IBM

IBM
Rational Development Studio for i:
ILE C/C++ 解説書

7.1

SC88-4026-02
(英文原典：SC09-7852-02)

IBM

IBM
Rational Development Studio for i:
ILE C/C++ 解説書

7.1

SC88-4026-02
(英文原典：SC09-7852-02)

ご注意

本書および本書で紹介する製品をご使用になる前に、 377 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM i 7.1 (プログラム 5770-WDS) ILE C/C++ コンパイラーに適用されます。また、改訂版で断りがない限り、それ以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼働するとは限りません。また CISC モデルでは稼働しません。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC09–7852–02
IBM
Rational Development Studio for i:
ILE C/C++ Language Reference
7.1

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2010.4

© Copyright International Business Machines Corporation 1998, 2010.

目次

ILE C/C++ 解説書 (SC09-7852-01) . . . ix
本書の対象読者 ix
強調表示の規則 x
構文図の読み方 xi
前提条件および関連情報 xii

変更の要約 xv

第 1 章 スコープおよびリンケージ. 1
スコープ. 1
ブロック/ローカル・スコープ. 2
関数スコープ 3
関数プロトタイプ・スコープ 3
ファイル/グローバル・スコープ 3
C でのスコープの例 4
クラス・スコープ (C++ のみ) 5
ID のネーム・スペース 5
名前の隠蔽 (C++ のみ) 6
プログラム・リンケージ 7
内部結合. 8
外部結合. 8
リンケージなし 9
言語リンケージ 9
名前マングリング (C++ のみ) 10

第 2 章 字句エレメント. 13
トークン 13
キーワード 13
言語拡張のキーワード 14
ID 15
ID に使用される文字 15
予約済みの ID 16
__func__ 事前定義 ID 16
リテラル 17
整数リテラル 17
10 進整数リテラル 18
16 進整数リテラル 18
8 進整数リテラル 19
ブール・リテラル 19
浮動小数点リテラル 19
2 進浮動小数点リテラル 19
16 進浮動小数点リテラル. 20
10 進浮動小数点リテラル. 21
パック 10 進数リテラル 23
文字リテラル 23
ストリング・リテラル 24
ストリング連結 26
区切り子と演算子 26
代替トークン 27
ソース・プログラムの文字セット 28
マルチバイト文字 28
エスケープ・シーケンス 29
ユニコード規格 30
2 文字表記文字 31
3 文字表記 31
コメント 32

第 3 章 データ・オブジェクトおよび宣言 35
データ・オブジェクトおよびデータ宣言の概要 . . 35
データ・オブジェクトの概念 35
不完全型 36
互換型および複合型 37
データ宣言およびデータ定義の概要 37
仮定義 38
ストレージ・クラス指定子 39
自動ストレージ・クラス指定子 40
自動変数のストレージ期間 40
自動変数のリンケージ 40
static ストレージ・クラス指定子 40
静的変数のリンケージ 41
extern ストレージ・クラス指定子 41
外部変数のストレージ期間 42
外部変数のリンケージ 42
mutable ストレージ・クラス指定子 (C++ のみ) 42
register ストレージ・クラス指定子. 43
レジスター変数のストレージ期間 43
レジスター変数のリンケージ 44
__thread ストレージ・クラス指定子 44
型指定子 45
整数型 45
ブール型 46
浮動小数点型 47
実数浮動小数点型 47
文字型 48
void 型 48
算術型の互換性 (C のみ) 49
ユーザー定義型 49
構造体および共用体 49
構造体および共用体の型定義 50
メンバー宣言 50
柔軟な配列メンバー 51
ゼロ長配列メンバー 52
ビット・フィールド・メンバー 53
構造体および共用体変数宣言 54
単一ステートメントでの構造体および共用体の型および変数の定義 55
構造体および共用体メンバーへのアクセス . . 56
無名共用体 56
列挙型 57
列挙型定義 57
列挙型メンバー 58
列挙型変数宣言 59

© Copyright IBM Corp. 1998, 2010

単一ステートメントでの列挙型の型および変数の定義 59
構造体、共用体、および列挙型の互換性 (C のみ) 59
別々のソース・ファイル間での互換性 . . . 60
typedef 定義 60
typedef 定義の例. 61
型修飾子 62
__align 型修飾子. 63
__align 修飾子の使用例 64
const 型修飾子 65
restrict 型修飾子 66
volatile 型修飾子. 67
型属性 67
aligned 型属性 68
packed 型属性 69
transparent_union 型属性 (C のみ) 69

第 4 章 宣言子 71
宣言子の概要 71
宣言子の例 72
型名 73
ポインター 74
ポインター演算 75
型ベースの別名割り当て 76
ポインターの互換性 (C のみ) 77
配列 78
可変長配列 (C++ のみ) 79
配列の互換性 80
参照 (C++ のみ). 80
初期化指定子 81
初期化およびストレージ・クラス 82
自動変数の初期化 82
静的変数の初期化 82
外部変数の初期化 83
レジスター変数の初期化 83
構造体および共用体の初期化 83
列挙型の初期化 85
ポインターの初期化 85
配列の初期化 86
文字配列の初期化 86
多次元配列の初期化 87
参照の初期化 (C++ のみ). 88
直接バインディング 89
変数属性 89
aligned 変数属性. 91
packed 変数属性 92
mode 変数属性 92
weak 変数属性 92

第 5 章 型変換 95
算術変換とプロモーション 95
整数変換 96
ブール変換 96
浮動小数点の型変換 97
整数および浮動小数点拡張 97
左辺値から右辺値への変換 98
ポインター型変換 99
void* への変換 100
参照変換 (C++ のみ) 100
修飾変換 (C++ のみ) 101
関数引数変換 101

第 6 章 式と演算子 103
左辺値と右辺値. 103
1 次式. 104
名前 105
リテラル 105
整数定数式 105
ID 式 (C++ のみ) 106
括弧で囲んだ式 ( ) 107
スコープ・レゾリューション演算子 :: (C++ のみ) 108
関数呼び出し式. 109
メンバー式 109
ドット演算子 109
矢印演算子 -> 110
単項式. 110
増分演算子 ++ 111
減分演算子 -- 111
単項正演算子 +. 112
単項負演算子 -. 112
論理否定演算子 ! 112
ビット単位否定演算子 ~. 112
アドレス演算子 &. 113
間接演算子 * 113
typeid 演算子 (C++ のみ) 114
__alignof__ 演算子. 115
sizeof 演算子 116
__typeof__ 演算子 117
2 項式. 118
代入演算子 119
単純代入演算子 = ... 119
複合代入演算子. 120
乗算演算子 * 121
除法演算子 / 121
剰余演算子 % 121
加法演算子 + ... 121
減法演算子 - 122
ビット単位左および右シフト演算子 << >> . . 122
関係演算子 < > <= >= 123
等価演算子 == および非等価演算子 != 124
ビット単位 AND 演算子 & 125
ビット単位排他 OR 演算子 ^ 126
ビット単位包含 OR 演算子 | 126
論理 AND 演算子 && 127
論理 OR 演算子 ||. 127
配列添え字演算子 [ ]. 128
コンマ演算子 , ... 129
メンバーを指すポインター演算子 .* ->* (C++ のみ) 130
条件式. 131
C の条件式の型 132

C++ の条件式の型. 132
条件式の例 132
キャスト式 133
キャスト演算子 (). 133
static_cast 演算子 (C++ のみ) 135
reinterpret_cast 演算子 (C++ のみ) 136
const_cast 演算子 (C++ のみ) 137
dynamic_cast 演算子 (C++ のみ) 139
複合リテラル式. 140
new 式 (C++ のみ) 141
配置構文 142
new 演算子で作成されたオブジェクトの初期化 143
new 割り振り失敗の処理 144
delete 式 (C++ のみ) 145
throw 式 (C++ のみ) 146
演算子優先順位と結合順序 146

第 7 章 ステートメント 151

ラベル付きステートメント 151
式ステートメント 152
あいまいなステートメントの解決 (C++ のみ) 152
ブロック・ステートメント 153
ブロックの例 153
選択ステートメント 154
if ステートメント 154
if ステートメントの例 155
switch ステートメント 156
switch ステートメントの制約事項 158
switch ステートメントの例 158
繰り返しステートメント. 160
while ステートメント. 160
do ステートメント 161
for ステートメント 162
for ステートメントの例 162
分岐ステートメント 164
break ステートメント. 164
continue ステートメント. 164
continue ステートメントの例 164
return ステートメント 166
return ステートメントの例 166
goto ステートメント 167
ヌル・ステートメント 168

第 8 章 関数. 169

関数宣言および関数定義. 169
関数宣言 169
関数定義 170
関数宣言の例 171
関数定義の例 172
互換関数 (C のみ). 172
複数の関数宣言 (C++ のみ) 173
関数ストレージ・クラス指定子 174
static ストレージ・クラス指定子 174
extern ストレージ・クラス指定子. 174
関数指定子 176
inline 関数指定子 176
インライン関数のリンケージ 177
関数からの戻りの型指定子 180
関数からの戻り値 181
関数宣言子 181
パラメーター宣言 182
パラメーター型. 183
パラメーターの名前 184
関数仮パラメーター宣言における静的配列指標 (C のみ) 184
関数属性 185
const 関数属性 186
noinline 関数属性 187
pure 関数属性 187
weak 関数属性 187
main() 関数 188
関数呼び出し 189
値による受け渡し 189
参照による受け渡し 190
割り振りおよび割り振り解除関数 (C++ のみ) . . 192
C++ 関数のデフォルト引数 (C++ のみ) 193
デフォルト引数に関する制約事項. 194
デフォルト引数の評価 194
関数へのポインター 195

第 9 章 ネーム・スペース (C++ のみ) 197

ネーム・スペースの定義. 197
ネーム・スペースの宣言. 197
ネーム・スペース別名の作成 197
ネストされたネーム・スペースの別名の作成 . . . 198
ネーム・スペースの拡張. 198
ネーム・スペースと多重定義 199
名前なしネーム・スペース 200
ネーム・スペース・メンバー定義. 201
ネーム・スペースとフレンド 201
using ディレクティブ. 202
using 宣言とネーム・スペース. 203
明示的アクセス. 203

第 10 章 多重定義 (C++ のみ) 205

関数の多重定義. 205
多重定義関数に関する制約事項 206
演算子の多重定義 207
単項演算子の多重定義 208
増分演算子と減分演算子の多重定義 209
2 項演算子の多重定義 210
代入の多重定義. 211
関数呼び出しの多重定義. 212
添え字の多重定義 213
クラス・メンバー・アクセスの多重定義 . . . 214
多重定義解決 215
暗黙的変換シーケンス 216
標準変換シーケンス 216
ユーザー定義の変換シーケンス 217
省略符号変換シーケンス. 217
多重定義された関数のアドレスの解決 217

第 11 章 クラス (C++ のみ) 219
クラス型の宣言. 220
クラス・オブジェクトの使用 220
クラスおよび構造体 222
クラス名のスコープ 223
不完全なクラス宣言 224
ネスト・クラス. 224
ローカル・クラス 226
ローカル型名 227

第 12 章 クラスのメンバーおよびフレンド (C++ のみ) 229
クラス・メンバー・リスト 229
データ・メンバー 230
メンバー関数 231
インライン・メンバー関数 231
定数および volatile メンバー関数. 232
仮想メンバー関数 232
特殊メンバー関数 233
メンバー・スコープ 233
メンバーへのポインター. 234
this ポインター. 236
静的メンバー 239
静的メンバーでのクラス・アクセス演算子の使用 239
静的データ・メンバー 240
静的メンバー関数 242
メンバー・アクセス 243
フレンド 245
フレンド・スコープ 247
フレンド・アクセス 250

第 13 章 継承 (C++ のみ) 251
派生 253
継承されたメンバー・アクセス 256
protected メンバー 256
基底クラス・メンバーのアクセス制御 257
using 宣言とクラス・メンバー. 258
基底クラスおよび派生クラスからのメンバー関数の多重定義 259
クラス・メンバーのアクセスの変更 261
多重継承 262
仮想基底クラス. 263
マルチアクセス. 264
あいまいな基底クラス 265
名前の隠蔽 266
あいまいさと using 宣言 267
明確なクラス・メンバー. 267
ポインター型変換 268
多重定義解決 268
仮想関数 269
あいまいな仮想関数呼び出し 272
仮想関数アクセス 274
抽象クラス 274

第 14 章 特殊メンバー関数 (C++ のみ) 277
コンストラクターとデストラクターの概要. . . . 277
コンストラクター 279
デフォルト・コンストラクター 279
コンストラクターによる明示的初期化 280
基底クラスおよびメンバーの初期化 282
派生クラス・オブジェクトの構築順序 285
デストラクター. 286
疑似デストラクター 288
ユーザー定義の型変換 289
変換コンストラクター 290
明示的指定子 292
変換関数 292
コピー・コンストラクター 293
コピー代入演算子 295

第 15 章 テンプレート (C++ のみ) . . 297
テンプレート・パラメーター 298
型テンプレート・パラメーター 298
「非型」テンプレート・パラメーター 298
「テンプレート」テンプレート・パラメーター 299
テンプレート・パラメーターのデフォルト引数 299
テンプレート引数 300
テンプレート型引数 300
テンプレート非型引数 301
「テンプレート」テンプレート引数 303
クラス・テンプレート 304
クラス・テンプレートの宣言と定義 305
静的データ・メンバーとテンプレート 306
クラス・テンプレートのメンバー関数 306
フレンドとテンプレート. 307
関数テンプレート 308
テンプレート引数の推定. 309
「型」テンプレート引数の推定 312
「非型」テンプレート引数の推定. 313
多重定義関数テンプレート 314
関数テンプレートの部分選択 315
テンプレートのインスタンス化 316
暗黙のインスタンス生成. 316
明示的インスタンス生成. 318
テンプレートの特殊化 319
明示的特殊化 319
明示的特殊化の定義と宣言 321
明示的特殊化とスコープ. 321
明示的特殊化のクラス・メンバー. 321
関数テンプレートの明示的特殊化. 322
クラス・テンプレートのメンバーの明示的特殊化 322
部分的特殊化 324
部分的特殊化のテンプレート・パラメーターと引数リスト 325
クラス・テンプレートの部分的特殊化のマッチング. 326
名前のバインディングと従属名 326
型名キーワード. 327
修飾子としての template キーワード 328

第 16 章 例外処理 (C++ のみ) 331

try ブロック 331
ネストされた try ブロック 332
catch ブロック 333
関数 try ブロック・ハンドラー 334
catch ブロックの引数. 337
スローされる例外とキャッチされる例外のマッチング 337
キャッチの順序. 338
throw 式 339
例外の再スロー. 339
スタックのアンワインド. 341
例外指定 342
特殊例外処理関数 345
unexpected() 関数 345
terminate() 関数. 346
set_unexpected() 関数および set_terminate() 関数 347
例外処理関数を使用した例 347

第 17 章 プリプロセッサー・ディレクティブ 351

マクロ定義ディレクティブ 351
#define ディレクティブ 351
オブジェクト類似マクロ. 352
関数類似マクロ. 353
可変個引数マクロ拡張 356
#undef dディレクティブ 356
演算子 356
演算子 357
ファイル・インクルード・ディレクティブ. . . . 358
#include ディレクティブ. 358
データ管理ファイル内のソースのコンパイル時における #include ディレクティブの使用 . 358
統合ファイル・システム・ファイル内のソースのコンパイル時における #include ディレクティブの使用 360
#include_next ディレクティブ 361
条件付きコンパイル・ディレクティブ 362
#if および #elif ディレクティブ 363
#ifdef ディレクティブ 364
#ifndef ディレクティブ 364
#else ディレクティブ. 365
#endif ディレクティブ 365
メッセージ生成ディレクティブ 366
#error ディレクティブ 366
#warning ディレクティブ 367
#line ディレクティブ. 367
アサーション・ディレクティブ 368
ヌル・ディレクティブ (#) 369
プラグマ・ディレクティブ 369
_Pragma プリプロセス演算子 370

付録 A. ILE C 言語拡張 371

C89 への拡張としての C99 フィーチャー. . . . 371
GNU C との互換性のための拡張. 371
10 進浮動小数点サポートのための拡張. 372

付録 B. ILE C++ 言語拡張 373

一般的な IBM 拡張 373
C99 との互換性のための拡張 373
GNU C との互換性のための拡張. 374
GNU C++ との互換性のための拡張 374
10 進浮動小数点サポートのための拡張. 375

特記事項. 377

プログラミング・インターフェース情報 378
商標 379
業界標準 379

索引 381

ILE C/C++ 解説書 (SC09-7852-01)

本書、「*C/C++ 解説書*」では、C および C++ プログラム言語の構文、セマンティクス、および IBM インプリメンテーションについて説明します。構文とセマンティクスにより、プログラム言語の完全な仕様が構成されますが、完全なインプリメンテーションは、拡張機能を伴うため、その内容はそれぞれ異なる可能性があります。Standard C および Standard C++ の IBM インプリメンテーションは、プログラム言語の本質を立証するものであると同時に、発達と変化を左右する重要な要因であるプログラミング手法における実用面の配慮と進歩を反映しています。C および C++ の言語拡張機能もまた、絶えず変化している現代のプログラミング環境のニーズを反映しています。

本書の目的は、C および C++ 言語の説明について、これまでとは総合的に異なる観点から、移植性を強調するプログラミング・スタイルの使用を促進することにあります。 *Standard C* という語句は、C 言語、プリプロセッサー、およびランタイム・ライブラリーの現行の正式定義を表す総称用語です。 C 言語の新しい正式定義が、まだ古い定義のインプリメンテーションを使用中に発表されたため、この語句の意味はあいまいです。本書では、C89 言語レベルに準拠した C 言語について説明します。あいまいさと K&R C との混同を今後避けるために、本書では、標準 C を意味する場合には「*ISO C*」を使用し、「*標準 C*」という用語を避けます。「*K&R C*」という用語は、C 言語と、ISO C より前に使用されていた Brian Kernighan および Dennis Ritchie (K&R C) によって作成され、一般的に受け入れられている拡張を指すために使用します。「*標準 C++*」という表現は、最新の承認されている C++ 言語標準 (ISO/IEC 14882:2003) を指します。

本書で説明する C および C++ 言語は、以下の標準に基づきます。
- Information Technology - Programming languages - C, ISO/IEC 9899:1990 (C89 とも呼ばれる)
- Information Technology - Programming languages - C, ISO/IEC 9899:1999 (C99 とも呼ばれる)
- Information Technology - Programming languages - C++, ISO/IEC 14882:1998 (C++98 とも呼ばれる)
- Information Technology - Programming languages - C++, ISO/IEC 14882:2003(E) (標準 C++ とも呼ばれる)
- Information Technology - Programming languages - Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732。このドラフトのテクニカル・レポートは、C 標準委員会に提出されており、 http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf で入手できます。

本書には、C99 の参照が含まれています。 C99 の関数は、ILE C++ によってのみサポートされており、ILE C によってはサポートされていません。 ILE C は、C89 言語レベルおよび一部の拡張をサポートしています。

400 本書で説明する C++ 言語は、Standard C++ と整合性があり、IBM C++ コンパイラーがサポートするフィーチャーを文書化しています。

本書の対象読者

本書は、C または C++ でのアプリケーションのプログラミング経験が既にあるユーザーを対象にした解説書です。 C または C++ の初心者でも、ILE C/C++ に固有の言語およびフィーチャーに関する情報を得るために本書を使用できますが、本書は、プログラミングの概念を教授したり、特定のプログラミング手法を勧めたりすることを目的とはしていません。

© Copyright IBM Corp. 1998, 2010

本書は、C 言語および C++ 言語の基礎および応用に焦点を当てています。特定言語レベルでの特定言語フィーチャーの可用性は、コンパイラー・オプションによって制御されます。コンパイラー・オプションから提供される広範囲な機能の可能性については、「*ILE C/C++ コンパイラー参照*」で説明されています。

説明が深いレベルに及ぶため、これまでに C またはその他のプログラム言語についてある程度経験していることが前提になります。本書は、良質のプログラムを作成するために役立つ各言語インプリメンテーションの構文およびセマンティクスを読者に提供することを目標にしています。プログラミング・スタイルの特定の規則が秩序立ったプログラムの作成に役立つものであっても、コンパイラーはその規則を強制しません。

言語仕様に厳密に準拠するプログラムは、異なる環境間で最大の移植性を発揮します。理論上、標準に準拠した、あるコンパイラーで正しくコンパイルされたプログラムは、ハードウェアの差異が許す範囲内で、他のすべての標準準拠のコンパイラーの下で正確にコンパイルされ、作動します。言語インプリメンテーションによって提供される言語への拡張子を正しく活用したプログラムは、そのオブジェクト・コードの効率性を向上させることができます。

強調表示の規則

太字
: コマンド、キーワード、ファイル、ディレクトリー、およびその名前がシステムによって事前定義されるその他の項目を識別します。

イタリック
: その実際の名前または値がプログラマーによって提供されるパラメーターを識別します。*イタリック* は、初出の新規用語を表すときに使用します。

`例`
: 特定のデータ値の例、ユーザーに表示されるテキストに類似したテキストの例、プログラム・コードの部分、システムからのメッセージ、またはユーザーが実際に入力するべき情報の例などを識別します。

これらの例は、言語の使用方法を説明するもので、実行時間の最小化、ストレージの節約、エラーのチェックを行うためのものではありません。これらの例は、可能な言語構成の使用をすべて例示しているわけではありません。例の中には、コードの一部分だけを示し、コードを追加しないとコンパイルできないものもあります。

本書は、テキストの大きなブロックを示すために、マークされた大括弧の区切り記号を使用し、以下のように、小さなセグメントのテキストを示すためにアイコンを使用します。

表 1. 修飾要素

修飾子/アイコン	意味
C のみ ▶ C	テキストは、C 言語でのみサポートされるフィーチャーについて説明します。または、C 言語に固有の動作について説明します。
C++ のみ ▶ C++	テキストは、C++ 言語でのみサポートされるフィーチャーについて説明します。または、C++ 言語に固有の動作について説明します。
IBM 拡張 ▶ 400	テキストは、標準言語仕様への IBM コンパイラーの拡張であるフィーチャーを説明します。

構文図の読み方

- 構文図は、左から右、上から下に、線のパスに従って読んでください。

  ▸▸─── は、コマンド、ディレクティブ、またはステートメントの先頭を示します。

  ───▸ は、コマンド、ディレクティブ、またはステートメント構文が、次の行に続いていることを示します。

  ▸─── は、コマンド、ディレクティブ、またはステートメントが、前の行からに続いていることを示します。

  ───▸◂ は、コマンド、ディレクティブ、またはステートメントの終わりを示します。

  完全なコマンド、ディレクティブ、またはステートメント以外の構文単位の図は、 ▸─── 記号で始まり、───▸ 記号で終わります。

  注: 以下のダイアグラムで、`statement` は、C または C++ コマンド、ディレクティブ、またはステートメントを表しています。

- 必須項目は、水平線 (メインパス) 上に記述されます。

```
▸▸── ステートメント──required_item──────────────────────────▸◂
```

- オプション項目は、メインパスの下に記述されます。

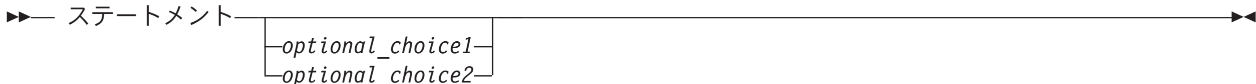
```
▸▸── ステートメント──┬───────────────┬──────────────────────▸◂
                     └─optional_item─┘
```

- 2 つ以上の項目から選択可能な場合は、スタック内に垂直に記述されます。

  いずれか 1 つの項目の選択が*必須* の場合は、スタック内の項目のいずれか 1 つがメインパス上に記述されます。

```
▸▸── ステートメント──┬─required_choice1─┬──────────────────▸◂
                     └─required_choice2─┘
```

  いずれか 1 つの項目の選択がオプションの場合は、スタック全体がメインパスの下に記述されます。



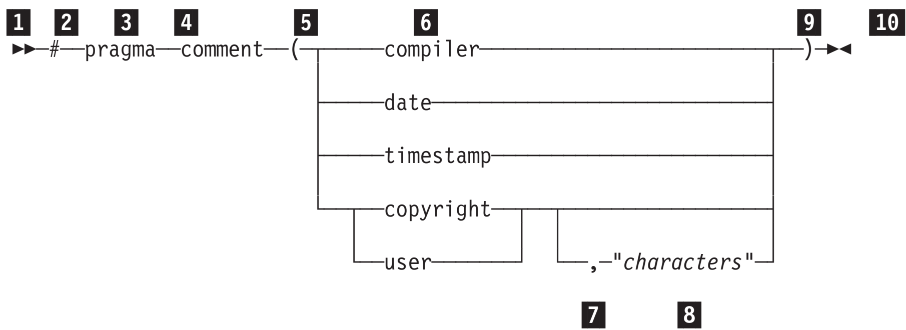
  デフォルト項目は、メインパスの上に記述されます。

- メインパスの線の上の左に戻る矢印は、繰り返し可能な項目を示します。

スタックの上の繰り返し矢印は、スタック内の項目から複数の項目を選択するか、1 つの項目を繰り返し選択できることを示しています。
- キーワードは、非イタリック体で記述されています。示されているとおりに正確に入力する必要があります (例えば extern)。

  変数は、イタリック体の小文字で記述されます (例えば、*identifier*)。これらはユーザーが指定する名前または値を表しています。
- 構文図に、句読記号、小括弧、算術演算子、または、ほかの非英数字文字が示されている場合は、構文の一部としてこれらの文字を入力する必要があります。

次の構文図の例では、**#pragma comment** ディレクティブの構文を示しています。 **#pragma** ディレクティブの詳細については、 369 ページの『プラグマ・ディレクティブ』を参照してください。



1 構文図の始まりを示します。

2 記号 # を最初に記述します。

3 キーワード pragma は、シンボル # の次に記述されます。

4 プラグマの名前 comment は、キーワード pragma の次に記述します。

5 左括弧が必要です。

6 コメントの型を、表示されている compiler、date、timestamp、copyright、または user のうちいずれか 1 つだけ入力します。

7 コンマが、コメントの型 copyright または user とオプションの文字ストリングの間に必要です。

8 文字ストリングをコンマの次に記述します。文字ストリングは、二重引用符で囲みます。

9 右小括弧は必須です。

10 これが、構文図の終わりを示します。

次の **#pragma comment** ディレクティブの例は、上記のダイアグラムに従っており、構文上正しい例です。

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

前提条件および関連情報

IBM i Information Center は、IBM i の技術情報の開始点として使用してください。

Information Center は、以下のようにアクセスできます。
- 以下の Web サイトから。

  http://www.ibm.com/systems/i/infocenter/

IBM i Information Center には、ソフトウェアおよびハードウェア・インストール、
Linux、WebSphere、Java、高可用性、データベース、論理区画、CL コマンド、およびシステムのアプリケーション・プログラミング・インターフェース (API) などの新規および更新システム情報が含まれています。また、システムのハードウェアおよびソフトウェアを計画、トラブルシューティング、および構成するのに役立つアドバイザーや検索機能も提供されています。

変更の要約

このエディションでの情報の変更は、以下のとおりです。

- C++ の10 進浮動小数点サポート
- C++ に追加された C99 フィーチャー

第 1 章 スコープおよびリンケージ

スコープ とは、名前を修飾しなくてもエンティティーを参照できるプログラム・テキスト内の最大の領域、つまり名前が潜在的に有効である最大領域のことです。広い意味で言えば、スコープは、エンティティー名の意味を差別化するために使用される一般的なコンテキストです。 スコープの規則をネーム・レゾリューションの規則と組み合わせることで、コンパイラーはファイル内の特定の場所で ID への参照が有効であるかどうかを判別できます。

宣言のスコープと ID の可視性は、互いに関連していますが、それぞれ異なる概念です。スコープは、プログラムの中の宣言の可視性を制限できるメカニズムのことです。 ID が*可視* であることは、ID に関連付けられたオブジェクトを含むプログラム・テキストの領域に合法的にアクセスできるということです。スコープが可視性より優先することはありますが、可視性がスコープより優先することはありません。内部宣言領域で重複 ID が使用されており、そのことによって外部宣言領域で宣言されたオブジェクトが隠蔽されている場合は、スコープが可視性よりも優先します。重複 ID のスコープ (2 番目のオブジェクトの存続期間) が終了しない限り、元の ID を使用して、最初のオブジェクトにアクセスすることはできません。

このように、ID のスコープは、識別されたオブジェクトの*ストレージ期間*、つまり、識別されたストレージ領域にオブジェクトが存在し続ける時間と相互に関係があります。オブジェクトの存続期間はストレージ期間の影響を受け、ストレージ期間はオブジェクト ID のスコープの影響を受けます。

リンケージ とは、複数の変換単位間または単一の変換単位内で名前を使用すること、または使用できることを意味します。*変換単位* という用語は、ソース・コード・ファイル、`#include` ディレクティブによるプリプロセス後に追加されるすべてのヘッダーおよびその他のファイルから、条件つきプリプロセス指令によってスキップされるソース行を除いたものを意味します。リンケージを使用すると、ID の各インスタンスに特定のオブジェクトまたは関数を正しく関連付けることができます。

スコープとリンケージの違いは、スコープはコンパイラーが利用し、リンケージはリンカーが利用する点です。ソース・ファイルをオブジェクト・コードに変換するときに、コンパイラーは外部結合を持つ ID を追跡し、最終的にオブジェクト・ファイル内のテーブルに格納します。これにより、リンカーは外部結合を持つ名前を判別することができますが、内部結合を持つ名前およびリンケージを持たない名前に関する情報は認知できません。

さまざまなタイプのスコープ間の区別については、『スコープ』に説明されています。さまざまなタイプのリンケージについては、 7 ページの『プログラム・リンケージ』に説明されています。

関連情報

- 39 ページの『ストレージ・クラス指定子』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

スコープ

ID の*スコープ* とは、ID を使用してオブジェクトを参照することができるプログラム・テキストの最大領域のことです。 C++ では、参照されるオブジェクトは固有のものでなければなりません。ただし、そのオブジェクトにアクセスするための名前、すなわち、ID 自体は再使用できます。ID の意味は、その ID が使用されるコンテキストに依存します。スコープは、名前の意味を差別化するために使用される一般的なコンテキストです。

© Copyright IBM Corp. 1998, 2010

ID のスコープは不連続になる場合があります。不連続なスコープが生じるケースの 1 つは、別のエンティティーを宣言するために同じ名前を再使用して、格納対象の宣言領域 (内部) と格納先の宣言領域 (外部) を作成した場合です。したがって、宣言する場所がスコープを決定する要因になります。不連続スコープが可能であることが、*情報の隠蔽* と呼ばれる手法の基礎です。

C におけるスコープの概念は、C++ で拡張および洗練化されています。次の表に、スコープの種類、および用語の若干の違いを示します。

表 2. スコープの種類

C	C++
ブロック	ローカル
関数	関数
関数プロトタイプ	関数プロトタイプ
ファイル	グローバル・ネーム・スペース
	ネーム・スペース
	クラス

すべての宣言において、ID は、初期化指定子の前のスコープにあります。次の例は、このことを示しています。

```
int x;
void f() {
  int x = x;
}
```

関数 `f()` の中で宣言された `x` は、ローカル・スコープを持っています (グローバル・スコープではありません)。

関連情報

- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

ブロック/ローカル・スコープ

名前がブロックで宣言される場合は、その名前には*ローカル・スコープ* または*ブロック・スコープ* があります。ローカル・スコープがある名前は、そのブロック、およびそのブロック内で囲まれたブロックで使用できますが、使用する前に名前を宣言する必要があります。ブロックを終了すると、ブロックに宣言された名前は、使用できなくなります。

関数のパラメーター名には、その関数の最外部ブロックのスコープがあります。さらに、関数が宣言されていて、定義されていない場合、これらのパラメーター名は、関数プロトタイプ・スコープを持っています。

あるブロックが別のブロックの内側でネストされると、外部ブロックからの変数は、通常、ネストされたブロック内で可視になります。ただし、ネストされたブロック内の変数の宣言が、囲みブロック内で宣言されている変数と同じ名前を持っている場合、ネストされたブロック内の宣言は、囲みブロック内で宣言された変数を隠蔽します。オリジナルの宣言は、プログラム制御が外部ブロックに戻されるときに、復元されます。これは、*ブロックの可視性* と呼ばれます。

ローカル・スコープ内のネーム・レゾリューションは、名前が使用されている即時に囲むスコープ内から開始され、次に、外側の囲みスコープを処理します。ネーム・レゾリューション中のスコープの検索順序によって、情報隠蔽の現象が生じます。囲みスコープ内の宣言は、ネストされたスコープ内に同じ ID の宣言がある場合、この宣言によって隠蔽されます。

関連情報

- 153 ページの『ブロック・ステートメント』

関数スコープ

関数スコープ 付きの ID の唯一の型は、ラベル名です。ラベルは、その出現によってプログラムのテキスト内で暗黙的に宣言され、ラベルが宣言された関数内で可視になります。

実際のラベルが見える前に、goto ステートメントの中で、そのラベルを使用することができます。

関連情報

- 151 ページの『ラベル付きステートメント』

関数プロトタイプ・スコープ

関数宣言 (*関数プロトタイプ*ともいう) の中、または任意の関数宣言子 (関数定義の宣言子を除く) の中では、パラメーター名は*関数プロトタイプ・スコープ* を持っています。関数プロトタイプ・スコープは、最も近い囲み関数宣言子の終わりで終了します。

関連情報

- 169 ページの『関数宣言』

ファイル/グローバル・スコープ

C のみ

ID の宣言がすべてのブロックの外側で現れる場合は、名前に、*ファイル・スコープ* があります。ファイル・スコープと内部結合付きの名前は、名前が宣言された位置から変換単位の終わりまで可視になります。

C のみ の終り

C++ のみ。

グローバル・スコープ または*グローバル・ネーム・スペース・スコープ* は、オブジェクト、関数、型、およびテンプレートを定義できるプログラムの最外部のネーム・スペース・スコープです。 ID の宣言が、すべてのブロック、ネーム・スペース、およびクラスの外側で現れる場合は、名前に*グローバル・ネーム・スペース・スコープ*があります。

グローバル・ネーム・スペース・スコープと内部結合付きの名前は、名前が宣言された位置から変換単位の終わりまで可視になります。

また、グローバル (ネーム・スペース) スコープ付きの名前は、グローバル変数を初期化するためにアクセス可能です。その名前が extern で宣言される場合は、リンク時に、リンク中のすべてのオブジェクト・ファイルで可視になります。

ネーム・スペース定義を使用して、グローバル・スコープ内でユーザー定義のネーム・スペースをネストすることができます。ユーザー定義のネーム・スペースは、それぞれグローバル・スコープと区別される別のスコープになります。

C++ のみ。 の終り

関連情報

- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』
- 8 ページの『内部結合』

C でのスコープの例

次の例では、2 行目で宣言する変数 x とは異なる変数 x を 1 行目で宣言します。2 行目で宣言される変数には、関数プロトタイプ・スコープがあり、プロトタイプ宣言の右小括弧までだけが可視になります。1 行目で宣言された変数 x の可視性は、プロトタイプの宣言の終了後に再び有効になります。

```
1   int x = 4;            /* variable x defined with file scope */
2   long myfunc(int x, long y); /* variable x has function      */
3                               /* prototype scope              */
4   int main(void)
5   {
6      /* . . . */
7   }
```

次のプログラムでは、ブロック、ネスティング、スコープを説明します。この例では、ファイル・スコープおよびブロック・スコープの 2 種類のスコープを示します。main 関数は、値 1、2、3、0、3、2、1 を別々の行に印刷します。i の各インスタンスは、異なる変数を表します。

```
            #include <stdio.h>
            int i = 1;                         /* i defined at file scope */

            int main(int argc, char * argv[])
 ┌─────     {
 1
 1             printf("%d¥n", i);              /* Prints 1 */
 1
 1  ┌───       {
 1  2             int i = 2, j = 3;            /* i and j defined at block scope */
 1  2                                         /* global definition of i is hidden */
 1  2             printf("%d¥n%d¥n", i, j);    /* Prints 2, 3 */
 1  2
 1  2 ┌──         {
 1  2 3              int i = 0;                /* i is redefined in a nested block     */
 1  2 3                                        /* previous definitions of i are hidden */
 1  2 3              printf("%d¥n%d¥n", i, j); /* Prints 0, 3 */
 1  2 └──         }
 1  2
 1  2             printf("%d¥n", i);           /* Prints 2 */
 1  2
 1  └───       }
 1
 1             printf("%d¥n", i);              /* Prints 1 */
 1
 1             return 0;
 1
 └──────    }
```

クラス・スコープ (C++ のみ)

メンバー関数内で宣言された名前は、そのスコープがメンバー関数のクラスの終わりまで広がっているか、または終わりを通過している、同じ名前の宣言を隠蔽します。

宣言のスコープがクラス定義の終わりまで及んでいるか、または終わりを通過している場合、そのクラスのメンバー定義によって定義されている領域は、このクラスのスコープに含まれます。クラス外部で字句的に定義されたメンバーも、このスコープに含まれます。さらに、宣言のスコープは、メンバー定義内の該当の ID に続く宣言子のどの部分も含みます。

クラス・メンバーの名前には、クラス・スコープ があり、次のケースでのみ使用できます。

- そのクラスのメンバー関数内
- そのクラスから派生したクラスのメンバー関数内
- そのクラスのインスタンスに適用された . (ドット) 演算子の後
- そのクラスから派生したクラスのインスタンスに適用された . (ドット) 演算子の後 (派生したクラスが名前を隠蔽しない場合)
- そのクラスのインスタンスを指すポインターに適用された -> (矢印) 演算子の後
- そのクラスから派生したクラスのインスタンスを指すポインターに適用された -> (矢印) 演算子の後 (派生したクラスが名前を隠蔽しない場合)
- クラスの名前に適用された :: (スコープ・レゾリューション) 演算子の後
- そのクラスから派生したクラスに適用された :: (スコープ・レゾリューション) 演算子の後

関連情報

- 219 ページの『第 11 章 クラス (C++ のみ)』
- 223 ページの『クラス名のスコープ』
- 233 ページの『メンバー・スコープ』
- 247 ページの『フレンド・スコープ』
- 257 ページの『基底クラス・メンバーのアクセス制御』
- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』

ID のネーム・スペース

ネーム・スペースは、ID を使用できるさまざまな構文コンテキストです。同じコンテキスト内および同じスコープ内では、ID によってエンティティーを一意的に識別できる必要があります。ここで使用する用語 ネーム・スペース は、C および C++ に適用され、C++ ネーム・スペース言語フィーチャーを指すものではありません。コンパイラーは、ネーム・スペース を設定して、種類が異なるエンティティーを参照する ID を区別します。異なるネーム・スペース内に同一 ID が存在する場合、これらの ID は同じスコープ内にあっても互いに干渉しません。

各 ID がそのネーム・スペース内で固有である場合には、同じ ID を使用して、別のオブジェクトを宣言することができます。 プログラム内の ID の構文上のコンテキストによって、コンパイラーは、そのネーム・スペースを明確に解決します。

次の 4 つの各ネーム・スペース内では、ID を固有にする必要があります。

- 次の型のタグ は、単一スコープ内で固有にする必要があります。
  - 列挙
  - 構造体および共用体

- 構造体、共用体、およびクラスのメンバー は、単一の構造体、共用体、またはクラスの型内で固有にする必要があります。
- *ステートメント・ラベル* には、関数スコープがあり、1 つの関数内で固有にする必要があります。
- 次に示すほかのすべての*通常 ID* は、単一のスコープ内で固有にする必要があります。
  - C 関数名 (C++ 関数名は、多重定義できる)
  - 変数名
  - 関数仮パラメーターの名前
  - 列挙型定数
  - typedef 名

ID は、同じネーム・スペース内で再定義できますが、閉じたプログラム・ブロック内で再定義します。

構造体タグ、構造体メンバー、変数名、およびステートメント・ラベルは、4 つの異なるネーム・スペースにあります。つまり、次の例の student 項目間では、名前の競合は発生しません。

```
int get_item()
{
   struct student       /* structure tag */
   {
      char student[20];  /* structure member */
      int section;
      int id;
   } student;           /* structure variable */

   goto student;
   student:;            /* null statement label  */
   return 0;
   }
```

コンパイラーは、student の各オカレンスを、プログラム内でのそのコンテキストによって解釈します。つまり、student は、キーワード struct の後にあるときは構造体タグであり、student 型を定義するブロックにあるときは構造体メンバー変数であり、構造体定義の終わりにあるときは構造体変数を宣言するものであり、goto ステートメントの後にあるときはラベルです。

名前の隠蔽 (C++ のみ)

クラス名または列挙名がスコープ内にあって、隠蔽されていなければ、それは*可視* です。クラス名または列挙名は、その同じ名前を (オブジェクト、関数、または列挙子として) ネストされた宣言領域または派生クラスの中で明示的に宣言することによって、隠蔽できます。クラス名または列挙名は、オブジェクト、関数、または列挙子の名前が可視である場所では、どこでも隠蔽されます。このプロセスは、*名前の隠蔽* と呼ばれています。

メンバー関数定義内で、ローカル名を宣言すると、同じ名前のクラスのメンバーの宣言を隠蔽します。派生クラス内でメンバーを宣言すると、同じ名前の基底クラスのメンバーの宣言を隠蔽します。

名前 x が、ネーム・スペース A のメンバーであると想定します。また、ネーム・スペース A のメンバーは、ネーム・スペース B で可視であると想定します (using の宣言のために)。ネーム・スペース B 内で x という名前のオブジェクトを宣言すると、A::x は隠蔽されます。次の例は、このことを示しています。

```
#include <iostream>
#include <typeinfo>
using namespace std;

namespace A {
  char x;
};
```

```
namespace B {
  using namespace A;
  int x;
};

int main() {
  cout << typeid(B::x).name() << endl;
}
```

次に、上記の例の出力を示します。

```
int
```

ネーム・スペース B 内での整数 x の宣言は、 using 宣言によって導入された文字 x を隠蔽します。

関連情報

- 219 ページの『第 11 章 クラス (C++ のみ)』
- 231 ページの『メンバー関数』
- 233 ページの『メンバー・スコープ』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

プログラム・リンケージ

リンケージ は、同一の名前を持つ複数の ID が、たとえこれらの ID が異なる変換単位に現れたとしても、同じオブジェクト、関数、または他のエンティティーを指しているかどうかを判別します。 ID のリンケージは、それがどのように宣言されていたかによって異なります。 リンケージには次の 3 つのタイプがあります。

- 内部結合: ID は、変換単位内でのみ参照することができます。
- 外部結合: ID は、他の変換単位内で参照することができます。
- リンケージなし: ID は、定義元のスコープ内でのみ参照することができます。

リンケージはスコープには影響しません。通常の名前ルックアップに関する考慮事項が適用されます。

C++ のみ。

異なるプログラミング言語で作成された変換単位間に、*言語リンケージ* と呼ばれるリンケージを設定することができます。言語リンケージを使用すると、ある ILE 言語のコードを別の ILE 言語で作成されたコードにリンクできるようにして、すべての ILE 言語の間の関係をクローズ状態にすることができます。 C++ では、すべての ID にデフォルトで C++ の言語リンケージが設定されています。言語リンケージは、異なる変換単位間で整合している必要があります。 C 以外または C++ 以外の言語リンケージは、ID は外部リンケージを持つことを意味します。 IBM i 固有の使用法については、「*ILE C/C++ プログラマーの手引き*」の第 25 章『ILE 呼び出し規則』を参照してください。

C++ のみ。 の終り

関連情報

- 40 ページの『static ストレージ・クラス指定子』
- 41 ページの『extern ストレージ・クラス指定子』
- 174 ページの『関数ストレージ・クラス指定子』
- 62 ページの『型修飾子』

- 56 ページの『無名共用体』

内部結合

次の種類の ID は、内部結合を持っています。

- static と明示的に宣言されたオブジェクト、参照、または関数。
- 指定子 const を使用して、ネーム・スペース・スコープ (または C のグローバル・スコープ) 内で宣言されているが、extern と明示的に宣言されておらず、以前に外部結合を持っていると宣言されていない、オブジェクトまたは参照。
- 無名共用体のデータ・メンバー。
- C++ 明示的に static として宣言されている関数テンプレート。
- C++ 名前なしのネーム・スペース内で宣言された ID。

ブロック内部で宣言された関数は、通常外部結合を持っています。ブロック内部で宣言されたオブジェクトは、extern と指定されていれば、通常外部結合を持っています。 static ストレージを持っている変数が、関数の外で定義されている場合、その変数は、内部結合を持っていて、定義された位置から現行変換単位の終わりまで有効です。

ID の宣言にキーワード extern があり、 ID の直前の宣言がネーム・スペースまたはグローバル・スコープで可視になっている場合は、 ID は、最初の宣言と同じリンケージを持っています。

外部結合

C のみ

グローバル・スコープでは、static ストレージ・クラス指定子なしで宣言された、以下の種類のエンティティーに対する ID は外部リンケージを持ちます。

- オブジェクト
- 関数

C 内の ID が extern キーワードを宣言されても、同じ ID を使ったオブジェクトまたは関数の以前の宣言が可視になっている場合は、2 番目の ID は、最初の宣言と同じリンケージを持ちます。例えば、キーワード static によって最初に宣言され、キーワード extern によって後で宣言された変数または関数には、内部結合があります。ただし、リンケージを持たずに、後で結合指定子を使用して宣言された変数または関数は、明白に指定されたリンケージを持ちます。

C のみ の終り

C++ のみ。

ネーム・スペース・スコープでは、次にあげる種類のエンティティの ID は、外部結合を持っています。

- 内部結合を持たない参照またはオブジェクト。
- 内部結合を持たない関数。
- 名前付きのクラスまたは列挙。
- typedef 宣言で定義された名前なしクラスまたは列挙。
- 外部結合を持っている列挙の列挙子。
- 内部結合を持つ関数テンプレートでない場合のテンプレート。

- 名前なしネーム・スペース内で宣言されていない場合のネーム・スペース。

クラスの ID が外部結合を持っている場合は、そのクラスのインプリメンテーションでは、以下のものの ID も外部結合を持ちます。

- メンバー関数。
- 静的データ・メンバー。
- クラス・スコープのクラス。
- クラス・スコープの列挙。

C++ のみ。 の終り

リンケージなし

次の種類の ID には、リンケージがありません。

- 外部結合も内部結合も持たない名前
- ローカル・スコープで宣言された名前 (extern キーワードを使用して宣言されたエンティティーのような例外はあります)
- オブジェクトまたは関数を表さない ID、インクルード・ラベル、列挙子、リンケージを持たないエンティティーを指す typedef 名、型名、関数仮パラメーター、およびテンプレート名

リンケージを持たない名前を使用して、リンケージを持つエンティティーを宣言することはできません。例えば、リンケージを持たないエンティティーを指す構造体または列挙の名前、あるいは typedef 名を使用して、リンケージを持つエンティティーを宣言することはできません。次の例は、このことを示しています。

```
int main() {
  struct A { };
//  extern A a1;
  typedef A myA;
//  extern myA a2;
}
```

コンパイラーは、外部結合を持つ a1 の宣言を許可しません。構造体 A は、リンケージを持っていません。コンパイラーは、外部結合を持つ a2 の宣言を許可しません。 A がリンケージを持っていないので、typedef 名 myA は、リンケージを持っていません。

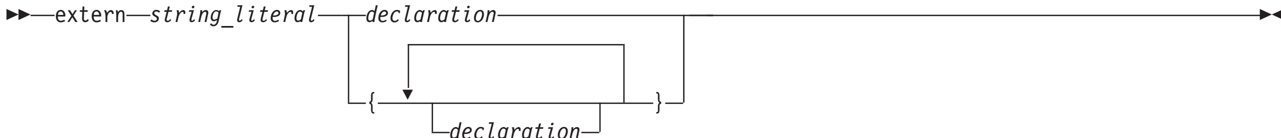
言語リンケージ

400 C IBM i システムでは、#pragma 引数 を使用して C の言語リンケージを使用することができます。詳しくは、「*ILE C/C++ コンパイラー参照*」の『ILE C/C++ のプラグマ』の章および「*ILE C/C++ プログラマーの手引き*」の『ILE 呼び出し規則』の章を参照してください。

C++ C++ コード・フラグメントと C++ 以外のコード・フラグメントの間のリンケージは、*言語リンケージ*と呼ばれます。すべての関数型、関数名、および変数名は、デフォルトで C++ の言語リンケージを持ちます。

リンケージ指定 を使用することによって、C など他のソース言語を使用して作成されたオブジェクト・モジュールに、C++ オブジェクト・モジュールをリンクすることができます。

リンケージ指定の構文



string_literal は、特定の関数に関連したリンケージを指定するために使われます。 リンケージ指定で使われるストリング・リテラルには、大文字小文字の区別があることに注意してください。 すべてのプラットフォームは、*string_literal* に以下の値をサポートしています。

″C++″ 別に指定されていなければ、オブジェクトおよび関数では、これがデフォルトのリンケージ指定です。

″C″ C プロシージャーへのリンケージを示します。

400 ILE C++ がサポートするその他の言語リンケージについては、「 *ILE C/C++ プログラマーの手引き*」の第 25 章『複数言語アプリケーションの処理』を参照してください。

C++ を考慮しないで作成された呼び出し側共用ライブラリーを使用するには、 `#include` ディレクティブを `extern "C" {}` 宣言内に指定する必要があります。

```
extern "C" {
#include "shared.h"
}
```

次の例では、C++ から呼び出される C 印刷関数を示します。

```
//  in C++ program
extern "C" int displayfoo(const char *);
int main() {
    return displayfoo("hello");
}

/*  in C program     */
#include <stdio.h>
extern int displayfoo(const char * str) {
    while (*str) {
       putchar(*str);
       putchar(' ');
       ++str;
    }
    putchar('¥n');
}
```

名前マングリング (C++ のみ)

名前マングリングは、関数名および変数名を固有の名前にエンコードして、リンカーが言語内の共通名を分離できるようにすることです。また、型名もマングルできます。名前マングリングは、多重定義フィーチャーの機能性および異なるスコープ内での可視性を向上させるために、一般的に使用されています。モジュールをコンパイルする場合、コンパイラーは関数引数の型をエンコードして、関数名を生成します。ネーム・スペース内に変数がある場合、同じ変数名が複数のネーム・スペース内に存在できるように、ネーム・スペースの名前はこの変数名にマングルされます。 C++ コンパイラーは、C 変数が存在するネーム・スペースを識別するために、C 変数名もマングルします。

マングル名を作成する方法は、ソース・コードをコンパイルするために使用するオブジェクト・モデルによって異なります。特定のオブジェクト・モデルを使用してコンパイルされたクラスのオブジェクトのマング

ル名は、別のオブジェクト・モデルを使用してコンパイルされた同じクラスのオブジェクトのマングル名とは異なります。オブジェクト・モデルは、コンパイラー・オプションまたはプラグマによって制御されます。

C++ コンパイラーによってコンパイルされたライブラリーやオブジェクト・ファイルを C モジュールにリンクする場合は、名前マングリングを使用すべきではありません。C++ コンパイラーが関数名のマングルを行わないようにするには、次の例のように、宣言またはディレクティブに extern "C" リンケージ指定子を適用します。

```
extern "C" {
   int f1(int);
   int f2(int);
   int f3(int);
};
```

この宣言によって、関数 f1、f2、および f3 への参照をマングルしないようにすることがコンパイラーに通知されます。

extern "C" リンケージ指定子は、C++ 内で定義された関数をマングルしないようにして、C から呼び出せるようにする場合にも使用できます。例えば、次のように指定します。

```
extern "C" {
   void p(int){
      /* not mangled */
   }
};
```

ネストされた extern 宣言の複数レベルでは、最内部の extern 指定が優先されます。

```
extern "C" {
      extern "C++" {
            void func();
      }
}
```

この例では、func は C++ リンケージを持っていません。

第 2 章 字句エレメント

字句エレメント とは、ソース・ファイルで使用できる個別の文字、または文字グループのことです。このセクションでは、基本字句エレメントと C および C++ プログラミング言語の規則について説明します。

- 『トークン』
- 28 ページの『ソース・プログラムの文字セット』
- 32 ページの『コメント』

トークン

プリプロセスおよびコンパイル時に、ソース・コードを*トークン* のシーケンスとして扱います。トークンとは、コンパイラーによって定義されている、プログラム内で意味を成す最小独立単位です。トークンには、次の 4 つの型があります。

- キーワード
- ID
- リテラル
- 区切り子と演算子

隣接する ID、キーワード、およびリテラルは、空白文字で分離する必要があります。他のトークンも、空白文字によって分離し、ソース・コードをより読みやすくする必要があります。空白文字には、ブランク、水平および垂直タブ、改行、改ページおよびコメントがあります。

キーワード

キーワード は、言語の特別な用途のために予約された ID です。キーワードは、プリプロセッサーのマクロ名に使用できますが、プログラミング・スタイルとしては良い方法ではないと見なされます。キーワードの厳密なスペルのみが予約されています。例えば、`auto` は予約されていますが、`AUTO` は予約されていません。

表 3. C および C++ のキーワード

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C++ のみ。

C++ 言語では、次のキーワードも予約されています。

© Copyright IBM Corp. 1998, 2010

表 4. C++ のキーワード

asm	export	private	true
bool	false	protected	try
catch	friend	public	typeid
class	inline	reinterpret_cast	typename
const_cast	mutable	static_cast	using
delete	namespace	template	virtual
dynamic_cast	new	this	wchar_t
明示的	operator	throw	

C++ のみ。 の終り

言語拡張のキーワード

IBM 拡張

ILE C/C++ では、標準言語キーワードのほか、言語拡張での使用および将来の使用のために次のキーワードが予約されています。

表 5. C および C++ 言語拡張のキーワード

__alignof	_Decimal32[1]	__restrict__restrict__	10 進_Decimal
__alignof__	_Decimal64[1]	__signed__	__align_Packed__ptr128__ptr64
__attribute__	_Decimal128[1]	__signed	
__attribute	__extension__	__volatile__	
__const__	__label__	__thread [1]	
	__inline__	typeof	
		__typeof__	

注:

1. これらのキーワードは、**LANGLVL(*EXTENDED)** が指定されているときにのみ認識されます。

C++ のみ。

ILE C++ は、C99 との互換性のための言語拡張機能として以下のキーワードを予約しています。

表 6. C99 に関連する C++ 言語拡張のキーワード

restrict
_Pragma

C++ のみ。 の終り

拡張キーワードが有効であるようなコンパイル・コンテキストの詳細は、本書中の各キーワードを説明しているセクションに記載されています。

関連情報

- 「*ILE C/C++ コンパイラー参照*」にある **LANGLVL(*EXTENDED)**

IBM 拡張 の終り

ID

ID は次の言語エレメントに名前を付けます。

- 関数
- オブジェクト
- ラベル
- 関数仮パラメーター
- マクロおよびマクロ・パラメーター
- 型定義
- 列挙型および列挙子
- 構造体および共用体の名前
- C++ クラスおよびクラス・メンバー
- C++ テンプレート
- C++ テンプレート・パラメーター
- C++ ネーム・スペース

ID は、次の形式で、任意の数の文字、数字、あるいは下線文字により構成されます。

関連情報

- 106 ページの『ID 式 (C++ のみ)』
- 30 ページの『ユニコード規格』
- 13 ページの『キーワード』

ID に使用される文字

ID の先頭文字は英字か _ (下線) でなければなりませんが、ID を下線で始めるのは良いプログラミング・スタイルとはいえません。

コンパイラーでは、ID 内の大文字と小文字が区別されます。例えば、PROFIT と profit は、別の ID を表します。小文字の a を ID 名の一部として指定する場合は、小文字の代わりに大文字の A を使用することはできず、小文字を使用しなければなりません。

C++ では、基本ソース文字セット外の文字および数字の汎用文字名を使用することができます。

400 インプリメンテーションとコンパイラー・オプションによっては、ドル記号 ($) や国別文字セット内の文字などその他の特殊 ID を ID の中で使用できる場合があります。

関連情報

- 「*ILE C/C++ コンパイラー参照*」にある **LANGLVL(*EXTENDED)**

予約済みの ID

2 つの下線文字で始まる ID、または 1 つの下線文字で始まり、その後に 1 つの大文字が続く ID は、コンパイラーが使用するためにグローバルに予約されています。

C 単一の下線で始まる ID は、通常のネーム・スペースでもタグ・ネーム・スペースでもファイル・スコープを持つ ID として予約されています。

C++ 単一の下線で始まる ID は、グローバル・ネーム・スペースで予約されています。

システム呼び出しおよびライブラリー関数の名前は、適切なヘッダーをインクルードしない場合は予約語ではありませんが、これらの名前を ID としては、使用しないようにしてください。事前定義の名前を重複使用すると、コードの保守を行う人が混乱したり、リンク時または実行時のエラーの原因になります。プログラムにライブラリーをインクルードする場合は、名前が重複しないように、そのライブラリーの関数名に注意を払ってください。標準のライブラリー関数を使用するときには常に、適切なヘッダーをインクルードする必要があります。

__func__ 事前定義 ID

C99 の事前定義 ID `__func__` は、関数内で関数名を使用できるようにします。 `__func__` は、各関数定義の左中括弧の直後にコンパイラーによって暗黙的に宣言されます。これによる振る舞いは、次の宣言が行われた場合と同じになります。

```
static const char __func__[] = "function-name";
```

ここで、*function-name* はこの ID を字句として含んでいる関数の名前です。この関数名はマングルされません。

C++ のみ。

関数名は、エンクロージング・クラス名または関数名で修飾されます。例えば、`foo` はクラス `X` のメンバー関数です。`foo` の事前定義 ID は `X::foo` です。`foo` が `main` の本体内で定義されている場合は、`foo` の 事前定義済み ID は `main::X::foo` になります。

テンプレート関数またはメンバー関数の名前は、インスタンスが生成された型を反映します。例えば、`int`, `template<classT> void foo()` でインスタンス化されたテンプレート関数 `foo` の事前定義 ID は `foo<int>` です。

C++ のみ。 の終り

デバッグ目的で、`__func__` ID を明示的に使用して、この ID が含まれている関数の名前を戻すようにすることができます。次に例を示します。

```
#include <stdio.h>

void myfunc(void)    {
         printf("%s\n",__func__);
         printf("size of __func__ = %d\n", sizeof(__func__));
}

int main() {
     myfunc();
}
```

このプログラムの出力は、次のようになります。

```
myfunc
size of __func__=7
```

関連情報

- 169 ページの『関数宣言および関数定義』

リテラル

リテラル定数 または*リテラル* という用語は、プログラム内で発生し、変更できない値を意味します。 C 言語は、名詞*リテラル* の代わりに用語*定数* を使います。形容詞*リテラル* は、定数の概念に、その定数に関しては値だけを扱うという考え方を追加します。リテラル定数はアドレス指定できません。つまり、定数の値はメモリー内のどこかに保管されていますが、そのアドレスにアクセスするための手段がありません。

どの*リテラル* も値とデータ型を持ちます。リテラルの値は、プログラムの実行中に変更されることはなく、その型の表現可能な値の範囲にする必要があります。リテラルの使用可能な型は、次のとおりです。

- 整数リテラル
- ブール・リテラル
- 浮動小数点リテラル
- パック 10 進数リテラル
- 文字リテラル
- ストリング・リテラル

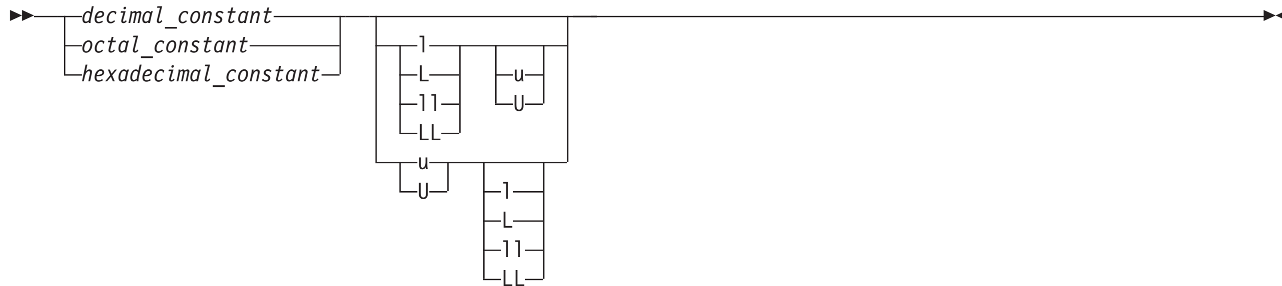
整数リテラル

*整数リテラル*は、小数点または指数部を持たない数値です。このリテラルは以下のものとして表すことができます。

- 10 進整数リテラル
- 16 進整数リテラル
- 8 進整数リテラル

整数リテラルは、基数を指定する接頭部、または型を指定するサフィックスを持つことができます。

整数リテラルの構文



整数リテラルのデータ型は、定数の形式、値、および接尾部により決まります。次の表に、整数リテラルをリストし、可能なデータ型を示します。定数値を表すことができる最小のデータ型が、定数を保管するのに使用されます。

整数リテラル	可能なデータ型
サフィックスがない 10 進数	int、long int、long long int

整数リテラル	可能なデータ型
接尾部なしの 8 進または 16 進数	int、unsigned int、long int、unsigned long int、long long int、unsigned long long int
u または U の接尾部が付く 10 進数、8 進数、または 16 進数	unsigned int、unsigned long int、unsigned long long int
サフィックスとして l または L が付く 10 進数	long int、long long int
サフィックスとして l または L が付く 8 進数または 16 進数	long int、unsigned long int、long long int、unsigned long long int
u または U と l または L の両方の接尾部が付く 10 進数、8 進数、または 16 進数	unsigned long int、unsigned long long int
サフィックスとして ll または LL が付く 10 進数	long long int
octal or hexadecimal suffixed by ll or LL	long long int、unsigned long long int
u または U と ll または LL の両方の接尾部が付く 10 進数、8 進数、または 16 進数	unsigned long long int

関連情報

- 45 ページの『整数型』
- 96 ページの『整数変換』
- 「ILE C/C++ コンパイラー参照」にある **LANGLVL**

10 進整数リテラル: *10 進整数リテラル* には、0 から 9 までの数字が含まれます。最初の数字を 0 にすることはできません。数字 0 で始まる整数リテラルは、10 進整数リテラルではなく、8 進整数リテラルと解釈されます。

10 進整数リテラルの構文

```
              ┌──────────────┐
              ▼              │
►►─digit_1_to_9───digit_0_to_9─┴──────────────────────────────────►◄
```

プラス (+) またはマイナス (-) 記号は、10 進整数リテラルの前に置くことができます。演算子は、リテラルの一部としてではなく、単項演算子として扱われます。

10 進リテラルの例を次に示します。

```
485976
-433132211
+20
            5
```

16 進整数リテラル: *16 進整数リテラル* は、数字 0 とそれに続く x または X で始まり、その後に 0 から 9 までの数字と a から f または A から F までの英字の任意の組み合わせが続きます。英字 A (または a) から F (または f) は、それぞれ値 10 から 15 を表します。

