzatori invece di `std::make_pair` :

C ++ 11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

I singoli valori della `std::pair` restituita possono essere recuperati utilizzando gli oggetti `first` e `second` member della coppia:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Produzione:

    10

Usando struct

Una `struct` può essere utilizzata per raggruppare più valori di ritorno:

C ++ 11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

auto calc = foo(5, 12);
```

C ++ 11

Invece di assegnare singoli campi, è possibile utilizzare un costruttore per semplificare la costruzione di valori restituiti:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
     return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

I singoli risultati restituiti dalla funzione `foo()` possono essere recuperati accedendo alle variabili membro del `calc struct` :

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Produzione:

    17 -7 60 0

Nota: quando si utilizza una `struct` , i valori restituiti vengono raggruppati in un singolo oggetto e accessibili utilizzando nomi significativi. Questo aiuta anche a ridurre il numero di variabili estranee create nell'ambito dei valori restituiti.

C ++ 17

Per decomprimere una `struct` restituita da una funzione, è possibile utilizzare i collegamenti strutturati . Questo pone i parametri di uscita su un piede uniforme con i parametri in-:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

L'output di questo codice è identico a quello sopra. La `struct` è ancora utilizzata per restituire i valori dalla funzione. Questo ti permette di trattare i campi individualmente.

Collegamenti strutturati

C ++ 17

C ++ 17 introduce i binding strutturati, il che rende ancora più semplice trattare tipi di reso multipli, poiché non è necessario fare affidamento su `std::tie()` o eseguire il disimballaggio manuale della tupla:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

I collegamenti strutturati possono essere utilizzati per impostazione predefinita con `std::pair` , `std::tuple` e qualsiasi tipo i cui membri di dati non statici sono tutti membri pubblici diretti o membri di una classe base non ambigua:

```
struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
auto& y = result.y;
```

Se rendi il tuo tipo "simile a una tupla", funzionerà automaticamente anche con il tuo tipo. Un tuple-like è un tipo con `tuple_size` appropriato, `tuple_element` e `get` scritto:

```
namespace my_ns {
    struct my_type {
        int x;
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

ora questo funziona:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}
```

Utilizzo di un oggetto oggetto Consumer

Siamo in grado di fornire un consumatore che verrà chiamato con i molteplici valori rilevanti:

C ++ 11

```
template <class F>
```

```
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});
```

Questo è noto come "continuation passing style" .

È possibile adattare una funzione che restituisce una tupla in una funzione di stile a passaggio continuo tramite:

C ++ 17

```
template<class Tuple>
struct continuation {
  Tuple t;
  template<class F>
  decltype(auto) operator->*(F&& f)&&{
    return std::apply( std::forward<F>(f), std::move(t) );
  }
};
std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
  std::cout << "sum is " << sum << '\n';
};
```

con versioni più complesse essendo scrivibili in C ++ 14 o C ++ 11.

Utilizzando std :: vector

Un `std::vector` può essere utile per restituire un numero dinamico di variabili dello stesso tipo. L'esempio seguente utilizza `int` come tipo di dati, ma un `std::vector` può contenere qualsiasi tipo che è banalmente copiato:

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

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);
```

```
    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Utilizzando Output Iterator

Diversi valori dello stesso tipo possono essere restituiti passando un iteratore di output alla funzione. Questo è particolarmente comune per le funzioni generiche (come gli algoritmi della libreria standard).

Esempio:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
}
```

Esempio di utilizzo:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Leggi Restituzione di diversi valori da una funzione online:
https://riptutorial.com/it/cplusplus/topic/487/restituzione-di-diversi-valori-da-una-funzione

Capitolo 96: Ricerca del nome dipendente dall'argomento

Examples

Quali funzioni sono state trovate

Le funzioni vengono trovate dapprima raccogliendo un insieme di *"classi associate"* e *"spazi dei nomi associati"* che includono uno o più dei seguenti, a seconda del tipo di argomento `T` Per prima cosa, mostriamo le regole per i nomi delle classi di classi, di enumerazione e di classe.

- Se `T` è una classe nidificata, l'enumerazione dei membri, quindi la classe circostante di essa.
- Se `T` è un'enumerazione (potrebbe *anche* essere un membro della classe!), Il namespace più interno di esso.
- Se `T` è una classe (può *anche* essere annidata!), Tutte le sue classi base e la classe stessa. Lo spazio dei nomi più interno di tutte le classi associate.
- Se `T` è un `ClassTemplate<TemplateArguments>` (questa è *anche* una classe!), Le classi e gli spazi dei nomi associati agli argomenti del tipo template, lo spazio dei nomi di qualsiasi argomento template template e la classe circostante di qualsiasi argomento template template, se un argomento template è un modello membro.

Ora ci sono anche alcune regole per i tipi built-in

- Se `T` è un puntatore a `U` o array di `U` , le classi e gli spazi dei nomi associati a `U` Esempio: `void (*fptr)(A); f(fptr);` , include gli spazi dei nomi e le classi associate a `void(A)` (vedi regola successiva).
- Se `T` è un tipo di funzione, le classi e gli spazi dei nomi associati ai parametri e ai tipi di ritorno. Esempio: `void(A)` include gli spazi dei nomi e le classi associati ad `A`
- Se `T` è un puntatore al membro, le classi e gli spazi dei nomi associati al tipo di membro (possono applicarsi sia al puntatore alle funzioni membro che al puntatore al membro dati!). Esempio: `BA::*p; void (A::*pf)(B); f(p); f(pf);` include gli spazi dei nomi e le classi associati a `A` , `B` , `void(B)` (che applica bullet sopra per i tipi di funzione).

Tutte le funzioni e i modelli all'interno di tutti gli spazi dei nomi associati vengono trovati dalla ricerca dipendente dall'argomento. Inoltre, *vengono trovate le funzioni degli amici dello scope dello spazio dei nomi dichiarate nelle classi associate* , che normalmente non sono visibili. L'uso delle direttive viene tuttavia ignorato.

Tutte le seguenti chiamate di esempio sono valide, senza qualificare `f` dal nome dello spazio dei nomi nella chiamata.

```
namespace A {
   struct Z { };
   namespace I { void g(Z); }
   using namespace I;
```

```
   struct X { struct Y { }; friend void f(Y) { } };
   void f(X p) { }
   void f(std::shared_ptr<X> p) { }
}

// example calls
f(A::X());
f(A::X::Y());
f(std::make_shared<A::X>());

g(A::Z()); // invalid: "using namespace I;" is ignored!
```

Leggi Ricerca del nome dipendente dall'argomento online:
https://riptutorial.com/it/cplusplus/topic/5163/ricerca-del-nome-dipendente-dall-argomento

Capitolo 97: Ricorsione in C ++

Examples

Usando la ricorsione della coda e la ricorsione in stile Fibonaci per risolvere la sequenza di Fibonacci

Il modo semplice e più ovvio di usare la ricorsione per ottenere l'ennesimo termine della sequenza di Fibonnaci è questo

```
int get_term_fib(int n)
{
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return get_term_fib(n - 1) + get_term_fib(n - 2);
}
```

Tuttavia, questo algoritmo non è scalabile per termini più elevati: per `n` più grandi, il numero di chiamate di funzione che è necessario fare cresce in modo esponenziale. Questo può essere sostituito con una semplice ricorsione della coda.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
  if (n == 0)
    return prev;
  if (n == 1)
    return curr;
  return get_term_fib(n - 1, curr, prev + curr);
}
```

Ogni chiamata alla funzione calcola immediatamente il termine successivo nella sequenza di Fibonnaci, quindi il numero di chiamate di funzione scala linearmente con `n` .

Ricorsione con memoization

Le funzioni ricorsive possono diventare piuttosto costose. Se sono funzioni pure (funzioni che restituiscono sempre lo stesso valore quando vengono richiamate con gli stessi argomenti e che non dipendono né modificano lo stato esterno), possono essere rese notevolmente più veloci a scapito della memoria memorizzando i valori già calcolati.

Quanto segue è un'implementazione della sequenza di Fibonacci con la memoizzazione:

```
#include <map>

int fibonacci(int n)
{
  static std::map<int, int> values;
```

```
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Si noti che nonostante l'uso della semplice formula di ricorsione, in prima chiamata questa funzione è $ O (n) $. Sulle chiamate successive con lo stesso valore, è ovviamente $ O (1) $.

Si noti tuttavia che questa implementazione non è rientranti. Inoltre, non consente di eliminare i valori memorizzati. Un'implementazione alternativa sarebbe quella di consentire alla mappa di essere passata come argomento aggiuntivo:

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Per questa versione, il chiamante è tenuto a mantenere la mappa con i valori memorizzati. Questo ha il vantaggio che la funzione è ora rientrante e che il chiamante può rimuovere i valori che non sono più necessari, risparmiando memoria. Ha lo svantaggio che rompe l'incapsulamento; il chiamante può modificare l'output popolando la mappa con valori errati.

Leggi Ricorsione in C ++ online: https://riptutorial.com/it/cplusplus/topic/5693/ricorsione-in-c-plusplus

Capitolo 98: Riferimenti

Examples

Definire un riferimento

I riferimenti si comportano allo stesso modo, ma non del tutto come puntatori const. Un riferimento è definito dal suffisso di una `&` commerciale `&` di un nome di tipo.

```
int i = 10;
int &refi = i;
```

Qui, `refi` è un riferimento legato a `i` .
I riferimenti astraggono la semantica dei puntatori, agendo come un alias per l'oggetto sottostante:

```
refi = 20; // i = 20;
```

Puoi anche definire più riferimenti in una singola definizione:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

I riferimenti **devono** essere inizializzati correttamente al momento della definizione e non possono essere modificati in seguito. Il seguente pezzo di codice causa un errore di compilazione:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

Inoltre, non è possibile associare direttamente un riferimento a `nullptr` , diversamente dai puntatori:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a
temporary of type 'nullptr_t'
```

I riferimenti C ++ sono alias di variabili esistenti

Un riferimento in C ++ è solo un `Alias` o un altro nome di una variabile. Proprio come la maggior parte di noi può essere indirizzata usando il nome del passaporto e il nickname.

I riferimenti non esistono letteralmente e non occupano memoria. Se stampiamo l'indirizzo della variabile di riferimento, stamperà lo stesso indirizzo di quello della variabile a cui si riferisce.

```
int main() {
    int i = 10;
    int &j = i;

    cout<<&i<<endl;
    cout<<&b<<endl;
    return 0;
}
```

Nell'esempio sopra, entrambi i `cout` stamperanno lo stesso indirizzo. La situazione sarà la stessa se prendiamo una variabile come riferimento in una funzione

```
void func (int &fParam ) {
   cout<<"Address inside function => "<<fParam<<endl;
}

int main() {
    int i = 10;
    cout<<"Address inside Main => "<<&i<<endl;

    func(i);

    return 0;
}
```

Anche in questo esempio, entrambi i `cout` stamperanno lo stesso indirizzo.

Come sappiamo ormai che le `C++ References` sono solo alias e per la creazione di un alias, dobbiamo avere qualcosa a cui l'Alias può riferirsi.

Questo è il motivo preciso per cui la dichiarazione come questa genera un errore del compilatore

```
int &i;
```

Perché, l'alias non si riferisce a nulla.

Leggi Riferimenti online: https://riptutorial.com/it/cplusplus/topic/1548/riferimenti

Capitolo 99: Risoluzione di sovraccarico

Osservazioni

La risoluzione del sovraccarico si verifica in diverse situazioni

- Chiamate a funzioni con sovraccarico con nome. I candidati sono tutte le funzioni trovate per la ricerca del nome.
- Chiama oggetto di classe. I candidati sono di solito tutti gli operatori di chiamata di funzione sovraccaricati della classe.
- Uso di un operatore. I candidati sono le funzioni dell'operatore sovraccarico nello scope dei nomi, le funzioni dell'operatore sovraccarico nell'oggetto di classe sinistro (se presente) e gli operatori integrati.
- Risoluzione di sovraccarico per trovare la funzione o il costruttore di conversione corretta da richiamare per un'inizializzazione
    - Per l'inizializzazione diretta non di elenco (`Class c(value)`), i candidati sono costruttori di `Class` .
    - Per l'inizializzazione della copia non di elenco (`Class c = value`) e per trovare la funzione di conversione definita dall'utente da richiamare in una sequenza di conversione definita dall'utente. I candidati sono i costruttori di `Class` e se l'origine è un oggetto di classe, le sue funzioni di operatore di conversione.
    - Per l'inizializzazione di una non classe da un oggetto di classe (`Nonclass c = classObject`). I candidati sono le funzioni dell'operatore di conversione dell'oggetto inizializzatore.
    - Per inizializzare un riferimento con un oggetto di classe (`R &r = classObject`), quando la classe ha funzioni di operatore di conversione che producono valori che possono essere associati direttamente a `r` . I candidati sono tali funzioni di operatore di conversione.
    - Per l'inizializzazione di una lista di un oggetto di classe non aggregato (`Class c{1, 2, 3}`), i candidati sono i costruttori dell'elenco di inizializzatori per una prima passata attraverso la risoluzione di sovraccarico. Se questo non trova un candidato valido, viene eseguito un secondo passaggio attraverso la risoluzione di sovraccarico, con i costruttori di `Class` come candidati.

Examples

Corrispondenza esatta

Un sovraccarico senza conversioni necessario per i tipi di parametri o solo le conversioni necessarie tra tipi che sono ancora considerati corrispondenze esatte è preferito su un sovraccarico che richiede altre conversioni per chiamare.

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

Quando un argomento si lega a un riferimento allo stesso tipo, si considera che la corrispondenza non richieda una conversione anche se il riferimento è più qualificato cv.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

Ai fini della risoluzione di sovraccarico, il tipo "array di `T` " viene considerato corrispondente esattamente al tipo "puntatore a `T` " e il tipo di funzione `T` viene considerato corrispondente esattamente al puntatore di funzione di tipo `T*` , anche se entrambi richiedono conversioni.

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Categorizzazione dell'argomento al costo del parametro

La risoluzione di sovraccarico divide il costo del passaggio di un argomento a un parametro in una di quattro diverse categorie, chiamate "sequenze". Ogni sequenza può includere zero, una o più conversioni

- Sequenza di conversione standard

  ```
  void f(int a); f(42);
  ```

- Sequenza di conversione definita dall'utente

  ```
  void f(std::string s); f("hello");
  ```

- Sequenza di conversione di ellissi

  ```
  void f(...); f(42);
  ```

- Elenco sequenza di inizializzazione

  ```
  void f(std::vector<int> v); f({1, 2, 3});
  ```

Il principio generale è che le sequenze di conversione standard sono le più economiche, seguite da sequenze di conversione definite dall'utente, seguite da sequenze di conversione ellissi.

Un caso speciale è la sequenza di inizializzazione dell'elenco, che non costituisce una conversione (una lista di inizializzazione non è un'espressione con un tipo). Il suo costo è determinato definendolo come equivalente a una delle altre tre sequenze di conversione, a seconda del tipo di parametro e della forma dell'elenco di inizializzazione.

Ricerca dei nomi e controllo degli accessi

La risoluzione del sovraccarico si verifica *dopo la* ricerca del nome. Ciò significa che una funzione di corrispondenza migliore non verrà selezionata dalla risoluzione di sovraccarico se perde la ricerca del nome:

```
void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};
```

La risoluzione del sovraccarico si verifica *prima del* controllo dell'accesso. Una funzione inaccessibile potrebbe essere selezionata dalla risoluzione di sovraccarico se è una corrispondenza migliore di una funzione accessibile.

```
class C {
  public:
    static void f(double x);
  private:
    static void f(int x);
};
C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.
```

Allo stesso modo, la risoluzione del sovraccarico si verifica senza verificare se la chiamata risultante è ben formata per quanto riguarda l' `explicit` :

```
struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit
```

Sovraccarico sul riferimento di inoltro

È necessario prestare molta attenzione quando si fornisce un sovraccarico di riferimento di inoltro poiché potrebbe corrispondere troppo bene:

```
struct A {
    A() = default;           // #1
    A(A const& ) = default;  // #2

    template <class T>
```

```
    A(T&& );                 // #3
};
```

L'intento qui era che `A` è copiabile e che abbiamo questo altro costruttore che potrebbe inizializzare qualche altro membro. Però:

```
A a;     // calls #1
A b(a);  // calls #3!
```

Ci sono due partite valide per la chiamata di costruzione:

```
A(A const& ); // #2
A(A& );       // #3, with T = A&
```

Entrambe sono corrispondenze esatte, ma `#3` prende un riferimento a un oggetto con meno *cv* rispetto a `#2` , quindi ha la migliore sequenza di conversione standard ed è la migliore funzione possibile.

La soluzione qui è di limitare sempre questi costruttori (ad es. Usando SFINAE):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
    >
A(T&& );
```

Il carattere type qui è quello di escludere qualsiasi `A` o classe derivata pubblicamente e senza ambiguità da `A` , che renderebbe questo costruttore malformato nell'esempio descritto in precedenza (e quindi rimosso dal set di overload). Di conseguenza, viene invocato il costruttore di copia, che è ciò che volevamo.

Passi per la risoluzione del sovraccarico

I passaggi della risoluzione di sovraccarico sono:

1. Trova le funzioni candidate tramite la ricerca del nome. Le chiamate non qualificate eseguiranno sia una ricerca regolare non regolare che una ricerca dipendente dall'argomento (se applicabile).

2. Filtra l'insieme di funzioni candidate in un insieme di funzioni *vitali* . Una funzione valida per la quale esiste una sequenza di conversione implicita tra gli argomenti con cui la funzione viene chiamata e i parametri che la funzione assume.

   ```
   void f(char);          // (1)
   void f(int ) = delete; // (2)
   void f();              // (3)
   void f(int& );         // (4)

   f(4); // 1,2 are viable (even though 2 is deleted!)
         // 3 is not viable because the argument lists don't match
         // 4 is not viable because we cannot bind a temporary to
   ```

```
      //     a non-const lvalue reference
```

3. Scegli il miglior candidato possibile. Una funzione valida `F1` è una funzione migliore di un'altra funzione valida `F2` se la sequenza di conversione implicita per ciascun argomento in `F1` non è peggiore della corrispondente sequenza di conversione implicita in `F2` , e ...:

   3.1. Per alcuni argomenti, la sequenza di conversione implicita per quell'argomento in `F1` è una sequenza di conversione migliore rispetto a quell'argomento in `F2` , o

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

   3.2. In una conversione definita dall'utente, la sequenza di conversione standard dal ritorno di `F1` al tipo di destinazione è una sequenza di conversione migliore rispetto a quella del tipo di ritorno di `F2` , o

```
struct A
{
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

   3.3. In un legame di riferimento diretto, `F1` ha lo stesso tipo di riferimento di `F2` non lo è, o

```
struct A
{
    operator X&();  // #1
    operator X&&(); // #2
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

   3.4. `F1` non è una specializzazione del modello di funzione, ma `F2` è, o

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

   3.5. `F1` e `F2` sono entrambe specializzazioni del modello di funzione, ma `F1` è più specializzato di `F2` .

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
```

```
f(p); // calls #2, more specialized
```

L'ordine qui è significativo. Il controllo della sequenza di conversione migliore avviene prima del controllo modello vs non modello. Ciò porta a un errore comune con sovraccarico sul riferimento di inoltro:

```
struct A {
    A(A const& ); // #1

    template <class T>
    A(T&& );      // #2, not constrained
};

A a;
A b(a); // calls #2!
        // #1 is not a template but #2 resolves to
        // A(A& ), which is a less cv-qualified reference than #1
        // which makes it a better implicit conversion sequence
```

Se alla fine non c'è un singolo candidato valido, la chiamata è ambigua:

```
void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous
```

Promozioni aritmetiche e conversioni

Convertire un tipo intero nel tipo promosso corrispondente è meglio che convertirlo in un altro tipo intero.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Promuovere un `float` da `double` è meglio che convertirlo in un altro tipo di floating point.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Le conversioni aritmetiche diverse dalle promozioni non sono né migliori né peggiori delle altre.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous

void g(long x);
```

```
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Pertanto, al fine di garantire che non ci sia ambiguità quando si chiama una funzione `f` con argomenti interi o in virgola mobile di qualsiasi tipo standard, sono necessari un totale di otto overload, in modo che per ogni tipo di argomento possibile sia una corrispondenza di sovraccarico esattamente o verrà selezionato l'overload univoco con il tipo di argomento promosso.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Sovraccarico all'interno di una gerarchia di classi

I seguenti esempi useranno questa gerarchia di classi:

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

La conversione dal tipo di classe derivata al tipo di classe base è preferibile alle conversioni definite dall'utente. Questo si applica quando si passa per valore o per riferimento, così come quando si converte il puntatore-derivato in puntatore-base.

```
struct Unrelated {
    Unrelated(B b);
};
void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

Anche una conversione puntatore dalla classe derivata alla classe base è migliore della conversione in `void*` .

```
void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)
```

Se ci sono sovraccarichi multipli all'interno della stessa catena di ereditarietà, viene preferito l'overload di classe base più derivato. Questo si basa su un principio simile a quello di dispacciamento virtuale: viene scelta l'implementazione "più specializzata". Tuttavia, la risoluzione del sovraccarico si verifica sempre al momento della compilazione e non verrà mai implicitamente abbassata.

```
void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable
```

Per i riferimenti ai membri, che sono controvarianti rispetto alla classe, una regola simile si applica nella direzione opposta: la classe derivata meno derivata è preferita.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Sovraccarico di costanza e volatilità

Passare un argomento puntatore a un parametro `T*` , se possibile, è meglio che passarlo a un parametro `const T*` .

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
       // constness is only a "tie-breaker" rule
```

Allo stesso modo, passare un argomento a un parametro `T&` , se possibile, è meglio che passarlo a un parametro `const T&` , anche se entrambi hanno una corrispondenza esatta.

```
void f(int& r);
void f(const int& r);
int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable
```

Questa regola si applica anche alle funzioni membro const-qualificato, dove è importante consentire l'accesso mutevole agli oggetti non-const e l'accesso immutabile agli oggetti const.

```
class IntVector {
  public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
  private:
    // ...
```

```
    int* m_data;
};
IntVector v1;
int* data1 = v1.data();       // Vector::data() is better than Vector::data() const;
                              // data1 can be used to modify the vector's data
const IntVector v2;
const int* data2 = v2.data(); // only Vector::data() const is viable;
                              // data2 can't be used to modify the vector's data
```

Allo stesso modo, un sovraccarico volatile sarà meno preferito rispetto a un sovraccarico non volatile.

```
class AtomicInt {
  public:
    // ...
    int load();
    int load() volatile;
  private:
    // ...
};
AtomicInt a1;
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1
```

Leggi Risoluzione di sovraccarico online: https://riptutorial.com/it/cplusplus/topic/2021/risoluzione-di-sovraccarico

Capitolo 100: RTTI: informazioni di tipo run-time

Examples

Nome di un tipo

È possibile recuperare il nome definito di implementazione di un tipo in runtime utilizzando la funzione membro `.name()` dell'oggetto `std::type_info` restituito da `typeid` .

```
#include <iostream>
#include <typeinfo>

int main()
{
    int speed = 110;

    std::cout << typeid(speed).name() << '\n';
}
```

Output (definito dall'implementazione):

```
int
```

dynamic_cast

Usa `dynamic_cast<>()` come una funzione, che ti aiuta a buttare giù attraverso una gerarchia di ereditarietà (descrizione principale).

Se devi eseguire un lavoro non polimorfico su alcune classi derivate `B` e `C` , ma hai ricevuto la `class A` base `class A` , scrivi in questo modo:

```
class A { public: virtual ~A(){} };

class B: public A
{ public: void work4B(){} };

class C: public A
{ public: void work4C(){} };

void non_polymorphic_work(A* ap)
{
  if (B* bp =dynamic_cast<B*>(ap))
    bp->work4B();
  if (C* cp =dynamic_cast<C*>(ap))
    cp->work4C();
}
```

La parola chiave typeid

La parola chiave `typeid` è un operatore unario che fornisce informazioni sul tipo di esecuzione sul suo operando se il tipo dell'operando è un tipo di classe polimorfico. Restituisce un lvalue di tipo `const std::type_info` . La qualifica di cv di alto livello viene ignorata.

```
struct Base {
    virtual ~Base() = default;
};
struct Derived : Base {};
Base* b = new Derived;
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` può anche essere applicato direttamente a un tipo. In questo caso, i primi riferimenti di livello superiore vengono rimossi, quindi la qualifica di cv di livello superiore viene ignorata. Pertanto, l'esempio precedente potrebbe essere stato scritto con `typeid(Derived)` invece di `typeid(Derived{})` :

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

Se `typeid` viene applicato a qualsiasi espressione che *non sia* di tipo di classe polimorfico, l'operando non viene valutato e il tipo di informazioni restituito è per il tipo statico.

```
struct Base {
    // note: no virtual destructor
};
struct Derived : Base {};
Derived d;
Base& b = d;
assert(typeid(b) == typeid(Base)); // not Derived
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Quando usare quale cast in c ++

Utilizza **dynamic_cast** per convertire i puntatori / riferimenti all'interno di una gerarchia di ereditarietà.

Usa **static_cast** per le conversioni di tipo ordinario.

Usa **reinterpret_cast** per la **reinterpretazione** a basso livello dei pattern di bit. Usare con estrema cautela.

Usa **const_cast** per lanciare via const / volatile. Evita questo se non sei bloccato usando un'API const-errata.

Leggi RTTI: informazioni di tipo run-time online: https://riptutorial.com/it/cplusplus/topic/3129/rtti--informazioni-di-tipo-run-time

Capitolo 101: Scopes

Examples

Semplice ambito di blocco

L'ambito di una variabile in un blocco `{ ... }` , inizia dopo la dichiarazione e termina alla fine del blocco. Se esiste un blocco nidificato, il blocco interno può nascondere l'ambito di una variabile dichiarata nel blocco esterno.

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
}   // <- Scope of `x` ends here
```

Se un blocco nidificato inizia all'interno di un blocco esterno, una nuova variabile dichiarata con lo stesso nome che è prima nella classe esterna, nasconde la prima.

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x;  // <- Output is 200
    }

    std::cout << x;  // <- Output is 100
}
```

Variabili globali

Per dichiarare una singola istanza di una variabile accessibile in diversi file sorgente, è possibile renderla nell'ambito globale con la parola chiave `extern` . Questa parola chiave dice al compilatore che da qualche parte nel codice esiste una definizione per questa variabile, quindi può essere utilizzata ovunque e tutte le operazioni di scrittura / lettura verranno eseguite in un unico luogo di memoria.

```
// File my_globals.h:

#ifndef __MY_GLOBALS_H__
#define __MY_GLOBALS_H__

extern int circle_radius; // Promise to the compiler that circle_radius
                          // will be defined somewhere

#endif
```

```
// File foo1.cpp:

#include "my_globals.h"

int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:

#include "my_globals.h"
#include <iostream>

int main()
{
    std::cout << "The radius is: " << circle_radius << "\n";'
    return 0;
}
```

Produzione:

```
The radius is: 123
```

Leggi Scopes online: https://riptutorial.com/it/cplusplus/topic/3453/scopes

Capitolo 102: Semaforo

introduzione

I semafori non sono disponibili in C ++ fin d'ora, ma possono essere facilmente implementati con un mutex e una variabile di condizione.

Questo esempio è stato preso da:

C ++ 0x non ha semafori? Come sincronizzare i thread?

Examples

Semaphore C ++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
    Semaphore (int count_ = 0)
    : count(count_)
    {
    }

    inline void notify( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        cout << "thread " << tid <<  " notify" << endl;
        //notify the waiting thread
        cv.notify_one();
    }
    inline void wait( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        while(count == 0) {
            cout << "thread " << tid << " wait" << endl;
            //wait on the mutex until notify is called
            cv.wait(lock);
            cout << "thread " << tid << " run" << endl;
        }
        count--;
    }
private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Classe di semaforo in azione

La seguente funzione aggiunge quattro thread. Tre thread competono per il semaforo, che è impostato su un conteggio di uno. Un thread più lento chiama `notify_one()` , consentendo a uno

dei thread in attesa di procedere.

Il risultato è che `s1` inizia immediatamente a girare, facendo sì che il `count` utilizzo del semaforo rimanga inferiore a 1. Gli altri thread attendono a turno la variabile di condizione finché non viene chiamato notify ().

```
int main()
{
    Semaphore sem(1);

    thread s1([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.wait( 1 );
        }
    });
    thread s2([&]() {
        while(true){
            sem.wait( 2 );
        }
    });
    thread s3([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::milliseconds(600));
            sem.wait( 3 );
        }
    });
    thread s4([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.notify( 4 );
        }
    });


    s1.join();
    s2.join();
    s3.join();
    s4.join();

    ...
    }
```

Leggi Semaforo online: https://riptutorial.com/it/cplusplus/topic/9785/semaforo

Capitolo 103: Semantica del valore e di riferimento

Examples

Copia e supporto per muovere in profondità

Se un tipo desidera avere una semantica del valore e deve archiviare oggetti che vengono allocati dinamicamente, quindi in operazioni di copia, il tipo dovrà allocare nuove copie di tali oggetti. Deve anche farlo per l'assegnazione delle copie.

Questo tipo di copia è chiamato "copia profonda". Prende efficacemente quella che altrimenti sarebbe stata semantica di riferimento e la trasforma in semantica del valore:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }
};
```

C ++ 11

Spostare la semantica consente un tipo di `Value` per evitare di copiare veramente i suoi dati di riferimento. Se l'utente usa il valore in un modo che provoca uno spostamento, il "copiato" dall'oggetto può essere lasciato vuoto dei dati a cui fa riferimento:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
```

```
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

In effetti, possiamo persino rendere questo tipo non copiabili, se vogliamo proibire le copie profonde mentre ancora permettiamo che l'oggetto sia spostato.

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) = delete;
  Value &operator=(const Value &val) = delete;
```

```
  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

Possiamo anche applicare la regola dello zero, attraverso l'uso di `unique_ptr` :

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  unique_ptr<Inner []>array_; //Move-only type.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //No need to explicitly delete. Or even declare.
  ~Value() = default; {delete[] array_;}

  //No need to explicitly delete. Or even declare.
  Value(const Value &val) = default;
  Value &operator=(const Value &val) = default;

  //Will perform an element-wise move.
  Value(Value &&val) noexcept = default;

  //Will perform an element-wise move.
  Value &operator=(Value &&val) noexcept = default;
};
```

definizioni

Un tipo ha semantica del valore se lo stato osservabile dell'oggetto è funzionalmente distinto da tutti gli altri oggetti di quel tipo. Ciò significa che se copi un oggetto, hai un nuovo oggetto e le modifiche del nuovo oggetto non saranno in alcun modo visibili dal vecchio oggetto.

I tipi C ++ di base hanno semantica del valore:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

La maggior parte dei tipi definiti di libreria standard ha anche la semantica del valore:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

Si dice che un tipo abbia una semantica di riferimento se un'istanza di quel tipo può condividere il suo stato osservabile con un altro oggetto (esterno ad esso), in modo tale che la manipolazione di un oggetto causerà lo stato di cambiare all'interno di un altro oggetto.

I puntatori C ++ hanno semantica del valore rispetto a quale oggetto puntano, ma hanno una semantica di riferimento rispetto allo *stato* dell'oggetto a cui puntano:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

Anche i riferimenti C ++ hanno una semantica di riferimento.

Leggi Semantica del valore e di riferimento online:
https://riptutorial.com/it/cplusplus/topic/1955/semantica-del-valore-e-di-riferimento

Capitolo 104: Separatori di cifre

Examples

Separatore di cifre

I valori letterali numerici di più di poche cifre sono difficili da leggere.

- Pronuncia 7237498123.
- Confronta 237498123 con 237499123 per l'uguaglianza.
- Decidi se 237499123 o 20249472 è più grande.

`C++14` definisce Simple Quotation Mark `'` come separatore di cifre, in numeri e letterali definiti dall'utente. Questo può rendere più facile per i lettori umani l'analisi di grandi numeri.

C ++ 14

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'001111ll;
```

Le virgolette singole vengono ignorate quando si determina il suo valore.

Esempio:

- I valori letterali `1048576` , `1'048'576` , `0X100000` , `0x10'0000` e `0'004'000'000` hanno tutti lo stesso valore.
- I letterali `1.602'176'565e-19` e `1.602176565e-19` hanno lo stesso valore.

La posizione delle virgolette singole è irrilevante. Tutti i seguenti sono equivalenti:

C ++ 14

```
long long a1 = 12345678911;
long long a2 = 123'456'78911;
long long a3 = 12'34'56'78'911;
long long a4 = 12345'678911;
```

È anche consentito in valori letterali `user-defined` :

C ++ 14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Leggi Separatori di cifre online: https://riptutorial.com/it/cplusplus/topic/10595/separatori-di-cifre

Capitolo 105: SFINAE (Errore di sostituzione non è un errore)

Examples

enable_if

`std::enable_if` è una comoda utilità per utilizzare le condizioni booleane per attivare SFINAE. È definito come:

```
template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};
```

Cioè, `enable_if<true, R>::type` è un alias per `R` , mentre `enable_if<false, T>::type` è mal formato poiché quella specializzazione di `enable_if` non ha un `type` membro di tipo.

`std::enable_if` può essere usato per vincolare i template:

```
int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }
```

Qui, una chiamata a `negate(1)` fallirebbe a causa dell'ambiguità. Ma il secondo sovraccarico non è destinato a essere utilizzato per i tipi interi, quindi possiamo aggiungere:

```
int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }
```

Ora, l'istanziazione di `negate<int>` comporterebbe un errore di sostituzione poiché `!std::is_arithmetic<int>::value` è `false` . A causa di SFINAE, questo non è un errore grave, questo candidato viene semplicemente rimosso dal set di sovraccarico. Di conseguenza, `negate(1)` ha un solo candidato valido, che viene poi chiamato.

Quando usarlo

Vale la pena ricordare che `std::enable_if` è un aiuto *su* SFINAE, ma non è ciò che fa funzionare SFINAE in primo luogo. Consideriamo queste due alternative per implementare funzionalità simili a `std::size` , ovvero una `size(arg)` set di overload `size(arg)` che produce la dimensione di un

contenitore o array:

```
// for containers
template<typename Cont>
auto size1(Cont const& cont) -> decltype( cont.size() );

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);
```

Supponendo che `is_sizeable` sia scritto appropriatamente, queste due dichiarazioni dovrebbero essere esattamente equivalenti rispetto a SFINAE. Qual è il modo più semplice per scrivere, e quale è il modo più semplice da rivedere e capire a colpo d'occhio?

Consideriamo ora come potremmo voler implementare help aritmetici che evitino l'overflow dei caratteri interi in favore di un comportamento wrap-around o modulare. Il che `incr(i, 3)` ad es. `incr(i, 3)` sarebbe lo stesso di `i += 3` salvo il fatto che il risultato sarebbe sempre definito anche se `i` sono un `int` con valore `INT_MAX` . Queste sono due alternative possibili:

```
// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);
```

Ancora una volta qual è il modo più semplice per scrivere, e quale è il modo più semplice da rivedere e capire a colpo d'occhio?

Un punto di forza di `std::enable_if` è il modo in cui gioca con il refactoring e la progettazione dell'API. Se `is_sizeable<Cont>::value` è pensato per riflettere se `cont.size()` è valido, basta usare l'espressione così come appare per `size1` può essere più conciso, anche se ciò potrebbe dipendere dal fatto che `is_sizeable` possa essere usato in più posti o meno . Contrasto a quello

con `std::is_signed` che riflette la sua intenzione molto più chiaramente di quando la sua implementazione `incr1` nella dichiarazione di `incr1` .

void_t

C ++ 11

`void_t` è una meta-funzione che mappa qualsiasi (numero di) tipi di tipo `void` . Lo scopo principale di `void_t` è facilitare la scrittura di caratteri tipografici.

`std::void_t` sarà parte di C ++ 17, ma fino ad allora, è estremamente semplice implementare:

```
template <class...> using void_t = void;
```

Alcuni compilatori richiedono un'implementazione leggermente diversa:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

L'applicazione principale di `void_t` è la scrittura di tratti di tipo che controllano la validità di una dichiarazione. Ad esempio, controlliamo se un tipo ha una funzione membro `foo()` che non accetta argomenti:

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

Come funziona? Quando provo a istanziare `has_foo<T>::value` , ciò farà sì che il compilatore cerchi di cercare la migliore specializzazione per `has_foo<T, void>` . Abbiamo due opzioni: la primaria, e questa secondaria che implica dover istanziare quella espressione sottostante:

- Se `T` *ha* una funzione membro `foo()` , allora qualsiasi tipo che restituisce viene convertito in `void` , e la specializzazione è preferito al primario basati sul modello regole ordinamento parziale. Quindi `has_foo<T>::value` sarà `true`
- Se `T` *non* ha una funzione membro (o richiede più di un argomento), la sostituzione non riesce per la specializzazione e abbiamo solo il modello principale su cui fare il fallback. Quindi, `has_foo<T>::value` è `false` .

Un caso più semplice:

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T&>> : std::true_type {};
```

questo non usa `std::declval` o `decltype` .

Si può notare un modello comune di un argomento vuoto. Possiamo tenerne conto:

```
struct details {
  template<template<class...>class Z, class=void, class...Ts>
  struct can_apply:
    std::false_type
  {};
  template<template<class...>class Z, class...Ts>
  struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...>:
    std::true_type
  {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

che nasconde l'uso di `std::void_t` e fa `can_apply` come un indicatore se il tipo fornito come primo argomento template è ben formato dopo aver sostituito gli altri tipi in esso. Gli esempi precedenti possono ora essere riscritti usando `can_apply` come:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>;    // Is T& well formed for T?
```

e:

```
template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply< dot_foo_r, T >;    // Is T.foo() well formed for T?
```

che sembra più semplice delle versioni originali.

Ci sono post-C ++ 17 proposte di `std` tratti simili a `can_apply` .

L'utilità di `void_t` stata scoperta da Walter Brown. Ha tenuto una meravigliosa presentazione al CppCon 2016.

finale decltype nei modelli di funzione

C ++ 11

Una delle funzioni di limitazione consiste nell'utilizzare il `decltype` finale per specificare il tipo restituito:

```
namespace details {
   using std::to_string;
```

```
    // this one is constrained on being able to call to_string(T)
    template <class T>
    auto convert_to_string(T const& val, int )
        -> decltype(to_string(val))
    {
        return to_string(val);
    }

    // this one is unconstrained, but less preferred due to the ellipsis argument
    template <class T>
    std::string convert_to_string(T const& val, ... )
    {
        std::ostringstream oss;
        oss << val;
        return oss.str();
    }
}

template <class T>
std::string convert_to_string(T const& val)
{
    return details::convert_to_string(val, 0);
}
```

Se chiamo `convert_to_string()` con un argomento con cui posso invocare `to_string()` , allora ho due funzioni valide per `details::convert_to_string()` . Il primo è preferito poiché la conversione da `0` a `int` è una sequenza di conversione implicita migliore rispetto alla conversione da `0` a `...`

Se chiamo `convert_to_string()` con un argomento dal quale non posso richiamare `to_string()` , allora la prima istanza del modello di funzione porta a un errore di sostituzione (non c'è `decltype(to_string(val))`). Di conseguenza, quel candidato viene rimosso dal set di sovraccarico. Il secondo modello di funzione non è vincolato, quindi è selezionato e passiamo invece attraverso l' `operator<<(std::ostream&, T)` . Se quello non è definito, allora abbiamo un errore di compilazione difficile con uno stack di template sulla linea `oss << val` .

Cos'è SFINAE

SFINAE sta per **S** ubstitution **F** ailure **I** s **N** OT **A** n **e** rror. Il codice malformato che deriva dal sostituire i tipi (o i valori) per creare un'istanza di un modello di funzione o un modello di classe **non** è un errore di compilazione difficile, viene considerato solo come un errore di deduzione.

I fallimenti di detrazioni su modelli di istanze di istanza o specializzazioni di modelli di classe rimuovono quel candidato dall'insieme di considerazioni - come se quel candidato fallito non esistesse per cominciare.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // OK. The first function template substitution fails because
             // vals.begin() is ill-formed. This is not an error! That function
```

```
            // is just removed from consideration as a viable overload candidate,
            // leaving us with the array overload.
```

Solo i guasti di sostituzione nel **contesto immediato** sono considerati errori di deduzione, tutti gli altri sono considerati errori gravi.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

enable_if_all / enable_if_any

C ++ 11

Esempio motivazionale

Quando si dispone di un pacchetto modello variadic nell'elenco dei parametri del modello, come nel seguente frammento di codice:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

La libreria standard (precedente a C ++ 17) non offre alcun modo diretto per scrivere **enable_if** per imporre vincoli SFINAE su ***tutti i parametri*** in `Args` o in ***qualsiasi parametro*** in `Args` . C ++ 17 offre `std::conjunction` e `std::disjunction` che risolvono questo problema. Per esempio:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };

/// C++17: SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };
```

Se non hai a disposizione C ++ 17, ci sono diverse soluzioni per raggiungerli. Uno di questi è usare una classe di casi base e **specializzazioni parziali** , come dimostrato nelle risposte a questa domanda .

In alternativa, si può anche implementare a mano il comportamento di `std::conjunction` e `std::disjunction` in modo piuttosto diretto. Nell'esempio seguente `std::enable_if` le implementazioni e le combineremo con `std::enable_if` per produrre due alias: `enable_if_all` e `enable_if_any` , che fanno esattamente ciò che dovrebbero semanticamente. Ciò potrebbe fornire una soluzione più scalabile.

Implementazione di `enable_if_all` e `enable_if_any`

Prima `seq_and` `std::conjunction` e `std::disjunction` usando rispettivamente `seq_and` e `seq_or` personalizzati:

```
/// Helper for prior to C++14.
template<bool B, class T, class F >
using conditional_t = typename std::conditional<B,T,F>::type;

/// Emulate C++17 std::conjunction.
template<bool...> struct seq_or: std::false_type {};
template<bool...> struct seq_and: std::true_type {};

template<bool B1, bool... Bs>
struct seq_or<B1,Bs...>:
  conditional_t<B1,std::true_type,seq_or<Bs...>> {};

template<bool B1, bool... Bs>
struct seq_and<B1,Bs...>:
  conditional_t<B1,seq_and<Bs...>,std::false_type> {};
```

Quindi l'implementazione è abbastanza semplice:

```
template<bool... Bs>
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;

template<bool... Bs>
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

Alla fine alcuni aiutanti:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

uso

Anche l'utilizzo è semplice:

```
    /// SFINAE constraints on all of the parameters in Args.
    template<typename ...Args,
             enable_if_all_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };

    /// SFINAE constraints on any of the parameters in Args.
    template<typename ...Args,
             enable_if_any_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };
```

is_detected

Per generalizzare la creazione di type_trait: basato su SFINAE ci sono tratti sperimentali `detected_or` , `detected_t` , `is_detected` .

Con i parametri del template `typename Default` , `template <typename...> Op` e `typename ... Args` :

- `is_detected` : alias di `std::true_type` o `std::false_type` seconda della validità di `Op<Args...>`
- `detected_t` : alias di `Op<Args...>` o `nonesuch` seconda della validità di `Op<Args...>` .
- `detected_or` : alias di una struct con `value_t` che è `is_detected` e `type` che è `Op<Args...>` o `Default` dipendente dalla validità di `Op<Args...>`

che può essere implementato usando `std::void_t` per SFINAE come segue:

C ++ 17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };

} // namespace detail

// special type to indicate detection failure
struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

I tratti per rilevare la presenza del metodo possono quindi essere implementati semplicemente:

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));
```

```
struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
              "Unexpected");

static_assert(std::is_same<void, // Default
                           detected_or<void, foo_type, C1, char>>::value,
              "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
              "Unexpected");
```

Risoluzione di sovraccarico con un gran numero di opzioni

Se è necessario selezionare tra diverse opzioni, abilitare solo uno tramite `enable_if<>` può essere piuttosto ingombrante, poiché anche diverse condizioni devono essere negate.

L'ordine tra sovraccarichi può invece essere selezionato usando l'ereditarietà, cioè l'invio di tag.

Invece di testare la cosa che deve essere ben formata, e anche testare la negazione di tutte le altre condizioni delle versioni, `decltype` invece solo ciò di cui abbiamo bisogno, preferibilmente in un `decltype` in un ritorno finale.
Ciò potrebbe lasciare diverse opzioni ben formate, differenziamo tra quelle che usano i 'tag', simili ai tag iterator-trait (`random_access_tag` et al). Funziona perché una corrispondenza diretta è migliore di una classe base, che è migliore di una classe base di una classe base, ecc.

```
#include <algorithm>
#include <iterator>

namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }
```

```
    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }

}

// this is the function the user calls. it will dispatch the call
// to the correct implementation with the help of 'tags'.
template<typename T>
void stable_sort(T& t)
{
    // use an N that is higher that any used above.
    // this will pick the highest overload that is well formed.
    detail::stable_sort(t, detail::pick<10>{});
}
```

Esistono altri metodi comunemente usati per distinguere tra sovraccarichi, ad esempio la corrispondenza esatta è migliore della conversione, essendo migliore dei puntini di sospensione.

Tuttavia, il tag-dispatch può estendersi a qualsiasi numero di scelte ed è un po 'più chiaro nell'intento.

Leggi SFINAE (Errore di sostituzione non è un errore) online:
https://riptutorial.com/it/cplusplus/topic/1169/sfinae--errore-di-sostituzione-non-e-un-errore-

Capitolo 106: sindacati

Osservazioni

I sindacati sono strumenti molto utili, ma sono forniti con alcune avvertenze importanti:

- È un comportamento indefinito, secondo lo standard C ++, accedere ad un elemento di un sindacato che non era il membro modificato più recentemente. Sebbene molti compilatori C ++ consentano questo accesso in modi ben definiti, si tratta di estensioni e non possono essere garantite tra i compilatori.

  Una `std::variant` (dal C ++ 17) è come un'unione, solo che ti dice cosa contiene attualmente (parte del suo stato visibile è il tipo del valore che detiene in un dato momento: impone l'accesso al valore che accade solo a quel tipo).

- Le implementazioni non necessariamente allineano membri di dimensioni diverse allo stesso indirizzo.

Examples

Caratteristiche di base dell'Unione

I sindacati sono una struttura specializzata all'interno della quale tutti i membri occupano memoria sovrapposta.

```
union U {
    int a;
    short b;
    float c;
};
U u;

//Address of a and b will be equal
(void*)&u.a == (void*)&u.b;
(void*)&u.a == (void*)&u.c;

//Assigning to any union member changes the shared memory of all members
u.c = 4.f;
u.a = 5;
u.c != 4.f;
```

Uso tipico

I sindacati sono utili per ridurre al minimo l'utilizzo della memoria per dati esclusivi, ad esempio quando si implementano tipi di dati misti.

```
struct AnyType {
    enum {
        IS_INT,
```

```
        IS_FLOAT
    } type;

    union Data {
        int as_int;
        float as_float;
    } value;

    AnyType(int i) : type(IS_INT) { value.as_int = i; }
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }

    int get_int() const {
        if(type == IS_INT)
            return value.as_int;
        else
            return (int)value.as_float;
    }

    float get_float() const {
        if(type == IS_FLOAT)
            return value.as_float;
        else
            return (float)value.as_int;
    }
};
```

Comportamento indefinito

```
union U {
    int a;
    short b;
    float c;
};
U u;

u.a = 10;
if (u.b == 10) {
   // this is undefined behavior since 'a' was the last member to be
   // written to. A lot of compilers will allow this and might issue a
   // warning, but the result will be "as expected"; this is a compiler
   // extension and cannot be guaranteed across compilers (i.e. this is
   // not compliant/portable code).
}
```

Leggi sindacati online: https://riptutorial.com/it/cplusplus/topic/2678/sindacati

Capitolo 107: Singleton Design Pattern

Osservazioni

Un **Singleton** è progettato per garantire che una classe abbia solo un'istanza *e* fornisca un accesso globale ad essa. Se si richiede solo un'istanza *o* un comodo punto di accesso globale, ma non entrambi, prendere in considerazione altre opzioni prima di passare al singleton.

Le variabili globali *possono* rendere più difficile ragionare sul codice. Ad esempio, se una delle funzioni di chiamata non è felice con i dati ricevuti da un Singleton, è necessario rintracciare ciò che inizialmente fornisce i dati non validi di Singleton.

I singleton incoraggiano anche l' accoppiamento , un termine usato per descrivere due componenti del codice che sono uniti, riducendo così ogni misura di autocontenimento di ogni componente.

I single non sono compatibili con la concorrenza. Quando una classe ha un punto di accesso globale, ogni thread ha la possibilità di accedervi che può portare a deadlock e condizioni di gara.

Infine, l'inizializzazione pigra può causare problemi di prestazioni se inizializzata nel momento sbagliato. La rimozione dell'inizializzazione pigra rimuove anche alcune delle caratteristiche che rendono Singleton interessante in primo luogo, come il polimorfismo (vedi Sottoclassi).

Fonti: Pattern di programmazione di gioco di Robert Nystrom

Examples

Inizializzazione pigra

Questo esempio è stato rimosso dalla sezione `Q & A` risposte qui:
http://stackoverflow.com/a/1008289/3807729

Vedi questo articolo per un design semplice per un pigro valutato con distruzione garantita singleton:
Qualcuno può fornirmi un campione di Singleton in c ++?

Il classico pigro valutato e correttamente distrutto singleton.

```
class S
{
    public:
        static S& getInstance()
        {
            static S    instance; // Guaranteed to be destroyed.
                                  // Instantiated on first use.
            return instance;
        }
    private:
```

```
        S() {};                    // Constructor? (the {} brackets) are needed here.

        // C++ 03
        // ========
        // Dont forget to declare these two. You want to make sure they
        // are unacceptable otherwise you may accidentally get copies of
        // your singleton appearing.
        S(S const&);               // Don't Implement
        void operator=(S const&); // Don't implement

        // C++ 11
        // =======
        // We can use the better technique of deleting the methods
        // we don't want.
    public:
        S(S const&)               = delete;
        void operator=(S const&)  = delete;

        // Note: Scott Meyers mentions in his Effective Modern
        //       C++ book, that deleted functions should generally
        //       be public as it results in better error messages
        //       due to the compilers behavior to check accessibility
        //       before deleted status
};
```

Vedi questo articolo su quando usare un singleton: (non spesso)
Singleton: come dovrebbe essere usato

Vedi questo articolo sull'ordine di inizializzazione e su come far fronte:
Ordine di inizializzazione delle variabili statiche
Ricerca di problemi di ordine di inizializzazione statici C ++

Vedi questo articolo che descrive le vite:
Qual è la durata di una variabile statica in una funzione C ++?

Vedi questo articolo che discute alcune implicazioni di threading ai singleton:
Istanza Singleton dichiarata come variabile statica del metodo GetInstance

Vedi questo articolo che spiega perché il double check locking non funzionerà su C ++:
Quali sono tutti i comuni comportamenti indefiniti di cui un programmatore C ++ dovrebbe essere a conoscenza?

sottoclassi

```
class API
{
public:
    static API& instance();

    virtual ~API() {}

    virtual const char* func1() = 0;
    virtual void func2() = 0;

protected:
```

```
    API() {}
    API(const API&) = delete;
    API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1()  override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM = LINUX
    static LinuxAPI instance;
#endif
    return instance;
}
```

In questo esempio, un semplice switch del compilatore associa la classe `API` alla sottoclasse appropriata. In questo modo, è possibile accedere `API` senza essere accoppiati al codice specifico della piattaforma.

Singeton sicuro per thread

C ++ 11

Gli standard C ++ 11 garantiscono che l'inizializzazione degli oggetti dell'ambito della funzione sia inizializzata in modo sincronizzato. Questo può essere usato per implementare un singleton thread-safe con inizializzazione pigra .

```
class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};
```

Deinitializzazione statica: sicuro singleton.

Ci sono volte con più oggetti statici in cui è necessario essere in grado di garantire che il *singleton* non verrà distrutto finché tutti gli oggetti statici che usano il *singleton* non ne hanno più bisogno.

In questo caso, `std::shared_ptr` può essere usato per mantenere vivo il *singleton* per tutti gli utenti anche quando i distruttori statici vengono chiamati alla fine del programma:

```
class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};
```

NOTA: questo esempio viene visualizzato come risposta nella sezione Domande e risposte qui.

Leggi Singleton Design Pattern online: https://riptutorial.com/it/cplusplus/topic/2713/singleton-design-pattern

Capitolo 108: Sovraccarico del modello di funzione

Osservazioni

- Una funzione normale non è mai correlata a un modello di funzione, nonostante lo stesso nome, lo stesso tipo.
- Una chiamata di funzione normale e una chiamata di modello di funzione generata sono diverse anche se condividono lo stesso nome, lo stesso tipo di ritorno e lo stesso elenco di argomenti

Examples

Cos'è un sovraccarico del modello di funzione valido?

Un modello di funzione può essere sovraccaricato in base alle regole per l'overloading delle funzioni non modello (stesso nome, ma diversi tipi di parametri) e in aggiunta a ciò, il sovraccarico è valido se

- Il tipo di reso è diverso, o
- L'elenco dei parametri del modello è diverso, ad eccezione della denominazione dei parametri e della presenza di argomenti predefiniti (non fanno parte della firma)

Per una funzione normale, il confronto di due tipi di parametri è facile per il compilatore, poiché ha tutte le informazioni. Ma un tipo all'interno di un modello potrebbe non essere ancora determinato. Pertanto, la regola per quando due tipi di parametri sono uguali è approssimativo e afferma che i tipi e i valori non dipendenti dalla dipendenza devono corrispondere e che l'ortografia dei tipi e delle espressioni dipendenti deve essere la stessa (più precisamente, devono essere conformi al le cosiddette regole ODR), con la differenza che i parametri del modello possono essere rinominati. Tuttavia, se con tali ortografie diverse, due valori all'interno dei tipi sono considerati diversi, ma verranno sempre istanziati agli stessi valori, l'overloading non è valido, ma non è richiesta alcuna diagnostica dal compilatore.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

Questo è un sovraccarico valido, poiché "T" e "T *" sono ortografie diverse. Ma quanto segue non è valido, senza necessità di diagnostica

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }
```

```
template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Leggi Sovraccarico del modello di funzione online:
https://riptutorial.com/it/cplusplus/topic/4164/sovraccarico-del-modello-di-funzione

Capitolo 109: Sovraccarico dell'operatore

introduzione

In C ++, è possibile definire operatori come `+` e `->` per i tipi definiti dall'utente. Ad esempio, l'intestazione `<string>` definisce un operatore `+` per concatenare le stringhe. Questo viene fatto definendo una *funzione dell'operatore* usando la parola chiave `operator` .

Osservazioni

Gli operatori per i tipi built-in non possono essere modificati, gli operatori possono essere sovraccaricati solo per i tipi definiti dall'utente. Cioè, almeno uno degli operandi deve essere di un tipo definito dall'utente.

I seguenti operatori *non possono* essere sovraccaricati:

- L'accesso membro o l'operatore "punto" `.`
- Il puntatore all'operatore di accesso membro `.*`
- L'operatore di risoluzione dell'oscilloscopio, `::`
- L'operatore condizionale ternario, `?:`
- `dynamic_cast` , `static_cast` , `reinterpret_cast` , `const_cast` , `typeid` , `sizeof` , `alignof` e `noexcept`
- Le direttive di preelaborazione, `#` e `##` , che vengono eseguite prima che qualsiasi tipo di informazione sia disponibile.

Ci sono alcuni operatori che **non** si dovrebbe (99,98% del tempo) di sovraccarico:

- `&&` e `||` (preferire, invece, usare la conversione implicita in `bool`)
- `,`
- L'indirizzo-dell'operatore (unario `&`)

Perché? Perché sovraccaricano gli operatori che un altro programmatore non si aspetterebbe mai, comportando un comportamento diverso da quello previsto.

Ad esempio, l'utente ha definito `&&` e `||` sovraccarichi di questi operatori perdono la loro valutazione di corto circuito e perdono le loro proprietà sequenziamento speciali (C ++ 17) , il numero di sequenza si applica anche a `,` overload dell'operatore.

Examples

Operatori aritmetici

Puoi sovraccaricare tutti gli operatori aritmetici di base:

- `+` e `+=`
- `-` e `-=`
- `*` e `*=`

- `/` e `/=`
- `&` e `&=`
- `|` e `|=`
- `^` e `^=`
- `>>` e `>>=`
- `<<` e `<<=`

Il sovraccarico per tutti gli operatori è lo stesso. *Scorri verso il basso per la spiegazione*

Sovraccarico al di fuori della `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Sovraccarico all'interno di `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Nota: l' `operator+` dovrebbe tornare con un valore non const, in quanto restituire un riferimento non avrebbe senso (restituisce un *nuovo* oggetto) né restituirebbe un valore `const` (in genere non si dovrebbe restituirlo con `const`). Il primo argomento è passato per valore, perché? Perché

1. Non è possibile modificare l'oggetto originale (`Object foobar = foo + bar;` non dovrebbe modificare `foo` dopo tutto, non avrebbe senso)
2. Non puoi renderlo `const` , perché dovrai essere in grado di modificare l'oggetto (perché `operator+` è implementato in termini di `operator+=` , che modifica l'oggetto)

Passare da `const&` sarebbe un'opzione, ma poi dovrai fare una copia temporanea dell'oggetto passato. Passando per valore, il compilatore lo fa per te.

`operator+=` restituisce un riferimento allo stesso, perché è quindi possibile incatenarli (non usare però la stessa variabile, sarebbe un comportamento indefinito a causa dei punti di sequenza).

Il primo argomento è un riferimento (vogliamo modificarlo), ma non `const` , perché non sarebbe possibile modificarlo. Il secondo argomento non dovrebbe essere modificato, e così per la ragione della prestazione viene passato da `const&` (passando per riferimento const è più veloce che per valore).

Operatori unari

Puoi sovraccaricare i 2 operatori unari:

- `++foo` e `foo++`
- `--foo` e `foo--`

Il sovraccarico è uguale per entrambi i tipi (`++` e `--`). *Scorri verso il basso per la spiegazione*

Sovraccarico al di fuori della `class` / `struct` :

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Sovraccarico all'interno di `class` / `struct` :

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Nota: l'operatore del prefisso restituisce un riferimento a se stesso, in modo da poter continuare le

operazioni su di esso. Il primo argomento è un riferimento, poiché l'operatore del prefisso modifica l'oggetto, che è anche il motivo per cui non è `const` (non sarebbe possibile modificarlo in altro modo).

L'operatore postfisso restituisce di valore un valore temporaneo (il valore precedente) e quindi non può essere un riferimento, in quanto sarebbe un riferimento a un valore temporaneo, che sarebbe un valore immondizia alla fine della funzione, poiché la variabile temporanea si spegne di scopo). Inoltre non può essere `const` , perché dovresti essere in grado di modificarlo direttamente.

Il primo argomento è un riferimento non `const` all'oggetto "calling", perché se fosse `const` , non sarebbe possibile modificarlo e, se non fosse un riferimento, non si cambierebbe il valore originale.

È a causa della copia necessaria nei sovraccarichi dell'operatore postfisso che è meglio prendere l'abitudine di usare prefisso ++ anziché postfix ++ in cicli `for` . Dalla `for` prospettiva di ciclo, sono di solito funzionalmente equivalenti, ma ci potrebbe essere un leggero vantaggio delle prestazioni di utilizzare il prefisso ++, in particolare con le classi di "grasso" con un sacco di membri da copiare. Esempio di utilizzo del prefisso ++ in un ciclo for:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Operatori di confronto

Puoi sovraccaricare tutti gli operatori di confronto:

- `==` e `!=`
- `>` e `<`
- `>=` e `<=`

Il modo consigliato per sovraccaricare tutti questi operatori è implementando solo 2 operatori (`==` e `<`) e quindi li usa per definire il resto. *Scorri verso il basso per la spiegazione*

Sovraccarico al di fuori della `class` / `struct` :

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Sovraccarico all'interno di `class` / `struct` :

```
//Note that the functions are const, because if they are not const, you wouldn't be able
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

Ovviamente gli operatori restituiscono un `bool` , che indica `true` o `false` per l'operazione corrispondente.

Tutti gli operatori prendono i loro argomenti da `const&` , perché l'unica cosa che fanno gli operatori è il confronto, quindi non dovrebbero modificare gli oggetti. Passing By `&` (di riferimento) è più veloce di per valore, e per assicurarsi che gli operatori non modificano, si tratta di un `const` - Referenza.

Si noti che gli operatori all'interno della `class` / `struct` sono definiti come `const` , la ragione di ciò è che senza le funzioni `const` , il confronto degli oggetti `const` non sarebbe possibile, in quanto il compilatore non sa che gli operatori non modificano nulla.

Operatori di conversione

È possibile sovraccaricare gli operatori di tipo, in modo che il tipo possa essere convertito implicitamente nel tipo specificato.

L'operatore di conversione **deve** essere definito in una `class` / `struct` :

```
operator T() const { /* return something */ }
```

Nota: l'operatore è `const` per consentire la conversione di oggetti `const` .

Esempio:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
    // ^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;
```

Operatore di sottoscrizione di matrice

È anche possibile sovraccaricare l'operatore di indice `[]` .

Dovresti **sempre** (il 99,98% delle volte) implementare 2 versioni, una versione `const` e una not `const` , perché se l'oggetto è `const` , non dovrebbe essere in grado di modificare l'oggetto restituito da `[]` .

Gli argomenti vengono passati da `const&` anziché da valore perché il passaggio per riferimento è più rapido che per valore e `const` modo che l'operatore non cambi accidentalmente l'indice.

Gli operatori restituiscono per riferimento, poiché per impostazione è possibile modificare l'oggetto `[]` return, ovvero:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

Puoi sovraccaricare **solo** all'interno di una `class` / `struct` :

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Gli operatori di pedici multipli, `[][]...` , possono essere raggiunti tramite oggetti proxy. Il seguente esempio di una semplice classe matrice matrice principale lo dimostra:

```
template<class T>
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
```

```
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Operatore di chiamata di funzione

È possibile sovraccaricare l'operatore di chiamata di funzione `()` :

Il sovraccarico deve essere fatto all'interno di una `class` / `struct` :

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
    //return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

Per esempio:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Operatore di assegnazione

L'operatore di assegnazione è uno degli operatori più importanti perché consente di modificare lo stato di una variabile.

Se non sovraccarichi l'operatore dell'assegnazione per la tua `class` / `struct` , viene automaticamente generato dal compilatore: l'operatore di assegnazione generato automaticamente esegue un "assegnazione membro", cioè richiamando gli operatori di assegnazione su tutti i membri, in modo che un oggetto venga copiato all'altro, un membro alla volta. L'operatore di assegnazione dovrebbe essere sovraccarico quando la semplice assegnazione membro non è adatta per la `class` / `struct` , ad esempio se è necessario eseguire una **copia profonda** di un oggetto.

Sovraccarico dell'operatore di assegnazione `=` è facile, ma è necessario seguire alcuni semplici passaggi.

1. **Prova per l'auto-assegnazione.** Questo controllo è importante per due motivi:
    - l'autoassegnazione è una copia inutile, quindi non ha senso eseguirla;
    - il prossimo passo non funzionerà nel caso di un autoassegnazione.
2. **Pulisci i vecchi dati.** I vecchi dati devono essere sostituiti con quelli nuovi. Ora puoi comprendere la seconda ragione del passaggio precedente: se il contenuto dell'oggetto è stato distrutto, un'autoassegnazione non riuscirà a eseguire la copia.
3. **Copia tutti i membri.** Se sovraccarichi l'operatore di assegnazione per la tua `class` o la tua `struct` , non viene generato automaticamente dal compilatore, quindi dovrai incaricarti di copiare tutti i membri dall'altro oggetto.
4. **Ritorna** `*this` . L'operatore ritorna da solo per riferimento, poiché consente il concatenamento (cioè `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Nota: l' `other` viene passato da `const&` , poiché l'oggetto che si sta assegnando non deve essere modificato, e il passaggio per riferimento è più rapido del valore, e per essere sicuro che `operator=` non lo modifichi accidentalmente, è `const` .

L'operatore di assegnazione può **solo** essere sovraccaricato nella `class` / `struct` , perché il valore di sinistra `=` è **sempre** la `class` / `struct` stessa. Definirlo come una funzione libera non ha questa garanzia e non è consentito a causa di ciò.

Quando lo dichiarate nella `class` / `struct` , il valore di sinistra è implicitamente la `class` / `struct` stessa, quindi non ci sono problemi con questo.

Operatore NOT bit a bit

Il sovraccarico del NOT bit a bit (`~`) è abbastanza semplice. *Scorri verso il basso per la spiegazione*

Sovraccarico al di fuori della `class` / `struct` :

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Sovraccarico all'interno di `class` / `struct` :

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Nota: l' `operator~` ritorna di valore, perché deve restituire un nuovo valore (il valore modificato) e non un riferimento al valore (sarebbe un riferimento all'oggetto temporaneo, che avrebbe un valore immondizia al suo interno non appena l'operatore è fatto). Non `const` neanche perché il codice chiamante dovrebbe essere in grado di modificarlo in seguito (cioè `int a = ~a + 1;` dovrebbe essere possibile).

All'interno della `class` / `struct` si deve fare un oggetto temporaneo, perché non è possibile modificare `this` , come sarebbe modificare l'oggetto originale, che non dovrebbe essere il caso.

Operatori di cambio di bit per I / O

Gli operatori `<<` e `>>` sono comunemente usati come operatori "write" e "read":

- `std::ostream` overload `std::ostream <<` per scrivere variabili nel flusso sottostante (esempio: `std::cout`)
- `std::istream` overload `>>` per leggere dal flusso sottostante a una variabile (esempio: `std::cin`)

Il modo in cui lo fanno è simile se si desidera sovraccaricarli "normalmente" al di fuori della `class` / `struct` , eccetto che specificando gli argomenti non sono dello stesso tipo:

- Il tipo di `std::ostream` è il flusso da sovraccaricare (ad esempio, `std::ostream`) passato per riferimento, per consentire il concatenamento (Concatenamento: `std::cout << a << b;`). Esempio: `std::ostream&`
- `lhs` sarebbe lo stesso del tipo di ritorno
- `rhs` è il tipo da cui vuoi consentire l'overloading (cioè `T`), passato da `const&` invece dal valore per il motivo delle prestazioni (`rhs` non dovrebbe essere comunque modificato). Esempio: `const Vector&` .

Esempio:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Numeri complessi rivisitati

Il codice seguente implementa un tipo di numero complesso molto semplice per il quale il campo sottostante viene automaticamente promosso, seguendo le regole di promozione del tipo della lingua, in applicazione dei quattro operatori di base (+, -, * e /) con un membro di un campo diverso (sia esso un altro `complex<T>` o un tipo scalare).

Questo è inteso come un esempio olistico che copre il sovraccarico dell'operatore accanto all'uso di base dei modelli.

```
#include <type_traits>

namespace not_std{

using std::decay_t;

//---------------------------------------------------------------------
// complex< value_t >
//---------------------------------------------------------------------

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
```

```
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}
    {}

    template<typename other_value_t>
    explicit complex(const complex<other_value_t> &other)
    : x{static_cast<const value_t &>(other.x)}
    , y{static_cast<const value_t &>(other.y)}
    {}

    complex &operator = (const complex &) = default;
    complex &operator = (complex &&) = default;
    complex(const complex &) = default;
    complex(complex &&) = default;
    complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//---------------------------------------------------------------
// operator - (negation)
//---------------------------------------------------------------

template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }
```

```
//-------------------------------------------------------------------
// operator +
//-------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//-------------------------------------------------------------------
// operator -
//-------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//-------------------------------------------------------------------
// operator *
//-------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
        };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
```

```
{ return {a.x * b, a.y * b}; }

//------------------------------------------------------------------
// operator /
//------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
        };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
         b.x * s,
        -b.y * s
        };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std


int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};

    // All of these are complex<double>
    auto c0 = a + b;
```

```
    auto c1 = a - b;
    auto c2 = a * b;
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}
```

Operatori denominati

È possibile estendere C ++ con operatori denominati che sono "quotati" da operatori C ++ standard.

Per prima cosa iniziamo con una libreria dozzina di righe:

```
namespace named_operator {
  template<class D>struct make_operator{constexpr make_operator(){}};

  template<class T, char, class O> struct half_apply { T&& lhs; };

  template<class Lhs, class Op>
  half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
    return {std::forward<Lhs>(lhs)};
  }

  template<class Lhs, class Op, class Rhs>
  auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
  -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
  {
    return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
  }
}
```

questo non fa ancora niente

Innanzitutto, aggiungendo i vettori

```
namespace my_ns {
```

```
  struct append_t : named_operator::make_operator<append_t> {};
  constexpr append_t append{};

  template<class T, class A0, class A1>
  std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const&
rhs ) {
      lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
      return std::move(lhs);
  }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

auto c = a *append* b;
```

Il nucleo qui è che definiamo un oggetto `append` di tipo `append_t:named_operator::make_operator<append_t>` .

Quindi sovraccarichiamo named_invoke (lhs, append_t, rhs) per i tipi che vogliamo a destra e a sinistra.

La libreria sovraccarica `lhs*append_t` , restituendo un oggetto `half_apply` temporaneo. `half_apply*rhs` anche `half_apply*rhs` per chiamare `named_invoke( lhs, append_t, rhs )` .

Dobbiamo semplicemente creare il token `append_t` appropriato e creare un `named_invoke` compatibile con ADL con la firma appropriata, e tutto si aggancia e funziona.

Per un esempio più complesso, si supponga di voler avere una moltiplicazione degli elementi di un array std :::

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
  return [](auto&& f) {
    return f( std::integral_constant<std::size_t, Is>{}... );
  };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
  struct e_times_t : named_operator::make_operator<e_times_t> {};
  constexpr e_times_t e_times{};

  template<class L, class R, std::size_t N,
    class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
  >
  std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N>
const& rhs ) {
    using result_type = std::array<Out, N>;
    auto index_over_N = indexer<N>();
    return index_over_N([&](auto...is)->result_type {
      return {{
        (lhs[is] * rhs[is])...
      }};
    });
```

```
  }
}
```

esempio dal vivo

Questo codice array element-element può essere esteso per lavorare su tuple o coppie o array in stile C, o anche contenitori di lunghezza variabile se si decide cosa fare se le lunghezze non corrispondono.

Si potrebbe anche digitare un tipo di operatore element-element e ottenere `lhs *element_wise<'+'>* rhs` .

Anche gli operatori di prodotto `*dot*` e `*cross*` sono ovvi usi.

L'uso di `*` può essere esteso per supportare altri delimitatori, come `+` . La precisione del delimitatore determina la precisione dell'operatore indicato, che può essere importante quando si traducono le equazioni fisiche in C ++ con l'uso minimo di extra `()` s.

Con una leggera modifica nella libreria sopra, possiamo supportare gli operatori `->*then*` ed estendere la `std::function` prima che lo standard sia aggiornato, o scrivere monadic `->*bind*` . Potrebbe anche avere un operatore con lo stato, in cui passiamo con attenzione l' `Op` alla funzione di invocazione finale, permettendo:

```
named_operator<'*'> append = [](auto lhs, auto&& rhs) {
  using std::begin; using std::end;
  lhs.insert( end(lhs), begin(rhs), end(rhs) );
  return std::move(lhs);
};
```

generazione di un operatore denominato in allegato al contenitore in C ++ 17.

Leggi Sovraccarico dell'operatore online: https://riptutorial.com/it/cplusplus/topic/562/sovraccarico-dell-operatore

Capitolo 110: Specifiche di collegamento

introduzione

Una specifica di collegamento indica al compilatore di compilare le dichiarazioni in un modo che consenta loro di essere collegate insieme alle dichiarazioni scritte in un'altra lingua, come ad esempio C.

Sintassi

- extern *string-literal* { *declaration-seq* (*opt*)}
- *dichiarazione* extern *stringa-letterale*

Osservazioni

Lo standard richiede che tutti i compilatori supportino l' `extern "C"` per consentire a C ++ di essere compatibile con C e `extern "C++"` , che può essere utilizzato per sovrascrivere una specifica di linkage e ripristinare l'impostazione predefinita. Altre specifiche del collegamento supportate sono definite dall'implementazione .

Examples

Gestore del segnale per sistema operativo Unix-like

Poiché un gestore di segnale verrà chiamato dal kernel usando la convenzione di chiamata C, dobbiamo dire al compilatore di usare la convenzione di chiamata C durante la compilazione della funzione.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
    bind(...);
    listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

Creare un'intestazione di libreria C compatibile con C ++

L'intestazione della libreria AC può essere generalmente inclusa in un programma C ++, poiché la maggior parte delle dichiarazioni sono valide sia in C che in C ++. Ad esempio, si consideri il seguente `foo.h` :

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

La definizione di `make_foo` viene compilata e distribuita separatamente con l'intestazione in forma di oggetto.

Un programma C ++ può `#include <foo.h>` , ma il compilatore non saprà che la funzione `make_foo` è definita come un simbolo C e probabilmente cercherà di cercarla con un nome `make_foo` e non riuscirà a `make_foo` . Anche se riesce a trovare la definizione di `make_foo` nella libreria, non tutte le piattaforme utilizzano le stesse convenzioni di chiamata per C e C ++ e il compilatore C ++ utilizzerà la convenzione di chiamata C ++ quando chiama `make_foo` , il che probabilmente causerà un errore di segmentazione se `make_foo` si aspetta di essere chiamato con la convenzione di chiamata C.

Il modo per ovviare a questo problema è quello di racchiudere quasi tutte le dichiarazioni nell'intestazione in un blocco `extern "C"` .

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

#ifdef __cplusplus
}  /* end of "extern C" block */
#endif
```

Ora quando `foo.h` è incluso da un programma C, apparirà semplicemente come dichiarazioni ordinarie, ma quando `foo.h` è incluso in un programma C ++, `make_foo` si troverà all'interno di un blocco `extern "C"` e il compilatore saprà di cercare un nome senza maglie e usa la convenzione di chiamata C.

Leggi Specifiche di collegamento online: https://riptutorial.com/it/cplusplus/topic/9268/specifiche-di-collegamento

Capitolo 111: Specifiers di classe di archiviazione

introduzione

Gli identificatori delle classi di archiviazione sono parole chiave che possono essere utilizzate nelle dichiarazioni. Non influenzano il tipo della dichiarazione, ma in genere modificano il modo in cui l'entità è archiviata.

Osservazioni

Esistono sei identificatori di classe di memoria, sebbene non tutti nella stessa versione della lingua: `auto` (fino a C ++ 11), `register` (fino a C ++ 17), `static` , `thread_local` (dal C ++ 11), `extern` e `mutable` .

Secondo lo standard,

> Al massimo un *identificatore della classe di memoria* deve apparire in un *declinatore-se-specificatore seq,* eccetto che `thread_local` può apparire con `static` o `extern` .

Una dichiarazione potrebbe non contenere alcun identificatore di classe di memoria. In tal caso, la lingua specifica un comportamento predefinito. Ad esempio, per impostazione predefinita, una variabile dichiarata a livello di blocco ha implicitamente durata di archiviazione automatica.

Examples

mutevole

Un identificatore che può essere applicato alla dichiarazione di un membro dati non statico e non di riferimento di una classe. Un membro mutabile di una classe non è `const` nemmeno quando l'oggetto è `const` .

```
class C {
    int x;
    mutable int times_accessed;
  public:
    C(): x(0), times_accessed(0) {
    }
    int get_x() const {
        ++times_accessed; // ok: const member function can modify mutable data member
        return x;
    }
    void set_x(int x) {
        ++times_accessed;
        this->x = x;
    }
};
```

C ++ 11

Un secondo significato per `mutable` stato aggiunto in C ++ 11. Quando segue la lista dei parametri di un lambda, sopprime il `const` implicito sull'operatore di chiamata della funzione lambda. Pertanto, un lambda mutabile può modificare i valori delle entità catturate dalla copia. Vedi mutable lambda per maggiori dettagli.

```
std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                  [start]() mutable { return start++; });
    return result;
}
```

Si noti che `mutable` *non* è uno specificatore della classe di memoria quando viene usato in questo modo per formare un lambda mutabile.

Registrare

C ++ 17

Un identificatore della classe di memoria che suggerisce al compilatore che una variabile sarà usata pesantemente. La parola "registro" è legata al fatto che un compilatore potrebbe scegliere di memorizzare tale variabile in un registro della CPU in modo che sia accessibile in un numero inferiore di cicli di clock. È stato deprecato a partire da C ++ 11.

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

Sia le variabili locali che i parametri di funzione possono essere dichiarati `register` . A differenza di C, C ++ non pone alcuna restrizione su cosa può essere fatto con una variabile di `register` . Ad esempio, è valido prendere l'indirizzo di una variabile di `register` , ma ciò potrebbe impedire al compilatore di memorizzare effettivamente tale variabile in un registro.

C ++ 17

Il `register` parole chiave non è utilizzato e riservato. Un programma che utilizza il `register` parole chiave è mal formato.

statico

Lo specificatore della classe di archiviazione `static` ha tre diversi significati.

1. Fornisce il collegamento interno a una variabile o funzione dichiarata nell'ambito dello spazio dei nomi.

   ```
   // internal function; can't be linked to
   ```

```
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Dichiara una variabile con durata di memorizzazione statica (a meno che non sia `thread_local`). Le variabili nell'ambito dello spazio dei nomi sono implicitamente statiche. Una variabile locale statica viene inizializzata solo una volta, il primo controllo del tempo passa attraverso la sua definizione e non viene distrutta ogni volta che si esce dall'ambito.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";
}
```

3. Quando viene applicato alla dichiarazione di un membro della classe, dichiara che il membro deve essere un membro statico .

```
struct S {
    static S* create() {
        return new S;
    }
};
int main() {
    S* s = S::create();
}
```

Si noti che nel caso di un membro di dati statici di una classe, entrambi 2 e 3 si applicano contemporaneamente: la parola chiave `static` fa sia il membro in un membro di dati statici che lo trasforma in una variabile con durata di archiviazione statica.

auto

C ++ 03

Dichiara una variabile con durata della memorizzazione automatica. È ridondante, poiché la durata della memorizzazione automatica è già l'impostazione predefinita nell'ambito del blocco e l'identificatore automatico non è consentito nello spazio dei nomi.

```
void f() {
    auto int x; // equivalent to: int x;
    auto y;     // illegal in C++; legal in C89
}
auto int z;     // illegal: namespace-scope variable cannot be automatic
```

In C ++ 11, l' `auto` cambiato completamente il significato e non è più un identificatore della classe di memoria, ma viene invece utilizzato per la deduzione del tipo .

extern

L' `extern` classe di archiviazione `extern` può modificare una dichiarazione in uno dei tre modi seguenti, a seconda del contesto:

1. Può essere usato per dichiarare una variabile senza definirla. In genere, questo viene utilizzato in un file di intestazione per una variabile che verrà definita in un file di implementazione separato.

```
// global scope
int x;             // definition; x will be default-initialized
extern int y;      // declaration; y is defined elsewhere, most likely another TU
extern int z = 42; // definition; "extern" has no effect here (compiler may warn)
```

2. Fornisce il collegamento esterno a una variabile in ambito namespace anche se `const` o `constexpr` avrebbero altrimenti causato un collegamento interno.

```
// global scope
const int w = 42;            // internal linkage in C++; external linkage in C
static const int x = 42;     // internal linkage in both C++ and C
extern const int y = 42;     // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                             // it's in an unnamed namespace
}
```

3. Ricalcola una variabile a livello di blocco se precedentemente dichiarata con linkage. Altrimenti, dichiara una nuova variabile con linkage, che è un membro del più vicino spazio dei nomi.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
}
void g() {
    extern int y; // y has external linkage; refers to global y defined elsewhere
}
```

Una funzione può anche essere dichiarata `extern` , ma ciò non ha alcun effetto. Solitamente viene usato come suggerimento per il lettore che una funzione dichiarata qui è definita in un'altra unità di traduzione. Per esempio:

```
void f();        // typically a forward declaration; f defined later in this TU
extern void g(); // typically not a forward declaration; g defined in another TU
```

Nel codice precedente, se `f` stato cambiato in `extern` e `g` in non- `extern` , non influenzerebbe affatto la correttezza o la semantica del programma, ma probabilmente confonderebbe il lettore del codice.

Leggi Specifiers di classe di archiviazione online:
https://riptutorial.com/it/cplusplus/topic/9225/specifiers-di-classe-di-archiviazione

Capitolo 112: Sposta semantica

Examples

Spostare la semantica

Spostare la semantica è un modo di spostare un oggetto in un altro in C ++. Per questo, svuotiamo il vecchio oggetto e posizioniamo tutto ciò che aveva nel nuovo oggetto.

Per questo, dobbiamo capire che cos'è un riferimento di valore. Un riferimento rvalue (`T&&` dove T è il tipo di oggetto) non è molto diverso da un riferimento normale (`T&` , ora chiamato riferimenti lvalue). Ma agiscono come 2 tipi diversi, quindi possiamo creare costruttori o funzioni che prendono un tipo o l'altro, che sarà necessario quando si ha a che fare con la semantica del movimento.

Il motivo per cui abbiamo bisogno di due tipi diversi è quello di specificare due comportamenti diversi. I costruttori di riferimento di Lvalue sono correlati alla copia, mentre i costruttori di riferimento di rvalue sono correlati allo spostamento.

Per spostare un oggetto, useremo `std::move(obj)` . Questa funzione restituisce un riferimento di rvalue all'oggetto, in modo che possiamo rubare i dati da quell'oggetto in uno nuovo. Ci sono diversi modi per farlo, che sono discussi di seguito.

È importante notare che l'uso di `std::move` crea solo un riferimento di rvalue. In altre parole, l'istruzione `std::move(obj)` non cambia il contenuto di obj, mentre `auto obj2 = std::move(obj)` (possibilmente).

Sposta costruttore

Supponiamo di avere questo frammento di codice.