16 進整数リテラル構文

16 進整数リテラルの例を次に示します。

```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```

8 進整数リテラル: *8 進整数リテラル* は、数字 0 で始まり、その後に 0 から 7 までの数字が含まれます。

8 進整数リテラル構文

8 進整数リテラルの例を次に示します。

```
0
0125
034673
03245
```

ブール・リテラル

C++ ブール・リテラルには、`true` と `false` の 2 つしかありません。

関連情報

- 46 ページの『ブール型』
- 96 ページの『ブール変換』

浮動小数点リテラル

*浮動小数点リテラル*は、小数点または指数部を持つ数値です。このリテラルは以下のものとして表すことができます。

- 2 進浮動小数点リテラル
- 16 進浮動小数点リテラル
- 400 21 ページの『10 進浮動小数点リテラル』

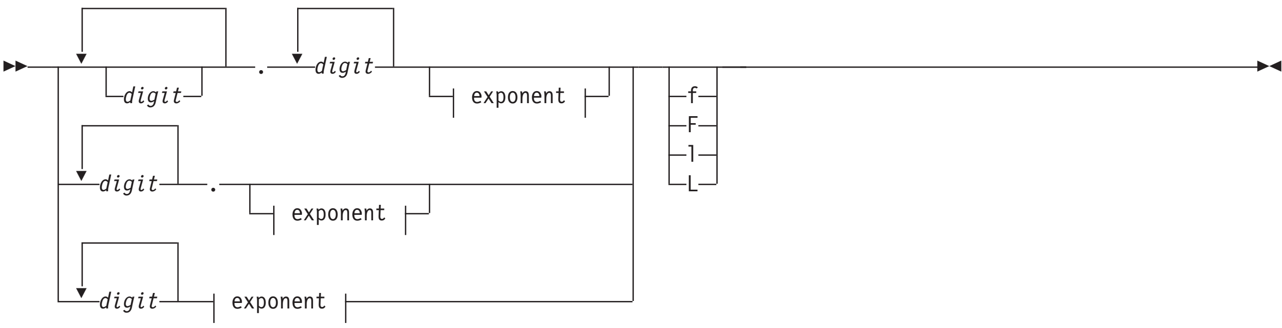
2 進浮動小数点リテラル: 2 進浮動小数点実定数は次のもので構成されます。

- 整数部分
- 小数点
- 小数部分
- 指数部分

- オプションの接尾部

整数部分と小数部分は、両方とも 10 進数で作成されます。整数部分か小数部分のいずれか一方を省略できますが、両方とも省略することはできません。また、小数点または指数部のいずれか一方を省略できますが、両方とも省略することはできません。

2 進浮動小数点リテラル構文



Exponent:

サフィックス f または F は float 型を示し、サフィックス l または L は long double 型を示します。サフィックスを指定しないと、浮動小数点定数には、double 型が指定されます。

プラス (+) またはマイナス (-) シンボルを浮動小数点リテラルの前に置くことができます。ただし、それはリテラルの一部ではありません。それは単項演算子と解釈されます。

浮動小数点リテラルの例を次に示します。

浮動小数点定数	値
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

関連情報

- 47 ページの『浮動小数点型』
- 97 ページの『浮動小数点の型変換』
- 110 ページの『単項式』

16 進浮動小数点リテラル: 16 進浮動小数点実定数 (C99 フィーチャー) は次のもので構成されます。

- 16 進接頭部
- 有効部分
- 2 進指数部
- オプションの接尾部

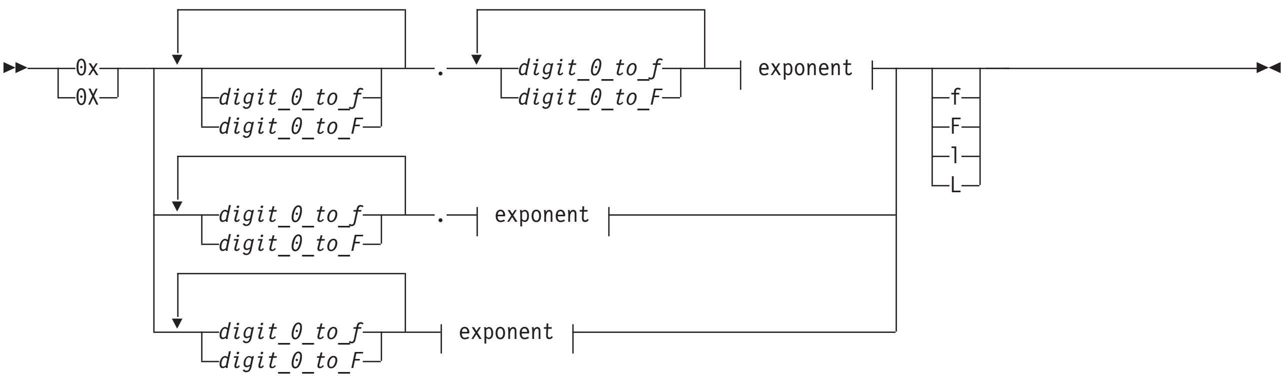
有効部分は有理数を表し、以下で構成されています。

- 16 進数字のシーケンス (整数部)
- オプションの小数部分

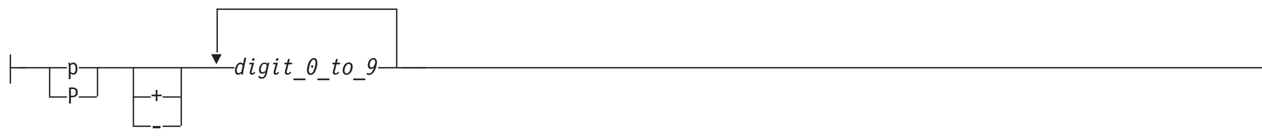
オプションの小数部は、ピリオドとその後の 16 進数字のシーケンスです。

指数部は、有効部分の 2 の累乗を表し、オプションで符号が付いた 10 進整数です。型を表すサフィックスはオプションです。完全な構文は次のとおりです。

16 進浮動小数点リテラル構文



Exponent:



サフィックス f または F は float 型を示し、サフィックス l または L は long double 型を示します。サフィックスを指定しないと、浮動小数点定数には、double 型が指定されます。整数部分か小数部分のいずれか一方を省略できますが、両方とも省略することはできません。 2 進指数部が必要なのは、型を示すサフィックス F があいまいなために 16 進数と間違われるのを避けるためです。

10 進浮動小数点リテラル:

IBM 拡張

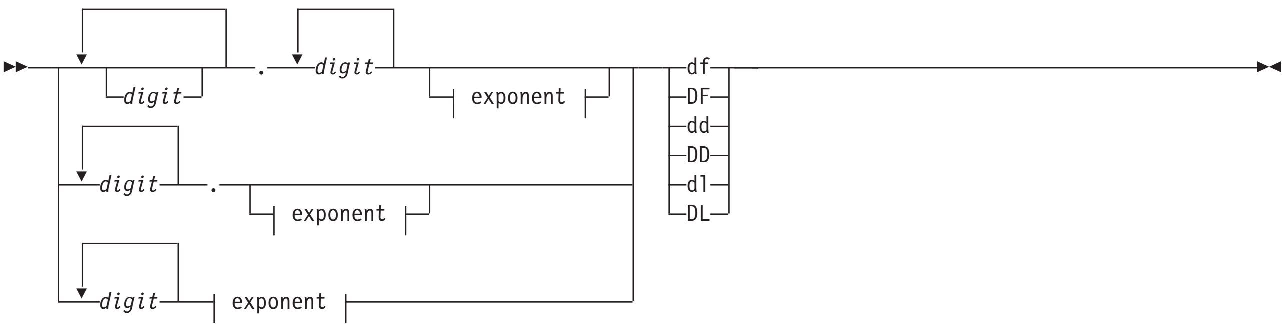
10 進浮動小数点実定数は次のもので構成されます。

- 整数部分
- 小数点
- 小数部分
- 指数部分

• オプションの接尾部

整数部分と小数部分は、両方とも 10 進数で作成されます。整数部分か小数部分のいずれか一方を省略できますが、両方とも省略することはできません。また、小数点または指数部のいずれか一方を省略できますが、両方とも省略することはできません。

10 進浮動小数点リテラル構文



Exponent:

サフィックス df または DF は _Decimal32 型を示し、サフィックス dd または DD は _Decimal64 型を示し、サフィックス dl または DL は _Decimal128 型を示します。サフィックスを指定しないと、浮動小数点定数には、double 型が指定されます。

リテラルの接尾部で大文字と小文字を混用することはできません。

10 進浮動小数点リテラル宣言の例を次に示します。

```
_Decimal32 a = 22.2df;
_Decimal64 b = 33.3dd;
```

10 進浮動小数点データを使用する場合、10 進浮動小数点リテラルを使用するとより正確な結果が得られます。前述のとおり、サフィックスがない浮動小数点定数には、double 型が指定されます。以下のように初期化を行うとします。

```
_Decimal64 rate = 0.1;
```

定数 0.1 は double 型なので、10 進数値 0.1 を正確に表すことはできません。変数の比率は 0.1 と多少異なる値になります。この場合、以下のように定義をコード化する必要があります。

```
_Decimal64 rate = 0.1dd;
```

10 進浮動小数点リテラルが変換されるとき、または定数 10 進浮動小数点式が解決されるときにデフォルトの丸めモードにより、最も近い値に丸められ、等しい値になります。

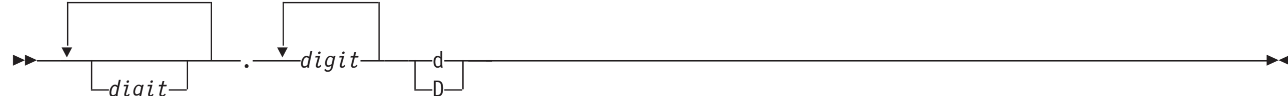
注: 10 進浮動小数点リテラルの接尾部は、**LANGLVL(*EXTENDED)** が指定されているときにのみ認識されます。

IBM 拡張 の終り

パック 10 進数リテラル

400 C *パック 10 進数リテラル* は、大きな数を正確に表すことができる浮動小数点リテラルの一種です。この形式は、整数部分、小数点、小数部分、および必須の接尾部 D で構成されます。パック 10 進数リテラルでは、有効数字を整数部分と小数部分を合わせて 63 桁まで指定できます。

パック 10 進数リテラルの形式は、次のとおりです。



整数部分と小数部分は、両方とも 10 進数で作成されます。整数部分か小数部分のいずれか一方を省略できますが、両方とも省略することはできません。

関連参照
- 詳細については、*ILE C/C++ Programmer's Guide* を参照してください。

文字リテラル

文字リテラル には、一重引用符で囲まれた連続した文字、またはエスケープ・シーケンスが含まれます。例えば、'c'。文字リテラルには、例えば L'c' のように、文字 L を接頭部として付けることができます。 L 接頭部のない文字リテラルは、*通常の文字リテラル*、すなわち*狭幅の文字リテラル* です。 L 接頭部のある文字リテラルは、*ワイド文字リテラル* です。複数の文字またはエスケープ・シーケンス (単一引用符 (')、円記号 (¥) または改行文字を除く) を含んでいる通常の文字リテラルは、*複数文字リテラル* です。

C 狭幅の文字リテラルの型は、int です。ワイド文字リテラルの型は、wchar_t です。複数文字リテラルの型は、int です。

C++ 文字を 1 つだけ含む文字リテラルの型は char で、これは整数型です。ワイド文字リテラルの型は、wchar_t です。複数文字リテラルの型は、int です。

文字リテラル構文

文字リテラルには少なくとも 1 つの文字またはエスケープ・シーケンスがなければならず、かつ文字リテラルは 1 行の論理ソース行に収める必要があります。

ソース・プログラムの文字セットに含まれている任意の文字を使用することができます。二重引用符記号はそれ自体で二重引用符記号を表せますが、一重引用符記号を表すためには、円記号の後に一重引用符記号を

付けたもの (¥' エスケープ・シーケンス) を使用する必要があります。 (エスケープ文字が表すその他の文字のリストについては、 29 ページの『エスケープ・シーケンス』を参照してください。)

400 単一文字を含む、狭幅のまたはワイド文字リテラルの値は、実行時に使われる文字セットの文字の数値表現です。複数の文字を含む整数文字リテラルの値を表す場合、最低 4 バイトが必要です。マルチバイト文字の場合、最低 2 バイトでワイド文字リテラルの値を表します。ロケール型 utf LOCALETYPE(*LOCALEUTF) の場合、マルチバイト文字でワイド文字リテラルの値を表すためには、最低 4 バイトが必要です。

C++ では、基本ソース文字セット外の文字および数字の汎用文字名を使用することができます。

文字リテラルの例を次に示します。

```
'a'
'\''
L'0'
'('
```

関連情報

- 28 ページの『ソース・プログラムの文字セット』
- 30 ページの『ユニコード規格』
- 48 ページの『文字型』

ストリング・リテラル

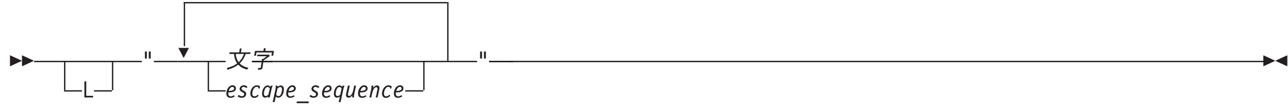
ストリング・リテラル には、二重引用符で囲まれた連続した文字またはエスケープ・シーケンスが含まれます。接頭部 L のあるストリング・リテラルは、*ワイド・ストリング・リテラル* です。接頭部 L のないストリング・リテラルは、*通常の*、すなわち*狭幅ストリング・リテラル* です。

C 狭幅ストリング・リテラルの型は、char の配列です。ワイド・ストリング・リテラルの型は wchar_t の配列です。

C++ 狭幅ストリング・リテラルの型は、const char の配列です。ワイド・ストリング・リテラルの型は const wchar_t の配列です。どちらの型にも静的ストレージ期間があります。

ヌル ('¥0') 文字が、各ストリングに付加されました。ワイド・ストリング・リテラルに対して、型 wchar_t の値 '\0' が付加されます。規則によって、プログラムでは、ヌル文字が検出されると、ストリングの最後と認識されます。

ストリング・リテラル構文



ストリング・リテラル内に含まれている複数のスペースは保存されます。

ストリングの一部として改行文字を表すには、エスケープ・シーケンス ¥n を使用します。ストリングの一部として円記号文字を表すには、エスケープ・シーケンス ¥¥ を使用します。一重引用符記号は、それ自体で、またはエスケープ・シーケンス ¥' を使用して表すことができます。二重引用符を表すには、エスケープ・シーケンス ¥" を使用する必要があります。

C++ では、基本ソース文字セット外の文字および数字の汎用文字名を使用することができます。

ストリング・リテラルの例を、次に示します。

```
char titles[ ] = "Handel's \"Water Music\"";
char *temp_string = "abc" "def" "ghi";  /* *temp_string = "abcdefghi\0" */
wchar_t *wide_string = L"longstring";
```

次の例はストリング・リテラル内のエスケープ・シーケンスを示しています。

```
#include <iostream>
using namespace std;

int main () {
       char *s ="Hi there! \n";
      cout << s;
       char *p = "The backslash character \\.";
       cout << p << endl;
       char *q = "The double quotation mark \".\n";
       cout << q ;
 }
```

このプログラムの出力は次のようになります。

```
Hi there!
The backslash character ¥.
The double quotation mark ".
```

次の行にストリングを継続するには、行継続文字 (¥ 記号) と、その後に続くオプションの空白文字と改行文字 (必須) を使用します。次に例を示します。

```
char *mail_addr = "Last Name    First Name    MI   Street Address \
       893    City     Province   Postal code ";
```

次の例では、ストリング・リテラル second が、コンパイル時のエラーを起こします。

```
char *first = "This string continues onto the next\
      line, where it ends.";                    /* compiles successfully.   */

char *second = "The comment makes the \         /* continuation symbol
      */   invisible to the compiler.";         /* compilation error.       */
```

注: プログラム・ソースにストリング・リテラルが複数回現れるときは、ストリングが読み取り専用または書き込み可能であるかによって、保管される方法が異なります。ILE C/C++ は、読み取り専用ストリングに対して 1 つの場所だけを割り振る場合があります。すべてのオカレンスは、その場所だけを参照することになります。ただし、そのストレージ域は、書き込み保護であると考えられます。ストリングが書き込み可能の場合は、ストリングの各オカレンスには、それぞれ別個の、常に変更可能なストレージ場所が割り当てられます。デフォルトでは、コンパイラーは、ストリングを書き込み可能であると見なします。**#pragma strings** ディレクティブまたは **PFROPTC** コンパイラー・オプションを使用し、ストリング・リテラルのデフォルトのストレージを変更できます。

関連情報

- 48 ページの『文字型』
- 28 ページの『ソース・プログラムの文字セット』
- 30 ページの『ユニコード規格』
- 「*ILE C/C++ コンパイラー参照*」にある **PFROPTC** コンパイラー・オプション
- 「*ILE C/C++ コンパイラー参照*」にある **#pragma strings**

ストリング連結: ストリングを継続する別の方法は、複数の連続ストリングを持つことです。隣接するストリング・リテラルを連結し、単一のストリングを作成します。次に例を示します。

```
"hello " "there"     /* is equivalent to "hello there"  */
"hello" "there"      /* is equivalent to "hellothere"                     */
```

連結されたストリングの文字は、別個のまま残っています。例えば、ストリング "\xab" と "3" は、連結されて "\xab3" を形成します。しかし、文字 \xab および 3 は、別個のまま残っていて、16 進文字 \xab3 形成するためにマージされることはありません。

ワイド・ストリング・リテラルと狭幅ストリング・リテラルとの連結は ILE C/C++ ではサポートされていません。次に示すように、ワイド・ストリング・リテラルと狭幅ストリング・リテラルが隣接している場合は、

```
"hello " L"there"
```

コンパイラーからエラー・メッセージが出されます。

連結の後で、各ストリングの終わりに、char 型の '\0' が付加されます。ワイド・ストリング・リテラルに対して、型 wchar_t の値 '¥0' が付加されます。 C++ プログラムは、この値をスキャンすることによって、ストリングの終わりを検出します。 次に例を示します。

```
char *first = "Hello ";            /* stored as "Hello \0"       */
char *second = "there";            /* stored as "there\0"        */
char *third = "Hello " "there";    /* stored as "Hello there¥0"  */
```

区切り子と演算子

区切り子 は、コンパイラーにとって構文上およびセマンティック上の意味を持つトークンですが、厳密な意味はコンテキストにより異なります。区切り子は、プリプロセッサーの構文の中でもトークンとして使用できます。

C99 と C++ では、以下のトークンが区切り子、演算子、またはプリプロセス・トークンとして定義されています。

表 7. C および C++ の区切り子

[ ]	( )	{ }	,	:	;
*	=	...	#		
.	->	++	--	##	
&	+	-	~	!	
/	%	<<	>>	!=	
<	>	<=	>=	==	
^	|	&&	||	?	
*=	/=	%=	+=	-=	
<<=	>>=	&=	^=	|=	

C++ のみ。

C++ では、C99 のプリプロセス・トークン、演算子、および区切り子に加えて、以下のトークンを区切り子として使用できます。

表 8. C++ の区切り子

::	.*	->*	new	delete	
and	and_eq	bitand	bitor	comp	
not	not_eq	or	or_eq	xor	xor_eq

C++ のみ。 の終り

関連情報

- 103 ページの『第 6 章 式と演算子』

代替トークン

C および C++ は両方とも、いくつかの演算子および区切り子に次の代替表記を提供します。代替表記は *2 文字表記文字*とも呼ばれます。

演算子または区切り子	代替表記
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

C++ と C (C99 言語レベル) では、上にリストされている演算子と区切り子のほかに、以下の代替表記も提供します。 C では、これらの代替表記はヘッダー・ファイル `iso646.h` の中でマクロとして定義されています。

演算子または区切り子	代替表記
&&	および
|	bitor
||	または
^	xor
~	compl
&	bitand
&=	and_eq
|=	or_eq
^=	xor_eq
!	否定
!=	not_eq

関連情報

- 31 ページの『2 文字表記文字』

ソース・プログラムの文字セット

コンパイル時と実行時の両方で使用できる基本ソース文字セットを次に示します。

- アルファベットの大文字および小文字

  a b c d e f g h i j k l m n o p q r s t u v w x y z

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- 10 進数字

  0 1 2 3 4 5 6 7 8 9
- 以下の図形文字

  ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] _ { } ~
  - ASCII の脱字記号 (^) 文字 (ビット単位の排他 OR 記号) または EBCDIC の等価否定 (¬) 文字
  - ASCII の分割垂直バー (¦) 文字または EBCDIC の等価垂直バー (|) 文字。
- 空白文字
- 改行、水平タブ、垂直タブ、改ページ、ストリングの終了 (NULL 文字)、アラート、バックスペース、および復帰を表す制御文字。

400 インプリメンテーションとコンパイラー・オプションによっては、ドル記号 ($) や国別文字セット内の文字などその他の特殊 ID を ID の中で使用できる場合があります。

関連情報

- 15 ページの『ID に使用される文字』

マルチバイト文字

コンパイラーは、ストリング・リテラルと文字定数に十分使用できる追加文字 (拡張文字セット) を認識し、サポートします。拡張文字のサポートには、*マルチバイト文字* セットが含まれます。*マルチバイト文字* とは、ビット表現が複数バイトに収まる文字のことです。

マルチバイト文字は次のいずれかのコンテキストで使用することができます。

- ストリング・リテラルと文字定数。マルチバイト・リテラルを宣言するには、通常の文字表現を使用します。たとえば、次のとおりです。

  ```
  char *a = "multibyte string";
  char b = "multibyte-char";
  ```

  マルチバイト文字を含むストリングは、本質的にはマルチバイト文字を含まないストリングと同様に扱われます。一般に、ワイド文字はマルチバイト文字が存在する所ならどこでも使用できますが、ワイド文字とマルチバイト文字とではビット・パターンが異なるため、これらの文字を同じストリング内で混用することはできません。許可されている所であれば、1 バイト文字とマルチバイト文字を同じストリング内で混用することができます。
- プリプロセッサー・ディレクティブ。以下のプリプロセッサー・ディレクティブでは、マルチバイト文字定数とマルチバイト文字ストリング・リテラルを使用することができます。
  - #define
  - #pragma comment
  - #include

#include ディレクティブに指定されたファイル名は、マルチバイト文字を含むことができます。次に例を示します。

```
#include <multibyte_char/mydir/mysource/multibyte_char.h>
#include "multibyte_char.h"
```

- マクロ定義。ストリング・リテラルと文字定数は #define ステートメントの一部とすることができるので、オブジェクト類似マクロ定義と関数類似マクロ定義の両方でもマルチバイト文字を使用することができます。
- # 演算子と ## 演算子
- プログラムのコメント

マルチバイト文字の使用に関する制約事項を次に示します。

- ID 内でマルチバイト文字を使用することはできません。
- マルチバイト文字の 16 進値は、使用しているコード・ページの範囲内になければなりません。
- ワイド文字とマルチバイト文字をマクロ定義内で混用することはできません。例えば、ワイド・ストリングとマルチバイト・ストリングを連結するマクロ展開を使用することはできません。
- ワイド文字とマルチバイト文字との間の割り当ては許可されません。
- ワイド文字ストリングとマルチバイト文字ストリングとの連結は許可されません。

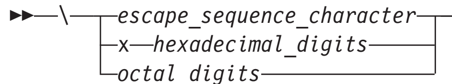
関連情報

- 23 ページの『文字リテラル』
- 30 ページの『ユニコード規格』
- 48 ページの『文字型』
- 「*ILE C/C++ コンパイラー参照*」にある **-qmbcs**

エスケープ・シーケンス

エスケープ・シーケンス によって、実行文字セットの任意のメンバーを表すことができます。エスケープ・シーケンスは、基本的に、印刷不能文字を文字リテラルまたはストリング・リテラルに入れるために使用されます。例えば、エスケープ・シーケンスを使用して、タブ、復帰、およびバックスペースなどの文字を出力ストリームに入れることができます。

エスケープ文字の構文



エスケープ・シーケンスでは、円記号 (¥) の後に、エスケープ・シーケンス文字または 8 進数か 16 進数が続きます。16 進数のエスケープ・シーケンスでは、x の後に、1 つまたは複数の 16 進数字 (0-9、A-F、a-f) が続きます。8 進数のエスケープ・シーケンスでは、8 進数字 (0-7) を最大 3 個使用します。16 進数または 8 進数の値は、必要な文字またはワイド文字の値を指定します。

注: 行継続シーケンス (¥ の後に改行文字が続く) は、エスケープ・シーケンスではありません。行継続シーケンスは、文字ストリングで使われ、ソース・コードの現在行が次の行に継続することを示します。

エスケープ・シーケンスと表される文字は、次のとおりです。

エスケープ・シーケンス	表される文字
\a	アラート (ベル、アラーム)
\b	バックスペース
\f	改ページ (新しいページ)
\n	改行
\r	復帰
\t	水平タブ
\v	垂直タブ
\'	一重引用符
\"	二重引用符
\?	疑問符
\\	円記号

エスケープ・シーケンスの値は、実行時に使われる文字セットのメンバーを表します。プリプロセスの間に、エスケープ・シーケンスを変換します。例えば、ASCII 文字コードを使用するシステムでは、エスケープ・シーケンス \x56 の値は英字 V です。EBCDIC 文字コードを使用するシステムでは、エスケープ・シーケンス \xE5 の値は英字 V です。

エスケープ・シーケンスは、文字定数またはストリング・リテラルでのみ使用してください。エスケープ・シーケンスが認識されない場合は、エラー・メッセージが出されます。

ストリングまたは文字シーケンスでは、円記号で円記号自体を表すとき (エスケープ・シーケンスの始まりではなく) は、¥¥ 円記号エスケープ・シーケンスを使用する必要があります。次に例を示します。

```
cout << "The escape sequence \\n." << endl;
```

このステートメントでは、次のように出力されます。

```
The escape sequence ¥n.
```

ユニコード規格

ユニコード規格 は、書かれた文字とテキストに対するコード化スキームの仕様です。ユニコード規格は、マルチリンガル・テキストのエンコードに整合性を持たせ、矛盾を起こさずにテキスト・データを国際的に交換できるようにした汎用の規格です。 C および C++ の ISO 規格は、*Information technology - Programming Languages - Universal Multiple-Octet Coded Character Set (UCS), ISO/IEC 10646:2003* です。(*octet* という用語は、ISO ではバイトを指す用語として使用されます。) ISO/IEC 10646 規格は、エンコードのフォーム数がユニコード規格より制限されています。すなわち、ISO/IEC 10646 に準ずる文字セットはユニコード規格も満たします。

ユニコード規格は、各文字に固有の数値と名前を指定しており、数値のビット表現に 3 つのエンコード方式を定義しています。名前と値のペアで文字の識別を行います。文字を表す 16 進値はコード・ポイントと呼ばれます。仕様には、全体の文字特性、すなわち、各文字の大/小文字、方向性、英字特性、その他のセマンティクス情報も記述されています。 ASCII に基づくと、ユニコード規格は英字、表意文字、およびシンボルを扱い、予約済みコード・ポイント範囲でインプリメンテーション別の文字コードを定義することができます。したがって、ユニコード規格のエンコード・スキームは、ユニコード規格のとおり、世界中のあらゆる歴史的文字の波及範囲を含めて、既知のすべての文字エンコード要件に対応できるだけの十分な柔軟性を備えています。

C99 および C++ では、ISO/IEC 10646 で定義されている汎用文字名構成を使用して、基本のソース文字セット外の文字を表すことができます。どちらの言語でも、ID、文字定数、およびストリング・リテラルの中で汎用文字名を使用することができます。

以下の表に、一般的な汎用文字名の構成と ISO/IEC 10646 のショート・ネームの対応を示します。

汎用文字名	ISO/IEC 10646 の短縮名
ここでは、N は 16 進数字です。	
\UNNNNNNNN	NNNNNNNN
\uNNNN	0000NNNN

C99 および C++ は、基本文字セット (基本ソース・コード・セット) 内の文字を表す 16 進値、および ISO/IEC 10646 で予約されているコード・ポイントを制御文字に使用することが禁じられています。

また以下の文字も使用できません。

- 00A0 より短い ID を持つ文字。例外は、0024 ($)、0040 (@)、または 0060 (') です。
- D800 ～ DFFF の範囲内 (両端を含む) のコード・ポイントにある短い ID を持つ文字。

2 文字表記文字

*2 重字表記文字*と呼ばれる 2 つのキー・ストロークの組み合わせを使用して、ソース・プログラムでは使用できない文字を表現できます。プリプロセッサーのフェーズでは、2 文字表記文字はトークンとして読み取られます。

2 文字表記文字は、次の通りです。

%: または %%	#	番号記号
<:	[	左大括弧
:>	]	右大括弧
<%	{	左中括弧
%>	}	右中括弧
%:%: または %%%%	##	プリプロセッサー・マクロ連結演算子

2 文字表記文字は、マクロの連結を使用して作成できます。ILE C/C++ では、ストリング・リテラルや文字リテラルに含まれる 2 文字表記文字は置換されません。次に例を示します。

```
char *s = "<%%>"; // stays "<%%>"

switch (c)  {
  case '<%' : { /* ... */ } // stays '<%'
  case '%>' : { /* ... */ } // stays '%>'
 }
```

関連情報

- 「*ILE C/C++ コンパイラー参照*」にある **__DIGRAPHS__**
- 「*ILE C/C++ コンパイラー参照*」にある **OPTION(*DIGRAPH)**

3 文字表記

C および C++ の文字セットの文字には、環境によっては使用可能でないものがあります。*3 文字表記* と呼ばれる 3 文字のシーケンスを使用して、これらの文字を C または C++ ソース・プログラムに入力できます。 3 文字表記は、次のとおりです。

3 文字表記	単一文字	説明
??=	#	ポンド記号
??(	[	左大括弧
??)	]	右大括弧
??<	{	左中括弧
??>	}	右中括弧
??/	\	円記号
??’	^	脱字記号
??!	|	垂直バー
??-	~	波形記号 (tilde)

プリプロセッサーが、3 文字表記を対応する単一文字表示に置換します。例えば、次のような場合です。

```
some_array??(i??) = n;
```

これは、次のものを表します。

```
some_array[i] = n;
```

コメント

コメント は、プリプロセスの間に、単一スペース文字に置き換えられるテキストです。したがって、コンパイラーはすべてのコメントを無視することになります。

2 種類のコメントがあります。

- /* (スラッシュ、アスタリスク) 文字があって、その後に文字の任意のシーケンス (改行を含む) が続き、さらにその後に */ 文字が続きます。この種類のコメントは、通常 *C スタイルのコメント* と呼ばれています。
- // (2 つのスラッシュ) 文字があり、その後に任意の文字のシーケンスが続きます。直前に円記号がない状態で改行すれば、この形式のコメントは終了します。 この種類のコメントは、通常、*単一行コメント* または *C++ コメント* と呼ばれています。 C++ コメントは、行結合 (¥) 文字を使用して 1 行の論理ソース行に結合することで、複数行の物理ソース行にまたがることができます。円記号文字は、3 文字表記で表すこともできます。

コメントは、言語が空白文字を許可する場所であればどこにでも記述できます。 C スタイルのコメントは、他の C スタイルのコメント内でネストできません。*/ が最初に現れた位置で、各コメントは終了します。

コメントにマルチバイト文字を含めることもできます。

注: 文字定数またはストリング・リテラルで検出される /* または */ 文字は、コメントの始まりまたは終わりを表すものではありません。

次のプログラムでは、2 番目の printf() がコメントです。

```
#include <stdio.h>

int main(void)
{
   printf("This program has a comment.¥n");
   /* printf("This is a comment line and will not print.¥n"); */
      return 0;
}
```

2 番目の printf() は、スペースと同等であるため、このプログラムの出力は次のようになります。

```
This program has a comment.
```

次のプログラムの printf() は、コメント区切り文字 (/*、*/) が、ストリング・リテラルの内側にあるため、コメントではありません。

```
#include <stdio.h>

    int main(void)
    {
       printf("This program does not have ¥
    /* NOT A COMMENT */ a comment.¥n");
       return 0;
    }
```

このプログラムの出力は、次のようになります。

```
This program does not have /* NOT A COMMENT */ a comment.
```

次の例では、コメントは強調表示されています。

```
/* A program with nested comments. */

    #include <stdio.h>

    int main(void)
    {
       test_function();
       return 0;
    }

    int test_function(void)
    {
        int number;
        char letter;
    /*
    number = 55;
    letter = 'A';
    /* number = 44; */
    */
    return 999;
    }
```

test_function では、コンパイラーは、最初の /* から最初の */ までをコメントとして読み取ります。 2 番目の */ は、エラーです。ソース・コードですでにコメントにされている個所をコメントにしないようにするには、条件付きコンパイルのプリプロセッサー・ディレクティブを使用して、コンパイラーにプログラムのセクションを迂回させる必要があります。例えば、上記のステートメントをコメントにする代わりに、ソース・コードを次のように変更します。

```
/* A program with conditional compilation to avoid nested comments. */

#define TEST_FUNCTION 0
#include <stdio.h>

int main(void)
{
   test_function();
   return 0;
}

int test_function(void)
{
    int number;
    char letter;
```

```
  #if TEST_FUNCTION
    number = 55;
    letter = 'A';
    /*number = 44;*/
  #endif  /*TEST_FUNCTION */
 }
```

単一行コメントは、C スタイルのコメント内でネストできます。例えば、次のプログラムは何も出力しません。

```
#include <stdio.h>

   int main(void)
   {
      /*
      printf("This line will not print.¥n");
      // This is a single line comment
      // This is another single line comment
      printf("This line will also not print.¥n");
      */
      return 0;
   }
```

注: #pragma comment ディレクティブを使用してコメントをオブジェクト・モジュールに入れることもできます。

関連情報

- 「*ILE C/C++ コンパイラー参照*」にある **#pragma comment**
- 28 ページの『マルチバイト文字』

第 3 章 データ・オブジェクトおよび宣言

このセクションでは、データ・オブジェクトの宣言を構成するさまざまな要素について説明し、以下のトピックが含まれています。

- 『データ・オブジェクトおよびデータ宣言の概要』
- 39 ページの『ストレージ・クラス指定子』
- 45 ページの『型指定子』
- 49 ページの『ユーザー定義型』
- 62 ページの『型修飾子』
- 67 ページの『型属性』

トピックは、宣言内で要素が使用される順序に大まかに従うように並んでいます。また、データ宣言の追加要素の説明は、 71 ページの『第 4 章 宣言子』で続けられます。

データ・オブジェクトおよびデータ宣言の概要

以下のセクションでは、本書全体で使用するデータ・オブジェクトおよびデータ宣言に関する基本的な概念を紹介します。

データ・オブジェクトの概念

データ・オブジェクト は、値または値のグループを含むストレージの領域です。それぞれの値には、その ID を使用して、またはそのオブジェクトを参照する複雑な式を使用してアクセスできます。さらに、各オブジェクトには、固有の*データ型* が指定されています。オブジェクトのデータ型によって、そのオブジェクトのストレージ割り当てと、以降のアクセスでの値の解釈が決まります。このデータ型は、型チェック演算でも使われます。オブジェクトの ID とデータ型は、両方とも、オブジェクトの*宣言* で確立されます。

C++ クラス型のインスタンスは、一般に*クラス・オブジェクト* と呼ばれます。個別クラスのメンバーもまた、オブジェクトと呼ばれます。すべてのメンバー・オブジェクトのセットが、1 つのクラス・オブジェクトを構成します。

データ型は、以下のような重複する型カテゴリーによくグループ化されます。

基本型対派生型

基本 データ型は、言語の「基礎」、「基本」、「組み込み」のデータ型とも呼ばれます。このようなデータ型には、整数、浮動小数点数、および文字が含まれます。*派生* 型 (標準 C++ では「複合」型とも呼ばれる) は、基本型のセットから作成され、配列、ポインター、構造体、共用体、および列挙型が含まれます。 C++ のクラスはすべて複合型と見なされます。

組み込み型対ユーザー定義型

組み込み データ型には、すべての基本型、および基本型のアドレスを参照する型 (配列やポインターなど) が含まれます。*ユーザー定義* 型は、基本型のセットから、`typedef`、構造体、共用体、および列挙型の定義でユーザーが作成したものです。 C++ のクラスはユーザー定義の型と見なされます。

スカラー型対集合体型

スカラー 型は単一のデータ値を表し、*集合体* 型は、同じ型または異なる型から成る複数の値を表

© Copyright IBM Corp. 1998, 2010

します。スカラーには、算術型およびポインターが含まれます。集合体型には、配列および構造体が含まれます。 C++ のクラスは集合体型と見なされます。

以下のマトリックスでは、サポートされるデータ型、および基本、派生、スカラー、および集合体の各型への分類をリストします。

表 9. C/C++ データ型

データ・オブジェクト	基本	複合	組み込み	ユーザー定義済み	スカラー	集合体
整数型	+		+		+	
浮動小数点型	+		+		+	
文字型			+		+	
ブール型	+		+		+	
void 型	+[1]		+		+	
ポインター		+	+		+	
配列		+	+			+
構造体		+		+		+
共用体		+		+		
列挙型		+		+	注[2] を参照	
C++ クラス		+		+		+
typedef 型		+		+		

注:

1. void 型は、『不完全型』で説明するように、実際には不完全な型です。それにもかかわらず、標準 C++ では、この型を基本型として定義しています。
2. C C 標準は、列挙型をスカラーとしても集合体としても分類していません。 C++ 標準 C++ は、列挙型をスカラーとして分類しています。

関連情報

- 219 ページの『第 11 章 クラス (C++ のみ)』

不完全型

以下は、不完全型です。

- void 型
- 不明サイズの配列
- 不完全型エレメントの配列
- 定義のない構造体、共用体、または列挙型
- C++ 宣言されているが、定義されていないクラス型に対するポインター
- C++ 宣言されているが定義されていないクラス

以下の例は、不完全型を示しています。

```
void *incomplete_ptr;
struct dimension linear; /* no previous definition of dimension */
```

関連情報

- 48 ページの『void 型』
- 224 ページの『不完全なクラス宣言』

互換型および複合型

C のみ

C では、互換型は以下のように定義されます。

- (代入式などで) 変更せずにともに使用できる 2 つの型
- 変更せずに一方を他方と置換できる 2 つの型

2 つの互換型を結合すると、その結果は*複合型* です。合成された 2 つの互換型から生まれた結果の複合型を判別する方法は、整数型が算術演算と結合するときに使われる整数型の通常のバイナリー変換と似ています。

明らかに、同一の 2 つの型は互換性があり、それらの複合型は同じ型です。同一でない型、ユーザー定義の型、型修飾型などの型互換性を決定する法則は、あまり明らかではありません。 45 ページの『型指定子』で、C における基本型およびユーザー定義型の互換性について説明します。

C のみ の終り

C++ のみ。

同じ型であること以外に型互換性の別の考え方は C++ にはありません。一般的に、C++ における型の検査は C よりも厳しく、C で互換型のみを必須条件とするような場合でも、同一型が必須と見なされます。

C++ のみ。 の終り

関連情報

- 80 ページの『配列の互換性 』
- 77 ページの『ポインターの互換性 (C のみ)』
- 172 ページの『互換関数 (C のみ)』

データ宣言およびデータ定義の概要

宣言 では、プログラムで使われるデータ・オブジェクトの名前および属性が確立されます。*定義* は、データ・オブジェクトにストレージを割り振り、ID をオブジェクトに関連付けます。*型*を宣言または定義するときは、ストレージは割り振られません。

次の表は、宣言と定義の例を示しています。最初の列に宣言されている ID は、ストレージを割り当てません。これらの ID は、対応する定義を参照します。2 番目の列に宣言されている ID は、ストレージを割り当てます。これらの ID は、宣言と定義の両方になります。

宣言	宣言と定義
`extern double pi;`	`double pi = 3.14159265;`

`struct payroll;`	`struct payroll {`
	`char *name;`
	`float salary;`
	`} employee;`

宣言は、データ・オブジェクトおよびそれらの ID の以下の属性を決定します。

- スコープ。これは、オブジェクトにアクセスするために ID を使用できるプログラム・テキスト領域を記述します。
- 可視性。これは、ID のオブジェクトへ正しくアクセスできるプログラム・テキスト領域を記述します。
- 期間。これは、ID がメモリー内に割り振られた実際の物理オブジェクトを保持する期間を定義します。
- リンケージ。これは、ある特定のオブジェクトへの ID の正確な関連付けを記述します。
- タイプ。オブジェクトにどのくらいのメモリーが割り振られるか、そのオブジェクトのストレージ割り当てで検出されたビット・パターンをプログラムがどのように解釈すべきかを決定します。

データ・オブジェクトを宣言する際のエレメントは、次のとおりです。

- ストレージ・クラス指定子。ストレージ期間およびリンケージを指定します
- 型指定子。データ型を指定します
- 型修飾子。データ値の可変性を指定します
- 71 ページの『宣言子の概要』。ID を導入して組み込みます
- 初期化指定子。ストレージを初期値で初期化します

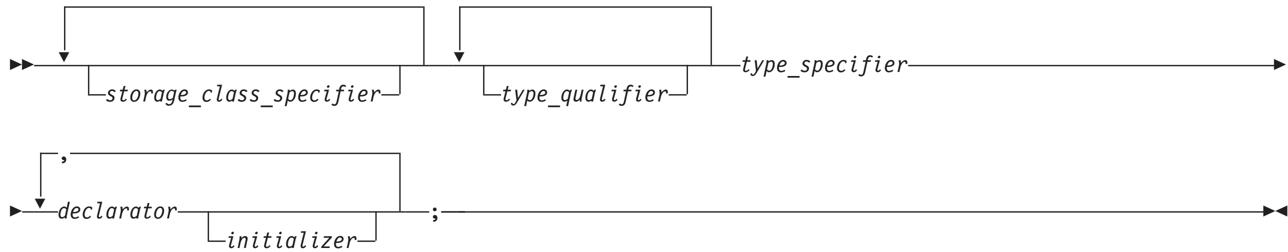
IBM 拡張

また、ILE C/C++ では、*属性* を使用して、データ・オブジェクトのプロパティーを変更できます。ユーザー定義型の定義を変更するために使用できる*型* 属性については、 67 ページの『型属性』で説明します。変数の宣言を変更するために使用できる変数属性については、 89 ページの『変数属性』で説明します。

IBM 拡張 の終り

すべての宣言の形式は、次のとおりです。

データ宣言の構文



関連情報

- 169 ページの『関数宣言および関数定義』

仮定義

C のみ

暫定定義 とは、ストレージ・クラス指定子と初期化指定子を持たない外部データ宣言です。変換単位の終りに達しても、その ID に対して初期化指定子が指定された定義が現れなかった場合、暫定定義は完全定義になります。この場合、コンパイラーは、定義済みオブジェクトに未初期化スペースを予約します。

以下のステートメントは、通常の定義と暫定定義を示しています。

```
int i1 = 10;         /* definition, external linkage */
static int i2 = 20;  /* definition, internal linkage */
extern int i3 = 30;  /* definition, external linkage */
int i4;              /* tentative definition, external linkage */
static int i5;       /* tentative definition, internal linkage */

int i1;              /* valid tentative definition */
int i2;              /* not legal, linkage disagreement with previous */
int i3;              /* valid tentative definition */
int i4;              /* valid tentative definition */
int i5;              /* not legal, linkage disagreement with previous */
```

C のみ の終り

C++ のみ。

C++ は、暫定定義の概念をサポートしていません。ストレージ・クラス指定子のない外部データ宣言は常に定義です。

C++ のみ。 の終り

ストレージ・クラス指定子

ストレージ・クラス指定子は、変数、関数、およびパラメーターの宣言を詳細化するために使用します。ストレージ・クラスは、以下を決定します。

- オブジェクトに、内部リンケージまたは外部リンケージがあるか、またはリンケージがないか
- オブジェクトは、メモリーまたはレジスター (使用可能な場合) のどちらに格納されるか
- オブジェクトが、デフォルトの初期値 0 または不確定デフォルトの初期値のどちらを受け取るか
- オブジェクトが、プログラム全体で参照されるか、または変数を定義した関数、ブロック、ソース・ファイル内でのみ参照されるか
- オブジェクトのストレージ期間が、プログラム実行時全体で維持されるか、またはオブジェクトが定義されたブロックの実行中しか維持されないか

変数の場合は、そのデフォルトのストレージ期間、スコープ、およびリンケージは、それがどこで宣言されたかによって異なります。つまり、ブロック・ステートメントまたは関数本体の内側なのか、外側なのかによります。これらのデフォルトが満足のいくものではない場合は、ストレージ・クラス指定子を使用して、ストレージ・クラスを明示的に設定できます。 C および C++ のストレージ・クラス指定子は、次のとおりです。

- auto
- static
- extern
- C++ mutable
- register
- 400 __thread

関連情報

- 174 ページの『関数ストレージ・クラス指定子』

- 81 ページの『初期化指定子』

自動ストレージ・クラス指定子

auto ストレージ・クラス指定子により、*自動ストレージ* を取る変数を明示的に定義できます。 auto ストレージ・クラスは、ブロック内部で宣言される変数のデフォルトです。自動ストレージを持つ変数 x は、x が宣言されたブロックが終了するときに削除されます。

auto ストレージ・クラス指定子は、ブロックで宣言された変数の名前または関数仮パラメーターの名前にだけ適用できます。ただし、これらの名前には、デフォルトで自動ストレージがあります。したがって、ストレージ・クラス指定子 auto は、データ宣言では通常冗長です。

自動変数のストレージ期間

auto ストレージ・クラス指定子が指定されたオブジェクトには、自動ストレージ期間が指定されます。ブロックに入るたびに、そのブロックに定義された auto オブジェクトに対するストレージが使用可能になります。ブロックが終了すると、そのオブジェクトは使用できなくなります。リンケージ指定がなく、static ストレージ・クラス指定子を使用しないで宣言されたオブジェクトは、自動ストレージ期間を持ちます。

再帰的に呼び出す関数内で auto オブジェクトを定義すると、ブロックの各呼び出しごとに、メモリーがオブジェクトに割り当てられます。

自動変数のリンケージ

auto 変数には、ブロック・スコープがありますが、リンケージはありません。

関連情報

- 82 ページの『初期化およびストレージ・クラス』
- 153 ページの『ブロック・ステートメント』
- 167 ページの『goto ステートメント』

static ストレージ・クラス指定子

static ストレージ・クラス指定子を使用して宣言されたオブジェクトは、*静的ストレージ期間* を持ちます。これは、これらのオブジェクトのメモリーがプログラムの実行開始時に割り振られ、プログラムの終了時に解放されることを意味します。変数の静的ストレージ期間は、ファイル・スコープまたはグローバル・スコープと異なります。変数は静的期間を持つことができますが、ローカル・スコープは持てません。

C キーワード static は、情報の隠蔽を強制するための C の主要な手段です。

C++ C++ は、情報の隠蔽をネーム・スペース言語フィーチャーとクラスのアクセス制御によって強制します。外部変数のスコープを制限するためのキーワード static の使用は、ネーム・スペース・スコープ内のオブジェクトの宣言には推奨できません。

static ストレージ・クラス指定子は、以下の宣言に適用できます。

- データ・オブジェクト
- C++ クラス・メンバー
- 無名共用体

static ストレージ・クラス指定子は、以下のものでは使用できません。

- 型宣言

- 関数仮パラメーター

関連情報

- 174 ページの『static ストレージ・クラス指定子』
- 239 ページの『静的メンバー』

静的変数のリンケージ

static ストレージ・クラス指定子を含み、ファイル・スコープを持つオブジェクトの宣言は、ID の内部結合を示します。したがって、特定の ID のそれぞれのインスタンスは、1 つのファイル内のみの同じオブジェクトを表します。例えば、以下のように、静的変数 x が関数 f で宣言されている場合に、プログラムがスコープ f から出ても、 x は破棄されません。

```
#include <stdio.h>

int f(void) {
  static int x = 0;
  x++;
  return x;
}

int main(void) {
  int j;
  for (j = 0; j < 5; j++) {
    printf("Value of f(): %d¥n", f());
  }
  return 0;
}
```

上記の例の出力は、以下のとおりです。

```
Value of f(): 1
Value of f(): 2
Value of f(): 3
Value of f(): 4
Value of f(): 5
```

x は静的変数であるので、f への継続的な呼び出しで 0 に再初期化されることはありません。

関連情報

- 82 ページの『初期化およびストレージ・クラス』
- 8 ページの『内部結合』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

extern ストレージ・クラス指定子

extern ストレージ・クラス指定子により、複数のソース・ファイルから使用できるオブジェクトを宣言できます。 extern 宣言は、現行ソース・ファイルの以降の部分によって、記述された変数を使用可能にします。この宣言は、定義を置換しません。この宣言は、外部的に定義された変数を記述します。

extern 宣言は、関数の外側またはブロックの先頭に現れます。宣言が、関数を記述するか、関数の外側で現れて外部結合を持つオブジェクトを記述する場合は、キーワード extern はオプションです。

ある ID の宣言がファイル・スコープにすでにある場合は、ブロック内にある同じ ID の extern 宣言は、同じオブジェクトを参照します。 ID に対するほかの宣言が、ファイル・スコープにない場合は、その ID は外部結合を持ちます。

C++ C++ は、extern ストレージ・クラス指定子の使用を、オブジェクト名または関数名に制限します。型宣言と一緒に extern 指定子を使用することは許可されていません。 extern 宣言を、クラス・スコープ内で発生させることはできません。

外部変数のストレージ期間

すべての extern オブジェクトには、静的ストレージ期間が指定されています。メモリーは、main 関数が実行される前に extern オブジェクトに割り当てられ、プログラムが終了すると解放されます。変数のスコープは、それがプログラム・テキスト内のどこで宣言されたかによって異なります。宣言をブロック内に置くと、その変数はブロック・スコープを持ちます。それ以外の場合は、ファイル・スコープを持ちます。

外部変数のリンケージ

C スコープと同様に、extern が宣言された変数のリンケージは、プログラム・テキスト内の宣言の場所によって異なります。変数宣言が関数定義の外側に置かれ、ファイル内で以前に static と宣言されている場合、その変数は内部結合を持ちます。それ以外の場合は、ほとんどの場合、外部結合を持ちます。関数の外側で発生するオブジェクトや、ストレージ・クラス指定子を含まないオブジェクトの宣言では、すべて外部リンケージを持つ ID を宣言します。

C++ 名前なしネーム・スペース内のオブジェクトの場合、リンケージは、外部リンケージであっても名前は固有であり、そのため、他の変換単位から見ると、名前は事実上は内部リンケージを持ちます。

関連情報

- 8 ページの『外部結合』
- 82 ページの『初期化およびストレージ・クラス』
- 174 ページの『extern ストレージ・クラス指定子』
- 5 ページの『クラス・スコープ (C++ のみ)』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

mutable ストレージ・クラス指定子 (C++ のみ)

C++ mutable ストレージ・クラス指定子は、クラス・データ・メンバーでのみ使用されます。そのメンバーが const として宣言されたオブジェクトの一部であったとしても、そのメンバーを変更可能にします。 mutable 指定子を、static または const と宣言された名前と一緒に、または参照メンバーと一緒に使用することはできません。

以下の例では、次のようになります。

```
class A
{
  public:
    A() : x(4), y(5) { };
    mutable int x;
    int y;
};

int main()
{
  const A var2;
  var2.x = 345;
  // var2.y = 2345;
}
```

コンパイラーは、代入 var2.y = 2345 を許可しません。なぜなら、var2 は const として宣言されているからです。コンパイラーは、代入 var2.x = 345 は許可します。なぜなら、A::x は、mutable として宣言されているからです。

関連情報

- 62 ページの『型修飾子』
- 80 ページの『参照 (C++ のみ)』

register ストレージ・クラス指定子

register ストレージ・クラス指定子は、オブジェクトをマシン・レジスターに保管するようにコンパイラーに指示します。 register ストレージ・クラス指定子は通常、アクセス時間を最小化することでパフォーマンスを向上させることを期待して、ループ制御変数などの頻繁に使用される変数で指定されます。ただし、コンパイラーは、この要求を満たす必要がありません。多くのシステムで使用できるレジスターのサイズと数は制限されているので、実際にレジスターに書き込まれる変数は少なくなります。コンパイラーが、マシン・レジスターを register オブジェクトに割り振らない場合、そのオブジェクトは、ストレージ・クラス指定子 auto を持っているものとして扱われます。

register ストレージ・クラス指定子を持つオブジェクトは、ブロック内に定義するか、または関数へのパラメーターとして宣言する必要があります。

register ストレージ・クラス指定子には、次の制約があります。

- C ポインターを使用して、register ストレージ・クラス指定子がある参照オブジェクトを指すことはできません。
- C register ストレージ・クラス指定子は、グローバル・スコープでオブジェクトを宣言する際には使用できません。
- C register にはアドレスがありません。したがって、アドレス演算子 (&) を register 変数に適用することはできません。
- C++ register ストレージ・クラス指定子は、ネーム・スペース・スコープでのデータ宣言には使用できません。

C++ のみ。

C とは異なり、C++ では、register ストレージ・クラスが指定されているオブジェクトのアドレスを取ることができます。次に例を示します。

```
register int i;
int* b = &i;      // valid in C++, but not in C
```

C++ のみ。 の終り

レジスター変数のストレージ期間

register ストレージ・クラス指定子があるオブジェクトには、自動ストレージ期間が指定されます。ブロックに入るたびに、そのブロックに定義された register オブジェクトに対するストレージが使用可能になります。ブロックが終了すると、そのオブジェクトは使用できなくなります。

再帰的に呼び出す関数に register オブジェクトが定義される場合は、ブロックが呼び出されるたびに、メモリーを変数に割り当てます。

レジスター変数のリンケージ

register オブジェクトは auto ストレージ・クラスのオブジェクトと同等に扱われるため、リンケージはありません。

関連情報

- 82 ページの『初期化およびストレージ・クラス』
- 2 ページの『ブロック/ローカル・スコープ』
- 80 ページの『参照 (C++ のみ)』

__thread ストレージ・クラス指定子

IBM 拡張

__thread ストレージ・クラスは、静的変数をスレッド・ローカル・ストレージ期間を持つものとしてマークします。つまり、マルチスレッド・アプリケーションでは、変数を使用するスレッドごとに変数の固有インスタンスが作成されて、スレッドの終了時に破棄されます。 __thread ストレージ・クラス指定子により、スレッド・セーフティーを容易に保証することができます。スレッドごとにオブジェクトを宣言することで、スレッド同期化に関する低水準のプログラミングや大幅なプログラム再構築を必要とせずに、競合状態を心配しないで、複数のスレッドからオブジェクトにアクセスできます。

注: __thread キーワードを認識するには、**LANGLVL(*EXTENDED)** オプションを指定してコンパイルする必要があります。詳細については、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** を参照してください。

指定子には次のいずれかを適用することができます。

- グローバル変数
- ファイル・スコープの静的変数
- 関数スコープの静的変数
- C++ クラスの静的データ・メンバー

ブロック・スコープの自動変数および非静的データ・メンバーには適用できません。

以下の例のように、この指定子は、それ自身だけで使用するか、あるいは static または extern 指定子を直前に付加して使用することができます。

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

__thread 指定子でマークされた変数は、初期化することも、未初期化にすることもできます。 C++ __thread 変数は、定数式を使用して初期化する必要があり、静的コンストラクターを持ってはなりません。

address-of 演算子 (&) をスレッド・ローカル変数に適用すると、変数の現行スレッドのインスタンスの実行時アドレスが返されます。そのスレッドは、このアドレスを任意の別のスレッドに渡すことができますが、最初のスレッドが終了すると、そのスレッド・ローカル変数へのポインターはすべて無効になります。

関連情報

- 「*ILE C/C++ コンパイラー参照*」にある **LANGLVL**

IBM 拡張 の終り

型指定子

型指定子は、宣言されるオブジェクトの型を指示します。使用可能な型指定子の種類は、以下のとおりです。

- 基本型または組み込み型:
  - 算術型
    - 整数型
    - ブール型
    - 浮動小数点型
    - 文字型
  - void 型
- ユーザー定義型

関連情報

- 180 ページの『関数からの戻りの型指定子』

整数型

整数型は、次のカテゴリーに分かれます。

- 符号付き整数型:
  - signed char
  - short int
  - int
  - long int
  - long long int
- 符号なし整数型:
  - unsigned char
  - unsigned short int
  - unsigned int
  - unsigned long int
  - unsigned long long int

`unsigned` 接頭部は、オブジェクトが負以外の整数であることを指示しています。各 unsigned 型は、signed 型のストレージと同じサイズのストレージを提供します。例えば、`int` は、`unsigned int` と同じストレージを予約します。 signed 型が符号ビットを予約するので、unsigned 型は等価の signed 型よりも大きい正の整数値を保持できます。

単純な整数の定義または宣言の宣言子は、 ID です。単純整数の定義の初期化は、整数定数、または整数に割り当て可能な値に数値化される式を使用して行うことができます。

C++ のみ。

多重定義関数および多重定義演算子の引数が整数型であるときは、同一グループの 2 つの整数型は、別個の型として扱われません。例えば、signed int 引数に対して、int 引数は、多重定義できません。

C++ のみ。 の終り

関連情報

- 17 ページの『整数リテラル』
- 96 ページの『整数変換』
- 95 ページの『算術変換とプロモーション』
- 205 ページの『第 10 章 多重定義 (C++ のみ)』

ブール型

ブール変数を使用して、整数値 0 または 1、あるいはリテラル true または false を保持できます。true および false は、算術値が必要になれば整数 0 および 1 に暗黙的にプロモートされます。ブール型は符号なしであって、標準の符号なし整数型カテゴリーの中では最も低いランキングです。指定子 signed、unsigned、short、または long によってさらに修飾できません。単純な代入では、左方オペランドがブール型の場合、右方オペランドは算術型またはポインターでなければなりません。

ブール型を使用して、*ブール論理テスト* を作成できます。ブール論理テストは、論理演算の結果を表すために使用されます。次に例を示します。

```
_Bool f(int a, int b)
{
  return a==b;
}
```

a と b が同じ値を持っている場合、f は true を戻します。そうでなければ、f は false を戻します。

C のみ

ブール型は C99 フィーチャーです。ブール変数を宣言するには、bool 型指定子を使用します。

C のみ の終り

C++ のみ。

C++ でブール変数を宣言するには、bool 型指定子を使用します。同等、関連、および論理の各演算子の結果は、型 bool の結果になります。つまりブール定数 true または false のいずれになります。

C++ のみ。 の終り

関連情報

- 19 ページの『ブール・リテラル』
- 96 ページの『ブール変換』

浮動小数点型

浮動小数点型指定子は、次のカテゴリーに分かれます。

- 『実数浮動小数点型』

実数浮動小数点型

汎用 (2 進) 浮動小数点型は、以下のもので構成されます。

- float
- double
- long double

IBM 拡張

10 進浮動小数点型は、以下のもので構成されます。

- _Decimal32
- _Decimal64
- _Decimal128

IBM 拡張 の終り

下表に、実数浮動小数点型の大きさの範囲を示します。

表 10. 実数浮動小数点型の大きさの範囲

Type	Range
float	約 1.2^{-38} から 3.4^{38}
double, long double	約 2.2^{-308} から 1.8^{308}
_Decimal32	0.000001^{-95} から 9.999999^{96}
_Decimal64	0.000000000000001^{-383} から 9.999999999999999^{384}
_Decimal128	$0.000000000000000000000000000000001^{-6143}$ から $9.999999999999999999999999999999999^{6144}$

浮動小数点定数が長すぎたり、短すぎたりする場合は、言語での結果は予期できません。

単純な浮動小数点宣言の宣言子は、ID です。単純な浮動小数点変数の初期化は、浮動定数、または、整数あるいは浮動小数点数に数値化される変数または式を使用して行います。

IBM 拡張

10 進浮動小数点型は、2 進浮動小数点型でサポートされるいずれの演算子とともに使用できます。 また、10 進浮動小数点型とすべての他の整数型または汎用浮動小数点型との間で、暗黙的または明示的な変換を実行できます。ただし、10 進浮動小数点型を他の算術型とともに使用する際には、以下の制限が適用されます。

- 明示的な変換を使用しない限り、算術式で、10 進浮動小数点型と汎用浮動小数点型とを混用することはできません。
- 10 進浮動小数点型と実数 2 進浮動小数点型との間の暗黙的な変換は、単純代入演算子 = を使用した代入を介してのみ許可されます。暗黙的な変換は、単純代入で実行されます。この代入には、関数引数の代入および関数からの戻り値も含まれます。

IBM 拡張 の終り

関連情報

- 19 ページの『浮動小数点リテラル』
- 97 ページの『浮動小数点の型変換』
- 95 ページの『算術変換とプロモーション』

文字型

文字型は、次のカテゴリーに分かれます。

- ナロー文字型:
  - char
  - signed char
  - unsigned char
- ワイド文字型 wchar_t

char 指定子は、整数型です。 wchar_t 型指定子は、ワイド文字リテラルを表すのに十分なストレージを持つ整数型です。(ワイド文字リテラルは、L'x' のように、文字 L を接頭部として付けた文字リテラルです。)

▶ C char は、 signed char および unsigned char とは異なる型であり、これら 3 つの型には互換性はありません。

▶ C++ 多重定義関数を区別する目的で、C++ の char は、 signed char および unsigned char とは別の型です。

char データ・オブジェクトが signed と unsigned のどちらであっても構わない場合、データ型 char としてオブジェクトを宣言できます。そうでない場合は、signed char または unsigned char を明示的に宣言して、単一バイトを占有する数値変数を宣言します。 char (signed または unsigned) が、int に広げられるときは、その値は保存されます。

デフォルトでは、char は unsigned char と同様に動作します。このデフォルトを変更するには、**DFTCHAR(*SIGNED|*UNSIGNED)** オプションまたは **#pragma chars** ディレクティブを使用します。詳細については、「*ILE C/C++ コンパイラー参照*」の **DFTCHAR(*SIGNED|*UNSIGNED)** を参照してください。

関連情報

- 23 ページの『文字リテラル』
- 24 ページの『ストリング・リテラル』
- 95 ページの『算術変換とプロモーション』
- 「*ILE C/C++ コンパイラー参照*」の **DFTCHAR(*SIGNED|*UNSIGNED)**

void 型

void データ型は、常に、値の空集合を表します。型指定子 void を指定して宣言できる唯一のオブジェクトは、ポインターです。

void 型の変数は、宣言できませんが、式は void 型に明示的に変換できます。結果の式は、次のいずれか 1 つでのみ使用できます。

- 式ステートメント
- コンマ式の左方オペランド
- 条件式の 2 番目または 3 番目のオペランド

関連情報

- 74 ページの『ポインター』
- 129 ページの『コンマ演算子 ,』
- 131 ページの『条件式』
- 169 ページの『関数宣言および関数定義』

算術型の互換性 (C のみ)

2 つの算術型は、それらが同じ型である場合に限って互換性があります。

算術型に対してさまざまな組み合わせで型指定子があるとき、異なる型を示す場合と、そうでない場合があります。例えば、型 signed int は、それがビット・フィールドの型として使用される場合を除き int と同じ型ですが、char、signed char、および unsigned char は異なる型です。

型修飾子があると、型は変更されます。つまり、const int は int と同じ型でないため、この 2 つの型には互換性がありません。

ユーザー定義型

以下は、ユーザー定義の型です。

- 構造体および共用体
- 列挙型
- typedef 定義
- C++ クラス
- C++ 詳述型指定子

C++ クラスについては、 219 ページの『第 11 章 クラス (C++ のみ)』 で説明します。詳述型指定子については、 223 ページの『クラス名のスコープ』 で説明しています。

関連情報

- 67 ページの『型属性』

構造体および共用体

構造体 は、データ・オブジェクトの順序付けられたグループから構成されます。配列のエレメントとは異なり、構造体の中のデータ・オブジェクトには、さまざまなデータ型を指定できます。構造体内の各データ・オブジェクトは、*メンバー* または*フィールド* です。

共用体 は、そのすべてのメンバーが、メモリー内の同じ場所から開始するということを除けは、構造体に類似しているオブジェクトです。共用体変数は、一度に、そのメンバーのうちの 1 つのメンバーの値しか表すことができません。

▶ C++ C++ では、構造体および共用体は、メンバーと継承がデフォルトで public であるという点を除いて、クラスと同じです。

構造体または共用体型は、『構造体および共用体の型定義』および 54 ページの『構造体および共用体変数宣言』で説明しているように、その型の変数の定義とは別に宣言できます。または、 55 ページの『単一ステートメントでの構造体および共用体の型および変数の定義』で説明しているように、構造体または共用体データ型とその型のすべての変数を単一のステートメントで定義することもできます。

構造体および共用体は、位置合わせの考慮事項に従います。位置合わせの完全な説明については、「*ILE C/C++ Programmer's Guide*」の 『Aligning data』を参照してください。

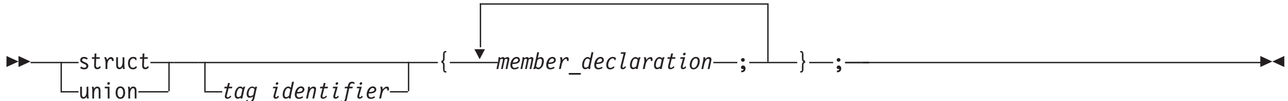
関連情報

- 222 ページの『クラスおよび構造体』

構造体および共用体の型定義

構造体または共用体の*型定義* には、`struct` または `union` キーワードと、その後に続くオプションの ID (構造体タグ) および中括弧で囲まれたメンバーのリストが含まれます。

構造体または共用体の型定義の構文



tag_identifier には、型の名前を指定します。タグ名を指定しない場合には、 55 ページの『単一ステートメントでの構造体および共用体の型および変数の定義』で説明されているように、型の宣言内に、その型を参照するすべての変数定義を配置する必要があります。同様に、型修飾子を、構造体または共用体定義で使用することはできません。 `struct` または `union` キーワードの前に配置される型修飾子は、型定義内で宣言された変数にのみ適用できます。

関連情報

- 68 ページの『aligned 型属性』
- 69 ページの『packed 型属性』

メンバー宣言

メンバーのリストで、構造体または共用体データ型を構造体または共用体に保管できる値の説明とともに指定します。メンバーの定義は、標準変数宣言の形式です。メンバー変数の名前は、単一の構造体または共用体内で固有でなければなりませんが、同じスコープ内で定義された別の構造体または共用体型で同じメンバー名を使用でき、さらに、変数、関数、または型名と同じ名前であっても構いません。

構造体または共用体メンバーは、以下の型以外の型にすることができます。

- すべての可変型
- すべての `void` 型
- ▶ C 関数

- すべての不完全型

不完全型はメンバーとして許可されないため、構造体型または共用体型はメンバーとしてそれ自身のインスタンスを含むことはできませんが、それ自身のインスタンスを指すポインターを含むことはできます。『柔軟な配列メンバー 』 に示すように、特別な場合として、複数のメンバーを持つ構造体の最後のエレメントは、不完全な配列型を持つことができます。この型を*柔軟な配列*メンバー と呼びます。

400 標準 C および C++ への拡張として、 ILE C/C++ では、 52 ページの『ゼロ長配列メンバー』で説明しているように、ゼロ長配列を構造体および共用体のメンバーとして使用することもできます。

C++ 共用体メンバーは、コンストラクター、デストラクター、または多重定義コピー代入演算子を持つクラス・オブジェクトにすることはできません。共用体メンバーは、static というキーワードでは宣言できません。

ビット・フィールドを表さないメンバーは、型修飾子 volatile または const のいずれかの型で修飾することができます。結果は左辺値です。

構造体メンバーはメモリー・アドレスに昇順に割り当てられます。最初のコンポーネントは、構造体名そのものの先頭アドレスから始まります。コンポーネントの適切な位置合わせを可能にするために、構造体レイアウト内の連続メンバーの間に、埋め込みバイトが配置されることがあります。

共用体に割り振られるストレージは、共用体の最も大きいメンバーで必要なストレージです (これに、最も厳格な要件を持つメンバーの自然な境界で共用体が終了するのに必要な埋め込みが加わります)。すべての共用体のコンポーネントは、メモリー内で効果的にオーバーレイされます。つまり、共用体の各メンバーは共用体の開始時に始動するストレージが割り振られ、一度に 1 メンバーしかそのストレージを占有することができません。

柔軟な配列メンバー ： *柔軟な配列*メンバー は、不完全型であっても、構造体に複数の名前付きメンバーが含まれていれば、構造体の最後のエレメントとして許可されます。柔軟な配列メンバーは C99 フィーチャーであり、可変長オブジェクトにアクセスするために使用できます。このメンバーは、以下のように、空指標を使用して宣言されます。

array_identifier[ ];

例えば、b は、Foo の柔軟な配列メンバーです。

```
struct Foo{
   int a;
   int b[];
};
```

柔軟な配列メンバーは不完全型であるため、sizeof 演算子を柔軟な配列に適用することはできません。

柔軟な配列メンバーを含むどの構造体も、別の構造体または配列のメンバーにすることはできません。

IBM 拡張

ILE C/C++ は、標準 C および C++ を拡張して、柔軟な配列に関する制限を緩和し、以下を可能にしています。

- 柔軟な配列メンバーは、最後のメンバーとしてだけではなく、構造体のどの部分でも宣言できます。

  C++ 柔軟な配列メンバーに続くすべてのメンバーの型は、柔軟な配列メンバーの型と互換性がある必要があります。

C 柔軟な配列メンバーに続くメンバーの型は、柔軟な配列メンバーの型との互換性を持つ必要はありませんが、互換性がない場合には警告が出されます。

- 柔軟な配列メンバーを含む構造体を、他の構造体のメンバーにすることができます。
- 柔軟な配列メンバーを静的に初期化できます。

以下の例では、次のようになります。

```
struct Foo{
   int a;
   int b[];
};

struct Foo foo1 = { 55, {6, 8, 10} };
struct Foo foo2 = { 55, {15, 6, 14, 90} };
```

foo1 は、3 エレメントの配列 b を作成しています。これらのエレメントは、6、8、および 10 に初期化されています。一方、foo2 は、4 エレメントの配列を作成しています。これらのエレメントは、15、6、14、および 90 に初期化されています。

柔軟な配列メンバーは、ネスト構造体の最外部内に存在する場合にのみ初期化できます。内側の構造体のメンバーは初期化できません。

IBM 拡張 の終り

関連情報

- 79 ページの『可変長配列 (C++ のみ)』

ゼロ長配列メンバー:

IBM 拡張

ゼロ長配列とは、次元が指定されていない配列です。柔軟な配列メンバーと同様に、ゼロ長配列は、可変長オブジェクトにアクセスするために使用できます。

以下のように、ゼロ長配列は、次元としてゼロを指定して明示的に宣言する必要があります。

array_identifier[0]

柔軟な配列メンバーと同様に、ゼロ長配列は、最後のメンバーとしてだけではなく、構造体のどの部分でも宣言できます。

C++ ゼロ長配列に続くすべてのメンバーの型は、ゼロ長配列の型と互換性がある必要があります。

C ゼロ長配列に続くメンバーの型は、ゼロ長配列の型との互換性を持つ必要はありませんが、互換性がない場合には警告が出されます。

柔軟な配列メンバーとは異なり、ゼロ長配列を含む構造体は、別の配列のメンバーにすることができます。また、ゼロ長配列に sizeof 演算子を適用することもでき、返される値は 0 になります。

ゼロ長配列は、空集合を使用した場合にのみ静的に初期化できます。次に例を示します。

```
struct foo{
   int a;
   char b[0];
}; bar = { 100, { } };
```

それ以外の場合には、動的に割り振られた配列として初期化する必要があります。

ゼロ長配列メンバーは、ネスト構造体の最外部内に存在する場合にのみ初期化できます。内側の構造体のメンバーは初期化できません。

IBM 拡張 の終り

ビット・フィールド・メンバー: C および C++ では、両方とも、コンパイラーが通常許容するよりも小さいメモリー・スペースに整数メンバーを保管することができます。 このようなスペース節約構造体メンバーはビット・フィールド と呼ばれ、その幅はビット数で明示的に宣言することができます。ビット・フィールドは、データ構造を固定のハードウェア表現に対応させなければならない、移植できそうにないプログラムで使用します。

ビット・フィールド・メンバー宣言の構文

```
►►─type_specifier──┬────────────┬─:─constant_expression─;──────────────►◄
                   └─declarator─┘
```

constant_expression は、フィールド幅をビット単位で示す定数整数式です。ビット・フィールド宣言では、型修飾子 `const` または `volatile` のいずれかを使用することができません。

C のみ

C99 では、ビット・フィールド用の許容データ型には、修飾と非修飾の `_Bool`、`signed int`、および `unsigned int` があります。ビット・フィールドのデフォルトの整数型は、`unsigned` です。

C のみ の終り

C++ のみ。

ビット・フィールドは、任意の整数型または列挙型にすることができます。

C++ のみ。 の終り

最大ビット・フィールド長は 64 バイトです。移植性を向上させるため、32 ビットを超えるサイズのビット・フィールドを使用しないでください。

次の構造体には、3 つのビット・フィールド・メンバー `kingdom`、`phylum`、および `genus` があり、それぞれ 12、6、2 ビットを占有します。

```
struct taxonomy {
     int kingdom : 12;
     int phylum : 6;
     int genus : 2;
     };
```

ビット・フィールドに範囲外の値を割り当てると、下位のビット・パターンは保存され、適切なビットが割り当てられます。

次の制約事項は、ビット・フィールドに適用されます。次のことはできません。

- ビット・フィールドの配列の定義
- ビット・フィールドのアドレスの取得

- ビット・フィールドを指すポインターの保持
- ビット・フィールドへの参照の保持

一連のビット・フィールドが int のサイズいっぱいにならない場合は、埋め込みが行われます。埋め込みの量は、構造体メンバーの位置合わせ特性によって決定されます。場合によっては、ビット・フィールドが、ワード境界にまたがることができます。

長さ 0 のビット・フィールドは、名前なしのビット・フィールドにする必要があります。名前なしのビット・フィールドは、参照または初期化することはできません。

次の例は、埋め込みの例を示します。この例は、すべてのインプリメンテーションに有効です。 int は、4 バイトを占めると想定します。例では、ID kitchen を、struct on_off 型になるように宣言します。

```
struct on_off {
                  unsigned light : 1;
                  unsigned toaster : 1;
                  int count;            /* 4 bytes */
                  unsigned ac : 4;
                  unsigned : 4;
                  unsigned clock : 1;
                  unsigned : 0;
                  unsigned flag : 1;
                 } kitchen ;
```

構造体 kitchen には、合計 16 バイトの 8 つのメンバーが含まれます。次の表に、各メンバーが占有するストレージを記述します。

メンバー名	占有されるストレージ
light	1 ビット
toaster	1 ビット
(埋め込み — 30 ビット)	次の int 境界まで
count	int のサイズ (4 バイト)
ac	4 ビット
(名前なしフィールド)	4 ビット
clock	1 ビット
(埋め込み — 23 ビット)	次の int 境界まで (名前なしフィールド)
flag	1 ビット
(埋め込み — 31 ビット)	次の int 境界まで

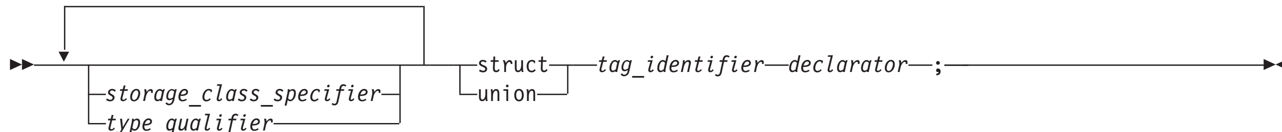
関連情報

- 「*ILE C/C++ Programmer's Guide*」の『Alignment of bit fields』

構造体および共用体変数宣言

構造体または共用体宣言 は、宣言が中括弧で囲まれたメンバーのリストを持っていないということを除けば、定義と同じ形式です。構造体または共用体データ型が指定された変数を定義する前に、構造体または共用体データ型を宣言する必要があります。

構造体または共用体変数宣言の構文



tag_identifier には、構造体または共用体の以前に定義したデータ型を指定します。

C++ 構造体変数宣言では、キーワード `struct` はオプションです。

任意のストレージ・クラスを持つ構造体または共用体を宣言できます。変数のストレージ・クラス指定子およびすべての型修飾子は、ステートメントの先頭で指定する必要があります。 `register` ストレージ・クラス指定子を使用して宣言された構造体または共用体は、自動変数として扱われます。

以下の例では、構造体型 `address` を定義しています。

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
    };
```

以下の例では、型 `address` の 2 つの構造体変数を宣言しています。

```
struct address perm_address;
struct address temp_address;
```

関連情報

- 91 ページの『aligned 変数属性』
- 63 ページの『__align 型修飾子』
- 92 ページの『packed 変数属性』
- 83 ページの『構造体および共用体の初期化』
- 59 ページの『構造体、共用体、および列挙型の互換性 (C のみ)』
- 109 ページの『ドット演算子 .』
- 110 ページの『矢印演算子 ->』

単一ステートメントでの構造体および共用体の型および変数の定義

変数定義の後に宣言子およびオプションの初期化指定子を指定することで、構造体 (または共用体) 型および構造体 (または共用体) 変数を単一のステートメントで定義できます。次の例では、共用体データ型 (名前なし) と共用体変数 (名前 `length` 付き) を定義します。

```
union {
        float meters;
        double centimeters;
        long inches;
      } length;
```

なお、この例ではデータ型の名前を付けていないので、`length` がこのデータ型にすることができる唯一の変数です。 `struct` または `union` キーワードの後に ID を指定することで、データ型の名前が付けられ、プログラムのこれ以降の部分で、このデータ型の追加の変数を宣言できます。

単一または複数の変数のストレージ・クラス指定子を指定するには、ステートメントの先頭にストレージ・クラス指定子を入れる必要があります。次に例を示します。

```
static struct {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
              } perm_address, temp_address;
```

この場合、perm_address と temp_address の両方に静的ストレージが割り当てられます。

型定義で宣言された単一または複数の変数に対して、型修飾子を適用できます。以下の例は両方とも有効です。

```
volatile struct class1 {
                        char descript[20];
                        long code;
                        short complete;
                     }  file1, file2;
```

```
struct class1 {
                        char descript[20];
                        long code;
                        short complete;
                     }  volatile file1, file2;
```

両方の場合で、構造体 file1 と file2 は、volatile として修飾されています。

関連情報

- 83 ページの『構造体および共用体の初期化』
- 39 ページの『ストレージ・クラス指定子』
- 62 ページの『型修飾子』

構造体および共用体メンバーへのアクセス

構造体または共用体変数を宣言した後には、変数名とドット演算子 (.) とメンバー名、またはポインターと矢印演算子 (->) とメンバー名を指定することで、メンバーを参照します。例えば、次の例

```
perm_address.prov = "Ontario";
p_perm_address -> prov = "Ontario";
```

は、ストリング "Ontario" を、構造体 perm_address にあるポインター prov に代入します。

ビット・フィールドを含む、構造体および共用体のメンバーへのすべての参照は完全修飾する必要があります。前の例では、4 番目のフィールドは、prov だけでは参照できず、perm_address.prov とした場合にのみ参照できます。

関連情報

- 109 ページの『ドット演算子 .』
- 110 ページの『矢印演算子 ->』

無名共用体

無名共用体 は、名前が付けられていない共用体です。その後に、宣言子を続けることはできません。無名共用体は、型ではありません。つまり、名前なしオブジェクトを定義します。

無名共用体のメンバー名は、共用体が宣言されているスコープ内のほかの名前と区別する必要があります。メンバー・アクセス構文を追加せずに、共用体スコープ内で、メンバー名を直接使用できます。

例えば、次のコードでは、データ・メンバー i および cptr に直接アクセスできます。これは、これらのデータ・メンバーが、無名共用体を含むスコープにあるからです。i と cptr は、共用体メンバーで、同じアドレスが指定されているので、一度にいずれか一方のみしか使用できません。メンバー cptr への代入は、メンバー i の値を変更します。

```
void f()
{
union { int i; char* cptr ; };
/* . . .  */
i = 5;
cptr = "string_in_union"; // overrides the value 5
}
```

C++ 無名共用体は、 protected メンバーも private メンバーも持つことができず、またメンバー関数も持つことができません。グローバルまたはネーム・スペース無名共用体は、キーワード static を使って宣言される必要があります。

関連情報

- 40 ページの『static ストレージ・クラス指定子』
- 231 ページの『メンバー関数』

列挙型

列挙型 は、整数定数を表す名前付き値のセット (*列挙型定数* と呼ばれる) で構成されるデータ型です。列挙は、値ごとに名前を作成するときにそれぞれの値をリスト (列挙) しなければならないため、*列挙された型* とも呼ばれます。列挙型は、整数定数のセットを定義しグループ化する方法を提供するのに加え、少数の可能な値を持つ変数にとって役立ちます。

列挙型は、『列挙型定義』および 59 ページの『列挙型変数宣言』で説明しているように、その型の変数の定義とは別に宣言できます。または、 59 ページの『単一ステートメントでの列挙型の型および変数の定義』で説明しているように、列挙データ型とその型のすべての変数を単一のステートメントで定義することもできます。

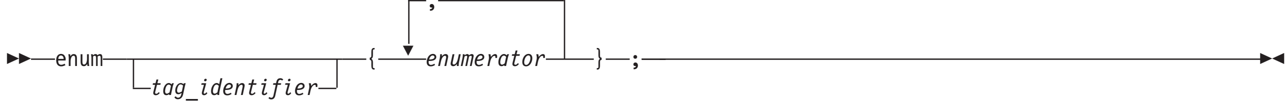
関連情報

- 95 ページの『算術変換とプロモーション』

列挙型定義

列挙型定義には、enum キーワードと、その後に続くオプションの ID (列挙型タグ) および中括弧で囲まれた列挙子のリストが含まれます。コンマは、列挙子のリスト内で各列挙子を分離します。

列挙型定義の構文



tag_identifier で、列挙型に名前を付けます。タグ名を指定しない場合には、 59 ページの『単一ステートメントでの列挙型の型および変数の定義』で説明されているように、列挙型の宣言内に、その型を参照するすべての変数定義を配置する必要があります。同様に、型修飾子を、列挙型定義で使用することはできません。 enum キーワードの前に配置される型修飾子は、型定義内で宣言された変数にのみ適用できます。

C++ C++ は、列挙子リストでの末尾コンマをサポートします。

列挙型メンバー: 列挙型メンバー (*列挙子*) のリストにより、データ型と値のセットが提供されます。

列挙型メンバー宣言の構文

```
►►—identifier—┬──────────────────────────┬——————————————————————————————►◄
              └=—enumeration_constant—┘
```

C C では、*列挙型定数* は int 型です。定数式が初期化指定子として使用されている場合、式の値は int の範囲を超えることはできません (すなわちヘッダー limits.h) に定義されているように、INT_MIN ～ INT_MAX となります)。

C++ C++ では、各列挙型定数には、符号付きまたは符号なし整数の値にプロモート可能な値と、整数でなくてもよい別個の型が指定されます。列挙型定数は、整数定数が許可される場所、または、列挙型の値が許可される場所であればどこでも使用できます。

列挙型定数の値は、次の方法で判別されます。

1. 列挙型定数の後の等号 (=) と定数式によって列挙型定数に明示値が与えられます。列挙型定数は、定数式の値を表します。
2. 明示値を割り当てられない場合は、リストの左端の列挙型定数がゼロ (0) の値を受け取ります。
3. 明示的に割り当てられた値がない列挙型定数は、以前の列挙型定数で表される値よりも 1 つ大きい整数値を受け取ります。

次のデータ型宣言は、列挙型定数として oats、wheat、barley、corn、および rice をリストします。各定数の下の番号は、整数値を表します。

```
enum grain { oats, wheat, barley, corn, rice };
   /*         0      1      2      3     4         */

enum grain { oats=1, wheat, barley, corn, rice };
   /*         1        2      3      4     5       */

enum grain { oats, wheat=10, barley, corn=20, rice };
   /*          0     10        11     20       21  */
```

同じ整数を 2 つの異なる列挙型定数に関連付けることができます。例えば、次の定義は有効です。ID suspend と hold には、同じ整数値が指定されます。

```
enum status { run, clear=5, suspend, resume, hold=6 };
   /*          0      5        6       7       6       */
```

各列挙型定数は、列挙が定義されるスコープ内で固有にする必要があります。次の例では、average と poor の 2 番目の宣言が、コンパイラー・エラーの原因になります。

```
func()
    {
       enum score { poor, average, good };
       enum rating { below, average, above };
       int poor;
    }
```

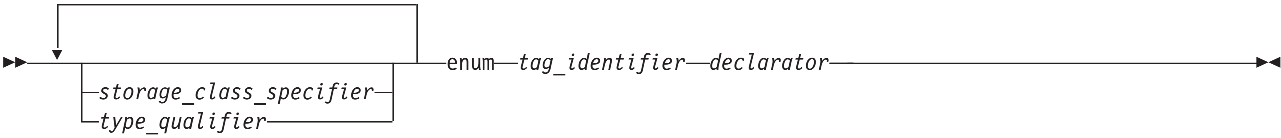
関連情報

- 45 ページの『整数型』

列挙型変数宣言

列挙データ型が指定された変数を定義する前に、列挙データ型を宣言する必要があります。

列挙型変数宣言の構文



tag_identifier には、列挙型の以前に定義したデータ型を指定します。

C++ 列挙型変数宣言では、キーワード enum はオプションです。

関連情報

- 85 ページの『列挙型の初期化』
- 『構造体、共用体、および列挙型の互換性 (C のみ)』

単一ステートメントでの列挙型の型および変数の定義

変数定義の後に、宣言子とオプションの初期化指定子を使用することによって、型と変数を 1 つのステートメントに定義できます。変数のストレージ・クラス指定子を指定するには、ストレージ・クラス指定子を宣言の先頭に入れる必要があります。たとえば、次のとおりです。

```
register enum score { poor=1, average, good } rating = good;
```

C++ のみ。

C++ でも、ストレージ・クラスを宣言子リストの直前に入れます。次に例を示します。

```
enum score { poor=1, average, good } register rating = good;
```

C++ のみ。 の終り

これらの例のいずれも、次の 2 つの宣言と同じです。

```
enum score { poor=1, average, good };
register enum score rating = good;
```

両方の例では、列挙データ型 score と変数 rating を定義します。 rating にはストレージ・クラス指定子 register、データ型 enum score、および初期値 good が指定されます。

データ型の定義をそのデータ型が指定されたすべての変数の定義と結合することによって、データ型に名前を付けないままにすることができます。次に例を示します。

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday } weekday;
```

この例では、変数 weekday を定義します。この変数には、指定した任意の列挙型定数を割り当てることができます。ただし、列挙型定数のこのセットを使用して、追加の列挙型変数を宣言することはできません。

構造体、共用体、および列挙型の互換性 (C のみ)

単一のソース・ファイル内で、構造体または共用体の定義ごとに、他のすべての構造体および共用体型と異なり、かつ互換性がない新規の型が作成されます。ただし、すでに定義された構造体型または共用体型への

参照である型指定子は同じ型です。タグが、参照と定義を関連付け、型名として効果的に機能します。つまり、以下の例では、構造体 j と k の型のみが互換性を持ちます。

```
struct   { int a; int b; } h;
struct   { int a; int b; } i;
struct S { int a; int b; } j;
struct S k;
```

互換性のある構造体は、相互に代入できます。

同一メンバーがあるが別のタグが付けられている構造体または共用体には、互換性はなく、互いに割り当てることはできません。同一メンバーを持っているが異なった位置合わせを使用している構造体または共用体にも互換性はなく、互いに割り当てることはできません。

コンパイラーが列挙型の変数および定数を整数型として扱うため、型互換性に関係なく、異なる列挙型の値を自由に混用できます。列挙型と、それを表す整数型の間の互換性は、コンパイラー・オプションと関連プラグマで制御されます。 **ENUM** コンパイラー・オプションおよび関連プラグマの詳細については、「*ILE C/C++ コンパイラー参照*」の **ENUM** および **#pragma enum** を参照してください。

関連情報

- 95 ページの『算術変換とプロモーション』
- 219 ページの『第 11 章 クラス (C++ のみ)』
- 50 ページの『構造体および共用体の型定義』
- 36 ページの『不完全型』

別々のソース・ファイル間での互換性

2 つの構造体、共用体、または列挙型の定義が別々のソース・ファイルで定義されると、各ファイルは、理論上は、その型のオブジェクトに対して同じ名前の個別の定義を持つ可能性があります。その 2 つの宣言は互換性がなければなりません。そうでなければ、プログラムのランタイムでの振る舞いは予期できません。そのため、このときの互換性規則は、同じソース・ファイル内の互換性に関する規則よりも制限が厳しくなり、詳細になります。別々にコンパイルされたファイルで定義された構造型、共用体型、および列挙型については、現行ソース・ファイル内の型は複合型です。

別々のソース・ファイル内で宣言された 2 つの構造型、共用体型、または列挙型間の互換性に関する要件は次のとおりです。

- 一方がタグ付きで宣言されたら、他方も同じタグ付きで宣言されること。
- 両方が完全型の場合、そのメンバーは、数が正確に一致し、互換型で宣言され、名前が一致すること。

列挙型の場合、対応するメンバーも同じ値を持つこと。

構造体と共用体については、型互換性に関して以下の追加要件が満たされる必要があります。

- 対応するメンバーは同じ順序で宣言されること (構造体にのみ適用されます)。
- 対応するビット・フィールドは同じ幅を持つこと。

typedef 定義

`typedef` 宣言を使用すると、`int`、`float`、および `double` の型指定子の代わりに使用できる、ユーザー独自の ID を定義できます。 `typedef` 宣言は、ストレージをとりません。 `typedef` を使用して定義した名前は、新しいデータ型ではありませんが、その名前が表すデータ型またはデータ型の組み合わせの同義語です。

typedef の名前のネーム・スペースは、他の ID と同じです。この規則の例外は、typedef の名前が可変的に変更される型を指定する場合です。この場合は、ブロック・スコープを持ちます。

オブジェクトが typedef ID を使用して定義されているときは、定義されたオブジェクトの属性は、あたかも、ID に関連したデータ型を明示的にリストすることによってオブジェクトが定義された場合と、まったく同じです。

関連情報

- 73 ページの『型名』
- 45 ページの『型指定子』
- 49 ページの『構造体および共用体』
- 219 ページの『第 11 章 クラス (C++ のみ)』

typedef 定義の例

次のステートメントは、LENGTH を int の同義語として定義し、この typedef を使用して length、width、および height を整変数として宣言します。

```
typedef int LENGTH;
LENGTH length, width, height;
```

次の宣言は、上記の宣言と同じです。

```
int length, width, height;
```

同様に、typedef は、クラスの型 構造体、共用体、または C++ クラスを定義するために使用できます。次に例を示します。

```
typedef struct {
                int scruples;
                int drams;
                int grains;
               } WEIGHT;
```

そうすると、構造体 WEIGHT は、以下の宣言で使用できます。

```
WEIGHT  chicken, cow, horse, whale;
```

以下の例では、yds の型は、「パラメーター指定なしで int を戻す関数を指すポインター」です。

```
typedef int SCROLL();
extern SCROLL *yds;
```

以下の typedef では、トークン struct は型名の一部です。ex1 の型は struct a で、ex2 の型は struct b です。

```
typedef struct a { char x; } ex1, *ptr1;
typedef struct b { char x; } ex2, *ptr2;
```

型 ex1 には、型 struct a、および ptr1 が指すオブジェクトの型との互換性があります。型 ex1 には、char、ex2、または struct b との互換性はありません。

C++ のみ。

C++ では、typedef 名は、同じスコープ内で宣言されたどのクラス型名とも異なっている必要があります。 typedef 名がクラス型名と同じである場合、その typedef がクラス名の同義語である場合に限ります。これは、C の場合には当てはまりません。標準 C ヘッダーでは、次のようになります。

```
typedef class C { /*  data and behavior  */ } C;
```

名前を付けずに typedef で定義された C++ のクラスには、ダミーの名前と、リンケージ用の typedef 名が付けられます。このようなクラスには、コンストラクターまたはデストラクターを指定することはできません。次に例を示します。

```
typedef class {
               Trees();
              } Trees;
```

関数 Trees() は、型名が指定されていないクラスの通常のメンバー関数です。上記の例では、Trees は、名前なしクラスの別名で、それ自体はクラス型名ではありません。したがって、Trees() は、そのクラスのコンストラクターになれません。

C++ のみ。 の終り

型修飾子

型修飾子は、以下を指定することで、変数、関数、およびパラメーターの宣言を詳細化するために使用します。

- オブジェクトの値を変更できるかどうか
- オブジェクトの値が常に、レジスターからではなくメモリーから読み取る必要があるかどうか
- 複数のポインターが単一の変更可能なメモリー・アドレスにアクセスできるかどうか

ILE C/C++ では、以下の型修飾子が認識されます。

- 400 __align
- const
- restrict
- volatile

標準 C++ では、型修飾子 const および volatile は、*cv 修飾子* と呼びます。どちらの言語においても、cv 修飾子は、左辺値である式でのみ意味があります。

const および volatile キーワードをポインターで使用する場合には、修飾する先がポインター自身なのか、ポインターが指すオブジェクトなのかを決定する上で、修飾子の配置が重要になります。 volatile または const の場合は、* と ID の間にキーワードを入れる必要があります。次に例を示します。

```
int * volatile x;        /* x is a volatile pointer to an int */
int * const y = &z;      /* y is a const pointer to the int variable z */
```

volatile または const データ・オブジェクトへのポインターの場合は、修飾子が * 演算子の後にないという条件の下で、型指定子および修飾子は任意の順序にすることができます。たとえば、次のとおりです。

```
volatile int *x;         /* x is a pointer to a volatile int
   or                                                        */
int volatile *x;         /* x is a pointer to a volatile int  */

const int *y;            /* y is a pointer to a const int
   or                                                        */
int const *y;            /* y is a pointer to a const int     */
```

以下の例で、これらの宣言のセマンティクスを対比して説明します。

宣言	内容
const int * ptr1;	定数整数へのポインターを定義します。指される値は変更できません。
int * const ptr2;	整数への定数ポインターを定義します。整数は変更できますが、ptr2 はそれ以外のものを指すことができません。
const int * const ptr3;	定数整数への定数ポインターを定義します。指される値もポインター自身も変更できません。

宣言には複数の修飾子を入れることができますが、コンパイラーは重複型修飾子を無視します。

型修飾子は、ユーザー定義型には適用できず、ユーザー定義型から作成されたオブジェクトにのみ適用できます。したがって、以下の宣言は誤りです。

```
volatile struct omega {
                         int limit;
                         char code;
                      }
```

ただし、型の同じ定義内で単一または複数の変数が宣言されている場合には、型修飾子は、ステートメントの先頭、あるいは単一または複数の変数宣言子の前に配置することで、その単一または複数の変数に適用できます。そのため、

```
volatile struct omega {
                      int limit;
                      char code;
                      } group;
```

上記の例では、次の例のストレージと同じストレージを提供します。

```
struct omega {
              int limit;
              char code;
              }  volatile group;
```

両方の例で、volatile 修飾子は、構造体変数 group にのみ適用されます。

型修飾子が構造体、クラス、共用体、またはクラス変数に適用される場合には、構造体、クラス、または共用体のメンバーにも適用されます。

関連情報

- 74 ページの『ポインター』
- 232 ページの『定数および volatile メンバー関数』

__align 型修飾子

IBM 拡張

__align 修飾子は、集合体または静的 (またはグローバル) 変数の明示的な位置合わせを指定できるようにする言語拡張です。ここで指定するバイト境界は、集合体のメンバーではなく集合体全体の位置合わせに影響を与えます。 __align 修飾子は、他の集合体定義の中にネストされている集合体定義には適用されますが、集合体の個々のエレメントには適用されません。パラメーターおよび自動変数については、位置合わせ指定は無視されます。

宣言は、次のいずれかの形式をとります。

単純変数の __align 修飾子の構文

```
▶▶─型指定子─__align─(─int_constant─)─declarator──────────────────────────────▶◀
```

構造体または共用体の __align 修飾子の構文

```
▶▶─__align─(─int_constant─)──┬─struct─┬──┬────────────────┬─{─member_declaration_list─}─;───────▶◀
                             └─union──┘  └─tag_identifier─┘
```

ここで *int_constant* は、バイト位置合わせ境界を示す正の整数値です。使用可能な値は、 1、2、4、8、またはその他の 2 の正の累乗です。

以下の制限および制約が適用されます。

- 変数位置合わせのサイズが型位置合わせのサイズより小さい場合は、__align 修飾子は使用できません。
- 1 つのオブジェクト・ファイル内ですべての位置合わせを表すことができない場合があります。
- __align 修飾子は以下のものには適用できません。
  - 集合体定義の中の個々のエレメント。
  - 配列の中の個々のエレメント。
  - 不完全な型の変数。
  - 宣言はされているが定義はされていない集合体。
  - 他の型の宣言または定義 (例えば typedef)、関数、または列挙型。

__align 修飾子の使用例

以下に、**__align** を静的変数またはグローバル変数に適用した例を示します。

```
int __align(1024) varA;        /* varA is aligned on a 1024-byte boundary
main()              and padded with 1020 bytes              */
{...}

static int __align(512) varB;  /* varB is aligned on a 512-byte boundary
                                    and padded with 508 bytes              */

int __align(128) functionB( ); /* An error                                 */

typedef int __align(128) T;    /* An error                                 */

__align enum C {a, b, c};      /* An error                                 */
```

以下に、集合体メンバーに影響を与えずに、**__align** を適用して、集合体タグの位置合わせおよび埋め込みを行った例を示します。

```
__align(1024) struct structA {int i; int j;}; /* struct structA is aligned
                                                  on a 1024-byte boundary
                                                  with size including padding
                                                  of 1024 bytes            */

__align(1024) union unionA {int i; int j;}; /* union unionA is aligned
                                                  on a 1024-byte boundary
                                                  with size including padding
                                                  of 1024 bytes           */
```

以下に、**__align** を構造体または共用体に適用した例を示します。ここでは、その構造体または共用体を使用する集合体のサイズおよび位置合わせは影響を受けます。

```
__align(128) struct S {int i;};     /* sizeof(struct S) == 128            */

struct S sarray[10];                /* sarray is aligned on 128-byte boundary
                                       with sizeof(sarray) == 1280        */

struct S __align(64) svar;          /* error - alignment of variable is
                                         smaller than alignment of type   */

struct S2 {struct S s1; int a;} s2; /* s2 is aligned on 128-byte boundary
                                         with sizeof(s2) == 256           */
```

以下に、**__align** を配列に適用した例を示します。

```
AnyType __align(64) arrayA[10]; /* Only arrayA is aligned on a 64-byte
                                  boundary, and elements within that array
                                  are aligned according to the alignment
                                  of AnyType.  Padding is applied after the
                                  back of the array and does not affect
                                  the size of the array member itself.   */
```

以下に、変数位置合わせのサイズが型位置合わせのサイズと異なる **__align** を適用した例を示します。

```
__align(64) struct S {int i;};

struct S __align(32) s1;          /* error, alignment of variable is smaller
                                     than alignment of type                */

struct S __align(128) s2;         /* s2 is aligned on 128-byte boundary     */

struct S __align(16) s3[10];      /* error                                  */

int __align(1) s4;                /* error                                  */

__align(1) struct S {int i;};     /* error                                  */
```

関連情報

- 91 ページの『aligned 変数属性』
- 115 ページの『__alignof__ 演算子』
- 「*ILE C/C++ Programmer's Guide*」の『Aligning data』

IBM 拡張 の終り

const 型修飾子

const 修飾子はデータ・オブジェクトを、変更できないものとして明示的に宣言します。初期化を行うときに、この値がセットされます。変更可能な左辺値を必要とする式には、const データ・オブジェクトを使用することはできません。例えば、const データ・オブジェクトは、代入ステートメントの左辺では使用できません。

C のみ

const オブジェクトは定数式では使用できません。明示的なストレージ・クラスが指定されていないグローバル const オブジェクトは、デフォルトで extern とみなされます。

C++ のみ。

C++ では、外部で定義された定数を参照するものを除き、すべての const 宣言に初期化指定子が必要です。const オブジェクトは、それが整数であって、定数に初期化される場合のみ、定数式で使用できます。次の例は、このことを示しています。

```
const int k = 10;
int ary[k];     /* allowed in C++, not legal in C */
```

C++ では、明示的なストレージ・クラスが指定されていないグローバル const オブジェクトは、デフォルトで static とみなされ、内部リンケージを持ちます。

```
const int k = 12;  /* Different meanings in C and C++ */

static const int k2 = 120;  /* Same meaning in C and C++ */
extern const int k3 = 121;  /* Same meaning in C and C++ */
```

リンケージは内部のものとみなされるため、const オブジェクトは、C より C++ の方が簡単にヘッダー・ファイル内に定義できます。

C++ のみ。 の終り

1 つの項目が、const および volatile の両方になり得ます。この場合、その項目はそれ自体のプログラムによって正当に変更することはできないが、ある種の非同期処理によって変更することはできます。

関連情報

- 351 ページの『#define ディレクティブ』
- 236 ページの『this ポインター』

restrict 型修飾子

ポインターは、メモリー内のロケーションのアドレスです。複数のポインターがメモリーの同じチャンクにアクセスし、プログラムの途中でそのチャンクを変更することができます。 restrict (または __restrict または __restrict__)[1] 型修飾子は、 restrict で修飾されたポインターによってアドレス指定されるメモリーが変更された場合に、他のポインターがその同じメモリーにアクセスしないようにする、コンパイラーへの指示です。 コンパイラーは、不適切な振る舞いが生じないような方法によって、restrict で修飾されたポインターを含めてコードを最適化することを選択する場合があります。restrict で修飾されたポインターが意図されたとおりに使用されていることを確かめることは、プログラマーの責任です。そうしない場合、定義されていない振る舞いが生じる場合があります。

メモリーの特定のチャンクが変更されていなければ、複数の制限付きポインターを使ってそれに別名を付けることができます。以下の例では、制限付きパラメーターを foo() のパラメーターとして示し、2 つの制限付きパラメーターを通じて未変更オブジェクトに別名を付ける方法が示されています。

```
void foo(int n, int * restrict  a, int * restrict b, int * restrict  c)
 {
     int i;
     for (i = 0; i < n; i++)
     a[i] = b[i] + c[i];
 }
```

制限付きポインター間の割り当ては制限されており、関数呼び出しと、ネストされた同等のブロックの間に区別はありません。

```
{
      int * restrict  x;
      int * restrict  y;
      x = y; // undefined
      {
         int * restrict  x1 = x; // okay
         int * restrict  y1 = y; // okay
         x = y1;  // undefined
      }
   }
```

制限付きポインターを含む、ネストされたブロックでは、外側のブロックから内側のブロックへの制限付きポインターの割り当てのみが許可されます。例外は、制限付きポインターが宣言されているブロックが実行を終了した時です。プログラムのこの時点では、制限付きポインターの値をそれが宣言されたブロックの外部に持ち出すことができます。

関連情報

volatile 型修飾子

volatile 修飾子は、コンパイラーの制御または検出以外の方法 (システム・クロックや別のプログラムによって更新される変数など) で、値を変更できるデータ・オブジェクトを宣言します。これにより、オブジェクトの値をレジスターに保管して、変更される可能性があるメモリーからではなくレジスターから再読み取りすることで、オブジェクトを参照するコードをコンパイラーが最適化することがなくなります。

volatile で修飾された任意の左辺値の式にアクセスすると、副次作用が生じます。副次作用とは、実行環境の状態が変化するということです。

オブジェクト型 ″pointer to volatile″ への参照は最適化されますが、それが指す先のオブジェクトへの参照を最適化することはできません。 ″pointer to volatile T″ 型の値を ″pointer to T″ 型のオブジェクトに割り当てるためには、明示的なキャストを使用しなければなりません。 volatile オブジェクトの有効な使用法は、以下のとおりです。

```
volatile int * pvol;
int *ptr;
pvol = ptr;              /* Legal */
ptr = (int *)pvol;       /* Explicit cast required */
```

C シグナル処理関数は、変数が volatile として宣言されていれば、型 sig_atomic_t の変数に値を保管できます。これは、シグナル処理関数が静的ストレージ期間を使用した変数にアクセスできないという規則の例外です。

1 つの項目が、const および volatile の両方になり得ます。この場合、その項目はそれ自体のプログラムによって正当に変更することはできないが、ある種の非同期処理によって変更することはできます。

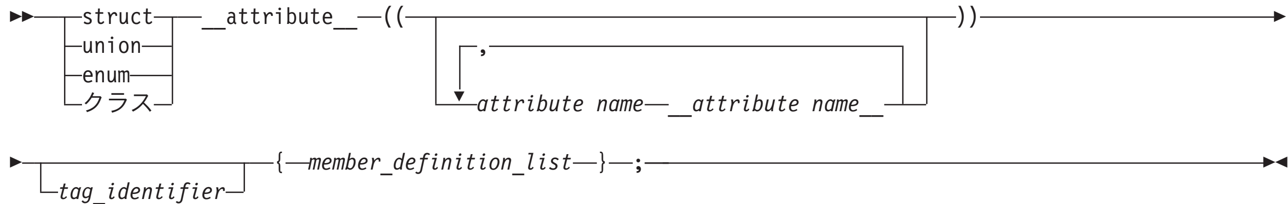
型属性

IBM 拡張

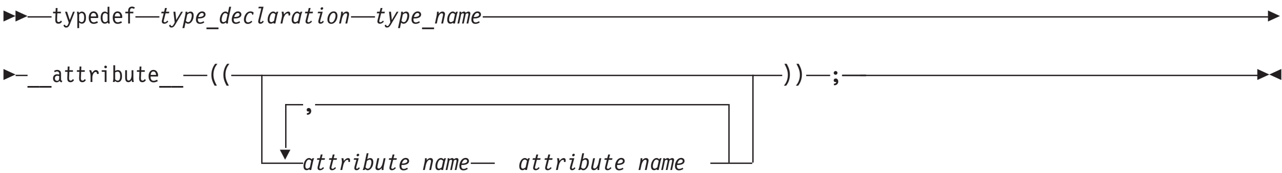
型属性は言語拡張です。この言語フィーチャーを使用することで、名前付き属性を使用して、データ・オブジェクトの特殊プロパティーを指定できます。*型* 属性は、ユーザー定義型の定義 (構造体、共用体、列挙型、クラス、および typedef の定義など) に適用されます。適用された型を持つものとして宣言されたすべての変数に、この属性が適用されます。

型属性は、キーワード __attribute__ に続いて、属性名、および属性名に必要な追加の引数がある場合にはその引数を設定して、指定します。バリエーションはありますが、型属性の構文は、一般的に以下のような形式を取ります。

型属性の構文 - 集合体型



型属性の構文 - typedef 宣言



attribute name は、2 つの下線文字を前後に付けて指定しても付けないで指定しても構いません。ただし、2 つの下線文字を使用すると、同じ名前のマクロとの名前の競合の可能性が小さくなります。サポートされない属性名については、コンパイラーは診断メッセージを出して、その属性の指定を無視します。同じ属性指定で複数の属性名を指定できます。

次の型属性がサポートされます。

- aligned 型属性
- packed 型属性
- transparent_union 型属性 (C のみ)

関連情報

- 89 ページの『変数属性』
- 185 ページの『関数属性』

aligned 型属性

aligned 型属性により、構造体、クラス、共用体、列挙型、または typedef 宣言で作成されたその他のユーザー定義型について、デフォルト位置合わせモードをオーバーライドして、バイト数で表した最小位置合わせ値を指定できます。 aligned 属性は通常、属性が適用される型で宣言されたすべての変数の位置合わせを増加させるために使用します。

aligned 型属性の構文

alignment_factor はバイト数であり、2 の正の累乗に評価される定数式として指定されます。最大で 1048576 バイトの値を指定できます。位置合わせ係数 (およびそれを囲む括弧) を省略した場合には、コン

パイラーは自動的に 16 バイトを使用します。最大値より大きな位置合わせ係数を指定した場合には、属性の指定は無視され、コンパイラーは単に、有効なデフォルト位置合わせを使用します。

指定した位置合わせ値は、その型のすべてのインスタンスに適用されます。また、位置合わせ値は、変数全体に適用されます。変数が集合体である場合には、位置合わせ値は、集合体の個々のメンバーにではなく、集合体全体に適用されます。

以下のすべての例で、aligned 属性は構造体型 A に適用されます。型 A の変数として宣言されているため、a も位置合わせの指定を受けます。これは、型 A として宣言されたすべての他のインスタンスにも当てはまります。

```
struct __attribute__((__aligned__(8))) A {};

struct __attribute__((__aligned__(8))) A {} a;

typedef struct __attribute__((__aligned__(8))) A {} a;
```

関連情報

- 63 ページの『__align 型修飾子』
- 91 ページの『aligned 変数属性』
- 115 ページの『__alignof__ 演算子』
- 「*ILE C/C++ Programmer's Guide*」の 『Aligning data』

packed 型属性

packed 型属性は、構造体、クラス、共用体、または列挙型のメンバーで、最小位置合わせを使用することを指定します。構造体、クラス、または共用体型の位置合わせは、メンバーの場合は 1 バイト、ビット・フィールド・メンバーの場合は 1 ビットです。列挙型の位置合わせは、列挙型内の値の範囲に対応できる最小のサイズです。適用される型のすべてのインスタンスのすべてのメンバーで最小位置合わせが使用されます。

packed 型属性の構文

```
►►──__attribute__──((──┬─packed─────┬─))──────────────────────────────────────►◄
                       └─__packed__─┘
```

aligned 型属性とは異なり、packed 型属性は、typedef 宣言で使用できません。

関連情報

- 63 ページの『__align 型修飾子』
- 92 ページの『packed 変数属性』
- 115 ページの『__alignof__ 演算子』
- 「*ILE C/C++ Programmer's Guide*」の『Aligning data』

transparent_union 型属性 (C のみ)

共用体定義または共用体 typedef 定義に適用される transparent_union 属性は、*透過共用体* としてその共用体が使用できることを指示します。透過共用体が関数仮パラメーターの型であり、その関数が呼び出されると常に、その透過共用体は、明示的キャストなしで、そのいずれかのメンバーの型に一致する任意の型の引数を受け入れることができます。この関数仮パラメーターに対する引数は、その共用体型の最初のメン

バーの呼び出し規則を使用して、透過共用体に渡されます。このため、共用体のすべてのメンバーは、同じマシン表現を持たなければなりません。透過共用体は、互換性の問題を解決するために複数のインターフェースを使用するライブラリー関数で有用です。

transparent_union 型属性の構文

```
►►—__attribute__—((—┬—transparent_union—————┬—))——————————————————————————————————►◄
                    └—__transparent_union__—┘
```

共用体は、完全な共用体型でなければなりません。transparent_union 型属性は、タグ名を持つ無名共用体に適用できます。

transparent_union 型属性がネスト共用体の外側の共用体に適用された場合は、内側の共用体 (つまり、最大メンバー) のサイズを使用して、外側の共用体の他のメンバーと同じマシン表現を持つかどうかを判別します。例えば、次のような場合です。

```
union __attribute__((__transparent_union__)) u_t {
      union u2_t {
      char a;
      short b;
      char c;
      char d;
   };
   int a;
} ;
```

この属性は無視されます。これは、共用体 u_t の先頭メンバー (自身も共用体) が 2 バイトのマシン表現を持っているのに対し、共用体 u_t のもう一方のメンバーの型は int であり、そのマシン表現が 4 バイトであるためです。

同じ規則が、構造体である共用体のメンバーにも適用されます。型属性 transparent_union が適用された共用体のメンバーが struct である場合、メンバーではなく、その struct 全体のマシン表現が考慮されます。

共用体のすべてのメンバーは、共用体の先頭メンバーと同じマシン表現を持たなければなりません。つまり、すべてのメンバーが、共用体の先頭メンバーと同じ量のメモリーで表現可能でなければならないということです。先頭メンバーのマシン表現は、残りの共用体メンバーの最大メモリー・サイズを表します。例えば、型属性 transparent_union が適用された共用体の先頭メンバーの型が int である場合、その後のすべてのメンバーは、4 バイト以内で表現可能でなければなりません。この透過共用体では、1 バイト、2 バイト、または 4 バイトで表現可能なメンバーは有効とみなされます。

浮動小数点型 (float、double、float _Complex、または double _Complex 型) は、透過共用体のメンバーにすることができますが、先頭メンバーにすることはできません。ここでも、透過共用体のすべてのメンバーが先頭メンバーと同じマシン表現を持たなければならないという制限は適用されます。

IBM 拡張 の終り

第 4 章 宣言子

このセクションでは、引き続きデータ宣言の説明を行います。以下のトピックについて説明します。

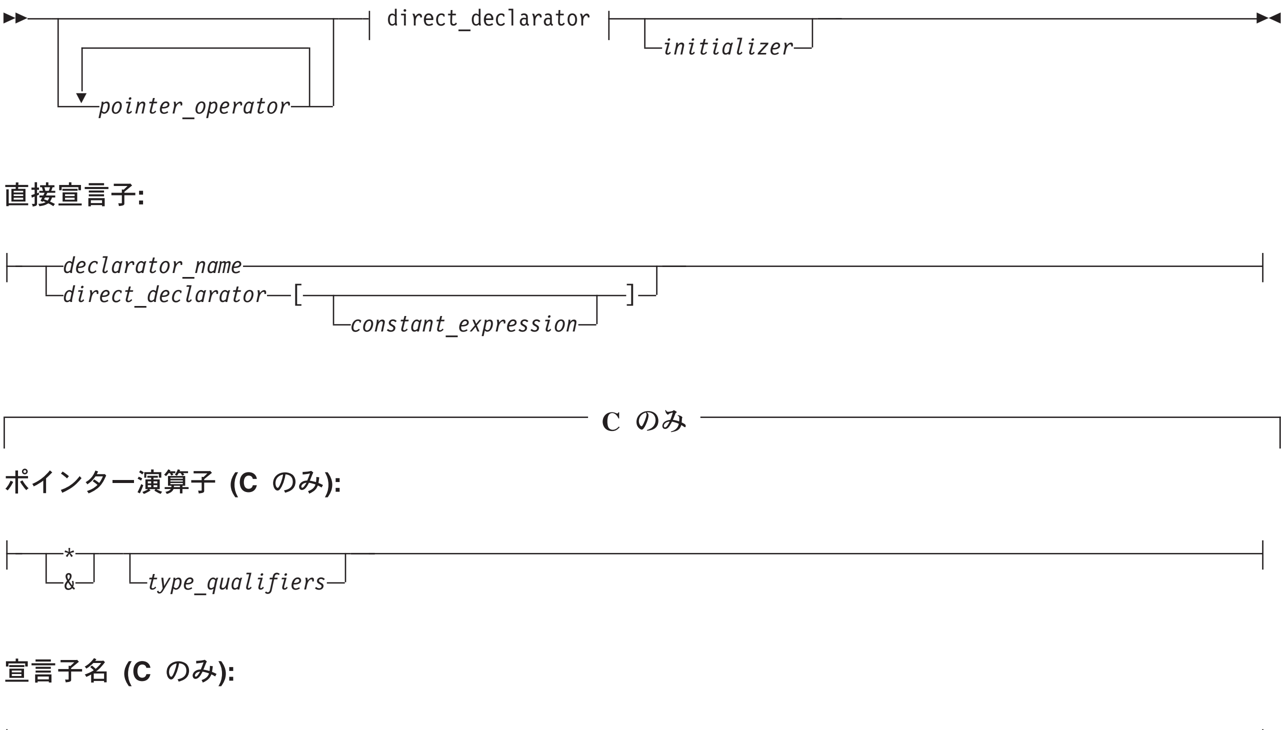
- 『宣言子の概要』
- 73 ページの『型名』
- 74 ページの『ポインター』
- 78 ページの『配列』
- 80 ページの『参照 (C++ のみ)』
- 81 ページの『初期化指定子』
- 89 ページの『変数属性』

宣言子の概要

宣言子 は、データ・オブジェクトまたは関数を指定します。宣言子には、初期化を含めることもできます。宣言子は、多くのデータ定義と宣言および一部の型定義で使用します。

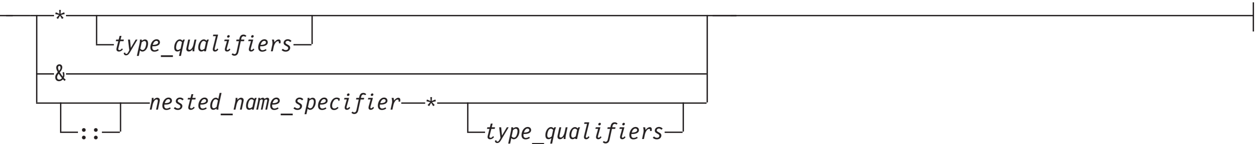
データ宣言の場合、宣言子の形式は次のとおりです。

宣言子の構文

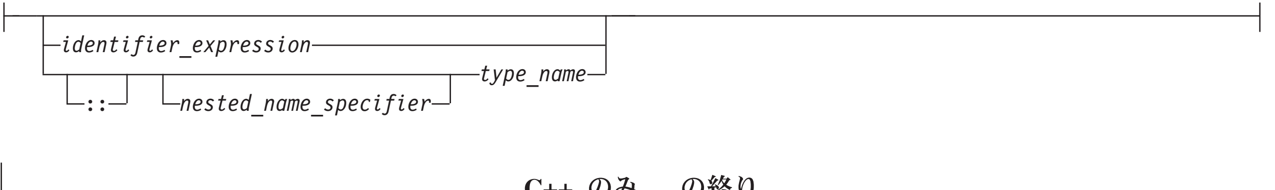


C のみ の終り

ポインター演算子 (C++ のみ):



宣言子名 (C++ のみ):



C++ のみ。 の終り

type_qualifiers は、const および volatile のいずれか、またはその両方を組み合わせたものを表します。

C++ *nested_name_specifier* は、修飾 ID 式です。 *identifier_expression* は、修飾 ID または非修飾 ID のいずれにもなれます。

*初期化指定子*については 81 ページの『初期化指定子』で説明します。

以下は、*派生宣言子* 型と呼ばれるため、このセクションで説明します。

- ポインター
- 配列
- 参照 (C++ のみ)

400 さらに、GNU C および C++ との互換性のため、ILE C/C++ では、*変数属性* を使用して、データ・オブジェクトのプロパティーを変更できます。これは通常、宣言内で宣言子の一部として指定するものであるため、このセクションの 89 ページの『変数属性』で説明しています。

関連情報

- 62 ページの『型修飾子』

宣言子の例

下表で、宣言内の宣言子を示します。

宣言	宣言子	内容
int owner;	owner	owner は、整数データ・オブジェクトです。
int *node;	*node	node は、整数データ・オブジェクトへのポインターです。
int names[126];	names[126]	names は、126 個の整数エレメントの配列です。
volatile int min;	min	min は、volatile 整数です。

宣言	宣言子	内容
int * volatile volume;	* volatile volume	volume は、整数への volatile ポインターです。
volatile int * next;	*next	next は、volatile 整数へのポインターです。
volatile int * sequence[5];	*sequence[5]	sequence は、volatile 整数データ・オブジェクトへの 5 つのポインターの配列です。
extern const volatile int clock;	clock	clock は、静的ストレージ期間および外部結合が指定された定数および volatile 整数です。

関連情報

- 62 ページの『型修飾子』
- 128 ページの『配列添え字演算子 [ ]』
- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』
- 181 ページの『関数宣言子』

型名

型名 は、オブジェクトを宣言しなくても指定する必要があるものとして、いくつかのコンテキスト内で必須です。例えば、明示的なキャスト式を書きこみしているとき、または sizeof 演算子を型へ適用しているときです。構文上、データ型の名前は、その型の関数の宣言またはその型のオブジェクトの宣言と同じですが、ID は使用しません。

型名を正確に読み書きするには、構文内に "虚数" ID を入れて、型名をより簡単なコンポーネントへ分割します。例えば、int は型指定子で、常に宣言内の ID の左側に表示されます。虚数 ID は、この簡単なケースでは必要ありません。ただし、int *[5] (int への 5 つのポインターの配列) もまた、型名です。型指定子 int * は常に ID の左側に表示され、配列添え字演算子は常に右側に表示されます。この場合、虚数 ID は型指定子を区別するときに役立ちます。

一般的な規則では、宣言内の ID は、常にサブスクリプト演算子および関数呼び出し演算子の左側、型指定子、型修飾子、または間接演算子の右側に表示されます。サブスクリプト演算子、関数呼び出し演算子、および間接演算子のみを、型名宣言内に表示することができます。これらは、標準の演算子優先順位に従ってバインドされます。この順位では、同じランクの優先順位を持つサブスクリプト演算子または関数演算子よりも、間接演算子の方が優先順位が低くなります。間接演算子のバインディングを制御するために、括弧を使用しても構いません。

型名内に型名を持つことができます。例えば、関数型において、パラメーター型構文が関数型名内でネストされます。同じ経験法則が、さらに繰り返し適用されます。

以下の構造体は、型の命名規則の適用について説明しています。

表 11. 型名

構文	内容
int *[5]	int への 5 つのポインターの配列
int (*)[5]	5 つの整数の配列へのポインター
int (*)[*]	指定されていない数の整数の可変長配列へのポインター

表 11. 型名 (続き)

構文	内容
int *()	int へのポインターを返す、パラメーターが指定されていない関数
int (*)(void)	int を返す、パラメーターのない関数
int (*const [])(unsigned int, ...)	int を返す、関数を指す指定されていない数の定数ポインターの配列。関数ごとに、unsigned int 型のパラメーター 1 つと、指定されていない数の他のパラメーターを取ります。

コンパイラーは、任意の関数指定子を、その関数を指すポインターに変換します。この振る舞いによって、構文呼び出しの構文が単純化します。

```
int foo(float);   /* foo is a function designator */
int (*p)(float);  /* p is a pointer to a function */
p=&foo;           /* legal, but redundant        */
p=foo;            /* legal because the compiler turns foo into a function pointer */
```

C++ C++ では、キーワード typename および class (これらは交換可能です) は、型の名前を示します。

関連情報

- 146 ページの『演算子優先順位と結合順序』
- 149 ページの『式および優先順位の例』
- 327 ページの『型名キーワード』
- 107 ページの『括弧で囲んだ式 ( )』

ポインター

ポインター 型変数は、データ・オブジェクトまたは関数のアドレスを保持します。ポインターは、1 つのデータ型のオブジェクトを参照できます。ビット・フィールドまたは参照は参照できません。

ポインターに共通な使用を次に示します。

- リンクされたリスト、ツリー、キューなどの動的データ構造にアクセスする。
- 配列のエレメント、構造体のメンバー、または C++ クラスのメンバーにアクセスする。
- 文字の配列に、ストリングとしてアクセスする。
- 変数のアドレスを関数に渡す。(C++ では、参照を使用してこれを行うこともできます。) そのアドレスを通して変数を参照することによって、関数はその変数の内容を変更できます。

なお、型修飾子 volatile および const を指定すると、ポインター宣言のセマンティクスに影響します。修飾子のいずれかが * の前にある場合には、宣言子は、型修飾オブジェクトへのポインターを記述します。修飾子のいずれかが * と ID の間にある場合には、宣言子は、型修飾ポインターを記述します。

次の表に、ポインター宣言の例を示します。

表 12. ポインター宣言

宣言	内容
`long *pcoat;`	pcoat は、long 型が指定されたオブジェクトを指すポインターです。
`extern short * const pvolt;`	pvolt は、short 型が指定されたオブジェクトを指す定数ポインターです。
`extern int volatile *pnut;`	pnut は、volatile 修飾子が指定された int オブジェクトを指すポインターです。
`float * volatile psoup;`	psoup は、float 型が指定されたオブジェクトを指す volatile ポインターです。
`enum bird *pfowl;`	pfowl は、bird 型の列挙オブジェクトを指すポインターです。
`char (*pvish)(void);`	pvish を、パラメーターを取らずに char を戻す関数を指すポインターです。

関連情報

- 62 ページの『型修飾子』
- 85 ページの『ポインターの初期化』
- 77 ページの『ポインターの互換性 (C のみ)』
- 99 ページの『ポインター型変換』
- 113 ページの『アドレス演算子 &』
- 113 ページの『間接演算子 *』
- 195 ページの『関数へのポインター』

ポインター演算

ポインターに関して行える算術演算は、限られています。これらの演算は、次のとおりです。

- 増分と減分
- 加算および減算
- 比較
- 代入

増分 (++) 演算子は、ポインターが参照するデータ・オブジェクトのサイズ分だけ、ポインターの値を増加させます。例えば、ポインターが配列の第 2 エレメントを参照している場合には、++ を使用すると、ポインターは配列の第 3 エレメントを参照するようになります。

減分 (--) 演算子は、ポインターが参照するデータ・オブジェクトのサイズ分だけ、ポインターの値を減少させます。例えば、ポインターが配列の第 2 エレメントを参照している場合には、-- を使用すると、ポインターは配列の第 1 エレメントを参照するようになります。

ポインターに整数を加算できますが、ポインターにはポインターを加算できません。

ポインター p が配列の 1 番目のエレメントを指している場合、次の式では、ポインターが同じ配列の 3 番目のエレメントを指すようにします。