```
class A {
public:
    int a;
    int b;

    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

Per creare un costruttore di copie, cioè, per creare una funzione che copi un oggetto e ne crei uno nuovo, normalmente sceglieremo la sintassi mostrata sopra, avremmo un costruttore per A che prende un riferimento a un altro oggetto di tipo A, e dovremmo copiare l'oggetto manualmente all'interno del metodo.

In alternativa, avremmo potuto scrivere `A(const A &) = default;` che copia automaticamente su

tutti i membri, facendo uso del suo costruttore di copie.

Tuttavia, per creare un costruttore di mosse, prenderemo un riferimento di rvalue invece di un riferimento di lvalue, come qui.

```
class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};
```

Si prega di notare che abbiamo impostato i vecchi valori a `zero` . Il costruttore di movimento predefinito (`Wallet(Wallet&&) = default;`) copia il valore di `nrOfDollars` , in quanto è un POD.

Poiché la semantica del movimento è progettata per consentire lo stato di "rubare" dall'istanza originale, è importante considerare come dovrebbe apparire l'istanza originale dopo questo furto. In questo caso, se non cambiassimo il valore a zero avremmo raddoppiato la quantità di dollari in gioco.

```
Wallet a;
a.nrOfDollars = 1;
Wallet b (std::move(a)); //calling B(B&& other);
std::cout << a.nrOfDollars << std::endl; //0
std::cout << b.nrOfDollars << std::endl; //1
```

Così abbiamo spostato un oggetto costruito da un vecchio.

Mentre sopra è un semplice esempio, mostra ciò che il costruttore di movimento è destinato a fare. Diventa più utile in casi più complessi, come quando è coinvolta la gestione delle risorse.

```
    // Manages operations involving a specified type.
    // Owns a helper on the heap, and one in its memory (presumably on the stack).
    // Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.
    template<typename T,
             template<typename> typename HeapHelper,
             template<typename> typename StackHelper>
    class OperationsManager {
        using MyType = OperationsManager<T, HeapHelper, StackHelper>;

        HeapHelper<T>* h_helper;
        StackHelper<T> s_helper;
        // ...

      public:
        // Default constructor & Rule of Five.
        OperationsManager() : h_helper(new HeapHelper<T>) {}
        OperationsManager(const MyType& other)
          : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}
```

```
        MyType& operator=(MyType copy) {
            swap(*this, copy);
            return *this;
        }
        ~OperationsManager() {
            if (h_helper) { delete h_helper; }
        }

        // Move constructor (without swap()).
        // Takes other's HeapHelper<T>*.
        // Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move
constructor.
        // Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
        //  new helper when it's destroyed.
        OperationsManager(MyType&& other) noexcept
          : h_helper(other.h_helper),
            s_helper(std::move(other.s_helper)) {
            other.h_helper = nullptr;
        }

        // Move constructor (with swap()).
        // Places our members in the condition we want other's to be in, then switches members
        //  with other.
        // OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
        //     swap(*this, other);
        // }

        // Copy/move helper.
        friend void swap(MyType& left, MyType& right) noexcept {
            std::swap(left.h_helper, right.h_helper);
            std::swap(left.s_helper, right.s_helper);
        }
    };
```

Sposta il compito

Analogamente a come possiamo assegnare un valore a un oggetto con un riferimento a lvalue, copiandolo, possiamo anche spostare i valori da un oggetto a un altro senza costruirne uno nuovo. Chiamiamo questo incarico di movimento. Spostiamo i valori da un oggetto a un altro oggetto esistente.

Per fare ciò, dovremo sovraccaricare l' `operator =` , non in modo che richieda un riferimento a lvalue, come nell'assegnazione delle copie, ma in modo che richieda un riferimento di rvalue.

```
class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};
```

Questa è la tipica sintassi per definire l'assegnazione del movimento. Noi sovraccarichiamo l' `operator =` modo che possiamo nutrirlo con un riferimento di rvalue e può assegnarlo a un altro oggetto.

```
A a;
a.a = 1;
A b;
b = std::move(a); //calling A& operator= (A&& other)
std::cout << a.a << std::endl; //0
std::cout << b.a << std::endl; //1
```

Quindi, possiamo spostare assegnare un oggetto a un altro.

Utilizzo di std :: move per ridurre la complessità da O (n²) a O (n)

C ++ 11 ha introdotto il linguaggio di base e il supporto della libreria standard per lo **spostamento di** un oggetto. L'idea è che quando un oggetto *o* è un temporanea e vuole una copia logica, allora il suo sicuro per soli risorse s' antifurto *o,* ad esempio un buffer allocato in modo dinamico, lasciando *o* logicamente vuota ma ancora distruttibili e copiabile.

Il supporto linguistico di base è principalmente

- il **costruttore** del **valore di riferimento rval** `&&` , ad esempio, `std::string&&` è un riferimento di rvalue a `std::string` , che indica che quello riferito all'oggetto è un temporaneo le cui risorse possono essere semplicemente rubate (ovvero spostate)
- supporto speciale per un **costruttore di movimenti** `T( T&& )` , che dovrebbe spostare efficientemente le risorse dall'oggetto specificato, invece di copiarle effettivamente, e
- supporto speciale per un **operatore di assegnazione del movimento operatore** `auto operator=(T&&) -> T&` , che dovrebbe anche spostarsi dalla sorgente.

Il supporto della libreria standard è principalmente il modello di funzione `std::move` dall'intestazione `<utility>` . Questa funzione produce un riferimento di rvalue all'oggetto specificato, indicando che può essere spostato da, proprio come se fosse un temporaneo.

Per un contenitore la copia effettiva è tipicamente di complessità O (*n*), dove *n* è il numero di elementi nel contenitore, mentre lo spostamento è O (1), tempo costante. E per un algoritmo che copia logicamente quel contenitore *n* volte, ciò può ridurre la complessità dal solito O poco pratico (*n* ²) al solo O (*n*) lineare.

Nel suo articolo "Containers That Never Change" del Dr. Dobbs Journal del 19 settembre 2013 , Andrew Koenig ha presentato un interessante esempio di inefficienza algoritmica quando utilizza uno stile di programmazione in cui le variabili sono immutabili dopo l'inizializzazione. Con questo stile i loop sono generalmente espressi usando la ricorsione. E per alcuni algoritmi come la generazione di una sequenza di Collatz, la ricorsione richiede la copia logica di un contenitore:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
// "Containers That Never Change" September 19, 2013, available at
// <url: http://www.drdobbs.com/cpp/containters-that-never-change/240161543>

// Includes here, e.g. <vector>

namespace my {
```

```
    template< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        return result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            return result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            return collatz_aux( n/2, new_result );
        }
        else
        {
            return collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        return collatz_aux( n, Vector_<int>() );
    }
}  // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}
```

Produzione:

```
42 21 64 32 16 8 4 2
```

Il numero di operazioni di copia degli elementi a causa della copia dei vettori è qui approssimativamente O (n^2), poiché è la somma 1 + 2 + 3 + ... n .

In numeri concreti, con i compilatori g ++ e Visual C ++ la precedente invocazione di `collatz(42)` prodotto una sequenza Collatz di 8 voci e 36 operazioni di copia degli elementi (8 * `collatz(42)` = 28, più alcune) nelle chiamate del costruttore di copie vettoriali.

Tutte queste operazioni di copia degli elementi possono essere rimosse semplicemente spostando i vettori i cui valori non sono più necessari. Per fare ciò è necessario rimuovere `const` e reference per gli argomenti del tipo vettoriale, passando i vettori *per valore* . I ritorni di funzione sono già ottimizzati automaticamente. Per le chiamate in cui i vettori vengono passati e non utilizzati di nuovo nella funzione, basta applicare `std::move` per *spostare* quei buffer anziché copiarli effettivamente:

```
using std::move;

auto concat( Vector_<int> v, int const x )
    -> Vector_<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-
Wpessimizing-move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector_<int> result )
    -> Vector_<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result;      // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector_<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector_<int>() );
}
```

Qui, con i compilatori g ++ e Visual C ++, il numero di operazioni di copia degli elementi dovute alle invocazioni del costruttore della copia vettoriale era esattamente 0.

L'algoritmo è necessariamente ancora O (*n*) nella lunghezza della sequenza di Collatz prodotta, ma questo è un miglioramento piuttosto drammatico: O (n^2) → O (*n*).

Con un supporto linguistico si potrebbe forse usare lo spostamento e ancora esprimere e applicare l'immutabilità di una variabile *tra la sua inizializzazione e la mossa finale* , dopo di che qualsiasi utilizzo di tale variabile dovrebbe essere un errore. Purtroppo, come in C ++ 14 C ++ non

supporta questo. Per il codice senza loop, il non utilizzo dopo lo spostamento può essere applicato tramite una nuova dichiarazione del nome pertinente come una `struct` incompleta, come nel caso di `struct result;` sopra, ma questo è brutto e difficilmente comprensibile da altri programmatori; anche la diagnostica può essere abbastanza fuorviante.

Riassumendo, il linguaggio C ++ e il supporto della libreria per lo spostamento consentono drastici miglioramenti nella complessità dell'algoritmo, ma a causa dell'incompletezza del supporto, al costo di abbandonare le garanzie di correttezza del codice e la chiarezza del codice che `const` può fornire.

Per completezza, la classe vettoriale instrumentata utilizzata per misurare il numero di operazioni di copia articolo a causa di invocazioni del costruttore di copie:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

    vector<Item>    items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

    Copy_tracking_vector(){}

    Copy_tracking_vector( Copy_tracking_vector const& other )
        : items_( other.items_ )
    { n_copy_ops() += items_.size(); }

    Copy_tracking_vector( Copy_tracking_vector&& other )
        : items_( move( other.items_ ) )
    {}
};
```

Utilizzare la semantica di movimento sui contenitori

Puoi spostare un contenitore invece di copiarlo:

```
void print(const std::vector<int>& vec) {
    for (auto&& val : vec) {
        std::cout << val << ", ";
    }
    std::cout << std::endl;
}

int main() {
```

```
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector
    std::vector<int> vec1{1, 2, 3, 4};
    std::vector<int> vec2;

    // The following line will print 1, 2, 3, 4
    print(vec1);

    // The following line will print a new line
    print(vec2);

    // The vector vec2 is assigned with move assingment.
    // This will "steal" the value of vec1 without copying it.
    vec2 = std::move(vec1);

    // Here the vec1 object is in an indeterminate state, but still valid.
    // The object vec1 is not destroyed,
    // but there's is no guarantees about what it contains.

    // The following line will print 1, 2, 3, 4
    print(vec2);
}
```

Riutilizzare un oggetto spostato

Puoi riutilizzare un oggetto spostato:

```
void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector as gained a determinate value, we can use it normally.
    vec.push_back(42);

    // Send the vector by move again.
    consumingFunction(std::move(vec));
}
```

Leggi Sposta semantica online: https://riptutorial.com/it/cplusplus/topic/2129/sposta-semantica

Capitolo 113: static_assert

Sintassi

- static_assert (*bool_constexpr* , *message*)
- static_assert (*bool_constexpr*) / * Dal C ++ 17 * /

Parametri

Parametro	Dettagli
bool_constexpr	Espressione da controllare
Messaggio	Messaggio da stampare quando *bool_constexpr* è *falso*

Osservazioni

A differenza delle asserzioni di runtime , le asserzioni statiche vengono verificate in fase di compilazione e vengono applicate anche durante la compilazione di build ottimizzate.

Examples

static_assert

Le asserzioni significano che una condizione dovrebbe essere controllata e se è falsa, è un errore. Per `static_assert()` , questo viene fatto in fase di compilazione.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

Un `static_assert()` ha un primo parametro obbligatorio, la condizione, ovvero un constexpr bool. *Potrebbe* avere un secondo parametro, il messaggio, che è una stringa letterale. Da C ++ 17, il secondo parametro è facoltativo; prima di ciò, è obbligatorio.

C ++ 17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

È usato quando:

- In generale, è richiesta una verifica in fase di compilazione su alcuni tipi sul valore di constexpr
- Una funzione modello deve verificare alcune proprietà di un tipo passato ad esso
- Uno vuole scrivere casi di test per:
  - metafunzioni modello
  - funzioni di constexpr
  - metaprogrammazione macro
- Alcune definizioni sono obbligatorie (per esempio, versione C ++)
- Porting di codice legacy, asserzioni su `sizeof(T)` (ad esempio, int a 32 bit)
- Alcune funzionalità del compilatore sono necessarie per il funzionamento del programma (compressione, ottimizzazione della classe base vuota, ecc.)

Si noti che `static_assert()` non partecipa a SFINAE : quindi, quando sono possibili sovraccarichi / specializzazioni aggiuntivi, non si dovrebbe usarlo al posto delle tecniche di metaprogrammazione del modello (come `std::enable_if<>`). Potrebbe essere utilizzato nel codice del modello quando il sovraccarico / specializzazione previsto è già stato trovato, ma sono necessarie ulteriori verifiche. In questi casi, potrebbe fornire più messaggi di errore concreti che fare affidamento su SFINAE per questo.

Leggi static_assert online: https://riptutorial.com/it/cplusplus/topic/3822/static-assert

Capitolo 114: std :: Atomics

Examples

tipi atomici

Ogni istanza e specializzazione completa del modello `std::atomic` definisce un tipo atomico. Se un thread scrive su un oggetto atomico mentre un altro thread legge da esso, il comportamento è ben definito (vedi il modello di memoria per i dettagli sulle gare di dati)

Inoltre, gli accessi agli oggetti atomici possono stabilire la sincronizzazione tra thread e ordinare accessi di memoria non atomici come specificato da `std::memory_order` .

std :: atomic può essere istanziato con qualsiasi `TriviallyCopyable type T. std::atomic` non è né copiabile né mobile.

La libreria standard fornisce le specializzazioni del modello std :: atomic per i seguenti tipi:

1. Viene definita una specializzazione completa per il tipo `bool` e il suo nome typedef che viene trattato come `std::atomic<T>` non specializzato, tranne che ha un layout standard, un costruttore banale di default, distruttori banali e supporta la sintassi di inizializzazione aggregata:

Nome typedef	Specializzazione completa
std::atomic_bool	std::atomic<bool>

2) Specializzazioni complete e typedef per tipi interi, come segue:

Nome typedef	Specializzazione completa
std::atomic_char	std::atomic<char>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>

Nome typedef	Specializzazione completa
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>

Nome typedef	Specializzazione completa
std::atomic_size_t	std::atomic<std::size_t>
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>

Semplice esempio di utilizzo di std :: atomic_int

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::memory_order_relaxed
#include <thread>         // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
  foo.store(x,std::memory_order_relaxed);     // set value atomically
}

void print_foo() {
  int x;
  do {
    x = foo.load(std::memory_order_relaxed);  // get value atomically
  } while (x==0);
  std::cout << "foo: " << x << '\n';
}

int main ()
{
  std::thread first (print_foo);
  std::thread second (set_foo,10);
  first.join();
  //second.join();
  return 0;
}
//output: foo: 10
```

Leggi std :: Atomics online: https://riptutorial.com/it/cplusplus/topic/7475/std----atomics

Capitolo 115: std :: coppia

Examples

Creare una coppia e accedere agli elementi

La coppia ci consente di trattare due oggetti come un unico oggetto. Le coppie possono essere facilmente costruite con l'aiuto della funzione template `std::make_pair` .

Il modo alternativo è quello di creare una coppia e assegnare i suoi elementi (`first` e `second`) più tardi.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements




    //We can also create a pair and assign the elements later
    std::pair<int,int> p1;
    p1.first = 3;
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //We can also create a pair using a constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;
}
```

Confronta gli operatori

I parametri di questi operatori sono `lhs` e `rhs`

- `operator==` verifica se entrambi gli elementi su `lhs` e `rhs` sono uguali. Il valore restituito è `true` se entrambi `lhs.first == rhs.first` AND `lhs.second == rhs.second` , altrimenti `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);

if (p1 == p2)
    std::cout << "equals";
else
    std::cout << "not equal"//statement will show this, because they are not identical
```

- `operator!=` verifica se alcuni elementi sulla coppia `lhs` e `rhs` non sono uguali. Il valore

restituito è `true` se `lhs.first != rhs.first` OR `lhs.second != rhs.second` , altrimenti restituisce `false` .

- `operator<` verifica se `lhs.first<rhs.first` , restituisce `true` . Altrimenti, se `rhs.first<lhs.first` restituisce `false` . Altrimenti, se `lhs.second<rhs.second` restituisce `true` , altrimenti restituisce `false` .
- `operator<=` restituisce `!(rhs<lhs)`
- `operator>` restituisce `rhs<lhs`
- `operator>=` restituisce `!(lhs<rhs)`

Un altro esempio con contenitori di coppie. Utilizza l' `operator<` perché ha bisogno di ordinare il contenitore.

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

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
                                                   {2, "bar"},
                                                   {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

Leggi std :: coppia online: https://riptutorial.com/it/cplusplus/topic/4834/std----coppia

Capitolo 116: std :: forward_list

introduzione

`std::forward_list` è un contenitore che supporta l'inserimento e la rimozione rapida di elementi da qualsiasi posizione nel contenitore. L'accesso casuale rapido non è supportato. È implementato come un elenco collegato singolarmente e, in sostanza, non ha alcun overhead rispetto alla sua implementazione in C. Rispetto a `std::list` questo contenitore offre uno spazio più efficiente di storage quando l'iterazione bidirezionale non è necessaria.

Osservazioni

L'aggiunta, la rimozione e lo spostamento degli elementi all'interno dell'elenco o tra più elenchi non invalida gli iteratori che si riferiscono ad altri elementi nell'elenco. Tuttavia, un iteratore o riferimento che fa riferimento a un elemento viene invalidato quando l'elemento corrispondente viene rimosso (tramite erase_after) dall'elenco. std :: forward_list soddisfa i requisiti del Container (eccetto per la funzione membro dimensione e la complessità dell'operatore == è sempre lineare), AllocatorAwareContainer e SequenceContainer.

Examples

Esempio

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::forward_list<std::string> words3(words1);
```

```
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Produzione:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

metodi

Nome del metodo	Definizione
`operator=`	assegna valori al contenitore
`assign`	assegna valori al contenitore
`get_allocator`	restituisce l'allocatore associato
------	------
Accesso all'elemento	
`front`	accedi al primo elemento
------	------
iteratori	
`before_begi` n	restituisce un iteratore all'elemento prima di iniziare
`cbefore_begin`	restituisce un iteratore costante all'elemento prima di iniziare
`begin`	restituisce un iteratore all'inizio
`cbegin`	restituisce un iteratore const all'inizio
`end`	restituisce un iteratore fino alla fine
`cend`	restituisce un iteratore fino alla fine
Capacità	
`empty`	controlla se il contenitore è vuoto
`max_size`	restituisce il numero massimo possibile di elementi

Nome del metodo	Definizione
modificatori	
`clear`	cancella il contenuto
`insert_after`	inserisce elementi dopo un elemento
`emplace_after`	costruisce elementi sul posto dopo un elemento
`erase_after`	cancella un elemento dopo un elemento
`push_front`	inserisce un elemento all'inizio
`emplace_front`	costruisce un elemento sul posto all'inizio
`pop_front`	rimuove il primo elemento
`resize`	cambia il numero di elementi memorizzati
`swap`	scambia il contenuto
operazioni	
`merge`	unisce due liste ordinate
`splice_after`	sposta elementi da un'altra forward_list
`remove`	rimuove elementi che soddisfano criteri specifici
`remove_if`	rimuove elementi che soddisfano criteri specifici
`reverse`	inverte l'ordine degli elementi
`unique`	rimuove gli elementi duplicati consecutivi
`sort`	ordina gli elementi

Leggi std :: forward_list online: https://riptutorial.com/it/cplusplus/topic/9703/std----forward-list

Capitolo 117: std :: function: per avvolgere qualsiasi elemento che è callable

Examples

Uso semplice

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

std :: funzione usata con std :: bind

Pensa a una situazione in cui dobbiamo richiamare una funzione con argomenti. `std::function` usata con `std::bind` fornisce un costrutto di design molto potente come mostrato di seguito.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }

};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ": " << i << std::endl;
    }
```

```
    void DoSomethingOnA()
    {
        anObjA.foo();
    }

    A anObjA;
};

int main(int argc, char *argv[])
{
     B anObjB;
     anObjB.DoSomethingOnA();
}
```

std :: function con lambda e std :: bind

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
    std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                             [capture_value](int param) -> int { return 7 + capture_value *
param; })
              << std::endl;
    // result: 15 ==  value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
    std::cout << c << std::endl;
```

```
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

overhead `function`

`std::function` può causare un overhead significativo. Poiché `std::function` ha [valore semantica] [1], deve copiare o spostare il dato callable in se stesso. Ma dal momento che può assumere callable di tipo arbitrario, spesso è necessario allocare la memoria in modo dinamico per farlo.

Alcune implementazioni di `function` hanno il cosiddetto "ottimizzazione di piccoli oggetti", dove piccoli tipi (come puntatori di funzioni, puntatori di membri o funtori con uno stato molto piccolo) saranno memorizzati direttamente nell'oggetto `function` . Ma anche questo funziona solo se il tipo non è `noexcept` . Inoltre, lo standard C ++ non richiede che tutte le implementazioni ne forniscano una.

Considera quanto segue:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;

void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

Un parametro template sarebbe la soluzione preferita per `SortMyContainer` , ma supponiamo che questo non sia possibile o desiderabile per qualsiasi motivo. `SortMyContainer` non ha bisogno di memorizzare `pred` oltre la propria chiamata. Eppure, `pred` può ben allocare memoria se il funtore che gli viene dato è di una dimensione non banale.

`function` alloca la memoria perché ha bisogno di qualcosa da copiare / spostare; `function` assume la proprietà del chiamabile. Ma `SortMyContainer` non ha bisogno di *possedere* il callable; lo sta solo facendo riferimento. Quindi usare la `function` qui è eccessivo; potrebbe essere efficiente, ma potrebbe non farlo.

Non esiste un tipo di funzione di libreria standard che si riferisca semplicemente a un callable. Quindi dovrà essere trovata una soluzione alternativa, oppure puoi scegliere di vivere con il sovraccarico.

Inoltre, la `function` non ha mezzi efficaci per controllare da dove provengono le allocazioni di memoria per l'oggetto. Sì, ha costruttori che accettano un `allocator` , ma [molte implementazioni non le implementano correttamente ... o addirittura *affatto*] [2].

C ++ 17

I costruttori di `function` che prendono un `allocator` non fanno più parte del tipo. Pertanto, non è

possibile gestire l'allocazione.

Chiamare una `function` è anche più lento di chiamare direttamente i contenuti. Poiché qualsiasi istanza di `function` può contenere un chiamabile, la chiamata attraverso una `function` deve essere indiretta. L'overhead della `function` di chiamata è nell'ordine di una chiamata di funzione virtuale.

Legatura std :: funzione a tipi diversi chiamabili

```
/*
 * This example show some ways of using std::function to call
 *  a) C-like function
 *  b) class-member function
 *  c) operator()
 *  d) lambda function
 *
 * Function call can be made:
 *  a) with right arguments
 *  b) argumens with different order, types and count
 */
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;



// simple function to be called
double foo_fn(int x, float y, double z)
{
  double res = x + y + z;
  std::cout << "foo_fn called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
  return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
    // this member function has different signature - but it can be used too
    // please not that argument order is changed too
    double foo_fn_4(int x, double z, float y, long xx)
    {
        double res = x + y + z + xx;
```

```
        std::cout << "foo_struct::foo_fn_4 called with arguments: "
                << x << ", " << z << ", " << y << ", " << xx
                << " result is : " << res
                << std::endl;
        return res;
    }
    // overloaded operator() makes whole object to be callable
    double operator()(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::operator() called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
};


int main(void)
{
  // typedefs
  using function_type = std::function<double(int, float, double)>;

  // foo_struct instance
  foo_struct fs;

  // here we will store all binded functions
  std::vector<function_type> bindings;

  // var #1 - you can use simple function
  function_type var1 = foo_fn;
  bindings.push_back(var1);

  // var #2 - you can use member function
  function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
  bindings.push_back(var2);

  // var #3 - you can use member function with different signature
  // foo_fn_4 has different count of arguments and types
  function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, 0l);
  bindings.push_back(var3);

  // var #4 - you can use object with overloaded operator()
  function_type var4 = fs;
  bindings.push_back(var4);

  // var #5 - you can use lambda function
  function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lambda  called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    };
  bindings.push_back(var5);

  std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
            << std::endl;
```

```
  for (auto f : bindings)
      f(1, 2, 3);

}
```

Vivere

Produzione:

```
Test stored functions with arguments: x = 1, y = 2, z = 3
foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6
lambda  called with arguments: 1, 2, 3 result is : 6
```

Memorizzazione degli argomenti delle funzioni in std :: tuple

Alcuni programmi richiedono quindi di archiviare argomenti per chiamate future di alcune funzioni.

Questo esempio mostra come chiamare qualsiasi funzione con argomenti memorizzati in std :: tuple

```
#include <iostream>
#include <functional>
#include <tuple>
#include <iostream>

// simple function to be called
double foo_fn(int x, float y, double z)
{
   double res =  x + y + z;
   std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z
             << " res=" << res;
   return res;
}

// helpers for tuple unrolling
template<int ...> struct seq {};
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };

// invocation helper
template<typename FN, typename P, int ...S>
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)
{
   return fn(std::get<S>(params) ...);
}
// call function with arguments stored in std::tuple
template<typename Ret, typename ...Args>
Ret call_fn(const std::function<Ret(Args...)>& fn,
            const std::tuple<Args...>& params)
{
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());
}
```

```
int main(void)
{
  // arguments
  std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);
  // function to call
  std::function<double(int, float, double)> fn = foo_fn;

  // invoke a function with stored arguments
  call_fn(fn, t);
}
```

Vivere

Produzione:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

Leggi std :: function: per avvolgere qualsiasi elemento che è callable online: https://riptutorial.com/it/cplusplus/topic/2294/std----function--per-avvolgere-qualsiasi-elemento-che-e-callable

Capitolo 118: std :: integer_sequence

introduzione

Il modello di classe `std::integer_sequence<Type, Values...>` rappresenta una sequenza di valori di tipo `Type` cui `Type` è uno dei tipi interi incorporati. Queste sequenze vengono utilizzate quando si implementano modelli di classi o funzioni che traggono vantaggio dall'accesso posizionale. La libreria standard contiene anche i tipi "factory" che creano sequenze ascendenti di valori interi solo dal numero di elementi.

Examples

Trasforma una tupla std :: in parametri di funzione

Una `std::tuple<T...>` può essere utilizzata per passare più valori in giro. Ad esempio, potrebbe essere usato per memorizzare una sequenza di parametri in qualche forma di coda. Quando si elabora una tale tupla, i suoi elementi devono essere convertiti in argomenti di chiamata di funzione:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// ------------------------------------------------------------------------
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\n";
}

// ------------------------------------------------------------------------
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
```

```
}

// ------------------------------------------------------------------------
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

Finché una classe supporta `std::get<I>(object)` e `std::tuple_size<T>::value` può essere espanso con la funzione `process()` sopra. La funzione stessa è completamente indipendente dal numero di argomenti.

Creare un pacchetto di parametri costituito da numeri interi

`std::integer_sequence` riguarda lo svolgimento di una sequenza di numeri interi che possono essere trasformati in un pacchetto di parametri. Il suo valore principale è la possibilità di creare modelli di classe "factory" creando queste sequenze:

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << '\n';
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

Il modello di funzione `print_sequence()` usa una `print_sequence() std::initializer_list<bool>` quando si espande la sequenza intera per garantire l'ordine di valutazione e non creare una variabile [array] inutilizzata.

Trasforma una sequenza di indici in copie di un elemento

L'espansione del pacchetto di parametri degli indici in un'espressione di virgola con un valore crea una copia del valore per ciascuno degli indici. Purtroppo, `gcc` e `clang` pensano che l'indice non

abbia alcun effetto e lo mettano in guardia (`gcc` può essere messo a tacere gettando l'indice a `void`):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
    auto array = make_array<20>(std::string("value"));
    std::copy(array.begin(), array.end(),
              std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << "\n";
}
```

Leggi std :: integer_sequence online: https://riptutorial.com/it/cplusplus/topic/8315/std----integer-sequence

Capitolo 119: std :: iomanip

Examples

std :: setw

```
int val = 10;
// val will be printed to the extreme left end of the  output console:
std::cout << val << std::endl;
 // val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;
```

Questo produce:

```
10
        10
1234567890
```

(dove l'ultima riga è lì per aiutare a vedere i caratteri offset).

A volte è necessario impostare la larghezza del campo di output, solitamente quando è necessario ottenere l'output in un layout strutturato e corretto. Questo può essere fatto usando **`std::setw`** di **std :: iomanip** .

La sintassi per `std::setw` è:

```
std::setw(int n)
```

dove n è la lunghezza del campo di output da impostare

std :: setprecision

Se utilizzato in un'espressione `out << setprecision(n)` o `in >> setprecision(n)` , imposta il parametro di precisione dello stream out o in esattamente a n. Il parametro di questa funzione è intero, che è un nuovo valore per la precisione.

Esempio:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
              << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
              << "max precision:         "
              << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
```

```
              << pi << '\n';
}
//Output
//default precision (6): 3.14159
//std::precision(10):    3.141592654
//max precision:         3.141592653589793239
```

std :: setfill

Se utilizzato in un'espressione `out << setfill(c)` imposta il carattere di riempimento dello stream su `c` .

Nota: il carattere di riempimento corrente può essere ottenuto con `std::ostream::fill` .

Esempio:

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
          << "setfill('*'): " << std::setfill('*')
                              << std::setw(10) << 42 << '\n';
}
//output::
//default fill:         42
//setfill('*'): ********42
```

std :: setiosflags

Se utilizzato in un'espressione `out << setiosflags(mask)` o `in >> setiosflags(mask)` , imposta tutti i flag di formato dello stream out o in come specificato dalla maschera.

Elenco di tutti gli `std::ios_base::fmtflags` :

- `dec` - usa la base decimale per l'I / O intero
- `oct` - usa la base ottale per l'intero I / O
- `hex` - usa la base esadecimale per l'I / O intero
- `basefield` - `dec|oct|hex|0` utile per le operazioni di mascheramento
- regolazione `left` / sinistra (aggiungi caratteri di riempimento a destra)
- regolazione `right` -destra (aggiunge caratteri di riempimento a sinistra)
- `internal` - aggiustamento interno (aggiunge caratteri di riempimento al punto designato interno)
- `adjustfield` - `left|right|internal` . Utile per le operazioni di mascheramento
- `scientific` - generi i tipi in virgola mobile usando la notazione scientifica o la notazione esadecimale se combinati con fissi
- `fixed` - generi i tipi a virgola mobile usando la notazione fissa o la notazione esadecimale se combinata con quella scientifica
- `floatfield` - `scientific|fixed|(scientific|fixed)|0` . Utile per le operazioni di mascheramento
- `boolalpha` - inserire ed estrarre `bool` tipo in formato alfanumerico
- `showbase` : genera un prefisso che indica la base numerica per l'output intero, richiede

l'indicatore di valuta `showbase` / O monetario
- `showpoint` - genera un carattere punto decimale incondizionatamente per l'uscita del numero in virgola mobile
- `showpos` - genera un carattere `+` per l'output numerico non negativo
- `skipws` : salta gli spazi bianchi prima di determinate operazioni di input
- `unitbuf` svuota l'output dopo ogni operazione di uscita
- `uppercase` - sostituire alcune lettere minuscole con i loro equivalenti maiuscoli in determinate operazioni di output output

Esempio di manipolatori:

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
  int l_iTemp = 47;
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl;
  //output: 57
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl;
  //output: 2f
  std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl;
  //output 2F
  std::cout<<std::setfill('0')<<std::setw(12);
  std::cout<<std::resetiosflags(std::ios_base::uppercase);
  std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl;
  //output: 00000000002f

  std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
  std::cout<<std::setfill('.')<<std::setw(10);
  std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl;
  //output: 47........

  std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
  std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
  std::cout<<std::setw(10)<<l_iTemp<<std::endl;
  //output +#######47

  double l_dTemp = -1.2;
  double pi = 3.14159265359;
  std::cout<<pi<<"    "<<l_dTemp<<std::endl;
  //output +3.14159   -1.2
  std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl;
  //output -1.20000
  std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl;
  //output: +3.141593e+00
  std::cout<<std::resetiosflags(std::ios_base::floatfield);
  std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
  //output: +3.141593
  bool b = true;
  std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
  //output: true
  return 0;
}
```

Leggi std :: iomanip online: https://riptutorial.com/it/cplusplus/topic/6936/std----iomanip

Capitolo 120: std :: map

Osservazioni

- Per usare qualsiasi di `std::map` o `std::multimap` dovrebbe essere incluso il file di intestazione `<map>` .
- `std::map` e `std::multimap` mantengono i loro elementi ordinati secondo l'ordine crescente delle chiavi. In caso di `std::multimap` , non avviene alcun ordinamento per i valori della stessa chiave.
- La differenza fondamentale tra `std::map` e `std::multimap` è che `std::map` one non consente valori duplicati per la stessa chiave dove `std::multimap` fa.
- Le mappe sono implementate come alberi di ricerca binari. Quindi `search()` , `insert()` , `erase()` richiede Θ (log n) tempo medio. Per il funzionamento a tempo costante usa `std::unordered_map` .
- `size()` funzioni `size()` e `empty()` hanno Θ (1) complessità temporale, il numero di nodi è memorizzato nella cache per evitare di camminare attraverso l'albero ogni volta che vengono chiamate queste funzioni.

Examples

Accesso agli elementi

Una coppia `std::map` accetta `(key, value)` come input.

Considera il seguente esempio di inizializzazione di `std::map` :

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),
                                        std::make_pair("docs-beta", 1) };
```

In una `std::map` , gli elementi possono essere inseriti come segue:

```
ranking["stackoverflow"]=2;
ranking["docs-beta"]=1;
```

Nell'esempio precedente, se lo `stackoverflow` della chiave è già presente, il suo valore verrà aggiornato a 2. Se non è già presente, verrà creata una nuova voce.

In una `std::map` , è possibile accedere direttamente agli elementi dando la chiave come indice:

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Si noti che l'utilizzo `operator[]` sulla mappa in realtà *inserirà un nuovo valore* con la chiave

interrogata nella mappa. Ciò significa che non puoi usarlo su una `const std::map` , anche se la chiave è già memorizzata nella mappa. Per impedire questo inserimento, controlla se l'elemento esiste (per esempio usando `find()`) o usa `at()` come descritto di seguito.

C ++ 11

Gli elementi di una `std::map` sono accessibili con `at()` :

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Nota che `at()` verrà `std::out_of_range` un'eccezione `std::out_of_range` se il contenitore non contiene l'elemento richiesto.

In entrambi i contenitori `std::map` e `std::multimap` , è possibile accedere agli elementi utilizzando gli iteratori:

C ++ 11

```
// Example using begin()
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
auto it = mmp.begin();
std::cout << it->first << " : " << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "1 : docs-beta"
```

Inizializzazione di std :: map o std :: multimap

`std::map` e `std::multimap` possono essere inizializzati fornendo coppie chiave-valore separate da virgola. Le coppie valore-chiave possono essere fornite da `{key, value}` o possono essere create esplicitamente da `std::make_pair(key, value)` . Poiché `std::map` non consente chiavi duplicate e l'operatore virgola va da destra a sinistra, la coppia a destra verrebbe sovrascritta con la coppia con lo stesso tasto a sinistra.

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };

// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange
```

```
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
```

Entrambi possono essere inizializzati con iteratore.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                                {6, 7} };
                        // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                         // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Eliminazione di elementi

Rimozione di tutti gli elementi:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap
```

Rimozione di elementi da qualche parte con l'aiuto di iteratore:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}
```

Rimozione di tutti gli elementi in un intervallo:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
it++; //moved first cursor on first {3, 4}
std::advance(it2,3);  //moved second cursor on first {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Rimozione di tutti gli elementi con un valore fornito come chiave:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Rimozione di elementi che soddisfano un predicato `pred` :

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
   if (pred(*it))
       it = m.erase(it);
   else
       ++it;
}
```

Inserimento di elementi

Un elemento può essere inserito in una `std::map` solo se la sua chiave non è già presente nella mappa. Dato ad esempio:

```
std::map< std::string, size_t > fruits_count;
```

- Una coppia chiave-valore viene inserita in una `std::map` attraverso la funzione membro `insert()` . Richiede una `pair` come argomento:

  ```
  fruits_count.insert({"grapes", 20});
  fruits_count.insert(make_pair("orange", 30));
  fruits_count.insert(pair<std::string, size_t>("banana", 40));
  fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));
  ```

  La funzione `insert()` restituisce una `pair` composta da un iteratore e un valore `bool` :

  - Se l'inserimento ha avuto successo, l'iteratore punta all'elemento appena inserito e il valore `bool` è `true` .
  - Se esisteva già un elemento con la stessa `key` , l'inserimento fallisce. Quando ciò accade, l'iteratore punta all'elemento che causa il conflitto e il valore `bool` è `false` .

  Il seguente metodo può essere utilizzato per combinare operazioni di inserimento e ricerca:

  ```
  auto success = fruits_count.insert({"grapes", 20});
  if (!success.second) {           // we already have 'grapes' in the map
      success.first->second += 20; // access the iterator to update the value
  }
  ```

- Per comodità, il contenitore `std::map` fornisce all'operatore subscript l'accesso agli elementi nella mappa e l'inserimento di nuovi se non esistono:

```
fruits_count["apple"] = 10;
```

Benché più semplice, impedisce all'utente di verificare se l'elemento esiste già. Se manca un elemento, `std::map::operator[]` lo crea implicitamente, inizializzandolo con il costruttore predefinito prima di sovrascriverlo con il valore fornito.

- `insert()` può essere usato per aggiungere più elementi contemporaneamente usando una lista di coppie rinforzate. Questa versione di insert () restituisce void:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` può anche essere usato per aggiungere elementi usando gli iteratori che denotano i valori di inizio e fine di `value_type` :

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Esempio:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){            // 'fruit' is already in the map
        ++ret.first->second;    // increment the counter
    }
}
```

La complessità temporale per un'operazione di inserimento è O (log n) perché `std::map` è implementata come alberi.