```
p = p + 2;
```

同じ配列を指す 2 つのポインターがある場合は、一方のポインターからもう一方のポインターを減算することができます。この演算で、配列のエレメントの数が、ポインターが参照する 2 つのアドレスに分離されます。

2 つのポインターの比較は、==、!=、<、 >、<=、および >= の各演算子を使用して行うことができます。

ポインターの比較は、ポインターが同じ配列のエレメントを指すときにのみ定義されます。== および != 演算子を使用したポインターの比較は、ポインターが異なる配列のエレメントを指すときにも実行できます。

ポインターには、データ・オブジェクトのアドレス、互換性がある別のポインターの値、または ヌル・ポインターを割り当てることができます。

関連情報

- 111 ページの『増分演算子 ++』
- 78 ページの『配列』
- 111 ページの『減分演算子 --』
- 103 ページの『第 6 章 式と演算子』

型ベースの別名割り当て

コンパイラーは、**ALIAS(*ANSI)** オプションが有効な場合 (デフォルトで有効)、 C 標準および C++ 標準の型ベースの別名割り当て規則に従います。この規則では、ANSI 別名割り当て規則としても知られているように、ポインターは、同じ型または互換型のオブジェクトに対してのみ参照解除が可能であると定めています。[1] 非互換型にポインターをキャストしてから、それを参照解除するという、よく行われるコーディングの方法はこのルールに違反しています。(ただし、char ポインターは、この規則に対し例外であることに注意してください。)

---

1. C 標準は、オブジェクトの保管値には、次のいずれかの型の左辺値からしかアクセスできないと規定しています。

- 宣言されたオブジェクト型、
- 宣言されたオブジェクト型の修飾バージョン、
- 宣言されたオブジェクト型と対応する、符号付きまたは符号なしの型、
- 宣言されたオブジェクト型の修飾バージョンと対応する、符号付きまたは符号なしの型、
- メンバー (さらに、副集合体、または包含されている共用体のメンバーを含む) の中に前述の型のいずれかが含まれる構造体または共用体の型または
- 文字型

C++ 規格では、プログラムが以下のいずれか以外の型の左辺値によってオブジェクトの保管値にアクセスしようとした場合の振る舞いについては、規定されていません。

- 動的型のオブジェクト、
- 動的型のオブジェクトの cv 修飾バージョン、
- 動的型のオブジェクトと対応する、符号付きまたは符号なしの型、
- 動的型のオブジェクトの cv 修飾バージョンと対応する、符号付きまたは符号なしの型、
- メンバー (さらに、副集合体、または包含されている共用体のメンバーを含む) の中に前述の型のいずれかが含まれる構造体または共用体の型
- 動的型のオブジェクトの基底クラス型 (おそらく cv 修飾された) である型、
- char または符号なし char 型

コンパイラーは、型ベースの別名割り当て情報を使用して、生成されたコードに対する最適化を実行します。型ベースの別名割り当ての規則に違反すると、予期しない振る舞いの原因となる可能性があります。これを次の例で説明します。

```
   int *p;
   double d = 0.0;

   int *faa(double *g);         /* cast operator inside the function   */

   void foo(double f) {
       p = faa(&f);       /* turning &f into a int ptr     */
       f += 1.0;                /* compiler may discard this statement */
       printf("f=%x\n", *p);
    }

   int *faa(double *g) { return (int*) g; }   /* questionable cast;       */
                                              /* the function can be in   */
                                              /* another translation unit */

int main() {
         foo(d);
    }
```

前述の printf 文では、ANSI 別名割り当て規則によると、*p が double を参照解除することはできません。コンパイラーは、f += 1.0; の結果を今後使用しないと判断します。そのため、最適化プログラムは、このステートメントを生成コードから破棄する可能性があります。最適化を使用可能にして上の例をコンパイルした場合には、printf ステートメントは 0 (ゼロ) を出力する可能性があります。

関連情報

- 136 ページの『reinterpret_cast 演算子 (C++ のみ)』
- 「*ILE C/C++ コンパイラー参照*」の **ALIAS(*ANSI)**

ポインターの互換性 (C のみ)

同じ型修飾子を指定された 2 つのポインター型は、それらが互換型のオブジェクトを指している場合、互換性があります。 2 つの互換ポインター型の複合型は、複合型と類似した修飾ポインターです。

次の例では、代入演算用の互換性がある宣言を示します。

```
float subtotal;
float * sub_ptr;
/* ... */
sub_ptr = &subtotal;
printf("The subtotal is %f\n", *sub_ptr);
```

次の例では、代入演算用の互換性がない宣言を示します。

```
double league;
int * minor;
/* ... */
minor = &league;     /* error */
```

配列

配列 は、同じデータ型のオブジェクトのコレクションであり、メモリー内に連続して割り振られます。配列内の個々のオブジェクト *elements* は、配列内のそれらの位置を基にしてアクセスされます。サブスクリプト演算子 ([]) は、配列エレメントへの指標を作成する方法を提供します。このアクセス形式は、*指標付け* または *サブスクリプト付け* と呼ばれます。配列では、各エレメントで実行されたステートメントを、配列内の各エレメントを通って繰り返されるループへ入れることができるために、反復タスクのコーディングが容易になります。

C および C++ 言語には、個々のエレメントの読み取りおよび書き込みができる、配列型用の限定された組み込みサポートがあります。 ある配列を別の配列に代入したり、 2 つの配列の等価性を比較したり、自己認識サイズを返したりする操作は、いずれの言語でもサポートされていません。

配列の型は、いわゆる *配列型派生* 内で、エレメントの型から派生します。配列オブジェクトの型が不完全な場合、配列型も不完全であると見なされます。配列エレメントは、`void` 型にも関数型にもすることができません。ただし、関数へのポインター配列は許可されます。 C++ 配列エレメントは参照型にも抽象クラス型にもすることができません。

配列宣言子には、ID と、その後に続く、オプションの*サブスクリプト宣言子* が含まれています。アスタリスク (*) が前に付く ID は、ポインターの配列です。

配列添え字宣言子の構文

constant_expression は、配列サイズを示す定数整数式です。これは正でなければいけません。

サブスクリプト宣言子は、配列の次元の数と各次元のエレメントの数を記述します。大括弧で囲まれた式、またはサブスクリプトは、別の次元を表します。これらは、定数式でなければなりません。

次の例では、`char` 型が指定された 4 つのエレメントを含む 1 次元配列を定義します。

```
char
list[4];
```

各次元の 1 番目のサブスクリプトは、`0` です。配列 `list` には以下のエレメントが含まれます。

```
list[0]
list[1]
list[2]
list[3]
```

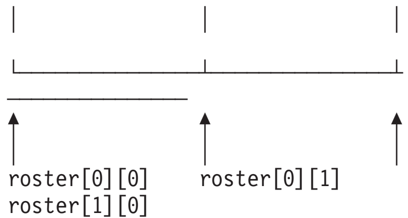
次の例では、`int` 型の 6 つのエレメントを含む 2 次配列を定義します。

```
int
roster[3][2];
```

多次元配列は、行方向優先順序で保管されます。エレメントが、保管場所の昇順で参照される場合、一番後のサブスクリプトが一番先に変わります。例えば、配列 `roster` のエレメントは、以下の順序で保管されます。

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

ストレージ内では、roster のエレメントは、以下のように保管されます。



以下の場合には、最初の (最初のみ) サブスクリプトの大括弧のセットを空にしたままにすることができます。

- 初期化を含む配列定義
- `extern` 宣言
- パラメーター宣言

最初のサブスクリプトの大括弧のセットを空にした配列定義では、初期化指定子が最初の次元のエレメントの数を決めます。1 次元配列では、初期化されたエレメントの数が、エレメントの合計数になります。多次元配列は、初期化指定子をサブスクリプト宣言子と比較し、第 1 次元のエレメントの数を決めます。

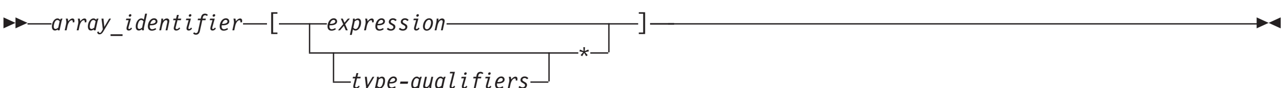
関連情報

- 128 ページの『配列添え字演算子 [ ]』
- 86 ページの『配列の初期化』

可変長配列 (C++ のみ)

可変長配列は、C99 のフィーチャーであり、長さが実行時に決定される自動ストレージ期間を持つ配列です。

可変長配列宣言子の構文



配列のサイズが式ではなく * で示されている場合には、可変長配列は、サイズが指定されていないものとしてみなされます。このような配列は完全型とみなされますが、関数プロトタイプ・スコープの宣言でのみ使用できます。

可変長配列および可変長配列を指すポインターは、*可変的に変更される型*と見なされます。可変的に変更される型の宣言は、ブロック・スコープまたは関数プロトタイプ・スコープで行う必要があります。 `extern` ストレージ・クラス指定子で宣言された配列オブジェクトは、可変長配列型を持つことができません。 `static` ストレージ・クラス指定子で宣言された配列オブジェクトは、可変長配列を指すポインターになれますが、実際の可変長配列にはなれません。 可変長配列は、初期化できません。

注: C++ アプリケーションでは、可変長配列によって使用するように割り振られたストレージは、可変長配列が使用されている関数の実行が完了するまで解放されません。

可変長配列は、sizeof 式のオペランドにすることができます。この場合、可変長配列の各インスタンスのサイズが存続時間中に変更されることがなくても、オペランドは実行時に評価され、このサイズは整数定数でも定数式でもありません。

可変長配列は、typedef ステートメントで使用できます。 typedef の名前は、ブロック・スコープのみを持ちます。配列長は、typedef の名前が使用されるたびに固定するのではなく、定義されるときに固定します。

関数仮パラメーターは、可変長配列にすることができます。関数定義で、必要なサイズ式を指定する必要があります。コンパイラーは、関数の入り口で可変パラメーターのサイズ式を評価します。以下のように可変長配列をパラメーターとして宣言した関数の場合、

```
void f(int x, int a[][x]);
```

可変長配列引数のサイズは、関数定義のサイズに一致する必要があります。

C++ の拡張では、可変長配列型への参照のサポートは組み込まれておらず、関数仮パラメーターを、可変長配列型への参照にすることもできません。

関連情報

- 51 ページの『柔軟な配列メンバー 』

配列の互換性

同じように修飾される 2 つの配列型は、それらのエレメントの型に互換性があれば、互換性があります。例えば、次のような場合です。

```
char ex1[25];
const char ex2[25];
```

これらには、互換性はありません。

2 つの互換配列型から成る複合型は、複合エレメント型を持つ配列です。元の型のサイズが両方とも既知である場合は、両方のサイズが同じでなければなりません。元の配列型のサイズがどちらか一方だけ既知である場合は、その既知のサイズが複合型のサイズです。次に例を示します。

```
char ex3[];
char ex4[42];
```

この場合、ex3 および ex4 の複合型は char[42] です。元の型のいずれかが可変長配列である場合、複合型はその型です。

関連情報

- 8 ページの『外部結合』

参照 (C++ のみ)

参照 は、オブジェクトの別名または代替名です。参照に適用されるすべての演算は、参照が参照するオブジェクトで動作します。参照のアドレスは、別名が付けられたオブジェクトのアドレスです。

参照型は、型指定子の後に 参照修飾子 & を入れることによって定義できます。関数仮パラメーターを除く参照はすべて、定義するときに初期化する必要があります。参照は一度定義すると再割り当てできません。これは、参照はターゲットの別名であるためです。参照を再割り当てしようとすると、ターゲットへ新しい値が割り当てられます。

関数の引数は、値によって渡されるため、関数呼び出しは、引数の実際の値を変更しません。関数が、引数の実際の値を修正する必要がある場合、または複数の値を戻す必要がある場合は、引数は*参照によって受け渡し* (*値による受け渡し* に対比) する必要があります。参照による引数の受け渡しは、参照またはポインターのいずれかを使用して行うことができます。 C と異なり、C++ は、参照によって引数を渡したい場合には、ポインターの使用を強制しません。参照を使用する構文は、ポインターを使用するときよりもいくらか簡単です。参照によってオブジェクトを渡すと、関数は、関数のスコープ内でオブジェクトのコピーを作成しないで、参照されているオブジェクトを作成することができます。オブジェクト全体ではなく、実際の元オブジェクトのアドレスのみがスタックに置かれます。

次に例を示します。

```
int f(int&);
int main()
{
      extern int i;
      f(i);
}
```

関数呼び出し f(i) からは、引数が参照によって渡されていることはわかりません。

NULL への参照は認められていません。

関連情報

- 88 ページの『参照の初期化 (C++ のみ)』
- 74 ページの『ポインター』
- 100 ページの『参照変換 (C++ のみ)』
- 113 ページの『アドレス演算子 &』
- 190 ページの『参照による受け渡し』

初期化指定子

初期化指定子 は、データ・オブジェクトの初期値を指定する、データ宣言のオプションの部分です。特定の宣言に使用できる初期化指定子は、初期化されるオブジェクトの型およびストレージ・クラスによって異なります。

初期化指定子は、= 記号と、その後に続く、初期式 *expression*、またはコンマで分離された初期式の中括弧で囲まれたリストで構成されます。個々の式は、コンマで分離される必要があります。式のグループは中括弧で囲み、コンマで分離することができます。文字ストリングの初期化指定子がストリング・リテラルの場合は、中括弧 ({ }) はオプションです。初期化指定子の数は、初期化されるエレメントの数よりも多くてはいけません。初期式は、データ・オブジェクトの最初の値に数値化されます。

算術型またはポインター型に値を代入するには、単純初期化指定子 = *expression* を使用します。例えば、次のデータ定義は、初期化指定子 = 3 を使用して、group の初期値を 3 に設定します。

```
int group = 3;
```

文字リテラル (1 文字からなる) を使用して、または整数に評価する式を使用して、文字型の変数を初期化します。

C++ ネーム・スペース・スコープで定数以外の式を指定して、変数を初期化できます。 C 非定数式を使用して、グローバル・スコープで変数を初期化することはできません。

『初期化およびストレージ・クラス』で、変数のストレージ・クラスに応じた初期化の規則について説明しています。

以下のセクションでは、派生型の初期化について説明しています。
- 83 ページの『構造体および共用体の初期化』
- 85 ページの『ポインターの初期化』
- 86 ページの『配列の初期化』
- 88 ページの『参照の初期化 (C++ のみ)』

関連情報
- 220 ページの『クラス・オブジェクトの使用』

初期化およびストレージ・クラス

自動変数の初期化

auto 変数 (関数仮パラメーターを除く) は、初期化できます。自動オブジェクトを明示的に初期化しない場合は、その値を確定することができません。初期値を提供する場合は、初期値を表す式を C または C++ の有効な式にすることができます。オブジェクトの定義を含むプログラム・ブロックに入るたびに、オブジェクトはその初期値にセットされます。

goto ステートメントを使用し、ブロックの中央にジャンプする場合は、そのブロック内の自動変数は初期化されないことに留意してください。

関連情報
- 40 ページの『自動ストレージ・クラス指定子』

静的変数の初期化

静的オブジェクトの初期化を、定数式、またはすでに extern または static と宣言されているオブジェクトのアドレスに変換する式 (多くは定数式によって修正される) によって行えます。static (または external) 変数は、明示的に初期化しなければ、ポインターでない限り、該当する型のゼロの値になります。ポインターの場合は、NULL に初期化されます。

ブロック内の static 変数は、プログラムの実行前に一度だけ初期化されます。一方、初期化指定子を持つ auto 変数は発生するごとに初期化されます。

C++ クラス型の静的オブジェクトは、それを初期化しない場合は、デフォルトのコンストラクターを使用します。初期化されない自動変数およびレジスター変数は、未定義の値を持つことになります。

関連情報
- 40 ページの『static ストレージ・クラス指定子』

外部変数の初期化

extern ストレージ・クラス指定子を持つオブジェクトは、C のグローバル・スコープまたは C++ のネーム・スペースで初期化できます。 extern オブジェクトの初期化指定子は、以下のうちのいずれかにする必要があります。

- 定義の一部として指定し、初期値を定数式によって記述する。
- 静的ストレージ期間が指定された、すでに宣言済みのオブジェクトのアドレスに変換する。このオブジェクトは、ポインター演算によって変更することができます。 (言い換えると、オブジェクトは、整数定数式の加算または減算によって変更することができます。)

extern 変数を明示的に初期化しない場合は、その初期値は、該当する型のゼロになります。 extern オブジェクトの初期化は、プログラムが実行を開始するときまでに完了しています。

関連情報

- 41 ページの『extern ストレージ・クラス指定子』

レジスター変数の初期化

関数仮パラメーターを除く register オブジェクトを初期化できます。自動オブジェクトを初期化しない場合は、その値は確定できません。初期値を提供する場合は、初期値を表す式を C または C++ の有効な式にすることができます。オブジェクトの定義を含むプログラム・ブロックに入るたびに、オブジェクトはその初期値にセットされます。

関連情報

- 43 ページの『register ストレージ・クラス指定子』

構造体および共用体の初期化

構造体の初期化指定子は、中括弧で囲んだ、コンマ区切りの値のリストであり、共用体の初期化指定子は、中括弧で囲んだ単一の値です。初期化指定子には、等号 (=) が前に付いています。

C++ では、共用体型または構造体型の自動メンバー変数の初期化指定子を定数式または非定数式にすることができます。

C++ 共用体型または構造体型の静的メンバー変数の初期化指定子は、定数式またはストリング・リテラルにする必要があります。詳しくは、 240 ページの『静的データ・メンバー』を参照してください。

構造体および共用体の初期化指定子は、以下のように指定できます。

- C89 スタイルの初期化指定子を使用する。この場合、構造体メンバーは、宣言した順序で初期化する必要があり、共用体の場合は第 1 メンバーのみを初期化できます。

次の例では、C89 スタイルの初期化を使用して、共用体変数 people の第 1 共用体メンバーである birthday を初期化する方法を示します。

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = {"23/07/57"};
```

次の定義では、完全に初期化される構造体を示します。

```
struct address {
                 int street_no;
                 char *street_name;
```

```
                char *city;
                char *prov;
                char *postal_code;
              };
static struct address perm_address =
              { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

perm_address の値は、次のとおりです。

メンバー	値
perm_address.street_no	3
perm_address.street_name	ストリング "Savona Dr." のアドレス
perm_address.city	ストリング "Dundas" のアドレス
perm_address.prov	ストリング "Ontario" のアドレス
perm_address.postal_code	ストリング "L4B 2A1" のアドレス

名前のない構造体または共用体の各メンバーは、初期化には関与せず、初期化が終わると不定値を持ちます。そのため、以下の例では、ビット・フィールドは初期化されず、初期化指定子 3 はメンバー b に適用されます。

```
struct {
     int a;
     int :10;
     int b;
     } w = { 2, 3 };
```

構造体または共用体のすべてのメンバーを初期化する必要はありません。未初期化の構造体メンバーの初期値は、構造体または共用体変数に関連付けられたストレージ・クラスによって異なります。 static として宣言された構造体では、未初期化のメンバーはすべて、該当する型のゼロに暗黙的に初期化されます。自動ストレージを持つ構造体のメンバーの場合は、デフォルトの初期化はありません。静的ストレージを持つ共用体のデフォルトの初期化指定子が、第 1 コンポーネントのデフォルトになります。自動ストレージを持つ共用体の場合は、デフォルトの初期化はありません。

次の定義では、一部のみが初期化される構造体を示します。

```
struct address {
                int street_no;
                char *street_name;
                char *city;
                char *prov;
                char *postal_code;
              };
struct address temp_address =
              { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

temp_address の値は、次のとおりです。

メンバー	値
temp_address.street_no	44
temp_address.street_name	ストリング "Knyvet Ave." のアドレス
temp_address.city	ストリング "Hamilton" のアドレス
temp_address.prov	ストリング "Ontario" のアドレス
temp_address.postal_code	temp_address 変数のストレージ・クラスによって異なる。static の場合は、値は NULL。

関連情報

- 54 ページの『構造体および共用体変数宣言』
- 280 ページの『コンストラクターによる明示的初期化』
- 119 ページの『代入演算子』

列挙型の初期化

列挙変数の初期化指定子には、= 記号と、その後に続く式 *enumeration_constant* が含まれます。

C++ C++ では、初期化指定子の型は、関連付けられた列挙型と同じである必要があります。

次の例の 1 行目は、列挙型 grain を宣言します。 2 行目は、変数 g_food を定義し、g_food に初期値 barley (2) を指定します。

```
enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;
```

関連情報

- 59 ページの『列挙型変数宣言』

ポインターの初期化

初期化指定子は、= (等号) と、その後に続く、ポインターに入れられるアドレスを表す式によって表されます。次の例では、変数 time と speed には double 型、amount には double を指す型ポインターが指定されるように定義します。この例では、ポインター amount が total を指すように初期化が行われています。

```
double total, speed, *amount = &total;
```

コンパイラーは、サブスクリプトがない配列名を、配列の 1 番目のエレメントを指すポインターに変換します。配列の名前を指定することによって、配列の 1 番目のエレメントのアドレスをポインターに割り当てることができます。次の 2 つの定義のセットは、同じです。定義は両方とも、ポインター student を定義し、student を section の 1 番目のエレメントのアドレスに初期化します。

```
int section[80];
int *student = section;
```

は、以下と同等です。

```
int section[80];
int *student = &section[0];
```

初期化指定子のストリング定数を指定することによって、ストリング定数の 1 番目の文字のアドレスをポインターに割り当てることができます。次の例では、ポインター変数 string とストリング定数 "abcd" を定義します。ポインター string は、ストリング "abcd" の文字 a を指すように初期化されます。

```
char *string = "abcd";
```

次の例では、weekdays を、ストリング定数を指すポインターの配列として定義します。各エレメントは、異なるストリングを指します。例えば、ポインター weekdays[2] は、ストリング "Tuesday" を指します。

```
static char *weekdays[ ] =
      {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
      };
```

ポインターは、0 に数値化される整数定数式 (例えば、char * a=0;) を使用して、ヌルに初期化することもできます。このようなポインターは、ヌル・ポインター です。このヌル・ポインターは、オブジェクトを指しません。

関連情報

- 74 ページの『ポインター』

配列の初期化

配列の初期化指定子は、中括弧 ({ }) で囲まれた定数式の、コンマで分離されたリストです。初期化指定子には、等号 (=) が前に付いています。配列内のすべてのエレメントを初期化する必要はありません。配列が部分的に初期化されている場合は、初期化されていないエレメントの値は、該当の型の値 0 となります。静的ストレージ期間を持つ配列のエレメントについても、同じことが言えます。(静的ストレージ期間を持つのは、static キーワードを指定して宣言されているすべてのファイル・スコープ変数および関数スコープ変数です。)

配列の初期化指定子を指定するには、次の 2 つの方法があります。

- C89 スタイルの初期化指定子を使用する。この場合、配列エレメントは、添え字順に初期化する必要があります。

以下の定義では、C89 スタイルの初期化指定子を使用して、完全に初期化された 1 次元配列を示します。

```
static int number[3] = { 5, 7, 2 };
```

配列 number には、次の値が含まれます。number[0] は 5、number[1] は 7、number[2] は 2 です。サブスクリプト宣言子内の式がエレメント数 (この場合は 3) を定義している場合、配列内のエレメント数よりも多くの初期化指定子を持つことはできません。

次の定義では、部分的に初期化される 1 次元配列を示します。

```
static int number1[3] = { 5, 7 };
```

number1[0] および number1[1] の値は以前の定義と同じですが、number1[2] は 0 です。

サブスクリプト宣言子の式でエレメントの数を定義する代わりに、次の 1 次元配列定義では、指定された各初期化指定子に対して 1 つのエレメントを定義します。

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

サイズが指定されていなくて、初期化エレメントが 5 つあるため、コンパイラーは item に次の 5 つの初期化されたエレメントを指定します。

文字配列の初期化

次のものを指定して、1 次元の文字配列を初期化できます。

- 中括弧で囲まれ、コンマで分離された定数のリスト。各定数は、文字に含めることができます。
- ストリング定数 (定数を囲む中括弧はオプション)

ストリング定数を初期化すると、空いている場所がある場合または配列の次元が指定されていない場合は、ヌル文字 (¥0) をストリングの終わりに入れます。

以下の定義は、文字配列の初期化を示します。

```
    static char name1[ ] = { 'J', 'a', 'n' };
 static char name2[ ] = { "Jan" };
 static char name3[4] = "Jan";
```

以下の定義では、次のエレメントを作成します。

エレメント	値	エレメント	値	エレメント	値
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	\0	name3[3]	\0

次の定義では、ヌル文字はなくなります。

```
static char name3[3]="Jan";
```

C++ 文字の配列をストリングで初期化する場合、ストリング内の文字数 (終端の「¥0」を含む) は、配列の中のエレメント数を超えてはなりません。

多次元配列の初期化

次の方法を使用して、多次元配列を初期化できます。

- 初期化するすべてのエレメントの値を、コンパイラーが値を割り当てる順序でリストする。コンパイラーは、最後の次元のサブスクリプトを最も速く増やして、値を割り当てます。この形式の多次元配列の初期化は、1 次元配列の初期化と似ています。次の定義では、配列 month_days を完全に初期化します。

```
    static month_days[2][12] =
  {
   31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
   31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  };
```

- 中括弧を使用して、初期化するエレメントの値をグループ化します。各エレメントかエレメントのネスト・レベルを中括弧で囲むことができます。次の定義には、第 1 次元の 2 つのエレメントが含まれます (これらのエレメントは、行と見なすことができます)。初期化には、これらの 2 つの各エレメントを囲む中括弧が含まれます。

```
static int month_days[2][12] =
{
 { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
 { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
```

- ネストされた中括弧を使用して、次元および次元のエレメントを選択的に初期化する。次の例では、配列 grid の最初の 8 つのエレメントのみが、明示的に初期化されます。明示的に初期化されない残りの 4 つのエレメントは、自動的にゼロに初期化されます。

```
static short grid[3] [4] = {8, 6, 4, 1, 9, 3, 1, 1};
```

grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	8	grid[1] [2]	1
grid[0] [1]	6	grid[1] [3]	1
grid[0] [2]	4	grid[2] [0]	0

エレメント	値	エレメント	値
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	9	grid[2] [2]	0
grid[1] [1]	3	grid[2] [3]	0

C のみ

- *指定* 初期化指定子を使用する。以下の例では、指定初期化指定子を使用して、配列の最後の 4 つのエレメントのみを明示的に初期化します。明示的に初期化されない最初の 8 つのエレメントは、自動的にゼロに初期化されます。

```
static short grid[3] [4] = { [2][0] = 8, [2][1] = 6,
                             [2][2] = 4, [2][3] = 1 };
```

grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	0	grid[1] [2]	0
grid[0] [1]	0	grid[1] [3]	0
grid[0] [2]	0	grid[2] [0]	8
grid[0] [3]	0	grid[2] [1]	6
grid[1] [0]	0	grid[2] [2]	4
grid[1] [1]	0	grid[2] [3]	1

C のみ の終り

関連情報

- 78 ページの『配列』

参照の初期化 (C++ のみ)

参照を初期化するために使用するオブジェクトは、参照と同じ型にする必要があります。同じ型でなければ、参照型に型変換できる型でなければなりません。型変換が必要なオブジェクトを使用して定数への参照を初期化する場合は、一時オブジェクトを作成します。次の例では、型 float の一時オブジェクトを作成します。

```
int i;
const float& f = i; // reference to a constant float
```

オブジェクトを使用して参照を初期化する場合、その参照をそのオブジェクトに*バインド* します。

型変換が必要なオブジェクトを使用して、定数以外の参照を初期化しようとすると、エラーになります。

参照は初期化されると、別のオブジェクトを参照するように変更することはできません。次に例を示します。

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;          // valid
int &RefOne = num2;          // error, two definitions of RefOne
RefOne = num2;                       // assign num2 to num1
int &RefTwo;                 // error, uninitialized reference
int &RefTwo = num2;          // valid
```

参照の初期化は、参照への代入と同じではないことに注意してください。初期化は、実際の参照に対する別名であるオブジェクトを使用して参照を初期化することによって、実際の参照で動作します。代入は、参照されるオブジェクトでの参照を通して動作します。

次の場合には、初期化指定子を使用せずに参照を宣言できます。

- パラメーター宣言で使用する場合
- 関数呼び出しの戻りの型の宣言内
- クラス・メンバーのクラス宣言での宣言内
- extern 指定子を明示的に使用する場合

以下のものへの参照は使用できません。

- 他の参照
- ビット・フィールド
- 参照の配列
- 参照を指すポインター

直接バインディング

型 T の参照 r が、型 U の式 e によって初期化されると想定します。

以下のステートメントが真であれば、参照 r は e に*直接バインド*されます。

- 式 e は、左辺値である。
- T は U と同じ型であるか、または T は、U の基底クラスである。
- T は、U と同じ、あるいはより多くの const または volatile 修飾子を持っている。

ステートメントの以前のリストが真であるように e を暗黙的に型に変換できる場合、参照 r も e に直接バインドされます。

関連情報

- 80 ページの『参照 (C++ のみ)』
- 190 ページの『参照による受け渡し』

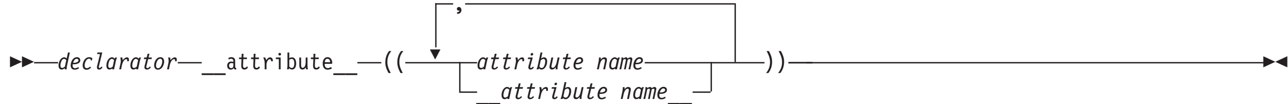
変数属性

IBM 拡張

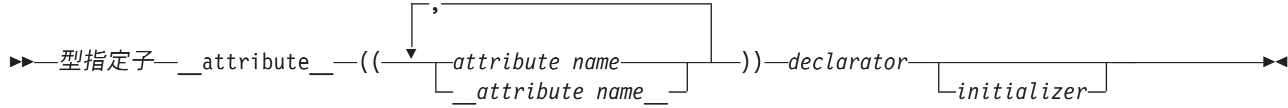
変数属性は、GNU C/C++ コンパイラーで開発されたプログラムのコンパイルを容易にするために提供されている言語拡張です。この言語フィーチャーを使用することで、名前付き属性を使用して、データ・オブジェクトの特殊プロパティーを指定できます。変数 属性は、単純変数、集合体、および集合体のメンバー変数の宣言に適用されます。

変数属性は、キーワード __attribute__ に続いて、属性名、および属性名に必要な追加の引数がある場合にはその引数を設定して、指定します。変数 __attribute__ の指定は、変数の宣言に組み込まれ、宣言子の前または後に配置できます。バリエーションはありますが、構文は通常、以下のいずれかの形式を取ります。

変数属性の構文: 宣言子の後



変数属性の構文: 宣言子の前



attribute name は、2 つの下線文字を前後に付けて指定しても付けないで指定しても構いません。ただし、2 つの下線文字を使用すると、同じ名前のマクロとの名前の競合の可能性が小さくなります。サポートされない属性名については、IBM i コンパイラーは診断メッセージを出して、その属性の指定を無視します。同じ属性指定で複数の属性名を指定できます。

単一の宣言行の宣言のコンマ区切りリストでは、変数属性がすべての宣言子より前に存在する場合には、その属性は宣言内のすべての宣言子に適用されます。属性が宣言子の後に存在する場合には、直前の宣言子にのみ適用されます。次に例を示します。

```
struct A  {

 int b __attribute__((aligned));         /* typical placement of variable */
                                                        /* attribute */

 int __attribute__((aligned))__ c = 10;   /* variable attribute can also be */
                                           /* placed here */

 int d, e, f __attribute__((aligned));    /* attribute applies to f only */

 int g __attribute__((aligned)), h, i;    /* attribute applies to g only */

 int __attribute__((aligned)) j, k, l;    /* attribute applies to j, k, and l */

 };
```

次の変数属性がサポートされます。

- aligned 変数属性
- packed 変数属性
- mode 変数属性
- weak 変数属性

関連情報

- 67 ページの『型属性』
- 185 ページの『関数属性』

aligned 変数属性

aligned 変数属性を使用することで、以下のいずれかについて、デフォルトの位置合わせモードをオーバーライドして、バイト数で表した最小位置合わせ値を指定できます。

- 非集合体変数
- 集合体変数 (構造体、クラス、または共用体など)
- 選択したメンバー変数

この属性は通常、指定した変数の位置合わせを増加させるために使用します。

aligned 変数属性の構文

```
►►──__attribute__──((─┬─aligned─────┬─┬─────────────────────────┬─))─────────────────────►◄
                      └─__aligned__─┘ └─(──alignment_factor──)─┘
```

alignment_factor はバイト数であり、2 の正の累乗に評価される定数式として指定されます。最大で 1048576 バイトの値を指定できます。位置合わせ係数 (およびそれを囲む括弧) を省略した場合には、コンパイラーは自動的に 16 バイトを使用します。 最大値より大きな位置合わせ係数を指定した場合には、属性の指定は無視され、コンパイラーは単に、有効なデフォルト位置合わせを使用します。

aligned 属性をビット・フィールド構造体メンバー変数に適用した場合には、属性の指定は、ビット・フィールド・コンテナー に適用されます。コンテナーのデフォルト位置合わせが位置合わせ係数より大きい場合には、デフォルト位置合わせが使用されます。

以下の例では、構造体 first_address および second_address が 16 バイトの位置合わせに設定されます。

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               } first_address __attribute__((__aligned__(16))) ;

struct address second_address __attribute__((__aligned__(16))) ;
```

以下の例では、メンバー first_address.prov および first_address.postal_code のみが、16 バイトの位置合わせに設定されます。

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov __attribute__((__aligned__(16))) ;
                 char *postal_code __attribute__((__aligned__(16))) ;
               } first_address  ;
```

関連情報

- 63 ページの『__align 型修飾子』
- 「*ILE C/C++ Programmer's Guide*」の『Aligning data』
- 115 ページの『__alignof__ 演算子』
- 68 ページの『aligned 型属性』

packed 変数属性

変数属性 packed を使用することで、デフォルトの位置合わせモードをオーバーライドして、集合体のすべてのメンバー、または集合体の選択したメンバーの位置合わせを、可能な最小の位置合わせ (メンバーの場合は 1 バイト、ビット・フィールド・メンバーの場合は 1 ビット) に減らすことができます。

packed 変数属性の構文

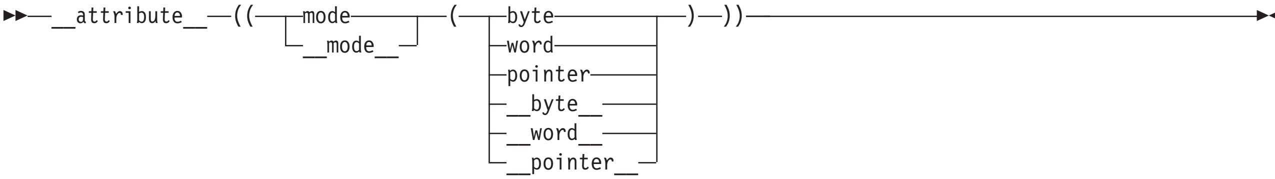
関連情報

- 63 ページの『__align 型修飾子』
- 「*ILE C/C++ Programmer's Guide*」の『Aligning data』
- 115 ページの『__alignof__ 演算子』
- 69 ページの『packed 型属性』

mode 変数属性

変数属性 mode を使用することで、変数宣言の型指定子をオーバーライドして、特定の整数型のサイズを指定できます。

mode 変数属性の構文



mode の有効な引数は、特定の幅を示す以下の型指定子のいずれかです。

- byte は、1 バイトの整数型です。
- word は、4 バイトの整数型です。
- pointer は、8 バイトの整数型です。

weak 変数属性

weak 変数属性があると、変数宣言の結果によるシンボルは、オブジェクト・ファイルの中に、グローバル・シンボルとしてではなく、弱いシンボルとして現れます。この言語フィーチャーを使用すると、ライブラリー関数を作成するプログラマーは、名前の重複エラーを生じることなく、ユーザー・コードの変数定義によってライブラリー宣言をオーバーライドできます。

weak 変数属性の構文

```
►►─__attribute__─((──┬─weak─────┬─))──────────────────────────────►◄
                     └─__weak__─┘
```

関連情報

- 「*ILE C/C++ コンパイラー参照*」の **#pragma weak**

- 187 ページの『weak 関数属性』

IBM 拡張 の終り

第 5 章 型変換

与えられた型の式は、以下の状況では*暗黙的に変換* されます。

- この式が算術演算または論理演算のオペランドとして使用される。
- 式が、`if` ステートメントまたは繰り返しステートメント (`for` ループなど) の中で、条件として使用される。式はブール値に変換されます。
- この式が `switch` ステートメントの中で使用される。式は整数型に変換されます。
- 式は初期化のために使用される。これには、次の場合が含まれます。
  – 代入される値とは型が異なる左辺値に対して、代入が行われる。
  – 関数に、パラメーターとは異なる型を持つ引数値が与えられる。
  – 関数の `return` ステートメントに、その関数用に定義されている戻りの型とは異なる値が指定される。

133 ページの『キャスト式』に説明されているように、*cast* を使用して*明示的な*型変換を行うことができます。以下のセクションでは、暗黙的変換または明示的変換のいずれかによって許可される変換と型プロモーションを決定する規則について説明します。

- 『算術変換とプロモーション』
- 98 ページの『左辺値から右辺値への変換』
- 99 ページの『ポインター型変換』
- 100 ページの『参照変換 (C++ のみ)』
- 101 ページの『修飾変換 (C++ のみ)』
- 101 ページの『関数引数変換』

関連情報

- 289 ページの『ユーザー定義の型変換』
- 290 ページの『変換コンストラクター』
- 292 ページの『変換関数』
- 156 ページの『switch ステートメント』
- 154 ページの『if ステートメント』
- 166 ページの『return ステートメント』

算術変換とプロモーション

以下のセクションでは、算術型の標準変換に関する規則について説明します。

- 96 ページの『整数変換』
- 97 ページの『浮動小数点の型変換』
- 96 ページの『ブール変換』

式の 2 つのオペランドの型がそれぞれ異なる場合は、 97 ページの『整数および浮動小数点拡張』に説明されているように、それらのオペランドは*通常の算術変換*の規則に従います。

© Copyright IBM Corp. 1998, 2010

整数変換

符号なし整数から符号なし整数へ、または符号付き整数から符号付き整数へ

型が同じである場合、変更はありません。型のサイズが異なり、かつ値を新しい型によって表せる場合、値は変更されません。値を新しい型によって表せない場合、切り捨てと符号シフトが行われます。

符号付き整数から符号なし整数へ

結果の変換元整数に適合する最小の符号なし整数型です。値を新しい型によって表せない場合は、切り捨てと符号シフトが行われます。

符号なし整数から符号付き整数へ

符号付き型が元の値を保持できるほど十分大きい場合は、変更は行われません。値を新しい型によって表せる場合は、値は変更されません。値を新しい型によって表せない場合は、切り捨てと符号シフトが行われます。

符号付き文字型および符号なし文字型から整数へ

元の値を int によって表せる場合は、元の値が int として表されます。元の値を int によって表せない場合は、元の値が unsigned int にプロモートされます。

ワイド文字型 wchar_t から整数へ

元の値を int によって表せる場合は、元の値が int として表されます。元の値を int によって表せない場合は、元の値がそれを保持できる最小の型 (unsigned int、long、または unsigned long) にプロモートされます。

符号付き整数ビット・フィールドおよび符号なし整数ビット・フィールドから整数へ

元の値を int によって表せる場合は、元の値が int として表されます。元の値を int によって表せない場合は、元の値が unsigned int にプロモートされます。

列挙型から整数へ

元の値を int によって表せる場合は、元の値が int として表されます。元の値を int によって表せない場合は、元の値がそれを保持できる最小の型 (unsigned int、long、または unsigned long) にプロモートされます。列挙型は整数型に変換できますが、整数型は列挙型に変換できないので、注意してください。

ブール変換

ブール値から整数へ

C ブール値が 0 であると、結果は値が 0 の int となります。ブール値が 1 であると、結果は値が 1 の int となります。

C++ ブール値が false であると、結果は値が 0 の int となります。ブール値が true であると、結果は値が 1 の int となります。

スカラーからブール値へ

C スカラー値が 0 に等しいと、ブール値は 0 です。その他の場合は、ブール値は 1 です。

C++ ゼロ、ヌル・ポインター、またはヌル・メンバー・ポインター値は、偽に変換されます。他の値はすべて 真に変換されます。

浮動小数点の型変換

実数浮動小数点型同士の変換 (2 進数から 2 進数へ、10 進数から 10 進数へ、および 10 進数から 2 進数へ) に関する標準規則は次のとおりです。

変換元の値を新しい型で正確に表せる場合は、値は変更されません。変換元の値が表せる値の範囲内にあるが、正確には表せない場合は、有効な現行のコンパイル時丸めモードまたは実行時丸めモードに従って結果が丸められます。変換元の値が表せる値の範囲外にある場合は、結果は丸めモードによって決まります。

整数から浮動小数点 (2 進または 10 進) へ

変換元の値を新しい型で正確に表せる場合は、値は変更されません。変換元の値が表せる値の範囲内にあるが、正確には表せない場合は、結果は正しく丸められます。変換元の値が表せる値の範囲外にある場合は、結果は静止 NaN です。

浮動小数点 (2 値または 10 進) から整数へ

小数部分は破棄されます (つまり、値はゼロに切り捨てられます)。整数部分の値を整数型によって表せない場合は、結果は次のいずれかです。

- 整数型が符号なしの場合は、浮動小数点数が正であると、結果は表現可能な最大数です。浮動小数点数が正でないと、結果は 0 です。
- 整数型が符号付きの場合は、浮動小数点数の符号に従って結果は負または正の表現可能な最大の数となります。

整数および浮動小数点拡張

ある種のタイプの式でさまざまな算術型がオペランドとして使用されているときは、*通常の算術変換*として知られている標準変換が適用されます。これらの変換は算術型のランクに従って適用されます。つまり、低いランクの型を持つオペランドが高いランクを持つオペランドの型に変換されます。これは整数プロモーションまたは浮動小数点プロモーションとして知られているものです。

例えば、2 つの異なる整数型の値が加算されるときは、まず両方の値が同じ型に変換されます。つまり、`short int` 値と `int` 値が加算されるときは、`short int` 値が `int` 型に変換されます。通常の算術変換に関与する演算子と式のリストが 103 ページの『第 6 章 式と演算子』に記載されています。

算術型のランキングは、高い方から順に次のとおりです。

表 13. 浮動小数点型の変換ランキング

オペランド型
`long double`
`double`
`float`

表 14. 10 進浮動小数点型の変換ランキング

オペランド型
▶ 400 _Decimal128
▶ 400 _Decimal64
▶ 400 _Decimal32

表 15. 整数の変換ランキング

オペランド型
unsigned long long または unsigned long long int
long long または long long int
unsigned long int
long int[1]
unsigned int[1]
int および列挙型
short int
char、signed char および unsigned char
ブール

注:

1. 一方のオペランドが unsigned int 型を持ち、他方のオペランドが long int 型を持っているが、unsigned int の値を long int で表せない場合は、両方のオペランドが unsigned long int に変換されます。

関連情報

- 45 ページの『整数型』
- 46 ページの『ブール型』
- 47 ページの『浮動小数点型』
- 48 ページの『文字型』
- 57 ページの『列挙型』
- 118 ページの『2 項式』

左辺値から右辺値への変換

コンパイラーが右辺値を期待している状況で左辺値が現れた場合、コンパイラーは、左辺値を右辺値に変換します。次の表はこれに対する例外をリストしたものです。

変換前の状況	結果の振る舞い
左辺値は関数型です。	
左辺値は配列です。	
左辺値の型は不完全型です。	コンパイル時エラー
左辺値は未初期化オブジェクトを参照します。	未定義の振る舞い
左辺値は右辺値の型でもなく、また右辺値の型から派生した型でもないオブジェクトを参照します。	未定義の振る舞い
C++ 左辺値は const または volatile のいずれかによって修飾される非クラス型です。	変換後の型は const によっても volatile によっても修飾されません。

関連情報

- 103 ページの『左辺値と右辺値』

ポインター型変換

ポインター型変換は、ポインターが使用されるときに実行されます。この型変換には、ポインターの割り当て、初期化、および比較が含まれます。

C のみ

ポインターを含む変換は、明示的型キャストを使用しなければなりません。この規則の例外は、C ポインターに関する許容代入変換です。次の表の中で、const 修飾された左辺値は、代入の左方オペランドとして使用できません。

表 16. C ポインターに関する有効な代入変換

左方オペランドの型	許可される右方オペランドの型
(オブジェクト) T へのポインター	• 定数 0 • T と互換性のある型へのポインター • void へのポインター (void*)
(関数) F へのポインター	• 定数 0 • F と互換性のある関数へのポインター

左方オペランドの参照された型は、右方オペランドと同じ修飾子を持ちます。他のポインターの型が void* の場合は、オブジェクト・ポインターの型が不完全になる場合があります。

C のみ の終り

ゼロ定数からヌル・ポインターへ

評価がゼロになる定数式は、ヌル・ポインター定数 です。この式をポインターに変換することができます。このポインターは、ヌル・ポインター (ゼロ値を持つポインター) となり、どのオブジェクトをも指さないようになります。 C++ また、評価がゼロになる定数式も、メンバーへのヌル・ポインターに変換されます。

配列からポインターへ

型「*N* の配列」(*N* は配列の単一エレメントの型) を持つ左辺値または右辺値から *N** へ。結果は、配列の初期エレメントを指すポインターです。しかし、式が & (アドレス) 演算子または sizeof 演算子のオペランドとして使用される場合には、この変換は行うことができません。

関数からポインターへ

関数である左辺値は同じ型の関数を指すポインターである右辺値に変換できます。ただし、式が & (アドレス) 演算子、() (関数呼び出し) 演算子、または sizeof 演算子のオペランドとして使用される場合を除きます。

関連情報

- 74 ページの『ポインター』
- 105 ページの『整数定数式』
- 78 ページの『配列』
- 195 ページの『関数へのポインター』
- 234 ページの『メンバーへのポインター』
- 268 ページの『ポインター型変換』

void* への変換

C ポインターは、必ずしも、型 int と同じサイズではありません。関数に渡されるポインター引数は、その関数によって期待される正しい型が必ず渡されるように、明示的にキャストでなければなりません。 C の汎用オブジェクト・ポインターは void* ですが、汎用関数ポインターは存在しません。

オプションとして型が修飾されたオブジェクトを指すすべてのポインターは、同じ const または volatile 修飾を保持しながら、void* に変換できます。

C のみ

以下の表に、左方オペランドとして void* を持つ許容代入変換を示します。

表 17. C での void に関する有効な代入変換*

左方オペランドの型	許可される右方オペランドの型
(void*)	• 定数 0。 • オブジェクトを指すポインター。オブジェクトは型が不完全な場合があります。 • (void*)

C のみ の終り

C++ のみ。

標準型変換を使用して、関数へのポインターを型 void* に変換することはできません。void* に関数を保持するだけの十分なビットがある場合には、明示的に行うことができます。

C++ のみ。 の終り

関連情報

- 48 ページの『void 型』

参照変換 (C++ のみ)

参照変換は、参照初期化が行われる場合には、いつでも実行することができます (引数の受け渡しおよび関数からの戻り値で実行される参照初期化の場合も含めて)。変換があいまいでなければ、クラスへの参照を、そのクラスのアクセス可能な基底クラスへの参照に変換することができます。変換の結果は、派生クラス・オブジェクトの基底クラス・サブオブジェクトへの参照になります。

参照変換は、対応するポインター型変換が許される場合に許されます。

関連情報

- 80 ページの『参照 (C++ のみ)』
- 88 ページの『参照の初期化 (C++ のみ)』
- 189 ページの『関数呼び出し』
- 181 ページの『関数からの戻り値』

修飾変換 (C++ のみ)

ゼロ個以上の const 修飾または volatile 修飾を含む任意の型の型修飾右辺値は、2 番目の右辺値が最初の右辺値より多くの const 修飾または volatile 修飾を含んでいれば、型修飾型の右辺値に変換できます。

クラスのメンバーを指す型ポインターの右辺値は、2 番目の右辺値が最初の右辺値より多くの const 修飾または volatile 修飾を含んでいれば、クラスのメンバーを指す型ポインターの右辺値に変換できます。

関連情報

- 62 ページの『型修飾子』

関数引数変換

関数が呼び出されたときに、関数宣言が存在していて宣言された引数型が含まれている場合、コンパイラーは型検査を行います。コンパイラーは、呼び出し側の関数によって提供されるデータ型を、呼び出し先の関数が予想するデータ型と比較し、必要な型変換を行います。例えば次の例で、関数 funct が呼び出されると、引数 f は、double に、引数 c は、int に変換されます。

```
char * funct (double d, int i);
     /* ... */
int main(void)
{
   float f;
   char c;
   funct(f, c) /* f is converted to a double, c is converted to an int */
   return 0;
}
```

関数が呼び出されたときに可視の関数宣言がない場合、またはプロトタイプ引数リストの可変部分に式が引数として現れると、コンパイラーは、関数に引数を渡す前にデフォルトの引数拡張を行うか、または式の値を変換します。自動変換では、次のことが行われます。

- 整数値と浮動小数点値がプロモートされます。
- 配列または関数はポインターに変換されます。
- C++ 非静的クラス・メンバー関数はメンバーを指すポインターに変換されます。

関連情報

- 97 ページの『整数および浮動小数点拡張』
- 69 ページの『transparent_union 型属性 (C のみ)』
- 109 ページの『関数呼び出し式』
- 189 ページの『関数呼び出し』

第 6 章 式と演算子

式は、計算を指定する演算子、オペランド、および区切り子のシーケンスです。式に含まれている演算子と、それらの演算子が使われるコンテキストに基づいて式の評価が行われます。式の結果、値が生じ、*副次作用*が生じる場合があります。副次作用とは、実行環境の状態における変化のことです。

以下のセクションで、それぞれのタイプの式について説明します。

- 『左辺値と右辺値』
- 104 ページの『1 次式』
- 109 ページの『関数呼び出し式』
- 109 ページの『メンバー式』
- 110 ページの『単項式』
- 118 ページの『2 項式』
- 131 ページの『条件式』
- 140 ページの『複合リテラル式』
- 133 ページの『キャスト式』
- 141 ページの『new 式 (C++ のみ)』
- 145 ページの『delete 式 (C++ のみ)』
- 146 ページの『throw 式 (C++ のみ)』

146 ページの『演算子優先順位と結合順序』には、上にリストしたさまざまなセクションで説明したすべての演算子の優先順位をリストした表があります。

C++ C++ の演算子は、クラス型のオペランドに適用されるときに、異なる振る舞いを行うように定義できます。これは演算子の*多重定義* と呼ばれ、 207 ページの『演算子の多重定義』 で説明しています。

左辺値と右辺値

オブジェクト は、検査して、保管に使用できるストレージの領域です。*左辺値* は、そのようなオブジェクトを参照する式です。左辺値は、それが指定するオブジェクトの変更を必ずしも許可するわけではありません。例えば、`const` オブジェクトは、変更が不可能な左辺値です。*変更可能な左辺値* という用語は、その左辺値は、指定されたオブジェクトを検査するだけでなく、変更できることを強調するために使用されます。次のオブジェクト・タイプは左辺値ですが、変更可能な左辺値ではありません。

- 配列型
- 不完全型
- `const` 修飾型
- そのメンバーの 1 つが `const` 型として修飾されている構造体または共用体型。

これらの左辺値は変更可能ではないため、代入ステートメントの左辺に置くことはできません。

用語*右辺値* は、メモリー内のいずれかのアドレスに保管されるデータ値のことです。右辺値は、それに代入する値を持つことができない式です。リテラル定数および変数は両方とも、右辺値として使用できます。

© Copyright IBM Corp. 1998, 2010

右辺値を必要とするコンテキストで左辺値が現れると、左辺値は暗黙的に右辺値に変換されます。しかし、その逆は行われません。つまり、右辺値は左辺値に変換されません。右辺値は常に、完全型または void 型を持っています。

C C は、*関数指定子* を関数型の式として定義します。関数指定子は、オブジェクト型または左辺値とは異なります。関数指定子は関数の名前、または関数ポインターを間接参照した結果を使用できます。C 言語でも、その関数ポインターとオブジェクト・ポインターの処理を区別しています。

C++ 一方 C++ では、参照を戻す関数呼び出しは左辺値です。それ以外の場合、関数呼び出しは右辺値式です。C++ では、式はすべて左辺値または右辺値 を生成するか、値をまったく生成しません。

C および C++ の両方において、演算子の中には、それらの一部のオペランドとして左辺値を必要とするものがあります。下表は、そのような演算子と、その使い方に対する追加制限を掲載しています。

演算子	**要件**
& (単項)	オペランドは、左辺値でなければなりません。
++ --	オペランドは、左辺値でなければなりません。これは、前置および後置の両形式に適用されます。
= += -= *= %= <<= >>= &= ^= |=	左方オペランドは左辺値である必要があります。

例えば、すべての代入演算子は、それらの右方オペランドを評価し、その値を左方オペランドに代入します。左方オペランドは、変更可能な左辺値、または変更可能なオブジェクトへの参照である必要があります。

アドレス演算子 (&) には、オペランドとして左辺値が必要です。一方、増分演算子 (++) と減分演算子 (--) には、修正可能な左辺値がオペランドとして必要です。以下の例は、式およびその対応する左辺値を示します。

式	**左辺値**
`x = 42`	`x`
`*ptr = newvalue`	`*ptr`
`a++`	`a`
C++ `int& f()`	`f()` への関数呼び出し

関連情報

- 78 ページの『配列』
- 98 ページの『左辺値から右辺値への変換』

1 次式

1 次式 は、次の一般的なカテゴリーに分かれます。

- 名前 (ID)
- リテラル (定数)
- 整数定数式
- ID 式 (C++ のみ)

- 括弧で囲んだ式 ( )
- C++ this ポインター (236 ページの『this ポインター』で説明)
- C++ スコープ・レゾリューション演算子 (::) によって修飾された名前

名前

名前の値は、その型によって異なります。型は、その名前がどのように宣言されるかによって決まります。次の表は、名前が左辺値式であるかどうかを示します。

表 18. 1 次式: 名前

名前の宣言	評価対象	左辺値である
算術型、ポインター型、列挙型、構造体型、または共用体型の変数	その型のオブジェクト	はい
列挙型定数	関連した整数値	いいえ
アレイ	その配列。変換の対象となるコンテキストでは、その配列の最初のオブジェクトへのポインター (sizeof 演算子への引数として名前が使用される場合を除く)。	C いいえ C++ はい
関数	その関数。変換が必要なコンテキストでは、名前が sizeof 演算子の引数として使用される場合を除いて、または関数呼び出し式の中で関数として使用される場合を除いて、その関数へのポインター。	C いいえ C++ はい

名前は、式として、ラベル、typedef 名、構造体メンバー、共用体メンバー、構造体タグ、共用体タグ、または列挙型タグを参照することはできません。これらの目的で使用される名前は、式で使用される名前のネーム・スペースとは異なるネーム・スペースにあります。ただし、これらの名前の一部は、特殊構造によって式の中から参照できます。例えば、ドット演算子または矢印演算子を使用して、構造体および共用体メンバーを参照できます。また、キャストで、または sizeof 演算子への引数として、typedef 名を使用できます。

リテラル

リテラルは、数値定数またはストリング・リテラルです。リテラルが式として評価されると、その値は定数です。字句定数は左辺値にはなりません。ただし、ストリング・リテラルは左辺値です。

関連情報

- 17 ページの『リテラル』
- 236 ページの『this ポインター』

整数定数式

整数コンパイル時定数 は、コンパイルの間に決定される値で、実行時に変更することはできません。*整数コンパイル時定数式* は、定数で構成されていて定数に対して評価される式です。

整数定数式は、以下の項目だけで構成される式です。

- リテラル

- 列挙子
- const 変数
- 整数または列挙型の静的データ・メンバー
- 整数型のキャスト
- sizeof 式。オペランドが可変長配列でない場合。

可変長配列型に適用される sizeof 演算子は、実行時に評価されるため、定数式ではありません。

以下のような状況においては、整数定数式を使用する必要があります。
- 添え字宣言子の中 (バインドされた配列の記述として)。
- switch ステートメントのキーワード case の後。
- 列挙子の中で、enum 定数の数値として。
- ビット・フィールド幅の指定子の中。
- プリプロセッサー #if ステートメントの中で。 (列挙型定数、アドレス定数、および sizeof は、プリプロセッサーの #if ステートメントでは指定できません。)

関連情報

- 116 ページの『sizeof 演算子』

ID 式 (C++ のみ)

ID 式 (*id-expression*) は、制限された形の 1 次式です。構文的に、*id-expression* は、C++ のすべての言語エレメントの名前を提供する上で単純な ID よりも複雑さのレベルが高くなります。

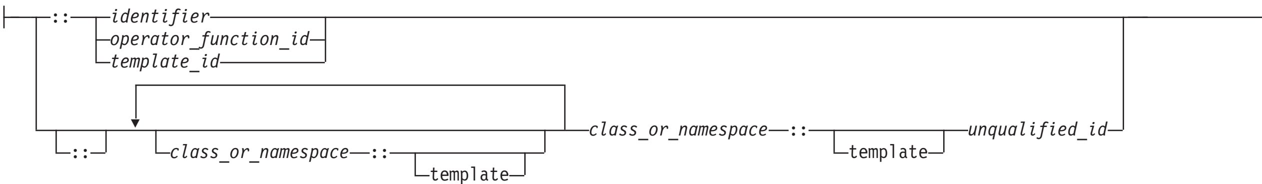
id-expression は、修飾された ID であっても、修飾されない ID であってもかまいません。また、ドット演算子および矢印演算子の後にあってもかまいません。

ID 式の構文

unqualified_id:

```
├──┬─identifier──────────────┬──────────────────────────────────────────┤
   ├─operator_function_id────┤
   ├─conversion_function_id──┤
   ├─~──class_name───────────┤
   └─template_id─────────────┘
```

qualified_id:

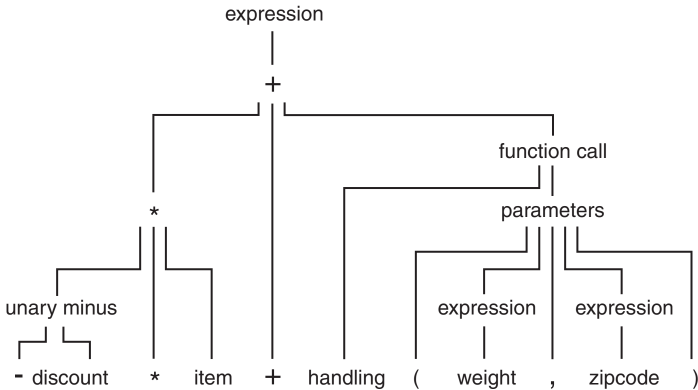


関連情報

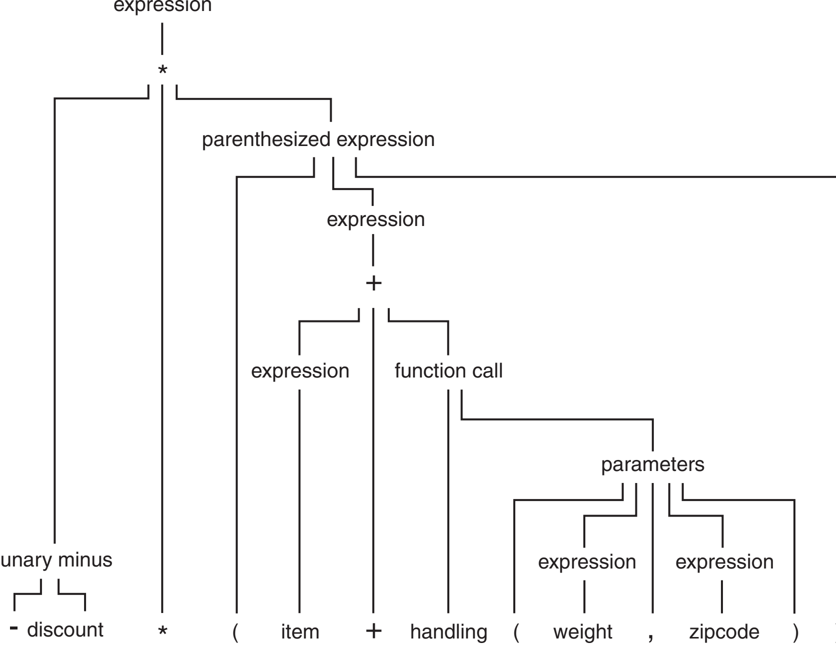
- 15 ページの『ID』
- 71 ページの『第 4 章 宣言子』

括弧で囲んだ式 ( )

小括弧を使用して、式の評価順序を明示的に強制します。次の式は、オペランドと演算子をグループ化するための小括弧は使用していません。 `weight, zipcode` を囲む式によって、関数呼び出しが形成されます。コンパイラーが式の中のオペランドと演算子を、演算子の優先順位や結合順序の規則に従って、グループ化する方法に注意してください。



次の式は、以前の式と似ていますが、オペランドおよび演算子のグループ化方法を変更する括弧を含んでいます。



結合属性および可換属性の両方がある演算子を含む式では、小括弧を使用して、オペランドと演算子をグループ化する方法を指定できます。次の式の小括弧は、オペランドと演算子をグループ化する順序を確実にします。

```
x = f + (g + h);
```

関連情報

- 146 ページの『演算子優先順位と結合順序』

スコープ・レゾリューション演算子 :: (C++ のみ)

:: (スコープ・レゾリューション) 演算子は、隠された名前を修飾して、それらの名前を引き続き使用できるようにするために使われます。ブロックまたはクラス内の同じ名前の明示宣言によって、ネーム・スペース名またはグローバル・スコープ名が隠されている場合は、単項スコープ演算子を使用できます。たとえば、次のとおりです。

```
int count = 0;

int main(void) {
  int count = 0;
  ::count = 1;  // set global count to 1
  count = 2;    // set local count to 2
  return 0;
}
```

main 関数で宣言された count の宣言は、グローバル・ネーム・スペース・スコープで宣言された count という名前の整数を隠蔽します。ステートメント ::count = 1 は、グローバル・ネーム・スペース・スコープで宣言された count という名前の変数にアクセスします。

また、クラス・スコープ演算子を使用して、クラス名またはクラス・メンバーの名前を修飾することもできます。隠されているクラス・メンバー名は、そのクラス名とクラス・スコープ演算子を修飾することによって、使用することができます。

次の例では、変数 X の宣言によって、クラス型 X が隠されますが、静的クラス・メンバー count は、クラス型 X とスコープ・レゾリューション演算子で修飾することによって、まだ使用することができます。