C ++ 11

Una `pair` può essere costruita esplicitamente usando `make_pair()` ed `emplace()` :

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

Se sappiamo dove verrà inserito il nuovo elemento, possiamo usare `emplace_hint()` per specificare un `hint` iteratore. Se il nuovo elemento può essere inserito prima del `hint` , l'inserimento può essere eseguito in tempo costante. Altrimenti si comporta allo stesso modo di `emplace()` :

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Iterating su std :: map o std :: multimap

`std::map` o `std::multimap` potrebbe essere attraversato dai seguenti modi:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

//Range based loop - since C++11
for(const auto &x: mmp)
    std::cout<< x.first <<":"<< x.second << std::endl;

//Forward iterator for loop: it would loop through first element to last element
//it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
std::cout<< it->first <<":"<< it->second << std::endl; //Do something with iterator

//Backward iterator for loop: it would loop through last element to first element
//it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
std::cout<< it->first <<" "<< it->second << std::endl; //Do something with iterator
```

Durante l'iterazione su `std::map` o `std::multimap` , l'uso di `auto` è preferito per evitare inutili conversioni implicite (vedere questa risposta per ulteriori dettagli).

Ricerca in std :: map o in std :: multimap

Esistono diversi modi per cercare una chiave in `std::map` o in `std::multimap` .

- Per ottenere l'iteratore della prima occorrenza di una chiave, è possibile utilizzare la funzione `find()` . Restituisce `end()` se la chiave non esiste.

  ```
  std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
  auto it = mmp.find(6);
  if(it!=mmp.end())
      std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
  else
      std::cout << "Value does not exist!" << std::endl;

  it = mmp.find(66);
  if(it!=mmp.end())
      std::cout << it->first << ", " << it->second << std::endl;
  else
      std::cout << "Value does not exist!" << std::endl; // This line would be executed.
  ```

- Un altro modo per scoprire se esiste una voce in `std::map` o in `std::multimap` sta utilizzando la funzione `count()` , che conta quanti valori sono associati a una determinata chiave. Poiché `std::map` associa un solo valore a ogni chiave, la sua funzione `count()` può solo restituire 0 (se la chiave non è presente) o 1 (se lo è). Per `std::multimap` , `count()` può restituire valori maggiori di 1 poiché possono essere presenti più valori associati alla stessa chiave.

  ```
  std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
  if(mp.count(3) > 0) // 3 exists as a key in map
      std::cout << "The key exists!" << std::endl; // This line would be executed.
  else
  ```

```
    std::cout << "The key does not exist!" << std::endl;
```

Se ti interessa solo che alcuni elementi esistano, `find` è strettamente migliore: documenta il tuo intento e, per `multimaps` , può fermarsi una volta trovato il primo elemento corrispondente.

- Nel caso di `std::multimap` , potrebbero esserci più elementi con la stessa chiave. Per ottenere questo intervallo, viene utilizzata la funzione `equal_range()` che restituisce `std::pair` con l'estremo inferiore di iteratore (incluso) e il limite superiore (esclusivo) rispettivamente. Se la chiave non esiste, entrambi gli iteratori puntano a `end()` .

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
    // prints: 6, 5
    //         6, 7
```

Controllo del numero di elementi

Il contenitore `std::map` ha una funzione membro `empty()` , che restituisce `true` o `false` , a seconda che la mappa sia vuota o meno. La funzione membro `size()` restituisce il numero di elementi memorizzati in un contenitore `std::map` :

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Tipi di mappe

Mappa normale

Una mappa è un contenitore associativo, contenente coppie chiave-valore.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

Nell'esempio precedente, `std::string` è il tipo di *chiave* e `size_t` è un *valore* .

La chiave agisce come un indice nella mappa. Ogni chiave deve essere unica e deve essere ordinata.

- Se hai bisogno di più elementi con la stessa chiave, considera l'utilizzo di `multimap` (spiegato sotto)

- Se il tuo tipo di valore non specifica alcun ordine, o se desideri sovrascrivere l'ordine predefinito, puoi fornire uno:

```
#include <string>
#include <map>
#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
               //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

  Se il comparatore `StrLess` restituisce `false` per due chiavi, vengono considerate uguali anche se i loro contenuti effettivi differiscono.

Multi-Map

Multimap consente di memorizzare più coppie di valori-chiave con lo stesso tasto nella mappa. Altrimenti, la sua interfaccia e la sua creazione sono molto simili alla normale mappa.

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

Hash-Map (Mappa non ordinata)

Una mappa hash memorizza coppie chiave-valore simili a una mappa normale. Tuttavia, non ordina gli elementi rispetto alla chiave. Invece, viene utilizzato un valore hash per la chiave per accedere rapidamente alle coppie chiave-valore necessarie.

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Le mappe non ordinate sono in genere più veloci, ma gli elementi non sono memorizzati in alcun ordine prevedibile. Ad esempio, l'iterazione di tutti gli elementi in una mappa `unordered_map` fornisce gli elementi in un ordine apparentemente casuale.

Creazione di std :: map con tipi definiti dall'utente come chiave

Per poter utilizzare una classe come chiave in una mappa, tutto ciò che è richiesto dalla chiave è che sia `copiable` e `assignable` . L'ordinamento all'interno della mappa è definito dal terzo argomento del modello (e dall'argomento del costruttore, se usato). Il *valore predefinito* è `std::less<KeyType>` , che per impostazione predefinita è `<` operatore, ma non è necessario utilizzare i valori predefiniti. Basta scrivere un operatore di confronto (preferibilmente come un oggetto funzionale):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        //  ...
    }
};
```

In C ++, il predicato "confronta" deve essere un severo ordinamento debole . In particolare, `compare(X,X)` deve restituire `false` per qualsiasi `X` cioè se `CmpMyType()(a, b)` restituisce true, quindi `CmpMyType()(b, a)` deve restituire false e, se entrambi restituiscono false, gli elementi sono considerati uguali (membri della stessa classe di equivalenza).

Ordinamento debole rigoroso

Questo è un termine matematico per definire una relazione tra due oggetti.
La sua definizione è:

> Due oggetti xey sono equivalenti se sia f (x, y) che f (y, x) sono falsi. Si noti che un oggetto è sempre (dall'invarianza dell'invarianza) equivalente a se stesso.

In termini di C ++ ciò significa che se si hanno due oggetti di un determinato tipo, si dovrebbero restituire i seguenti valori rispetto all'operatore <.

```
X    a;
X    b;

Condition:                  Test:     Result
a is equivalent to b:       a < b     false
a is equivalent to b        b < a     false

a is less than b            a < b     true
a is less than b            b < a     false

b is less than a            a < b     false
b is less than a            b < a     true
```

Il modo in cui definisci l'equivalente / meno dipende totalmente dal tipo di oggetto.

Leggi std :: map online: https://riptutorial.com/it/cplusplus/topic/681/std----map

Capitolo 121: std :: matrice

Parametri

Parametro	Definizione
`class T`	Specifica il tipo di dati dei membri dell'array
`std::size_t N`	Specifica il numero di membri nell'array

Osservazioni

L'uso di uno `std::array` richiede l'inclusione dell'intestazione `<array>` usando `#include <array>` .

Examples

Inizializzazione di uno std :: array

Inizializzazione di `std::array<T, N>` , dove `T` è un tipo scalare e `N` è il numero di elementi di tipo `T`

Se `T` è un tipo scalare, `std::array` può essere inizializzato nei seguenti modi:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Inizializzazione di `std::array<T, N>` , dove `T` è un tipo non scalare e `N` è il numero di elementi di tipo `T`

Se `T` è un tipo non scalare, `std::array` può essere inizializzato nei seguenti modi:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
```

```
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Accesso all'elemento

1. `at(pos)`

Restituisce un riferimento all'elemento in posizione `pos` con controllo dei limiti. Se `pos` non è all'interno dell'intervallo del contenitore, viene generata un'eccezione di tipo `std::out_of_range` .

La complessità è costante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) `operator[pos]`

Restituisce un riferimento all'elemento in posizione `pos` senza controllo dei limiti. Se `pos` non è compreso nell'intervallo del contenitore, può verificarsi un errore di *violazione della segmentazione* runtime. Questo metodo fornisce un accesso agli elementi equivalente agli array classici e quindi più efficiente rispetto `at(pos)` .

La complessità è costante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

Questa funzione **non membro** restituisce un riferimento all'elemento al **momento della compilazione** posizione **costante** `pos` senza controllo dei limiti. Se `pos` non è compreso nell'intervallo del contenitore, può verificarsi un errore di *violazione della segmentazione* runtime.

La complessità è costante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Restituisce un riferimento al primo elemento nel contenitore. Chiamare `front()` su un contenitore vuoto non è definito.

La complessità è costante O (1).

Nota: per un contenitore c, l'espressione `c.front()` è equivalente a `*c.begin()` .

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) **`back()`**

Restituisce il riferimento all'ultimo elemento nel contenitore. La richiamata `back()` su un contenitore vuoto non è definita.

La complessità è costante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) **`data()`**

Restituisce il puntatore all'array sottostante che funge da memoria dell'elemento. Il puntatore è tale che `range [data(); data() + size())` è sempre un intervallo valido, anche se il contenitore è vuoto (`data()` non è dereferenziabile in quel caso).

La complessità è costante O (1).

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
}
```

Controllo della dimensione della matrice

Uno dei principali vantaggi di `std::array` rispetto all'array in stile `C` è che possiamo controllare la dimensione dell'array usando la funzione membro `size()`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Iterazione attraverso la matrice

`std::array` essendo un contenitore STL, può usare loop per loop simile ad altri contenitori come il `vector`

```
int main() {
     std::array<int, 3> arr = { 1, 2, 3 };
     for (auto i : arr)
         cout << i << '\n';
}
```

Modifica di tutti gli elementi dell'array contemporaneamente

La funzione membro `fill()` può essere utilizzata su `std::array` per modificare i valori in una volta dopo l'inizializzazione

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

Leggi std :: matrice online: https://riptutorial.com/it/cplusplus/topic/2712/std----matrice

Capitolo 122: std :: opzionale

Examples

introduzione

Optionals (noto anche come Maybe types) sono usati per rappresentare un tipo il cui contenuto può o non può essere presente. Sono implementate in C ++ 17 come `std::optional` class `std::optional` . Ad esempio, un oggetto di tipo `std::optional<int>` può contenere un valore di tipo `int` , oppure potrebbe non contenere alcun valore.

Gli optionals sono comunemente usati per rappresentare un valore che potrebbe non esistere o come un tipo di ritorno da una funzione che può fallire nel restituire un risultato significativo.

Altri approcci a facoltativo

Ci sono molti altri approcci per risolvere il problema che `std::optional` risolve `std::optional` , ma nessuno di questi è abbastanza completo: usando un puntatore, usando una sentinella, o usando una `pair<bool, T>` .

Opzionale vs puntatore

In alcuni casi, possiamo fornire un puntatore a un oggetto esistente o `nullptr` per indicare un errore. Ma questo è limitato a quei casi in cui gli oggetti esistono già - `optional` , come un tipo di valore, può anche essere usato per restituire nuovi oggetti senza ricorrere all'assegnazione della memoria.

Opzionale vs Sentinel

Un idioma comune è usare un valore speciale per indicare che il valore non ha significato. Questo può essere 0 o -1 per i tipi interi o `nullptr` per i puntatori. Tuttavia, questo riduce lo spazio dei valori validi (non è possibile distinguere tra uno 0 valido e uno 0 senza significato) e molti tipi non hanno una scelta naturale per il valore sentinella.

Opzionale vs `std::pair<bool, T>`

Un altro idioma comune è quello di fornire una coppia, in cui uno degli elementi è un `bool` indica se il valore è significativo o meno.

Ciò si basa sul fatto che il tipo di valore è predefinito-constructible in caso di errore, che non è possibile per alcuni tipi e possibili ma indesiderabile per gli altri. Un `optional<T>` , in caso di errore, non ha bisogno di costruire nulla.

Utilizzo degli optionals per rappresentare l'assenza di un valore

Prima di C ++ 17, avere puntatori con un valore di `nullptr` rappresentava comunemente l'assenza di un valore. Questa è una buona soluzione per oggetti di grandi dimensioni che sono stati allocati dinamicamente e sono già gestiti da puntatori. Tuttavia, questa soluzione non funziona bene per i tipi piccoli o primitivi come `int` , che raramente vengono allocati o gestiti dinamicamente dai puntatori. `std::optional` fornisce una soluzione praticabile a questo problema comune.

In questo esempio, `struct Person` è definito. È possibile che una persona abbia un animale domestico, ma non è necessario. Pertanto, il membro `pet` di `Person` è dichiarato con un wrapper `std::optional` .

```
#include <iostream>
#include <optional>
#include <string>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Utilizzo degli optionals per rappresentare l'errore di una funzione

Prima del C ++ 17, una funzione in genere rappresentava un errore in uno dei seguenti modi:

- È stato restituito un puntatore nullo.
    - Ad es. Chiamare una funzione `Delegate *App::get_delegate()` su un'istanza di `App` che non aveva un delegato restituirebbe `nullptr` .
    - Questa è una buona soluzione per oggetti che sono stati allocati dinamicamente o sono grandi e gestiti da puntatori, ma non è una buona soluzione per gli oggetti di piccole dimensioni che sono in genere allocati allo stack e passati copiando.
- Un valore specifico del tipo restituito è stato riservato per indicare un errore.
    - Ad es. Chiamando una funzione `unsigned shortest_path_distance(Vertex a, Vertex b)` su due vertici che non sono collegati può restituire zero per indicare questo fatto.
- Il valore è stato accoppiato con un `bool` per indicare che il valore restituito era significativo.
    - es. Chiamando una funzione `std::pair<int, bool> parse(const std::string &str)` con un argomento stringa che non è un intero restituirebbe una coppia con un indefinito `int` e un `bool` impostato su `false` .

In questo esempio, a John vengono dati due animali domestici, Fluffy e Furball. La funzione `Person::pet_with_name()` viene quindi chiamata per recuperare i Whiskers di animali domestici di John. Poiché John non ha un animale domestico di nome Whiskers, la funzione fallisce e `std::nullopt` viene invece restituito.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;

    std::optional<Animal> pet_with_name(const std::string &name) {
        for (const Animal &pet : pets) {
            if (pet.name == name) {
                return pet;
            }
        }
        return std::nullopt;
    }
};

int main() {
    Person john;
    john.name = "John";

    Animal fluffy;
    fluffy.name = "Fluffy";
    john.pets.push_back(fluffy);

    Animal furball;
    furball.name = "Furball";
    john.pets.push_back(furball);

    std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
    if (whiskers) {
        std::cout << "John has a pet named Whiskers." << std::endl;
    }
    else {
        std::cout << "Whiskers must not belong to John." << std::endl;
    }
}
```

opzionale come valore di ritorno

```
std::optional<float> divide(float a, float b) {
  if (b!=0.f) return a/b;
  return {};
}
```

Qui restituiamo la frazione `a/b` , ma se non è definita (sarebbe infinita), restituiamo invece

l'opzione vuota.

Un caso più complesso:

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
  using std::begin; using std::end;
  auto b = begin(r), e = end(r);
  auto r = std::find_if(b, e , p );
  using iterator = decltype(r);
  if (r==e)
    return std::optional<iterator>();
  return std::optional<iterator>(r);
}
template<class Range, class T>
auto find( Range&& r, T const& t ) {
  return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}
```

`find( some_range, 7 )` cerca nel contenitore o range `some_range` per qualcosa uguale al numero `7` . `find_if` fa con un predicato.

Restituisce un opzionale vuoto se non è stato trovato, o un opzionale contenente un iteratore per l'elemento se lo era.

Questo ti permette di fare:

```
if (find( vec, 7 )) {
  // code
}
```

o anche

```
if (auto oit = find( vec, 7 )) {
  vec.erase(*oit);
}
```

senza dover scherzare con iteratori di inizio / fine e test.

value_or

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
  std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

`value_or` o restituisce il valore memorizzato `value_or` , o l'argomento se non c'è niente memorizza lì.

Ciò ti consente di prendere l'opzionale may-zero e di dare un comportamento predefinito quando hai effettivamente bisogno di un valore. In questo modo, la decisione del "comportamento predefinito" può essere rinviata al punto in cui è meglio realizzata e immediatamente necessaria, invece di generare un valore predefinito nelle profondità di alcuni motori.

Leggi std :: opzionale online: https://riptutorial.com/it/cplusplus/topic/2423/std----opzionale

Capitolo 123: std :: qualsiasi

Osservazioni

La classe `std::any` fornisce un contenitore sicuro per i tipi a cui possiamo mettere singoli valori di qualsiasi tipo.

Examples

Utilizzo di base

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
  std::any_cast<int>(an_object);
} catch(std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Produzione

```
hello world
Wrong type
42
```

Leggi std :: qualsiasi online: https://riptutorial.com/it/cplusplus/topic/7894/std----qualsiasi

Capitolo 124: std :: set e std :: multiset

introduzione

`set` è un tipo di contenitore i cui elementi sono ordinati e unici. `multiset` è simile, ma, nel caso del `multiset` , più elementi possono avere lo stesso valore.

Osservazioni

In questi esempi sono stati usati diversi stili di C ++. Fai attenzione se stai usando un compilatore C ++ 98; parte di questo codice potrebbe non essere utilizzabile.

Examples

Inserimento di valori in un set

Tre diversi metodi di inserimento possono essere usati con i set.

- Innanzitutto, un semplice inserimento del valore. Questo metodo restituisce una coppia che consente al chiamante di verificare se l'inserimento è realmente avvenuto.
- Secondo, un inserto dando un suggerimento su dove verrà inserito il valore. L'obiettivo è ottimizzare il tempo di inserimento in questo caso, ma sapere dove un valore deve essere inserito non è il caso comune. **Stai attento in quel caso; il modo di dare un suggerimento è diverso con le versioni del compilatore** .
- Infine puoi inserire un intervallo di valori dando un puntatore iniziale e finale. L'iniziale sarà incluso nell'inserimento, quello finale è escluso.

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> sut;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // Basic insert
  sut.insert(7);
  sut.insert(5);
  sut.insert(12);

  ret = sut.insert(23);
  if (ret.second==true)
    std::cout << "# 23 has been inserted!" << std::endl;

  ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert
should fail
  if (ret.second==false)
    std::cout << "# 23 already present in set!" << std::endl;
```

```
  // Insert with hint for optimization
  it = sut.end();
  // This case is optimized for C++11 and above
  // For earlier version, point to the element preceding your insertion
  sut.insert(it, 30);

  // inserting a range of values
  std::set<int> sut2;
  sut2.insert(20);
  sut2.insert(30);
  sut2.insert(45);
  std::set<int>::iterator itStart = sut2.begin();
  std::set<int>::iterator itEnd = sut2.end();

  sut.insert (itStart, itEnd); // second iterator is excluded from insertion

  std::cout << std::endl << "Set under test contains:" << std::endl;
  for (it = sut.begin(); it != sut.end(); ++it)
  {
    std::cout << *it << std::endl;
  }

  return 0;
}
```

L'output sarà:

```
# 23 has been inserted!

# 23 already present in set!



Set under test contains:

5

7

12

20

23

30

45
```

Inserimento di valori in un multiset

Tutti i metodi di inserimento dei set si applicano anche ai multiset. Tuttavia, esiste un'altra possibilità che fornisce una lista_inizializzazione:

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
```

```
msut.insert(il);
```

Modifica l'ordinamento predefinito di un set

`set` e `multiset` hanno metodi di confronto predefiniti, ma in alcuni casi potrebbe essere necessario sovraccaricarli.

Immaginiamo di memorizzare i valori stringa in un set, ma sappiamo che queste string contengono solo valori numerici. Di default l'ordinamento sarà un confronto di stringhe lessicografiche, quindi l'ordine non corrisponderà all'ordinamento numerico. Se vuoi applicare un ordinamento equivalente a quello che avresti con i valori `int` , hai bisogno di un functor per sovraccaricare il metodo di confronto:

```
#include <iostream>
#include <set>
#include <stdlib.h>

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"},
                                                     custom_compare{}); //< Compare object
optional as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
        std::cout << data << std::endl;

    return 0;
}
```

L'output sarà:

```
### Default sort on std::set<std::string> :
1
2
23
290
5
6
### Custom sort on set :
1
2
5
6
23
290

### Lambda sort on set :
6
5
290
23
2
1
```

Nell'esempio sopra, si possono trovare 3 diversi modi di aggiungere operazioni di confronto a `std::set` , ognuno di essi è utile nel proprio contesto.

Ordinamento predefinito

Questo utilizzerà l'operatore di confronto della chiave (primo argomento del modello). Spesso, la chiave fornirà già un buon valore predefinito per la funzione `std::less<T>` . A meno che questa funzione non sia specializzata, utilizza l' `operator<` dell'oggetto. Ciò è particolarmente utile quando un altro codice tenta anche di utilizzare alcuni ordini, poiché ciò consente la coerenza su tutto il codice base.

Scrivere il codice in questo modo ridurrà lo sforzo di aggiornare il codice quando la chiave cambia API, come: una classe contenente 2 membri che cambia in una classe contenente 3 membri. Aggiornando l' `operator<` nella classe, tutte le occorrenze verranno aggiornate.

Come ci si potrebbe aspettare, l'uso dell'ordinamento predefinito è un valore predefinito ragionevole.

Ordinamento personalizzato

L'aggiunta di un ordinamento personalizzato tramite un oggetto con un operatore di confronto viene spesso utilizzata quando il confronto predefinito non è conforme. Nell'esempio sopra questo è perché le stringhe si riferiscono a numeri interi. In altri casi, viene spesso utilizzato quando si desidera confrontare i puntatori (intelligenti) in base all'oggetto cui si riferiscono o perché sono necessari diversi vincoli per il confronto (esempio: confronto di `std::pair` con il valore di `first`).

Quando si crea un operatore di confronto, questo dovrebbe essere un ordinamento stabile. Se il risultato dell'operatore di confronto cambia dopo l'inserimento, si avrà un comportamento

indefinito. Come buona pratica, l'operatore di confronto dovrebbe utilizzare solo i dati costanti (membri const, funzioni const ...).

Come nell'esempio sopra, spesso incontrerai le classi senza membri come operatori di confronto. Ciò si traduce in costruttori e costruttori di copia predefiniti. Il costruttore predefinito consente di omettere l'istanza in fase di costruzione e il costruttore della copia è richiesto poiché il set accetta una copia dell'operatore di confronto.

Lambda sort

Lambdas è un modo più breve per scrivere oggetti funzione. Ciò consente di scrivere l'operatore di confronto su meno righe, rendendo più leggibile il codice generale.

Lo svantaggio dell'uso di lambda è che ogni lambda ottiene un tipo specifico in fase di compilazione, quindi `decltype(lambda)` sarà diverso per ogni compilazione della stessa unità di compilazione (file cpp) come su più unità di compilazione (se inclusa tramite file di intestazione). Per questo motivo, è consigliabile utilizzare gli oggetti funzione come operatore di confronto se utilizzato all'interno dei file di intestazione.

Questa costruzione si incontra spesso quando un oggetto `std::set` viene usato all'interno dell'ambito locale di una funzione, mentre l'oggetto funzione è preferito quando viene usato come argomenti di funzione o membri di classe.

Altre opzioni di ordinamento

Poiché l'operatore di confronto di `std::set` è un argomento modello, tutti gli oggetti chiamabili possono essere utilizzati come operatori di confronto e gli esempi sopra riportati sono solo casi specifici. Le uniche restrizioni che hanno questi oggetti callable sono:

- Devono essere copiabili costruibili
- Devono essere chiamabili con 2 argomenti del tipo di chiave. (le conversioni implicite sono consentite, sebbene non raccomandate in quanto possono danneggiare le prestazioni)

Ricerca di valori in set e multiset

Esistono diversi modi per cercare un determinato valore in `std::set` o in `std::multiset` :

Per ottenere l'iteratore della prima occorrenza di una chiave, è possibile utilizzare la funzione `find()` . Restituisce `end()` se la chiave non esiste.

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3); // contains 3, 10, 15, 22

auto itS = sut.find(10); // the value is found, so *itS == 10
itS = sut.find(555); // the value is not found, so itS == sut.end()
```

```
  std::multiset<int> msut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(15);
  sut.insert(3); // contains 3, 10, 15, 15, 22

  auto itMS = msut.find(10);
```

Un altro modo è usare la funzione `count()` , che conta quanti valori corrispondenti sono stati trovati nel `set` / `multiset` (nel caso di un `set` , il valore di ritorno può essere solo 0 o 1). Usando gli stessi valori di cui sopra, avremo:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

Nel caso di `std::multiset` , potrebbero esserci più elementi con lo stesso valore. Per ottenere questo intervallo, è possibile utilizzare la funzione `equal_range()` . Restituisce `std::pair` con limite inferiore iteratore (incluso) e limite superiore (esclusivo) rispettivamente. Se la chiave non esiste, entrambi gli iteratori puntano al valore superiore più vicino (basato sul metodo di confronto utilizzato per ordinare il `multiset`).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'

eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

Eliminazione di valori da un set

Il metodo più ovvio, se vuoi semplicemente resettare il tuo set / multiset su uno vuoto, è usare `clear` :

```
  std::set<int> sut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(3);
  sut.clear(); //size of sut is 0
```

Quindi è possibile utilizzare il metodo di `erase` . Offre alcune possibilità che sembrano in qualche modo equivalenti all'inserimento:

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
```

```
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
  std::cout << *it << std::endl;
}
```

L'output sarà:

```
Set under test contains:

10

15

30
```

Tutti questi metodi si applicano anche al `multiset` . Si noti che se si chiede di eliminare un elemento da un `multiset` ed è presente più volte, **tutti i valori equivalenti verranno eliminati** .

Leggi std :: set e std :: multiset online: https://riptutorial.com/it/cplusplus/topic/9005/std----set-e-std----multiset

Capitolo 125: std :: string

introduzione

Le stringhe sono oggetti che rappresentano sequenze di personaggi. La classe di `string` standard fornisce un'alternativa semplice, sicura e versatile all'utilizzo di array espliciti di `char` quando si tratta di testo e altre sequenze di caratteri. La classe di `string` C ++ fa parte dello spazio dei nomi `std` ed è stata standardizzata nel 1998.

Sintassi

- *// Dichiarazione di stringa vuota*

  std :: string s;

- *// Costruire da const char * (c-string)*

  std :: string s ("Hello");

  std :: string s = "Ciao";

- *// Costruire usando il costruttore di copie*

  std :: string s1 ("Hello");

  std :: string s2 (s1);

- *// Costruire dalla sottostringa*

  std :: string s1 ("Hello");

  std :: string s2 (s1, 0, 4); *// Copia 4 caratteri dalla posizione 0 di s1 a s2*

- *// Costruire da un buffer di caratteri*

  std :: string s1 ("Hello World");
  std :: string s2 (s1, 5); *// Copia i primi 5 caratteri di s1 in s2*

- *// Costruisci usando fill constructor (solo char)*

  std :: string s (5, 'a'); *// s contiene aaaaa*

- *// Costruisce usando il costruttore di intervalli e l'iteratore*

  std :: string s1 ("Hello World");

  std :: string s2 (s1.begin (), s1.begin () + 5); *// Copia i primi 5 caratteri di s1 in s2*

Osservazioni

Prima di usare `std::string` , dovresti includere la `string` intestazione, poiché include funzioni / operatori / sovraccarichi che altre intestazioni (ad esempio `iostream`) non includono.

L'uso del costruttore const char * con un nullptr porta a un comportamento non definito.

```
std::string oops(nullptr);
std::cout << oops << "\n";
```

Il metodo `at` lancia uno `std::out_of_range` un'eccezione se `index >= size()` .

Il comportamento di `operator[]` è un po 'più complicato, in tutti i casi ha un comportamento indefinito se `index > size()` , ma quando `index == size()` :

C ++ 11

1. Su una stringa non const, il comportamento *non* è *definito* ;
2. Su una stringa const, viene restituito un riferimento a un carattere con valore `CharT()` (il carattere *null*).

C ++ 11

1. Viene restituito un riferimento a un carattere con valore `CharT()` (il carattere *null*).
2. La modifica di questo riferimento è un *comportamento non definito* .

Dal momento che C ++ 14, invece di usare `"foo"` , si consiglia di usare `"foo"s` , poichè `s` è un suffisso letterale definito dall'utente , che converte il `const char* "foo"` in `std::string "foo"` .

Nota: è necessario utilizzare lo spazio dei nomi `std::string_literals` *o* `std::literals` *per ottenere i letterali* `s` *.*

Examples

scissione

Usa `std::string::substr` per dividere una stringa. Ci sono due varianti di questa funzione membro.

Il primo prende una *posizione di partenza* da cui dovrebbe iniziare la sottostringa restituita. La posizione di partenza deve essere valida nell'intervallo `(0, str.length()]` :

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

Il secondo prende una posizione di partenza e una *lunghezza* totale della nuova sottostringa. Indipendentemente dalla *lunghezza* , la sottostringa non andrà mai oltre la fine della stringa di

origine:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(15, 3); // "and"
```

Nota che puoi anche chiamare `substr` senza argomenti, in questo caso viene restituita una copia esatta della stringa

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Sostituzione di corde

Sostituisci per posizione

Per sostituire una parte di una `std::string` puoi utilizzare il metodo `replace` da `std::string` .

`replace` ha molti sovraccarichi utili:

```
//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

C ++ 14

```
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
```

C ++ 11

```
//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Sostituisci le occorrenze di una stringa con un'altra stringa

Sostituisci solo la prima occorrenza di `replace` con `with` in `str` :

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Sostituisci tutte le occorrenze di `replace` con `with` in `str` :

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

Concatenazione

Puoi concatenare `std::string` s usando gli operatori `+` e `+=` sovraccaricati. Usando l'operatore `+` :

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Usando l'operatore `+=` :

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

È anche possibile aggiungere stringhe C, compresi i letterali stringa:

```
std::string hello = "Hello";
std::string world = "world";
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

Puoi anche usare `push_back()` per respingere i singoli `char` :

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

C'è anche `append()` , che è più o meno come `+=` :

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Accedere a un personaggio

Esistono diversi modi per estrarre i caratteri da una `std::string` e ognuno è sottilmente differente.

```
std::string str("Hello world!");
```

---

operatore [] (n)

Restituisce un riferimento al carattere all'indice n.

`std::string::operator[]` non è controllato e non genera un'eccezione. Il chiamante è responsabile per affermare che l'indice si trova all'interno dell'intervallo della stringa:

```
char c = str[6]; // 'w'
```

---

a (n)

Restituisce un riferimento al carattere all'indice n.

`std::string::at` *is* bounds checked, e getterà `std::out_of_range` se l'indice non è all'interno dell'intervallo della stringa:

```
char c = str.at(7); // 'o'
```

C ++ 11

Nota: entrambi questi esempi generano un comportamento indefinito se la stringa è vuota.

---

davanti()

Restituisce un riferimento al primo carattere:

```
char c = str.front(); // 'H'
```

---

indietro()

Restituisce un riferimento all'ultimo carattere:

```
char c = str.back(); // '!'
```

tokenize

Elencato dal meno costoso al più costoso in fase di esecuzione:

1. `str::strtok` è il metodo di tokenizzazione standard più economico fornito, consente inoltre di modificare il delimitatore tra i token, ma comporta 3 difficoltà con il C ++ moderno:

    - `std::strtok` non può essere utilizzato su più `strings` contemporaneamente (anche se alcune implementazioni si estendono per supportare questo, come ad esempio: `strtok_s`)
    - Per lo stesso motivo, `std::strtok` non può essere utilizzato su più thread simultaneamente (questo potrebbe essere tuttavia definito dall'implementazione, ad esempio: l'implementazione di Visual Studio è thread-safe)
    - Chiamare `std::strtok` modifica la `std::string` cui sta operando, quindi non può essere usata su `const string` s, `const char*` s o stringhe letterali, per tokenizzare uno di questi con `std::strtok` o per operare su un `std::string` chi ha il contenuto da conservare, l'input dovrebbe essere copiato, quindi la copia può essere utilizzata

    Generalmente, qualsiasi costo di queste opzioni sarà nascosto nel costo di allocazione dei token, ma se è richiesto l'algoritmo più economico e le difficoltà di `std::strtok` non sono sovrastimabili, si consideri una soluzione filata a mano .

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Esempio dal vivo

2. `std::istream_iterator` utilizza iterativamente l'operatore di estrazione del flusso. Se l'input `std::string` è delimitato da spazi bianchi, questo è in grado di espandersi sull'opzione `std::strtok` eliminando le sue difficoltà, consentendo la tokenizzazione in linea supportando in tal modo la generazione di un `const vector<string>` e aggiungendo il supporto per più delimitare il carattere dello spazio bianco:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
```

```
                                        std::istream_iterator<std::string>());
```

Esempio dal vivo

3. `std::regex_token_iterator` usa una `std::regex` to iterativamente tokenize. Fornisce una definizione delimitatore più flessibile. Ad esempio, virgole non delimitate e spazi bianchi:

C ++ 11

```
// String to tokenize
const std::string str{ "The ,qu\\,ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\\\\,]|\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Esempio dal vivo

Vedi l' esempio `regex_token_iterator` per maggiori dettagli.

Conversione in (const) char *

Per ottenere un accesso `const char*` ai dati di una `std::string` puoi utilizzare la funzione membro `c_str()` della stringa. Tieni presente che il puntatore è valido solo finché l'oggetto `std::string` è all'interno dell'ambito e rimane invariato, il che significa che solo i metodi `const` possono essere richiamati sull'oggetto.

C ++ 17

La funzione membro `data()` può essere utilizzata per ottenere un `char*` modificabile, che può essere utilizzato per manipolare i dati dell'oggetto `std::string` .

C ++ 11

Un `char*` modificabile `char*` può anche essere ottenuto prendendo l'indirizzo del primo carattere: `&s[0]` . All'interno di C ++ 11, questo è garantito per produrre una stringa ben formata, con terminazione nulla. Nota che `&s[0]` è ben formato anche se `s` è vuoto, mentre `&s.front()` non è definito se `s` è vuoto.

C ++ 11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);
```

```
// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Trovare caratteri (s) in una stringa

Per trovare un carattere o un'altra stringa, puoi usare `std::string::find` . Restituisce la posizione del primo carattere della prima partita. Se non sono state trovate corrispondenze, la funzione restituisce `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

Trovato in posizione: 21

Le opportunità di ricerca sono ulteriormente ampliate dalle seguenti funzioni:

```
find_first_of     // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of      // Find last occurrence of characters
find_last_not_of  // Find last absence of characters
```

Queste funzioni consentono di cercare caratteri dalla fine della stringa e di trovare il caso negativo (cioè i caratteri che non sono nella stringa). Ecco un esempio:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

Trovato in posizione: 6

Nota: tenere presente che le funzioni di cui sopra non cercano sottostringhe, ma piuttosto per i caratteri contenuti nella stringa di ricerca. In questo caso, l'ultima occorrenza di `'g'` stata trovata nella posizione `6` (gli altri caratteri non sono stati trovati).

Taglio di caratteri all'inizio / fine

Questo esempio richiede le intestazioni `<algorithm>` , `<locale>` e `<utility>` .

C ++ 11

Tagliare una sequenza o una stringa significa rimuovere tutti gli elementi iniziali e finali (o caratteri) che corrispondono a un determinato predicato. Per prima cosa tagliamo gli elementi finali, perché non implica lo spostamento di alcun elemento e quindi taglia gli elementi principali. Si noti che le generalizzazioni di seguito funzionano per tutti i tipi di `std::basic_string` (ad es. `std::string` e `std::wstring`), e accidentalmente anche per contenitori di sequenze (ad es. `std::vector` e `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Ritagliare gli elementi finali significa trovare l' *ultimo* elemento che non corrisponde al predicato e cancellare da lì in poi:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Il taglio degli elementi principali comporta la ricerca del *primo* elemento che non corrisponde al predicato e che cancella fino a lì:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

Per specializzare quanto sopra per il taglio degli spazi bianchi in una `std::string` possiamo usare la funzione `std::isspace()` come predicato:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

Allo stesso modo, possiamo usare la funzione `std::iswspace()` per `std::wstring` ecc.

Se desideri creare una *nuova* sequenza che è una copia tagliata, puoi utilizzare una funzione separata:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Confronto lessicografico

Due `std::string` s possono essere confrontati lessicograficamente usando gli operatori `==` `!=` , `<` , `<=` , `>` E `>=` :

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

Tutte queste funzioni utilizzano il metodo `std::string::compare()` sottostante per eseguire il confronto e restituiscono valori convenienza booleani. L'operazione di queste funzioni può essere interpretata come segue, indipendentemente dall'attuale implementazione:

- operatore `==` :

  Se `str1.length() == str2.length()` e ogni coppia di caratteri corrisponde, restituisce `true` , altrimenti restituisce `false` .

- operatore `!=` :

  Se `str1.length() != str2.length()` o una coppia di caratteri non corrisponde, restituisce `true` , altrimenti restituisce `false` .

- operatore `<` o operatore `>` :

  Trova la prima coppia di caratteri diversa, li confronta e restituisce il risultato booleano.

- operatore `<=` o operatore `>=` :

  Trova la prima coppia di caratteri diversa, li confronta e restituisce il risultato booleano.

Nota: il termine **coppia di caratteri** indica i caratteri corrispondenti in entrambe le stringhe delle stesse posizioni. Per capire meglio, se due stringhe di esempio sono `str1` e `str2` e le loro lunghezze sono rispettivamente `n` e `m` , allora le coppie di caratteri di entrambe le stringhe significano ogni `str1[i]` e `str2[i]` dove *i = 0, 1, 2* `str1[i]` *.., max (n, m)* . Se per qualsiasi *i* dove il carattere corrispondente non esiste, cioè quando *i* è maggiore o uguale a `n` o `m` , sarebbe

considerato come il valore più basso.

Ecco un esempio di utilizzo di `<` :

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

I passi sono come segue:

1. Confronta i primi caratteri, `'B' == 'B'` - vai avanti.
2. Confronta i secondi caratteri, `'a' == 'a'` - vai avanti.
3. Confronta i terzi caratteri, `'r' == 'r'` - vai avanti.
4. La gamma `str2` è ora esaurita, mentre la gamma `str1` ha ancora caratteri. Quindi, `str2 < str1` .