```
#include <iostream>
using namespace std;

class X
{
public:
      static int count;
};
int X::count = 10;                // define static data member

int main ()
{
      int X = 0;                  // hides class type X
      cout << X::count << endl;   // use static member of class X
}
```

関連情報

- 223 ページの『クラス名のスコープ』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

関数呼び出し式

関数呼び出し は、関数名と、それに続く関数呼び出し演算子を含む式です。(). 関数がパラメーターを受け取るように定義されている場合は、その関数に送られる値が関数呼び出し演算子の括弧内にリストされます。引数リストには、コンマで区切った式をいくつでも入れることができます。引数リストは、空にすることもできます。

関数呼び出し式の型は、その関数の戻りの型です。この型は、完全型、参照型、または型 void のいずれかです。 ▶ C 関数呼び出し式は常に右辺値です。 ▶ C++ 関数呼び出しは、関数の型が参照である場合に限って、左辺値です。

以下に、関数呼び出し演算子の例をいくつか示します。

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

関数呼び出しの引数の評価順序は、指定されていません。以下の例では、次のようになります。

```
method(sample1, batch.process--, batch.process);
```

引数 batch.process-- が最後に評価されることもあり、最後の 2 つの引数が同じ値で渡されることが生じます。

関連情報

- 101 ページの『関数引数変換』
- 189 ページの『関数呼び出し』

メンバー式

メンバー式は、クラス、構造体、または共用体のメンバーを示します。メンバー演算子は次のとおりです。

- ドット演算子 .
- 矢印演算子 ->

ドット演算子 .

. (ドット) 演算子は、クラス、構造体、または共用体メンバーにアクセスするために使われます。メンバーは、後置式によって指定され、その後に . (ドット) 演算子、さらにその後に、できるだけ修飾された ID または疑似デストラクター名が続きます。(*疑似デストラクター* は、非クラス型のデストラクターです。) 後置式は、class、struct、または union 型のオブジェクトでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。後置式と名前が左辺値の場合は、その式の値も左辺値です。後置式が型修飾されている場合、結果式内の指定メンバーには同じ型修飾子が適用されます。

関連情報

- 56 ページの『構造体および共用体メンバーへのアクセス』
- 288 ページの『疑似デストラクター』

矢印演算子 ->

-> (矢印) 演算子は、ポインターを使用するクラス、構造体または共用体メンバーにアクセスするために使われます。後置式、その後に続く -> (矢印) 演算子、さらにその後に、できるだけ修飾された ID または疑似デストラクター名が続き、ポインターが指すオブジェクトのメンバーを指定します。 (*疑似デストラクター* は、非クラス型のデストラクターです。)後置式は、`class`、`struct`、または `union` 型のオブジェクトを指すポインターでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。名前が左辺値の場合は、式の値も左辺値です。式が修飾型へのポインターである場合、同じ型修飾子が結果の式の中の指定されたメンバーに適用されます。

関連情報

- 74 ページの『ポインター』
- 56 ページの『構造体および共用体メンバーへのアクセス』
- 229 ページの『第 12 章 クラスのメンバーおよびフレンド (C++ のみ)』
- 288 ページの『疑似デストラクター』

単項式

単項式 には、1 つのオペランドと単項演算子が含まれています。

サポートされる単項演算子は次のとおりです。

- 増分演算子 ++
- 減分演算子 --
- 単項正演算子 +
- 単項負演算子 -
- 論理否定演算子 !
- ビット単位否定演算子 ~
- アドレス演算子 &
- 間接演算子 *
- C++ typeid
- 400 alignof
- sizeof
- 400 __typeof__

147 ページの表 22 に示すように、すべての単項演算子には同じ優先順位が付けられ、右から左の結合順序が指定されます。

演算子の説明で示されているように、ほとんどの単項式のオペランドで、通常の算術変換を実行することができます。

関連情報

- 75 ページの『ポインター演算』
- 103 ページの『左辺値と右辺値』
- 95 ページの『算術変換とプロモーション』

増分演算子 ++

++ (増分) 演算子は、スカラー・オペランドの値に 1 を加えます。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ増分します。オペランドは、増分演算の結果を受け取ります。オペランドは、算術型またはポインター型の変更可能な左辺値でなければなりません。

++ は、オペランドの前にも後にも置くことができます。それがオペランドの前にくると、オペランドは増分されます。増分された値が、式の中で使用されます。オペランドの後に ++ を置くと、オペランドを増分する*前に*、そのオペランドの値が使用されます。次に例を示します。

```
play = ++play1 + play2++;
```

これは、以下の式に似ています。play2 は play の前で変更されます。

```
int temp, temp1, temp2;

temp1 = play1 + 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 + 1;
play = temp;
```

結果は、整数拡張後のオペランドと同じ型となります。

オペランドには、通常の算術変換が実行されます。

減分演算子 --

-- (減分) 演算子は、スカラー・オペランドの値から 1 を減算します。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ、減じます。オペランドは、減分演算の結果を受け取ります。オペランドは、変更可能な左辺値でなければなりません。

減分演算子は、オペランドの前後に -- を入れることができます。この演算子がオペランドの前にあると、オペランドを減分し、減らした値が式で使われます。 -- がオペランドの後にある場合は、オペランドの現行値が式で使われ、その後でオペランドが減らされます。

次に例を示します。

```
play = --play1 + play2--;
```

これは、以下の式に似ています。play2 は play の前で変更されます。

```
int temp, temp1, temp2;

temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;
```

その結果には、オペランドと同じ型が指定されます (可能な整数拡張の場合を除く) が、左辺値ではありません。

オペランドには、通常の算術変換が実行されます。

単項正演算子 +

+ (単項正) 演算子は、オペランドの値を保持します。オペランドには、任意の算術型またはポインター型を指定できます。結果は、左辺値ではありません。

結果は、整数拡張後のオペランドと同じ型となります。

注: 定数の前の正符号は、定数の一部ではありません。

単項負演算子 -

- (単項負) 演算子は、オペランドの値を否定します。オペランドには、任意の算術型を指定できます。結果は、左辺値ではありません。

例えば、`quality` に値 `100` が指定された場合は、`-quality` は 値 `-100` になります。

結果は、整数拡張後のオペランドと同じ型となります。

注: 定数の前の負符号は、定数の一部ではありません。

論理否定演算子 !

! (論理否定) 演算子は、オペランドが、0 (false) またはゼロ以外 (true) のいずれに評価されるかを決定します。

C オペランドの評価の結果が 0 になる場合は、式の値は 1 (真) になり、オペランドの評価の結果がゼロ以外の値になる場合は、式の値は 0 (偽) になります。

C++ 式は、オペランドが false (0) に評価されると値 true になり、オペランドが true (ゼロ以外) に評価されると値 false になります。オペランドは、暗黙的にブールに変換されます。そして、結果の型はブールです。

次の 2 つの式は、同じです。

```
!right;
right == 0;
```

関連情報

- 46 ページの『ブール型』

ビット単位否定演算子 ~

~ (ビット単位否定) 演算子は、オペランドのビット単位の補数を生成します。結果の 2 進表示では、すべてのビットは、オペランドの 2 進表示の同じビットの値と反対の値を保持します。オペランドには、整数型が指定されている必要があります。結果には、オペランドと同じ型が指定され、左辺値ではありません。

`x` が、10 進数の値 5 を表すとします。`x` の 16 ビットの 2 進表示は次のとおりです。

```
0000000000000101
```

式 `~x` の結果は、次のようになります (ここでは、16 ビットの 2 進数で表されます)。

```
1111111111111010
```

~ 文字は、三文字表記文字の `??-` によって表されることに注意してください。

~0 の 16 ビットの 2 進表示は、次のとおりです。

1111111111111111

関連情報

- 31 ページの『3 文字表記』

アドレス演算子 &

& (アドレス) 演算子は、そのオペランドを指すポインターを生成します。オペランドは、左辺値、関数指定子、または修飾名でなければなりません。オペランドは、ビット・フィールドであることも、ストレージ・クラス register を指定することもできません。

オペランドが左辺値または関数である場合は、結果の型は式型を指すポインターです。例えば、式に型 int が指定されている場合、結果は、型 int が指定されたオブジェクトを指すポインターになります。

オペランドが修飾名で、メンバーが静的でない場合は、結果は、クラスのメンバーを指すポインターになり、メンバーと同じ型になります。結果は、左辺値ではありません。

p_to_y が int を指すポインターとして定義され、 y が int として定義されている場合、次の式では、変数 y のアドレスをポインター p_to_y に代入します。

```
p_to_y = &y;
```

C++ のみ。

アンパーサンド記号 & は、C++ では、アドレス演算子のほかに、参照宣言子として使用されます。意味は関連していますが、同一ではありません。

```
int target;
int &rTarg = target;  // rTarg is a reference to an integer.
                      // The reference is initialized to refer to target.
void f(int*& p);      // p is a reference to a pointer
```

参照のアドレスを取ると、そのターゲットのアドレスが戻されます。直前の宣言を使用して、&rTarg は &target と同じメモリー・アドレスとなります。

レジスター変数のアドレスを取ることができます。

使用する多重定義関数のバージョンを、左側が固有に判別する初期化または代入の場合に限り、多重定義関数で & 演算子を使用することができます。

C++ のみ。 の終り

関連情報

- 『間接演算子 *』
- 74 ページの『ポインター』
- 80 ページの『参照 (C++ のみ)』

間接演算子 *

* (間接) 演算子は、ポインター型オペランドによって参照される値を決めます。オペランドは、不完全型を指すポインターであってはなりません。オペランドがオブジェクトを指し示している場合は、演算子の結

果、そのオブジェクトを参照する左辺値が導き出されます。オペランドが関数を指している場合、結果は C++ では、そのオペランドが指しているオブジェクトを参照する左辺値です。 配列と関数はポインターに変換されます。

オペランドの型は、結果の型を決定します。例えば、オペランドが int を指すポインターの場合は、結果は、int 型になります。

無効なアドレス (NULL など) を含むポインターに間接演算子を適用しないでください。結果は、予測できません。

p_to_y が int を指すポインターとして定義され、 y が int として定義されている場合、式は次のようになります。

```
p_to_y = &y;
*p_to_y = 3;
```

変数 y は値 3 を受け取ります。

関連情報

- 78 ページの『配列』
- 74 ページの『ポインター』

typeid 演算子 (C++ のみ)

typeid 演算子によって、プログラムは、ポインターまたは参照により参照されるオブジェクトの実際の派生型を検索することができます。この演算子は、dynamic_cast 演算子とともに、C++ における RTTI (run-time type identification) サポート用に提供されています。

typeid 演算子構文

typeid 演算子は、実行時型情報 (RTTI) の生成を要求します。これは、コンパイラー・オプションによってコンパイル時に明示的に指定する必要があります。

typeid 演算子は、式 *expr* の型を表す型 const std::type_info の左辺値を戻します。 typeid 演算子を使用するためには、標準テンプレート・ライブラリー・ヘッダー <typeinfo> をインクルードしておく必要があります。

expr が、ポリモアフィック・クラスへの参照または参照解除ポインターである場合、typeid は、実行時に参照またはポインターが示すオブジェクトを表す type_info オブジェクトを戻します。ポリモアフィック・クラスでない場合、typeid は、参照の型または参照解除ポインターを表す type_info オブジェクトを戻します。次の例は、このことを示しています。

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };
```

```
int main() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;

  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}
```

次に、上記の例の出力を示します。

```
ap: B
ar: B
cp: C
cr: C
```

クラス A と B はポリモアフィックで、クラス C と D はそうではありません。 cp と cr は、型 D のオブジェクトを参照していますが、typeid(*cp) と typeid(cr) は、クラス C を表すオブジェクトを戻します。

左辺値から右辺値へ、配列からポインターへ、および関数からポインターへの変換は、*expr* へは適用されません。例えば、次の例の出力は int [10] であって、int * ではありません。

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main() {
  int myArray[10];
  cout << typeid(myArray).name() << endl;
}
```

expr がクラス型であれば、そのクラスは、完全に定義される必要があります。

typeid 演算子は、トップレベルの const または volatile 修飾子を無視します。

関連情報

- 73 ページの『型名』
- 117 ページの『__typeof__ 演算子』

__alignof__ 演算子

IBM 拡張

__alignof__ 演算子は、C99 および標準 C++ に対する言語拡張であり、オペランドの位置合わせに使用されるバイト数を返します。オペランドは式であっても、括弧で囲んだ型 ID であってもかまいません。オペランドが左辺値を表す式である場合、__alignof__ によって戻される数は、左辺値が持つと認知されている位置合わせを表します。式の型はコンパイル時に判別されますが、式そのものは評価されません。オペランドが型である場合、その数値は、ターゲット・プラットフォーム上でその型に対して通常必要な位置合わせを表します。

__alignof__ 演算子を次の項目に適用することはできません。

- ビット・フィールドを表す左辺値

- 関数型
- 未定義の構造体またはクラス
- 不完全型 (`void` など)

__alignof__ 演算子構文

```
▶▶─__alignof__──┬─unary_expression─┬──────────────────────────────────▶◀
                └─(─type-id─)──────┘
```

type-id が参照、または参照された型である場合、結果は、参照された型の位置合わせです。 *type-id* が配列である場合、結果は、配列エレメント型の位置合わせです。 *type-id* が基本型である場合、結果はインプリメンテーションでの定義によります。

関連情報

- 91 ページの『aligned 変数属性』

IBM 拡張 の終り

sizeof 演算子

`sizeof` 演算子は、オペランドのサイズをバイトで生成します。オペランドは式、または型の括弧付きの名前とすることができます。

sizeof 演算子構文

```
▶▶─sizeof──┬─expr──────────────┬──────────────────────────────────▶◀
           └─(─type-name─)─────┘
```

どちらの種類のオペランドであっても、結果は左辺値ではなく、定数整数値です。結果の型は、ヘッダー・ファイル `stddef.h` で定義された符号なし整数型 `size_t` です。

プリプロセッサー・ディレクティブの中以外では、整数定数が必要なときは、`sizeof` 式を使用できます。`sizeof` 演算子がよく使われる 1 つの例は、ストレージ割り当て時、入力関数、および出力関数で参照されるオブジェクトのサイズを決める場合です。

もう 1 つの `sizeof` の使い方は、プラットフォームをまたがってコードを移植する場合です。データ型が表すサイズを決めるために、`sizeof` 演算子を使用できます。次に例を示します。

```
sizeof(int);
```

型名に適用された `sizeof` 演算子は、その型のオブジェクトによって使用されるメモリー量 (内部または末尾の埋め込みを含む) を結果として出します。

複合型の場合、結果は以下のようになります。

オペランド	結果
配列	結果は、配列内のバイトの合計数になります。例えば、10 のエレメントがある配列では、サイズは、単一のエレメントのサイズの 10 倍になります。コンパイラーは、式を評価する前には、配列をポインターに変換しません。
C++ クラス	結果は常に非ゼロで、そのクラスのオブジェクトのバイト数になります (配列にクラス・オブジェクトを配置するために必要な埋め込みを含む)。

オペランド	結果
C++ 参照	結果は、参照されるオブジェクトのサイズになります。

sizeof 演算子を次の項目に適用することはできません。

- ビット・フィールド
- 関数型
- 未定義の構造体またはクラス
- 不完全型 (void など)

式に適用された sizeof 演算子は、式の型の名前にのみ適用された場合と同じ結果を出します。コンパイル時に、コンパイラーは式を分析してその型を判別します。式の型の分析で行われる通常の型変換は、いずれも sizeof 式に直接付随するものではありません。ただし、オペランドに型変換を行う演算子が含まれている場合、コンパイラーは、型を判別するときに、これらの型変換を考慮します。例えば、以下の例の 2 行目では、通常の算術変換を行います。short は 2 バイトのストレージを使用し、int は 4 バイト使用しています。

```
short x; ... sizeof (x)        /* the value of sizeof operator is 2 */
short x; ... sizeof (x + 1)    /* value is 4, result of addition is type int */
```

式 x + 1 の結果は int 型で、sizeof(int) と同じです。値は、x に char、short、または int 型、あるいは任意の列挙型を指定する場合も 4 です。

関連情報

- 73 ページの『型名』
- 105 ページの『整数定数式』
- 78 ページの『配列』
- 80 ページの『参照 (C++ のみ)』

__typeof__ 演算子

IBM 拡張

__typeof__ 演算子は、その引数の型を返します。引数は式または型にすることができます。この言語フィーチャーは、式から型を引き出す手段を提供していることになります。式 e を例にとると、__typeof__(e) は、型名が必要な場所 (例えば、宣言の中やキャストの中) でも使用できます。

__typeof__ 演算子構文

```
►►──__typeof__──(──┬─expr──────┬──)──────────────────────────►◄
                   └─type-name─┘
```

__typeof__ 構成自体は式ではなく、型の名前です。__typeof__ 構成は、__typedef__ を使用して定義された型名のように動作します (構文は sizeof に似ています)。

以下の例は、その基本的な構文を示しています。式 e の場合:

```
int e;
__typeof__(e + 1) j;   /* the same as declaring int j;     */
e = (__typeof__(e)) f; /* the same as casting e = (int) f; */
```

__typeof__ 構成の使用は、typedef 名を定義した場合と同じです。例えば、

```
int T[2];
int i[2];
```

と定義した場合、以下を記述することができます。

```
__typeof__(i) a;        /* all three constructs have the same meaning */
__typeof__(int[2]) a;
__typeof__(T) a;
```

このコードの振る舞いは、int a[2]; を宣言した場合と同じです。

ビット・フィールドについては、__typeof__ はビット・フィールドの基本となる型を表します。 例えば int m:2; では、__typeof__(m) は int です。ビット・フィールドのプロパティーは予約されないので、_typeof__(m) n; 内の n は int n と同じですが、int n:2 と同じではありません。

__typeof__ 演算子は、sizeof 内およびそれ自身の中でネストが可能です。以下の int へのポインター配列としての arr の宣言は、同じです。

```
int *arr[10];                     /* traditional C declaration          */
__typeof__(__typeof__ (int *)[10]) a;  /* equivalent declaration  */
```

__typeof__ 演算子は、式 e がパラメーターであるマクロ定義で使用すると便利です。例えば、次のような場合です。

```
#define SWAP(a,b) { __typeof__(a) temp; temp = a; a = b; b = temp; }
```

関連情報

- 73 ページの『型名』
- 60 ページの『typedef 定義』
- 「*ILE C/C++ コンパイラー参照*」にある LANGLVL(*EXTENDED)

IBM 拡張 の終り

2 項式

2 項式 には、1 つの演算子によって分離される 2 つのオペランドが含まれます。サポートされる 2 項演算子は次のとおりです。

- 119 ページの『代入演算子』
- 乗算演算子 *
- 除法演算子 /
- 剰余演算子 %
- 加法演算子 +
- 減法演算子 -
- ビット単位左および右シフト演算子 << >>
- 関係演算子 < > <= >=
- 等価演算子 == および非等価演算子 !=
- ビット単位 AND 演算子 &
- ビット単位排他 OR 演算子 ^
- ビット単位包含 OR 演算子 |

- 論理 AND 演算子 &&
- 論理 OR 演算子 ||
- 配列添え字演算子 [ ]
- 129 ページの『コンマ演算子 ,』
- C++ メンバーを指すポインター演算子 .* ->* (C++ のみ)

すべての 2 項演算子は、左から右への結合順序が指定されますが、すべての 2 項演算子に、同じ優先順位が付けられるわけではありません。2 項演算子のランキングおよび優先順位の規則は、 148 ページの表 23 で要約しています。

ほとんどの 2 項演算子のオペランドが評価される順序は、指定されていません。正しい結果を得るためには、コンパイラーがオペランドを評価する順序に依存する 2 項式を作成しないようにします。

演算子の説明で示されているように、ほとんどの 2 項式のオペランドで、通常の算術変換を実行することができます。

関連情報

- 103 ページの『左辺値と右辺値』
- 95 ページの『算術変換とプロモーション』

代入演算子

代入式 は、左方オペランドによって指定されたオブジェクトに値を保管します。代入演算子には次の 2 つのタイプがあります。

- 単純代入演算子 =
- 複合代入演算子

すべての代入式の左方オペランドは、変更可能な左辺値でなければなりません。式の型は、左方オペランドの型です。式の値は、代入が完了した後の左方オペランドの値です。

C 代入式の結果は左辺値ではありません。 C++ 代入式の結果は左辺値です。

すべての代入演算子には、同じ優先順位が付けられ、右から左の結合順序が指定されます。

関連情報

- 103 ページの『左辺値と右辺値』
- 74 ページの『ポインター』
- 62 ページの『型修飾子』

単純代入演算子 =

単純代入演算子の形式は、次のとおりです。

lvalue = *expr*

演算子は、右方オペランド *expr* の値を、左方オペランド *lvalue* によって指定されたオブジェクトに保管します。

左方オペランドは、変更可能な左辺値でなければなりません。割り当て演算の型は、左方オペランドの型です。

左方オペランドがクラス型ではない場合は、右方オペランドは暗黙的に左方オペランドの型に変換されます。この変換された型は、`const` または `volatile` によって修飾されることはありません。

左方オペランドがクラス型である場合、その型は完全でなければなりません。左方オペランドのコピー代入演算子が呼び出されます。

左方オペランドが参照型のオブジェクトの場合は、参照によって示されたオブジェクトに右方オペランドの値を代入します。

複合代入演算子

複合代入演算子は、2 項演算子と単純代入演算子で構成されます。複合代入演算子は、両方のオペランドに 2 項演算子の演算を実行し、その演算の結果を左方オペランドに保管します。左方オペランドは変更可能な左辺値でなければなりません。

次の表では、複合代入式のオペランドの型を示します。

<table>
<tr><th>演算子</th><th>左方オペランド</th><th>右方オペランド</th></tr>
<tr><td>+= または -=</td><td>算術</td><td>算術</td></tr>
<tr><td>+= または -=</td><td>ポインター</td><td>整数型</td></tr>
<tr><td>*=、/=、および %=</td><td>算術</td><td>算術</td></tr>
<tr><td><<=、>>=、&=、^=、および |=</td><td>整数型</td><td>整数型</td></tr>
</table>

次の式

```
a *= b + c
```

は、以下と同等です。

```
a = a * (b + c)
```

そして、次の式とは同等でない ことに注意してください。

```
a = a * b + c
```

次の表では、複合代入演算子をリストし、各演算子を使用した式を示します。

<table>
<tr><th>演算子</th><th>例</th><th>等価な式</th></tr>
<tr><td>+=</td><td>index += 2</td><td>index = index + 2</td></tr>
<tr><td>-=</td><td>*(pointer++) -= 1</td><td>*pointer = *(pointer++) - 1</td></tr>
<tr><td>*=</td><td>bonus *= increase</td><td>bonus = bonus * increase</td></tr>
<tr><td>/=</td><td>time /= hours</td><td>time = time / hours</td></tr>
<tr><td>%=</td><td>allowance %= 1000</td><td>allowance = allowance % 1000</td></tr>
<tr><td><<=</td><td>result <<= num</td><td>result = result << num</td></tr>
<tr><td>>>=</td><td>form >>= 1</td><td>form = form >> 1</td></tr>
<tr><td>&=</td><td>mask &= 2</td><td>mask = mask & 2</td></tr>
<tr><td>^=</td><td>test ^= pre_test</td><td>test = test ^ pre_test</td></tr>
<tr><td>|=</td><td>flag |= ON</td><td>flag = flag | ON</td></tr>
</table>

等価な式の列では、左方オペランド (例の列の) を 2 回示していますが、左方オペランドを事実上 1 回しか評価しません。

C++ オペランド型のテーブルに加え、式は、暗黙的に左方オペランドの cv 非修飾型に変換されます (クラス型でない場合)。しかし、左方オペランドがクラス型の場合、そのクラスは完全になり、そのクラスのオブジェクトへの代入は、コピー代入操作として行われます。 C++ では、複合式および条件式は左辺値であり、複合代入式の左方オペランドとすることができます。

乗算演算子 *

* (乗算) 演算子は、そのオペランドの積を生成します。オペランドは、算術型または列挙型でなければなりません。結果は、左辺値ではありません。オペランドには、通常の算術変換が実行されます。

乗算演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数の乗算演算子を含む式の中でオペランドの再配置を行うことができます。例えば、次の式

```
sites * number * cost
```

は、次のいずれかの方法に解釈できます。

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

除法演算子 /

/ (除法) 演算子はそのオペランドの商を生成します。オペランドが両方とも整数である場合は、小数部分 (剰余) は破棄されます。小数部分の破棄は、多くの場合、*ゼロに切り捨て* と呼ばれます。オペランドは、算術型または列挙型でなければなりません。右方オペランドはゼロにできません。右方オペランドが `0` の場合、結果は未定義となります。例えば、式 `7 / 4` は、値 `1` を生成します (`1.75` または `2` ではありません)。結果は、左辺値ではありません。

オペランドには、通常の算術変換が実行されます。

剰余演算子 %

% (剰余) 演算子は、左方オペランドを右方オペランドで割り算した剰余を生成します。例えば、式 `5 % 3` は `2` になります。結果は、左辺値ではありません。

オペランドは両方とも、整数型または列挙型でなければなりません。右方オペランドが `0` になる場合は、結果は予期できません。いずれかのオペランドに負の値がある場合は、`b` が `0` でなく、`a/b` が表示可能な場合は、結果は次の式のように、常に値 `a` になります。

```
( a / b ) * b + a %b;
```

オペランドには、通常の算術変換が実行されます。

剰余の符号は商の符号と同じです。

加法演算子 +

+ (加法) 演算子は、そのオペランドの合計を生成します。オペランドは両方とも算術型を保持するか、一方のオペランドがオブジェクト型を指すポインターで、もう一方のオペランドが整数型または列挙型を保持する必要があります。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

配列内のオブジェクトを指すポインターは、整数型を持つ値に加算できます。結果は、ポインター・オペランドと同じ型のポインターになります。結果は、オリジナルのエレメントから、添え字として扱われる整数値の量だけオフセットされた、配列の中の別のエレメントを参照します。結果のポインターが、配列の外側のストレージ (配列の外側の最初のロケーション以外) を指す場合、結果は予期できません。配列の終わりより 1 つ後のエレメントを指すポインターを使用して、そのアドレスにあるメモリーの内容にアクセスすることはできません。コンパイラーは、ポインターの境界検査は行いません。例えば、以下の例で、加算の後で、`ptr` は配列の 3 番目のエレメントを指します。

```
int array[5];
int *ptr;
ptr = array + 2;
```

関連情報

- 75 ページの『ポインター演算』
- 99 ページの『ポインター型変換』

減法演算子 -

- (減法) 演算子は、そのオペランドの差を生成します。オペランドは両方とも算術型または列挙型を保持するか、左方オペランドがポインター型で、右方オペランドがポインターの型と同じ型か整数型または列挙型を保持する必要があります。整数値からポインターを減算することはできません。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

左方オペランドがポインターで、右方オペランドが整数型を保持するときは、コンパイラーは、右方の値を相対位置のアドレスに変換します。結果は、ポインター・オペランドと同じ型のポインターになります。

オペランドが両方とも同じ配列内のエレメントを指すポインターである場合は、結果は、2 つのアドレスを分離するオブジェクトの数です。ポインターが同じ配列のオブジェクトを参照しない場合は、振る舞いは未定義です。

▶ 400 ポインター差分演算の結果の型は、 TERASPACE(*NO) コンパイラー・オプションを指定した場合、`ptrdiff_t` (`stddef.h` で定義) になります。 TERASPACE(*YES) コンパイラー・オプションを指定すると、ポインター差分演算の結果は signed long long 型になります。

関連情報

- 75 ページの『ポインター演算』
- 99 ページの『ポインター型変換』

ビット単位左および右シフト演算子 << >>

ビット単位シフト演算子は、2 進数オブジェクトのビット値を移動します。左方オペランドには、シフトされる値を指定します。右方オペランドには、値のビットがシフトされる桁数を指定します。結果は、左辺値ではありません。両方のオペランドに同じ優先順位が付けられ、左から右の結合順序が指定されます。

演算子	使用法
<<	ビットが左方にシフトされることを指示します。

演算子	使用法
>>	ビットが右方にシフトされることを指示します。

各オペランドは、整数型または列挙型でなければなりません。コンパイラーは、オペランドの整数拡張を実行し、その後、右方オペランドが `int` 型に変換されます。結果には、左方オペランドと同じ型が指定されます (算術変換の後)。

右方オペランドが負の値、またはシフトされる式のビット幅より大きいかまたは等しい値を保持しないようにする必要があります。このような値でビット単位シフトを行うと、結果は予測不能な値になります。

右方オペランドに `0` がある場合は、結果は左方オペランドの値になります (通常の算術変換が実行された後)。

<< 演算子は、空になったビットをゼロで埋めます。例えば、`left_op` が値 `4019` を持っている場合、`left_op` のビット・パターン (16 ビットの形式) は次のようになります。

```
0000111110110011
```

式 `left_op << 3` は、次のビット・パターンを生成します。

```
0111110110011000
```

式 `left_op >> 3` は、次のビット・パターンを生成します。

```
0000000111110110
```

関係演算子 < > <= >=

関係演算子は、2 つのオペランドを比較して、リレーションシップの妥当性を判別します。次の表では、4 つの関係演算子を説明します。

演算子	使用法
<	左方オペランドの値が、右方オペランドの値より小さいかどうかを示します。
>	左方オペランドの値が、右方オペランドの値より大きいかどうかを示します。
<=	左方オペランドの値が、右方オペランドの値より小さいまたは等しいかどうかを示します。
>=	左方オペランドの値が、右方オペランドの値より大きいまたは等しいかどうかを示します。

オペランドは両方とも、算術型または列挙型を保持するか、同じ型を指すポインターでなければなりません。

C 結果の型は `int` で、指定された関係が真であれば値 `1` を持ち、偽であれば `0` を持ちます。

C++ 結果の型は `bool` で、値 真 または 偽 を持ちます。

結果は、左辺値ではありません。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。ポインターが同じ配列のオブジェクトを参照しない場合は、結果は未定義になります。

ポインターは、0 に評価される定数式と比較することができます。また、ポインターを `void*` 型のポインターと比較することもできます。ポインターを `void*` 型のポインターに変換します。

2 つのポインターが同じオブジェクトを参照する場合は、この 2 つのポインターは等しいと見なされます。 2 つのポインターが同一オブジェクトの非静的メンバーを参照する場合、これらのポインターがアクセス指定子によって分離されていなければ、後で宣言されるオブジェクトを指すポインターの方がより大きいです。分離されていれば、比較は未定義です。 2 つのポインターが同じ共用体のデータ・メンバーを参照する場合は、この 2 つのポインターは同じアドレス値を保持します。

2 つのポインターが同じ配列のエレメント、または配列の最後のエレメントを超えて最初のエレメントを参照する場合、より大きいサブスクリプト値を持っているエレメントを指すポインターの方が、より大きいです。

関係演算子では、同じオブジェクトのメンバーだけを比較できます。

関係演算子には、左から右の結合順序が指定されます。例えば、次の式

```
a < b <= c
```

は、次のように解釈されます。

```
(a < b) <= c
```

`a` の値が `b` の値よりも小さい場合は、最初の関係演算は、C では値 `1` を、C++ では `true` を生成します。その後で、コンパイラーは、値 `true` (または 1) を `c` の値と比較します (必要であれば、整数拡張が実行されます)。

等価演算子 == および非等価演算子 !=

等価演算子は、関係演算子と同様に、リレーションシップの妥当性について 2 つのオペランドを比較します。ただし、等価演算子には、関係演算子よりも低い優先順位が付けられます。次の表で、2 つの等価演算子を説明します。

演算子	使用法
==	左方オペランドの値が、右方オペランドの値と等価かどうかを示します。
!=	左方オペランドの値が、右方オペランドの値と等価でないかどうかを示します。

オペランドは両方とも算術型または列挙型を保持するか、同じ型を指すポインターである必要があります。あるいは、一方のオペランドがポインター型を保持し、もう一方のオペランドが void を指すポインターまたはヌル・ポインターである必要があります。

C 結果の型は `int` で、指定された関係が真であれば値 `1` を持ち、偽であれば `0` を持ちます。

C++ 結果の型は `bool` で、値 真 または 偽 を持ちます。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。

一方のオペランドがポインターで、もう一方のオペランドが値 0 の整数の場合、== 式は、ポインターのオペランドが NULL に評価される場合にだけ真になります。!= 演算子は、ポインター・オペランドが NULL に評価されない 場合に、真になります。

等価演算子を使用して、型が同じでも、同じオブジェクトに属さないメンバーを指すポインターを比較することもできます。次の式には、等価演算子と関係演算子の例が含まれています。

```
time < max_time == status < complete
letter != EOF
```

注: 等価演算子 (==) を、代入 (=) 演算子と混同しないでください。

次に例を示します。

if (x == 3)
x が 3 の場合真 (または 1) になります。このような等価テストでは、意図しない代入を防ぐために、演算子とオペランドの間にスペースを入れてコーディングする必要があります。

一方、

if (x = 3)
(x = 3) がゼロ以外の値 (3) になるので、真になります。また、この式では、3 が x に代入されます。

関連情報

- 119 ページの『単純代入演算子 =』

ビット単位 AND 演算子 &

& (ビット単位 AND) 演算子は、第 1 オペランドの各ビットと第 2 オペランドの対応するビットを比較します。ビットが両方とも 1 であれば、対応する結果のビットを 1 にセットします。 1 でなければ、対応する結果のビットを 0 にセットします。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。

ビット単位 AND 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位 AND 演算子を含む式の中でオペランドの再配置を行うことができます。

次の例では、16 ビットの 2 進数で表された、a と b の値、および、a & b の結果を示します。

a のビット・パターン	0000000001011100
b のビット・パターン	0000000000101110
a & b のビット・パターン	0000000000001100

注: ビット単位 AND (&) を、論理 AND (&&) 演算子と混同しないでください。例えば、次のような場合です。

1 & 4 は 0 になります。
一方、
1 && 4 は、真になります。

ビット単位排他 OR 演算子 ^

ビット単位排他 OR 演算子 (EBCDIC では、^ シンボルは ¬ シンボルで表します) は、第 1 オペランドの各ビットと第 2 オペランドの対応するビットを比較します。ビットが両方とも 1 か、両方とも 0 の場合、対応する結果ビットを 0 にセットします。それ以外の場合は、対応する結果ビットを 1 にセットします。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。結果は、左辺値ではありません。

ビット単位排他 OR 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位排他 OR 演算子を含む式の中でオペランドの再配置を行うことができます。 ^ 文字は、3 文字表記文字の ??' によって表されることに注意してください。

次の例では、16 ビットの 2 進数で表された a と b の値と、a ^ b の結果を示します。

a のビット・パターン	0000000001011100
b のビット・パターン	0000000000101110
a ^ b のビット・パターン	0000000001110010

関連情報

- 31 ページの『3 文字表記』

ビット単位包含 OR 演算子 |

| (ビット単位包含 OR) 演算子は、各オペランドの値 (2 進形式) を比較し、いずれかのオペランドのどのビットの値が 1 であるかを示すビット・パターンの値を生成します。両方のビットが 0 であると、その結果のビットは 0 になり、それ以外の結果は 1 になります。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。結果は、左辺値ではありません。

ビット単位包含 OR 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位包含 OR 演算子を含む式の中でオペランドの再配置を行うことができます。 | 文字は、3 文字表記文字の ??! によって表されることに注意してください。

次の例では、16 ビットの 2 進数で表された、a と b の値と、a | b の結果を示します。

a のビット・パターン	0000000001011100
b のビット・パターン	0000000000101110
a | b のビット・パターン	0000000001111110

注: ビット単位 OR (|) を、論理 OR (||) 演算子と混同しないでください。例えば、次のような場合です。

1 | 4 は 5 になります。

一方、

1 || 4 は、真になります。

関連情報

- 31 ページの『3 文字表記』

論理 AND 演算子 &&

&& (論理 AND) 演算子は、両方のオペランドが真であるかどうかを示します。

C 両オペランドともにゼロ以外の値が含まれている場合には、結果は値 1 になります。それ以外の場合には、結果は値 0 になります。その結果の型は、int です。オペランドは両方とも、算術型またはポインター型でなければなりません。各オペランドには、通常の算術変換が実行されます。

C++ 両方のオペランドの値が真である場合は、その結果の値は、真になります。そうでない場合は、結果の値は偽になります。両オペランドは、暗黙的に bool に変換されます。結果型は bool です。

& (ビット単位 AND) 演算子と異なり、&& 演算子では、オペランドは必ず左から右に評価されます。左方オペランドが 0 (または偽) になる場合、右方オペランドは評価されません。

次の例では、論理 AND 演算子を含む式が評価される方法を示します。

式	結果
1 && 0	偽または 0
1 && 4	真または 1
0 && 0	偽または 0

次の例では、論理 AND 演算子を使用して、ゼロによる割り算を行わないようにします。

```
(y != 0) && (x / y)
```

y != 0 が 0 (または 偽) に評価されるときは、式 x / y は評価されません。

注: 論理 AND 演算子 (&&) を、ビット単位 AND 演算子 (&) と混同しないでください。たとえば、次のとおりです。

1 && 4 は 1 (または真) になります。
一方、
1 & 4 は 0 になります。

論理 OR 演算子 ||

|| (論理 OR) 演算子は、いずれかのオペランドが真であるかどうかを示します。

C オペランドのいずれかにゼロ以外の値がある場合は、結果の値は 1 になります。そうでない場合は、結果の値は 0 になります。その結果の型は int です。オペランドは両方とも、算術型またはポインター型でなければなりません。各オペランドには、通常の算術変換が実行されます。

C++ オペランドのいずれかの値が真である場合は、その結果の値は、真になります。そうでない場合は、結果の値は偽になります。両オペランドは、暗黙的に bool に変換されます。結果型は bool です。

| (ビット単位包含 OR) 演算子と異なり、|| 演算子では、オペランドは必ず左から右に評価されます。左方オペランドがゼロ以外の値 (または真) である場合は、右方オペランドは評価されません。

次の例では、論理 OR 演算子を含む式が評価される方法を示します。

式	結果
1 || 0	真または 1
1 || 4	真または 1
0 || 0	偽または 0

次の例では、論理 OR 演算子を使用して、y を条件付きで増分します。

```
++x || ++y;
```

式 ++x が、ゼロ以外 (または真) の値になる場合、式 ++y は、評価されません。

注: 論理 OR 演算子 (||) を、ビット単位 OR 演算子 (|) と混同しないでください。たとえば、次のとおりです。

1 || 4 は 1 (または真) になります。

一方、

1 | 4 は 5 になります。

配列添え字演算子 [ ]

後置式とその後に続く [ ] (大括弧) 内の式が、配列のエレメントを指定します。大括弧内の式は、添え字と呼ばれます。配列の最初のエレメントの添え字はゼロです。

定義により、式 a[b] は、式 *((a) + (b)) と等価です (また、加算は結合であるので、b[a] とも等価です)。式 a と b の間で、一方は、型 T に対するポインターでなければなりません。そして、他方は整数型または列挙型を持っていなければなりません。配列添え字の結果は、左辺値になります。次の例は、このことを示しています。

```
#include <stdio.h>

int main(void) {
  int a[3] = { 10, 20, 30 };
  printf("a[0] = %d¥n", a[0]);
  printf("a[1] = %d¥n", 1[a]);
  printf("a[2] = %d¥n", *(2 + a));
  return 0;
}
```

上記の例の出力は、以下のとおりです。

```
a[0] = 10
a[1] = 20
a[2] = 30
```

C++ サブスクリプト演算子が必要とする、式の型に関する上述の制限だけでなく、サブスクリプト演算子とポインター演算の間の関係も、クラスの operator[] を多重定義する場合、適用されません。

各配列の最初のエレメントに、サブスクリプト 0 が付けられます。式 contract[35] は、配列 contract の 36 番目のエレメントを参照します。

多次元配列では、最も頻繁に右端サブスクリプトを増分することによって (保管場所の昇順で)、各エレメントを参照できます。

例えば、次のステートメントは、配列 code[4][3][6] の各エレメントに 100 を与えます。

```
for (first = 0; first < 4; ++first)
   {
   for (second = 0; second < 3; ++second)
      {
      for (third = 0; third < 6; ++third)
         {
         code[first][second][third] =
         100;
         }
      }
   }
```

関連情報

- 74 ページの『ポインター』
- 45 ページの『整数型』
- 103 ページの『左辺値と右辺値』
- 78 ページの『配列』
- 213 ページの『添え字の多重定義』
- 75 ページの『ポインター演算』

コンマ演算子 ,

コンマ式 には、任意の型の 2 つのオペランドがコンマで区切られて含まれており、左から右の結合順序が適用されます。左方オペランドは評価されますが、副次作用が生じる可能性があり、値がある場合、その値は廃棄されます。次に、右方オペランドが評価されます。コンマ式の結果の型および値は、通常の単項変換後の右方オペランドの型および値です。

C のみ

コンマ式の結果は左辺値ではありません。

C のみ の終り

C++ のみ。

C++ では、結果は、右方オペランドが左辺値であれば、左辺値です。次のステートメントは同じです。

```
r = (a,b,...,c);
a; b; r = c;
```

相違点として、コンマ演算子は、ループ制御式などの式のコンテキストに適したものとすることができます。

右方オペランドが左辺値である場合、複合式のアドレスを取ることができます。

```
&(a, b)
a, &b
```

C++ のみ。 の終り

コンマ演算子は結合するので、コンマで区切られた任意の数の式は単一の式を形成します。コンマ演算子を使用すると、副次式は左から右の順に評価されることが保証され、最後の副次式の値が式全体の値になります。次の例では、omega が 11 の場合は、式は delta を増分し、値 3 を alpha に代入します。

```
alpha = (delta++, omega % 4);
```

評価順序点は、第 1 オペランドの評価後に生じます。delta の値は廃棄されます。同様に、次の例の式

```
intensity++, shade * increment, rotate(direction);
```

の値は、次の式の値になります。

```
rotate(direction)
```

コンマ文字が使われているコンテキストによっては、あいまいさを避けるために括弧が必要な場合があります。例えば、次の関数

```
f(a, (t = 3, t + 2), c);
```

の引数は、値 a、値 5、および値 c の 3 個だけです。括弧を必要とするその他のコンテキストとしては、構造および共用宣言子内のフィールド長の式、列挙宣言子リストの列挙値の式、ならびに宣言および初期化指定子の初期化式があります。

上の例では、関数呼び出しの中の引数の式を区切るためにコンマが使用されています。このコンテキストでは、コンマを使用しても、関数の引数の評価順序 (左から右) は保証されません。

コンマ演算子の基本的な使用目的は、次のような状況で、副次作用をもたらすことです。

- 関数の呼び出し
- 反復ループへの入力または繰り返し
- 条件のテスト
- 副次作用は必要であるが、式の結果は今すぐに必要ではないその他の状況

次の表では、いくつかのコンマ演算子の使用例を示します。

ステートメント	効果
for (i=0; i<2; ++i, f());	for ステートメント では、i が増分され、反復のたびに f() が呼び出されます。
if (f(), ++i, i>1) { /* ... */ }	if ステートメントでは、関数 f() が呼び出され、変数 i が増分され、変数 i が値に対してテストされます。このコンマ式内の最初の 2 つの式は、式 i>1 の前に評価されます。最初の 2 つの式の結果に関係なく、3 番目の式が評価され、その結果によって、if ステートメントを処理するかどうかが決まります。
func((++a, f(a)));	func() への関数呼び出しでは、 a が増分され、結果の値が関数 f() に渡され、 f() の戻り値が f() に渡されます。関数 func() には、引数が 1 つだけ渡されます。これは、関数引数のリスト内で、コンマ式が括弧で囲まれているからです。

メンバーを指すポインター演算子 .* ->* (C++ のみ)

メンバーを指すポインター演算子には、.* と ->* の 2 つがあります。

クラス・メンバーを指すポインターを間接参照するには、.* 演算子を使用します。第 1 オペランドは、クラス型でなければなりません。第 1 オペランドの型がクラス型 T、またはクラス型 T から派生したクラスの場合は、第 2 オペランドはクラス型 T のメンバーを指すポインターでなければなりません。

クラス・メンバーを指すポインターを間接参照するには、->* 演算子を使用します。第 1 オペランドは、クラス型を指すポインターでなければなりません。第 1 オペランドの型がクラス型 T を指すポインター、またはクラス型 T から派生したクラスを指すポインターの場合は、第 2 オペランドはクラス型 T のメンバーを指すポインターでなければなりません。

.* および ->* 演算子は、第 2 オペランドを第 1 オペランドにバインドします。結果は、第 2 オペランドで指定された型のオブジェクトまたは関数になります。

.* または ->* の結果が関数の場合は、結果を ( ) (関数呼び出し) 演算子のオペランドとしてだけ使用できます。第 2 オペランドが左辺値の場合は、.* または ->* の結果は左辺値になります。

関連情報

- 229 ページの『クラス・メンバー・リスト』
- 234 ページの『メンバーへのポインター』

条件式

条件式 は、C++ では暗黙的にタイプ bool へ変換される条件 (*operand*$_1$)、条件が true に評価される場合に評価される式 (*operand*$_2$)、および条件が値 false を持っている場合に評価される式 (*operand*$_3$) を含む複合式です。

条件式には、2 つの部分で構成される 1 つの演算子があります。 ? 記号は、条件の後に続き、 : 記号は 2 つのアクション式の間に使用されます。 ? と : の間の式は、すべて 1 つの式として扱われます。

第 1 オペランドは、スカラー型を持つ必要があります。第 2 オペランドと第 3 オペランドの型は、次のいずれかでなければなりません。

- 算術型
- 互換ポインター、構造体、または共用体型
- void

第 2 オペランドと第 3 オペランドは、ポインターまたはヌル・ポインター定数であってもかまいません。

2 つのオブジェクトが同じ型を持つが、必ずしも同じ型の修飾子 (volatile または const) でない場合、この 2 つのオブジェクトは互換性があります。ポインター・オブジェクトが同じ型を持つか、 void を指すポインターの場合は、これらのポインター・オブジェクトには互換性があります。

第 1 オペランドが評価され、その値によって第 2 オペランドまたは第 3 オペランドを評価するかどうかが判別されます。

- 値が真の場合は、第 2 オペランドが評価されます。
- 値が偽の場合は、第 3 オペランドが評価されます。

結果は、第 2 オペランドまたは第 3 オペランドの値になります。

2 番目または 3 番目の式が算術型になる場合、値に通常の算術変換を実行します。次の表は、第 2 オペランドと第 3 オペランドの型により結果の型がどのように決まるかを示します。

条件式は、第 1 オペランドと第 3 オペランドについては右から左の結合順序が適用されます。左端のオペランドが最初に評価され、次に、残りの 2 つのオペランドのいずれか 1 つだけが評価されます。次の式は、同等です。

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

C の条件式の型

C のみ

C では、条件式は左辺値またはその結果ではありません。

表 19. C の条件式のオペランドおよび結果の型

一方のオペランドの型	もう一方のオペランドの型	結果の型
算術	算術	通常の算術変換後の算術型
構造体または共用体型	互換構造体または共用体型	両方のオペランドにすべての修飾子が付く構造体または共用体型
void	void	void
互換型を指すポインター	互換型を指すポインター	型に指定されたすべての修飾子が付く型を指すポインター
型を指すポインター	NULL ポインター (定数 0)	型を指すポインター
オブジェクトまたは不完全型を指すポインター	void を指すポインター	型に指定されたすべての修飾子が付く void を指すポインター

C のみ の終り

C++ の条件式の型

C++ のみ。

C++ では、条件式は、その型が void でなく、その結果が左辺値の場合、有効な左辺値となります。

表 20. C++ の条件式のオペランドおよび結果の型

一方のオペランドの型	もう一方のオペランドの型	結果の型
型への参照	型への参照	通常の参照変換後の参照
クラス T	クラス T	クラス T
クラス T	クラス X	型変換が存在する場合のクラス型。可能な型変換が複数ある場合は、結果は不確定になります。
throw 式	その他 (型、ポインター、参照)	throw 式でない式の型

C++ のみ。 の終り

条件式の例

次の式では、値が大きい方の変数が y か z かを判別し、大きい方の値を変数 x に代入します。

```
x = (y > z) ? y : z;
```

次に等価なステートメントを示します。

```
if (y > z)
   x = y;
else
   x = z;
```

次の式では、関数 printf を呼び出し、c が数字に評価される場合に、この関数が、変数 c の値を受け取ります。そうでない場合は、printf は文字定数 'x' を受け取ります。

```
printf(" c = %c¥n", isdigit(c) ? c : 'x');
```

条件式の最後のオペランドに代入演算子が含まれる場合は、小括弧を使用して、式が正しい評価を行うようにします。例えば、次の式では、= 演算子には ?: 演算子より高い優先順位が付けられます。

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

このコンパイラーは、次のように括弧で囲まれているように解釈されるので、エラーになります。

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

つまり、k が代入式 k = j 全体としてではなく、第 3 オペランドとして扱われます。

j の値を k に代入するのは、i == 7 が偽のときで、最後のオペランドを小括弧で囲みます。

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

キャスト式

キャスト演算子は、*明示的型変換* のために使用されます。キャスト演算子は、式の値を指定された型に変換します。

以下のキャスト演算子がサポートされています。

- 『キャスト演算子 ()』
- 135 ページの『static_cast 演算子 (C++ のみ)』
- 136 ページの『reinterpret_cast 演算子 (C++ のみ)』
- 137 ページの『const_cast 演算子 (C++ のみ)』
- 139 ページの『dynamic_cast 演算子 (C++ のみ)』

キャスト演算子 ()

キャスト式構文

```
►►—(—type—)—expression——————————————————————————————►◄
```

C この演算子の結果は左辺値ではありません。 C++ この演算子の結果は、 *type* が参照である場合に左辺値になります。それ以外のすべての場合には、結果は右辺値になります。

以下の例では、キャスト演算子を使用して、サイズ 10 の整数配列を動的に作成しています。

```
#include <stdlib.h>

int main(void) {
```

```
   int* myArray = (int*) malloc(10 * sizeof(int));
   free(myArray);
   return 0;
}
```

malloc ライブラリー関数は、その引数のサイズのオブジェクトを保持するメモリーを指すvoid ポインターを戻します。ステートメント int* myArray = (int*) malloc(10 * sizeof(int)) は、以下のことを行います。

- 10 個の整数を保持できるメモリーを指す void ポインターを作成します。
- その void ポインターを、キャスト演算子を使用して整数ポインターに変換します。
- その整数ポインターを myArray に代入します。配列の名前は、配列の初期のエレメントを指すポインターと同じなので、myArray は、malloc() への呼び出しによって作成されたメモリーに保管されている、10 個の整数の配列です。

C++ のみ。

C++ では、キャスト式で以下も使用できます。

- 関数スタイル・キャスト
- C++ 変換演算子 (static_cast など)

関数スタイル表記は、*expression* の値を型 *type* に変換します。

expression(*type*)

以下の例では、C スタイル・キャスト、C++ 関数スタイル・キャスト、および C++ キャスト演算子で同じ値をキャストしています。

```
#include <iostream>
using namespace std;

int main() {
  float num = 98.76;
  int x1 = (int) num;
  int x2 = int(num);
  int x3 = static_cast<int>(num);

  cout << "x1 = " << x1 << endl;
  cout << "x2 = " << x2 << endl;
  cout << "x3 = " << x3 << endl;
}
```

上記の例の出力は、以下のとおりです。

```
x1 = 98
x2 = 98
x3 = 98
```

整数 x1 には、C スタイル・キャストを使用して明示的に num を int に変換した値が代入されます。整数 x2 には、関数スタイル・キャストを使用して変換された値が代入されます。整数 x3 には、static_cast 演算子を使用して変換された値が代入されます。

キャストは、そのオペランドが左辺値の場合、有効な左辺値です。以下の単純な代入式では、最初に右側が、指定された型に変換され、次に内側の左側の式の型に変換され、結果が保管されます。その値が、指定された型に変換し直されて、それが代入の値になります。次の例では、i は char * 型です。

```
(int)i = 8     // This is equivalent to the following expression
(int)(i = (char*) (int)(8))
```

キャストに適用される複合代入演算の場合、複合代入の算術演算子は、キャスト結果の型を使って実行され、それから、単純な代入の場合と同じように進みます。次の式は同等です。また、i の型は char * です。

```
(int)i += 8     // This is equivalent to the following expression
(int)(i = (char*) (int)((int)i = 8))
```

C++ の場合は、キャスト式のオペランドには、クラス型を指定できます。オペランドにクラス型が指定された場合は、クラスにユーザー定義の型変換関数がある任意の型にキャストすることができます。これらのキャストは、ターゲット型がクラスであれば、コンストラクターを呼び出すことができますし、ソース型がクラスであれば、型変換関数を呼び出すことができます。これらのキャストは、両方の条件が該当する場合は、不明瞭になります。

C++ のみ。 の終り

関連情報

- 73 ページの『型名』
- 292 ページの『変換関数』
- 290 ページの『変換コンストラクター』
- 103 ページの『左辺値と右辺値』

static_cast 演算子 (C++ のみ)

*static_cast 演算子*は、与えられた式を指定された型に変換します。

static_cast 演算子構文

```
►►—static_cast—<—Type—>—(—expression—)——————————————————►◄
```

次に static_cast 演算子の例を示します。

```
#include <iostream>
using namespace std;

int main() {
  int j = 41;
  int v = 4;
  float m = j/v;
  float d = static_cast<float>(j)/v;
  cout << "m = " << m << endl;
  cout << "d = " << d << endl;
}
```

上記の例の出力は、以下のとおりです。

```
m = 10
d = 10.25
```

この例では、m = j/v; は、型 int の答えを作成します。なぜなら、j と v の両方とも整数であるからです。逆に、d = static_cast<float>(j)/v; は、型 float の答えを作成します。 static_cast 演算子は、変数 j を、型 float に変換します。これによって、コンパイラーは、型 float の答えを持つ割り算を生成できます。すべての static_cast 演算子は、コンパイル時に解決します。そして、どの const または volatile 修飾子も除去しません。

static_cast 演算子をヌル・ポインターに適用すると、その演算子は、ターゲット型のヌル・ポインター値に変換されます。

A が B の基底クラスであれば、型 A のポインターを型 B のポインターに、明示的に変換できます。 A が B の基底クラスでなければ、コンパイラー・エラーになります。

以下が真であれば、型 A の左辺値を、型 B& にキャストすることができます。

- A は、B の基底クラスである。
- 型 A のポインターを、型 B のポインターへ変換できる。
- 型 B は、型 A と同じまたはより大きい、const または volatile 修飾子を持っている。
- A は、B の仮想基底クラスではない。

結果は、型 B の左辺値になります。

メンバー型を指すポインターは、別のメンバー型を指すポインターへ明示的に変換できます。ただし、両方の型が、同じクラスのメンバーを指すポインターである場合に限ります。明示的変換のこの形式は、メンバー型を指すポインターが、別のクラスからのものである場合に行われることがあります。ただし、クラス型のうちの 1 つは、他のものから派生していなければなりません。

関連情報

- 289 ページの『ユーザー定義の型変換』

reinterpret_cast 演算子 (C++ のみ)

reinterpret_cast 演算子 は、無関係の型の間の変換を扱います。

reinterpret_cast 演算子構文

```
▶▶—reinterpret_cast—<—Type—>—(—expression—)——————————————————▶◀
```

reinterpret_cast 演算子は、引数と同じビット・パターンを持っている、新規の型の値を作成します。 const または volatile 修飾をキャストすることはできません。以下の変換を明示的に実行することができます。

- ポインターから、それを保持するのに十分に大きい任意の整数型へ
- 整数値または列挙型から、ポインターへ
- 関数を指すポインターから、別の型の関数を指すポインターへ
- オブジェクトを指すポインターから、別の型のオブジェクトを指すポインターへ
- メンバーを指すポインターから、別のクラスまたは型のメンバーを指すポインターへ。ただし、メンバーの型が両方の関数型またはオブジェクト型である場合。

ヌル・ポインター値は、宛先型のヌル・ポインター値に変換されます。

型 T の左辺値式およびオブジェクト x が与えられているとして、以下の 2 つの変換は同義です。

- `reinterpret_cast<T&>(x)`
- `*reinterpret_cast<T*>(&x)`

C++ は、C スタイル・キャストもサポートします。明示的なキャストの 2 つのスタイルは、構文は異なりますが同じセマンティクスを持っています。ポインターの一方の型をポインターの非互換型であるとする、

どちら側からの再解釈も、通常無効です。 reinterpret_cast 演算子は、他の名前付きキャスト演算子と同様に、C スタイル・キャストよりも容易にスポットされ、明示的キャストを可能にする、強くタイプされた言語の矛盾をハイライトします。

C++ コンパイラーは、全部ではないがほとんどの違反を検出し、修正します。プログラムがコンパイルされても、そのソース・コードが、完全には正しくない場合があるということを覚えておくことは重要です。一部のプラットフォームでは、パフォーマンスの最適化が、標準別名割り当て規則に厳格に準拠して記述されます。 C++ コンパイラーは、型ベースの別名割り当て違反についてヘルプしようと試みるけれども、すべての可能なケースを検出することはできません。

次の例は、別名割り当て規則に違反しています。しかし、C++ または K&R C で、最適化されずにコンパイルされると、期待通りに実行されます。またそれは、C++ で、最適化して正常にコンパイルできますが、必ずしも期待通りには実行されません。問題の 7 行目は、x の古いまたは未初期化の値を印刷してしまいます。

```
1  extern int y = 7.;
2
3  int main() {
4       float x;
5       int i;
6       x = y;
7       i = *(int *) &x;
8       printf("i=%d. x=%f.¥n", i, x);
9  }
```

次のコードの例は、キャストが 2 つの異なるファイルにまたがっているので、コンパイラーが検出すらもできない、誤ったキャストを含んでいます。

```
1  /* separately compiled file 1 */
2      extern float f;
3      extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */
4
5  /* separately compiled file 2 */
6      extern float f;
7      extern int * int_pointer_to_f;
8      f = 1.0;
9      int i = *int_pointer_to_f;           /* no suspicious cast but wrong */
```

8 行目において、`int i = *int_pointer_to_f` がロード元としている同じオブジェクトを、 `f = 1.0` が保管先としていることを、コンパイラーが知る方法はありません。

関連情報

- 289 ページの『ユーザー定義の型変換』

const_cast 演算子 (C++ のみ)

const_cast 演算子 は、const または volatile 修飾子を、型へ追加または型から除去するために使用されます。

const_cast 演算子構文

```
►►—const_cast—<—Type—>—(—expression—)——————————————————————————►◄
```

Type と *expression* の型は、それらの const および volatile 修飾子に関してのみ異なります。それらのキャストは、コンパイル時に解決されます。単一の const_cast 式で、任意の数の const または volatile 修飾子を追加または除去できます。

const_cast 式の結果は、*Type* が参照型でない限り、右辺値です。この場合、結果は左辺値です。

型は const_cast 内では定義できません。

以下は、const_cast 演算子の使用法を示したものです。

```
#include <iostream>
using namespace std;

void f(int* p) {
  cout << *p << endl;
}

int main(void) {
  const int a = 10;
  const int* b = &a;

  // Function f() expects int*, not const int*
  //   f(b);
  int* c = const_cast<int*>(b);
  f(c);

  // Lvalue is const
  //  *b = 20;

  // Undefined behavior
  //  *c = 30;

  int a1 = 40;
  const int* b1 = &a1;
  int* c1 = const_cast<int*>(b1);

  // Integer a1, the object referred to by c1, has
  // not been declared const
  *c1 = 50;

  return 0;
}
```

コンパイラーは、関数呼び出し f(b) を許可しません。関数 f() は、const int ではなく int を指すポインターを期待します。ステートメント int* c = const_cast<int>(b) は、a の const 修飾なしに a を指すポインター c を戻します。 const_cast を使用してオブジェクトの const 修飾を除去するこのプロセスは、 *casting away constness* と呼ばれています。したがって、コンパイラーは、関数呼び出し f(c) を行うことができます。

コンパイラーは、代入 *b = 20 を許可しません。なぜなら、b は、型 const int のオブジェクトを指すからです。コンパイラーは *c = 30 を許可します。しかし、このステートメントの振る舞いは未定義です。 const として明示的に宣言されているオブジェクトの constness をキャストし、それを変更しようとする場合、結果は予期できません。

しかし、const として明示的に宣言されていないオブジェクトの constness をキャストする場合、それを安全に変更することができます。上記の例では、b1 が参照しているオブジェクトは、const と宣言されていません。しかし、このオブジェクトを b1 によって変更することはできません。 b1 の constness をキャストし、それが参照している値を変更できます。

関連情報

- 62 ページの『型修飾子』

dynamic_cast 演算子 (C++ のみ)

dynamic_cast 演算子は、実行時に型変換を実行します。dynamic_cast 演算子によって、基底クラスへのポインターが派生クラスへのポインターへと確実に変換されるか、または基底クラスを参照する左辺値が派生クラスへの参照へと確実に変換されます。それによって、プログラムはクラス階層を安全に使用することができます。この演算子と typeid 演算子は、C++ における RTTI (runtime type information) サポートを提供します。

式 dynamic_cast<T>(v) は、式 v を型 T に変換します。型 T は、完全クラス型を指すポインターまたは参照、あるいは void を指すポインターでなければなりません。 T がポインターであって、dynamic_cast 演算子が失敗した場合、演算子は、型 T のヌル・ポインターを戻します。 T が参照であって、dynamic_cast 演算子が失敗した場合、演算子は、例外 std::bad_cast を throw します。このクラスは、標準ライブラリー・ヘッダー <typeinfo> の中で検出することができます。

dynamic_cast 演算子は、実行時型情報 (RTTI) の生成を要求します。これは、コンパイラー・オプションによってコンパイル時に明示的に指定する必要があります。

T が void ポインターの場合、dynamic_cast は、v が指すオブジェクトの開始アドレスを戻します。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
  virtual ~A() { };
};

struct B : A { };

int main() {
  B bobj;
  A* ap = &bobj;
  void * vp = dynamic_cast<void *>(ap);
  cout << "Address of vp  : " << vp << endl;
  cout << "Address of bobj: " << &bobj << endl;
}
```

この例の出力は、次の出力に似ています。 vp および &bobj の両方とも同じアドレスを参照します。

```
Address of vp  : SPP:0000 :1aefQPADEV0001TSTUSR  369019:220:0:6c
Address of bobj: SPP:0000 :1aefQPADEV0001TSTUSR  369019:220:0:6c
```

dynamic_cast 演算子の主目的は、型が安全な *downcasts* を実行することです。ダウン・キャストは、クラス A がクラス B の基底クラスである場合に、クラス A へのポインターまたは参照を、クラス B へのポインターまたは参照に変換することを指します。ダウン・キャストの問題は、型 A* のポインターが、A から派生されているクラスの任意のオブジェクトを指すことができ、また指す必要がある点にあります。dynamic_cast 演算子を使用すると、クラス A のポインターをクラス B のポインターに変換する場合に、A が指すオブジェクトが、クラス B または B から派生されるクラスに確実に属するようになります。

以下の例は、dynamic_cast 演算子の使用法を示したものです。

```
#include <iostream>
using namespace std;

struct A {
  virtual void f() { cout << "Class A" << endl; }
};

struct B : A {
  virtual void f() { cout << "Class B" << endl; }
```

```
};

struct C : A {
  virtual void f() { cout << "Class C" << endl; }
};

void f(A* arg) {
  B* bp = dynamic_cast<B*>(arg);
  C* cp = dynamic_cast<C*>(arg);

  if (bp)
    bp->f();
  else if (cp)
    cp->f();
  else
    arg->f();
};

int main() {
  A aobj;
  C cobj;
  A* ap = &cobj;
  A* ap2 = &aobj;
  f(ap);
  f(ap2);
}
```

上記の例の出力は、以下のとおりです。

```
Class C
Class A
```

関数 f() は、ポインター arg が、型 A、B、または C のオブジェクトを指すかどうかを判別します。関数は、dynamic_cast 演算子を使用して、arg を、型 B のポインターへ、次に型 C のポインターに変換しようと試みることによって、この判別を行います。 dynamic_cast 演算子が正常に行われると、arg によって表されるオブジェクトを指すポインターを戻します。 dynamic_cast が失敗すると、0 が戻されます。

downcast は、ポリモアフィック・クラスにおいてのみ、dynamic_cast 演算子を使用して、実行することができます。上記の例では、クラス A は、仮想関数を持っているので、すべてのクラスはポリモアフィックです。 dynamic_cast 演算子は、ポリモアフィック・クラスから生成された実行時の型情報を使用します。

関連情報

- 253 ページの『派生』
- 289 ページの『ユーザー定義の型変換』

複合リテラル式

複合リテラル とは、初期化指定子リストで与えられる値を持ち、かつ名前の付けられていないオブジェクトを指定するための接尾辞式です。 C99 言語フィーチャーを使用することで、一時変数を使用することなく、パラメーターを関数に渡すことができます。これは、集合体型 (配列、構造体、共用体) の定数を指定する際に、そのような型のインスタンスが 1 つしか必要でない場合に役立ちます。

複合リテラルの構文はキャスト式の構文に似ています。ただし、複合リテラルが左辺値であるのに対し、キャスト式の結果は左辺値ではありません。さらに、キャストはスカラー型または void にしか変換できませんが、複合リテラルは指定された型のオブジェクトになります。

複合リテラル構文

type_name は、ユーザー定義型を含む任意のデータ型にすることができます。サイズが不明な配列にすることもできますが、可変長配列にすることはできません。サイズが不明の配列型の場合、そのサイズは初期化指定子リストによって決まります。

以下の例では、2 つの整数メンバーを含む `point` 型の定数構造体変数を、関数 `drawline` に渡しています。

```
drawline((struct point){6,7});
```

複合リテラルが関数本体の外側で使用されている場合には、初期化指定子リストは定数式で構成されている必要があり、また無名オブジェクトは静的ストレージ期間を持ちます。複合リテラルが関数本体内で使用されている場合には、初期化指定子リストは定数式で構成されている必要はなく、また無名オブジェクトは自動ストレージ期間を持ちます。

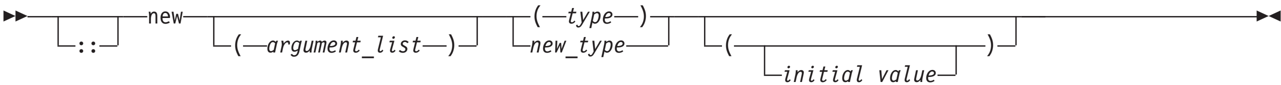
関連情報

- 24 ページの『ストリング・リテラル』

new 式 (C++ のみ)

`new` 演算子は、動的ストレージ割り当てを提供します。

new 演算子構文



スコープ・レゾリューション演算子 (`::`) に `new` を接頭部として付けると、global `operator new()` が使用されます。 *argument_list* を指定した場合は、その *argument_list* に対応する、多重定義された `new` 演算子が使われます。 *type* は、既存のインクルード型またはユーザー定義の型です。 *new_type* は、まだ定義されていない型で、型指定子と宣言子をインクルードすることができます。

`new` 演算子を含む割り当て式は、作成されたオブジェクトのフリー・ストレージを検出するために使われます。 *new* 式 は、作成されたオブジェクトを指すポインターを戻し、これを使用してオブジェクトを初期化することができます。オブジェクトが配列の場合は、最初のエレメントを指すポインターが戻されます。

関数型、`void`、または不完全クラス型はオブジェクトの型ではないので、`new` 演算子を使用してこれらの型を割り振ることはできません。ただし、`new` 演算子を使用して、関数を指すポインターを割り振ることはできます。`new` 演算子を使用して、参照を作成することはできません。

作成されるオブジェクトが配列の場合は、最初の次元だけが汎用式になります。以降のすべての次元は、整数定数式でなければなりません。最初の次元は、既存の *type* が使われているときにも、汎用式になります。 `new` 演算子を使用して、ゼロ境界付きの配列を作成できます。次に例を示します。

```
char * c = new char[0];
```

この場合、固有なオブジェクトへのポインターが戻されます。

operator new() または operator new[]() を使用して作成されたオブジェクトは、operator delete() または operator delete[]() が、オブジェクトのメモリーを割り振り解除するために呼び出されるまで、存在します。 delete 演算子またはデストラクターは、new を使用して作成されたオブジェクトで、プログラムの終了の前に明示的に割り振り解除されていないものに対して、暗黙的に呼び出されることはありません。

小括弧が new 型内で使用される場合、構文エラーを避けるために、その new 型も小括弧で囲む必要があります。

次の例では、関数を指すポインターの配列用ストレージが割り当てられます。

```
void f();
void g();

int main(void)
{
      void (**p)(), (**q)();
      // declare p and q as pointers to pointers to void functions
      p = new (void (*[3])());
      // p now points to an array of pointers to functions
      q = new void(*[3])(); // error
      // error - bound as 'q = (new void) (*[3])();'
      p[0] = f;  // p[0] to point to function f
      q[2] = g;  // q[2] to point to function g
      p[0]();    // call f()
      q[2]();    // call g()
      return (0);
}
```

ただし、2 番目の new の使用では、 q = (new void) (*[3])() のように間違ったバインディングになります。

作成されるオブジェクトの型には、クラス宣言、列挙宣言、const 型、または volatile 型を含めることはできません。const または volatile オブジェクトを指すポインターは含めることができます。

例えば、const char* は使用できますが、char* const は使用できません。

関連情報

- 192 ページの『割り振りおよび割り振り解除関数 (C++ のみ)』

配置構文

追加引数は、*argument_list* を使用することによって、new に引数を追加することができます (*配置構文* とも呼ばれます)。配置引数を使う場合は、それらの引数が指定された operator new() または operator new[]() が宣言されていなければなりません。次に例を示します。

```
#include <new>
using namespace std;

class X
{
public:
      void* operator new(size_t,int, int){ /* ... */ }
};

// ...
```

```
int main ()
{
      X* ptr = new(1,2) X;
}
```

配置構文は通常、グローバル配置 new 関数を呼び出すときに使用されます。グローバル配置 new 関数は、配置 new 式の中で配置引数によって指定されたロケーションにあるオブジェクト (1 つまたは複数) を初期化します。グローバル配置 new 関数はそれ自体にメモリーを割り当てることはしないので、このロケーションには他の方法で事前に割り当てられていたストレージをアドレッシングしなければなりません。次の例では、new(whole) X(8);、new(seg2) X(9);、または new(seg3) X(10); を呼び出しても新規メモリーは割り当てられません。代わりに、コンストラクター X(8)、X(9)、および X(10) を呼び出して、バッファー whole に割り当てられたメモリーを再初期化します。

配置 new はメモリーを割り振らないので、配置構文で作成されたオブジェクトを割り振り解除するために delete を使わないでください。削除できるのは、メモリー・プール全体だけです (delete whole)。次の例では、メモリー・バッファーを残しておいて、デストラクターを明示的に呼び出すことにより、そこに保管されていたオブジェクトを破棄することができます。

```
#include <new>
class X
{
   public:
      X(int n): id(n){ }
      ~X(){ }
   private:
      int id;
      // ...
};

int main()
{
   char* whole = new char[ 3 * sizeof(X) ];  // a 3-part buffer
   X * p1 = new(whole) X(8);                 // fill the front
   char* seg2 = &whole[ sizeof(X) ];         // mark second segment
   X * p2 = new(seg2) X(9);                  // fill second segment
   char* seg3 = &whole[ 2 * sizeof(X) ];     // mark third segment
   X * p3 = new(seg3) X(10);                 // fill third segment

   p2->~X();   // clear only middle segment, but keep the buffer
   // ...
   return 0;
}
```

配置 new 構文は、コンストラクターではなく、割り振りルーチンにパラメーターを渡すためにも使用できます。

関連情報

- 145 ページの『delete 式 (C++ のみ)』
- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』
- 277 ページの『コンストラクターとデストラクターの概要』

new 演算子で作成されたオブジェクトの初期化

new 演算子を使用して作成されたオブジェクトは、いくつかの方法で初期化できます。クラス以外のオブジェクトまたはコンストラクターなしのクラス・オブジェクトの場合は、(式) または () を指定することによって、*new 初期化指定子* の式が new 式に提供されます。次に例を示します。

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

クラスにデフォルトのコンストラクターが指定されていない場合は、そのクラスのオブジェクトが割り振られる際に new 初期化指定子を指定する必要があります。new 初期化指定子の引数は、コンストラクターの引数と一致している必要があります。

配列に初期化指定子を指定することはできません。クラスにデフォルトのコンストラクターが指定されている場合のみ、クラス・オブジェクトの配列を初期化することができます。コンストラクターを呼び出して、各配列エレメント (クラス・オブジェクト) を初期化します。

new 初期化指定子を使用した初期化は、new がストレージを正常に割り振った場合にのみ実行されます。

関連情報

- 277 ページの『コンストラクターとデストラクターの概要』

new 割り振り失敗の処理

new 演算子が新しいオブジェクトを作成すると、この演算子は、operator new() または operator new[]() 関数を呼び出して、必要なストレージを獲得します。

new は、新しいオブジェクトを作成するためのストレージを割り当てできないときには、set_new_handler() への呼び出しによって *new* ハンドラー 関数 (インストールされている場合) を呼び出します。std::set_new_handler() 関数は、ヘッダー <new> で宣言されます。この関数を使用して、定義済みの new ハンドラーまたはデフォルトの new ハンドラーを呼び出します。

new ハンドラーは、次のうちのどれかを実行する必要があります。

- メモリー割り振りのためにさらにストレージを取得し、それから戻ります。
- 型 std::bad_alloc の例外、または std::bad_alloc から派生したクラスをスローします。
- abort() または exit() のどちらかを呼び出します。

set_new_handler() 関数はプロトタイプを持ちます。

```
typedef void(*PNH)();
PNH set_new_handler(PNH);
```

set_new_handler() は、引数として関数 (new handler) を指すポインターを取りますが、この関数は引数がなく void を戻します。set_new_handler() は、直前の new ハンドラー関数を指すポインターを戻します。

独自の set_new_handler() 関数を指定しない場合は、 new は、型 std::bad_alloc の例外をスローします。

次のプログラムでは、new 演算子がストレージを割り当てできない場合に、set_new_handler() を使用してメッセージを戻す方法を示します。

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

void no_storage()
{
    std::cerr << "Operator new failed: no storage is
    available.\n";
```

```
      std::exit(1);
}
int main(void)
{
     std::set_new_handler(&no_storage);
   // Rest of program ...
}
```

new がストレージを割り当てできないためにプログラムが失敗した場合は、プログラムは次のメッセージを出して終了します。

```
  Operator new failed:
no storage is available.
```

delete 式 (C++ のみ)

delete 演算子は、オブジェクトに関連付けられたメモリーを割り当て解除することによって、new を使用して作成されたオブジェクトを破棄します。

delete 演算子には、void 戻り型があります。

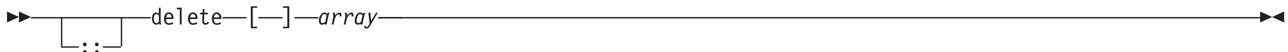
delete 演算子構文

```
>>-+----+--delete--object_pointer-----------------------------><
   '-::-'
```

delete のオペランドは、new によって戻されるポインターでなければなりません。定数を指すポインターであってはなりません。ヌル・ポインターを削除しても影響はありません。

delete[] 演算子は、new[] 演算子を使用して作成された配列オブジェクトに割り当てられたストレージを解放します。delete 演算子は、new を使用して作成された個々のオブジェクトに割り当てられたストレージを解放します。

delete[] 演算子構文



delete によって配列オブジェクトを削除した結果は、未定義です。 delete[] によって個々のオブジェクトを削除する場合も同じです。配列の次元は、delete[] で指定する必要はありません。

削除されたオブジェクトまたは配列へアクセスしようとする試みの結果は、未定義です。

デストラクターがクラスに定義されている場合は、delete によってそのデストラクターが呼び出されます。デストラクターがあるかどうかに関係なく、delete は、クラスの関数 operator delete() がある場合は、この関数呼び出しによって指し示されたストレージを解放します。

次の場合には、グローバル ::operator delete() が使用されます。

- クラスに operator delete() がない場合
- オブジェクトがクラス以外の型の場合
- ::delete 式によってオブジェクトが削除される場合

次の場合には、グローバル ::operator delete[]() が使用されます。

- クラスに operator delete[]() がない場合
- オブジェクトがクラス以外の型の場合
- ::delete[] 式によってオブジェクトが削除される場合

デフォルトのグローバル operator delete() だけが、デフォルトのグローバル operator new() によって割り当てられたストレージを解放することができます。デフォルトのグローバル operator delete[]() のみが、デフォルトのグローバル operator new[]() によって配列に割り当てられたストレージを解放することができます。

関連情報

- 277 ページの『コンストラクターとデストラクターの概要』
- 48 ページの『void 型』

throw 式 (C++ のみ)

throw 式は、C++ の例外ハンドラーに例外をスロー (throw) するために使われます。 throw 式は、void 型です。

関連情報

- 331 ページの『第 16 章 例外処理 (C++ のみ)』
- 48 ページの『void 型』

演算子優先順位と結合順序

優先順位 と*結合順序* という 2 つの演算子の特性によって、オペランドが演算子とグループ化される方法が決まります。優先順位は、型が異なる演算子をオペランドとグループ化させるときの優先順位です。結合順序は、オペランドを、同じ優先順位の演算子にグループ化するときの左から右、または右から左の順序です。演算子の優先順位は、より高いまたは低い優先順位の他の演算子がある場合にのみ、意味があります。より高い優先順位の演算子を持つ式が、最初に評価されます。小括弧を使用して、強制的にオペランドをグループ化することができます。

例えば、次のステートメントでは、値 5 は、= 演算子の右から左への結合順序を使用して、a と b の両方に代入されます。最初に値 c が b に代入され、次に値 b が a に代入されます。

```
b = 9;
c = 5;
a = b = c;
```

副次式の評価順序が指定されていないので、小括弧を使用して、演算子付きのオペランドのグループ化を明示的に強制することができます。

次の式では、

```
a + b * c / d
```

優先順位により、+ 演算の前に、* および / 演算が実行されます。結合順序により、b は d によって除算される前に、c と乗算されます。

次の表に、C および C++ 言語の演算子を優先順位の順序でリストし、各演算子の結合順序の方向を示します。同位にある演算子には、同じ優先順位が付けられています。

表 21. 後置演算子の優先順位と結合順序

ランク	右結合 ？	演算子関数	使用法
1	はい	C++ グローバル・スコープ・レゾリューション	:: *name_or_qualified name*
1		C++ クラスまたはネーム・スペース・スコープ・レゾリューション	*class_or_namespace* :: *member*
2		メンバー選択	*object* . *member*
2		メンバー選択	*pointer* -> *member*
2		サブスクリプト	*pointer* [*expr*]
2		関数呼び出し	*expr* (*expr_list*)
2		値生成	*type* (*expr_list*)
2		後置増分	*lvalue* ++
2		後置減分	*lvalue* --
2	はい	C++ 型の識別	typeid (*type*)
2	はい	C++ 実行時の型識別	typeid (*expr*)
2	はい	C++ コンパイル時にチェックされる型変換	static_cast < *type* > (*expr*)
2	はい	C++ 実行時にチェックされる型変換	dynamic_cast < *type* > (*expr*)
2	はい	C++ チェックされない型変換	reinterpret_cast < *type* > (*expr*)
2	はい	C++ `const` の変換	const_cast < *type* > (*expr*)

表 22. 単項演算子の優先順位と結合順序

ランク	右結合 ？	演算子関数	使用法
3	はい	オブジェクトのサイズ (バイト)	sizeof *expr*
3	はい	型のサイズ (バイト)	sizeof (*type*)
3	はい	接頭部増分	++ *lvalue*
3	はい	接頭部減分	-- *lvalue*
3	はい	ビット単位否定	~ *expr*
3	はい	not	! *expr*
3	はい	単項負	- *expr*
3	はい	単項正	+ *expr*
3	はい	アドレス	& *lvalue*
3	はい	間接または参照解除	* *expr*
3	はい	C++ 作成 (メモリーの割り振り)	new *type*
3	はい	C++ 作成 (メモリーの割り振りと初期化)	new *type* (*expr_list*) *type*

表 22. 単項演算子の優先順位と結合順序 (続き)

ランク	右結合 ?	演算子関数	使用法
3	はい	C++ 作成 (配置)	new *type* (*expr_list*) *type* (*expr_list*)
3	はい	C++ 破棄 (メモリーの割り振り解除)	delete *pointer*
3	はい	C++ 配列の破棄	delete [ ] *pointer*
3	はい	型変換 (キャスト)	(*type*) *expr*

表 23. 2 項演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
4		C++ メンバー選択	*object* .* *ptr_to_member*
4		C++ メンバー選択	*object* ->* *ptr_to_member*
5		乗算	*expr* * *expr*
5		除法	*expr* / *expr*
5		モジュロ (剰余)	*expr* % *expr*
6		2 項加算	*expr* + *expr*
6		2 項減算	*expr* - *expr*
7		ビット単位シフト	*expr* << *expr*
7		右へのビット単位シフト	*expr* >> *expr*
8		より小さい	*expr* < *expr*
8		より小さいまたは等しい	*expr* <= *expr*
8		より大きい	*expr* > *expr*
8		より大きいまたは等しい	*expr* >= *expr*
9		等しい	*expr* == *expr*
9		等しくない	*expr* != *expr*
10		ビット単位 AND	*expr* & *expr*
11		ビット単位排他 OR	*expr* ^ *expr*
12		ビット単位包含 OR	*expr* | *expr*
13		論理 AND	*expr* && *expr*
14		論理包含 OR	*expr* || *expr*
15		条件式	*expr* ? *expr* : *expr*
16	はい	単純代入	*lvalue* = *expr*
16	はい	乗算および代入	*lvalue* *= *expr*
16	はい	除算および代入	*lvalue* /= *expr*
16	はい	モジュロおよび代入	*lvalue* %= *expr*
16	はい	加算および代入	*lvalue* += *expr*
16	はい	減算および代入	*lvalue* -= *expr*
16	はい	左へのシフトおよび代入	*lvalue* <<= *expr*
16	はい	右へのシフトおよび代入	*lvalue* >>= *expr*
16	はい	ビット単位 AND および代入	*lvalue* &= *expr*

表 23. 2 項演算子の優先順位と結合順序 (続き)

ランク	右結合 ?	演算子関数	使用法
16	はい	ビット単位排他 OR および代入	*lvalue* ^= *expr*
16	はい	ビット単位包含 OR および代入	*lvalue* |= *expr*
17	はい	C++ throw 式	throw *expr*
18		コンマ (順序付け)	*expr* , *expr*

式および優先順位の例

以下の式では、小括弧は、コンパイラーがオペランドや演算子をグループ化する方法を明示的に示します。

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

これらの式に小括弧がない場合、小括弧によって指示されるのと同じ方法で、オペランドと演算子がグループ化されます。例えば、次の式は同じ出力を作成します。

```
total = (4+(5*3));
total = 4+5*3;
```

結合属性と可換属性の両方がある演算子とオペランドをグループ化する順序は規定されていないので、次の式で、コンパイラーはオペランドと演算子をグループ化します。

```
total = price + prov_tax +
city_tax;
```

例えば、次のような方法が (小括弧で示されているように) 可能です。

```
total = (price + (prov_tax + city_tax));
total = ((price + prov_tax) + city_tax);
total = ((price + city_tax) + prov_tax);
```

ある順序付けがオーバーフローを引き起こし、他の順序付けがオーバーフローを引き起こさないというのでないかぎり、オペランドおよび演算子のグループ化が、結果に影響を与えることはありません。例えば、`price = 32767`、`prov_tax = -42`、および `city_tax = 32767`、ならびにこれら 3 つの変数すべてが整数として宣言されていた場合、3 番目のステートメント `total = ((price + city_tax) + prov_tax)` は、整数オーバーフローを引き起こすけれども、他のステートメントは引き起こしません。

中間値が丸められるため、浮動小数点演算子の異なるグループ化は、異なる結果になる場合があります。

ある式では、オペランドや演算子のグループ化によっては、結果に影響を与えます。例えば、次の式では、各関数呼び出しは同じグローバル変数を変更します。

```
a = b() + c() + d();
```

この式は、関数が呼び出される順序によって、異なる結果になります。

式に結合属性と可換属性の両方がある演算子が含まれており、オペランドを演算子にグループ化する順序が式の結果に影響を与える場合は、式をいくつかの式に区切ります。例えば、呼び出し先関数が変数 `a` に影響を与えるような副次作用を作成しない場合は、次の式によって、前の例の式を置換できます。

```
a = b();
a += c();
a += d();
```

関数呼び出しの引数または 2 項演算子のオペランドの評価順序は、指定されていません。そのため、以下の式はあいまいです。

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

最初のステートメントの前に、y に値 1 が指定されている場合は、値 1 または値 2 が func1() に渡されるかどうかは不明です。2 番目のステートメントでは、式が評価される前に i の値が 1 である場合は、x[1] または x[2] が 2 番目の引数として func2() に渡されるかどうかは不明です。

第 7 章 ステートメント

ステートメントは最も小さな独立した計算単位で、実行される処理を指定します。ほとんどの場合、ステートメントは順序どおりに実行されます。以下に、C および C++ で使用可能なステートメントの要約を示します。

- ラベル付きステートメント
- 式ステートメント
- ブロック・ステートメント
- 選択ステートメント
- 繰り返しステートメント
- 分岐ステートメント
- 宣言ステートメント
- C++ `try` ブロック
- ヌル・ステートメント

関連情報

- 35 ページの『第 3 章 データ・オブジェクトおよび宣言』
- 169 ページの『関数宣言』
- 331 ページの『try ブロック』

ラベル付きステートメント

ラベルには identifier、case、および default の 3 つの種類があります。

ラベル付きステートメントの構文

```
▶▶—identifier—:—statement——————————————————————————————▶◀
```

ラベルは、*identifier* およびコロン (**:**) 文字から構成されます。

identifier ラベルは goto ステートメントのターゲットとして使用することができます。 goto ステートメントでは、ラベルをその定義よりも前に使用することができます。 identifier ラベルは、それ自体のネーム・スペースを持っています。identifier ラベルが、他の ID と競合することについて心配する必要はありません。 ただし、関数内ではラベルを再宣言することはできません。

400 identifier ラベルは `#pragma exception_handler` ディレクティブのターゲットとしても使用できます。 `#pragma exception_handler` ディレクティブの使用例と使用法については、「*IBM Rational Development Studio for i: ILE C/C++ プログラマーの手引き*」を参照してください。

case および default ラベル・ステートメントは、`switch` ステートメントでのみ使用されます。 これらのラベルは、最も近い `switch` ステートメント内でのみアクセス可能です。

© Copyright IBM Corp. 1998, 2010

case ステートメントの構文

```
▶▶─case─constant_expression─:─statement──────────────────────────────────▶◀
```

default ステートメントの構文

```
▶▶─default─:─statement────────────────────────────────────────────────────▶◀
```

ラベルの例を次に示します。

```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```

関連情報

- 167 ページの『goto ステートメント』
- 156 ページの『switch ステートメント』

式ステートメント

式ステートメント は、式を含んでいます。式は、ヌルでもかまいません。

式ステートメントの構文

```
▶▶──┬────────────┬─;──────────────────────────────────────────────────────▶◀
    └─expression─┘
```

式ステートメントは式 を評価し、次に式の値を破棄します。式のない式ステートメントは、ヌル・ステートメントです。

ステートメントの例を次に示します。

```
printf("Account Number: \n");         /* call to the printf    */
marks = dollars * exch_rate;                /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

関連情報

- 103 ページの『第 6 章 式と演算子』

あいまいなステートメントの解決 (C++ のみ)

C++ 構文は、式ステートメントと宣言ステートメントの間のあいまいさを明確化していません。式ステートメントの左端の副次式に関数スタイル・キャストがあると、あいまいさが生じます。 (C では、関数スタイルのキャストをサポートしないため、C プログラムではこのようなあいまいさは起きません。) ステートメントを宣言または式のいずれにも解釈できる場合、ステートメントは宣言ステートメントとして解釈されます。

注: あいまいさは、構文レベルでのみ解決されます。明確化に際して、名前の意味が型名であるかどうかを評価する場合を除いて、名前の意味を使用しません。

以下の式は、あいまいな副次式のあとに、代入または演算子が続いているため、式ステートメントとして解決されます。これらの式の `type_spec` は、任意の型指定子にすることができます。

```
type_spec(i)++;             // expression statement
type_spec(i,3)<<d;          // expression statement
type_spec(i)->l=24;         // expression statement
```

以下の例では、あいまいさを構文的に解決できません。コンパイラーは、ステートメントを宣言として解釈します。 type_spec は、任意の型指定子です。

```
type_spec(*i)(int);         // declaration
type_spec(j)[5];            // declaration
type_spec(m) = { 1, 2 };    // declaration
type_spec(*k) (float(3));   // declaration
```

上記の最後のステートメントは、浮動値を用いてポインターを初期化することができないため、コンパイル時エラーとなります。

上記の規則で解析されないあいまいなステートメントは、デフォルトにより宣言ステートメントであると見なされます。以下のステートメントはすべて宣言ステートメントです。

```
type_spec(a);               // declaration
type_spec(*b)();            // declaration
type_spec(c)=23;            // declaration
type_spec(d),e,f,g=0;       // declaration
type_spec(h)(e,3);          // declaration
```

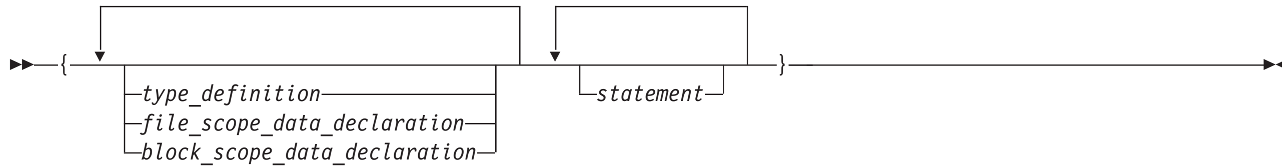
関連情報

- 35 ページの『第 3 章 データ・オブジェクトおよび宣言』
- 103 ページの『第 6 章 式と演算子』
- 109 ページの『関数呼び出し式』

ブロック・ステートメント

ブロック・ステートメント または*複合ステートメント* を使用すると、任意の数のデータ定義、宣言、およびステートメントを 1 つのステートメントにグループ化します。 1 組の中括弧に囲まれた定義、宣言、およびステートメントはすべて、単一のステートメントとして扱います。単一のステートメントが使用可能な場所ならばどこでもブロックを使用することができます。

ブロック・ステートメントの構文



ブロックは、ローカル・スコープを定義します。データ・オブジェクトがブロック内で使用可能であり、その ID が再定義されていない場合には、ネストされたすべてのブロックが、そのデータ・オブジェクトを使用できます。

ブロックの例

以下のプログラムは、ネストされたブロック内でデータ・オブジェクトの値がどのように変更されるかを示しています。

```
/**
 ** This example shows how data objects change in nested blocks.
 **/
  #include <stdio.h>

  int main(void)
  {
     int x = 1;                    /* Initialize x to 1  */
     int y = 3;

     if (y > 0)
     {
        int x = 2;                 /* Initialize x to 2  */
        printf("second x = %4d¥n", x);
     }
     printf("first  x = %4d¥n", x);

     return(0);
  }
```

プログラムは、以下の出力を作成します。

```
second x =    2
first  x =    1
```

x という名前の 2 つの変数が main の中で定義されています。 x の最初の定義は、main が実行されている間、ストレージを保存します。しかし、x の 2 番目の定義 (再定義) がネストされたブロック内で発生するため、 printf("second x = %4d¥n", x); は、x を前の行で定義された変数であると認識します。printf("first x = %4d¥n", x); は、ネストされたブロックの一部ではないので、x は、x の最初の定義として認識されます。

選択ステートメント

選択ステートメントは次のタイプのステートメントで構成されます。

- if ステートメント
- switch ステートメント

if ステートメント

if ステートメントは、複数の制御フローが可能な選択ステートメントです。

C++ *if* ステートメント を使用すると、指定されたテスト式 (暗黙的に bool に変換されている) が 真 に評価されたときに、ステートメントを条件付きで処理することができます。 bool への暗黙的な変換が失敗した場合は、プログラムが不適格です。

C C では、if ステートメントを使用すると、指定されたテスト式の評価が非ゼロ値になった場合に、ステートメントを条件付きで処理することができます。テスト式は、算術型またはポインター型でなければなりません。

オプションで、if ステートメントで else 節を指定することができます。テスト式の評価が false (または C では、ゼロ値) になり、else 文節がある場合、else 文節に関連付けられているステートメントが実行されます。テスト式の評価が true になった場合、その式に続くステートメントが実行され、else 文節は無視されます。

if ステートメントの構文

```
►►—if—(—expression—)—statement—┬────────────────┬──────────────────────────►◄
                               └else—statement─┘
```

if ステートメントがネストされていて、else 節が存在する場合、指定された else は、同じブロック内の直前の if ステートメントに関連付けられます。

任意の選択ステートメント (if、switch) に続く単一のステートメントは、オリジナルのステートメントを含んでいる複合ステートメントとして扱われます。結果として、そのステートメントで宣言されたすべての変数は、if ステートメントの後、スコープの外にあります。次に例を示します。

```
if (x)
   int i;
```

は、以下と同等です。

```
if (x)
{  int i; }
```

変数 i は、if ステートメント内でのみ可視です。同じ規則が if ステートメントの else 部分にも適用されます。

if ステートメントの例

以下の例では、score の値が 90 以上である場合に、grade が A という値を受け取るようにします。

```
if (score >= 90)
   grade = 'A';
```

以下の例では、number の値が 0 またはそれ以上である場合に Number is positive と表示します。number の値が 0 より小さい場合には、 Number is negative と表示します。

```
if (number >= 0)
   printf("Number is positive¥n");
else
   printf("Number is negative¥n");
```

以下の例は、ネストされた if ステートメントを示しています。

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
cout << "salary is " << salary << endl;
```

以下の例は、else 節を持たない、ネストされた if ステートメントを示しています。 else 節は、常に、最も近い if ステートメントに関連付けられるため、特定の else 節を強制的に正しい if ステートメントに関連付けるには、中括弧が必要なことがあります。この例では、中括弧を省略すると、else 節は、ネストされた if ステートメントに関連付けられることになります。

```
if (kegs > 0) {
   if (furlongs > kegs)
      fxph = furlongs/kegs;
}
else
   fxph = 0;
```

以下の例は、else 節の中にネストされた if ステートメントを示しています。この例では、複数の条件がテストされます。テストは、それらの条件が書かれている順序で行われます。 1 つのテストが非ゼロ値に評価されると、ステートメントが実行され、if ステートメント全体が終了します。

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

関連情報

- 46 ページの『ブール型』

switch ステートメント

switch ステートメント は、switch 式の値に応じて、switch 本体内の別のステートメントに制御を移す選択ステートメントです。switch 式の評価は、整数値または列挙値にならなければなりません。 switch ステートメントの本体には、以下で構成される *case 文節* が含まれています。

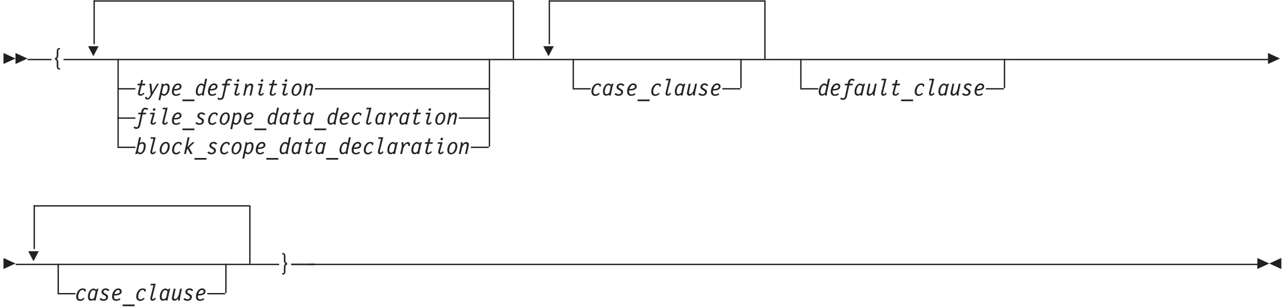
- case ラベル
- オプションの default ラベル
- case 式
- ステートメントのリスト。

switch 式の値が case 式の 1 つの値と同じ場合、その case 式に続くステートメントが処理されます。そうでない場合、default ラベル・ステートメント (あれば) が処理されます。

switch ステートメントの構文

```
►►—switch—(—expression—)—switch_body——————————————————►◄
```

switch 本体 は、中括弧で囲まれ、定義、宣言、*case 文節*、および *default 文節* を含むことができます。 case 文節と default 文節のそれぞれにステートメントを含めることができます。



注: *type_definition*、*file_scope_data_declaration* または *block_scope_data_declaration* 内の初期化指定子は、無視されます。

case 文節 には、任意の数のステートメントが続く *case label* が含まれます。 case 文節の形式は、次のとおりです。

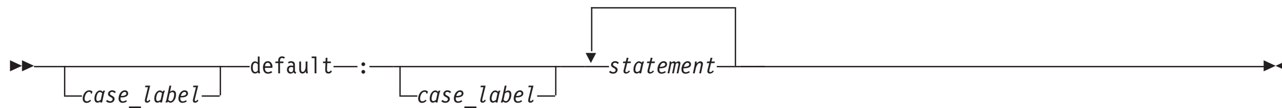
case 文節の構文

case ラベル には、case というワードと、それに続く整数定数式とコロンが入っています。 各整数定数式の値は、別々の値を表している必要があります。重複した case ラベルを持つことはできません。1 つの case ラベルを置けるところであればどこでも、複数の case ラベルを置くことができます。 case ラベルの形式は、次のとおりです。

case ラベルの構文

default 文節 には、default ラベルと、それに続く 1 つまたは複数のステートメントが入っています。 case ラベルは、default ラベルのどちらの側にも置くことができます。 switch ステートメントに入れることができる default ラベルは、1 つだけです。 *default_clause* の形式は、次のとおりです。

default 文節ステートメント



switch ステートメントは、いずれか 1 つのラベルに続くステートメント、または switch 本体に続くステートメントに制御を渡します。switch 本体の前にある式の値によって、制御を受け取るステートメントが決まります。この式は *switch 式* と呼ばれます。

switch 式の値は、各 case ラベルの式の値と比較されます。一致している値が検出されれば、一致したその値が入っている case ラベルに続くステートメントに、制御が渡されます。一致する値はないが、switch 本体の中に default ラベルがある場合には、default ラベルの付いたステートメントに制御が渡されます。一致する値が見つからず、switch 本体の中のどこにも default ラベルがない場合には、switch 本体のどの部分も処理されません。

switch 本体の中のステートメントに制御が渡されると、 break ステートメントが検出されたとき、または switch 本体の中の最後のステートメントが処理されたときにのみ、制御は switch 本体を離れます。

必要であれば、整数拡張が、制御式上で実行されます。また、case ステートメント内のすべての式が、制御式と同じ型に変換されます。 switch 式は、整数型または列挙型への単一変換があれば、クラス型の式にもなります。

CHECKOUT(*GENERAL) オプションでコンパイルすると、意に反して失敗する case ラベルが検出されます。

switch ステートメントの制約事項

データ定義を switch 本体の先頭に置くことができますが、コンパイラーは、switch 本体の先頭にある auto および register 変数は、初期化しません。 switch ステートメントの本体の中に、宣言を入れることができます。

switch ステートメントを使用して、初期化を飛び越えることはできません。

可変変更型の ID のスコープに switch ステートメントの case または default ラベルが含まれる場合、switch ステートメント全体がその ID のスコープの中にあるものと見なされます。すなわち、この ID の宣言は switch ステートメントの前になければなりません。

C++ C++ では、暗黙的または明示的な初期化指定子を含んでいる宣言を超えて、制御権を移動することはできません。ただし、宣言が、制御権の移動によって完全にう回される内部ブロックに入っている場合は、制御権を移動することができます。初期化指定子が入っている switch ステートメント本体内の宣言はすべて、内部ブロックに入れる必要があります。

switch ステートメントの例

以下の switch ステートメントには、複数の case 文節と 1 つの default 文節が入っています。各文節には、関数呼び出しと break ステートメントが入っています。break ステートメントは、switch 本体内の各ステートメントに制御が移されていくのを防止します。

switch 式の評価が '/' となった場合、その switch ステートメントは関数 divide を呼び出すことになります。その後、switch 本体に続くステートメントに制御が渡されます。

```
char key;

printf("Enter an arithmetic operator¥n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;

     default:
      printf("invalid key¥n");
      break;
}
```

switch 式と case 式が一致した場合には、break ステートメントが検出されるまで、または switch 本体の終わりに達するまで、case 式に続くステートメントが処理されます。以下の例には、break ステートメントはありません。 text[i] の値が 'A' である場合、コンパイラーは、3 つのカウンターすべてを増やします。 text[i] の値が 'a' と等しい場合には、lettera と total が増やされます。 text[i] が 'A' または 'a' と等しくない場合には、total のみが増やされます。

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

次の switch ステートメントは、複数の case ラベルに対して同じステートメントを実行します。

```
/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
 **/

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
     case 2:
        printf("month %d is a winter month¥n", month);
      break;

     case 3:
     case 4:
     case 5:
        printf("month %d is a spring month¥n", month);
      break;

     case 6:
     case 7:
     case 8:
        printf("month %d is a summer month¥n", month);
      break;

     case 9:
     case 10:
     case 11:
        printf("month %d is a fall month¥n", month);
      break;

     case 66:
     case 99:
       default:
        printf("month %d is not a valid month¥n", month);
```

```
  }

   return(0);
}
```

式 month の値が 3 の場合には、次のステートメントに制御が渡されます。

```
printf("month %d is a spring month¥n",
month);
```

break ステートメントは、switch 本体に続くステートメントに制御を渡します。

関連情報

- 「「*ILE C/C++ コンパイラー参照*」にある **CHECKOUT(*GENERAL)**
- 151 ページのCase ラベルと Default ラベル
- 164 ページの『break ステートメント』

繰り返しステートメント

繰り返しステートメントは次のタイプのステートメントで構成されます。

- while ステートメント
- do ステートメント
- for ステートメント

関連情報

- 46 ページの『ブール型』

while ステートメント

while ステートメント は、制御式の評価が false (または C では 0) になるまで、ループの本体を繰り返し実行します。

while ステートメントの構文

```
▶▶—while—(—expression—)—statement——————————————————————————————▶◀
```

C *expression* は、算術型またはポインター型でなければなりません。 C++ expression は、bool に変換可能なものでなければなりません。

式は評価されて、ループの本体を処理するかどうかを判別します。式の評価が false の場合、ループの本体は実行されません。式が false に評価されないと、ループ本体は処理されます。本体が実行された後、制御は式に戻されます。それ以降の処理は、条件の値によって決まります。

break、return、または goto ステートメントがあると、条件の評価が false でない場合でも、while ステートメントを終了できます。

C++ throw 式がある場合も、条件が評価される前に while ステートメントが終了することがあります。

次の例では、式 ++index の値が MAX_INDEX より小さい間は、 item[index] は 3 倍され、印刷されます。++index の評価が MAX_INDEX になると、while ステートメントは終了します。

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d¥n", index, item[index]);
      ++index;
   }

   return(0);
}
```

do ステートメント

do ステートメント は、テスト式の評価が false (または C では 0) になるまで、ステートメントを繰り返し実行します。処理の順序のために、そのステートメントは少なくとも 1 回は実行されます。

do ステートメントの構文

```
►►—do—statement—while—(—expression—)—;——————————————————►◄
```

C *expression* は、算術型またはポインター型でなければなりません。 C++ 制御する *expression* は、型 bool に変換可能でなければなりません。

制御する while 文節が評価される前に、ループの本体が実行されます。それ以降の do ステートメントの処理は、while 文節の値によって決まります。 while 文節が false に評価されない場合には、再度ステートメントが実行されます。while 文節の評価が false になると、ステートメントは終了します。

break、return、または goto ステートメントは、while 文節が false に評価されなくても、do ステートメントの処理を終了させることができます。

C++ throw 式がある場合も、条件が評価される前に do ステートメントが終了することがあります。

次の例では、i が 5 より小さい間は、i を増分し続けます。

```
#include <stdio.h>

int main(void) {
  int i = 0;
  do {
    i++;
    printf("Value of i: %d¥n", i);
  }
  while (i < 5);
  return 0;
}
```

次に、上記の例の出力を示します。

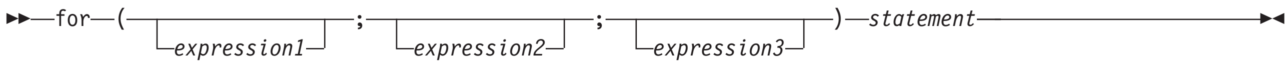
```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

for ステートメント

for ステートメント を使用すると、以下のことを行うことができます。

- ステートメントの最初の反復の前に式を評価する。(*初期化*)
- 式を指定して、ステートメントを処理するかどうかを判別する (*条件*)
- ステートメントが反復されるたびに、その後で式を評価する (反復のための増分によく使用される)
- 制御部分の評価が `false` (C では、`0`) にならない場合に、ステートメントを繰り返し処理する。

for ステートメントの構文



expression1 は*初期化式*です。これは、*ステートメント* が始めて処理される前においてのみ評価されます。この式を使用すると、変数を初期化することができます。この式を使用して変数を宣言することもできます。ただし、その変数が static として宣言されていない場合に限られます (変数は automatic でなければならず、`register` として宣言することもできます)。この式で、またはステートメント の中のどこででも、変数を宣言すると、その変数は、`for` ループの終わりでスコープの外に出ます。ステートメントの最初の反復の前に式の評価を行いたくない場合には、この式を省略することができます。

expression2 は条件式です。ステートメント の各反復の前に評価されます。 C *expression2* は算術型かポインター型でなければなりません。 C++ *expression3* は `bool` 型に変換可能でなければなりません。

評価が `false` (または C では `0`) であった場合、そのステートメントは処理されず、制御は `for` ステートメントの次のステートメントに移ります。 *expression2* の評価が `false` でない場合、ステートメントは処理されます。*expression2* を省略すると、この式が `true` によって、置き換えられたのと同様になり、この条件の不備により `for` ステートメントが終了しないことになります。

expression3 は、ステートメントの毎回の反復後に評価されます。この式は、変数に対する増分、減分、または代入のために頻繁に使用されます。この式はオプションです。

`break`、`return`、または `goto` ステートメントを使用すると、 2 番目の式の評価が `false`でなくても `for` ステートメントが終了させられます。*expression2* を省略する場合には、`break`、`return`、または `goto` ステートメントを使用して `for` ステートメントを終了することが必要になります。

関連情報

- 「*ILE C/C++ コンパイラー参照*」にある **LANGLVL**

for ステートメントの例

次の `for` ステートメントは、`count` の値を `20` 回印刷します。 `for` ステートメントは、`count` の値を `1` に初期設定します。ステートメントが反復されるたびに `count` が増やされます。

```
int count;
for (count = 1; count <= 20; count++)
   printf("count = %d¥n", count);
```

次の一連のステートメントも同じタスクを実行します。for ステートメントの代わりに while ステートメントを使用していることに注意してください。

```
int count = 1;
while (count <= 20)
{
   printf("count = %d¥n", count);
   count++;
}
```

以下の for ステートメントには、初期化の式が含まれていません。

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d¥n", index,
   list[index]);
}
```

以下の for ステートメントは、 scanf が e という文字を受け取るまで実行し続けます。

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '¥n')
         continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c¥n", letter);
}
```

以下の for ステートメントには、複数の初期化と増分が含まれています。コンマ演算子によってこの構造が可能となります。 for 式内の最初のコンマは、宣言の区切り子です。これは、i と j の 2 つの整数を宣言して、初期化します。 2 番目のコンマ (コンマ演算子) によって、ループ内の各ステップを通るたびに、i と j を両方とも増加することが可能になります。

```
for (int i = 0,
j = 50; i < 10; ++i, j += 50)
{
   cout << "i = " << i << "and j = " << j
   << endl;
}
```

以下の例は、ネストされた for ステートメントを示しています。このステートメントは、[5][3] という次元の配列の値を印刷します。

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d¥n",
      table[row][column]);
```

row の値が 5 より小さい間、外部ステートメントが処理されます。外部の for ステートメントが実行されるたびに、内部の for ステートメントが column の初期値をゼロに設定します。そして、内部の for ステートメントのステートメントが 3 回実行されます。 column の値が 3 より小さい間は、内部ステートメントが実行されます。

分岐ステートメント

分岐ステートメントは次のタイプのステートメントで構成されます。

- break ステートメント
- continue ステートメント
- return ステートメント
- goto ステートメント

break ステートメント

break ステートメント を使用すると、*反復* (do、for、while) ステートメントまたは switch ステートメントを終了して、論理終了以外の任意のポイントで、ステートメントから出ることができます。 break は、これらのステートメントのいずれかだけに使用できます。

break ステートメントの構文

```
►►──break──;──────────────────────────────────────────────────────►◄
```

反復するステートメントにおいては、break ステートメントはループを終了して、ループの外側にある次のステートメントに制御を移します。ネストされたステートメントの中では、break ステートメントは、囲んでいる最小の do、for、switch、または while ステートメントのみを終了します。

switch ステートメントにおいては、break は、制御を switch 本体から switch 本体の外側にある、次のステートメントに渡します。

continue ステートメント

continue ステートメント を使用すると、進行中のループの反復を終了することができます。プログラム制御は、continue ステートメントからループ本体の終わりに渡されます。

continue ステートメントの形式は、次のとおりです。

```
►►──continue──;───────────────────────────────────────────────────►◄
```

continue ステートメントは、do、for、または while のような反復するステートメントの本体の中にしか存在できません。

continue ステートメントは、反復するステートメントのアクション部分の処理を終了します。そして、制御を、ステートメントのループ連結部分へ移します。例えば、反復するステートメントが for ステートメントである場合、制御は、ステートメントの条件部分の 3 番目の式に移動します。次に、ステートメントの条件部分の 2 番目の式 (テスト) に移動します。

ネストされたステートメントの中では、continue ステートメントは、直接に continue ステートメントを囲んでいる do、for、または while ステートメントの現行の反復だけを終了します。

continue ステートメントの例

以下の例は、for ステートメントにおける continue ステートメントを示しています。continue ステートメントを使用すると、値が 1 以下の配列 rates のエレメントに対する処理がスキップされます。

```
/**
 ** This example shows a continue statement in a for statement.
 **/
```

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

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00¥n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
            continue;
      printf("rate = %.2f¥n", rates[i]);
   }

   return(0);
}
```

プログラムは、以下の出力を作成します。

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

以下の例は、ネストされたループにおける continue ステートメントを示しています。内部ループが配列 strings の中である数に遭遇すると、そのループの反復は終了します。処理は、内部ループの 3 番目の式から続けられます。内部ループは、'¥0' エスケープ・シーケンスを検出した時点で終了します。

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '¥0';
      ++pointer)
      {                                  /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d¥n", letter_count);

   return(0);
}
```

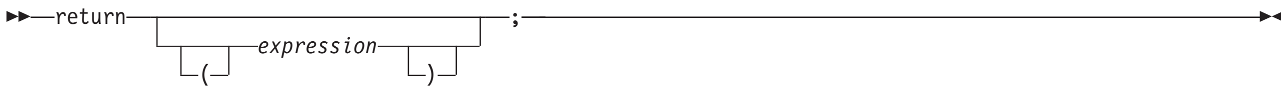
プログラムは、以下の出力を作成します。

```
letter count = 5
```

return ステートメント

return ステートメント は、現行の関数の処理を終了して、関数の呼び出し元に制御を戻します。

return ステートメントの構文



値を返す関数は `return` ステートメントを含み、式が入っていなければなりません。 C++ 非 `void` の戻りの型を宣言されている関数内の `return` ステートメントに式が指定されていない場合は、コンパイラーによってエラー・メッセージが出力されます。

式のデータ型が関数の戻りの型と異なる場合には、式の値が、あたかも、関数の戻りの型が同じであるオブジェクトに割り当てられたかのように、戻り値の変換が行われます。

戻りの型 `void` の関数の場合、return ステートメントは必ず必要というわけではありません。 `return` ステートメントを検出することなく関数の終わりに達すると、あたかも式のない `return` ステートメントが検出されたかのように、制御は呼び出し元に渡されます。つまり、最終ステートメントの完了時に暗黙的な戻りが実行され、制御は自動的に呼び出し元関数に戻ります。 C++ `return` ステートメントが使用されている場合は、式を含まないようにする必要があります。

return ステートメントの例

return ステートメントの例を次に示します。

```
return;            /* Returns no value           */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1        */
return (x * x);    /* Returns the value of x * x  */
```

以下の関数は、整数の配列を検索して、変数 `number` と一致するものが存在するかどうか判別します。一致するものが存在すると、関数 `match` は `i` の値を戻します。一致するものが存在しない場合、関数 `match` は `-1` (マイナス 1) という値を戻します。

```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
      if (number == array[i])
         return (i);
   return(-1);
}
```

1 つの関数に、複数の `return` ステートメントを含めることができます。 次に例を示します。

```
void copy( int *a, int *b, int c)
{
   /* Copy array a into b, assuming both arrays are the same size */

   if (!a || !b)       /* if either pointer is 0, return */
      return;

   if (a == b)         /* if both parameters refer */
      return;          /*    to same array, return */

   if (c == 0)         /* nothing to copy */
```

```
      return;

   for (int i = 0; i < c; ++i;) /* do the copying */
      b[i] = a[1];
                       /* implicit return */
}
```

この例では、return ステートメントは、break ステートメントのように、関数を途中で終了するために使用されています。

return ステートメントに現れる式は、このステートメントが現れる関数の戻りの型に変換されます。暗黙の変換ができない場合、return ステートメントは無効になります。

関連情報

- 180 ページの『関数からの戻りの型指定子』
- 181 ページの『関数からの戻り値』

goto ステートメント

goto ステートメント を使用すると、プログラムは、goto ステートメントで指定されたラベルに関連付けられたステートメントに無条件に制御を移します。

goto ステートメントの構文

```
►►—goto—label_identifier—;————————————————————————————————————►◄
```

goto ステートメントは、通常の処理シーケンスを妨げる可能性があるため、これを使用すると、プログラムの読み取りおよび保守が難しくなります。多くの場合、break ステートメント、continue ステートメント、または関数呼び出しを使えば、goto ステートメントを使用しなくてもすみます。

goto ステートメントを使用してアクティブ・ブロックを終了する場合、そのブロックから制御が移動するときに、すべてのローカル変数は破棄されます。

goto ステートメントを使用して、初期化を飛び越えることはできません。

goto ステートメントは、可変長配列のスコープ内にジャンプすることができますが、可変的に変更される型を持つオブジェクトの宣言を飛び越えてジャンプすることはできません。

以下の例は、ネストされたループからジャンプするために使用される goto ステートメントを示しています。この関数は、goto ステートメントを使用しなくても作成することができます。

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
   int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
```

```
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d¥n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6¥n");
}
```

関連情報

- 151 ページの『ラベル付きステートメント』

ヌル・ステートメント

ヌル・ステートメント はオペレーションを実行しません。形式は次のとおりです。

```
►►─;─────────────────────────────────────────────────────►◄
```

ヌル・ステートメントは、ラベル付きステートメントのラベルを保持したり、あるいは反復するステートメントの構文を完了したりすることができます。

以下の例は、配列 `price` のエレメントを初期化します。初期化は `for` 式の中で起きるため、`for` 構文を終了するのに必要なものはステートメントのみで、オペレーションは必要ありません。

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

ブロック・ステートメントの終わりの前にラベルを必要とするときに、ヌル・ステートメントを使用できます。次に例を示します。

```
void func(void) {
  if (error_detected)
    goto depart;
  /* further processing */
  depart: ;  /* null statement required */
}
```

第 8 章 関数

プログラム言語のコンテキストにおいて、*関数* という語は、出力値を計算するために使用されるステートメントの集まりを意味します。この語は、数学で使われるときほど厳密に使われていません。数学では、関数は、入力変数を出力変数に*一意的に* 関連付ける集合を意味します。 C または C++ プログラムにおける関数は、すべての入力に対して一貫性のある出力を生成するとは限らず、出力をまったく生成しないこともあり、副次作用を持つこともあります。関数は、パラメーター・リストのパラメーター (存在する場合) をオペランドとする、ユーザー定義の演算と考えることができます。

このセクションでは、以下のトピックについて説明します。

- 『関数宣言および関数定義』
- 174 ページの『関数ストレージ・クラス指定子』
- 176 ページの『関数指定子』
- 180 ページの『関数からの戻りの型指定子』
- 181 ページの『関数宣言子』
- 185 ページの『関数属性』
- 188 ページの『main() 関数』
- 189 ページの『関数呼び出し』
- 193 ページの『C++ 関数のデフォルト引数 (C++ のみ)』
- 195 ページの『関数へのポインター』

関数宣言および関数定義

関数*宣言* と関数*定義* の違いは、データ宣言とデータ定義の違いに似ています。宣言では、関数の名前と特性を設定しますが、関数用にストレージは割り振りません。*定義* では、関数の本体を指定し、ID を関数に関連付け、関数用にストレージを割り振ります。そのため、以下の例で宣言される ID では、ストレージは割り振られません。

```
float square(float x);
```

関数定義 には、関数宣言と関数本体が含まれます。本体は、関数の処理を実行するステートメントのブロックです。この例に宣言されている ID は、ストレージを割り当てます。これらの ID は、宣言と定義の両方になります。

```
float square(float x)
{ return x*x; }
```

1 つの関数を 1 つのプログラム内で複数回宣言することができますが、所定の関数についての宣言はすべて互換性がなければなりません。つまり、戻りの型が同じで、パラメーターの型が同じです。ただし、1 つの関数には 1 つの定義しか認められません。通常、宣言はヘッダー・ファイルに入れますが、定義はソース・ファイルに入れます。

関数宣言

関数の戻りの型が先行し、関数のパラメーター・リストが後続している関数 ID を*関数宣言* または*関数プロトタイプ* と呼びます。プロトタイプは、コンパイラーが関数を使用する前に、コンパイラーに対して、

© Copyright IBM Corp. 1998, 2010

関数の形式と存在を通知します。コンパイラーは、関数呼び出しのパラメーターと関数宣言におけるパラメーターとの不一致を検査することができます。コンパイラーは、引数の型の検査および引数の変換のためにも、この宣言を使います。

C++ 関数の暗黙宣言は使用できません。すべての関数は、呼び出す前に明示的に宣言する必要があります。

C 関数宣言がその関数の呼び出し時点で可視でない場合には、コンパイラーは、`extern int func();` の暗黙宣言を仮定します。ただし、C99 に準拠するためには、関数を呼び出す前に、すべての関数を明示的にプロトタイプ化してください。

関数を宣言する際のエレメントは、次のとおりです。

- 関数ストレージ・クラス指定子。リンケージを指定します
- 関数からの戻りの型指定子。返す値のデータ型を指定します
- 関数指定子。関数の追加のプロパティーを指定します
- 関数宣言子。関数 ID およびパラメーターのリストが入ります

すべての関数宣言の形式は、次のとおりです。

関数宣言構文

```
>>-+------------------------------------------+--return_type_specifier--function_declarator--;-----><
   '-storage_class_specifier--function_specifier-'
```

400 また、 C++ との互換性のために、*属性* を使用して、関数のプロパティーを変更できます。これについては 185 ページの『関数属性』で説明してあります。

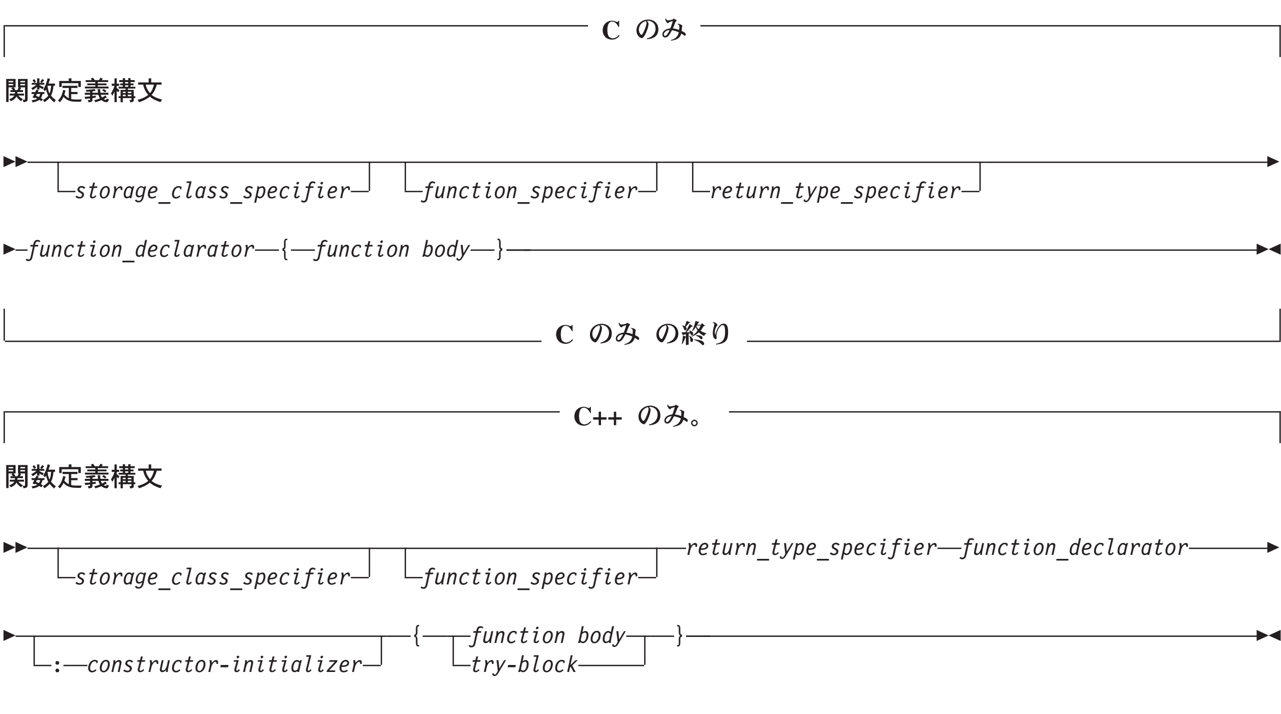
関数定義

関数定義のエレメントには次のものがあります。

- 関数ストレージ・クラス指定子。リンケージを指定します
- 関数からの戻りの型指定子。返す値のデータ型を指定します
- 関数指定子。関数の追加のプロパティーを指定します
- 関数宣言子。関数 ID およびパラメーターのリストが入ります
- *関数本体*。関数が実行するアクションを表す、中括弧で囲まれた一連のステートメントです
- C++ コンストラクター初期化指定子。クラスで宣言されるコンストラクター関数でのみ使用されます。これについては、 279 ページの『コンストラクター』で説明します
- C++ 試行ブロック。クラス関数で使用されます。これについては、 331 ページの『try ブロック』で説明します

400 また、 C++ との互換性のために、*属性* を使用して、関数のプロパティーを変更できます。これについては 185 ページの『関数属性』で説明してあります。

関数定義は、以下の形式をとります。



関数宣言の例

次のコード・フラグメントは、複数の関数宣言 (またはプロトタイプ) を示しています。最初の部分は、2 つの整数の引数を採用し、戻りの型が `void` である関数 `f` を宣言しています。

```
void f(int, int);
```

このフラグメントは、固定文字へのポインターを使用して整数を戻す関数へのポインター `p1` を宣言します。

```
int (*p1) (const char*);
```

次のコードは、関数 `f1` を宣言し、関数 f1 は、1 つの整数の引数を取り、整数の引数を取って整数を戻す関数へのポインターを戻します。

```
int (*f1(int)) (int);
```

関数 `f1` の複雑な戻りの型に対して、上記の関数の代わりに、`typedef` を使用することができます。

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

次の宣言は、最初の引数として定数整数を取る外部関数 `f2()` の宣言です。この宣言は、可変数で可変型の他の引数を持つことができ、型 `int` を戻します。

```
int extern f2(const int, ...);  /*  C version  */
```

```
int extern f2(const int ...);  //  C++ version
```

関数 `f6` は、クラス `X` の `const` クラス・メンバー関数で、引数は取りません。`int` の戻りの型を持っています。

```
class X
{
public:
      int f6() const;
};
```

関数 f4 は、引数を取らず、戻りの型が void で、X 型および Y 型のクラス・オブジェクトをスロー (throw) することができます。

```
class X;
class Y;

// ...

void f4() throw(X,Y);
```

関数定義の例

次の例は、関数 sum の定義です。

```
int sum(int x,int y)
{
   return(x + y);
}
```

関数 sum には、外部結合があり、int 型のオブジェクトを戻し、 x および y と宣言された int の 2 つのパラメーターがあります。この関数本体には、x と y の合計を戻す 1 個のステートメントが入っています。

次の関数 set_date は、date 型の構造体へのポインターをパラメーターとして宣言します。 date_ptr は、ストレージ・クラス指定子 register を持っています。

```
void set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

互換関数 (C のみ)

2 つの関数型が互換性を持つようにするには、以下の要件を満たす必要があります。

- パラメーターの数 (および省略符号の使用) が一致する。
- 互換性のある戻りの型を持つ。
- 対応するパラメーターが、デフォルト引数上位変換を適用した結果の型と互換性がある。

2 つの関数型の複合型は、以下のように決まります。

- 関数型の 1 つがパラメーター型リストを持つ場合は、複合型は同じパラメーター型リストを持つ関数プロトタイプです。
- 両方の関数型にパラメーター型リストが存在する場合には、各パラメーターの複合型は以下のように決まります。
  - 異なるランクのパラメーターの複合は、デフォルト引数上位変換を適用した結果の型になります。
  - 配列または関数型のパラメーターの複合は、調整型になります。
  - 修飾型のパラメーターの複合は、宣言型の修飾されないバージョンになります。

たとえば、次の 2 つの関数宣言の場合、

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

結果の複合型は、以下のようになります。