Conversione a std :: wstring

In C ++, le sequenze di caratteri sono rappresentate specializzando la classe `std::basic_string` con un tipo di carattere nativo. Le due principali raccolte definite dalla libreria standard sono `std::string` e `std::wstring` :

- `std::string` è costruito con elementi di tipo `char`
- `std::wstring` è costruito con elementi di tipo `wchar_t`

Per convertire tra i due tipi, utilizzare `wstring_convert` :

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

Al fine di migliorare l'usabilità e / o la leggibilità, è possibile definire le funzioni per eseguire la conversione:

```
#include <string>
#include <codecvt>
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;
```

```
std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Esempio di utilizzo:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

Questo è sicuramente più leggibile di
`std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` .

Si noti che `char` e `wchar_t` non implicano la codifica e non forniscono indicazioni sulla dimensione in byte. Ad esempio, `wchar_t` viene comunemente implementato come un tipo di dati da 2 byte e in genere contiene dati codificati UTF-16 in Windows (o UCS-2 nelle versioni precedenti a Windows 2000) e come un tipo di dati di 4 byte codificati con UTF-32 in Linux. Questo è in contrasto con i nuovi tipi `char16_t` e `char32_t` , che sono stati introdotti in C ++ 11 e sono garantiti per essere abbastanza grandi da contenere un "carattere" UTF16 o UTF32 (o più precisamente, *punto di codice*), rispettivamente.

Usando la classe std :: string_view

C ++ 17

C ++ 17 introduce `std::string_view` , che è semplicemente un intervallo non proprietario di `const char` , implementabile come una coppia di puntatori o un puntatore e una lunghezza. È un tipo di parametro superiore per le funzioni che richiedono dati stringa non modificabili. Prima di C ++ 17, c'erano tre opzioni per questo:

```
void foo(std::string const& s);      // pre-C++17, single argument, could incur
                                     // allocation if caller's data was not in a string
                                     // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                     // both everywhere

void foo(const char* s);             // pre-C++17, single argument, but need to call
                                     // strlen()

template <class StringT>
void foo(StringT const& s);          // pre-C++17, caller can pass arbitrary char data
                                     // provider, but now foo() has to live in a header
```

Tutti questi possono essere sostituiti con:

```
void foo(std::string_view s);        // post-C++17, single argument, tighter coupling
                                     // zero copies regardless of how caller is storing
                                     // the data
```

Si noti che `std::string_view` ***non può*** *modificare i suoi dati sottostanti* .

`string_view` è utile quando si desidera evitare copie non necessarie.

Offre un sottoinsieme utile della funzionalità che `std::string` fa, sebbene alcune funzioni si comportino diversamente:

```
std::string str = "lllloooonnnnggggg sssstttrrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

In loop attraverso ogni personaggio

C ++ 11

`std::string` supporta gli iteratori e quindi puoi usare un loop *basato su intervalli* per scorrere ogni carattere:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

Puoi usare un ciclo "tradizionale" `for` far passare in loop ogni personaggio:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Conversione in numeri interi / in virgola mobile

Una `std::string` contenente un numero può essere convertita in un tipo intero o in un tipo a virgola mobile, utilizzando le funzioni di conversione.

Nota che tutte queste funzioni smettono di analizzare la stringa di input non appena incontrano un carattere non numerico, quindi `"123abc"` sarà convertito in `123` .

La famiglia di funzioni `std::ato*` converte stringhe in stile C (matrici di caratteri) in numeri interi o in virgola mobile:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

C ++ 11

```
long long num4 = std::atoll(ten.c_str());
```

Tuttavia, l'uso di queste funzioni è sconsigliato perché restituiscono `0` se non riescono a analizzare la stringa. Questo è negativo perché `0` potrebbe anche essere un risultato valido, se per esempio la stringa di input era "0", quindi è impossibile determinare se la conversione è effettivamente fallita.

La più recente famiglia di funzioni `std::sto*` converte `std::string` s in interi o in virgola mobile e genera eccezioni se non possono analizzare il loro input. *Dovresti usare queste funzioni se possibile* :

C ++ 11

```
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Inoltre, queste funzioni gestiscono anche le stringhe ottale e esagonale a differenza della famiglia `std::ato*` . Il secondo parametro è un puntatore al primo carattere non convertito nella stringa di input (non illustrato qui) e il terzo parametro è la base da utilizzare. `0` è il rilevamento automatico di ottale (a partire da `0`) e hex (a partire da `0x` o `0X`), e qualsiasi altro valore è la base da usare

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16);  // Returns 10
long num4 = std::stol(ten_hex);  // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Conversione tra codifiche di caratteri

La conversione tra codifiche è facile con C ++ 11 e la maggior parte dei compilatori è in grado di `<codecvt>` modo multipiattaforma attraverso le `<codecvt>` e `<locale>` .

```
#include <iostream>
```

```
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Ricorda che Visual Studio 2015 fornisce supporto per queste conversioni, ma un bug nella loro implementazione di libreria richiede l'utilizzo di un modello diverso per `wstring_convert` quando si ha a che fare con `char16_t` :

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
  std::basic_string<utf16_char> tmp;
  tmp.resize(utf16.length());
  std::copy(utf16.begin(), utf16.end(), tmp.begin());
  utf8 = conv_utf8_utf16.to_bytes(tmp);
}
void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
  std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
  utf16.clear();
  utf16.resize(tmp.length());
  std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Verifica se una stringa è un prefisso di un'altra

C ++ 14

In C ++ 14, questo è fatto facilmente da `std::mismatch` che restituisce la prima coppia di mismatching da due intervalli:

```
std::string prefix = "foo";
```

```
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Si noti che una versione di intervallo e mezza di `mismatch()` esisteva prima di C ++ 14, ma questo non è sicuro nel caso in cui la seconda stringa sia la più corta delle due.

C ++ 14

Possiamo ancora usare la versione range-and-half di `std::mismatch()` , ma dobbiamo prima verificare che la prima stringa sia al massimo grande come la seconda:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

C ++ 17

Con `std::string_view` , possiamo scrivere il confronto diretto che vogliamo senza doverci preoccupare del sovraccarico dell'allocazione o fare copie:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Conversione in std :: string

`std::ostringstream` può essere utilizzato per convertire qualsiasi tipo di streamable in una rappresentazione di stringa, inserendo l'oggetto in un oggetto `std::ostringstream` (con l'operatore di inserimento del flusso `<<`) e quindi convertendo l'intero `std::ostringstream` in uno `std::string` .

Per `int` per esempio:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Scrivendo la tua funzione di conversione, il semplice:

```
template<class T>
std::string toString(const T& x)
{
  std::ostringstream ss;
```

```
  ss << x;
  return ss.str();
}
```

funziona ma non è adatto per il codice critico delle prestazioni.

Le classi definite dall'utente possono implementare l'operatore di inserimento del flusso se lo si desidera:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

C ++ 11

A parte i flussi, dal momento che in C ++ 11 è possibile utilizzare anche la funzione `std::to_string` (e `std::to_wstring`) che è sovraccaricata per tutti i tipi fondamentali e restituisce la rappresentazione in stringa del suo parametro.

```
std::string s = to_string(0x12f3);  // after this the string s contains "4851"
```

Leggi std :: string online: https://riptutorial.com/it/cplusplus/topic/488/std----string

Capitolo 126: std :: variante

Osservazioni

La variante è un rimpiazzo per l'utilizzo di una grezza `union` . È sicuro dal tipo e sa di che tipo si tratta, e costruisce e distrugge attentamente gli oggetti al suo interno quando dovrebbe.

Non è quasi mai vuoto: solo nei casi d'angolo in cui il suo contenuto viene sostituito e non può essere ritirato in sicurezza, finisce per trovarsi in uno stato vuoto.

Si comporta in qualche modo come una `std::tuple` , e in qualche modo come un `std::optional` .

Usare `std::get` e `std::get_if` solito è una cattiva idea. La risposta corretta è solitamente `std::visit` , che ti consente di gestire ogni possibilità proprio lì. `if constexpr` può essere utilizzato all'interno della `visit` se è necessario `if constexpr` il comportamento, piuttosto che eseguire una sequenza di controlli di runtime che duplicano la `visit` modo più efficiente.

Examples

Base std :: uso variante

Questo crea una variante (un'unione con tag) in grado di memorizzare una `string int` o una `string` .

```
std::variant< int, std::string > var;
```

Possiamo memorizzare uno dei due tipi in esso:

```
var = "hello"s;
```

E possiamo accedere ai contenuti tramite `std::visit` :

```
// Prints "hello\n":
visit( [](auto&& e) {
  std::cout << e << '\n';
}, var );
```

passando in un lambda polimorfo o oggetto di funzione simile.

Se siamo certi di sapere di che tipo si tratta, possiamo ottenerlo:

```
auto str = std::get<std::string>(var);
```

ma questo verrà buttato se ci sbagliamo. `get_if` :

```
auto* str  = std::get_if<std::string>(&var);
```

restituisce `nullptr` se si indovina sbagliato.

Le varianti non garantiscono l'allocazione dinamica della memoria (diversa da quella assegnata dai loro tipi contenuti). Qui viene memorizzato solo uno dei tipi di una variante e, in rari casi (con l'eccezione di eccezioni durante l'assegnazione e nessun modo sicuro di uscire), la variante può diventare vuota.

Le varianti consentono di memorizzare più tipi di valore in una variabile in modo sicuro ed efficiente. Sono fondamentalmente intelligenti, di tipo sicuro `union` s.

Crea puntatori pseudo-metodi

Questo è un esempio avanzato.

È possibile utilizzare la variante per cancellare il tipo di peso leggero.

```
template<class F>
struct pseudo_method {
  F f;
  // enable C++17 class type deduction:
  pseudo_method( F&& fin ):f(std::move(fin)) {}

  // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
  template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
  friend decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
    // var->*method returns a lambda that perfect forwards a function call,
    // behaving like a method pointer basically:
    return [&](auto&&...args)->decltype(auto) {
      // use visit to get the type of the variant:
      return std::visit(
        [&](auto&& self)->decltype(auto) {
          // decltype(x)(x) is perfect forwarding in a lambda:
          return method.f( decltype(self)(self), decltype(args)(args)... );
        },
        std::forward<Var>(var)
      );
    };
  }
};
```

questo crea un tipo che sovraccarica l' `operator->*` con una `Variant` sul lato sinistro.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
  return decltype(self)(self).print( decltype(args)(args)... );
};
```

Ora se abbiamo 2 tipi ciascuno con un metodo di `print` :

```
struct A {
  void print( std::ostream& os ) const {
    os << "A";
```

```
  }
};
struct B {
  void print( std::ostream& os ) const {
    os << "B";
  }
};
```

si noti che sono tipi non correlati. Noi possiamo:

```
std::variant<A,B> var = A{};

(var->*print)(std::cout);
```

e invierà la chiamata direttamente ad `A::print(std::cout)` per noi. Se invece inizializziamo la `var` con `B{}` , si invierà a `B::print(std::cout)` .

Se abbiamo creato un nuovo tipo C:

```
struct C {};
```

poi:

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

non riuscirà a compilare, perché non esiste un `C.print(std::cout)` .

Estendere quanto sopra consentirebbe di rilevare e utilizzare le funzioni di `print` libere, eventualmente con l'uso di `if constexpr` all'interno dello pseudo-metodo di `print` .

esempio live attualmente usando `boost::variant` al posto di `std::variant` .

Costruire un `std :: variant`

Questo non copre gli allocatori.

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

Leggi std :: variante online: https://riptutorial.com/it/cplusplus/topic/5239/std----variante

Capitolo 127: std :: vector

introduzione

Un vettore è un array dinamico con storage gestito automaticamente. È possibile accedere agli elementi di un vettore con la stessa efficienza di quelli di un array con il vantaggio che i vettori possono cambiare in modo dinamico nelle dimensioni.

In termini di spazio di archiviazione, i dati vettoriali sono (di solito) collocati in una memoria allocata dinamicamente e richiedono quindi un piccolo overhead; viceversa `C-arrays` e `std::array` usano lo storage automatico relativo alla posizione dichiarata e quindi non hanno alcun overhead.

Osservazioni

L'uso di un `std::vector` richiede l'inclusione dell'intestazione `<vector>` usando `#include <vector>` .

Gli elementi in un file `std::vector` vengono archiviati in modo contiguo nel negozio gratuito. Va notato che quando i vettori sono nidificati come in `std::vector<std::vector<int> >` , gli elementi di ciascun vettore sono contigui, ma ciascun vettore alloca il proprio buffer sottostante nell'archivio gratuito.

Examples

Inizializzazione di un vettore std ::

Un `std::vector` può essere inizializzato in vari modi pur dichiarandolo:

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };  // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };     // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);  // v becomes {6, 6, 6}

std::vector<int> v(4);     // v becomes {0, 0, 0, 0}
```

Un vettore può essere inizializzato da un altro contenitore in diversi modi:

Copia la costruzione (solo da un altro vettore), che copia i dati dalla `v2` :

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

C ++ 11

Sposta la costruzione (solo da un altro vettore), che sposta i dati da `v2` :

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Iterator (range) copy-construction, che copia elementi in `v` :

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}

// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3);                   // v becomes {1, 2, 3}

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

C ++ 11

Costruzione di movimenti di Iterator, usando `std::make_move_iterator` , che sposta elementi in `v` :

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end());

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

Con l'aiuto della funzione membro `assign()` , un `std::vector` può essere reinizializzato dopo la sua costruzione:

```
v.assign(4, 100);                       // v becomes {100, 100, 100, 100}

v.assign(v2.begin(), v2.begin() + 3);   // v becomes {v2[0], v2[1], v2[2]}

int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4);                 // v becomes {2, 3, 4}
```

Inserimento di elementi

Aggiunta di un elemento alla fine di un vettore (copiando / spostando):

```
struct Point {
  double x, y;
  Point(double x, double y) : x(x), y(y) {}
};
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p);  // p is copied into the vector.
```

C ++ 11

Aggiunta di un elemento alla fine di un vettore costruendo l'elemento al suo posto:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Si noti che `std::vector` *non* ha una funzione membro `push_front()` causa di motivi di prestazioni. L'aggiunta di un elemento all'inizio fa muovere tutti gli elementi esistenti nel vettore. Se si desidera inserire spesso elementi all'inizio del contenitore, è possibile utilizzare invece `std::list` o `std::deque` .

Inserimento di un elemento in qualsiasi posizione di un vettore:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9);          // v now contains {9, 1, 2, 3}
```

C ++ 11

Inserimento di un elemento in qualsiasi posizione di un vettore costruendo l'elemento al suo posto:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9);     // v now contains {1, 9, 2, 3}
```

Inserimento di un altro vettore in qualsiasi posizione del vettore:

```
std::vector<int> v(4);      // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Inserimento di un array in qualsiasi posizione di un vettore:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Utilizzare `reserve()` prima di inserire più elementi se la dimensione vettoriale risultante è nota in anticipo per evitare riallocazioni multiple (vedere dimensioni e capacità del vettore):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Assicurati di non commettere l'errore di chiamare `resize()` in questo caso, altrimenti creerai inavvertitamente un vettore con 200 elementi in cui solo gli ultimi cento avranno il valore desiderato.

Iterating Over std :: vector

È possibile eseguire iterazioni su un `std::vector` in diversi modi. Per ciascuna delle seguenti sezioni, `v` è definito come segue:

```
std::vector<int> v;
```

Iterating nella direzione di andata

C ++ 11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);

// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

C ++ 11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Iterazione nella direzione inversa

C ++ 14

```
// There is no standard way to use range based for for this.
// See below for alternatives.
```

```
// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Anche se non esiste un modo integrato per utilizzare l'intervallo in base a per invertire l'iterazione; è relativamente semplice risolvere questo problema. L'intervallo basato sugli usi `begin()` e `end()` per ottenere gli iteratori e quindi simulare questo con un oggetto wrapper può ottenere i risultati che richiediamo.

C ++ 14

```
template<class C>
struct ReverseRange {
  C c; // could be a reference or a copy, if the original was a temporary
  ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
  ReverseRange(ReverseRange&&)=default;
  ReverseRange& operator=(ReverseRange&&)=delete;
  auto begin() const {return std::rbegin(c);}
  auto end()   const {return std::rend(c);}
};
// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Imporre elementi const

Dal C ++ 11 i `cbegin()` e `cend()` consentono di ottenere un *iteratore costante* per un vettore, anche se il vettore non è const. Un iteratore costante consente di leggere ma non modificare il contenuto del vettore che è utile per applicare la correttezza const:

C ++ 11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
```

```
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

C ++ 17

as_const estende questa estensione all'intervallo:

```
for (auto const& e : std::as_const(v)) {
  std::cout << e << '\n';
}
```

È facile da implementare nelle versioni precedenti di C ++:

C ++ 14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
  return t;
}
```

Una nota sull'efficienza

Poiché la classe `std::vector` è fondamentalmente una classe che gestisce una matrice contigua allocata dinamicamente, lo stesso principio spiegato qui si applica ai vettori C ++. L'accesso al contenuto del vettore per indice è molto più efficiente quando si segue il principio dell'ordine delle righe principali. Ovviamente, ogni accesso al vettore mette anche il suo contenuto di gestione nella cache, ma come è stato discusso molte volte (in particolare qui e qui), la differenza di prestazioni per iterare su un `std::vector` rispetto a un array raw è trascurabile. Quindi lo stesso principio di efficienza per le matrici raw in C vale anche per lo `std::vector` di C ++.

Accesso agli elementi

Esistono due modi principali per accedere agli elementi in un `std::vector`

- accesso basato su indice
- iteratori

Accesso basato su indice:

Questo può essere fatto sia con l'operatore pedice `[]` , sia con la funzione membro `at()` .

Entrambi restituiscono un riferimento all'elemento nella rispettiva posizione in `std::vector` (a meno che non sia un `vector<bool>`), in modo che possa essere letto e modificato (se il vettore non è `const`).

`[]` e `at()` differiscono per il fatto che `[]` non è garantito il controllo dei limiti, mentre `at()` fa. Accedere agli elementi in cui `index < 0` o `index >= size` è un comportamento non definito per `[]` , mentre `at()` genera un'eccezione `std::out_of_range` .

Nota: gli esempi seguenti utilizzano l'inizializzazione in stile C ++ 11 per chiarezza, ma gli operatori possono essere utilizzati con tutte le versioni (a meno che non sia contrassegnato C ++ 11).

C ++ 11

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

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Poiché il metodo `at()` esegue il controllo dei limiti e può generare eccezioni, è più lento di `[]` . Questo rende `[]` codice preferito in cui la semantica dell'operazione garantisce che l'indice sia limitato. In ogni caso, gli accessi agli elementi dei vettori vengono eseguiti in tempo costante. Ciò significa che l'accesso al primo elemento del vettore ha lo stesso costo (nel tempo) dell'accesso al secondo elemento, al terzo elemento e così via.

Ad esempio, considera questo ciclo

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Qui sappiamo che la variabile indice `i` è sempre in perenne, quindi sarebbe uno spreco di cicli della CPU per verificare che `i` limiti per ogni chiamata `operator[]` .

Le funzioni membro `front()` e `back()` consentono un facile accesso di riferimento al primo e all'ultimo elemento del vettore, rispettivamente. Queste posizioni sono usate frequentemente e gli accessor speciali possono essere più leggibili delle loro alternative usando `[]` :

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose
```

```
int a = v.front();   // a is 4, v.front() is equivalent to v[0]
v.front() = 3;       // v now contains {3, 5, 6}
int b = v.back();    // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7;        // v now contains {3, 5, 7}
```

Nota : è un comportamento indefinito invocare `front()` o `back()` su un vettore vuoto. È necessario verificare che il contenitore non sia vuoto usando la funzione membro `empty()` (che controlla se il contenitore è vuoto) prima di chiamare `front()` o `back()` . Segue un semplice esempio dell'uso di "empty ()" per verificare un vettore vuoto:

```
int main ()
{
  std::vector<int> v;
  int sum (0);

  for (int i=1;i<=10;i++) v.push_back(i);//create and initialize the vector

  while (!v.empty())//loop through until the vector tests to be empty
  {
     sum += v.back();//keep a running total
     v.pop_back();//pop out the element which removes it from the vector
  }

  std::cout << "total: " << sum << '\n';//output the total to the user

  return 0;
}
```

L'esempio sopra crea un vettore con una sequenza di numeri da 1 a 10. Quindi apre gli elementi del vettore fino a quando il vettore non è vuoto (usando 'empty ()') per impedire un comportamento indefinito. Quindi la somma dei numeri nel vettore viene calcolata e visualizzata all'utente.

C ++ 11

Il metodo `data()` restituisce un puntatore alla memoria grezza utilizzata da `std::vector` per memorizzare internamente i suoi elementi. Questo è più spesso usato quando si passa il dato vettoriale al codice legacy che si aspetta un array in stile C.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4;            // v now contains {4, 2, 3, 4}
++p;               // p points to 2
*p = 3;            // v now contains {4, 3, 3, 4}
p[1] = 2;          // v now contains {4, 3, 2, 4}
*(p + 2) = 1;      // v now contains {4, 3, 2, 1}
```

C ++ 11

Prima di C ++ 11, il metodo `data()` può essere simulato chiamando `front()` e prendendo l'indirizzo del valore restituito:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

Questo funziona perché i vettori sono sempre garantiti per memorizzare i loro elementi in posizioni di memoria contigue, assumendo che il contenuto del vettore non sovrascriva l' `operator&` unario `operator&` . Se lo fa, dovrai implementare nuovamente `std::addressof` in pre-C ++ 11. Presume anche che il vettore non sia vuoto.

iteratori:

Gli iteratori sono spiegati in modo più dettagliato nell'esempio "Iterating over `std::vector` " e l'articolo Iterators . In breve, agiscono in modo simile ai puntatori agli elementi del vettore:

C ++ 11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;         // i is 4
++it;
i = *it;             // i is 5
*it = 6;             // v contains { 4, 6, 6 }
auto e = v.end();    // e points to the element after the end of v. It can be
                     // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();       // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();       // true
```

È coerente con lo standard che gli iteratori di `std::vector<T>` *siano in* realtà `T*` s, ma la maggior parte delle librerie standard non lo fa. Non farlo migliora entrambi i messaggi di errore, cattura codice non portabile e può essere usato per strumentare gli iteratori con controlli di debug in build non a rilascio. Quindi, in build di rilascio, la classe che avvolge il puntatore sottostante viene ottimizzata.

È possibile mantenere un riferimento o un puntatore a un elemento di un vettore per l'accesso indiretto. Questi riferimenti o puntatori agli elementi nel `vector` rimangono stabili e l'accesso rimane definito a meno che non si aggiungano / rimuovano elementi ao prima dell'elemento nel `vector` o si provochi la modifica della capacità del `vector` . Questa è la stessa regola per invalidare gli iteratori.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;      // p points to 2
v.insert(v.begin(), 0);     // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;           // p points to 1
v.reserve(10);              // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;           // p points to 1
v.erase(v.begin());         // p is now invalid, accessing *p is a undefined behavior.
```

Usando std :: vector come array C

Esistono diversi modi per utilizzare un `std::vector` come un array C (ad esempio, per la

compatibilità con le librerie C). Questo è possibile perché gli elementi in un vettore sono memorizzati in modo contiguo.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

A differenza delle soluzioni basate su precedenti standard C ++ (vedi sotto), la funzione membro `.data()` può anche essere applicata ai vettori vuoti, perché in questo caso non causa un comportamento indefinito.

Prima di C ++ 11, dovresti prendere l'indirizzo del primo elemento del vettore per ottenere un puntatore equivalente, se il vettore non è vuoto, questi due metodi sono intercambiabili:

```
int* p = &v[0];      // combine subscript operator and 0 literal

int* p = &v.front(); // explicitly reference the first element
```

Nota: se il vettore è vuoto, `v[0]` e `v.front()` non sono definiti e non possono essere utilizzati.

Quando si memorizza l'indirizzo di base dei dati del vettore, si noti che molte operazioni (come `push_back` , `resize` , ecc.) Possono modificare la posizione della memoria di dati del vettore, invalidando così i puntatori di dati precedenti . Per esempio:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Iterator / Pointer Invalidation

Iteratori e puntatori che puntano in un `std::vector` possono diventare non validi, ma solo quando si eseguono determinate operazioni. L'utilizzo di iteratori / puntatori non validi comporterà un comportamento indefinito.

Le operazioni che invalidano gli iteratori / i puntatori includono:

- Qualsiasi operazione di inserimento che modifichi la `capacity` del `vector` invaliderà *tutti gli* iteratori / puntatori:

```
vector<int> v(5); // Vector has a size of 5; capacity is unknown.
int *p1 = &v[0];
v.push_back(2);   // p1 may have been invalidated, since the capacity was unknown.

v.reserve(20);    // Capacity is now at least 20.
int *p2 = &v[0];
v.push_back(4);   // p2 is *not* invalidated, since the size of `v` is now 7.
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the
                          // requested capacity of 20, so `p2` is (probably) invalidated.
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // Iterators were invalidated.
```

- Qualsiasi operazione di inserimento, che non aumenta la capacità, invalida ancora iteratori / puntatori che puntano agli elementi nella posizione di inserimento e oltre. Questo include l'iteratore `end` :

```
vector<int> v(5);
v.reserve(20);                  // Capacity is at least 20.
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity
                               // did not change, `p1` remains valid.
int *p3 = &v[v.size() - 1];
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Qualsiasi operazione di rimozione invaliderà iteratori / puntatori che puntano agli elementi rimossi e a qualsiasi elemento oltre gli elementi rimossi. Questo include l'iteratore `end` :

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (copia, sposta o altrimenti) e `clear()` invalideranno tutti gli iteratori / puntatori che puntano nel vettore.

Eliminazione di elementi

Eliminazione dell'ultimo elemento:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back();                           // v becomes {1, 2}
```

Eliminazione di tutti gli elementi:

```
std::vector<int> v{ 1, 2, 3 };
v.clear();                              // v becomes an empty vector
```

Eliminazione elemento per indice:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3);                 // v becomes {1, 2, 3, 5, 6}
```

Nota: per un `vector` cancella un elemento che non è l'ultimo elemento, tutti gli elementi oltre l'elemento eliminato devono essere copiati o spostati per riempire il vuoto, vedere la nota sotto e std :: list .

Eliminazione di tutti gli elementi in un intervallo:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5);  // v becomes {1, 6}
```

Nota: i metodi sopra riportati non modificano la capacità del vettore, ma solo la dimensione. Vedi Dimensioni e capacità del vettore .

Il metodo di `erase` , che rimuove un intervallo di elementi, viene spesso utilizzato come parte dell'idioma di **cancellazione-rimozione** . Cioè, prima `std::remove` mosse alcuni elementi alla fine del vettore, e quindi `erase` costolette di dosso. Si tratta di un'operazione relativamente inefficiente per qualsiasi indice inferiore all'ultimo indice del vettore, in quanto tutti gli elementi dopo i segmenti cancellati devono essere trasferiti in nuove posizioni. Per applicazioni critiche che richiedono la rimozione efficiente di elementi arbitrari in un contenitore, vedere std :: list .

Eliminazione di elementi in base al valore:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Eliminazione di elementi in base alla condizione:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Eliminazione di elementi di lambda, senza creare una funzione di predicato aggiuntiva

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
     [](auto& element){return element > 3;} ), v.end()
);
```

Eliminazione di elementi in base alla condizione da un ciclo:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
    else
        ++it;             // manually set 'it' to the next element in v
}
```

Mentre è importante *non* incrementare `it` in caso di cancellazione, si dovrebbe considerare l'utilizzo di un metodo diverso quando poi cancellare più volte in un ciclo. Considerare `remove_if` per un modo più efficiente.

Eliminazione di elementi in base alla condizione da un ciclo inverso:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
    } else
        ++it;
}
```

Nota alcuni punti per il ciclo precedente:

- Dato un iteratore inverso `it` punta a qualche elemento, il metodo `base` fornisce l'iteratore regolare (non inverso) che punta allo stesso elemento.
- `vector::erase(iterator)` cancella l'elemento puntato da un iteratore e restituisce un iteratore all'elemento che ha seguito l'elemento specificato.

- `reverse_iterator::reverse_iterator(iterator)` costruisce un iteratore inverso da un iteratore.

Mettere tutto, la linea `it = rev_itr(v.erase(it.base()))` dice: prendere l'iteratore inverso `it` , hanno `v` cancellare l'elemento puntato dal suo iteratore regolare; prendere l'iteratore risultante, costruire un iteratore inverso da esso, e assegnare all'iteratore contrario `it` .

L'eliminazione di tutti gli elementi mediante `v.clear()` non libera la memoria (la `capacity()` del vettore rimane invariata). Per recuperare spazio, utilizzare:

```
std::vector<int>().swap(v);
```

C ++ 11

`shrink_to_fit()` libera la capacità vettoriale inutilizzata:

```
v.shrink_to_fit();
```

Il `shrink_to_fit` non garantisce di reclamare realmente lo spazio, ma la maggior parte delle implementazioni attuali lo fanno.

Trovare un elemento in std :: vector

La funzione `std::find` , definita nell'intestazione `<algorithm>` , può essere utilizzata per trovare un elemento in un `std::vector` .

`std::find` utilizza l' `operator==` per confrontare gli elementi per l'uguaglianza. Restituisce un iteratore al primo elemento nell'intervallo che è uguale al valore.

Se l'elemento in questione non viene trovato, `std::find` restituisce `std::vector::end` (o `std::vector::cend` se il vettore è `const`).

C ++ 11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

C ++ 11

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

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1
```

```
auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Se è necessario eseguire molte ricerche in un vettore di grandi dimensioni, è consigliabile prendere in considerazione l'ordinamento del vettore prima di utilizzare l'algoritmo `binary_search` .

Per trovare il primo elemento in un vettore che soddisfa una condizione, può essere utilizzato `std::find_if` . Oltre ai due parametri dati a `std::find` , `std::find_if` accetta un terzo argomento che è un oggetto funzione o un puntatore a funzione di un predicato. Il predicato dovrebbe accettare un elemento dal contenitore come argomento e restituire un valore convertibile in `bool` , senza modificare il contenitore:

C ++ 11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

C ++ 11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Convertire una matrice in std :: vector

Un array può essere facilmente convertito in un file `std::vector` usando `std::begin` e `std::end` :

C ++ 11

```
int values[5] = { 1, 2, 3, 4, 5 }; // source array

std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector

for(auto &x: v)
    std::cout << x << " ";
std::cout << std::endl;
```

1 2 3 4 5

```
int main(int argc, char* argv[]) {
    // convert main arguments into a vector of strings.
    std::vector<std::string>  args(argv, argv + argc);
}
```

È anche possibile utilizzare un inizializzatore_list C ++ 11 <> per inizializzare il vettore in una sola volta

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

vettore : L'eccezione a così tante, tante regole

Lo standard (sezione 23.3.7) specifica che viene fornita una specializzazione del `vector<bool>` , che ottimizza lo spazio impacchettando i valori `bool` , in modo che ciascuno occupi solo un bit. Poiché i bit non sono indirizzabili in C ++, ciò significa che diversi requisiti sul `vector` non sono posti sul `vector<bool>` :

- Non è necessario che i dati memorizzati siano contigui, quindi un `vector<bool>` non può essere passato a un'API C che si aspetta un array `bool` .
- `at()` , `operator []` e dereferenziazione di iteratori non restituiscono un riferimento a `bool` . Piuttosto restituiscono un oggetto proxy che simula (imperfettamente) un riferimento a un `bool` sovraccaricando i suoi operatori di assegnazione. Ad esempio, il seguente codice potrebbe non essere valido per `std::vector<bool>` , perché il dereferenziamento di un iteratore non restituisce un riferimento:

C ++ 11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

Allo stesso modo, le funzioni che prevedono un `bool&` argomento non possono essere usate con il risultato `operator []` o `at()` applicato al `vector<bool>` , o con il risultato del dereferenziamento del suo iteratore:

```
  void f(bool& b);
  f(v[0]);             // error
  f(*v.begin());       // error
```

L'implementazione di `std::vector<bool>` dipende sia dal compilatore che dall'architettura. La specializzazione viene implementata comprimendo `n` Booleani nella sezione di memoria più bassa indirizzabile. Qui, `n` è la dimensione in bit della memoria indirizzabile più bassa. Nella maggior parte dei sistemi moderni questo è 1 byte o 8 bit. Ciò significa che un byte può memorizzare 8 valori booleani. Questo è un miglioramento rispetto all'implementazione tradizionale in cui 1 valore booleano è memorizzato in 1 byte di memoria.

Nota: l'esempio seguente mostra i possibili valori bit a bit dei singoli byte in un `vector<bool>` tradizionale vs ottimizzato `vector<bool>` . Questo non sarà sempre valido in tutte le architetture. È, tuttavia, un buon modo di visualizzare l'ottimizzazione. Negli esempi seguenti un byte è rappresentato come [x, x, x, x, x, x, x, x].

Tradizionale *`std::vector<char>` memorizzando 8 valori booleani:*

C ++ 11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Rappresentazione bit a bit:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Specializzato *`std::vector<bool>` memorizzando 8 valori booleani:*

C ++ 11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Rappresentazione bit a bit:

```
[1,0,0,0,1,0,1,1]
```

Nota nell'esempio sopra, che nella versione tradizionale di `std::vector<bool>` , 8 valori booleani occupano 8 byte di memoria, mentre nella versione ottimizzata di `std::vector<bool>` , usano solo 1 byte di memoria. Questo è un miglioramento significativo sull'utilizzo della memoria. Se è necessario passare un `vector<bool>` a un'API in stile C, potrebbe essere necessario copiare i valori in un array o trovare un modo migliore di utilizzare l'API, se la memoria e le prestazioni sono a rischio.

Dimensioni e capacità del vettore

La dimensione del vettore è semplicemente il numero di elementi nel vettore:

1. La **dimensione del** vettore corrente viene interrogata dalla funzione membro `size()` . La funzione di convenienza `empty()` restituisce `true` se la dimensione è 0:

   ```
   vector<int> v = { 1, 2, 3 }; // size is 3
   ```

```
const vector<int>::size_type size = v.size();
cout << size << endl; // prints 3
cout << boolalpha << v.empty() << endl; // prints false
```

2. Il vettore predefinito costruito inizia con una dimensione di 0:

```
vector<int> v; // size is 0
cout << v.size() << endl; // prints 0
```

3. L'aggiunta di `N` elementi al vettore aumenta la **dimensione** di `N` (ad es. Con le `push_back()` , `insert()` o `resize()`).

4. La rimozione di `N` elementi dal vettore diminuisce le **dimensioni** di `N` (ad es. Con `pop_back()` , `erase()` o `clear()`).

5. Vector ha un limite superiore specifico per l'implementazione sulla sua dimensione, ma è probabile che si esaurisca la RAM prima di raggiungerla:

```
vector<int> v;
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Errore comune: la **dimensione** non è necessariamente (o anche di solito) `int` :

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

La capacità del vettore differisce dalle **dimensioni** . Mentre la **dimensione** è semplicemente il numero di elementi attualmente presenti nel vettore, la **capacità** è per il numero di elementi per cui è stata allocata / riservata la memoria. Ciò è utile perché allocazioni troppo frequenti (ri) di dimensioni troppo grandi possono essere costose.

1. La **capacità** vettoriale corrente viene interrogata dalla funzione membro `capacity()` . **La capacità** è sempre maggiore o uguale alla **taglia** :

```
vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
const vector<int>::size_type capacity = v.capacity();
cout << capacity << endl; // prints number >= 3
```

2. È possibile prenotare manualmente la capacità mediante la funzione di `reserve( N )` (cambia la capacità del vettore in `N`):

```
// !!!bad!!!evil!!!
vector<int> v_bad;
for( int i = 0; i < 10000; ++i ) {
```

```
    v_bad.push_back( i ); // possibly lot of reallocations
}

// good
vector<int> v_good;
v_good.reserve( 10000 ); // good! only one allocation
for( int i = 0; i < 10000; ++i ) {
    v_good.push_back( i ); // no allocations needed anymore
}
```

3. Puoi richiedere il rilascio della capacità in eccesso da `shrink_to_fit()` (ma l'implementazione non deve `shrink_to_fit()`). Questo è utile per conservare la memoria utilizzata:

```
vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but
possibly false)
```

Il vettore in parte gestisce automaticamente la capacità, quando aggiungi elementi che potrebbero decidere di crescere. Agli implementatori piace usare 2 o 1.5 per il fattore di crescita (il rapporto aureo sarebbe il valore ideale - ma non è pratico a causa del numero razionale). D'altra parte il vettore di solito non si riduce automaticamente. Per esempio:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!

v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Vettori concatenanti

Uno `std::vector` può essere aggiunto a un altro utilizzando la funzione membro `insert()` :

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

Tuttavia, questa soluzione non riesce se si tenta di aggiungere un vettore a se stesso, poiché lo standard specifica che gli iteratori dati a `insert()` non devono essere dello stesso intervallo degli elementi dell'oggetto destinatario.

c ++ 11

Invece di utilizzare le funzioni membro del vettore, è possibile utilizzare le funzioni `std::begin()` e `std::end()` :

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

Questa è una soluzione più generale, ad esempio, perché `b` può anche essere una matrice. Tuttavia, anche questa soluzione non ti consente di aggiungere un vettore a se stesso.

Se l'ordine degli elementi nel vettore ricevente non ha importanza, considerando il numero di elementi in ciascun vettore è possibile evitare operazioni di copia non necessarie:

```
if (b.size() < a.size())
  a.insert(a.end(), b.begin(), b.end());
else
  b.insert(b.end(), a.begin(), a.end());
```

Ridurre la capacità di un vettore

Un `std::vector` aumenta automaticamente la sua capacità al momento dell'inserimento, ma non riduce mai la sua capacità dopo la rimozione degli elementi.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
v.erase(v.begin() + 50, v.end());  // Reduces the size from 100 to 50 (v.size() == 50),
                                   // but not the capacity (v.capacity() == old_capacity)
```

Per ridurre la sua capacità, possiamo copiare il contenuto di un vettore in un nuovo vettore temporaneo. Il nuovo vettore avrà la capacità minima necessaria per memorizzare tutti gli elementi del vettore originale. Se la riduzione delle dimensioni del vettore originale era significativa, è probabile che la riduzione della capacità per il nuovo vettore sia significativa. Possiamo quindi scambiare il vettore originale con quello temporaneo per mantenere la sua capacità minima:

```
std::vector<int>(v).swap(v);
```

C ++ 11

In C ++ 11 possiamo usare la funzione membro `shrink_to_fit()` per un effetto simile:

```
v.shrink_to_fit();
```

Nota: la funzione membro `shrink_to_fit()` è una richiesta e non garantisce la riduzione della capacità.

Utilizzo di un vettore ordinato per la ricerca rapida degli elementi

L'intestazione `<algorithm>` fornisce un numero di funzioni utili per lavorare con vettori ordinati.

Un importante prerequisito per lavorare con vettori ordinati è che i valori memorizzati sono paragonabili a `<` .

Un vettore non ordinato può essere ordinato usando la funzione `std::sort()` :

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

I vettori ordinati consentono una ricerca efficiente degli elementi utilizzando la funzione `std::lower_bound()` . A differenza di `std::find()` , questo esegue un'efficiente ricerca binaria sul vettore. Lo svantaggio è che dà solo risultati validi per gli intervalli di input ordinati:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Nota: se il valore richiesto non è parte del vettore, `std::lower_bound()` restituirà un iteratore al primo elemento che è *maggiore* del valore richiesto. Questo comportamento ci consente di inserire un nuovo elemento nella sua giusta posizione in un vettore già ordinato:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

Se è necessario inserire molti elementi contemporaneamente, potrebbe essere più efficiente chiamare prima `push_back()` per tutti e quindi chiamare `std::sort()` una volta che tutti gli elementi sono stati inseriti. In questo caso, l'aumento del costo dell'ordinamento può ripagare il costo ridotto di inserire nuovi elementi alla fine del vettore e non nel mezzo.