```
int f(int (*)(char *), double (*)[3]);
```

関数宣言子が関数宣言の一部ではない場合は、パラメーターの型は不完全型を持つことができます。このパラメーターは、宣言子指定子のシーケンスで [*] 表記を使って、可変長の配列の型を指定することもできます。以下は、互換性のある関数プロトタイプ宣言子の例です。

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

関連情報

- 37 ページの『互換型および複合型』

複数の関数宣言 (C++ のみ)

特定の関数に対する複数の関数宣言はすべて、パラメーターの数と型が同じでなければなりません。また、戻りの型も同じでなければなりません。

これらの戻りの型およびパラメーター型は、関数型の一部ですが、デフォルトの引数と例外指定は、関数型の一部ではありません。

すでになされたオブジェクトまたは関数の宣言が、囲みスコープで可視になっている場合は、 ID には、最初の宣言と同じリンケージが指定されています。ただし、リンケージを持たずに、後でリンケージ指定子を使用して宣言される変数または関数は、ユーザーが指定したリンケージを持ちます。

引数のマッチングの観点では、省略符号とリンケージ・キーワードは、関数型の一部と見なされます。これらは、関数のすべての宣言において、矛盾しないように使用する必要があります。 2 つの宣言におけるパラメーター型で、typedef 名または未指定の配列境界の使用だけが相違している場合、その宣言は同じです。 const または volatile 型修飾子も関数型の一部ですが、これらは、非静的メンバー関数の宣言または定義の一部となることができるだけです。

2 つの関数において、戻りの型およびパラメーター・リストの両方が一致している場合は、2 番目の宣言は最初の宣言の再宣言と見なされます。以下の例では、同じ関数が宣言されています。

```
int foo(const string &bar);
int foo(const string &);
```

戻りの型が違うだけの 2 つの関数宣言は、無効な関数多重定義となり、コンパイル時エラーのフラグが付けられます。たとえば、次のとおりです。

```
void f();
int f();      // error, two definitions differ only in
              // return type
int g()
{
   return f();
}
```

関連情報

- 205 ページの『関数の多重定義』

関数ストレージ・クラス指定子

関数の場合は、ストレージ・クラス指定子が関数のリンケージを決めます。デフォルトで、関数定義は、外部リンケージを持ち、他のファイルで定義された関数で呼び出すことができます。 C 例外は、インライン関数です。インライン関数は、デフォルトで、内部リンケージを持つものとして処理されます。詳細については、 177 ページの『インライン関数のリンケージ』を参照してください。

ストレージ・クラス指定子は、関数宣言と関数定義の両方で使用できます。関数のストレージ・クラス・オプションは以下のもののみです。

- 静的 (`static`)
- `extern`

static ストレージ・クラス指定子

`static` ストレージ・クラス指定子を使って宣言された関数は、内部結合を持ちます。これは、この関数が、関数が定義されている変換単位内でしか呼び出しできないことを意味します。

`static` ストレージ・クラス指定子は、ファイル・スコープである場合にのみ、関数宣言で使用できます。関数はブロック内で `static` として宣言できません。

C++ `static` をこのように使用することは、C++ の場合は推奨できません。その代わりに、関数を名前なしネーム・スペースに入れます。

関連情報

- 8 ページの『内部結合』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

extern ストレージ・クラス指定子

`extern` ストレージ・クラス指定子を使って宣言された関数は、外部結合を持ちます。これは、この関数が、他の変換単位から呼び出し可能であるということを意味します。キーワード `extern` はオプションです。ストレージ・クラス指定子を指定しない場合は、関数に外部結合があると想定されています。

C++ `extern` 宣言を、クラス・スコープ内で発生させることはできません。

C++ のみ。

`extern` キーワードは、リンケージの型を指定する引数を指定して使用できます。

extern 関数ストレージ・クラス指定子の構文

```
►►—extern—"—linkage_specification—"——————————————————————————————►◄
```

すべてのプラットフォームは、*linkage_specification* に以下の値をサポートしています。

- C
- C++

400 ILE C++ によってサポートされる追加言語リンケージについては、「*ILE C/C++ Programmer's Guide*」の第 25 章「Working with Multi-Language Applications」を参照してください。

次のコードの部分は、extern "C" の使用を示しています。

```
extern "C" int cf();      //declare function cf to have C linkage

extern "C" int (*c_fp)(); //declare a pointer to a function,
                          // called c_fp, which has C linkage

extern "C" {
  typedef void(*cfp_T)(); //create a type pointer to function with C
                          // linkage
  void cfn();             //create a function with C linkage
  void (*cfp)();          //create a pointer to a function, with C
                          // linkage
}
```

リンケージの互換性は、qsort のように、パラメーターとしてユーザー関数ポインターを受け入れるすべての C ライブラリー関数に影響を与えます。 extern "C" リンケージ指定を使用して、宣言されたリンケージを同じものにする必要があります。次の例の一部で、qsort を指定した extern "C" を使用しています。

```
#include <stdlib.h>

// function to compare table elements
extern "C" int TableCmp(const void *, const void *); // C linkage
extern void * GenTable();                            // C++ linkage

int main() {
  void *table;

  table = GenTable();               // generate table
  qsort(table, 100, 15, TableCmp);  // sort table, using TableCmp
                                    // and C library routine qsort();
}
```

C++ 言語は、多重定義をサポートしますが、ほかの言語では、多重定義をサポートできません。これは次のことを意味します。

- 関数に C++ (デフォルト) のリンケージが指定されている限り、その関数を多重定義できます。したがって、次の一連のステートメントが許可されます。

  ```
  int func(int);            // function with C++ linkage
  int func(char);           // overloaded function with C++ linkage
  ```

  対照的に、C++ 以外のリンケージを持つ関数は多重定義できません。

  ```
  extern "C"{int func(int);}
  extern "C"{int func(int,int);}  // not allowed
                                  //compiler will issue an error message
  ```

- 多重定義関数と同じ名前を持つことができる C++ 以外のリンケージの関数は 1 つのみです。次に例を示します。

  ```
  int func(char);
  int func(int);
  extern "C"{int func(int,int);}
  ```

  ただし、C++ 以外のリンケージの関数は、同じ名前の C++ 関数のいずれとも同じパラメーターを持つことはできません。

  ```
  int func(char);      // first function with C++ linkage
  int func(int, int);  // second function with C++ linkage
  extern "C"{int func(int,int);}  // not allowed since the parameter
                                  // list is the same as the one for
                                  // the second function with C++ linkage
                                  // compiler will issue an error message
  ```

C++ のみ。 の終り

関連情報

- 8 ページの『外部結合』
- 9 ページの『言語リンケージ』
- 5 ページの『クラス・スコープ (C++ のみ)』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』

関数指定子

関数定義で使用可能な関数指定子は、以下のとおりです。

- inline。関数呼び出しの時点で関数定義を展開するようにコンパイラーに指示します。
- C++ explicit。クラスのメンバー関数でのみ使用できます。内容については、 292 ページの『明示的指定子』で説明します。
- C++ virtual。クラスのメンバー関数でのみ使用できます。内容については、 269 ページの『仮想関数』で説明します。

inline 関数指定子

インライン関数とは、コンパイラーが、個別の命令セットをメモリー内に作成するのではなく、関数定義からのコードを呼び出し元関数のコードに直接コピーする関数です。関数コード・セグメントとの間で制御権を移動するのではなく、変更された関数本体のコピーを関数呼び出しの代わりに直接使用することができます。このようにすると、関数呼び出しのパフォーマンス・オーバーヘッドを回避することができます。
inline 指定子の使用は、インライン展開が実行可能であることをコンパイラーに提案するものにすぎません。コンパイラーはこの提案を無視してもかまいません。

C すべての関数 (main を除く) は、 inline 関数指定子を使用して、インラインとして宣言または定義できます。インライン関数の本体内で、静的ローカル変数を定義することはできません。

C++ クラス宣言内でインプリメントされる C++ 関数は、自動的にインラインとして定義されます。クラス宣言の外側で宣言される通常の C++ 関数およびメンバー関数は、main を除いて、inline 関数指定子を使用して、インラインとして宣言または定義できます。インライン関数の本体内で定義される静的ローカルおよびストリング・リテラルは、すべての変換単位で同じオブジェクトとして処理されます。詳細については、 177 ページの『インライン関数のリンケージ』を参照してください。

以下のコードは、インライン関数の定義を示しています。

```
inline int add(int i, int j) { return i + j; }
```

inline 指定子を使用しても、関数の意味は変わりません。ただし、関数のインライン展開を行うと、実引数の評価の順序が保たれなくなる場合があります。

インライン関数を最も効率良くコーディングするには、ヘッダー・ファイル内にインライン関数定義を配置してから、インライン化する関数への呼び出しを含む任意のファイルにそのヘッダーをインクルードします。

注: inline 指定子は、以下のキーワードで表されます。

- C __inline__ キーワードは、すべての言語レベルでサポートされます。C99 により、inline キーワードのサポートが追加されます。
- C++ inline および __inline__ キーワードは、すべての言語レベルで認識されます。

関連情報

- 187 ページの『noinline 関数属性』
- 「*ILE C/C++ コンパイラー参照*」にある **LANGLVL**

インライン関数のリンケージ

C のみ

C では、インライン関数は、デフォルトで、*静的* リンケージを持つものとして処理されます。つまり、単一の変換単位内でのみ可視になります。そのため、以下の例では、関数 foo はまったく同じように定義されているのにもかかわらず、ファイル A の foo とファイル B の foo は、別々の関数として処理されます。つまり、2 つの関数本体が生成されて、メモリー内に 2 つの異なるアドレスが割り当てられます。

```
// File A

#include <stdio.h>

__inline__ int foo(){
return 3;
}

void g() {
printf("foo called from g: return value = %d, address = %#p\n", foo(), &foo);
}


// File B

#include <stdio.h>

__inline__ int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d, address = %#p\n", foo(), &foo);
g();
}
```

コンパイルされたプログラムの出力は、以下のようになります。

```
foo called from main: return value = 3, address = A100000000000000D8ED5D51EA000B68
foo called from g: return value = 3, address = A100000000000000D8ED5D51EA000B58
```

インライン関数は、内部リンケージを持つものとして処理されるため、インライン関数定義は、別の変換単位内にある、名前が同じ関数の通常の外部定義と共存できます。ただし、インライン定義を含むファイルから関数を呼び出した場合には、コンパイラーは、同じファイル内に定義されたインライン・バージョンと呼び出し用に別のファイルで定義された外部バージョンのどちらか を選択する可能性があります。プログラムは、呼び出されるインライン・バージョンに依存してはなりません。 以下の例では、関数 g からの foo の呼び出しは、6 または 3 を返す可能性があります。

```
// File A

#include <stdio.h>

__inline__ int foo(){
return 6;
}

void g() {
printf("foo called from g: return value = %d\n", foo());
}


// File B

#include <stdio.h>

int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d\n", foo());
g();
}
```

同様に、関数を extern inline として定義した場合、または inline 関数を extern として再宣言した場合には、関数は単に通常の外部関数になり、インライン化されません。

C のみ の終り

C++ のみ。

インライン関数は、関数の使用または呼び出しを行うすべての変換単位でまったく同じように定義する必要があります。さらに、関数は、inline として定義されているが、同じ変換単位内で使用されることも呼び出されることもない場合には、コンパイラーによって破棄されます。

それにもかかわらず、C++ ではインライン関数は、デフォルトで、*外部* リンケージを持つものとして処理されます。これは、プログラムが、関数のコピーが 1 つしか存在しないかのように動作することを意味します。すべての変換単位で、関数のアドレスは同じものになり、すべての変換単位で、すべての静的ローカルおよびストリング・リテラルが共有されます。そのため、前の例をコンパイルすると、出力は以下のようになります。

```
foo called from main: return value = 3, address =  A100000000000000D8ED5D51EA000B58
foo called from g: return value = 3, address = A100000000000000D8ED5D51EA000B68
```

同じ名前のインライン関数を、別の関数本体で再定義するのは正しくありません。ただし、コンパイラーは、これを行ってもエラーのフラグを立てず、コンパイル・コマンド行で入力された最初のファイルで定義されているバージョンの関数本体を単に生成し、それ以外のものを破棄します。そのため、以下の例では、インライン関数 foo を 2 つのファイルで別の方法で定義していますが、期待したような結果は得られません。

```
// File A

#include <stdio.h>

inline int foo(){
return 6;
}
```

```
void g() {
printf("foo called from g: return value = %d, address = %#p\n", foo(), &foo);
}


// File B

#include <stdio.h>

inline int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d, address = %#p\n", foo(), &foo);
g();
}
```

ファイル A およびファイル B が単一の ILE プログラムにバインドされる場合には、出力は以下のようになります。

```
foo called from main: return value = 6, address = A100000000000000F3551B782F000B38
foo called from g: return value = 6, address = A100000000000000F3551B782F000B38
```

main からの foo の呼び出しでは、ファイル B で提供されるインライン定義は使用せず、ファイル A で定義されている通常の外部関数として foo を呼び出しています。予期せぬ結果が生じないように、同じ名前のインライン関数定義がすべての変換単位で完全に一致するようにユーザーが配慮する必要があります。

インライン関数は外部リンケージを持つものとして処理されるため、インライン関数の本体内で定義されるすべての静的ローカル変数またはストリング・リテラルは、すべての変換単位で同じオブジェクトとして処理されます。次の例は、このことを示しています。

```
// File A

#include <stdio.h>

inline int foo(){
static int x = 23;
printf("address of x = %#p\n", &x);
x++;
return x;
}

void g() {
printf("foo called from g: return value = %d\n", foo());
}


// File B

#include <stdio.h>

inline int foo()
{
static int x=23;
printf("address of x = %#p\n", &x);
x++;
return x;
}

void g();
```

```
int main() {
printf("foo called from main: return value = %d\n", foo());
g();
}
```

このプログラムの出力は、**foo** の両方の定義での x が実際に同じオブジェクトであることを示しています。

```
address of x = A100000000000000F3551B782F000B38
foo called from main: return value = 24
address of x = A100000000000000F3551B782F000B38
foo called from g: return value = 25
```

インラインとして定義された関数の各インスタンスを確実に別個の関数として処理する場合には、各変換単位の関数定義で、static 指定子を使用することにより行えます。ただし、静的インライン関数は、テンプレートのインスタンス生成時に名前ルックアップから除去されるため、検索されなくなります。

関連情報

- 174 ページの『static ストレージ・クラス指定子』
- 174 ページの『extern ストレージ・クラス指定子』

C++ のみ。 の終り

関数からの戻りの型指定子

関数の結果は、その関数の*戻り値* と呼ばれ、戻り値のデータ型は、*戻りの型* と呼ばれます。

C++ すべての関数宣言および関数定義では、実際に値を返すかどうかによらず、戻りの型を指定する必要があります。

C 関数宣言で戻りの型を指定しなければ、コンパイラーは、暗黙の戻りの型 int を仮定します。ただし、C99 に準拠するためには、関数が int を返すかどうかによらず、すべての関数宣言および関数定義で戻りの型を指定してください。

関数は、配列型または関数型を除き、任意の型の値を返すように定義できます。配列型または関数型を処理する場合には、配列または関数へのポインターを返す必要があります。関数が値を返さない場合には、関数宣言および関数定義の型指定子は void です。

関数は、volatile 型または const 型のデータ・オブジェクトを戻すものとして宣言することはできません。しかし、volatile または const オブジェクトへのポインターを戻すことはできます。

関数には、ユーザー定義型の戻りの型を指定できます。次に例を示します。

```
enum count {one, two, three};
enum count counter();
```

C ユーザー定義型は、関数宣言内でも定義できます。 C++ ユーザー定義型は、関数宣言内では定義できません。

```
enum count{one, two, three} counter();   // legal in C
enum count{one, two, three} counter();   // error in C++
```

C++ 関数に対する戻りの型として、参照も使用できます。参照は、参照しているオブジェクトの左辺値を戻します。

関連情報

- 45 ページの『型指定子』

関数からの戻り値

C 戻りの型が `void` であるように関数を定義した場合には、値を返してはなりません。 C++ C++ では、戻りの型が `void` であるように定義された関数、およびコンストラクターまたはデストラクターである関数は、値を返してはなりません。

C 戻りの型が `void` 以外であるように関数を定義した場合には、値を返す必要があります。

C++ 戻りの型を指定して定義した関数には、返す値を含む式を組み込む必要があります。

関数が値を返すときには、その値は、`return` ステートメントを介して関数の呼び出し元に返されます。返される前に、値は、定義されている関数の戻りの型に暗黙的に変換されます。以下のコードは、`return` ステートメントを含む関数定義を示しています。

```
int add(int i, int j)
{
  return i + j; // return statement
}
```

以下のコードで示すように、関数 `add()` を呼び出すことができます。

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

この例では、return ステートメントは、戻された型の変数を初期化します。変数 `answer` を `int` の値 30 で初期化しています。戻された式の型を、戻りの型と照らし合わせて検査します。必要に応じて、すべての標準型変換およびユーザー定義の型変換が適用されます。

関数が呼び出されるたびに、自動ストレージを持つその変数の新しいコピーが作成されます。関数が終了した後に、これらの自動変数のストレージは再利用されることがあるので、自動変数を指すポインターまたは参照は戻してはなりません。 C++ クラス・オブジェクトが戻された場合、そのクラスに copy コンストラクターまたはデストラクターがあるときには、一時オブジェクトを作成することがあります。

関連情報

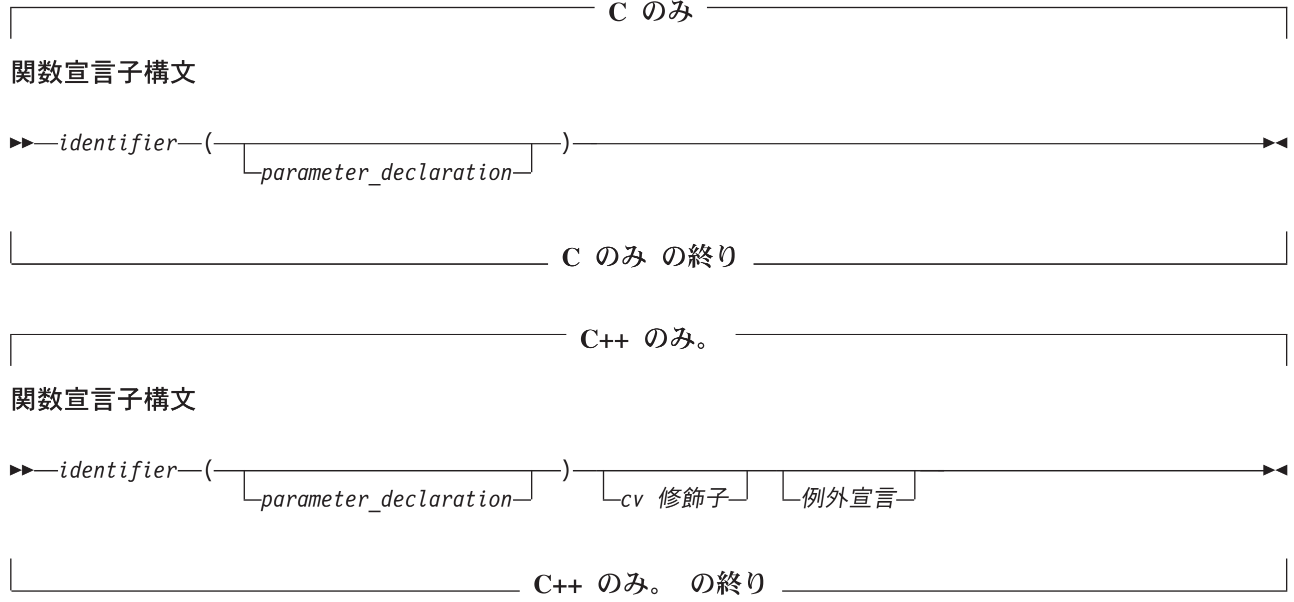
- 166 ページの『return ステートメント』
- 211 ページの『代入の多重定義』
- 213 ページの『添え字の多重定義』
- 40 ページの『自動ストレージ・クラス指定子』

関数宣言子

関数宣言子は、以下の要素で構成されます。

- *ID* (名前)
- パラメーター宣言。関数呼び出しで関数に渡すことができるパラメーターを指定します。
- C++ 例外宣言。これは `throw` 式を含みます。例外の指定については、 331 ページの『第 16 章 例外処理 (C++ のみ)』で説明します。

- C++ 型修飾子 const および volatile。クラス・メンバー関数でのみ使用します。内容については、 232 ページの『定数および volatile メンバー関数』で説明します。



関連情報

- 193 ページの『C++ 関数のデフォルト引数 (C++ のみ)』

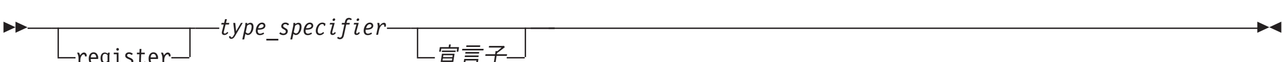
パラメーター宣言

関数宣言子には、別の関数に呼び出されたとき、または自身によって呼び出されたときに、関数に渡すことができるパラメーターのリストを入れます。

C++ C++ では、関数のパラメーター・リストは、その関数のシグニチャー と呼ばれます。関数の名前とシグニチャーにより、その関数が一意的に識別されます。この言葉自体が表しているように、関数のシグニチャーは、多重定義された関数の異なるインスタンスを区別するために、コンパイラーによって使用されます。

関数仮パラメーター宣言構文

パラメーター



C++ のみ。

関数宣言または関数定義で空の引数リストを指定した場合には、引数を取らない関数になります。関数が引

数を取らないことを明示的に示す方法は 2 つあります。空のパラメーター・リストを指定して関数を宣言するか、キーワード void を指定して関数を宣言します。

```
int f(void);
int f();
```

C++ のみ。 の終り

C のみ

関数定義 で空の引数リストを指定した場合には、引数を取らない関数になります。関数*宣言* で空の引数リストを指定した場合には、関数は任意の数または型の引数を取ることができるようになります。そのため、次のようになります。

```
int f()
{
...
}
```

これは、関数 f が引数を取らないことを示します。一方、

```
int f();
```

これは、パラメーターの数および型は不明であることを単に示します。関数が引数を取らないことを明示的に示すには、キーワード void を指定して関数を定義する必要があります。

C のみ の終り

パラメーター指定の終わりにある省略符号は、関数が、可変数のパラメーターを持っていることを指定するために使用します。パラメーターの数は、パラメーター指定の数に等しいか、またはそれより多くなります。

```
int f(int, ...);
```

C++ 省略符号の前のコンマは、オプションです。さらに、パラメーター宣言は、省略符号の前には必要ありません。

C 1 つ以上のパラメーター宣言、同じく省略符号の前のコンマは、C では両方とも必須です。

関連情報

- 48 ページの『void 型』
- 45 ページの『型指定子』
- 62 ページの『型修飾子』
- 342 ページの『例外指定』

パラメーター型

関数*宣言* またはプロトタイプでは、各パラメーターの型を指定する必要があります。 C++ 関数定義 でも、各パラメーターの型を指定する必要があります。 C 関数定義 では、パラメーターの型が指定されていない場合には、 int であると仮定されます。

ユーザー定義型の変数は、(x がはじめて宣言される) 以下の例のように、パラメーター宣言で定義できます。

```
struct X { int i; };
void print(struct X x);
```

C ユーザー定義型は、パラメーター宣言内でも定義できます。 C++ ユーザー定義型は、パラメーター宣言内では定義できません。

```
void print(struct X { int i; } x);   // legal in C
void print(struct X { int i; } x);   // error in C++
```

パラメーターの名前

関数定義 では、各パラメーターに ID を指定する必要があります。関数宣言 またはプロトタイプでは、ID の指定はオプションです。そのため、以下の例は、関数宣言では許可されます。

```
int func(int,long);
```

C++ のみ。

以下の制限は、関数宣言内のパラメーターの名前の使用に適用されます。

- 単一の宣言内で、2 つのパラメーターの名前を同じにすることはできません。
- パラメーターの名前が関数外の名前と同じである場合、関数外の名前は隠され、パラメーター宣言でこの名前を使用することはできません。次の例では、3 番目のパラメーター名 intersects は、列挙型 subway_line を持つことを意味します。しかし、この名前は、最初のパラメーターの名前によって隠蔽されています。関数 subway() の宣言は、subway_line が有効な型名ではないので、コンパイル時エラーを引き起こします。なぜ有効でないかというと、最初のパラメーター名 subway_line が、ネーム・スペース・スコープ enum 型を隠蔽し、2 番目のパラメーターで再度使用することができないからです。

```
enum subway_line {yonge,
university, spadina, bloor};
int subway(char * subway_line, int stations,
                  subway_line intersects);
```

C++ のみ。 の終り

関数仮パラメーター宣言における静的配列指標 (C のみ)

特定のコンテキスト内を除いて、サブスクリプトされていない配列名 (例えば、region[4] の代わりに region) は、配列がすでに宣言されている場合、配列の最初のエレメントのアドレスを値に持つポインターを表します。 また、関数のパラメーター・リスト内の配列型も、対応するポインター型に変換されます。引数配列のサイズ情報は、配列が関数本体内部からアクセスされるときに失われます。

最適化に有益なこの情報を保存するには、C99 を使用して static キーワードを使用する引数配列の指標を宣言できます。定数式は、最適化の前提事項として使用できる最小のポインター・サイズを指定します。static キーワードのこの特定の使用は、厳格に規定されています。このキーワードは、最外部配列型派生および関数仮パラメーター宣言においてのみ表示することができます。関数の呼び出し元がこれらの制限に従わないと、未定義の振る舞いがとられることになります。

フィーチャーの使用例を次に示します。

```
void foo(int arr [static 10]);       /* arr points to the first of at least
                                           10 ints                         */
void foo(int arr [const 10]);        /* arr is a const pointer             */
void foo(int arr [static const i]);  /* arr points to at least i ints;
                                           i is computed at run time.      */
void foo(int arr [const static i]);  /* alternate syntax to previous example */
void foo(int arr [const]);           /* const pointer to int               */
```

関連情報

- 40 ページの『static ストレージ・クラス指定子』
- 78 ページの『配列』
- 128 ページの『配列添え字演算子 [ ]』

関数属性

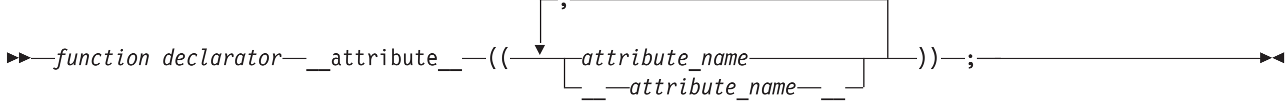
IBM 拡張

関数属性は、GNU C で開発されたプログラムの移植性を向上させるためにインプリメントされている拡張です。関数に対して指定可能な属性を使用すると、コンパイラーによる関数呼び出しの最適化を補助し、コンパイラーに対してコードをより多面的に検査するよう明示的に指示することができます。追加機能を提供する属性もあります。

関数属性は、キーワード `__attribute__` に続いて、属性名、および属性名に必要な追加の引数がある場合にはその引数を設定して、指定します。関数 `__attribute__` の指定は、関数の宣言または定義に組み込みます。構文は次の形式をとります。

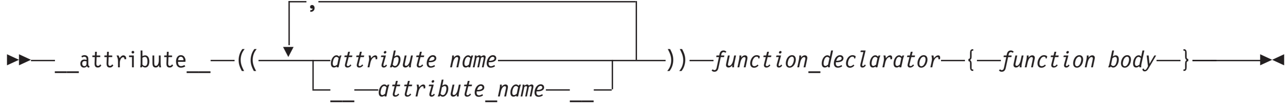
C++

関数属性の構文: 関数宣言



C のみ

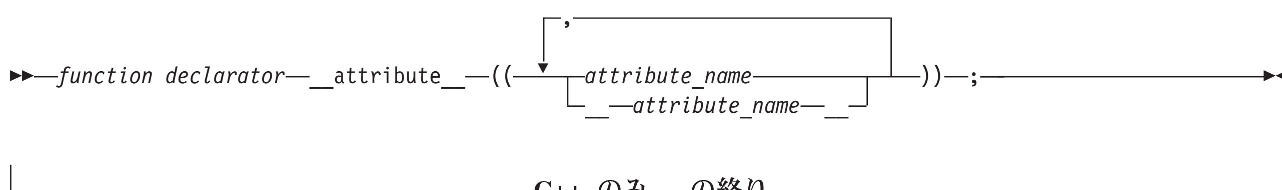
関数属性の構文: 関数定義



C のみ の終り

C++ のみ。

関数属性の構文: 関数定義



C++ のみ。 の終り

関数宣言内の関数属性は常に、括弧で囲まれたパラメーター宣言を含む宣言子の後に配置します。

```
/* Specify the attribute on a function prototype declaration */
void f(int i, int j) __attribute__((individual_attribute_name));
void f(int i, int j) { }
```

C++ のみ。

C++ では、関数で存在する可能性があるすべての例外宣言の後に、属性の指定が続く必要もあります。

C++ のみ。 の終り

C のみ

旧スタイルのパラメーター宣言の構文解析にあいまいさがあるため、関数定義では、属性指定は宣言子の*前*で行わなければなりません。

```
int __attribute__((individual_attribute_name)) foo(int i) { }
```

C のみ の終り

▶ C++ *__attribute_name__* という形式 (すなわち、前後に 2 つの下線文字が付いた属性名) を使用した関数属性を指定すると、同じ名前のマクロとの名前の競合の可能性が低くなります。

次の関数属性がサポートされます。

- 187 ページの『noinline 関数属性』
- 187 ページの『pure 関数属性』
- 187 ページの『weak 関数属性』

関連情報

- 89 ページの『変数属性』

const 関数属性

const 関数属性は、その関数の呼び出し回数が、ソース・コードで指定された回数より少なくても問題ないことをコンパイラーに知らせます。この言語フィーチャーを使用して、関数はその引数以外の値を検査しないこと、およびその戻り値以外は何の影響も及ぼさないことを示すことにより、明示的にコンパイラーによるコードの最適化を補助することができます。

const 関数属性の構文

```
▶▶──__attribute__──((──┬─const─────┬─))──────────────────────────────────▶◀
                       └─__const__─┘
```

次の種類の関数には、const を宣言しないでください。

- 指し示されたデータを調べるポインター引数を持つ関数。
- const 関数以外の関数を呼び出す関数。

関連情報

- 「*ILE C/C++ コンパイラー参照*」の **#pragma isolated_call**

noinline 関数属性

noinline 関数属性を使用すると、適用される関数は、その関数がインラインとして宣言されているか非インラインとして宣言されているかによらず、インライン化されなくなります。この属性は、インライン化コンパイラー・オプションおよび inline キーワードよりも優先されます。

noinline 関数属性の構文

```
►►──__attribute__──((──┬─noinline─────┬──))────────────────────────────────────►◄
                       └─__noinline__─┘
```

この属性は、インライン化は防ぎますが、インライン関数のセマンティクスを除去することはありません。

pure 関数属性

pure 関数属性は、その関数の呼び出し回数が、ソース・コード明記された回数より少なくてもよい関数を宣言します。属性 pure で関数を宣言すると、その関数は、パラメーター、グローバル変数、またはその両方にだけ依存する戻り値以外は何の影響も与えないことが示されます。

pure 関数属性の構文

```
►►──__attribute__──((──┬─純粋─────┬──))──────────────────────────────────────►◄
                       └─__pure__─┘
```

関連情報

- 「*ILE C/C++ コンパイラー参照*」の **#pragma isolated_call**

weak 関数属性

weak 関数属性があると、関数宣言の結果によるシンボルは、オブジェクト・ファイルの中に、グローバル・シンボルとしてではなく、弱いシンボルとして現れます。この言語フィーチャーを使用すると、ライブラリー関数を作成するプログラマーは、名前の重複エラーを生じることなく、ユーザー・コードの関数定義によってライブラリー関数宣言をオーバーライドできます。

weak 関数属性の構文

```
►►──__attribute__──((──┬─weak─────┬──))──────────────────────────────────────►◄
                       └─__weak__─┘
```

関連情報

- 「*ILE C/C++ コンパイラー参照*」の **#pragma weak**
- 92 ページの『weak 変数属性』

IBM 拡張 の終り

main() 関数

プログラムが実行を開始すると、システムは main 関数を呼び出します。この関数は、プログラムのエントリー・ポイントを表します。デフォルトでは、main は、ストレージ・クラス extern を持ちます。どのプログラムにも、main という名前の関数が 1 つなければならず、次の制限が適用されます。

- プログラムにあるそれ以外の関数を main と呼ぶことはできません。
- main を inline または static として定義することはできません。
- C++ main をプログラム内から呼び出すことはできません。
- C++ main のアドレスを取得することはできません。
- C++ main 関数は、多重定義できません。

関数 main は、以下の形式のいずれかを使用して、パラメーターを指定して、または指定せずに定義できます。

```
int main (void)
```

```
int main ()
```

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

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

どのような名前もこれらのパラメーターに付けることはできますが、それらは通常、argc および argv と呼ばれています。第 1 パラメーターの argc (引数カウント) は、プログラムの開始時にコマンド行で入力された引数の数を示す整数です。第 2 パラメーターの argv (引数ベクトル) は、文字オブジェクトの配列へのポインターの配列です。配列オブジェクトはヌル終了ストリングであり、プログラムの開始時にコマンド行に入力された引数を示します。

配列の第 1 エレメントの argv[0] は、コマンド行から実行されているプログラムのプログラム名または呼び出し名が入った文字配列へのポインターです。 argv[1] は、プログラムに渡された第 1 引数を示し、argv[2] は第 2 引数を示し、というように続きます。 以下のプログラム例 backward は、コマンド行に入力された引数を、最後の引数が最初に出力されるように出力します。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
  printf("\n");
}
```

以下を指定してコマンド行からこのプログラムを呼び出します。

```
backward string1 string2
```

出力は、次のとおりです。

```
string2 string1
```

引数の argc と argv には以下の値が入ります。

オブジェクト	値
argc	3
argv[0]	ストリング "backward" を指すポインター
argv[1]	ストリング "string1" を指すポインター
argv[2]	ストリング "string2" へのポインター

オブジェクト	値
argv[3]	NULL

関連情報

- 41 ページの『extern ストレージ・クラス指定子』
- 176 ページの『inline 関数指定子』
- 40 ページの『static ストレージ・クラス指定子』
- 『関数呼び出し』

関数呼び出し

関数は、宣言して定義した後には、プログラム内のどこからでも (main 関数内からでも、別の関数からでも、さらにはその関数自体からでも)、*呼び出す* ことができます。関数の呼び出しでは、関数名に続いて、関数呼び出し演算子、および関数が受け取る必要があるデータ値がある場合にはそのデータ値を指定します。これらの値は、関数に定義したパラメーターに対する*引数* で、今説明したプロセスのことを、関数に*引数を渡す* といいます。

C++ 関数は、まだ宣言されていない場合には、呼び出されない場合があります。

引数は、以下の 2 つの方法で渡すことができます。

- 値による受け渡し。引数の*値* を、呼び出し先関数の対応するパラメーターにコピーします。
- 参照による受け渡し。引数の*アドレス* を、呼び出し先関数の対応するパラメーターに渡します。

C++ のみ。

クラスに、デストラクターまたはビット単位のコピー以上を行うコピー・コンストラクターがある場合、クラス・オブジェクトを値で渡すと、実際には参照によって渡される一時オブジェクトが構築されます。

関数引数がクラス・オブジェクトであり、以下の特性のすべてを持っている場合には、これはエラーです。

- クラスが copy コンストラクターを必要としている。
- クラスに、ユーザー定義の copy コンストラクターがない。
- そのクラス用に copy コンストラクターを生成することができない。

C++ のみ。 の終り

関連情報

- 101 ページの『関数引数変換』
- 109 ページの『関数呼び出し式』
- 279 ページの『コンストラクター』

値による受け渡し

値 による受け渡しを使用した場合には、コンパイラーは、呼び出し元関数の引数の値を、呼び出し先関数定義の対応する非ポインターまたは非参照パラメーターにコピーします。 呼び出し先関数内のパラメータ

ーは、渡された引数の値を使用して初期化されます。パラメーターが定数として宣言されていない限り、パラメーターの値を変更できます。ただし、変更は呼び出し先関数のスコープ内でのみ実行され、呼び出し元関数の引数の値は影響を受けません。

次の例では、main から func に、5 および 7 の 2 つの値が渡されます。関数 func は、これらの値のコピーを受け取り、ID a と b によって、それらにアクセスします。関数 func は、値 a を変更します。main に制御が戻るときには、x と y の実際の値は、変わっていません。

```
/**
 ** This example illustrates calling a function by value
 **/

#include <stdio.h>

void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d\n", a, b);
}

int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d\n", x, y);
   return 0;
}
```

このプログラムの出力は、次のようになります。

```
In func, a = 12   b = 7
In main, x = 5    y = 7
```

参照による受け渡し

*参照による受け渡し*とは、呼び出し元関数の引数のアドレスを、呼び出し先関数の対応するパラメーターに渡す方法のことを指します。 C C では、呼び出し先関数の対応するパラメーターは、ポインター型として宣言する必要があります。 C++ C++ では、対応するパラメーターは、ポインター型だけではなく、任意の参照型として宣言できます。

このようにすることで、呼び出し元関数の引数の値を呼び出し先関数で変更できます。

次の例は、参照によって引数がどのように渡されるかを示しています。 C++ では、この関数を呼び出すと、参照パラメーターが実引数によって初期化されます。 C では、この関数を呼び出すと、ポインター・パラメーターがポインター値によって初期化されます。

C++ のみ。

```
#include <stdio.h>

void swapnum(int &i, int &j) {
  int temp = i;
  i = j;
  j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;
```

```
  swapnum(a, b);
  printf("A is %d and B is %d¥n", a, b);
  return 0;
}
```

C++ のみ。 の終り

C のみ

```
#include <stdio.h>

void swapnum(int *i, int *j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;

  swapnum(&a, &b);
  printf("A is %d and B is %d¥n", a, b);
  return 0;
}
```

C のみ の終り

関数 swapnum() の呼び出し時に、変数 a および b の実際の値は、参照によって渡されているため、交換されます。出力は次のとおりです。

```
A is 20 and B is 10
```

C++ のみ。

const で修飾されている参照を変更するためには、const_cast 演算子を使用して、その const 型をキャストする必要があります。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

void f(const int& x) {
  int& y = const_cast<int&>(x);
  y++;
}

int main() {
  int a = 5;
  f(a);
  cout << a << endl;
}
```

この例では 6 を出力します。

C++ のみ。 の終り

関連情報

- 80 ページの『参照 (C++ のみ)』
- 137 ページの『const_cast 演算子 (C++ のみ)』

割り振りおよび割り振り解除関数 (C++ のみ)

ユーザーは、クラス・メンバー関数またはグローバル・ネーム・スペース関数としての、ユーザー自身の new 演算子あるいは割り振り関数を、以下の制限の下で定義することができます。

- 最初のパラメーターは、型 std::size_t のパラメーターでなければなりません。デフォルトのパラメーターを持つことはできません。
- 戻りの型は、型 void* でなければなりません。
- 割り振り関数はテンプレート関数でもかまいません。最初のパラメーターも、戻りの型もテンプレート・パラメーターに依存しません。
- 空の例外指定 throw() を指定して割り振り関数を宣言する場合、関数が失敗すると、割り振り関数はヌル・ポインターを戻す必要があります。そうしなければ、ユーザーの関数は、失敗した場合には、型 std::bad_alloc の例外または std::bad_alloc から派生したクラスを throw する必要があります。

ユーザーは、ユーザー自身の delete 演算子、あるいはクラス・メンバー関数またはグローバル・ネーム・スペース関数としての割り振り解除関数を、以下の制限の下で定義することができます。

- この最初のパラメーターの型は void* でなければなりません。
- 戻りの型は、型 void でなければなりません。
- 割り振り解除関数はテンプレート関数でもかまいません。最初のパラメーターも、戻りの型もテンプレート・パラメーターに依存しません。

次の例では、グローバル・ネーム・スペース new および delete の置換関数を定義しています。

```
#include <cstdio>
#include <cstdlib>

using namespace std;

void* operator new(size_t sz) {
  printf("operator new with %d bytes¥n", sz);
  void* p = malloc(sz);
  if (p == 0) printf("Memory error¥n");
  return p;
}

void operator delete(void* p) {
  if (p == 0) printf ("Deleting a null pointer¥n");
  else {
    printf("delete object¥n");
    free(p);
  }
}

struct A {
  const char* data;
  A() : data("Text String") { printf("Constructor of S¥n"); }
  ~A() { printf("Destructor of S¥n"); }
};

int main() {
  A* ap1 = new A;
  delete ap1;

  printf("Array of size 2:¥n");
  A* ap2 = new A[2];
  delete[] ap2;
}
```

次に、上記の例の出力を示します。

```
operator new with 16 bytes
Constructor of S
Destructor of S
delete object
Array of size 2:
operator new with 48 bytes
Constructor of S
Constructor of S
Destructor of S
Destructor of S
delete object
```

関連情報

- 141 ページの『new 式 (C++ のみ)』

C++ 関数のデフォルト引数 (C++ のみ)

関数仮パラメーターに対してデフォルト値を与えることができます。次に例を示します。

```
#include <iostream>
using namespace std;

int a = 1;
int f(int a) { return a; }
int g(int x = f(a)) { return x; }

int h() {
  a = 2;
  {
    int a = 3;
    return g();
  }
}

int main() {
  cout << h() << endl;
}
```

この例では、標準出力に 2 を印刷します。その理由は、 g() の宣言で参照される a は、ファイル・スコープのもので、この値は、g() が呼び出されるときは 2 であるためです。

デフォルト引数は、暗黙的に、パラメーター型へ変換可能でなければなりません。

関数へのポインターは、その関数と同じ型でなければなりません。関数の型を指定せずに参照によって関数のアドレスを得ようとすると、エラーになります。関数の型は、デフォルト値を持つ引数によって影響を受けません。

以下の例は、デフォルト引数が、関数の型の一部とは見なされていないことを示しています。デフォルト引数を使用すると、すべての引数を指定しなくても関数を呼び出すことができます。すべての引数の型を指定しない関数へのポインターを作成することはできません。関数 f は、明示的引数がなくても呼び出すことができますが、ポインター badpointer は、引数の型を指定しないと定義することができません。

```
int f(int = 0);
void g()
{
   int a = f(1);              // ok
   int b = f();               // ok, default argument used
}
int (*pointer)(int) = &f;     // ok, type of f() specified (int)
int (*badpointer)() = &f;     // error, badpointer and f have
```

```
                                // different types. badpointer must
                                // be initialized with a pointer to
                                // a function taking no arguments.
```

この例では、関数 f3 は、戻りの型 int を持っています。そして、その関数が取る int 引数は、関数 f2 から戻された値であるデフォルト値を持ちます。

```
      const int j = 5;
      int f3( int x = f2(j) );
```

関連情報

- 195 ページの『関数へのポインター』

デフォルト引数に関する制約事項

演算子の中で、多重定義時にデフォルト引数を持つことができるのは、関数呼び出し演算子と演算子 new だけです。

デフォルト引数を持つパラメーターは、関数宣言パラメーター・リスト内の末尾のパラメーターでなければなりません。次に例を示します。

```
void f(int a, int b = 2, int c = 3);  // trailing defaults
void g(int a = 1, int b = 2, int c);  // error, leading defaults
void h(int a, int b = 3, int c);      // error, default in middle
```

いったん宣言または定義でデフォルト引数を指定すると、その引数を、同じ値にする場合であっても、再定義することはできません。ただし、それまでの宣言で指定されていないデフォルト引数を追加することはできます。例えば、次の例の最後の宣言では、a および b に対するデフォルト値を再定義しようとしています。

```
void f(int a, int b, int c=1);     // valid
void f(int a, int b=1, int c);     // valid, add another default
void f(int a=1, int b, int c);     // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

関数宣言または定義では、いかなるデフォルト引数値でも与えることができます。デフォルト引数値に続くパラメーター・リストの中のパラメーターはすべて、関数の、この宣言または前の宣言に指定されたデフォルト引数値を持っていなければなりません。

デフォルト引数の式では、ローカル変数を使用することはできません。例えば、コンパイラーは、次の g() および h() の両方の関数に対してエラーとします。

```
void f(int a)
{
      int b=4;
      void g(int c=a); // Local variable "a" cannot be used here
      void h(int d=b); // Local variable "b" cannot be used here
}
```

関連情報

- 109 ページの『関数呼び出し式』
- 141 ページの『new 式 (C++ のみ)』

デフォルト引数の評価

デフォルト引数を使用して定義された関数が、末尾の引数が欠落している状態で呼び出されると、デフォルト式が評価されます。次に例を示します。

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);            // same as call f(a,2,3)
f(a,10);         // same as call f(a,10,3)
f(a,10,20);      // no default arguments
```

デフォルト引数は、関数宣言に対して検査されます。そして、関数が呼び出されるときに評価されます。デフォルトの引数の評価の順序は、定義されていません。デフォルト引数式では、関数の他のパラメーターは使用できません。次に例を示します。

```
int f(int q = 3, int r = q); // error
```

q の値は、r に代入されるときには決まっていない場合があるので、引数 r を引数 q の値で初期化することはできません。上記の関数宣言を、次のように書き直すものとします。

```
int q=5;
int f(int q = 3, int r = q); // error
```

関数宣言内の r の値はやはりエラーとなります。それは、関数の外側で定義された変数 q が、関数に対して宣言されている引数 q によって隠されているからです。同様に、

```
typedef double D;
int f(int D, int z = D(5.3) ); // error
```

ここでは、型 D は、関数宣言内で整数の名前として解釈されます。型 D は、引数 D により隠されます。したがって、D が引数の名前であり、型ではないため、キャスト D(5.3) が、キャストとして解釈されません。

次の例では、非静的メンバー a を初期化指定子として使用できません。 a は、クラス X のオブジェクトが作成されるまで存在しないからです。 b は、クラス X のどのオブジェクトとも無関係に作成されるので、静的メンバー b を初期化指定子として使用することができます。デフォルト値は、クラス宣言の最後の大括弧 } の後まで分析されないので、メンバー b をデフォルト引数として使用した後で、それを宣言することができます。

```
class X
{
   int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

関数へのポインター

関数へのポインターは、その関数の実行可能コードのアドレスを示します。ポインターを使用して関数を呼び出し、関数を引数として他の関数に渡すことができます。関数へのポインターに対してポインター演算を行うことはできません。

関数の戻りの型とパラメーター型の両方によって、関数へのポインターの型が決まります。

関数へのポインターの宣言では、ポインター名を小括弧に入れることが必要です。関数呼び出し演算子 () は、間接参照演算子 * よりも高い優先順位を持っています。括弧がないと、コンパイラーは、指定された戻りの型へのポインターを戻す関数として、そのステートメントを解釈します。次に例を示します。

```
int *f(int a);       /* function f returning an int*                    */
int (*g)(int a);     /* pointer g to a function returning an int        */
char (*h)(int, int)  /* h is a function
                          that takes two integer parameters and returns char */
```

最初の宣言では、`f` は、`int` を引数として取り、 `int` へのポインターを戻す関数として解釈します。 2 番目の宣言では、`g` を、`int` 引数を受け取り、`int` を戻す関数へのポインターと解釈します。

関連情報

- 9 ページの『言語リンケージ』
- 74 ページの『ポインター』
- 99 ページの『ポインター型変換』
- 174 ページの『extern ストレージ・クラス指定子』

第 9 章 ネーム・スペース (C++ のみ)

ネーム・スペース は、オプションとして命名されたスコープです。クラスまたは列挙に対してするように、ネーム・スペース内で名前を宣言します。ネストされたクラス名にアクセスするのと同じように、スコープ・レゾリューション (::) 演算子を使用することによって、ネーム・スペース内で宣言された名前にアクセスできます。ただし、ネーム・スペースは、クラスや列挙型が持つ追加のフィーチャーを持ちません。ネーム・スペースの主要な目的は、追加の ID (ネーム・スペースの名前) を名前に追加することです。

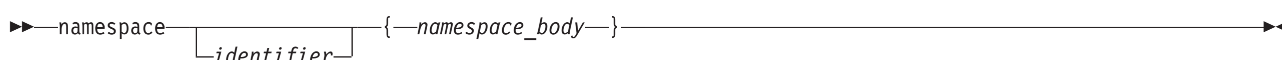
関連情報

- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』

ネーム・スペースの定義

ネーム・スペースを一意的に識別するためには、**namespace** キーワードを使用します。

ネーム・スペースの構文



オリジナルのネーム・スペース定義の中の *identifier* (ID) は、ネーム・スペースの名前です。オリジナルのネーム・スペース定義が現れる宣言領域では、ネーム・スペースを拡張するケースを除いて、ID は前に定義されていないことがあります。 ID が使用されない場合、そのネーム・スペースは、*名前なしネーム・スペース* です。

関連情報

- 200 ページの『名前なしネーム・スペース』

ネーム・スペースの宣言

ネーム・スペース名に使用される ID は、固有でなければなりません。前にグローバル ID として使用されていてはなりません。

```
namespace Raymond {
  // namespace body here...
  }
```

この例では、Raymond は、ネーム・スペースの ID です。ネーム・スペースのエレメントにアクセスする意図がある場合、ネーム・スペースの ID は、すべての変換単位で既知である必要があります。

関連情報

- 3 ページの『ファイル/グローバル・スコープ』

ネーム・スペース別名の作成

特定のネーム・スペース ID を参照するために、代替名を使用できます。

© Copyright IBM Corp. 1998, 2010

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  void f();
}

namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

この例では、IBM ID は INTERNATIONAL_BUSINESS_MACHINES の別名です。これは、長いネーム・スペース ID を参照するのに有用です。

ネーム・スペース名または別名が、同じ宣言領域内の他のエンティティーの名前として宣言されると、コンパイラー・エラーの結果を生じます。また、グローバル・スコープで定義されたネーム・スペース名が、プログラムのいずれかのグローバル・スコープ内の他のエンティティーの名前として宣言されると、コンパイラー・エラーの結果を生じます。

関連情報

- 3 ページの『ファイル/グローバル・スコープ』

ネストされたネーム・スペースの別名の作成

ネーム・スペース定義は、宣言を持っています。ネーム・スペース定義は宣言そのものであるので、ネーム・スペース定義をネストすることができます。

ネストされたネーム・スペースに、別名を適用することもできます。

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  int j;
    namespace NESTED_IBM_PRODUCT {
      void a() { j++; }
  int j;
      void b() { j++; }
  }
}
namespace NIBM = INTERNATIONAL_BUSINESS_MACHINES::NESTED_IBM_PRODUCT
```

この例では、NIBM ID は、ネーム・スペース NESTED_IBM_PRODUCT の別名です。このネーム・スペースは、INTERNATIONAL_BUSINESS_MACHINES ネーム・スペース内でネストされます。

関連情報

- 197 ページの『ネーム・スペース別名の作成』

ネーム・スペースの拡張

ネーム・スペースは拡張可能です。前に定義されたネーム・スペースに、後続の宣言を追加できます。拡張部分は、オリジナルのネーム・スペース定義から分離されたファイル、またはオリジナルのネーム・スペース定義に付加されたファイルに現れます。次に例を示します。

```
namespace X { // namespace definition
  int a;
  int b;
  }

namespace X { // namespace extension
  int c;
  int d;
  }

namespace Y { // equivalent to namespace X
  int a;
```

```
  int b;
  int c;
  int d;
  }
```

この例では、namespace X は、a および b を使用して定義され、後で c および d を使用して拡張されます。その結果、 namespace X は 4 つのメンバーを含むようになっています。すべての必要なメンバーを 1 つのネーム・スペース内に宣言することもできます。この方式は namespace Y によって表されています。このネーム・スペースには a、b、c、および d が含まれています。

ネーム・スペースと多重定義

複数のネーム・スペースにまたがって関数を多重定義することができます。次に例を示します。

```
// Original X.h:
  f(int);

// Original Y.h:
  f(char);

// Original program.c:
  #include "X.h"
  #include "Y.h"

void z()
{
  f('a'); // calls f(char) from Y.h
}
```

ソース・コードを大幅に変更することなく、ネーム・スペースを上記の例に導入できます。

```
// New X.h:
namespace X {
  f(int);
  }

// New Y.h:
namespace Y {
  f(char);
  }

// New program.c:
  #include "X.h"
  #include "Y.h"

  using namespace X;
  using namespace Y;

void z()
{
  f('a'); // calls f() from Y.h
}
```

program.c で、関数 void z() は、ネーム・スペース Y のメンバーである関数 f() を呼び出します。using ディレクティブをヘッダー・ファイルに入れると、program.c のソース・コードは、変更されないままです。

関連情報

- 205 ページの『第 10 章 多重定義 (C++ のみ)』

名前なしネーム・スペース

左中括弧の前に ID のないネーム・スペースは、*名前なしネーム・スペース*になります。各変換単位は、それ自体の固有の名前なしネーム・スペースを含むことができます。次の例は、名前なしネーム・スペースがいかに有用であるかを示しています。

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   int variable;
   }

int main()
{
   cout << i << endl;
   variable = 100;
   return 0;
}
```

上記の例で、名前なしネーム・スペースは、スコープ・レゾリューション演算子を使用しないで `i` および `variable` にアクセスすることを許可しています。

名前なしネーム・スペースを、不適切に使用している例を以下に示します。

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   }

int i = 2;

int main()
{
   cout << i << endl; // error
   return 0;
}
```

コンパイラーは、グローバル名と、同じ名前を持つ名前なしネーム・スペースのメンバーとを区別できないので、`main` の内部では `i` はエラーの原因となります。 上記の例が作用するためには、ネーム・スペースは ID によって一意的に識別される必要があります。そして、`i` は、使用しているネーム・スペースを指定する必要があります。

名前なしネーム・スペースは、同じ変換単位内で拡張できます。次に例を示します。

```
#include <iostream>

using namespace std;

namespace {
   int variable;
   void funct (int);
   }

namespace {
   void funct (int i) { cout << i << endl; }
   }

int main()
```

```
{
   funct(variable);
   return 0;
}
```

funct のプロトタイプおよび定義の両方共、同じ名前なしネーム・スペースのメンバーです。

注: 名前なしネーム・スペースで定義された項目は、内部結合を持っています。キーワード static を使用して、内部結合を持つ項目を定義するよりも、代わりに名前なしネーム・スペースでそれらの項目を定義します。

関連情報

- 7 ページの『プログラム・リンケージ』
- 8 ページの『内部結合』

ネーム・スペース・メンバー定義

ネーム・スペースは、それ自体の内部、または明示的修飾を使用して外部で、自身のメンバーを定義できます。以下に、ネーム・スペースが、内部的にメンバーを定義する例を示します。

```
namespace A {
  void b() { /* definition */ }
}
```

ネーム・スペース A 内で、メンバー void b() が内部的に定義されます。

ネーム・スペースは、定義されようとしている名前に明示的修飾を使用して、外部的にそのメンバーを定義することもできます。定義されようとしているエンティティーは、ネーム・スペース内ですでに宣言されている必要があります。定義は、宣言のネーム・スペースを囲むネーム・スペース内の、宣言のポイントの後に現れる必要があります。

ネーム・スペースが、外部的にメンバーを指定する例を以下に示します。

```
namespace A {
  namespace B {
    void f();
  }
  void B::f() { /* defined outside of B */ }
}
```

この例では、関数 f() は、ネーム・スペース B 内で宣言され、A 内で (B の外で) 定義されています。

ネーム・スペースとフレンド

最初にネーム・スペース内で宣言された名前はすべて、そのネーム・スペースのメンバーです。非ローカルのクラス内のフレンド宣言が、最初にクラスまたは関数を宣言する場合、そのフレンド・クラスまたは関数は、最内部の囲みネーム・スペースのメンバーです。

この構成の例を以下に示します。

```
// f has not yet been defined
void z(int);
namespace A {
  class X {
    friend void f(X);  // A::f is a friend
    };
  // A::f is not visible here
  X x;
```

```
  void f(X) { /* definition */}  // f() is defined and known to be a friend
}

using A::x;

void z()
{
  A::f(x);    // OK
  A::X::f(x); // error: f is not a member of A::X
}
```

この例では、関数 f() は、呼び出し A::f(s); を使用して、ネーム・スペース A によってのみ、呼び出すことができます。 A::X::f(x); 呼び出しを使用して class X によって、関数 f() を呼び出そうとする試みは、コンパイラー・エラーの結果になります。フレンド宣言は、最初に非ローカルのクラス内で行われるので、フレンド関数は、最内部の囲みネーム・スペースのメンバーです。このフレンド関数は、そのネーム・スペースによってのみアクセスすることができます。

関連情報

- 245 ページの『フレンド』

using ディレクティブ

using ディレクティブは、すべてのネーム・スペース修飾子およびスコープ演算子へのアクセスを提供します。これは、using キーワードをネーム・スペース ID に適用することによって行われます。

using ディレクティブの構文

```
►►──using──namespace──name──;──────────────────────────────────────────►◄
```

name は、前に定義されたネーム・スペースでなければなりません。 using ディレクティブは、グローバルおよびローカルのスコープで適用できますが、クラス・スコープでは適用できません。ローカル・スコープは、類似の宣言を隠蔽することによって、グローバル・スコープに優先します。

スコープが、2 番目のネーム・スペースを指名する using ディレクティブを含んでいる場合、そしてその 2 番目のネーム・スペースが別の using ディレクティブを含んでいる場合、 2 番目のネーム・スペースの using ディレクティブは、あたかも 1 番目のスコープ内に常駐しているかのように振る舞います。

```
namespace A {
  int i;
}
namespace B {
  int i;
  using namespace A;
}
void f()
{
  using namespace B;
  i = 7; // error
}
```

初期設定この例で、関数 f() 内で i を初期化しようとすると、コンパイラー・エラーを生じます。なぜなら、関数 f() は、どの i を呼び出すのか、つまりネーム・スペース A から i を呼び出すのか、またはネーム・スペース B から i を呼び出すのか、を知ることができないからです。

関連情報

- 258 ページの『using 宣言とクラス・メンバー』

using 宣言とネーム・スペース

using 宣言は、特定のネーム・スペース・メンバーへのアクセスを提供します。これは、対応するネーム・スペース・メンバーを持っているネーム・スペース名に using キーワードを適用することによって行われます。

using 宣言の構文

```
►►──using──namespace──::──member──────────────────────────────────────►◄
```

この構文図で、修飾子名が using 宣言の後にきます。そして、*member* が修飾子名の後にきます。宣言が機能するには、メンバーは与えられたネーム・スペース内で宣言される必要があります。次に例を示します。

```
namespace A {
  int i;
  int k;
  void f;
  void g;
  }

using A::k
```

この例では、using 宣言の後にネーム・スペース A の名前である A がきます。次にその後にスコープ演算子 (::) および k がきます。この形式は、using 宣言によって、ネーム・スペース A の外で k にアクセスすることを可能にします。 using 宣言を出した後、その特定のネーム・スペースに対して行われたすべての拡張は、using 宣言が行われた時点では、認識されません。

特定の関数の多重定義されたバージョンは、その特定の関数の宣言の前に、ネーム・スペースに含める必要があります。 using 宣言は、ネーム・スペース、ブロックおよびクラス・スコープの中に置くことができます。

関連情報

- 258 ページの『using 宣言とクラス・メンバー』

明示的アクセス

ネーム・スペースのメンバーを明示的に修飾するには、ネーム・スペース ID を :: スコープ・レゾリューション演算子と一緒に使用します。

明示的アクセス修飾の構文

```
►►──namespace_name──::──member──────────────────────────────────────►◄
```

次に例を示します。

```
namespace VENDITTI {
  void j()
};

VENDITTI::j();
```

この例では、スコープ・レゾリューション演算子は、ネーム・スペース VENDITTI 内に保持されている関数 j へのアクセスを提供します。スコープ・レゾリューション演算子 :: は、グローバルおよびローカルの両

方のネーム・スペース内の ID にアクセスするために使用されます。アプリケーションの中の ID はすべて、十分な修飾によってアクセスできます。明示的なアクセスは、名前なしネーム・スペースには適用できません。

関連情報

- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』

第 10 章 多重定義 (C++ のみ)

関数名または演算子に対して同じスコープ内で複数の定義を指定すると、その関数名または演算子を*多重定義した* ことになります。多重定義された関数と演算子については、それぞれ『関数の多重定義』と 207 ページの『演算子の多重定義』に説明されています。

多重定義された宣言 は、同じスコープで前に宣言された宣言と同じ名前を使用して宣言された宣言です。ただし、両方の宣言は、異なる型を持っています。

多重定義された関数名または演算子を呼び出す場合、コンパイラーは、関数または演算子を呼び出すのに使用した引数型を、定義に指定されているパラメーター型と比較することによって、使用するのに最も適切な定義を判別します。 215 ページの『多重定義解決』 に示すように、最も適切な多重定義された関数または演算子を選択するプロセスは、*多重定義解決* と呼ばれています。

関数の多重定義

同じスコープ内で名前 `f` を持つ関数を複数個宣言すると、関数名 `f` を多重定義することになります。 `f` の宣言は、型または引数リスト中の引数の数、またはその両方で、互いに異なるものでなければなりません。 `f` という名前の多重定義された関数を呼び出すとき、関数呼び出しの引数リストを、名前 `f` を持つ多重定義された候補関数のそれぞれのパラメーター・リストと比較することによって、正しい関数が選択されます。*候補関数* とは、多重定義された関数名の呼び出しのコンテキストに基づいて呼び出すことのできる関数のことです。

関数 `print` (`int` を表示する) について考えてみましょう。次の例に示すように、関数 `print` を多重定義して、他の型 (例えば、`double` および `char*`) を表示することができます。異なるデータ型に対して、それぞれ同様のオペレーションを実行する、同じ名前の関数を 3 つ持つことができます。

```
#include <iostream>
using namespace std;

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}

void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
}
```

次に、上記の例の出力を示します。

```
 Here is int 10
 Here is float 10.1
 Here is char* ten
```

関連情報

© Copyright IBM Corp. 1998, 2010

- 『多重定義関数に関する制約事項』
- 253 ページの『派生』

多重定義関数に関する制約事項

以下の関数宣言は、同じスコープ内にあっても、それらを多重定義することはできません。このリストは、明示的に宣言された関数および using 宣言によって導入された関数にのみ適用されることに注意してください。

- 戻りの型が異なるだけの関数宣言。例えば、以下の宣言を宣言することはできません。

```
int f();
float f();
```

- 同じ名前および同じパラメーター型を持つメンバー関数宣言。ただし、これらの宣言のうちの 1 つは、静的メンバー関数宣言です。例えば、以下の f() の 2 つのメンバー関数宣言を宣言することはできません。