Se il vettore contiene più elementi dello stesso valore, `std::lower_bound()` tenterà di restituire un iteratore al primo elemento del valore cercato. Tuttavia, se è necessario inserire un nuovo elemento *dopo* l'ultimo elemento del valore cercato, è necessario utilizzare la funzione `std::upper_bound()` quanto ciò causerà meno spostamento di elementi:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

Se sono necessari entrambi gli iteratori del limite superiore e del limite inferiore, è possibile utilizzare la funzione `std::equal_range()` per recuperarli entrambi in modo efficiente con una chiamata:

```
std::pair<std::vector<int>::iterator,
          std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

Per verificare se un elemento esiste in un vettore ordinato (sebbene non specifico per i vettori), puoi utilizzare la funzione `std::binary_search()` :

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Funzioni che restituiscono vettori di grandi dimensioni

C ++ 11

In C ++ 11, i compilatori sono obbligati a spostarsi implicitamente da una variabile locale che viene restituita. Inoltre, la maggior parte dei compilatori può eseguire copie elision in molti casi ed elidere la mossa del tutto. Di conseguenza, la restituzione di oggetti di grandi dimensioni che possono essere spostati a basso costo non richiede più una gestione speciale:

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

// If the compiler is unable to perform named return value optimization (NRVO)
// and elide the move altogether, it is required to move from v into the return value.
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}

int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // print vector
    for (auto value : vec)
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "

    std::cout << std::endl;

    return 0;
}
```

C ++ 11

Prima del C ++ 11, la copia elision era già consentita e implementata dalla maggior parte dei compilatori. Tuttavia, a causa dell'assenza di semantica di spostamento, nel codice o nel codice legacy che deve essere compilato con versioni di compilatore precedenti che non implementano questa ottimizzazione, è possibile trovare i vettori passati come argomenti di output per impedire la copia non necessaria:

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

// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
```

```
}

int main() {// declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}
```

Trova l'elemento massimo e minimo e l'indice rispettivo in un vettore

Per trovare l'elemento più grande o più piccolo memorizzato in un vettore, puoi usare rispettivamente i metodi `std::max_element` e `std::min_element` . Questi metodi sono definiti nell'intestazione `<algorithm>` . Se diversi elementi sono equivalenti all'elemento più grande (il più piccolo), i metodi restituiscono l'iteratore al primo di tali elementi. Restituisce `v.end()` per i vettori vuoti.

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(),v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(),v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Produzione:

> maxElementIndex: 3, maxElement: 10
> minElementIndex: 1, minElement: 2

C ++ 11

L'elemento minimo e massimo in un vettore può essere richiamato allo stesso tempo usando il metodo `std::minmax_element` , che è anche definito nell'intestazione `<algorithm>` :

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Produzione:

> elemento minimo: 2
> elemento massimo: 10

Matrici che usano i vettori

I vettori possono essere usati come una matrice 2D definendoli come vettori di vettori.

Una matrice con 3 righe e 4 colonne con ogni cella inizializzata come 0 può essere definita come:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

C ++ 11

La sintassi per inizializzarli usando gli elenchi di inizializzatori o altrimenti sono simili a quelli di un vettore normale.

```
  std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                           {4,5,6,7},
                                           {8,9,10,11}
                                         };
```

È possibile accedere ai valori in tale vettore in modo simile a un array 2D

```
int var = matrix[0][2];
```

L'iterazione su tutta la matrice è simile a quella di un vettore normale ma con una dimensione extra.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

C ++ 11

```
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

Un vettore di vettori è un modo conveniente per rappresentare una matrice ma non è il più efficiente: i singoli vettori sono sparsi nella memoria e la struttura dei dati non è cache-friendly.

Inoltre, in una matrice appropriata, la lunghezza di ogni riga deve essere uguale (non è il caso di un vettore di vettori). La flessibilità aggiuntiva può essere fonte di errori.

Leggi std :: vector online: https://riptutorial.com/it/cplusplus/topic/511/std----vector

Capitolo 128: Stream C ++

Osservazioni

Il costruttore predefinito di `std::istream_iterator` costruisce un iteratore che rappresenta la fine del flusso. Quindi, `std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(), ....` significa copiare dalla posizione corrente in `ifs` fino alla fine.

Examples

Stream di stringhe

`std::ostringstream` è una classe i cui oggetti hanno l'aspetto di un flusso di output (ovvero, è possibile scrivere su di essi tramite l' `operator<<`), ma in realtà memorizzano i risultati di scrittura e li forniscono sotto forma di flusso.

Considera il seguente codice breve:

```
#include <sstream>
#include <string>


using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

La linea

```
ostringstream ss;
```

crea un tale oggetto. Questo oggetto viene prima manipolato come un normale flusso:

```
ss << "the answer to everything is " << 42;
```

In seguito, tuttavia, lo stream risultante può essere ottenuto in questo modo:

```
const string result = ss.str();
```

(il `result` della stringa sarà uguale a `"the answer to everything is 42"`).

Ciò è utile soprattutto quando abbiamo una classe per la quale è stata definita la serializzazione del flusso e per la quale vogliamo una stringa. Ad esempio, supponiamo di avere qualche classe

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

Per ottenere la rappresentazione della stringa di un oggetto `foo` ,

```
foo f;
```

potremmo usare

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Quindi il `result` contiene la rappresentazione della stringa dell'oggetto `foo` .

Leggere un file fino alla fine

Lettura di un file di testo riga per riga

Un modo corretto di leggere un file di testo riga per riga fino alla fine di solito non è chiaro dalla documentazione di `ifstream` . Consideriamo alcuni errori comuni fatti dai programmatori C ++ per principianti e un modo corretto di leggere il file.

Linee senza caratteri di spazi bianchi

Per semplicità, supponiamo che ogni riga del file non contenga simboli di spazi vuoti.

`ifstream` ha `operator bool()` , che restituisce true quando uno stream non ha errori ed è pronto per essere letto. Inoltre, `ifstream::operator >>` restituisce un riferimento al flusso stesso, così possiamo leggere e controllare EOF (così come gli errori) in una riga con una sintassi molto elegante:

```
std::ifstream ifs("1.txt");
std::string s;
while(ifs >> s) {
    std::cout << s << std::endl;
}
```

Linee con caratteri di spaziatura

`ifstream::operator >>` legge lo stream fino a quando non si verifica un carattere di spazio `ifstream::operator >>` , quindi il codice sopra riportato stamperà le parole da una riga su righe separate. Per leggere tutto fino alla fine della riga, usa `std::getline` invece di `ifstream::operator`

`>>` . `getline` restituisce il riferimento al thread con cui ha lavorato, quindi la stessa sintassi è disponibile:

```
while(std::getline(ifs, s)) {
    std::cout << s << std::endl;
}
```

Ovviamente, `std::getline` dovrebbe essere usato anche per leggere un file a riga singola fino alla fine.

Lettura di un file in un buffer in una volta

Infine, leggiamo il file dall'inizio alla fine senza fermarci a nessun carattere, inclusi gli spazi bianchi e le nuove righe. Se sappiamo che le dimensioni esatte del file o il limite superiore della lunghezza sono accettabili, possiamo ridimensionare la stringa e quindi leggere:

```
s.resize(100);
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    s.begin());
```

Altrimenti, dobbiamo inserire ogni carattere alla fine della stringa, quindi `std::back_inserter` è ciò di cui abbiamo bisogno:

```
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::back_inserter(s));
```

In alternativa, è possibile inizializzare una raccolta con dati di flusso, utilizzando un costruttore con argomenti dell'intervallo iteratore:

```
std::vector v(std::istreambuf_iterator<char>(ifs),
    std::istreambuf_iterator<char>());
```

Nota che questi esempi sono applicabili anche se `ifs` è aperto come file binario:

```
std::ifstream ifs("1.txt", std::ios::binary);
```

Copia di flussi

Un file può essere copiato su un altro file con stream e iteratori:

```
std::ofstream ofs("out.file");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(ofs));
ofs.close();
```

o reindirizzato a qualsiasi altro tipo di flusso con un'interfaccia compatibile. Ad esempio Stream di

rete Boost.Asio:

```
boost::asio::ip::tcp::iostream stream;
stream.connect("example.com", "http");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(stream));
stream.close();
```

Array

Poiché gli iteratori possono essere pensati come una generalizzazione dei puntatori, i contenitori STL negli esempi sopra possono essere sostituiti con matrici native. Ecco come analizzare i numeri nell'array:

```
int arr[100];
std::copy(std::istream_iterator<char>(ifs), std::istream_iterator<char>(), arr);
```

Attenzione all'overflow del buffer, poiché gli array non possono essere ridimensionati al volo dopo che sono stati allocati. Ad esempio, se il codice sopra verrà alimentato con un file che contiene più di 100 numeri interi, tenterà di scrivere all'esterno della matrice e di eseguire un comportamento indefinito.

Stampa di collezioni con iostream

Stampa di base

`std::ostream_iterator` consente di stampare il contenuto di un container STL su qualsiasi flusso di output senza loop espliciti. Il secondo argomento del costruttore `std::ostream_iterator` imposta il delimitatore. Ad esempio, il seguente codice:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

stamperà

```
1 ! 2 ! 3 ! 4 !
```

Cast di tipo implicito

`std::ostream_iterator` consente di eseguire il cast del tipo di contenuto del contenitore implicitamente. Ad esempio, sintonizziamo `std::cout` per stampare valori a virgola mobile con 3 cifre dopo il punto decimale:

```
std::cout << std::setprecision(3);
```

```
std::fixed(std::cout);
```

e istanziare `std::ostream_iterator` con `float` , mentre i valori contenuti rimangono `int` :

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

quindi il codice sopra produce

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

nonostante `std::vector` detenga `int` s.

Generazione e trasformazione

`std::generate` funzioni `std::generate` , `std::generate_n` e `std::transform` forniscono uno strumento molto potente per la manipolazione dei dati al volo. Ad esempio, avendo un vettore:

```
std::vector<int> v = {1,2,3,4,8,16};
```

possiamo facilmente stampare il valore booleano dell'istruzione "x is even" per ogni elemento:

```
std::boolalpha(std::cout); // print booleans alphabetically
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),
[](int val) {
    return (val % 2) == 0;
});
```

o stampare l'elemento quadrato:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Stampa di numeri casuali delimitati da spazio:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Array

Come nella sezione sulla lettura dei file di testo, quasi tutte queste considerazioni possono essere applicate agli array nativi. Ad esempio, stampiamo i valori al quadrato da un array nativo:

```
int v[] = {1,2,3,4,8,16};
```

```
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Analisi dei file

Analisi dei file nei contenitori STL

`istream_iterator` s sono molto utili per leggere sequenze di numeri o altri dati analizzabili in contenitori STL senza loop espliciti nel codice.

Utilizzando la dimensione del contenitore esplicita:

```
std::vector<int> v(100);
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    v.begin());
```

o con l'inserimento di iteratore:

```
std::vector<int> v;
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    std::back_inserter(v));
```

Si noti che i numeri nel file di input possono essere divisi per qualsiasi numero di caratteri di spazi vuoti e nuove righe.

Analisi di tabelle di testo eterogenee

Come `istream::operator>>` legge il testo fino ad un simbolo spazio bianco, può essere utilizzato in `while` condizioni per analizzare tabelle di dati complessi. Ad esempio, se abbiamo un file con due numeri reali seguiti da una stringa (senza spazi) su ogni riga:

```
1.12 3.14 foo
2.1 2.2 barr
```

può essere analizzato in questo modo:

```
std::string s;
double a, b;
while(ifs >> a >> b >> s) {
    std::cout << a << " " << b << " " << s << std::endl;
}
```

Trasformazione

Qualsiasi funzione di manipolazione degli intervalli può essere utilizzata con `std::istream_iterator` intervalli `std::istream_iterator` . Uno di questi è `std::transform` , che consente di elaborare i dati al volo. Ad esempio, leggiamo valori interi, moltiplicandoli per 3,14 e archiviamo il risultato in un contenitore a virgola mobile:

```
std::vector<double> v(100);
std::transform(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
v.begin(),
[](int val) {
    return val * 3.14;
});
```

Leggi Stream C ++ online: https://riptutorial.com/it/cplusplus/topic/7660/stream-c-plusplus

Capitolo 129: Strumenti e tecniche di debug in C ++ per debugging e debug

introduzione

Un sacco di tempo dagli sviluppatori C ++ è dedicato al debugging. Questo argomento è pensato per aiutare con questo compito e dare ispirazione per le tecniche. Non aspettarti un ampio elenco di problemi e soluzioni risolti dagli strumenti o un manuale sugli strumenti menzionati.

Osservazioni

Questo argomento non è ancora completo, sarebbero utili esempi su tecniche / strumenti seguenti:

- Menzione più strumenti di analisi statica
- Strumenti di strumentazione binaria (come UBSan, TSan, MSan, ESan ...)
- Indurimento (CFI ...)
- fuzzing

Examples

Il mio programma C ++ termina con segfault - valgrind

Diamo un programma fallito di base:

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p3 << std::endl;
    }
}

int main() {
    fail();
}
```

Costruiscila (aggiungi -g per includere le informazioni di debug):

```
g++ -g -o main main.cpp
```

Correre:

```
$ ./main
Segmentation fault (core dumped)
$
```

Facciamo il debug con valgrind:

```
$ valgrind ./main
==8515== Memcheck, a memory error detector
==8515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8515== Command: ./main
==8515==
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==8515==
==8515==
==8515== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==8515==  Access not within mapped region at address 0x0
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  If you believe this happened as a result of a stack
==8515==  overflow in your program's main thread (unlikely but
==8515==  possible), you can try to increase the size of the
==8515==  main thread stack using the --main-stacksize= flag.
==8515==  The main thread stack size used in this run was 8388608.
==8515==
==8515== HEAP SUMMARY:
==8515==     in use at exit: 72,704 bytes in 1 blocks
==8515==   total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated
==8515==
==8515== LEAK SUMMARY:
==8515==    definitely lost: 0 bytes in 0 blocks
==8515==    indirectly lost: 0 bytes in 0 blocks
==8515==      possibly lost: 0 bytes in 0 blocks
==8515==    still reachable: 72,704 bytes in 1 blocks
==8515==         suppressed: 0 bytes in 0 blocks
==8515== Rerun with --leak-check=full to see details of leaked memory
==8515==
==8515== For counts of detected and suppressed errors, rerun with: -v
==8515== Use --track-origins=yes to see where uninitialised values come from
==8515== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
$
```

Per prima cosa ci concentriamo su questo blocco:

```
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

La prima riga ci dice che il segfault è causato dalla lettura di 4 byte. La seconda e la terza riga sono chiamate stack. Significa che la lettura non valida viene eseguita nella funzione `fail()` , riga

8 di main.cpp, che è chiamata da main, riga 13 di main.cpp.

Guardando la riga 8 di main.cpp vediamo

```
std::cout << *p3 << std::endl;
```

Ma prima controlliamo il puntatore, quindi cosa c'è che non va? Consente di controllare l'altro blocco:

```
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
```

Ci dice che c'è una variabile unitaria alla linea 7 e la leggiamo:

```
if (p3) {
```

Che ci indica la linea in cui controlliamo p3 invece di p2. Ma com'è possibile che p3 non sia inizializzato? Inizializziamo per:

```
int *p3 = p1;
```

Valgrind ci consiglia di ripetere con `--track-origins=yes` , facciamolo:

```
valgrind --track-origins=yes ./main
```

L'argomento per Valgrind è appena dopo valgrind. Se lo inseriamo dopo il nostro programma, verrebbe passato al nostro programma.

L'output è quasi lo stesso, c'è solo una differenza:

```
==8517== Conditional jump or move depends on uninitialised value(s)
==8517==    at 0x400813: fail() (main.cpp:7)
==8517==    by 0x40083F: main (main.cpp:13)
==8517==  Uninitialised value was created by a stack allocation
==8517==    at 0x4007F6: fail() (main.cpp:3)
```

Il che ci dice che il valore non inizializzato che usavamo alla linea 7 è stato creato alla riga 3:

```
int *p1;
```

che ci guida al nostro puntatore non inizializzato.

Analisi segfault con GDB

Consente di utilizzare lo stesso codice di cui sopra per questo esempio.

```
#include <iostream>
```

```
void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

Per prima cosa compilarlo

```
g++ -g -o main main.cpp
```

Lo eseguiamo con gdb

```
gdb ./main
```

Ora saremo nella shell gdb. Digitare run.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400850 in fail () at debugging_with_gdb.cc:11
11            std::cout << *p2 << std::endl;
```

Vediamo che l'errore di segmentazione sta accadendo alla riga 11. Quindi l'unica variabile utilizzata in questa linea è il puntatore p2. Esaminiamo il suo contenuto digitando stampa.

```
(gdb) print p2
$1 = (int *) 0x0
```

Ora vediamo che p2 è stato inizializzato a 0x0 che sta per NULL. A questa linea, sappiamo che stiamo cercando di dereferenziare un puntatore NULL. Quindi andiamo a sistemarlo.

Codice pulito

Il debug inizia con la comprensione del codice che stai cercando di eseguire il debug.

Codice errato:

```
int main() {
    int value;
    std::vector<int> vectorToSort;
    vectorToSort.push_back(42); vectorToSort.push_back(13);
```

```
    for (int i = 52; i; i = i - 1)
    {
    vectorToSort.push_back(i *2);
    }
    /// Optimized for sorting small vectors
    if (vectorToSort.size() == 1);
    else
        {
        if (vectorToSort.size() <= 2)
            std::sort(vectorToSort.begin(), std::end(vectorToSort));
        }
    for (value : vectorToSort) std::cout << value << ' ';
return 0; }
```

Codice migliore:

```
std::vector<int> createSemiRandomData() {
    std::vector<int> data;
    data.push_back(42);
    data.push_back(13);
    for (int i = 52; i; --i)
        vectorToSort.push_back(i *2);
    return data;
}

/// Optimized for sorting small vectors
void sortVector(std::vector &v) {
    if (vectorToSort.size() == 1)
        return;
    if (vectorToSort.size() > 2)
        return;

    std::sort(vectorToSort.begin(), vectorToSort.end());
}

void printVector(const std::vector<int> &v) {
    for (auto i : v)
        std::cout << i << ' ';
}

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
 }
```

Indipendentemente dagli stili di codifica che preferisci e utilizzi, avere uno stile coerente di codifica (e formattazione) ti aiuterà a capire il codice.

Guardando il codice sopra, è possibile identificare un paio di miglioramenti per migliorare la leggibilità e la debugabilità:

L'uso di funzioni separate per azioni separate

L'uso di funzioni separate consente di saltare alcune funzioni nel debugger se non si è interessati

ai dettagli. In questo caso specifico, potresti non essere interessato alla creazione o alla stampa dei dati e vuoi solo entrare nell'ordinamento.

Un altro vantaggio è che è necessario leggere meno codice (e memorizzarlo) mentre si passa attraverso il codice. Ora devi solo leggere 3 righe di codice in `main()` per capirlo, invece dell'intera funzione.

Il terzo vantaggio è che hai semplicemente meno codice da guardare, il che aiuta un occhio esperto a individuare questo bug in pochi secondi.

Usando una formattazione / costruzioni coerenti

L'uso di formattazione e costruzioni consistenti rimuoverà il disordine dal codice, rendendo più facile concentrarsi sul codice anziché sul testo. Molte discussioni sono state alimentate dallo stile di formattazione "giusto". Indipendentemente da questo stile, avere uno stile unico e coerente nel codice migliorerà la familiarità e renderà più facile concentrarsi sul codice.

Poiché il codice di formattazione richiede molto tempo, si consiglia di utilizzare uno strumento dedicato per questo. La maggior parte degli IDE ha almeno un qualche tipo di supporto per questo e può fare una formattazione più coerente degli umani.

Potresti notare che lo stile non è limitato agli spazi e ai newline poiché non mescoliamo più lo stile libero e le funzioni membro per ottenere inizio / fine del contenitore. (`v.begin()` vs `std::end(v)`).

Fai attenzione alle parti importanti del tuo codice.

Indipendentemente dallo stile che decidi di scegliere, il codice sopra riportato contiene un paio di indicatori che potrebbero darti un suggerimento su ciò che potrebbe essere importante:

- Un commento che afferma `optimized` , questo indica alcune tecniche di fantasia
- Alcuni ritorni anticipati in `sortVector()` indicano che stiamo facendo qualcosa di speciale
- `std::ref()` indica che qualcosa sta succedendo con `sortVector()`

Conclusione

Avere un codice pulito ti aiuterà a capire il codice e ridurrà il tempo necessario per il debug. Nel secondo esempio, un revisore del codice potrebbe persino individuare il bug a prima vista, mentre il bug potrebbe essere nascosto nei dettagli nel primo. (PS: il bug è in confronto con `2`).

Analisi statica

L'analisi statica è la tecnica con cui controlla il codice per i pattern collegati ai bug noti. L'utilizzo di questa tecnica richiede meno tempo di una revisione del codice, tuttavia i suoi controlli sono limitati a quelli programmati nello strumento.

I controlli possono includere il punto e virgola errato dietro l'istruzione `if (var);` (`if (var);`) fino agli algoritmi di grafi avanzati che determinano se una variabile non è inizializzata.

Avvisi del compilatore

Abilitare l'analisi statica è facile, la versione più semplice è già integrata nel compilatore:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

Se abiliti queste opzioni, noterai che ogni compilatore troverà bug che gli altri non hanno e che otterrai errori su tecniche che potrebbero essere valide o valide in un contesto specifico. `while (staticAtomicBool);` potrebbe essere accettabile anche se `while (localBool);` non lo è.

Quindi, a differenza della revisione del codice, stai combattendo uno strumento che capisce il tuo codice, ti dice molti bug utili ea volte non sei d'accordo con te. In quest'ultimo caso, potrebbe essere necessario sopprimere l'avviso localmente.

Poiché le opzioni precedenti abilitano tutti gli avvisi, potrebbero attivare gli avvisi che non si desidera. (Perché il tuo codice dovrebbe essere compatibile con C ++ 98?) Se è così, puoi semplicemente disabilitare quell'avvertimento specifico:

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Dove gli avvertimenti del compilatore ti assistono durante lo sviluppo, rallentano la compilazione un po '. Questo è il motivo per cui potresti non voler sempre abilitarli di default. O li esegui di default o abiliti un'integrazione continua con i controlli più costosi (o tutti).

Strumenti esterni

Se si decide di avere un'integrazione continua, l'uso di altri strumenti non è così lungo. Uno strumento come clang-tidy ha una lista di controlli che copre una vasta gamma di problemi, alcuni esempi:

- Bug effettivi
  - Prevenzione dell'affettatura
  - Asserisce con effetti collaterali
- Controlli di leggibilità
  - Rientranza fuorviante
  - Controlla la denominazione dell'identificatore
- Controlli di modernizzazione
  - Usa make_unique ()
  - Usa nullptr
- Controlli sulle prestazioni
  - Trova copie non necessarie
  - Trova chiamate algoritmiche inefficienti

L'elenco potrebbe non essere così grande, poiché Clang ha già molti avvisi sul compilatore,

tuttavia ti porterà un passo avanti verso una base di codice di alta qualità.

Altri strumenti

Esistono altri strumenti con scopi simili, come:

- l'analizzatore statico di studio visivo come strumento esterno
- clazy , un plugin per compilatore Clang per il controllo del codice Qt

Conclusione

Esistono molti strumenti di analisi statica per C ++, entrambi incorporati nel compilatore come strumenti esterni. Provarle non richiede molto tempo per le configurazioni semplici e troveranno bug che potrebbero mancare nella revisione del codice.

Safe-stack (Stack corruzioni)

Le corruzioni dello stack sono fastidiosi bug da guardare. Dato che lo stack è corrotto, il debugger spesso non può darti una buona traccia di stack di dove sei e come ci sei arrivato.

È qui che entra in gioco lo stack sicuro. Invece di usare una singola pila per i tuoi thread, ne userà due: uno stack sicuro e uno stack pericoloso. Lo stack sicuro funziona esattamente come prima, tranne che alcune parti vengono spostate nella pila pericolosa.

Quali parti della pila vengono spostate?

Ogni parte che ha il potenziale di corrompere lo stack verrà spostata fuori dallo stack sicuro. Non appena una variabile nello stack viene passata per riferimento o uno prende l'indirizzo di questa variabile, il compilatore deciderà di allocarlo sul secondo stack invece che su quello sicuro.

Di conseguenza, qualsiasi operazione eseguita con quei puntatori, qualsiasi modifica apportata alla memoria (basata su quei puntatori / riferimenti) può solo influenzare la memoria nel secondo stack. Dato che non si ottiene mai un puntatore vicino allo stack sicuro, lo stack non può corrompere lo stack e il debugger può comunque leggere tutte le funzioni nello stack per dare una bella traccia.

A cosa serve effettivamente?

Lo stack sicuro non è stato inventato per darti una migliore esperienza di debug, tuttavia è un bell'effetto collaterale per i bug cattivi. Lo scopo originale è come parte del progetto Code-Pointer Integrity (CPI) , in cui si tenta di impedire l'override degli indirizzi di ritorno per prevenire l'iniezione di codice. In altre parole, cercano di impedire l'esecuzione di un codice hacker.

Per questo motivo, la funzione è stata attivata su cromo e è stato segnalato avere un sovraccarico della CPU <1%.

Come abilitarlo?

Al momento, l'opzione è disponibile solo nel compilatore clang , dove si può passare `-fsanitize=safe-stack` al compilatore. È stata presentata una proposta per implementare la stessa funzionalità in GCC.

Conclusione

Le corruzioni dello stack possono diventare più facili da eseguire il debug quando lo stack sicuro è abilitato. A causa di un sovraccarico di prestazioni ridotte, puoi anche essere attivato di default nella configurazione di build.

Leggi Strumenti e tecniche di debug in C ++ per debugging e debug online: https://riptutorial.com/it/cplusplus/topic/9814/strumenti-e-tecniche-di-debug-in-c-plusplus-per-debugging-e-debug

Capitolo 130: Strutture dati in C ++

Examples

Implementazione di liste collegate in C ++

Creazione di un nodo Elenco

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Creazione della classe List

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
    ~List();
};
```

Inserisci un nuovo nodo all'inizio dell'elenco

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Inserisci un nuovo nodo alla fine dell'elenco

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
```

```
    {
        ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

Inserisci in una posizione particolare nell'elenco

```
void List::insertAtPos(int pos,int val)
{
    listNode *newnode=new listNode(val);
    if(pos==1)
    {
        //as head
        newnode->next=this->head;
        this->head=newnode;
        return;
    }
    pos--;
    listNode *ptr=this->head;
    while(ptr!=NULL && --pos)
    {
        ptr=ptr->next;
    }
    if(ptr==NULL)
    return;//not enough elements
    newnode->next=ptr->next;
    ptr->next=newnode;
}
```

Rimozione di un nodo dall'elenco

```
void List::remove(int toBeRemoved)//removing an element
{
    if(this->head==NULL)
    return; //empty
    if(this->head->data==toBeRemoved)
    {
        //first node to be removed
        listNode *temp=this->head;
        this->head=this->head->next;
        delete(temp);
        return;
    }
    listNode *ptr=this->head;
    while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
    ptr=ptr->next;
    if(ptr->next==NULL)
    return;//not found
    listNode *temp=ptr->next;
    ptr->next=ptr->next->next;
    delete(temp);
}
```

Stampa la lista

```
void List::print()//printing the list
```

```
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" " ;
        ptr=ptr->next;
    }
    cout<<endl;
}
```

Destructor per la lista

```
 List::~List()
{
    listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

Leggi Strutture dati in C ++ online: https://riptutorial.com/it/cplusplus/topic/7485/strutture-dati-in-c-plusplus

Capitolo 131: Strutture di sincronizzazione del filo

introduzione

Lavorare con i thread potrebbe richiedere alcune tecniche di sincronizzazione se i thread interagiscono. In questo argomento, puoi trovare le diverse strutture fornite dalla libreria standard per risolvere questi problemi.

Examples

std :: shared_lock

Un shared_lock può essere utilizzato in combinazione con un lock univoco per consentire a più lettori e scrittori esclusivi.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
    public:
        string getPhoneNo( const std::string & name )
        {
            shared_lock<shared_timed_mutex> r(_protect);
            auto it =  _phonebook.find( name );
            if ( it == _phonebook.end() )
                return (*it).second;
            return "";
        }
        void addPhoneNo ( const std::string & name, const std::string & phone )
        {
            unique_lock<shared_timed_mutex> w(_protect);
            _phonebook[name] = phone;
        }

        shared_timed_mutex _protect;
        unordered_map<string,string>  _phonebook;
    };
```

std :: call_once, std :: once_flag

`std::call_once` garantisce l'esecuzione di una funzione esattamente una volta dai thread concorrenti. Getta `std::system_error` nel caso in cui non possa completare il suo compito.

Utilizzato in combinazione con s `td::once_flag` .

```
#include <mutex>
#include <iostream>

std::once_flag flag;
void do_something(){
      std::call_once(flag, [](){std::cout << "Happens once" << std::endl;});

      std::cout << "Happens every time" << std::endl;
}
```

Blocco degli oggetti per un accesso efficiente.

Spesso si desidera bloccare l'intero oggetto mentre si eseguono più operazioni su di esso. Ad esempio, se è necessario esaminare o modificare l'oggetto utilizzando gli *iteratori* . Ogni volta che è necessario chiamare più funzioni membro è generalmente più efficiente bloccare l'intero oggetto piuttosto che le singole funzioni membro.

Per esempio:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

public:
    // This returns a scoped lock that can be shared by multiple
    // readers at the same time while excluding any writers
    [[nodiscard]]
    reading_lock lock_for_reading() const { return reading_lock(mtx); }

    // This returns a scoped lock that is exclusing to one
    // writer preventing any readers
    [[nodiscard]]
    updates_lock lock_for_updates() { return updates_lock(mtx); }

    char* data() { return buf; }
    char const* data() const { return buf; }

    char* begin() { return buf; }
    char const* begin() const { return buf; }

    char* end() { return buf + sizeof(buf); }
    char const* end() const { return buf + sizeof(buf); }

    std::size_t size() const { return sizeof(buf); }

private:
    char buf[1024];
    mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

Quando si calcola un checksum, l'oggetto è bloccato per la lettura, consentendo agli altri thread che desiderano leggere dall'oggetto allo stesso tempo di farlo.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
    auto lock = buf.lock_for_reading();

    for(auto c: buf)
        sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

    return sum;
}
```

Cancellare l'oggetto aggiorna i suoi dati interni quindi deve essere fatto utilizzando un blocco di esclusione.

```
void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}
```

Quando si ottiene più di un blocco, è necessario prestare attenzione ad acquisire sempre i blocchi nello stesso ordine per tutti i fili.

```
void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}
```

nota: questo è il modo migliore per usare std :: deferred :: lock e chiamare std :: lock

std :: condition_variable_any, std :: cv_status

Una generalizzazione di `std::condition_variable` , `std::condition_variable_any` funziona con qualsiasi tipo di struttura BasicLockable.

`std::cv_status` come stato di ritorno per una variabile di condizione ha due possibili codici di ritorno:

- std :: cv_status :: no_timeout: non è stato raggiunto il timeout, la variabile di condizione è stata notificata
- std :: cv_status :: no_timeout: la variabile delle condizioni è scaduta

Leggi Strutture di sincronizzazione del filo online:
https://riptutorial.com/it/cplusplus/topic/9794/strutture-di-sincronizzazione-del-filo

Capitolo 132: Tecniche di refactoring

introduzione

Refactoring si riferisce alla modifica del codice esistente in una versione migliorata. Sebbene il refactoring venga spesso eseguito cambiando il codice per aggiungere funzionalità o correggere bug, il termine si riferisce in particolare al miglioramento del codice senza necessariamente aggiungere funzionalità o correggere bug.

Examples

Ristrutturazione a piedi attraverso

Ecco un programma che potrebbe beneficiare del refactoring. È un programma semplice che utilizza C ++ 11 che ha lo scopo di calcolare e stampare tutti i numeri primi da 1 a 100 e si basa su un programma che è stato pubblicato su CodeReview per la revisione.

```
#include <iostream>
#include <vector>
#include <cmath>

int main()
{
    int l = 100;
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < l; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                std::cout << no << "\n";
                break;
            }
        }
        if (isprime) {
            std::cout << no << " ";
            primes.push_back(no);
        }
    }
    std::cout << "\n";
}
```

L'output di questo programma è simile a questo:

```
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

La prima cosa che notiamo è che il programma non riesce a stampare `2` che è un numero primo.

Potremmo semplicemente aggiungere una riga di codice per stampare semplicemente una costante senza modificare il resto del programma, ma potrebbe essere più ordinario *rifattorizzare* il programma in modo da dividerlo in due parti: una che crea l'elenco dei numeri primi e un'altra che le stampa . Ecco come potrebbe apparire:

```
#include <iostream>
#include <vector>
#include <cmath>

std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}

int main()
{
    std::vector<int> primes = prime_list(100);
    for (std::size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << ' ';
    }
    std::cout << '\n';
}
```

Provando questa versione, vediamo che funziona davvero correttamente ora:

    2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Il prossimo passo è notare che la seconda clausola `if` non è realmente necessaria. La logica nel ciclo cerca i fattori primi di ogni dato numero fino alla radice quadrata di quel numero. Questo funziona perché se ci sono dei fattori primi di un numero almeno uno di essi deve essere inferiore o uguale alla radice quadrata di quel numero. Rielaborando solo quella funzione (il resto del programma rimane lo stesso) otteniamo questo risultato:

```
std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
```

```
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}
```

Possiamo andare oltre, cambiando i nomi delle variabili per essere un po 'più descrittivi. Ad esempio, il `primecount` non è davvero un conteggio di numeri primi. Invece è una variabile indice nel vettore dei numeri primi conosciuti. Inoltre, mentre a volte `no` viene usato come abbreviazione di "numero", nella scrittura matematica, è più comune usare `n` . Possiamo anche apportare alcune modifiche eliminando l' `break` e dichiarando le variabili più vicine al punto in cui vengono utilizzate.

```
std::vector<int> prime_list(int limit)
{
    std::vector<int> primes{2};
    for (int n = 3; n < limit; n += 2) {
        bool isprime = true;
        for (int i=0; isprime && primes[i] <= std::sqrt(n); ++i) {
            isprime &= (n % primes[i] != 0);
        }
        if (isprime) {
            primes.push_back(n);
        }
    }
    return primes;
}
```

Possiamo anche refactoring `main` usare un "range-for" per renderlo un po 'più ordinato:

```
int main()
{
    std::vector<int> primes = prime_list(100);
    for (auto p : primes) {
        std::cout << p << ' ';
    }
    std::cout << '\n';
}
```

Questo è solo un modo in cui potrebbe essere fatto il refactoring. Altri potrebbero fare scelte diverse. Tuttavia, lo scopo del refactoring rimane lo stesso, ovvero migliorare la leggibilità e possibilmente le prestazioni del codice senza necessariamente aggiungere funzionalità.

Goto Cleanup

Nelle basi di codice C ++ che erano C, si può trovare il modello `goto cleanup` . Poiché il comando `goto` rende il flusso di lavoro di una funzione più difficile da capire, questo è spesso evitato.

Spesso, può essere sostituito da dichiarazioni di ritorno, loop, funzioni. Tuttavia, con il `goto cleanup` necessario eliminare la logica di pulizia.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
       goto cleanup;  //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

In C ++ si potrebbe usare RAII per risolvere questo problema:

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
       return FALSE;  //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

Da questo punto in poi, si potrebbe continuare a rifare il codice attuale. Ad esempio, sostituendo `VectorRAII` con `std::unique_ptr` o `std::vector` .

Leggi Tecniche di refactoring online: https://riptutorial.com/it/cplusplus/topic/7600/tecniche-di-refactoring

Capitolo 133: Test unitario in C ++

introduzione

Il test delle unità è un livello nel test del software che convalida il comportamento e la correttezza delle unità di codice.

In C ++, "unità di codice" si riferiscono spesso a classi, funzioni o gruppi di entrambi. I test unitari vengono spesso eseguiti utilizzando "framework di test" specializzati o "librerie di test" che spesso utilizzano sintassi o schemi di utilizzo non banali.

Questo argomento esaminerà diverse strategie e librerie o framework di test unitari.

Examples

Google Test

Google Test è un framework di test C ++ gestito da Google. Richiede la creazione della libreria `gtest` e il suo collegamento al framework di test durante la creazione di un file di test case.

Esempio minimo

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// Build command: g++ main.cpp -lgtest
```

Catturare

Catch è una libreria di sola intestazione che consente di utilizzare sia lo stile di test di unità TDD

che BDD .

Il seguente frammento è tratto dalla pagina di documentazione di Catch a questo link :

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Convenientemente, questi test verranno riportati come segue quando si esegue:

```
Scenario: vectors can be sized and resized
     Given: A vector with some items
      When: more capacity is reserved
      Then: the capacity changes but not the size
```

Leggi Test unitario in C ++ online: https://riptutorial.com/it/cplusplus/topic/9928/test-unitario-in-c-plusplus

Capitolo 134: The This Pointer

Osservazioni

`this` puntatore è una parola chiave per C ++ quindi non è necessaria alcuna libreria per implementarlo. E non dimenticare che `this` è un puntatore! Quindi non puoi fare:

```
 this.someMember();
```

Mentre accedi alle funzioni membro o alle variabili membro dai puntatori usando il simbolo freccia `->` :

```
this->someMember();
```

Altri link utili per una migliore comprensione di `this` puntatore:

Qual è il puntatore "questo"?

http://www.geeksforgeeks.org/this-pointer-in-c/

https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm

Examples

questo puntatore

Tutte le funzioni membro non statiche hanno un parametro nascosto, un puntatore a un'istanza della classe, chiamato `this` ; questo parametro viene inserito silenziosamente all'inizio dell'elenco dei parametri e gestito interamente dal compilatore. Quando un membro della classe si accede all'interno di una funzione membro, viene silenziosamente accede attraverso `this` ; ciò consente al compilatore di utilizzare un'unica funzione membro non statica per tutte le istanze e consente a una funzione membro di chiamare polimorficamente altre funzioni membro.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
// As the constructor is responsible for creating the object, 'this' will not be "fully"
```

```
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int  ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int  ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

In un costruttore, `this` può essere tranquillamente usato per accedere (implicitamente o esplicitamente) a qualsiasi campo che è già stato inizializzato, o qualsiasi campo in una classe genitore; viceversa, (implicitamente o esplicitamente) l'accesso a tutti i campi che non sono ancora stati inizializzati, o qualsiasi campo in una classe derivata, non è sicuro (a causa della classe derivata non ancora costruita, e quindi i suoi campi non vengono né inizializzati né esistenti). È anche pericoloso chiamare le funzioni dei membri virtuali attraverso `this` nel costruttore, poiché qualsiasi funzione di classe derivata non sarà considerata (a causa della classe derivata che non è ancora stata costruita, e quindi il suo costruttore non ha ancora aggiornato il vtable).

Si noti inoltre che mentre in un costruttore, il tipo dell'oggetto è il tipo che il costruttore costruisce. Ciò vale anche se l'oggetto è dichiarato come un tipo derivato. Ad esempio, nell'esempio seguente, `ctd_good` e `ctd_bad` sono di tipo `CtorThisBase` all'interno di `CtorThisBase()` e digitano `CtorThis` all'interno di `CtorThis()` , anche se il loro tipo canonico è `CtorThisDerived` . Poiché le classi più derivate sono costruite attorno alla classe base, l'istanza passa gradualmente attraverso la gerarchia delle classi fino a quando non è un'istanza completamente costruita del tipo desiderato.

```
class CtorThisBase {
    short s;

  public:
    CtorThisBase() : s(516) {}
};

class CtorThis : public CtorThisBase {
    int i, j, k;

  public:
```

```
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

  public:
    CtorThisDerived()       : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

Con queste classi e funzioni membro:

- Nel buon costruttore, per `ctd_good` :
  - `CtorThisBase` è completamente costruito dal momento in `CtorThis` viene inserito il costruttore `CtorThis` . Pertanto, `s` è in uno stato valido durante l'inizializzazione di `i` , e può quindi essere accessibile.
  - `i` viene inizializzato prima che `j(this->i)` venga raggiunto. Pertanto, `i` è in uno stato valido durante l'inizializzazione di `j` e può quindi essere accessibile.
  - `j` viene inizializzato prima che `k(j)` venga raggiunto. Pertanto, `j` è in uno stato valido durante l'inizializzazione di `k` e può quindi essere accessibile.
- Nel costruttore `ctd_bad` , per `ctd_bad` :
  - `k` viene inizializzato dopo che `j(this->k)` viene raggiunto. Pertanto, `k` è in uno stato non valido durante l'inizializzazione di `j` , e accedervi causa un comportamento indefinito.
  - `CtorThisDerived` non viene `CtorThisDerived` fino a dopo la `CtorThis` . Pertanto, `b` è in uno stato non valido durante l'inizializzazione di `k` , e accedervi causa un comportamento indefinito.
  - L'oggetto `ctd_bad` è ancora un `CtorThis` fino a quando non lascia `CtorThis()` , e non sarà aggiornato per utilizzare `CtorThisDerived` vtable 's fino `CtorThisDerived()` . Pertanto, `virt_func()` chiamerà `CtorThis::virt_func()` , indipendentemente dal fatto che sia destinato a chiamarlo o `CtorThisDerived::virt_func()` .

Usando questo puntatore per accedere ai dati dei membri

In questo contesto, l'utilizzo di `this` puntatore non è del tutto necessario, ma renderà il codice più chiaro al lettore, indicando che una determinata funzione o variabile è un membro della classe. Un esempio in questa situazione:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
  public:
    Class();
    ~Class();
    int getPrivateNumber () const;
  private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

Guardalo in azione qui .

Usando questo puntatore per distinguere tra dati e parametri dei membri

Questa è una strategia utile per differenziare i dati dei membri dai parametri ... Prendiamo questo esempio:

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
* @class Dog
*   @member name
*       Dog's name
*   @function bark
*       Dog Barks!
*   @function getName
*       To Get Private
*       Name Variable
*/
class Dog
{
 public:
```

```
    Dog(std::string name);
    ~Dog();
    void  bark() const;
    std::string  getName() const;
 private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
    *  this->name is the
    *  name variable from
    *  the class dog . and
    *  name is from the
    *  parameter of the function
    */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
  cout << "BARK" << endl;
}

std::string  Dog::getName() const
{
    return this->name;
}


int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}
```

Puoi vedere qui nel costruttore che eseguiamo quanto segue:

```
this->name = name;
```

Qui, puoi vedere che stiamo assingendo il nome del parametro al nome della variabile privata dalla classe Dog (this-> name).

Per vedere l'output del codice precedente: http://cpp.sh/75r7

questo Pointer CV-qualificatori

`this` può anche essere qualificato cv, lo stesso di qualsiasi altro puntatore. Tuttavia, poiché `this` parametro non è elencato nell'elenco dei parametri, è necessaria una sintassi speciale; i qualificatori di cv sono elencati dopo la lista dei parametri, ma prima del corpo della funzione.

```
struct ThisCVQ {
```

```
    void no_qualifier()                {} // "this" is: ThisCVQ*
    void  c_qualifier() const          {} // "this" is: const ThisCVQ*
    void  v_qualifier() volatile       {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

Poiché `this` tratta di un parametro, è possibile sovraccaricare una funzione in base a `this` cv-qualificatore / i .

```
struct CVOverload {
    int func()                { return    3; }
    int func() const          { return   33; }
    int func() volatile       { return  333; }
    int func() const volatile { return 3333; }
};
```

Quando `this` è `const` (includendo `const volatile`), la funzione non è in grado di scrivere attraverso le variabili membro, sia implicitamente che esplicitamente. L'unica eccezione è rappresentata dalle variabili dei membri `mutable` , che possono essere scritte indipendentemente dalla costanza. A causa di ciò, `const` viene utilizzato per indicare che la funzione membro non modifica lo stato logico dell'oggetto (il modo in cui l'oggetto appare al mondo esterno), anche se modifica lo stato fisico (il modo in cui l'oggetto appare sotto la cappa).

> Lo stato logico è il modo in cui l'oggetto appare agli osservatori esterni. Non è direttamente legato allo stato fisico, anzi, potrebbe non essere nemmeno memorizzato come stato fisico. Finché gli osservatori esterni non possono vedere alcuna modifica, lo stato logico è costante, anche se si capovolge ogni singolo bit nell'oggetto.
>
> Lo stato fisico, noto anche come stato bit per bit, è il modo in cui l'oggetto viene archiviato in memoria. Questo è il nitty-gritty dell'oggetto, i raw 1 e 0 che compongono i suoi dati. Un oggetto è fisicamente costante solo se la sua rappresentazione in memoria non cambia mai.

Si noti che le basi C ++ `const` ness sullo stato logico, non lo stato fisico.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somehow(const Param& p);

    // ...

  public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somehow(p);
    }

    void change_state(Param p) {
        modify_somehow(p);
        state_changed = true;
    }
```

```
    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};
ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    //  by caching the result, so it doesn't need to be recalculated every time the function
    //  is called.  This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Nota che mentre tecnicamente *potresti* usare `const_cast` su `this` per renderlo non-cv-qualificato, in realtà, ***DAVVERO*** , non dovresti, e dovresti usare invece il `mutable` . Un `const_cast` è suscettibile di invocare un comportamento indefinito quando viene utilizzato su un oggetto che è effettivamente `const` , mentre `mutable` è progettato per essere sicuro da usare. Tuttavia, è possibile che si possa imbattersi in questo codice estremamente vecchio.

Un'eccezione a questa regola è la definizione di accessori non cv-qualificati in termini di accessors `const` ; poiché l'oggetto è garantito non essere `const` se viene chiamata la versione non cv-qualificata, non c'è alcun rischio di UB.

```
class CVAccessor {
    int arr[5];

  public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};
```

Questo previene inutili duplicazioni di codice.

Come con i puntatori regolari, se `this` è `volatile` (incluso `const volatile`), viene caricato dalla memoria ogni volta che si accede, invece di essere memorizzato nella cache. Questo ha gli stessi effetti sull'ottimizzazione come dichiarare qualsiasi altro puntatore `volatile` , quindi bisogna fare attenzione.

Si noti che se un'istanza è cv qualificato, le uniche funzioni membro è autorizzato ad accedere sono funzioni membro cui `this` puntatore è almeno altrettanto cv qualificato come istanza stessa:

- Le istanze non cv possono accedere a qualsiasi funzione membro.
- `const` istanze `const` possono accedere a funzioni `const` e `const volatile` .

- `volatile` istanze `volatile` possono accedere a funzioni `volatile` e `const volatile` .
- `const volatile` istanze `const volatile` possono accedere a funzioni `const volatile` .

Questo è uno dei principi chiave della correttezza `const` .

```
struct CVAccess {
    void    func()                {}
    void  func_c() const          {}
    void  func_v() volatile       {}
    void func_cv() const volatile {}
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();    // Error.
c_cva.func_c();  // Good.
c_cva.func_v();  // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();    // Error.
v_cva.func_c();  // Error.
v_cva.func_v();  // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();    // Error.
cv_cva.func_c();  // Error.
cv_cva.func_v();  // Error.
cv_cva.func_cv(); // Good.
```

questo Pointer Ref-Qualificatori

C ++ 11

Analogamente a `this` CV-qualificazioni, possiamo applicare anche *ref-qualificazioni* a `*this` . I qualificatori Ref vengono utilizzati per scegliere tra semantica di riferimento normale e valore di riferimento, consentendo al compilatore di utilizzare la copia o spostare la semantica a seconda di quali sono più appropriati e vengono applicati a `*this` invece di `this` .

Si noti che, nonostante i qualificatori di riferimento che usano la sintassi di riferimento, `this` stesso è ancora un puntatore. Si noti inoltre che i qualificatori di ref non cambiano effettivamente il tipo di `*this` ; è solo più facile descrivere e capire i loro effetti guardandoli come se lo avessero fatto.

```
struct RefQualifiers {
    std::string s;

    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
```

```
    void func() &  { std::cout << "Accessed on normal instance "    << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() &  { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func();              // Output:  Accessed on normal instance Fred
RefQualifiers{}.func(); // Output:  Accessed on temporary instance The nameless one
```

Una funzione membro non può avere sovraccarichi sia con che senza qualificatori di ref; il programmatore deve scegliere tra l'uno o l'altro. Fortunatamente, i qualificatori di cv possono essere usati insieme ai qualificatori di ref, permettendo di seguire le regole di correttezza delle `const` .

```
struct RefCV {
    void func() &                {}
    void func() &&               {}
    void func() const&           {}
    void func() const&&          {}
    void func() volatile&        {}
    void func() volatile&&       {}
    void func() const volatile&  {}
    void func() const volatile&& {}
};
```

Leggi The This Pointer online: https://riptutorial.com/it/cplusplus/topic/7146/the-this-pointer

Capitolo 135: threading

Sintassi

- filo()
- thread (thread e altro)
- thread esplicito (Function && func, Args && ... args)

Parametri

Parametro	Dettagli
`other`	Prende la proprietà di `other` , `other` non possiedono più il thread
`func`	Funzione per chiamare un thread separato
`args`	Argomenti per `func`

Osservazioni

Alcune note:

- Due oggetti `std::thread` **non** possono **mai** rappresentare lo stesso thread.
- Un oggetto `std::thread` può trovarsi in uno stato in cui non rappresenta **alcun** thread (ovvero dopo uno spostamento, dopo aver chiamato `join` , ecc.).

Examples

Operazioni di thread

Quando si avvia un thread, verrà eseguito fino al completamento.

Spesso, ad un certo punto, è necessario (probabilmente - il thread potrebbe già essere fatto) attendere il termine del thread, perché si desidera utilizzare il risultato, ad esempio.

```
int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();

//Now 'n' has the result of the calculation done in the seperate thread
std::cout << n << '\n';
```

Puoi anche `detach` il thread, lasciandolo eseguire liberamente:

```
std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns
```

Passando un riferimento a un thread

Non è possibile passare un riferimento (o riferimento `const`) direttamente a un thread perché `std::thread` lo copia / sposta. Invece, usa `std::reference_wrapper` :

```
void foo(int& b)
{
    b = 10;
}

int a = 1;
std::thread thread{ foo, std::ref(a) }; //'a' is now really passed as reference

thread.join();
std::cout << a << '\n'; //Outputs 10
```

```
void bar(const ComplexObject& co)
{
    co.doCalculations();
}

ComplexObject object;
std::thread thread{ bar, std::cref(object) }; //'object' is passed as const&

thread.join();
std::cout << object.getResult() << '\n'; //Outputs the result
```

Creare uno std :: thread

In C ++, i thread vengono creati utilizzando la classe std :: thread. Una discussione è un flusso separato di esecuzione; è analogo al fatto che un aiutante esegua un compito mentre contemporaneamente ne esegue un altro. Quando viene eseguito tutto il codice nel thread, *termina* . Quando si crea un thread, è necessario passare qualcosa da eseguire su di esso. Alcune cose che puoi passare ad un thread:

- Funzioni libere
- Funzioni membro
- Oggetti Functor
- Espressioni Lambda

Esempio di funzione libera - esegue una funzione su un thread separato (esempio dal vivo):

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                 // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Esempio di funzione membro - esegue una funzione membro su un thread separato (esempio dal vivo):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Esempio di oggetto Functor (esempio dal vivo):

```
#include <iostream>
#include <thread>
```

```
class Bar
{
public:
    void operator()(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Esempio di espressione Lambda (esempio dal vivo):

```
#include <iostream>
#include <thread>

int main()
{
    auto lambda = [](int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Operazioni sul thread corrente

`std::this_thread` è uno `namespace` che ha funzioni per fare cose interessanti sul thread corrente dalla funzione da cui è chiamato.

Funzione	Descrizione
`get_id`	Restituisce l'id del thread
`sleep_for`	Dorme per un determinato periodo di tempo
`sleep_until`	Dorme fino a un'ora specifica

Funzione	Descrizione
`yield`	Ripianifica i thread in esecuzione, dando priorità agli altri thread

---

Ottenere l'id corrente dei thread usando `std::this_thread::get_id` :

```
void foo()
{
    //Print this threads id
    std::cout << std::this_thread::get_id() << '\n';
}

std::thread thread{ foo };
thread.join(); //'threads' id has now been printed, should be something like 12556

foo(); //The id of the main thread is printed, should be something like 2420
```

Dormire per 3 secondi usando `std::this_thread::sleep_for` :

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Dormire fino a 3 ore in futuro usando `std::this_thread::sleep_until` :

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Lasciando che altri thread `std::this_thread::yield` priorità usando `std::this_thread::yield` :

```
void foo(int a)
{
    for (int i = 0; i < a1 ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                   //isn't doing anything important

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

std::thread thread{ foo, 10 };
```

```
thread.join();
```

Usando std :: async invece di std :: thread

`std::async` è anche in grado di creare thread. Rispetto a `std::thread` è considerato meno potente ma più facile da usare quando si desidera eseguire una funzione in modo asincrono.

Chiamata in modo asincrono di una funzione

```
#include <future>
#include <iostream>

unsigned int square(unsigned int i){
    return i*i;
}

int main() {
    auto f = std::async(std::launch::async, square, 8);
    std::cout << "square currently running\n"; //do something while square is running
    std::cout << "result is " << f.get() << '\n'; //getting the result from square
}
```

Insidie comuni

- `std::async` restituisce uno `std::future` che contiene il valore restituito che verrà calcolato dalla funzione. Quando quel `future` viene distrutto, attende il completamento del thread, rendendo il tuo codice effettivamente thread singolo. Questo è facilmente trascurato quando non hai bisogno del valore di ritorno:

  ```
  std::async(std::launch::async, square, 5);
  //thread already completed at this point, because the returning future got destroyed
  ```

- `std::async` funziona senza una politica di avvio, quindi `std::async(square, 5);` compila. Quando lo fai, il sistema decide se vuole creare o meno una discussione. L'idea era che il sistema scelga di creare un thread a meno che non stia già eseguendo più thread di quanto possa funzionare in modo efficiente. Sfortunatamente le implementazioni comunemente scelgono di non creare un thread in quella situazione, mai, quindi è necessario sovrascrivere quel comportamento con `std::launch::async` che forza il sistema a creare un thread.

- Attenzione alle condizioni di gara.

Altro su async su Futures and Promises

Assicurandosi che un thread sia sempre unito

Quando viene chiamato il distruttore per `std::thread` , **deve** essere stata effettuata una chiamata a `join()` o a `detach()` . Se un thread non è stato unito o rimosso, per default verrà chiamato `std::terminate` . Usando RAII , questo è generalmente abbastanza semplice da realizzare:

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }

private:

    std::thread t_;
}
```

Questo viene quindi utilizzato in questo modo:

```
 void perform_work()
 {
     // Perform some work
 }

 void t()
 {
     thread_joiner j{std::thread(perform_work)};
     // Do some other calculations while thread is running
 } // Thread is automatically joined here
```

Questo fornisce anche la sicurezza delle eccezioni; se avessimo creato normalmente il nostro thread e il lavoro svolto in `t()` eseguendo altri calcoli avesse generato un'eccezione, `join()` non sarebbe mai stato chiamato sul nostro thread e il nostro processo sarebbe stato terminato.

Riassegnazione degli oggetti thread

Possiamo creare oggetti thread vuoti e assegnare loro il lavoro in un secondo momento.

Se assegniamo un oggetto thread a un altro thread `joinable` attivo, `std::terminate` verrà chiamato automaticamente prima che il thread venga sostituito.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
//create 100 thread objects that do nothing
std::thread executors[100];

// Some code

// I want to create some threads now
```

```
for (int i = 0;i < 100;i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
         executors[i] = std::thread(foo);
}
```

Sincronizzazione di base

La sincronizzazione dei thread può essere eseguita utilizzando i mutex, tra le altre primitive di sincronizzazione. Esistono diversi tipi di mutex forniti dalla libreria standard, ma il più semplice è `std::mutex` . Per bloccare un mutex, costruisci un blocco su di esso. Il tipo di blocco più semplice è `std::lock_guard` :

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

Con `std::lock_guard` il mutex è bloccato per l'intera vita dell'oggetto lock. Nei casi in cui è necessario controllare manualmente le regioni per il blocco, utilizzare invece `std::unique_lock` :

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

Altre strutture di sincronizzazione dei thread

Uso delle variabili di condizione

Una variabile di condizione è una primitiva usata in congiunzione con un mutex per orchestrare la comunicazione tra i thread. Anche se non è né il modo esclusivo né il modo più efficace per farlo, può essere tra i più semplici per coloro che hanno familiarità con il modello.

Si attende una `std::unique_lock<std::mutex>` `std::condition_variable` con `std::unique_lock<std::mutex>` . Ciò consente al codice di esaminare in sicurezza lo stato condiviso prima di decidere se procedere o meno con l'acquisizione.

Di seguito è riportato uno schizzo produttore-consumatore che utilizza `std::thread` , `std::condition_variable` , `std::mutex` e pochi altri per rendere le cose interessanti.

```
#include <condition_variable>
#include <cstddef>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>


int main()
{
    std::condition_variable cond;
    std::mutex mtx;
    std::queue<int> intq;
    bool stopped = false;

    std::thread producer{[&]()
    {
        // Prepare a random number generator.
        // Our producer will simply push random numbers to intq.
        //
        std::default_random_engine gen{};
        std::uniform_int_distribution<int> dist{};

        std::size_t count = 4006;
        while(count--)
        {
            // Always lock before changing
            // state guarded by a mutex and
            // condition_variable (a.k.a. "condvar").
            std::lock_guard<std::mutex> L{mtx};

            // Push a random int into the queue
            intq.push(dist(gen));

            // Tell the consumer it has an int
            cond.notify_one();
        }

        // All done.
        // Acquire the lock, set the stopped flag,
        // then inform the consumer.
        std::lock_guard<std::mutex> L{mtx};

        std::cout << "Producer is done!" << std::endl;

        stopped = true;
        cond.notify_one();
    }};

    std::thread consumer{[&]()
    {
        do{
            std::unique_lock<std::mutex> L{mtx};
            cond.wait(L,[&]()
            {
                // Acquire the lock only if
                // we've stopped or the queue
                // isn't empty
                return stopped || ! intq.empty();
            });
```

```
            // We own the mutex here; pop the queue
            // until it empties out.

            while( ! intq.empty())
            {
                const auto val = intq.front();
                intq.pop();

                std::cout << "Consumer popped: " << val << std::endl;
            }

            if(stopped){
                // producer has signaled a stop
                std::cout << "Consumer is done!" << std::endl;
                break;
            }

        }while(true);
    }};

    consumer.join();
    producer.join();

    std::cout << "Example Completed!" << std::endl;

    return 0;
}
```

Creare un pool di thread semplice

I primitivi di filettatura del C ++ 11 sono ancora relativamente bassi. Possono essere usati per scrivere un costrutto di livello superiore, come un pool di thread:

C ++ 14

```
struct tasks {
  // the mutex, condition variable and deque form a single
  // thread-safe triggered queue of tasks:
  std::mutex m;
  std::condition_variable v;
  // note that a packaged_task<void> can store a packaged_task<R>:
  std::deque<std::packaged_task<void()>> work;

  // this holds futures representing the worker threads being done:
  std::vector<std::future<void>> finished;

  // queue( lambda ) will enqueue the lambda into the tasks for the threads
  // to use.  A future of the type the lambda returns is given to let you get
  // the result out.
  template<class F, class R=std::result_of_t<F&()>>
  std::future<R> queue(F&& f) {
    // wrap the function object into a packaged task, splitting
    // execution from the return value:
    std::packaged_task<R()> p(std::forward<F>(f));

    auto r=p.get_future(); // get the return value before we hand off the task
    {
      std::unique_lock<std::mutex> l(m);
```

```
      work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
    }
    v.notify_one(); // wake a thread to work on the task

    return r; // return the future result of the task
  }

  // start N threads in the thread pool.
  void start(std::size_t N=1){
    for (std::size_t i = 0; i < N; ++i)
    {
      // each thread is a std::async running this->thread_task():
      finished.push_back(
        std::async(
          std::launch::async,
          [this]{ thread_task(); }
        )
      );
    }
  }
  // abort() cancels all non-started tasks, and tells every working thread
  // stop running, and waits for them to finish up.
  void abort() {
    cancel_pending();
    finish();
  }
  // cancel_pending() merely cancels all non-started tasks:
  void cancel_pending() {
    std::unique_lock<std::mutex> l(m);
    work.clear();
  }
  // finish enques a "stop the thread" message for every thread, then waits for them:
  void finish() {
    {
      std::unique_lock<std::mutex> l(m);
      for(auto&&unused:finished){
        work.push_back({});
      }
    }
    v.notify_all();
    finished.clear();
  }
  ~tasks() {
    finish();
  }
private:
  // the work that a worker thread does:
  void thread_task() {
    while(true){
      // pop a task off the queue:
      std::packaged_task<void()> f;
      {
        // usual thread-safe queue code:
        std::unique_lock<std::mutex> l(m);
        if (work.empty()){
          v.wait(l,[&]{return !work.empty();});
        }
        f = std::move(work.front());
        work.pop_front();
      }
      // if the task is invalid, it means we are asked to abort:
```

```
      if (!f.valid()) return;
      // otherwise, run the task:
      f();
    }
  }
};
```

`tasks.queue( []{ return "hello world"s; } )` restituisce uno `std::future<std::string>` , che quando l'oggetto delle attività ottiene l'esecuzione, viene popolato con `hello world` .

Puoi creare thread eseguendo `tasks.start(10)` (che avvia 10 thread).

L'uso di `packaged_task<void()>` è semplicemente perché non esiste un equivalente `std::function` cancellato dal tipo che memorizza i tipi di spostamento. Scrivendo uno di quelli personalizzati sarebbe probabilmente più veloce di usare `packaged_task<void()>` .

Esempio dal vivo

C ++ 11

In C ++ 11, sostituire `result_of_t<blah>` con `typename result_of<blah>::type` .

Altro su Mutex .

Memorizzazione locale del thread

L'archiviazione locale del thread può essere creata usando la parola chiave `thread_local` . Si dice che una variabile dichiarata con l' `thread_local` abbia **durata di memorizzazione del thread.**

- Ogni thread in un programma ha la propria copia di ogni variabile locale del thread.
- Una variabile locale del thread con ambito di funzione (locale) verrà inizializzata al primo passaggio del controllo attraverso la sua definizione. Tale variabile è implicitamente statica, a meno che non sia dichiarata `extern` .
- Una variabile locale del thread con spazio dei nomi o classe (non locale) verrà inizializzata come parte dell'avvio del thread.
- Le variabili locali del thread vengono distrutte al termine della terminazione del thread.
- Un membro di una classe può essere thread-local solo se è statico. Ci sarà quindi una copia di quella variabile per thread, piuttosto che una copia per coppia (thread, istanza).

Esempio:

```
void debug_counter() {
    thread_local int count = 0;
    Logger::log("This function has been called %d times by this thread", ++count);
}
```

Leggi threading online: https://riptutorial.com/it/cplusplus/topic/699/threading

Capitolo 136: Tipi atomici

Sintassi

- std :: atomica <T>
- std :: atomic_flag

Osservazioni

`std::atomic` consente l'accesso atomico a un tipo TriviallyCopyable, dipende dall'implementazione se questo viene eseguito tramite operazioni atomiche o utilizzando i lock. L'unico tipo atomico senza blocco garantito è `std::atomic_flag` .

Examples

Accesso multi-thread

Un tipo atomico può essere utilizzato per leggere e scrivere in modo sicuro in una posizione di memoria condivisa tra due thread.

Un cattivo esempio che potrebbe causare una corsa di dati:

```
#include <thread>
#include <iostream>


//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
        std::cout << shared << std::endl;
    }
```

```
    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

L'esempio sopra può causare una lettura corrotta e può portare a un comportamento non definito.

Un esempio di sicurezza del thread:

```
#include <atomic>
#include <thread>
#include <iostream>


    //function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }


    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

L'esempio sopra è sicuro perché tutte le operazioni `store()` e `load()` del tipo di dati `atomic` proteggono l' `int` incapsulato dall'accesso simultaneo.

Leggi Tipi atomici online: https://riptutorial.com/it/cplusplus/topic/3804/tipi-atomici

Capitolo 137: Tipi senza nome

Examples

Classi senza nome

A differenza di una classe o di una struttura denominate, le classi e le strutture senza nome devono essere istanziate dove sono definite e non possono avere costruttori o distruttori.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Membri anonimi

Essendo un'estensione non standard del C ++, i compilatori comuni consentono l'uso di classi come membri anonimi.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;

    //The anonymous struct's members are accessed as if members of the parent struct
    Example() : inner_b(2), outer_b(4) {
        inner_b = outer_b + 2;
    }
};

Example ex;

//The same holds true for external code referencing the struct
ex.inner_b -= ex.outer_b;
```

Come alias di tipo

I tipi di classe senza nome possono essere utilizzati anche durante la creazione di alias di tipo, ad esempio tramite `typedef` e `using` :

C ++ 11

```
using vec2d = struct {
    float x;
    float y;
};
```

```
typedef struct {
    float x;
    float y;
} vec2d;
```

```
vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

Unione anonima

I nomi dei membri di un'unione anonima appartengono allo scopo della dichiarazione di unione e devono essere distinti da tutti gli altri nomi di questo ambito. L'esempio qui ha la stessa costruzione dei membri anonimi di esempio che usano "struct" ma è conforme allo standard.

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
  Sample sa;
  sa.a =3;
  sa.b =4;
  sa.c =5;
}
```

Leggi Tipi senza nome online: https://riptutorial.com/it/cplusplus/topic/2704/tipi-senza-nome

Capitolo 138: tipo deduzione

Osservazioni

A novembre 2014, il Comitato di standardizzazione del C ++ ha adottato la proposta N3922, che elimina la speciale regola di deduzione del tipo per gli inizializzatori automatici e rinforzati usando la sintassi di inizializzazione diretta. Questo non fa parte dello standard C ++ ma è stato implementato da alcuni compilatori.

Examples

Deduzione del parametro Template per costruttori

Prima di C ++ 17, la deduzione del modello non può dedurre il tipo di classe per te in un costruttore. Deve essere specificato esplicitamente. A volte, tuttavia, questi tipi possono essere molto ingombranti o (nel caso di lambda) impossibili da nominare, quindi abbiamo una proliferazione di fabbriche di tipi (come `make_pair()` , `make_tuple()` , `back_inserter()` , ecc.).

C ++ 17

Questo non è più necessario:

```
std::pair p(2, 4.5);     // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
    std::back_insert_iterator(vi2)); // constructs a back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Si presume che i costruttori deducano i parametri del modello di classe, ma in alcuni casi questo non è sufficiente e possiamo fornire guide esplicative sulla deduzione:

```
template <class Iter>
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Detrazione del tipo di modello

Sintassi generica dei modelli

```
template<typename T>
void f(ParamType param);

f(expr);
```

Caso 1: `ParamType` è un riferimento o un puntatore, ma non un riferimento universale o `ParamType` .

In questo caso, la deduzione di tipo funziona in questo modo. Il compilatore ignora la parte di riferimento se esiste in `expr` . Il compilatore quindi modellisti partite `expr` tipo S' contro `ParamType` a determing `T` .

```
template<typename T>
void f(T& param);      //param is a reference

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a reference to x as a const int

f(x);                  // T is int, param's type is int&
f(cx);                 // T is const int, param's type is const int&
f(rx);                 // T is const int, param's type is const int&
```

Caso 2: `ParamType` è un riferimento universale o di riferimento `ParamType` . In questo caso, la deduzione di tipo è la stessa del caso 1 se l' `expr` è un valore. Se `expr` è un lvalue, sia `T` che `ParamType` sono dedotti come riferimenti lvalue.

```
template<typename T>
void f(T&& param);     // param is a universal reference

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a reference to x as a const int

f(x);                  // x is lvalue, so T is int&, param's type is also int&
f(cx);                 // cx is lvalue, so T is const int&, param's type is also const int&
f(rx);                 // rx is lvalue, so T is const int&, param's type is also const int&
f(27);                 // 27 is rvalue, so T is int, param's type is therefore int&&
```

Caso 3: `ParamType` è né un puntatore né un riferimento. Se `expr` è un riferimento, la parte di riferimento viene ignorata. Se `expr` è const che viene ignorato pure. Se è volatile, viene anche ignorato quando si deduce il tipo di T.

```
template<typename T>
void f(T param);       // param is now passed by value

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a reference to x as a const int

f(x);                  // T's and param's types are both int
f(cx);                 // T's and param's types are again both int
f(rx);                 // T's and param's types are still both int
```

Deduzione del tipo automatico

C ++ 11

Digitare la deduzione utilizzando la parola chiave `auto` funziona quasi come Deduzione tipo di modello. Di seguito sono riportati alcuni esempi:

```
auto x = 27;           // (x is neither a pointer nor a reference), x's type is int
const auto cx = x;     // (cx is neither a pointer nor a reference), cs's type is const int
const auto& rx = x;    // (rx is a non-universal reference), rx's type is a reference to a
const int

auto&& uref1 = x;      // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx;     // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27;     // 27 is an int and rvalue, so uref3's type is int&&
```

Le differenze sono descritte di seguito:

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // type is int, value is 27
auto x3 = { 27 };      // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 };         // type is std::initializer_list<int>, value is { 27 }
                       // in some compilers type may be deduced as an int with a
                       // value of 27. See remarks for more information.
auto x5 = { 1, 2.0 }   // error! can't deduce T for std::initializer_list<t>
```

Come puoi vedere se usi inizializzatori rinforzati, auto è costretto a creare una variabile di tipo `std::initializer_list<T>` . Se non può dedurre il valore di `T` , il codice viene rifiutato.

Quando `auto` è usato come tipo di ritorno di una funzione, specifica che la funzione ha un tipo di ritorno finale .

```
auto f() -> int {
    return 42;
}
```

C ++ 14

C ++ 14 consente, oltre agli usi di auto consentiti in C ++ 11, i seguenti:

1. Se utilizzato come tipo di ritorno di una funzione senza un tipo di ritorno finale, specifica che il tipo di ritorno della funzione deve essere dedotto dalle istruzioni di ritorno nel corpo della funzione, se presenti.

   ```
   // f returns int:
   auto f() { return 42; }
   // g returns void:
   auto g() { std::cout << "hello, world!\n"; }
   ```

2. Quando viene utilizzato nel tipo di parametro di una lambda, definisce la lambda come una lambda generica .

   ```
   auto triple = [](auto x) { return 3*x; };
   const auto x = triple(42); // x is a const int with value 126
   ```

La forma speciale `decltype(auto)` deduce un tipo usando le regole di deduzione tipo di `decltype` piuttosto che quelle di `auto` .

```
int* p = new int(42);
```

```
auto x = *p;           // x has type int
decltype(auto) y = *p; // y is a reference to *p
```

In C ++ 03 e precedenti, la parola chiave `auto` aveva un significato completamente diverso come identificatore di una classe di memoria ereditata da C.

Leggi tipo deduzione online: https://riptutorial.com/it/cplusplus/topic/7863/tipo-deduzione

Capitolo 139: Tipo di inferenza

introduzione

Questo argomento tratta sul tipo di inferenze che coinvolge la parola chiave `auto` tipo che è disponibile da C ++ 11.

Osservazioni

Di solito è meglio dichiarare `const` e `& constexpr` ogni volta che si utilizza l' `auto` se è necessario per evitare comportamenti indesiderati come la copia o le mutazioni. Questi ulteriori suggerimenti assicurano che il compilatore non generi altre forme di inferenza. Inoltre, non è consigliabile utilizzare l' `auto` e deve essere utilizzato solo quando la dichiarazione effettiva è molto lunga, specialmente con i modelli STL.

Examples

Tipo di dati: automatico

Questo esempio mostra le inferenze di tipo di base che il compilatore può eseguire.

```
auto a = 1;         //    a = int
auto b = 2u;        //    b = unsigned int
auto c = &a;        //    c = int*
const auto  d = c; //    d = const int*
const auto& e = b; //    e = const unsigned int&

auto x = a + b     //    x = int, #compiler warning unsigned and signed

auto v = std::vector<int>;    //    v = std::vector<int>
```

Tuttavia, la parola chiave auto non esegue sempre l'inferenza di tipo prevista senza ulteriori suggerimenti per `&` o `const` o `constexpr`

```
//    y = unsigned int,
//    note that y does not infer as const unsigned int&
//    The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Lambda auto

Il tipo di dati parola chiave automatica è un modo conveniente per i programmatori di dichiarare funzioni lambda. Aiuta accorciando la quantità di programmatori di testo che è necessario digitare per dichiarare un puntatore a funzione.

```
auto DoThis = [](int a, int b) { return a + b; };
```

```
//    Do this is of type (int)(*DoThis)(int, int)
//    else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2);    //    c = int
auto d = pDothis(1, 2);   //    d = int

//    using 'auto' shortens the definition for lambda functions
```

Per impostazione predefinita, se il tipo restituito di funzioni lambda non è definito, verrà automaticamente dedotto dai tipi di espressione di ritorno.

Questi 3 sono fondamentalmente la stessa cosa

```
[](int a, int b) -> int  { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

Cicli e auto

Questo esempio mostra come auto può essere usato per abbreviare la dichiarazione del tipo per i loop

```
std::map<int, std::string> Map;
for (auto pair : Map)            //    pair = std::pair<int, std::string>
for (const auto pair : Map)      //    pair = const std::pair<int, std::string>
for (const auto& pair : Map)     //    pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i)         //    i = int
for (auto i = 0; i < Map.size(); ++i)   //    Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i)   //    i = size_t
```

Leggi Tipo di inferenza online: https://riptutorial.com/it/cplusplus/topic/8233/tipo-di-inferenza

Capitolo 140: Tipo di ritorno Covariance

Osservazioni

La covarianza di un parametro o un valore di ritorno per una funzione membro virtuale `m` è dove il suo tipo `T` diventa più specifico in un override della classe derivata di `m` . Il tipo `T` poi varia (*varianza*) in specificità allo stesso modo (*co*) delle classi che forniscono `m` . C ++ fornisce supporto linguistico per i *tipi di ritorno* covarianti che sono puntatori grezzi o riferimenti grezzi - la covarianza è per il tipo punta o referente.

Il supporto C ++ è limitato ai tipi restituiti perché i valori restituiti dalla funzione sono gli unici **argomenti out-out** in C ++ e la covarianza è solo sicura per un puro argomento out. Altrimenti, il codice chiamante potrebbe fornire un oggetto di tipo meno specifico di quanto previsto dal codice ricevente. La professoressa Barbara Liskov del MIT ha studiato questo problema correlato alla sicurezza della varianza, ed è ora noto come Principio di sostituzione di Liskov, o **LSP** .

Il supporto della covarianza aiuta essenzialmente ad evitare downcast e controllo dinamico dei tipi.

Poiché puntatori intelligenti sono di tipo classe non si può usare il supporto integrato per covarianza direttamente risultati di puntatore intelligente, ma si può definire *apparentemente covariante* non `virtual` funzioni risultato puntatore involucro intelligenti per una covariante `virtual` funzione che produce i puntatori prime.

Examples

1. Esempio di base senza rendimenti covarianti, mostra perché sono desiderabili

```
// 1. Base example not using language support for covariance, dynamic type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{

public:
    Top* clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
```

```
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}

    Top* clone() const override
    { return new DD( *this ); }
};

#include <assert.h>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
    cout << boolalpha;

    DD* p1 = new DD();
    Top* p2 = p1->clone();
    bool const  correct_dynamic_type = (typeid( *p2 ) == typeid( DD ));
    cout << correct_dynamic_type << endl;               // "true"

    assert( correct_dynamic_type ); // This is essentially dynamic type checking. :(
    auto p2_most_derived = static_cast<DD*>( p2 );
    cout << p2_most_derived->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

2. Versione di risultato covariant dell'esempio di base, controllo di tipo statico.

```
// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}
```

```
    DD* /* ← Covariant return */ clone() const override
    { return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

3. Risultato puntatore intelligente covariant (pulizia automatica).

```
// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
    { return up( virtual_clone() ); }

    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
    D* /* ← Covariant return */ virtual_clone() const override
    { return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

class DD : public D
{
private:
    int answer_ = 42;

    DD* /* ← Covariant return */ virtual_clone() const override
    { return new DD( *this ); }
```

```
public:
    int answer() const
    { return answer_;}

    unique_ptr<DD> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

#include <iostream>
using namespace std;

int main()
{
    auto  p1 = unique_ptr<DD>(new DD());
    auto  p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    // Cleanup is automated via unique_ptr.
 }
```

Leggi Tipo di ritorno Covariance online: https://riptutorial.com/it/cplusplus/topic/5411/tipo-di-ritorno-covariance

Capitolo 141: Tipo di ritorno finale

Sintassi

- *function_name* ([*function_args*]) [*function_attributes*] [*function_qualifiers*] `->` *trailing-return-type* [*requires_clause*]

Osservazioni

La sintassi precedente mostra una dichiarazione di funzione completa utilizzando un tipo finale, in cui le parentesi quadre indicano una parte facoltativa della dichiarazione di funzione (come la lista di argomenti se una funzione no-arg).

Inoltre, la sintassi del tipo di ritorno finale non consente di definire un tipo di classe, unione o enum all'interno di un tipo di ritorno finale (si noti che questo non è consentito neanche in un tipo di ritorno iniziale). Oltre a questo, i tipi possono essere scritti allo stesso modo dopo il `->` come sarebbero altrove.

Examples

Evitare di qualificare un nome di tipo nidificato

```
class ClassWithAReallyLongName {
  public:
    class Iterator { /* ... */ };
    Iterator end();
};
```

Definizione del membro `end` con un tipo di ritorno finale:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Definizione della `end` del membro senza un tipo di ritorno finale:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

Il tipo di ritorno finale viene cercato nell'ambito della classe, mentre un tipo di ritorno iniziale viene ricercato nell'ambito del namespace che lo racchiude e può quindi richiedere una qualifica "ridondante".

Espressioni Lambda

Un lambda può avere *solo* un tipo di ritorno finale; la sintassi principale del tipo di ritorno non è applicabile a lambdas. Si noti che in molti casi non è necessario specificare un tipo di ritorno per una lambda.

```
struct Base {};
struct Derived1 : Base {};
struct Derived2 : Base {};
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Leggi Tipo di ritorno finale online: https://riptutorial.com/it/cplusplus/topic/4142/tipo-di-ritorno-finale

Capitolo 142: Tipo Tratti

Osservazioni

I tratti di tipo sono costrutti basati su modelli utilizzati per confrontare e testare le proprietà di diversi tipi in fase di compilazione. Possono essere utilizzati per fornire una logica condizionale in fase di compilazione che può limitare o estendere la funzionalità del codice in un modo specifico. La libreria dei caratteri di tipo è stata introdotta con lo standard `c++11` che fornisce un numero di funzionalità diverse. È anche possibile creare modelli di confronto trait personalizzati.

Examples

Tratti di tipo standard

C ++ 11

L'intestazione `type_traits` contiene un insieme di classi template e helper per trasformare e controllare le proprietà dei tipi in fase di compilazione.

Questi tratti sono in genere utilizzati nei modelli per verificare gli errori degli utenti, supportare la programmazione generica e consentire ottimizzazioni.

La maggior parte dei caratteri tipografici viene utilizzata per verificare se un tipo soddisfa alcuni criteri. Questi hanno la seguente forma:

```
template <class T> struct is_foo;
```

Se la classe template è istanziata con un tipo che soddisfa alcuni criteri `foo` , allora `is_foo<T>` eredita da `std::integral_constant<bool,true>` (aka `std::true_type`), altrimenti eredita da `std::integral_constant<bool,false>` (aka `std::false_type`). Questo conferisce al tratto i seguenti membri:

costanti

`static constexpr bool value`

`true` se `T` soddisfa i criteri `foo` , `false` altrimenti

funzioni

`operator bool`

Restituisce `value`

C ++ 14

`bool operator()`

Restituisce `value`

tipi

Nome	Definizione
`value_type`	`bool`
`type`	`std::integral_constant<bool,value>`

Il tratto può quindi essere utilizzato in costrutti come `static_assert` o `std::enable_if` . Un esempio con `std::is_pointer` :

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}
```

Ci sono anche vari tratti che trasformano i tipi, come `std::add_pointer` e `std::underlying_type` . Questi tratti generalmente espongono un `type` membro di tipo singolo che contiene il tipo trasformato. Ad esempio, `std::add_pointer<int>::type` è `int*` .

Digitare le relazioni con std :: is_same

C ++ 11

La relazione di tipo `std::is_same<T, T>` viene utilizzata per confrontare due tipi. Valuterà come booleano, vero se i tipi sono uguali e falso se diversamente.

per esempio

```
// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
```

```
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout  << std::is_same<unsigned int, int>::value << "\n";
```

Anche la relazione di tipo `std::is_same` funzionerà indipendentemente da typedef. Questo è effettivamente dimostrato nel primo esempio quando si confronta `int == int32_t` tuttavia questo non è completamente chiaro.

per esempio

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value <<  "\n";
```

Usando `std::is_same` per avvisare quando si utilizza impropriamente una classe o una funzione `std::is_same` su modello.

Quando combinato con un `std::is_same` statico, il modello `std::is_same` può essere uno strumento prezioso per far rispettare l'uso corretto di classi e funzioni `std::is_same` su modelli.

Ad esempio una funzione che consente solo l'input da un `int` e una scelta di due strutture.

```
#include <type_traits>
struct foo {
  int member;
  // Other variables
};

struct bar {
  char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
  // If type T != foo || T != bar then show error message.
  static_assert(std::is_same<T, foo>::value ||
    std::is_same<T, bar>::value,
    "This function does not support the specified type.");
  return var1.member + var2;
}
```

Caratteri fondamentali del tipo

C ++ 11

Ci sono un certo numero di tratti di tipo diverso che confrontano tipi più generali.

È integrale:

Valuta come vero per tutti i tipi interi `int` , `char` , `long` , `unsigned int` ecc.

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
```

```
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

È in virgola mobile:

Valuta come vero per tutti i tipi di virgola mobile. `float` , `double` , `long double` ecc.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

È Enum:

Valuta come vero per tutti i tipi enumerati, inclusa la `enum class` .

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

È puntatore:

Valuta come vero per tutti i puntatori.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

È classe:

Valuta come vero per tutte le classi e struct, ad eccezione della `enum class` .

```
struct FOO {int x, y;};
class BAR {
 public:
  int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

Tipo Proprietà

C ++ 11

Le proprietà del tipo confrontano i modificatori che possono essere posizionati su variabili diverse. L'utilità di questi tratti del genere non è sempre ovvia.

Nota: l'esempio seguente offre solo un miglioramento su un compilatore non ottimizzante. È una semplice dimostrazione di concetto, piuttosto che un esempio complesso.

es. Divisione veloce per quattro.

```
template<typename T>
inline T FastDivideByFour(cont T &var) {
  // Will give an error if the inputted type is not an unsigned integral type.
  static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
    "This function is only designed for unsigned integral types.");
  return (var >> 2);
}
```

È costante:

Questo valuterà come vero quando il tipo è costante.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

È volatile:

Questo verrà valutato come true quando il tipo è volatile.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

È firmato:

Questo valuterà come vero per tutti i tipi firmati.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Non firmato:

Valuterà come vero per tutti i tipi non firmati.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

Leggi Tipo Tratti online: https://riptutorial.com/it/cplusplus/topic/4750/tipo-tratti

Capitolo 143: Typedef e digita alias

introduzione

Il `typedef` e (dal C ++ 11) `using` parole chiave possono essere usati per dare un nuovo nome ad un tipo esistente.

Sintassi

- typedef *type-specificatore-seq init-declarator-list* ;
- *attributo-specificatore-seq* typedef *decl-specifier-seq init-declarator-list* ; // dal C ++ 11
- using *identifier attribute-specifier-seq* (*opt*) = *type-id* ; // dal C ++ 11

Examples

Sintassi typedef di base

Una dichiarazione `typedef` ha la stessa sintassi di una dichiarazione di variabile o funzione, ma contiene la parola `typedef` . La presenza di `typedef` fa sì che la dichiarazione dichiari un tipo invece di una variabile o funzione.

```
int T;         // T has type int
typedef int T; // T is an alias for int

int A[100];         // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Una volta definito un alias di tipo, può essere utilizzato in modo intercambiabile con il nome originale del tipo.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

`typedef` non crea mai un tipo distinto. Dà solo un altro modo di riferirsi a un tipo esistente.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Usi più complessi di typedef

La regola che le dichiarazioni `typedef` hanno la stessa sintassi delle dichiarazioni di variabili e funzioni ordinarie può essere utilizzata per leggere e scrivere dichiarazioni più complesse.

```
void (*f)(int);         // f has type "pointer to function of int returning void"
typedef void (*f)(int); // f is an alias for "pointer to function of int returning void"
```

Ciò è particolarmente utile per i costrutti con sintassi confusa, come i puntatori ai membri non statici.

```
void (Foo::*pmf)(int);         // pmf has type "pointer to member function of Foo taking int
                               // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                               // taking int and returning void"
```

È difficile ricordare la sintassi delle seguenti dichiarazioni di funzione, anche per i programmatori esperti:

```
void (Foo::*Foo::f(const char*))(int);
int (&g())[100];
```

`typedef` può essere usato per renderli più facili da leggere e scrivere:

```
typedef void (Foo::pmf)(int);  // pmf is a pointer to member function type
pmf Foo::f(const char*);       // f is a member function of Foo

typedef int (&ra)[100];        // ra means "reference to array of 100 ints"
ra g();                        // g returns reference to array of 100 ints
```

Dichiarazione di più tipi con typedef

La parola chiave `typedef` è un identificatore, quindi si applica separatamente a ciascun dichiarante. Pertanto, ogni nome dichiarato si riferisce al tipo che quel nome avrebbe in assenza di `typedef` .

```
int *x, (*p)();         // x has type int*, and p has type int(*)()
typedef int *x, (*p)(); // x is an alias for int*, while p is an alias for int(*)()
```

Dichiarazione alias con "utilizzo"

C ++ 11

La sintassi `using` è molto semplice: il nome da definire va sul lato sinistro e la definizione va sul lato destro. Non c'è bisogno di scansionare per vedere dove si trova il nome.

```
using I = int;
using A = int[100];             // array of 100 ints
using FP = void(*)(int);        // pointer to function of int returning void
using MP = void (Foo::*)(int);  // pointer to member function of Foo of int returning void
```

La creazione di un alias di tipo con l' `using` ha esattamente lo stesso effetto della creazione di un

alias di tipo con `typedef` . È semplicemente una sintassi alternativa per realizzare la stessa cosa.

A differenza di `typedef` , l' `using` può essere `using` modelli. Un "modello typedef" creato con l' `using` è chiamato un modello alias .

Leggi Typedef e digita alias online: https://riptutorial.com/it/cplusplus/topic/9328/typedef-e-digita-alias

Capitolo 144: Una regola di definizione (ODR)

Examples

Funzione definita in modo multiplo

La conseguenza più importante della One Definition Rule è che le funzioni non inline con link esterno dovrebbero essere definite solo una volta in un programma, sebbene possano essere dichiarate più volte. Pertanto, tali funzioni non dovrebbero essere definite nelle intestazioni, poiché un'intestazione può essere inclusa più volte da diverse unità di traduzione.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() { std:: cout << "bar"; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In questo programma, la funzione `foo` è definita nell'intestazione `foo.h` , che è inclusa due volte: una volta da `foo.cpp` e una volta da `main.cpp` . Ogni unità di traduzione contiene quindi la propria definizione di `foo` . Si noti che le guardie di `foo.h` in `foo.h` non impediscono che ciò accada, dal momento che `foo.cpp` e `main.cpp` entrambi includono *separatamente* `foo.h` Il risultato più probabile del tentativo di creare questo programma è un errore link-time che identifica `foo` come se fosse stato definito in modo multiplo.

Per evitare tali errori, è necessario *dichiarare le* funzioni nelle intestazioni e *definirle* nei corrispondenti file `.cpp` , con alcune eccezioni (vedere altri esempi).

Funzioni inline

Una funzione dichiarata in `inline` può essere definita in più unità di traduzione, a condizione che tutte le definizioni siano identiche. Deve anche essere definito in ogni unità di traduzione in cui

viene utilizzato. Pertanto, le funzioni inline *dovrebbero* essere definite nelle intestazioni e non è necessario menzionarle nel file di implementazione.

Il programma si comporterà come se ci fosse una singola definizione della funzione.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In questo esempio, la funzione più semplice `foo` è definita in linea nel file di intestazione mentre la `bar` funzione più complessa non è in linea ed è definita nel file di implementazione. Entrambe le unità di traduzione `foo.cpp` e `main.cpp` contengono le definizioni di `foo` , ma questo programma è ben formato poiché `foo` è in linea.

Una funzione definita all'interno di una definizione di classe (che può essere una funzione membro o una funzione amico) è *implicitamente in* linea. Pertanto, se una classe è definita in un'intestazione, le funzioni membro della classe possono essere definite all'interno della definizione della classe, anche se le definizioni possono essere incluse in più unità di traduzione:

```
// in foo.h
class Foo {
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
void Foo::baz() {
   // definition
}
```

La funzione `Foo::baz` è definito fuori linea, quindi *non* è una funzione inline, e non deve essere definito nell'intestazione.

Violazione di ODR tramite risoluzione di sovraccarico

Anche con token identici per funzioni inline, l'ODR può essere violato se la ricerca di nomi non si riferisce alla stessa entità. consideriamo `func` in following:

- header.h

```
void overloaded(int);
inline void func() { overloaded('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    func(); // `overloaded` refers to `void overloaded(int)`
}
```

- bar.cpp

```
void overloaded(char); // can come from other include
#include "header.h"

void bar()
{
    func(); // `overloaded` refers to `void overloaded(char)`
}
```

Abbiamo una violazione ODR come `overloaded` riferisce a entità diverse a seconda dell'unità di traduzione.

Leggi Una regola di definizione (ODR) online: https://riptutorial.com/it/cplusplus/topic/4907/una-regola-di-definizione--odr-

Capitolo 145: Utilizzando la dichiarazione

introduzione

Una dichiarazione `using` introduce un singolo nome nello scope corrente precedentemente dichiarato altrove.

Sintassi

- usando typename (*opt*) *nested-name-specifier unqualified-id* ;
- usando :: *id non qualificato* ;

Osservazioni

Una *dichiarazione d'uso* è distinta da una direttiva using , che dice al compilatore di cercare in uno spazio dei nomi particolare quando cerca *un* nome *qualsiasi* . Una *direttiva* `using namespace` inizia con l' `using namespace` .

Una *dichiarazione d'uso* è anche distinta da una dichiarazione di alias, che dà un nuovo nome ad un tipo esistente allo stesso modo di `typedef` . Una dichiarazione alias contiene un segno di uguale.

Examples

Importazione di nomi individuali da uno spazio dei nomi

Una volta `using` viene utilizzato per introdurre il nome `cout` dallo spazio dei nomi `std` nello scope della funzione `main` , l'oggetto `std::cout` può essere indicato come `cout` alone.

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Riconoscere i membri di una classe base per evitare l'occultamento del nome

Se una *dichiarazione di utilizzo si* verifica nell'ambito della classe, è consentito solo ridichiarare un membro di una classe base. Ad esempio, l' `using std::cout` non è consentito nell'ambito della classe.

Spesso, il nome redeclared è uno che altrimenti sarebbe nascosto. Ad esempio, nel codice seguente, `d1.foo` fa riferimento solo a `Derived1::foo(const char*)` e si verificherà un errore di compilazione. La funzione `Base::foo(int)` è nascosta e non viene considerata affatto. Tuttavia, `d2.foo(42)` va bene perché la *dichiarazione using* porta `Base::foo(int)` nell'insieme di entità

denominate `foo` in `Derived2` . Nome ricerca trova quindi sia `foo` s e la risoluzione di sovraccarico seleziona `Base::foo` .

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42);  // error
    Derived2 d2;
    d2.foo(42);  // OK
}
```

Costruttori ereditari

C ++ 11

Come caso speciale, una *dichiarazione di utilizzo* nell'ambito della classe può fare riferimento ai costruttori di una classe di base diretta. Questi costruttori vengono quindi *ereditati* dalla classe derivata e possono essere utilizzati per inizializzare la classe derivata.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

Nel codice precedente, sia `Derived1` che `Derived2` hanno costruttori che inoltrano gli argomenti direttamente al costruttore corrispondente di `Base` . `Derived1` esegue esplicitamente l'inoltro, mentre `Derived2` , utilizzando la funzione C ++ 11 dei costruttori ereditari, lo fa in modo implicito.

Leggi Utilizzando la dichiarazione online: https://riptutorial.com/it/cplusplus/topic/9301/utilizzando-la-dichiarazione

Capitolo 146: Utilizzando std :: unordered_map

introduzione

std :: unordered_map è solo un contenitore associativo. Funziona su chiavi e le loro mappe. Chiave come dice il nome, aiuta ad avere unicità nella mappa. Mentre il valore mappato è solo un contenuto associato alla chiave. I tipi di dati di questa chiave e mappa possono essere qualsiasi tipo di dati predefinito o definito dall'utente.

Osservazioni

Come dice il nome, gli elementi nella mappa non ordinata non vengono memorizzati nella sequenza ordinata. Sono memorizzati in base ai loro valori hash e quindi l'utilizzo di una mappa non ordinata ha molti vantaggi, come ad esempio O (1) per cercare qualsiasi elemento da esso. È anche più veloce di altri contenitori di mappe. È anche visibile dall'esempio che è molto facile da implementare poiché l'operatore ([]) ci aiuta ad accedere direttamente al valore mappato.

Examples

Dichiarazione e uso

Come già accennato, puoi dichiarare una mappa non ordinata di qualsiasi tipo. Disponiamo di una mappa non ordinata denominata prima con tipo stringa e intero.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Alcune funzioni di base

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value

variable_name.begin(); // iterator to the first element

variable_name.end(); // iterator to the last + 1 element
```

Leggi Utilizzando std :: unordered_map online:
https://riptutorial.com/it/cplusplus/topic/10540/utilizzando-std----unordered-map

Capitolo 147: Variabili in linea

introduzione

Una variabile in linea può essere definita in più unità di traduzione senza violare la regola della definizione unica . Se è definito a più livelli, il linker unirà tutte le definizioni in un singolo oggetto nel programma finale.

Examples

Definizione di un membro dati statico nella definizione della classe

Un membro di dati statici della classe può essere completamente definito all'interno della definizione di classe se è dichiarato in `inline` . Ad esempio, la seguente classe può essere definita in un'intestazione. Prima di C ++ 17, sarebbe stato necessario fornire un file `.cpp` per contenere la definizione di `Foo::num_instances` modo che fosse definito solo una volta, ma in C ++ 17 le definizioni multiple della variabile inline `Foo::num_instances` riferiscono tutti allo stesso oggetto `int` .

```
// warning: not thread-safe...
class Foo {
  public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

Come caso speciale, un membro dati statico di `constexpr` è implicitamente in linea.

```
class MyString {
  public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};
// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

Leggi Variabili in linea online: https://riptutorial.com/it/cplusplus/topic/9265/variabili-in-linea

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con C ++	Adhokshaj Mishra, ankit dassor, aquirdturtle, ArchbishopOfBanterbury, Bakhtiar Hasan, Bart van Nierop, Ben H, Bo Persson, Brandon, Brian, BullshitPingu, cb4, celtschk, Cheers and hth. - Alf, chrisb2244, Cody Gray, Community, cpatricio, Curious, Daemon, Daksh Gupta, Danh, darkpsychic, David Bippes, David G., DeepCoder, Dim_ov, dlemstra, Donald Duck, Dr t, Dylan Little, Edward, emlai, Erick Q., ethanwu10, Fantastic Mr Fox, Florian, GIRISH kuniyal, greatwolf, honk, Humam Helfawi, Hurkyl, Ilyas Mimouni, Isak Combrinck, itzmukeshy7, Jason Watkins, JedaiCoder, Jerry Coffin, Jim Clark, Johan Lundberg, Jon Harper, jotik, Justin, Justin Time, JVApen, K48, Ken Y-N, Keshav Sharma, kiner_shah, krOoze, Leandros, maccard, Malcolm, Malick, Manan Sharma, manetsus, manlio, Marco A., Mark Gardner, MasterHD, Matt, Matt Lord, mnoronha, Muhammad Aladdin, Mustaghees, muXXmit2X, mynameisausten, Nathan Osman, Neil A., Nemanja Boric, neuro, Nicol Bolas, noɥʎdʎzɐɹƆ, Optimus Prime, Pavel Strakhov, Peter, Praetorian, Qchmqs, Quirk, RamenChef, Rushikesh Deshpande, SajithP, Sam Cristall, Serikov, Shoe, SirGuy, Soapy, Soul_man, theo2003, ɬolɐəz əɥɬ qoq, Tom K, TriskalJM, Trizzle, UncleZeiv, VermillionAzure, Walter, Wen Qin, Wexiwa, πάντα ῥεῖ, パスカル
2	Algoritmi di libreria standard	Ami Tavory, Barry, Daniel, Duly Kinsky, Edgar Rokyan, Guillaume Pascal, Jarod42, NinjaDeveloper, Patryk Obara, Peter, Riom
3	Allineamento	Brian, Marco A., Nicol Bolas
4	Altri comportamenti non definiti in C ++	didiz
5	Aritmetica in virgola mobile	Xirema
6	Array	Cheers and hth. - Alf, Isak Combrinck, manlio, Matthew Brien, Wen Qin, Wolf, ΦXocę Пepeúpa ツ
7	attributi	ibrahim5253, JVApen, Kerrek SB, MathSquared, SingerOfTheFall

8	auto	Artalus, Barry, celtschk, Daniele Pallastrelli, Edward, Igor Oks, Jarod42, Johan Lundberg, manlio, Yakk
9	C ++ 11 Modello di memoria	NonNumeric
10	C incompatibilità	パスカル
11	Campi bit	Ajay, Perette Barella
12	Categorie di valore	Barry, ChemiCalChems, Curious, fefe, Johannes Schaub - litb, mnoronha, Nicol Bolas, Praetorian, SirGuy
13	Classi / Strutture	Alexey Voytenko, anderas, aquirdturtle, Brian, callyalater, chrisb2244, Colin Basnett, Dan Hulme, darkpsychic, Dragma, Fantastic Mr Fox, Firas Moalla, Jarod42, Jerry Coffin, jotik, Justin Time, Kerrek SB, Nicol Bolas, Null, OliPro007, PcAF, Ph03n1x, pingul, Rakete1111, Sándor Mátyás Márton, Sergey , silvergasp, Skywrath, Yakk
14	Compilazione e costruzione	4444, Adhokshaj Mishra, Ami Tavory, ArchbishopOfBanterbury, Barry, Ben Steffan, celtschk, Curious, Donald Duck, Dr t, elvis.dukaj, Fantastic Mr Fox, Florian, greatwolf, Griffin, Isak Combrinck, Jahid, Jarod42, Jason Watkins, Johan Lundberg, jotik, Justin, Justin Time, JVApen, madduci, Malick, manetsus, manlio, Matt, Michael Gaskill, Morten Kristensen, MSD, muXXmit2X, n.m., Nathan Osman, Nemanja Boric, Peter, Quirk, Richard Dally, Sergey, Tharindu Kumara, Toby, Trygve Laugstøl, VermillionAzure
15	Comportamento definito dall'implementazione	2501, Bo Persson, Brian, Dutow, Jahid, Jarod42, jotik, Justin Time, lz96, manlio, Nicol Bolas, Peter
16	Comportamento indefinito	Ami Tavory, AndreiM, Ben Steffan, Brian, Cody Gray, cshu, Dovahkiin, Elias Kosunen, emlai, Emma X, FedeWar, fefe, ggrr, GIRISH kuniyal, Hiura, Jeremi Podlasek, Johannes Schaub - litb, JVApen, kd1508, Ken Y-N, manetsus, manlio, Marco A., Mat, mceo, Motti, Naor Hadar, nbro, Nicol Bolas, Peter, Rakete1111, ralismark, RamenChef, Sebastian Ärleryd, Tannin, Trevor Hickey, Tyler Durden
17	Comportamento non specificato	AndreiM, Brian, Jarod42, Yakk
18	Concorrenza con OpenMP	Andrea Chua, JVApen, Nicol Bolas, Sumurai8
19	Confronti affiancati di	wasthishelpful

	classici esempi C ++ risolti tramite C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17	
20	Const Correctness	amanuel2, Justin Time
21	constexpr	Ajay, Brian, diegodfrf, mtb, Null
22	Contenitori C ++	John DiFini
23	Controllo del flusso	anotherGatsby, Brian, JVApen, mkluwe, Qchmqs, RamenChef , Tejendra
24	Conversioni di tipo esplicito	4444, Brian, JVApen, Nikola Vasilev
25	Copia Elision	Nicol Bolas, TartanLlama
26	Copia vs Assegnazione	amanuel2, Roland
27	Costruire sistemi	Ami Tavory, celtschk, Florian, Jahid, Jason Watkins, Justin, JVApen, Nathan Osman, RamenChef, VermillionAzure
28	Data e ora usando intestazione	Edward, marcinj, Naor Hadar, RamenChef
29	decltype	Ajay
30	Digita la cancellazione	Brian, celtschk, greatwolf, Jarod42, Yakk
31	Digitare parole chiave	Brian, Justin Time, Omnifarious, RamenChef
32	eccezioni	Alexey Guseynov, Brian, callyalater, Dr t, Jahid, Jarod42, Johan Lundberg, jotik, Martin Ba, Nemanja Boric, Null, Peter, Rakete1111, Ronen Ness
33	Enumerazione	Denkkar, Fantastic Mr Fox, Jarod42, Nicol Bolas, SajithP, stackptr, T.C.
34	Errori comuni di compilazione / linker (GCC)	Asu, immerhart
35	Esempi di server client	Abhinav Gauniyal
36	Espressioni regolari	honk, Jonathan Mee, Justin, JVApen

37	File di intestazione	RamenChef, VermillionAzure
38	File I / O	anderas, ankit dassor, Anonymous1847, AProgrammer, Bakhtiar Hasan, bitek, Chachmu, ComicSansMS, didiz, Dietmar Kühl, Dr t, Emanuel Vintilă, Galik, honk, Hurkyl, Jérémie Bolduc, John Strood, JVApen, Loki Astari, manlio, Mathieu K., MikeMB, mindriot, Nicol Bolas, nwp, patmanpato, Rakete1111, RomCoo, Serikov, sheng09, shrike, svgspnr, Алексей Неудачин
39	Funzione C ++ "call by value" vs. "call by reference"	Error - Syntactical Remorse, Henkersmann
40	Funzione di sovraccarico	Bakhtiar Hasan, Barry, Cody Gray, CoffeeandCode, didiz, Galik, Jatin, Johannes Schaub - litb, Justin Time, JVApen, Rakete1111, Sean, Sumurai8, Tim Straubinger
41	Funzioni costanti dei membri della classe	Vijayabhaskarreddy CH, Yakk
42	Funzioni dei membri virtuali	0x5f3759df, Daksh Gupta, Johan Lundberg, Justin Time, Motti , Sergey, T.C.
43	Funzioni inline	amanuel2, Aravind .KEN, Bim, Brian, legends2k
44	Funzioni membro non statico	Justin Time, RamenChef
45	Funzioni speciali per gli utenti	Barry, krOoze, OliPro007, Reuben Thomas, TriskalJM
46	Futures e promesse	didiz, Nicol Bolas
47	Generazione di numeri casuali	Ha., manlio, merlinND, Sumurai8
48	Gestione della memoria	Andrei, Brian, callyalater, Daksh Gupta, Galik, JVApen, madduci, nnrales, RamenChef, ThyReaper
49	Gestione delle risorse	Anonymous1847
50	Idolo di Pimpl	Danh, Daniele Pallastrelli, emlai, Jordan Chapman, JVApen, manlio, Stephen Cross, Yakk
51	Implementazione del modello di progettazione in C ++	Antonio Barreto, datosh, didiz, Jarod42, JVApen, Nikola Vasilev
52	Inoltro perfetto	In silico, Johannes Schaub - litb, Nicol Bolas, Roland

53	Input / output di base in c ++	Daemon, Nicol Bolas, Владимир Стрелец
54	Internazionalizzazione in C ++	John Bargman
55	iteratori	Barry, chrisb2244, cute_ptr, Daniel Jour, Edgar Rokyan, EvgeniyZh, fbrereto, Gal Dreiman, Gaurav Kumar Garg, GIRISH kuniyal, honk, Hurkyl, JPNotADragon, JVApen, Mike H-R, Null, Oz., Sergey, Serikov, tilz0R, Yakk
56	Iterazione	Brian, Daniel Käfer, Emmanuel Mathi-Amorim, marquesm91, RamenChef
57	La regola del tre, cinque e zero	Adrien Descamps, Barry, ChrisN, hello, honk, Johan Lundberg , Justin Time, JVApen, Loki Astari, mpromonet, Nicol Bolas, Nirmal4G, NonNumeric, Null, Peter, relgukxilef, Scott Weldon, T.C., TriskalJM, Venemo
58	lambda	Adi Lester, Aganju, Ajay, alain, anderas, Andrea Corbelli, Barry, bcmpinc, Brian, Christopher Oezbek, Community, cpplearner, derekerdmann, Edd, Falias, Firas Moalla, honk, Jean-Baptiste Yunès, Johan Lundberg, Johannes Schaub - litb , John Slegers, JVApen, Loki Astari, Loufylouf, M. Viaz, Mike Dvorkin, Nicol Bolas, Patryk, Praetorian, Rakete1111, RamenChef, Ryan Haining, Sergio, Serikov, Snowhawk, teivaz, Yakk, ygram
59	Layout dei tipi di oggetto	Brian, Justin Time
60	letterali	Brian, Nikola Vasilev, RamenChef
61	Letterali definiti dall'utente	Brian, Cid1025, Jarod42, Roland, sigalor, sth
62	Lo standard ISO C ++	Bakhtiar Hasan, Barry, C.W.Holeman II, ComicSansMS, didiz, diegodfrf, Guillaume Pascal, Ivan Kush, Johan Lundberg, Justin Time, JVApen, manlio, Marco A., MSalters, Nicol Bolas, sth, vishal
63	Loops	ankit dassor, anotherGatsby, Barry, ChemiCalChems, Chris, ChrisN, Christian Rau, ColleenV, Debanjan Dhar, DrZoo, Edward, emlai, holmicz, honk, Johannes Schaub - litb, Justin Time, L.V.Rao, manlio, Nicholas, Nicol Bolas, Null, Ped7g, pmelanson, Pyves, Rakete1111, Sergey, sp2danny, user1336087, VladimirS, Yakk
64	Manipolatori di flusso	Nicol Bolas, Владимир Стрелец

65	Manipolazione bit	A. Sarid, Barry, Cody Gray, CroCo, FedeWar, Jarod42, JVApen, manlio, tambre, Tarod, Trevor Hickey, Алексей Неудачин
66	metaprogrammazione	anderas, Barry, Brian, celtschk, Colin Basnett, DawidPi, deepmax, dmi_, Holt, Jarod42, Justin, manlio, Matthieu M., Nicol Bolas, Oz., rhynodegreat, rtmh, sth, TartanLlama, Venki, W.F., ysdx, πάντα ῥεῖ
67	Metaprogrammazione aritmitica	Meena Alfons
68	Modelli	Barry, Benjy Kessler, Brian, callyalater, cb4, celtschk, CodeMouse92, Colin Basnett, DeepCoder, Diligent Key Presser, Eldritch Cheese, eXPerience, FedeWar, Gabriel, Greg, Holt, honk, J_T, Johannes Schaub - litb, Justin, JVApen, Loki Astari, M. Viaz, manlio, Maxito, MSalters, Nicol Bolas, Pontus Gagge, Praetorian, Rakete1111, Ricardo Amores, Ryan Haining, Sergey, SirGuy, Smeeheey, Sumurai8, user1887915, W.F., WMios, Wolf, πάντα ῥεῖ
69	Modelli di espressione	Ajay, BigONotation, celtschk, defube, Jarod42, Jonathan Lee, Roland, T.C., Yakk
70	Modello di modello curiosamente ricorrente (CRTP)	Barry, Brian, Gabriel, honk, Nicol Bolas, Ryan Haining
71	mutex	didiz, hyoslee, JVApen
72	Mutex ricorsivo	didiz
73	Namespace	anderas, Andrea Chua, Barry, Brian, DeepCoder, emlai, Isak Combrinck, Jarod42, Jérémy Roy, Johannes Schaub - litb, Julien-L, JVApen, Nicol Bolas, Null, Rakete1111, randag, Roland, T.C., tenpercent, Yakk
74	Oggetti callable	JVApen, turoni
75	Operatori di bit	Loki Astari, Mads Marquart, manlio, txtechhelp, Алексей Неудачин
76	Ordinamento	anatolyg, Barry, Daniel, Ivan Kush, maccard, manetsus, manlio, MikeMB, MKAROL, Nicol Bolas, Patrick, Ravi Chandra , SajithP, timrau, Trevor Hickey
77	Ottimizzazione	chema989, ralismark
78	Ottimizzazione in C	4444, JVApen, lorro, mindriot

	++	
79	Pacchetti di parametri	Marco A.
80	Parola chiave amico	Perette Barella, Sergey
81	parola chiave const	Barry, Jarod42, Jatin, Justin, Podgorskiy, tenpercent, ThyReaper
82	parola chiave mutevole	Barry, Community, Dean Seo, start2learn, T.C., tenpercent
83	parole	ADITYA, amanuel2, Brian, Danh, John London, Justin Time, JVApen, Kerrek SB, Loki Astari, manlio, Marco A., Nicol Bolas, OliPro007, Rakete1111, RamenChef, Roland, start2learn
84	Parole chiave di base	amanuel2, Brian, Kerrek SB, RamenChef
85	Parole chiave di dichiarazione variabile	Brian, RamenChef, start2learn
86	Piega le espressioni	AndyG, Barry, cpplearner, Firas Moalla, Marco A., Rakete1111, T.C., Yakk
87	Polimorfismo	A. Sarid, Christophe, Jarod42, Jeremi Podlasek, Justin Time, manlio
88	precedenza dell'operatore	an0o0nym, Brian, didiz, JVApen, start2learn, turoni
89	preprocessore	alain, callyalater, Cheers and hth. - Alf, CygnusX1, Fantastic Mr Fox, Fox, Francisco P., Ian Ringrose, immerhart, InitializeSahib, Johan Lundberg, Justin, Justin Time, Ken Y-N, Kieran Chandler, krOoze, manlio, Marco A., Maxito, n.m., Nicol Bolas, Peter, phandinhlan, Richard Dally, Sean, signal, silvergasp, Sumurai8, T.C., Tanjim Hossain, tenpercent, The Philomath, Владимир Стрелец, パスカル
90	profiling	Ami Tavory, paul-g
91	puntatori	Baron, daB0bby, FedeWar, Hindrik Stegenga, Nicol Bolas, Nitinkumar Ambekar, Pietro Saccardi, Reverie Wisp, West
92	Puntatori ai membri	John Burger, start2learn
93	Puntatori intelligenti	Abyx, Ajay, Alexey Voytenko, anderas, Barry, CaffeineToCode, Christopher Oezbek, Cody Gray, ComicSansMS, cpplearner, Daksh Gupta, Danh, Daniele Pallastrelli, DeepCoder, Edward, emlai, foxcub, Francis Cugler, honk, Jack Zhou, Jared Payne, Jarod42, Johan Lundberg, Johannes Schaub - litb, jotik, Justin

		, JVApen, Kerrek SB, King's jester, Loki Astari, manlio, Marco A., MC93, Menasheh, Meysam, PcAF, Rakete1111, Reuben Thomas, Richard Dally, rodrigo, Roland, sami1592, sth, Sumurai8, tysonite, user3684240, Xirema, Yakk
94	RAII: l'acquisizione delle risorse è inizializzata	Barry, defube, Jarod42, JVApen, Loki Astari, Niall, Nicol Bolas , RamenChef, Sumurai8, Tannin
95	Restituzione di diversi valori da una funzione	aaronsnoswell, Bakhtiar Hasan, Barry, bitek, celtschk, Christopher Oezbek, Community, DeepCoder, Dr t, Ela782, Fantastic Mr Fox, Galik, honk, J_T, Jarod42, Johan Lundberg, Johannes Schaub - litb, John Slegers, Jon Chesterfield, Kevin Katzke, Let_Me_Be, Loki Astari, M. Sadeq H. E., manetsus, Menasheh, Michael Gaskill, mnoronha, Niall, Nicol Bolas, Null, Peter, Rakete1111, Richard Forrest, Ryan Hilbert, Stephen, T.C., templatetypedef, tenpercent, user3384414, Yakk, Ze Rubeus, パスカル
96	Ricerca del nome dipendente dall'argomento	Fanael, Johannes Schaub - litb
97	Ricorsione in C ++	celtschk, R_Kapp
98	Riferimenti	Andrea Corbelli, Asu, Daksh Gupta, darkpsychic, rockoder
99	Risoluzione di sovraccarico	Barry, Brian, didiz, Johannes Schaub - litb
100	RTTI: informazioni di tipo run-time	Brian, deepmax, Pankaj Kumar Boora, Roland, Savas Mikail KAPLAN
101	Scopes	deepmax, Error - Syntactical Remorse
102	Semaforo	didiz
103	Semantica del valore e di riferimento	JVApen, Nicol Bolas
104	Separatori di cifre	diegodfrf, JVApen
105	SFINAE (Errore di sostituzione non è un errore)	Barry, Fox, Jarod42, Jason R, Jonathan Lee, Luc Danton, sp2danny, SU3, w1th0utnam3, Xosdy, Yakk
106	sindacati	manlio, ThyReaper, txtechhelp
107	Singleton Design Pattern	deepmax, Galik, Jarod42, Johan Lundberg, JVApen, Stradigos

108	Sovraccarico del modello di funzione	Johannes Schaub - litb, Kunal Tyagi, RamenChef
109	Sovraccarico dell'operatore	ArchbishopOfBanterbury, Archie Gertsman, Ates Goral, Barry, Brian, Candlemancer, chrisb2244, defube, enzom83, James Adkison, Rakete1111, Sergey, start2learn, Xeverous, Yakk
110	Specifiche di collegamento	Brian
111	Specifiers di classe di archiviazione	Brian, start2learn
112	Sposta semantica	Barry, Cheers and hth. - Alf, ChemiCalChems, David Doria, didiz, Guillaume Racicot, Justin Time, JVApen
113	static_assert	Jarod42, JVApen, lorro, Marco A., Richard Dally, T.C.
114	std :: Atomics	Nikola Vasilev
115	std :: coppia	Ajay, Bim, demonplus, kiner_shah, Nikola Vasilev, Ravi Chandra
116	std :: forward_list	Nikola Vasilev
117	std :: function: per avvolgere qualsiasi elemento che è callable	elimad, Evgeniy, Nicol Bolas, Tarod
118	std :: integer_sequence	Dietmar Kühl
119	std :: iomanip	kiner_shah, Nikola Vasilev, Yakk
120	std :: map	Andrea Corbelli, ankit dassor, ChrisN, CinCout, ComicSansMS, CygnusX1, davidsheldon, diegodfrf, Fantastic Mr Fox, foxcub, Galik, honk, jmmut, manetsus, manlio, Meysam, Naveen Mittal, Null, Peter, Richard Dally, rick112358 , Savan Morya, user1336087, vdaras, VolkA, Wyzard
121	std :: matrice	CinCout, Daksh Gupta, Dinesh Khandelwal, Error - Syntactical Remorse, Malcolm, Nikola Vasilev, plasmacel
122	std :: opzionale	Barry, diegodfrf, Jahid, Jared Payne, JVApen, Null, Yakk
123	std :: qualsiasi	demonplus, Marco A.
124	std :: set e std :: multiset	G-Man, JVApen, Mikitori

125	std :: string	1337ninja, 3442, Andrea Corbelli, Barry, Bim, caps, Christopher Oezbek, cpplearner, crea7or, Curious, drov, Edward, Emil Rowland, emlai, Fantastic Mr Fox, fbrereto, ggrr, Holt, honk, immerhart, Jack, Jahid, Jerry Coffin, Jonathan Mee , jotik, JPNotADragon, jpo38, Justin, JVApen, Ken Y-N, Leandros, Loki Astari, manetsus, manlio, Marc.2377, Matthew, Matthieu M., Meysam, Michael Gaskill, mpromonet, Niall, Null, Rakete1111, RamenChef, Richard Dally, SajithP, Serikov, sigalor, Skipper, Soapy, sth, T.C., Tharindu Kumara, Trevor Hickey, user1336087, user2176127, W.F., Wolf, Yakk
126	std :: variante	Yakk
127	std :: vector	2power10, A. Sarid, Aaron Stein, alain, Alex Logan, Ami Tavory, anatolyg, anderas, Andy, AndyG, arunmoezhi, Bakhtiar Hasan, Barry, Benjamin Lindley, bluefog, bone, CHess, CinCout, Cody Gray, Colin Basnett, ComicSansMS, Community, cute_ptr, Daksh Gupta, Daniel, Daniel Stradowski , Dario, David G., David Yaw, DeepCoder, diegodfrf, dkg, Dr t, Duly Kinsky, Ed Cottrell, Edward, ehudt, emlai, enrico.bacis, Falias, Fantastic Mr Fox, Fox, foxcub, gaazkam, Galik, gartenriese, granmirupa, Holt, honk, Hurkyl, iliketocode, immerhart, Isak Combrinck, Jarod42, Jason Watkins, JHBonarius, Johan Lundberg, John Slegers, jotik, jpo38, JVApen, Kevin Katzke, krOoze, Loki Astari, lordjohncena, manetsus, manlio, Marco A., Matt, Michael Gaskill, Misha Brukman, MotKohn, Motti, mtk, NageN, Niall, Nicol Bolas, Null, patmanpato, Paul Beckingham, paul-g, Ped7g, Praetorian, Pyves, R. Martinho Fernandes, Rakete1111, Randy Taylor, Richard Dally, Roddy, Romain Vincent, Rushikesh Deshpande , Ryan Hilbert, Saint-Martin, Samar Yadav, Samer Tufail, Sayakiss, Serikov, Shoe, silvergasp, Skipper, solidcell, Stephen, sth, strangeqargo, T.C., Tamarous, theo2003, Tom, towi, Trevor Hickey, TriskalJM, user1336087, user2176127, Vladimir Gamalyan, Wolf, Yakk
128	Stream C ++	Ami Tavory, didiz, JVApen, mpromonet, Sergey
129	Strumenti e tecniche di debug in C ++ per debugging e debug	Adam Trhon, JVApen, King's jester, Misgevolution
130	Strutture dati in C ++	Gaurav Sehgal
131	Strutture di sincronizzazione del filo	didiz, Galik, JVApen

132	Tecniche di refactoring	asantacreu, Cody Gray, Edward, Jarod42, JVApen, RamenChef
133	Test unitario in C ++	elvis.dukaj, VermillionAzure
134	The This Pointer	amanuel2, Justin Time, RamenChef
135	threading	Alejandro, amchacon, Brian, CaffeineToCode, ComicSansMS, Dair, defube, didiz, Diligent Key Presser, Galik, James Adkison, james large, Jason Watkins, Jeremi Podlasek, mpromonet, Niall, nwp, Rakete1111, Stephen Cross, Sumurai8, Yakk, ysdx, Yuushi
136	Tipi atomici	JVApen, Stephen
137	Tipi senza nome	jotik, Roland, ThyReaper
138	tipo deduzione	Barry, Brian, Emmanuel Mathi-Amorim
139	Tipo di inferenza	Andrea Chua, Jim Clark
140	Tipo di ritorno Covariance	Cheers and hth. - Alf, sorosh_sabz
141	Tipo di ritorno finale	Brian, define cindy const, Torbjörn
142	Tipo Tratti	Jarod42, silvergasp, TartanLlama
143	Typedef e digita alias	Brian
144	Una regola di definizione (ODR)	Brian, Jarod42
145	Utilizzando la dichiarazione	Brian
146	Utilizzando std :: unordered_map	tulak.hord
147	Variabili in linea	Brian