```
struct A {
  static int f();
  int f();
};
```

- 同じ名前、同じパラメーター型、および同じテンプレート・パラメーター・リストを持つメンバー関数テンプレート宣言。ただし、これらの宣言のうちの 1 つは、静的テンプレート・メンバー関数宣言です。
- 等価のパラメーター宣言を持つ関数宣言。これらの宣言は、同じ関数を宣言することになるので、許可されません。
- 同じ型を表す typedef 名の使用のみが異なるパラメーターを持つ関数宣言。 typedef は、別の型名に同義語で、独立した型を表すのではないことに注意してください。例えば、次の 2 つの f の宣言は、同じ関数の宣言です。

```
typedef int I;
void f(float, int);
void f(float, I);
```

- 一方はポインター、もう一方は配列であることのみが異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

```
f(char*);
f(char[10]);
```

  パラメーターを差別化するときに、最初の配列次元が無意味で、他のすべての配列次元が有効であるもの。例えば、次の宣言は、同じ関数の宣言です。

```
g(char(*)[20]);
g(char[5][20]);
```

  次の 2 つの宣言は、等価ではありません。

```
g(char(*)[20]);
g(char(*)[40]);
```

- 一方は関数型、もう一方は同じ型の関数を指すポインターであることによってのみ異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

```
void f(int(float));
void f(int (*)(float));
```

- cv 修飾子 const、volatile、および restrict のせいのみで異なるパラメーターを持つ関数宣言。この制限事項は、これらの修飾子のいずれかがパラメーター型指定の最外部レベルにある場合にのみ適用されます。例えば、次の宣言は、同じ関数の宣言です。

```
int f(int);
int f(const int);
int f(volatile int);
```

const 修飾子、volatile 修飾子、および restrict 修飾子を持つパラメーターは、パラメーター型指定*内で* 適用することによって区別することができるので、注意してください。例えば、以下の宣言は等価ではありません。理由は、const と volatile が * ではなく int を修飾しているために、パラメーター型指定の最外部レベルに存在しないからです。

```
void g(int*);
void g(const int*);
void g(volatile int*);
```

次の宣言も等価ではありません。

```
void g(float&);
void g(const float&);
void g(volatile float&);
```

- それらのデフォルトの引数が異なっているということのみが異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

```
void f(int);
void f(int i = 10);
```

- extern "C" 言語リンケージおよび同じ名前を持つ、複数の関数 (それらのパラメーター・リストが異なっているかどうかに関係なく)。

関連情報

- 203 ページの『using 宣言とネーム・スペース』
- 60 ページの『typedef 定義』
- 62 ページの『型修飾子』
- 9 ページの『言語リンケージ』

演算子の多重定義

C++ のほとんどの組み込み演算子の関数を、再定義または多重定義することができます。これらの演算子は、グローバルに、またはクラス単位で多重定義できます。多重定義された演算子は、関数としてインプリメントされ、メンバー関数またはグローバル関数になることができます。

多重定義された演算子は、*演算子関数* と呼ばれます。演算子の前にキーワード operator を置いて、演算子関数を宣言します。多重定義された演算子は、多重定義された関数とは別個のものです。ただし、多重定義された関数と同様、演算子で使用されるオペランドの数と型によって区別されます。

標準の + (プラス) 演算子について考えます。この演算子が異なる標準型のオペランドで使用される場合は、演算子の意味が多少変わります。例えば、2 個の整数の加算は、2 個の浮動小数点数の加算と同様にはインプリメントされていません。 C++ では、標準の C++ 演算子をクラス型に適用するときに、それらにユーザー独自の意味を定義することができます。

次の演算子は、いずれも多重定義できます。

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	<<=	>>=	==	!=
<=	>=	&&	||	++	--	,	->*	->
( )	[ ]	new	delete	new[]	delete[]			

ここで、() は関数呼び出し演算子、 [] は添え字演算子です。

下記の演算子の単項形式と 2 項形式の両方共、多重定義が可能です。

+	-	*	&

次の演算子は、多重定義できません。

.	.*	::	?:

プリプロセッサー記号の # と ## は、多重定義できません。

演算子関数は、非静的メンバー関数であってもかまわないし、あるいは、クラス、クラスへの参照、列挙、または列挙型への参照であるパラメーターを少なくとも 1 つ持っている、非メンバー関数であってもかまいません。

演算子の優先順位、グループ分け、またはオペランドの数は変更できません。

多重定義された演算子 (関数呼び出し演算子を除く) は、引数リストの中にデフォルト引数や省略符号を入れることはできません。

多重定義された =、[]、()、および -> の各演算子は、第 1 オペランドとして必ず左辺値を受け取ることができるように、非静的メンバー関数として宣言する必要があります。

演算子 new、delete、new[]、および delete[] は、本節で説明する一般規則には従いません。

= 演算子を除く全演算子は継承されます。

単項演算子の多重定義

パラメーターを持たない非静的メンバー関数、またはパラメーターを 1 つ持っている非メンバー関数のいずれかを使用して、単項演算子を多重定義します。単項演算子 @ は、ステートメント @t の形で呼び出されるものと想定します。ここで、t は、型 T のオブジェクトです。この演算子を多重定義する非静的メンバー関数は、以下の形式になっています。

```
return_type operator@()
```

同じ演算子を多重定義する非メンバー関数は、以下の形式になっています。

```
return_type operator@(T)
```

多重定義された単項演算子は、どのような型でも戻すことができます。

次の例では、! 演算子を多重定義します。

```
#include <iostream>
using namespace std;
```

```
struct X { };

void operator!(X) {
  cout << "void operator!(X)" << endl;
}

struct Y {
  void operator!() {
    cout << "void Y::operator!()" << endl;
  }
};

struct Z { };

int main() {
  X ox; Y oy; Z oz;
  !ox;
  !oy;
//  !oz;
}
```

次に、上記の例の出力を示します。

```
void operator!(X)
void Y::operator!()
```

演算子関数呼び出し `!ox` は、`operator!(X)` と解釈されます。呼び出し `!oy` は、`Y::operator!()` と解釈されます。 (コンパイラーは、`!oz` を許可しません。なぜなら、`!` 演算子は、クラス `Z` に対して定義されていないからです。)

関連情報

- 110 ページの『単項式』

増分演算子と減分演算子の多重定義

1 個のクラス型の引数かクラス型への参照を持つ非メンバー関数演算子を使用して、あるいは引数のないメンバー関数演算子を使用して、前置増分演算子 (++) を多重宣言します。

次の例では、増分演算子は以下に示す両方の方法で多重定義されます。

```
class X {
public:

  // member prefix ++x
  void operator++() { }
};

class Y { };

// non-member prefix ++y
void operator++(Y&) { }

int main() {
  X x;
  Y y;

  // calls x.operator++()
  ++x;

  // explicit call, like ++x
  x.operator++();

  // calls operator++(y)
  ++y;
```

```
  // explicit call, like ++y
  operator++(y);
}
```

2 つの引数 (1 番目はクラス型、2 番目は型 int を持っている) を持つ非メンバー関数演算子 operator++() を宣言することによって、あるクラス型に対して、後置増分演算子 (++) を多重定義することができます。あるいは、型 int の 1 個の引数を持つ、メンバー関数演算子 operator++() を宣言することができます。コンパイラーは int 引数を使用して、前置増分演算子と後置増分演算子を区別します。暗黙の呼び出しの場合、デフォルト値はゼロです。

次に例を示します。

```
class X {
public:

  // member postfix x++
  void operator++(int) { };
};

class Y { };

// nonmember postfix y++
void operator++(Y&, int) { };

int main() {
  X x;
  Y y;

  // calls x.operator++(0)
  // default argument of zero is supplied by compiler
  x++;
  // explicit call to member postfix x++
  x.operator++(0);

  // calls operator++(y, 0)
  y++;

  // explicit call to non-member postfix y++
  operator++(y, 0);
}
```

前置減分演算子と後置減分演算子は、対応する増分演算子と同じ規則に従います。

関連情報

- 111 ページの『増分演算子 ++』
- 111 ページの『減分演算子 --』

2 項演算子の多重定義

パラメーターを 1 つ持っている非静的メンバー関数、またはパラメーターを 2 つ持っている非メンバー関数のいずれかを使用して、2 項演算子を多重定義します。 2 項演算子 @ は、ステートメント t @ u の形で呼び出されるものと想定します。ここで、t は、型 T のオブジェクト、u は、型 U のオブジェクトです。この演算子を多重定義する非静的メンバー関数は、以下の形式になっています。

```
return_type operator@(T)
```

同じ演算子を多重定義する非メンバー関数は、以下の形式になっています。

```
return_type operator@(T, U)
```

多重定義された 2 項演算子は、どのような型でも戻すことができます。

次の例では、* 演算子を多重定義します。

```
struct X {

  // member binary operator
  void operator*(int) { }
};

// non-member binary operator
void operator*(X, float) { }

int main() {
  X x;
  int y = 10;
  float z = 10;

  x * y;
  x * z;
}
```

呼び出し x * y は、x.operator*(y) と解釈されます。呼び出し x * z は、operator*(x, z) と解釈されます。

関連情報

- 118 ページの『2 項式』

代入の多重定義

パラメーターを 1 つだけ持っている非静的メンバー関数を使用して、代入演算子 operator= を多重定義します。非メンバー関数である、多重定義された代入演算子を宣言することはできません。次の例では、特定のクラスについて、代入演算子を多重定義する方法を示します。

```
struct X {
  int data;
  X& operator=(X& a) { return a; }
  X& operator=(int a) {
    data = a;
    return *this;
  }
};

int main() {
  X x1, x2;
  x1 = x2;      // call x1.operator=(x2)
  x1 = 5;       // call x1.operator=(5)
}
```

代入 x1 = x2 は、コピー代入演算子 X& X::operator=(X&) を呼び出します。代入 x1 = 5 は、コピー代入演算子 X& X::operator=(int) を呼び出します。ユーザー自身があるクラスのコピー代入演算子を定義しない場合、コンパイラーがそれを暗黙的に宣言します。 したがって、派生クラスのコピー代入演算子 (operator=) が、その基底クラスのコピー代入演算子を隠蔽します。

ただし、コピー代入演算子はいずれも、仮想として宣言できます。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
  A& operator=(char) {
    cout << "A& A::operator=(char)" << endl;
```

```
    return *this;
  }
  virtual A& operator=(const A&) {
    cout << "A& A::operator=(const A&)" << endl;
    return *this;
  }
};

struct B : A {
    B& operator=(char) {
      cout << "B& B::operator=(char)" << endl;
      return *this;
    }
    virtual B& operator=(const A&) {
      cout << "B& B::operator=(const A&)" << endl;
      return *this;
    }
};

struct C : B { };

int main() {
  B b1;
  B b2;
  A* ap1 = &b1;
  A* ap2 = &b1;
  *ap1 = 'z';
  *ap2 = b2;

  C c1;
//  c1 = 'z';
}
```

次に、上記の例の出力を示します。

```
A& A::operator=(char)
B& B::operator=(const A&)
```

代入 `*ap1 = 'z'` は、`A& A::operator=(char)` を呼び出します。この演算子は `virtual` と宣言されていないので、コンパイラーは、ポインター `ap1` の型に基づいて関数を選択します。代入 `*ap2 = b2` は、`B& B::operator=(const &A)` を呼び出します。この演算子は `virtual` と宣言されているため、コンパイラーはポインター `ap1` が指すオブジェクトの型に基づいて関数を選択します。クラス `C` で宣言された、暗黙的に宣言されたコピー代入演算子は、 `B& B::operator=(char)` を隠蔽するので、コンパイラーは代入 `c1 = 'z'` を許可しません。

関連情報

- 295 ページの『コピー代入演算子』
- 119 ページの『代入演算子』

関数呼び出しの多重定義

関数呼び出し演算子は、多重定義された場合には、関数が呼び出される方法を変更しません。むしろ、演算子が与えられた型のオブジェクトに適用された場合の、演算子の解釈の仕方を変更します。

任意の数のパラメーターを持つ非静的メンバー関数を使用して、関数呼び出し演算子 `operator()` を多重定義します。あるクラスに対して関数呼び出し演算子を多重定義する場合、その宣言は以下の形式になります。

return_type operator()(*parameter_list*)

他のすべての多重定義された演算子と異なり、関数呼び出し演算子の引数リスト内に、デフォルト引数と省略符号を指定することができます。

次の例は、コンパイラーがどのように関数呼び出し演算子を解釈するかを示しています。

```
struct A {
  void operator()(int a, char b, ...) { }
  void operator()(char c, int d = 20) { }
};

int main() {
  A a;
  a(5, 'z', 'a', 0);
  a('z');
//  a();
}
```

関数呼び出し a(5, 'z', 'a', 0) は、a.operator()(5, 'z', 'a', 0) と解釈されます。これは void A::operator()(int a, char b, ...) を呼び出します。 関数呼び出し a('z') は、a.operator()('z') と解釈されます。これは、void A::operator()(char c, int d = 20) を呼び出します。コンパイラーは、関数呼び出し a() を許可しません。なぜなら、その引数リストが、クラス A に定義された関数呼び出しパラメーター・リストのいずれにも一致しないからです。

次の例は、多重定義された関数呼び出し演算子を示しています。

```
class Point {
private:
  int x, y;
public:
  Point() : x(0), y(0) { }
  Point& operator()(int dx, int dy) {
    x += dx;
    y += dy;
    return *this;
  }
};

int main() {
  Point pt;

  // Offset this coordinate x with 3 points
  // and coordinate y with 2 points.
  pt(3, 2);
}
```

上記の例は、クラス Point のオブジェクトの関数呼び出し演算子を再解釈しています。 Point のオブジェクトを関数のように扱って、2 つの整数引数を渡す場合、関数呼び出し演算子は、渡した引数の値を、それぞれ Point::x および Point::y に加えます。

関連情報

- 109 ページの『関数呼び出し式』

添え字の多重定義

パラメーターを 1 つだけ持っている非静的メンバー関数を使用して、operator[] を多重定義します。次の例は、多重定義された添え字演算子を持っている単純配列クラスです。多重定義された添え字演算子は、ユーザーが指定された境界の外で配列にアクセスしようとすると、例外をスローします。

```
#include <iostream>
using namespace std;
```

```
template <class T> class MyArray {
private:
  T* storage;
  int size;
public:
  MyArray(int arg = 10) {
    storage = new T[arg];
    size = arg;
  }

  ~MyArray() {
    delete[] storage;
    storage = 0;
  }

  T& operator[](const int location) throw (const char *);
};

template <class T> T& MyArray<T>::operator[](const int location)
  throw (const char *) {
    if (location < 0 || location >= size) throw "Invalid array access";
    else return storage[location];
}

int main() {
  try {
     MyArray<int> x(13);
     x[0] = 45;
     x[1] = 2435;
     cout << x[0] << endl;
     cout << x[1] << endl;
     x[13] = 84;
  }
  catch (const char* e) {
    cout << e << endl;
  }
}
```

次に、上記の例の出力を示します。

```
45
2435
Invalid array access
```

式 x[1] は、x.operator[](1) と解釈されます。そして int& MyArray<int>::operator[](const int) を呼び出します。

関連情報

- 128 ページの『配列添え字演算子 [ ]』

クラス・メンバー・アクセスの多重定義

パラメーターを持っていない非静的メンバー関数を使用して、operator-> を多重定義します。次の例は、コンパイラーがどのように、多重定義されたクラス・メンバー・アクセス演算子を解釈するかを示しています。

```
struct Y {
  void f() { };
};

struct X {
 Y* ptr;
 Y* operator->() {
   return ptr;
```

```
 };
};

int main() {
  X x;
  x->f();
}
```

ステートメント x->f() は、(x.operator->())->f() と解釈されます。

operator-> は、「スマート・ポインター」をインプリメントするために使用されます (しばしばポインター参照解除演算子と組にして)。これらのポインターは、普通のポインターのように振る舞うオブジェクトですが、次の点で異なります。すなわち、それらのポインターによってユーザーがオブジェクトにアクセスするときに、自動オブジェクト削除 (ポインターが破棄されるとき、または別のオブジェクトを指すためにポインターが使用されるとき)、あるいは参照カウント (同じオブジェクトを指すスマート・ポインターの数をカウントし、そして、そのカウントがゼロになったときに、オブジェクトを自動的に削除する) などの別の作業を実行します。

auto_ptr と呼ばれるスマート・ポインターの 1 つの例が、C++ 標準ライブラリーに含まれています。 <memory> ヘッダーの中に、それを見出すことができます。 auto_ptr クラスは、自動オブジェクト削除をインプリメントします。

関連情報

- 110 ページの『矢印演算子 ->』

多重定義解決

最も適切な多重定義された関数または演算子を選択するプロセスは、*多重定義解決* と呼ばれています。

f が、多重定義された関数名であると想定します。多重定義された関数 f() を呼び出す場合、コンパイラーは*候補関数* のセットを作成します。この関数のセットには、f() を呼び出したポイントからアクセスできる、f という名前のすべての関数が含まれています。コンパイラーは、多重定義解決を促進するために、f という名前のこれらのアクセス可能な関数の中の 1 つの関数の代替表記を、候補関数として含めることができます。

候補関数のセットを作成した後、コンパイラーは、*実行可能関数* のセットを作成します。この関数のセットは、候補関数のサブセットです。各実行可能関数のパラメーター数は、f() を呼び出すのに使用した引数の数に一致します。

コンパイラーは、実行可能関数のセットから*最良の実行可能関数* を選択します。これは、f() を呼び出すときに C++ ランタイム環境が使用する関数宣言です。コンパイラーは、*暗黙的変換シーケンス* によって、これを行います。暗黙的変換シーケンスは、関数呼び出しの中の引数を、関数宣言の中の対応するパラメーターの型へ変換するのに必要な変換のシーケンスです。暗黙的変換シーケンスはランク付けされます。いくつかの暗黙的変換シーケンスは、他のものより良いシーケンスです。最良の実行可能関数は、そのすべてのパラメーターが他のすべての実行可能関数よりも良い、または等しいランク付けの暗黙的変換シーケンスを持つものです。コンパイラーは、コンパイラーが複数の最良実行可能関数を検出できたプログラムは、許可しません。暗黙的変換シーケンスについては、 216 ページの『暗黙的変換シーケンス』で詳しく説明します。

可変長配列が関数仮パラメーターであると、左端の配列次元は候補関数間の関数を区別しません。次の例では、void f(int []) がすでに定義されているため、f の 2 番目の定義は許可されません。

```
void f(int a[*]) {}
void f(int a[5]) {} // illegal
```

ただし、可変長配列の左端以外の配列次元は、可変長配列が関数仮パラメーターであると、候補関数を区別します。例えば、関数 f の多重定義セットは以下のもので構成されることがありますが、

```
void f(int a[][5]) {}
void f(int a[][4]) {}
void f(int a[][g]) {}   // assume g is a global int
```

次のものを含むことはできません。

```
void f(int a[][g2]) {} // illegal, assuming g2 is a global int
```

理由は、第 2 レベル配列次元 g および g2 を持つ候補関数があると、関数 f を呼び出す際にあいまいさが生じるからです。つまり、g も g2 もコンパイル時に認識されません。

明示的キャストを使用することによって、正確な一致をオーバーライドすることができます。次の例では、f() に対する 2 回目の呼び出しが f(void*) と一致します。

```
void f(int) { };
void f(void*) { };

int main() {
   f(0xaabb);            // matches f(int);
   f((void*) 0xaabb);    // matches f(void*)
}
```

暗黙的変換シーケンス

*暗黙的変換*シーケンスは、関数呼び出しの中の引数を、関数宣言の中の対応するパラメーターの型へ変換するのに必要な変換のシーケンスです。

コンパイラーは、各引数に対する暗黙的変換シーケンスを決定しようとします。コンパイラーは次に、各暗黙的変換シーケンスを 3 つのカテゴリーの中の 1 つにカテゴリー化し、カテゴリーに応じてそれらをランク付けします。コンパイラーは、引数に対する暗黙的変換シーケンスを検出できないようなプログラムは、許可しません。

以下に、変換シーケンスの 3 つのカテゴリーを、最良から最悪への順序で示します。

- 標準変換シーケンス
- ユーザー定義の変換シーケンス
- 省略符号変換シーケンス

注 : 2 つの標準変換シーケンスまたは 2 つのユーザー定義の変換シーケンスが、異なるランクを持つことがあります。

標準変換シーケンス

標準変換シーケンスは、3 つのランクの中の 1 つにカテゴリー化されます。ランクは、最良から最悪への順序でリストされています。

- 完全一致 : このランクには、以下の変換が含まれます。
  - 識別変換
  - 左辺値から右辺値への変換
  - 配列からポインターへの変換
  - 修飾変換

- 拡張: このランクには整数および浮動小数点拡張が含まれます。
- 変換 : このランクには、以下の変換が含まれます。
  - 整数および浮動小数点変換
  - 浮動 - 整数変換
  - ポインター型変換
  - ポインターからメンバーへの変換
  - ブール変換

コンパイラーは、標準変換シーケンスを、その最悪ランクの標準変換によってランク付けします。例えば、標準変換シーケンスが浮動小数点変換を持っている場合、そのシーケンスは、変換ランクを持っていることになります。

関連情報

- 98 ページの『左辺値から右辺値への変換』
- 99 ページの『ポインター型変換』
- 101 ページの『修飾変換 (C++ のみ)』
- 96 ページの『整数変換』
- 97 ページの『浮動小数点の型変換』
- 96 ページの『ブール変換』

ユーザー定義の変換シーケンス

ユーザー定義の変換シーケンス は、以下のもので構成されています。

- 標準変換シーケンス
- ユーザー定義の変換
- 2 番目の標準変換シーケンス

ユーザー定義の変換シーケンス A およびユーザー定義の変換シーケンス B の両者が、同じユーザー定義の変換関数またはコンストラクターを持っていて、A の 2 番目の標準変換シーケンスが、B の 2 番目の標準変換シーケンスよりも良ければ、前者の方が良いランクの変換シーケンスです。

省略符号変換シーケンス

省略符号変換シーケンス は、コンパイラーが関数呼び出しの中の引数を、対応する省略符号パラメーターに突き合わせるときに生じます。

多重定義された関数のアドレスの解決

引数が指定されていない多重定義された関数名 f を使用する場合、その名前は、関数、関数を指すポインター、メンバー関数を指すポインター、または関数テンプレートの特殊化を参照することができます。引数が指定されなかったので、コンパイラーは、関数呼び出しまたは演算子の使用に対して実行するのと同じ方法では、多重定義解決を実行することができません。その代わりに、コンパイラーは、f を使用した場所に応じて、以下の式のうちの 1 つの式の型に一致する、最良の実行可能関数を選択しようと試みます。

- 初期化しようとしているオブジェクトまたは参照
- 代入の左辺
- 関数またはユーザー定義の演算子のパラメーター
- 関数、演算子、または変換の戻り値

- 明示的型変換

ユーザーが f を使用したときに、コンパイラーが非メンバー関数または静的メンバー関数の宣言を選択した場合、コンパイラーは、その宣言を型「ポインターから関数へ」または「参照から関数へ」の式に突き合わせました。コンパイラーが非静的メンバー関数の宣言を選択した場合、コンパイラーは、その宣言を型「ポインターからメンバー関数へ」の式に突き合わせました。次の例は、このことを示しています。

```
struct X {
  int f(int) { return 0; }
  static int f(char) { return 0; }
};

int main() {
  int (X::*a)(int) = &X::f;
//  int (*b)(int) = &X::f;
}
```

コンパイラーは、関数ポインター b の初期化を許可しません。型 int(int) の非メンバー関数または静的関数は、宣言されていません。

f がテンプレート関数の場合、コンパイラーは、テンプレート引数推論を実行して、どのテンプレート関数を使用するかを決めます。うまくいけば、その関数を実行可能関数のリストに追加します。このセットに非テンプレート関数を含めて複数の関数がある場合、コンパイラーはセットからすべてのテンプレート関数を除去して、非テンプレート関数を選択します。 このセットにテンプレート関数だけがある場合、コンパイラーは、最も特殊化されたテンプレート関数を選択します。次の例は、このことを示しています。

```
template<class T> int f(T) { return 0; }
template<> int f(int) { return 0; }
int f(int) { return 0; }

int main() {
  int (*a)(int) = f;
  a(1);
}
```

関数呼び出し a(1) は、int f(int) を呼び出します。

関連情報

- 195 ページの『関数へのポインター』
- 234 ページの『メンバーへのポインター』
- 308 ページの『関数テンプレート』
- 319 ページの『明示的特殊化』

第 11 章 クラス (C++ のみ)

クラス は、ユーザー定義のデータ型を作成するためのメカニズムの 1 つです。これは、C 言語の構造体のデータ型と似ています。 C では、構造体は、データ・メンバーのセットで構成されます。 C++ では、クラス型は C 構造体と似ていますが、クラスは、データ・メンバーのセット、およびそのクラスで実行できるオペレーションのセットで構成される点が異なります。

C++ において、クラスの型は `union`、`struct`、または `class` というキーワードを用いて宣言することができます。共用体オブジェクトは、名前付きメンバーのセットの 1 つを保持できます。構造体およびクラス・オブジェクトは、全メンバーのセットを保持します。各クラス型はそれぞれ、データ・メンバー、メンバー関数、および他の型名を含む、クラス・メンバーの固有のセットを表します。メンバーに対するデフォルトのアクセスは、クラス・キーによって決まります。

- クラス・キー `class` を用いて宣言されたクラスのメンバーは、デフォルトにより private になります。クラスは、デフォルトで private で継承されます。
- キーワード `struct` を用いて宣言されたクラスのメンバーは、デフォルトでは public になります。構造体は、デフォルトで public で継承されます。
- (キーワード `union` を用いて宣言された) 共用体のメンバーは、デフォルトにより public になります。共用体は、派生における基底クラスとして使用することはできません。

クラス型を作成すると、1 つまたは複数のそのクラス型のオブジェクトを宣言することができます。次に例を示します。

```
class X
{
     /* define class members here */
};
int main()
{
      X xobject1;       // create an object of class type X
      X xobject2;       // create another object of class type X
}
```

C++ には、ポリモアフィック・クラスがあります。ポリモアフィズムにより、コンパイル時において関数が所属するクラスを正確に把握しなくても、別々のクラス (継承に関連した) に現れる関数名を使用することができます。

C++ では、多重定義の概念から、標準の演算子および関数を再定義することができます。演算子の多重定義により、組み込み (標準装備の) 型と同じように簡単にクラスを使用できるので、データ抽出が容易になります。

関連情報

- 49 ページの『構造体および共用体』
- 229 ページの『第 12 章 クラスのメンバーおよびフレンド (C++ のみ)』
- 251 ページの『第 13 章 継承 (C++ のみ)』
- 205 ページの『第 10 章 多重定義 (C++ のみ)』
- 269 ページの『仮想関数』

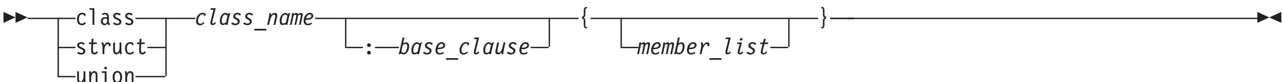
© Copyright IBM Corp. 1998, 2010

クラス型の宣言

クラス宣言は、固有の型のクラス名を作成します。

クラス指定子 は、クラスを宣言する際に使用される型指定子です。そのクラスのメンバー関数がまだ定義されていない場合でも、クラス指定子が見つけられてそのメンバーが宣言されると、クラスは定義されていると見なされます。

クラス指定子構文



class_name は、スコープ内で予約語となる固有の ID です。クラス名が宣言されると、そのクラス名は、囲みスコープ内にある同じ名前の他の宣言を隠蔽します。

member_list は、クラス *class_name* のクラス・メンバー (データと関数の両方) を指定します。クラスの *member_list* が空の場合、そのクラスのオブジェクトのサイズはゼロではありません。クラスのサイズが必要でなければ、クラス指定子自体の *member_list* 内で *class_name* を使用することができます。

これは、クラス *class_name* が継承するメンバーの元の *base_clause* (複数の場合もある) を指定します。*base_clause* が空でない場合、クラス *class_name* は、*派生クラス* と呼ばれます。

構造体 は、*class_key* `struct` を使用して宣言されたクラスです。構造体のメンバーおよび基底クラスは、デフォルトにより public になります。*共用体* は、 *class_key* `union` を使用して宣言されたクラスです。共用体のメンバーは、デフォルトにより public になります。共用体は、一時に 1 つのデータ・メンバーのみ保持します。

集合体クラス は、ユーザー定義のコンストラクター、private またはプロテクトされた非静的データ・メンバー、基底クラス、および仮想関数のないクラスです。

関連情報

- 229 ページの『クラス・メンバー・リスト』
- 253 ページの『派生』

クラス・オブジェクトの使用

クラス型を使用して、そのクラス型のインスタンスつまり*オブジェクト* を作成することができます。例えば、クラス名の X、Y、および Z を指定して、それぞれクラス、構造体、および共用体を宣言することができます。

```
class X {
  // members of class X
};

struct Y {
  // members of struct Y
};

union Z {
  // members of union Z
};
```

これで、各クラス型のオブジェクトを宣言することができます。クラス、構造体、および共用体は、すべて C++ クラス型であることを忘れないでください。

```
int main()
{
      X xobj;      // declare a class object of class type X
      Y yobj;      // declare a struct object of class type Y
      Z zobj;      // declare a union object of class type Z
}
```

C++ では、C とは異なり、クラスの名前が隠れていない限り、クラス・オブジェクトの宣言の前に、キーワード union、struct、および class を入れる必要はありません。次に例を示します。

```
struct Y { /* ... */ };
class X { /* ... */ };
int main ()
{
      int X;            // hides the class name X
      Y yobj;           // valid
      X xobj;           // error, class name X is hidden
      class X xobj;     // valid
}
```

1 つの宣言で複数のクラス・オブジェクトを宣言すると、宣言子は個々に宣言されたように扱われます。例えば、以下のように、クラス S の 2 つのオブジェクトを単一の宣言で宣言する場合、

```
class S { /* ... */ };
int main()
{
      S S,T; // declare two objects of class type S
}
```

この宣言は、以下と同等です。

```
class S { /* ... */ };
int main()
{
      S S;
      class S T;       // keyword class is required
                       // since variable S hides class type S
}
```

しかし、以下と同等ではありません。

```
class S { /* ... */ };
int main()
{
      S S;
      S T;             // error, S class type is hidden
}
```

また、クラスへの参照、クラスへのポインター、およびクラスの配列を宣言することもできます。次に例を示します。

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
int main()
{
      X xobj;
      X &xref = xobj;           // reference to class object of type X
      Y *yptr;                  // pointer to struct object of type Y
      Z zarray[10];             // array of 10 union objects of type Z
}
```

外部、静的、および自動定義内のクラスを初期化できます。初期化指定子には、= (等号) と、その後に続く、中括弧で囲まれ、コンマで分離された値のリストが含まれます。クラスのすべてのメンバーを初期化する必要はありません。

コピー制限のないクラス型のオブジェクトは、関数への引数として、代入または引き渡しを行うことができ、また関数により戻すことができます。

関連情報

- 49 ページの『構造体および共用体』
- 80 ページの『参照 (C++ のみ)』
- 223 ページの『クラス名のスコープ』

クラスおよび構造体

C++ のクラスは、C 言語の構造体の拡張機能です。構造体とクラスの唯一の相違点は、デフォルトによるアクセスが、構造体メンバーは public アクセスで、クラス・メンバーは private アクセスであることです。したがって、キーワードの `class` または `struct` を使用して、同等のクラスを定義できます。

例えば、以下のコードにおいて、クラス `X` は、構造体 `Y` と同等です。

```
class X {

  // private by default
  int a;

public:

  // public member function
  int f() { return a = 5; };
};

struct Y {

  // public by default
  int f() { return a = 5; };

private:

  // private data member
  int a;
};
```

構造体を定義してから、キーワード `class` を使用して、その構造体のオブジェクトを宣言すると、デフォルトによりそのオブジェクトのメンバーは、public のままです。以下の例において、`obj_X` が、クラス・キー `class` を使用する詳述型指定子の使用を宣言していますが、`main()` は、`obj_X` のメンバーへのアクセスを行います。

```
#include <iostream>
using namespace std;

struct X {
 int a;
 int b;
};

class X obj_X;

int main() {
```

```
  obj_X.a = 0;
  obj_X.b = 1;
  cout << "Here are a and b: " << obj_X.a << " " << obj_X.b << endl;
}
```

上記の例の出力は、以下のとおりです。

```
Here are a and b: 0 1
```

関連情報

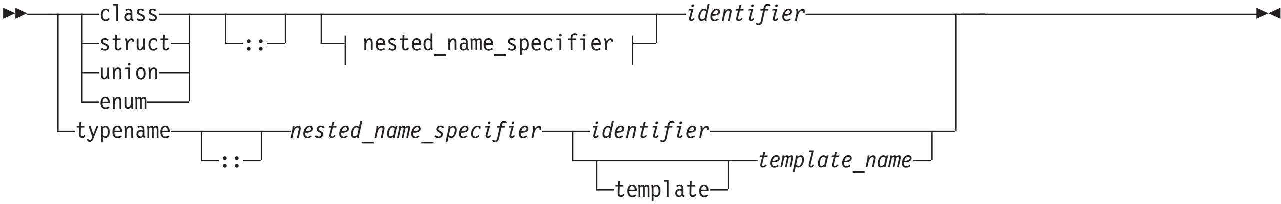
- 49 ページの『構造体および共用体』

クラス名のスコープ

クラス宣言は、クラスが宣言されたスコープ内にクラス名を導入します。囲みスコープ内にある、その名前のクラス、オブジェクト、関数、または他の宣言はすべて隠蔽されます。

クラス名が、同じ名前を持つ関数、列挙子、またはオブジェクトと同じスコープ内で宣言される場合は、*詳述型指定子* を使用してそのクラスを参照してください。

詳述型指定子構文



ネストされた名前指定子:

```
├──┬─class_name─────┬─::──┬─────────────────────────────────┬──────────┤
   └─namespace_name─┘     ├─template─nested_name_specifier─┤
                          └─nested_name_specifier──────────┘
```

以下の例では、関数 A() の定義がクラス A を隠しているため、このクラスを参照するには、詳述型指定子を使用する必要があります。

```
class A { };

void A (class A*) { };

int main()
{
      class A* x;
      A(x);
}
```

宣言 class A* x が、詳述型指定子です。上記で示したような、別の関数、列挙子、またはオブジェクトと同じ名前でクラスを宣言することは、お勧めしません。

また、詳述型指定子をクラス型の不完全な宣言で使用して、現行スコープ内のクラス型のための名前を予約することもできます。

関連情報

- 5 ページの『クラス・スコープ (C++ のみ)』
- 『不完全なクラス宣言』

不完全なクラス宣言

不完全なクラス宣言 とは、クラス・メンバーを定義していないクラス宣言のことです。宣言が完全なものになるまでは、そのクラス型のオブジェクトを宣言したり、クラスのメンバーを参照することはできません。ただし、クラスのサイズが必要でなければ、不完全な宣言を使用して、クラスの定義の前にそのクラスに特定の参照を行うことはできます。

例えば、以下に示すように、構造体 `second` の定義で、構造体 `first` へのポインターを定義することができます。 `second` の定義の前に、不完全なクラス宣言で、構造体 `first` が宣言されています。構造体 `second` 内の `oneptr` の定義では、`first` のサイズは必要ありません。

```
struct first;           // incomplete declaration of struct first

struct second           // complete declaration of struct second
{
      first* oneptr;    // pointer to struct first refers to
                        // struct first prior to its complete
                        // declaration

      first one;        // error, you cannot declare an object of
                        // an incompletely declared class type
      int x, y;
};

struct first            // complete declaration of struct first
{
      second two;       // define an object of class type second
      int z;
};
```

しかし、空のメンバー・リストを指定してクラスを宣言すると、それは完全なクラス宣言となります。次に例を示します。

```
class X;                // incomplete class declaration
class Z {};             // empty member list
class Y
{
public:
      X yobj;           // error, cannot create an object of an
                        // incomplete class type
      Z zobj;           // valid
};
```

関連情報

- 229 ページの『クラス・メンバー・リスト』

ネスト・クラス

ネスト・クラス は、別のクラスのスコープ内で宣言されるものです。ネスト・クラスの名前は、その囲みクラスに対してローカルです。ポインター、参照、またはオブジェクト名を明示的に使用しない限り、ネスト・クラスでの宣言で使用できるのは可視構成だけで、これには、囲みクラスからの型名、静的メンバー、および列挙子と、グローバル変数が含まれます。

ネスト・クラスのメンバー関数は、正規のアクセス規則に従い、囲みクラスのメンバーへの特別なアクセス権を持ちません。囲みクラスのメンバー関数は、ネスト・クラスのメンバーへの特別なアクセスは行いません。次の例は、このことを示しています。

```
class A {
  int x;

  class B { };

  class C {

    // The compiler cannot allow the following
    // declaration because A::B is private:
    //   B b;

    int y;
    void f(A* p, int i) {

    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;

    }
  };

  void g(C* p) {

    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
  }
};

int main() { }
```

コンパイラーは、クラス A::B が private なので、オブジェクト b の宣言を許可しません。コンパイラーは、A::x が private なので、ステートメント p->x = i を許可しません。コンパイラーは、C::y が private なので、ステートメント int i = p->y を許可しません。

ネーム・スペース・スコープで、ネスト・クラスのメンバー関数および静的データ・メンバーを定義することができます。例えば、以下のコードでは、修飾された型名を使用して、静的メンバー x と y およびネスト・クラス nested のメンバー関数 f() と g() にアクセスすることができます。修飾された型名を使用すると、typedef を定義して、修飾されたクラス名を表すことができます。その後、:: (スコープ・レゾリューション) 演算子を用いた typedef を使用して、ネスト・クラスまたはクラス・メンバーを参照することができます。

```
class outside
{
public:
      class nested
      {
public:
            static int x;
            static int y;
            int f();
            int g();
      };
};
int outside::nested::x = 5;
int outside::nested::f() { return 0; };

typedef outside::nested outnest;       // define a typedef
int outnest::y = 10;                   // use typedef with ::
int outnest::g() { return 0; };
```

しかし、ネスト・クラス名を示す typedef を使用すると、情報を隠蔽し、理解が困難なコードが作成されます。

詳述型指定子では、typedef 名を使用できません。例えば、上記の例で次の宣言は、使用できません。

```
  class outnest obj;
```

ネスト・クラスは、その囲みクラスの private メンバーから継承します。次の例は、このことを示しています。

```
class A {
private:
  class B { };
  B *z;

  class C : private B {
  private:
      B y;
//      A::B y2;
      C *x;
//      A::C *x2;
    };
};
```

ネスト・クラス A::C は A::B から継承します。コンパイラーは、A::B も A::C も private なので、宣言 A::B y2 および A::C *x2 を許可しません。

関連情報

- 5 ページの『クラス・スコープ (C++ のみ)』
- 223 ページの『クラス名のスコープ』
- 243 ページの『メンバー・アクセス』
- 239 ページの『静的メンバー』

ローカル・クラス

ローカル・クラス は、関数定義の中で宣言されます。ローカル・クラスでの宣言が使用できるのは、外部変数および関数に加えて、囲みスコープからの型名、列挙、静的変数だけです。

次に例を示します。

```
int x;                         // global variable
void f()                       // function definition
{
      static int y;            // static variable y can be used by
                               // local class
      int x;                   // auto variable x cannot be used by
                               // local class
      extern int g();          // extern function g can be used by
                               // local class

      class local              // local class
      {
            int g() { return x; }      // error, local variable x
                                       // cannot be used by g
            int h() { return y; }      // valid,static variable y
            int k() { return ::x; }    // valid, global x
            int l() { return g(); }    // valid, extern function g
      };
}

int main()
{
      local* z;                // error: the class local is not visible
      // ...}
```

ローカル・クラスのメンバー関数は、メンバー関数が定義される場合、そのクラス定義の中で定義してください。結果として、ローカル・クラスのメンバー関数は、インライン関数です。ローカル・クラスのスコープの中で定義されているメンバー関数は、すべてのメンバー関数と同様に、キーワード inline は必要ありません。

ローカル・クラスが静的データ・メンバーを持つことはできません。以下の例において、ローカル・クラスの静的メンバーを定義しようとするとエラーになります。

```
void f()
{
    class local
    {
       int f();              // error, local class has noninline
                             // member function
       int g() {return 0;}   // valid, inline member function
       static int a;         // error, static is not allowed for
                             // local class
       int b;                // valid, nonstatic variable
    };
}
//      . . .
```

囲み関数は、ローカル・クラスのメンバーに特別なアクセスを行いません。

関連情報

- 231 ページの『メンバー関数』
- 176 ページの『inline 関数指定子』

ローカル型名

ローカル型名は、他の名前と同じスコープ規則に従います。クラス宣言の中で定義される型名にはクラス・スコープがあり、修飾をしなければ、それらのクラスの外側で使用することはできません。

型名で使用されているクラス名、typedef 名、または定数名をクラス宣言で使用すると、その名前をクラス宣言で再定義することはできません。

次に例を示します。

```
int main ()
{
      typedef double db;
      struct st
      {
            db x;
            typedef int db; // error
            db y;
      };
}
```

次の宣言は有効です。

```
typedef float T;
class s {
      typedef int T;
      void f(const T);
};
```

ここで、関数 f() は型 s::T の引数を取ります。しかし、s のメンバーの順序が逆になっている以下の宣言は、エラーとなります。

```
typedef float T;
class s {
      void f(const T);
      typedef int T;
};
```

クラス宣言で一度その名前を使用したクラス名または typedef 名に対して、クラス名または typedef 名でない名前をクラス宣言で再定義することはできません。

関連情報

- 1 ページの『スコープ』
- 60 ページの『typedef 定義』

第 12 章 クラスのメンバーおよびフレンド (C++ のみ)

本節では、情報隠蔽メカニズムに関するクラス・メンバーの宣言について説明するとともに、クラスがフレンド・メカニズムを使用して、クラスの非 public メンバーへの関数およびクラス・アクセスをどのように認可するかについても述べます。 C++ では、情報隠蔽の概念を拡大して、private インプリメンテーションを除くパブリック・クラス・インターフェースを持つという概念を追加しています。これは、プログラム内の関数による、クラス型の内部表記への直接アクセスを制限するためのメカニズムです。

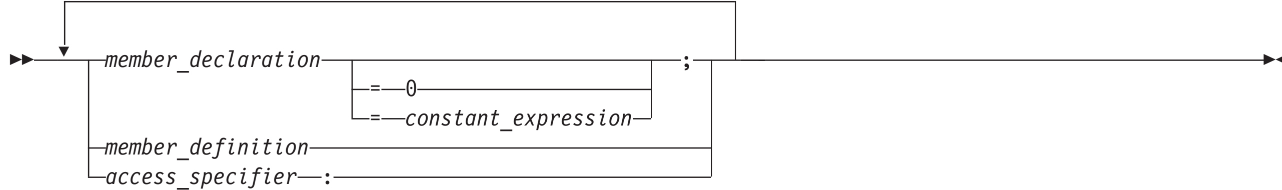
関連情報

- 243 ページの『メンバー・アクセス』
- 256 ページの『継承されたメンバー・アクセス』

クラス・メンバー・リスト

オプションのメンバー・リスト は、クラス・メンバー と呼ばれるサブオブジェクトを宣言します。クラス・メンバーとしては、データ、関数、ネストされた型、および列挙子が可能です。

クラス・メンバー・リスト構文



メンバー・リストは、クラス名の後に続き、中括弧の中に入れられます。以下が、メンバー・リストおよびメンバー・リストのメンバーに適用されます。

- *member_declaration* または *member_definition* は、データ・メンバー、メンバー関数、ネスト型、または列挙型の宣言または定義です。 (また、クラス・メンバー・リストに定義されている列挙型の列挙子も、クラスのメンバーです。)
- メンバー・リストは、ユーザーがクラス・メンバーを宣言できる唯一の場所です。
- フレンド宣言は、クラス・メンバーではありませんが、メンバー・リストに入っていることが必要です。
- クラス定義でのメンバー・リストには、クラスのメンバーをすべて宣言します。メンバーを他の場所で追加することはできません。
- メンバー・リストに、同じメンバーを 2 回宣言することはできません。
- データ・メンバー、またはメンバー関数を `static` として宣言できますが、`auto`、`extern`、または `register` としては宣言できません。
- ネスト・クラス、メンバー・クラス・テンプレート、またはメンバー関数を宣言し、クラスの外側でそれらを定義できます。
- 静的データ・メンバーは、クラスの外側で定義する必要があります。

© Copyright IBM Corp. 1998, 2010

- クラス・オブジェクトである非静的メンバーは、事前に定義されたクラスのオブジェクトでなければなりません。つまり、クラス A に A のオブジェクトを含めることはできませんが、クラス A のオブジェクトを指すポインターや参照を含めることはできます。
- 非静的配列メンバーの寸法は、すべて指定する必要があります。

定数初期化指定子 (= *constant_expression*) は、static と宣言された整数または列挙型のクラス・メンバー内にのみ現れます。

純粋指定子 (= 0) は、関数に定義がないことを示します。これは、virtual として宣言されたメンバー関数のみと一緒に使用され、メンバー・リスト内のメンバー関数の関数定義を置き換えます。

アクセス指定子 は、public、private、または protected のいずれかです。

メンバー宣言 は、宣言が入っているクラスのクラス・メンバーを宣言します。

access_specifier が分離する非静的クラス・メンバーの割り振り順は、インプリメンテーションに依存します。

access_specifier が分離する非静的クラス・メンバーの割り振り順は、インプリメンテーションに依存します。コンパイラーでは、クラス・メンバーが宣言された順序に従ってそれらを割り振ります。

A がクラスの名前だとします。以下のような A のクラス・メンバーは、A と異なる名前にする必要があります。
- すべてのデータ・メンバー
- すべての型のメンバー
- すべての列挙型メンバーの列挙子
- すべての無名共用体メンバーのメンバーすべて

関連情報

- 220 ページの『クラス型の宣言』
- 243 ページの『メンバー・アクセス』
- 256 ページの『継承されたメンバー・アクセス』
- 239 ページの『静的メンバー』

データ・メンバー

データ・メンバーには、基本型だけでなく、ポインター、参照、配列の型、ビット・フィールド、およびユーザー定義の型などの、他の型を用いて宣言されるメンバーが含まれます。データ・メンバーは、変数と同じ方法で宣言できますが、明示的初期化指定子をクラス定義の内部に設定することはできません。しかし、整数型または列挙型の const 静的データ・メンバーは、明示的初期化指定子を持つこともあります。

配列を非静的クラス・メンバーとして宣言する場合には、その配列のすべての次元を指定する必要があります。

クラスには、クラス型のメンバー、またはクラス型へのポインターまたは参照であるメンバーを入れることができます。クラス型のメンバーは、事前に宣言されているクラス型のメンバーでなければなりません。クラスのサイズが必要でなければ、メンバー宣言で、不完全なクラス型を使用することができます。例えば、不完全なクラス型へのポインターであるメンバーを宣言することができます。

クラス X には、型 X のメンバーを入れることはできません。ただし、X へのポインター、X への参照、および X の静的オブジェクトは入れることができます。 X のメンバー関数は、型 X の引数を取り、X 型を戻します。次に例を示します。

```
class X
{
      X();
      X *xptr;
      X &xref;
      static X xcount;
      X xfunc(X);
};
```

関連情報

- 243 ページの『メンバー・アクセス』
- 256 ページの『継承されたメンバー・アクセス』
- 239 ページの『静的メンバー』

メンバー関数

メンバー関数 とは、クラスのメンバーとして宣言される演算子および関数のことです。メンバー関数には、friend 指定子を用いて宣言される演算子および関数は含まれません。これらは、クラスの*フレンド* と呼ばれます。メンバー関数を static として宣言できます。これは、*静的メンバー関数* と呼ばれます。static として宣言されていないメンバー関数は、*非静的メンバー関数* と呼ばれます。

メンバー関数の定義は、それを囲むクラスのスコープ内にあります。メンバー関数の定義が、クラス・メンバー・リストのそのメンバーの宣言の前にある場合であっても、メンバー関数の本体は、クラス宣言の後で分析され、そのクラスのメンバーを、メンバー関数の本体で使用できるようにします。以下の例では、関数 add() が呼び出されるとき、データ変数の a、b、および c を add() の本体で使用することができます。

```
class x
{
public:
      int add()             // inline member function add
      {return a+b+c;};
private:
      int a,b,c;
};
```

インライン・メンバー関数

クラス定義の内部にメンバー関数を定義するか、またはクラス定義にメンバー関数をすでに宣言している (しかし、定義はされていない) 場合は、メンバー関数をクラス定義の外側で定義できます。

そのクラス・メンバー・リスト内で定義されるメンバー関数は、*インライン・メンバー関数* と呼ばれます。コードを数行含んでいるメンバー関数は、通常インラインで宣言されます。上記の例では、add() がインライン・メンバー関数です。クラス定義の外側にメンバー関数を定義する場合、メンバー関数は、クラス定義を囲むネーム・スペース・スコープ内に入れる必要があります。スコープ・レゾリューション (::) 演算子を使用して、メンバー関数名を修飾することもできます。

インライン・メンバー関数を宣言するのと同等な方法は、inline キーワードを指定してクラス内でその関数を宣言するか (そしてクラスの外側で関数を定義する)、または inline キーワードを使用して、クラス宣言の外側でその関数を定義することです。

次の例では、メンバー関数 Y::f() は、インライン・メンバー関数です。

```
struct Y {
private:
  char* a;
public:
  char* f() { return a; }
};
```

次の例は、直前の例と同等なものです。`Y::f()` が、インライン・メンバー関数です。

```
struct Y {
private:
  char* a;
public:
  char* f();
};

inline char* Y::f() { return a; }
```

リンケージはデフォルトでは外部結合なので、`inline` 指定子はメンバー関数または非メンバー関数のリンケージには影響を与えません。

ローカル・クラスのメンバー関数は、そのクラス定義の中で定義する必要があります。結果として、ローカル・クラスのメンバー関数は、暗黙的にインライン関数です。これらのインライン・メンバー関数には、リンケージはありません。

関連情報

- 245 ページの『フレンド』
- 242 ページの『静的メンバー関数』
- 176 ページの『inline 関数指定子』
- 226 ページの『ローカル・クラス』

定数および volatile メンバー関数

`const` 修飾子を指定して宣言されたメンバー関数は、定数オブジェクトおよび非定数オブジェクトに対して呼び出すことができます。非定数メンバー関数は、非定数オブジェクトに対してのみ呼び出すことができます。同様に、`volatile` 修飾子を指定して宣言されたメンバー関数は、volatile オブジェクトおよび非 volatile オブジェクトに対して呼び出すことができます。非 volatile メンバー関数は、非 volatile オブジェクトに対してのみ呼び出すことができます。

関連情報

- 62 ページの『型修飾子』
- 236 ページの『this ポインター』

仮想メンバー関数

仮想メンバー関数は、キーワード `virtual` によって宣言されます。この関数を使用すると、メンバー関数の動的バインディングが可能となります。仮想関数は、すべてメンバー関数でなければならないため、仮想メンバー関数は、単に*仮想関数* と呼ばれます。

関数の宣言内の純粋指定子によって仮想関数の定義が置き換えられた場合、その関数は純粋を宣言されたと言います。純粋仮想関数を 1 つでも持つクラスは、*抽象クラス* と呼ばれます。

関連情報

- 269 ページの『仮想関数』

- 274 ページの『抽象クラス』

特殊メンバー関数

特殊なメンバー関数 は、クラス・オブジェクトの作成、破棄、初期化、変換、およびコピーを行う場合に使用されます。そのような関数には、以下のものが含まれます。

- コンストラクター
- デストラクター
- 変換コンストラクター
- 変換関数
- コピー・コンストラクター

これらの関数の詳細については、 277 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』を参照してください。

メンバー・スコープ

クラス・メンバー・リスト内ですでに宣言してあるが、定義はしていないメンバー関数と静的メンバーは、それらのクラス宣言の外側で定義することができます。非静的データ・メンバーは、それらのクラスのオブジェクトが作成されたときに定義されます。静的データ・メンバーの宣言は、定義ではありません。メンバー関数の宣言は、関数の本体も指定されている場合には、定義になります。

クラス・メンバーの定義がクラス宣言の外側にある場合、メンバー名は、**::** (スコープ・レゾリューション) 演算子を使用して、クラス名によって修飾する必要があります。

以下の例では、クラス宣言の外側でメンバー関数を定義しています。

```
#include <iostream>
using namespace std;

struct X {
  int a, b ;

  // member function declaration only
  int add();
};

// global variable
int a  = 10;

// define member function outside its class declaration
int X::add() { return a + b; }

int main() {
  int answer;
  X xobject;
  xobject.a = 1;
  xobject.b = 2;
  answer = xobject.add();
  cout << xobject.a << " + " << xobject.b << " = " << answer << endl;
}
```

この例の出力は `1 + 2 = 3` となります。

すべてのメンバー関数は、それらがクラス宣言の外側で定義されている場合であっても、クラス・スコープ内にあります。上記の例では、メンバー関数 `add()` はデータ・メンバー `a` を戻しますが、グローバル変数 `a` は戻しません。

クラス・メンバーの名前は、そのクラスに対してローカルなものです。 . (ドット)、-> (矢印)、または :: (スコープ・レゾリューション) 演算子の、いずれのクラス・アクセス演算子も使用しない場合、クラス・メンバーは、クラスのメンバー関数内およびネスト・クラス内でのみ使用できます。ネスト・クラス内で、:: 演算子で修飾されていないものは、型、列挙、および静的メンバーしか使用できません。

メンバー関数本体で名前を検索する順序は、次のとおりです。
1. メンバー関数本体自体の中
2. すべての囲みクラスの中 (それらのクラスの継承メンバーを含めて)
3. 本体宣言の字句範囲の中

継承メンバーを含めた、囲みクラスの検索の例を、以下に示します。

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class Z : A {
      class Y : B {
            class X : C { int f(); /* ... */ };
      };
};
int Z::Y::X f()
{
      char j;
      return 0;
}
```

この例で、関数 f の定義内での名前 j の検索は、以下の順序に従います。
1. 関数 f の本体の中
2. X およびその基底クラス C の中
3. Y およびその基底クラス B の中
4. Z およびその基底クラス A の中
5. f の本体の字句範囲の中。この場合はグローバル・スコープ

収容クラスが検索されるときは、収容クラスの定義とそれらの基底クラスだけが検索されるということに留意してください。基底クラスの定義を含むスコープ (この例ではグローバル・スコープ) は、検索しません。

関連情報
- 5 ページの『クラス・スコープ (C++ のみ)』

メンバーへのポインター

メンバーへのポインターを使用すると、クラス・オブジェクトの非静的メンバーを参照することができます。メンバーへのポインターを使用して静的クラス・メンバーを指すことはできません。静的メンバーのアドレスは、特定のオブジェクトに関連付けられていないからです。静的クラス・メンバーを指すためには、標準のポインターを使用する必要があります。

メンバー関数へのポインターは、関数へのポインターと同じ方法で使用することができます。メンバー関数へのポインターを比較し、値を割り当て、さらにそれらを使用してメンバー関数を呼び出すことができます。メンバー関数の型は、番号、引数の型、および戻りの型が同じ非メンバー関数と同じではないことに注意してください。

メンバーへのポインターは、以下の例に示すように宣言し、使用することができます。

```
#include <iostream>
using namespace std;

class X {
public:
  int a;
  void f(int b) {
    cout << "The value of b is "<< b << endl;
  }
};

int main() {

  // declare pointer to data member
  int X::*ptiptr = &X::a;

  // declare a pointer to member function
  void (X::* ptfptr) (int) = &X::f;

  // create an object of class type X
  X xobject;

  // initialize data member
  xobject.*ptiptr = 10;

  cout << "The value of a is " << xobject.*ptiptr << endl;

  // call member function
  (xobject.*ptfptr) (20);
}
```

この例の出力は次のようになります。

```
The value of a is 10
The value of b is 20
```

複雑な構文を簡単にするために、`typedef` がメンバーへのポインターであると宣言することができます。メンバーへのポインターは、以下のコード・フラグメントに示すように宣言し、使用することができます。

```
typedef int X::*my_pointer_to_member;
typedef void (X::*my_pointer_to_function) (int);

int main() {
  my_pointer_to_member ptiptr = &X::a;
  my_pointer_to_function ptfptr = &X::f;
  X xobject;
  xobject.*ptiptr = 10;
  cout << "The value of a is " << xobject.*ptiptr << endl;
  (xobject.*ptfptr) (20);
}
```

メンバーへのポインター演算子 `.*` および `->*` は、特定のクラス・オブジェクトのメンバーへのポインターをバインドする際に用いられます。 `()` (関数呼び出し演算子) の優先順位の方が `.*` および `->*` よりも高いため、`ptf` によって指示される関数を呼び出す際は小括弧を使用することが必要です。

メンバーを指すポインターの変換は、メンバーを指すポインターの変換が初期化、代入、または比較されるときに行われます。メンバーへのポインターは、オブジェクトへのポインターまたは関数へのポインターと同じではないことに注意してください。

関連情報

- 130 ページの『メンバーを指すポインター演算子 .* ->* (C++ のみ)』

this ポインター

キーワード this は、特定の型のポインターを識別します。クラス A の x という名前のオブジェクトを作成し、クラス A には、非静的メンバー関数 f() があるとします。関数 x.f() を呼び出す場合、f() の本体にあるキーワード this は、x のアドレスを格納します。 this ポインターを宣言したり、またはそれに割り当てたりすることはできません。

静的メンバー関数は、this ポインターを持ちません。

クラス型 X のメンバー関数に対する this ポインターの型は、X* const です。メンバー関数が **const** 修飾子を用いて宣言されている場合、クラス X のそのメンバー関数に対する this ポインターの型は、const X* const です。

const this ポインターは、const メンバー関数を使うときのみに使用できます。クラスのデータ・メンバーは、その関数内で定数になります。その場合でも、関数はその値を変更することができますが、そのためには、次のように const_cast が必要です。

```
void foo::p() const{
member = 1;                       // illegal
const_cast <int&> (member) = 1;   // a bad practice but legal
}
```

もっといい方法は、member mutable を宣言することです。

メンバー関数が **volatile** 修飾子を用いて宣言されている場合、クラス X のそのメンバー関数に対する this ポインターの型は、volatile X* const です。例えば、コンパイラーは次のコードを許可しません。

```
struct A {
  int a;
  int f() const { return a++; }
};
```

コンパイラーは、関数 f() の本体で、ステートメント a++ を許可しません。関数 f() では、this ポインターは、A* const 型です。関数 f() は、this が指すオブジェクトの一部を変更しようとしています。

this ポインターは、すべての非静的メンバー関数呼び出しに隠れた引数として渡され、すべての非静的関数本体の中のローカル変数として使用することができます。

例えば、メンバー関数本体内で this ポインターを使用することによって、メンバー関数が呼び出される特定のクラス・オブジェクトを参照することができます。以下の例で示すコードによって作成される出力は、a = 5 です。

```
#include <iostream>
using namespace std;

struct X {
private:
  int a;
public:
  void Set_a(int a) {

    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
```

```
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

メンバー関数 Set_a() では、ステートメント this->a = a は、this ポインターを使用して、自動変数 a によって隠された xobj.a を検索します。

クラス・メンバー名が隠蔽されていない限り、クラス・メンバー名の使用は、this ポインターとクラス・メンバー・アクセス演算子 (->) を用いたクラス・メンバー名の使用と同じです。

以下のテーブルの最初の列は、this ポインターを指定しないで、クラス・メンバーを使用するコードの例を示しています。 2 番目の列にあるコードは、変数 THIS を使用して、最初の列で、その使用が隠蔽されている this ポインターをシミュレートします。

<table>
<tr><th>this ポインターを使用しないコード</th><th>等価のコード。隠蔽されている this ポインターをシミュレートする THIS 変数を使用。</th></tr>
<tr><td>

```
#include <string>
#include <iostream>
using namespace std;

struct X {
private:
  int len;
  char *ptr;
public:
  int GetLen() {
    return len;
  }
  char * GetPtr() {
    return ptr;
  }
  X& Set(char *);
  X& Cat(char *);
  X& Copy(X&);
  void Print();
};

X& X::Set(char *pc) {
  len = strlen(pc);
  ptr = new char[len];
  strcpy(ptr, pc);
  return *this;
}

X& X::Cat(char *pc) {
  len += strlen(pc);
  strcat(ptr,pc);
  return *this;
}

X& X::Copy(X& x) {
  Set(x.GetPtr());
  return *this;
}

void X::Print() {
  cout << ptr << endl;
}

int main() {
  X xobj1;
  xobj1.Set("abcd")
       .Cat("efgh");

  xobj1.Print();
  X xobj2;
  xobj2.Copy(xobj1)
       .Cat("ijkl");

  xobj2.Print();
}
```

</td><td>

```
#include <string>
#include <iostream>
using namespace std;

struct X {
private:
  int len;
  char *ptr;
public:
  int GetLen (X* const THIS) {
    return THIS->len;
  }
  char * GetPtr (X* const THIS) {
    return THIS->ptr;
  }
  X& Set(X* const, char *);
  X& Cat(X* const, char *);
  X& Copy(X* const, X&);
  void Print(X* const);
};

X& X::Set(X* const THIS, char *pc) {
  THIS->len = strlen(pc);
  THIS->ptr =  new char[THIS->len];
  strcpy(THIS->ptr, pc);
  return *THIS;
}

X& X::Cat(X* const THIS, char *pc) {
  THIS->len += strlen(pc);
  strcat(THIS->ptr, pc);
  return *THIS;
}

X& X::Copy(X* const THIS, X& x) {
  THIS->Set(THIS, x.GetPtr(&x));
  return *THIS;
}

void X::Print(X* const THIS) {
  cout << THIS->ptr << endl;
}

int main() {
  X xobj1;
  xobj1.Set(&xobj1 , "abcd")
       .Cat(&xobj1 , "efgh");

  xobj1.Print(&xobj1);
  X xobj2;
  xobj2.Copy(&xobj2 , xobj1)
       .Cat(&xobj2 , "ijkl");

  xobj2.Print(&xobj2);
}
```

</td></tr>
</table>

両方の例は、以下の出力を作成します。

```
abcdefgh
abcdefghijkl
```

関連情報

- 211 ページの『代入の多重定義』
- 293 ページの『コピー・コンストラクター』

静的メンバー

クラス・メンバーは、クラス・メンバー・リストで、ストレージ・クラス指定子 static を使用して宣言することができます。プログラム中の 1 つのクラスのすべてのオブジェクトが、静的メンバーの 1 つのコピーのみを共用します。静的メンバーを持つクラスのオブジェクトを宣言すると、その静的メンバーはそのクラス・オブジェクトの一部にはなりません。

静的メンバーの一般的な使用法は、クラスの全オブジェクトに共通なデータを記録する際に使用することです。例えば、静的データ・メンバーをカウンターとして使用して、作成された特定のクラス型のオブジェクト数を保管することができます。新しいオブジェクトが作成されるたびに、この静的データ・メンバーを増やして、オブジェクトの総数を記録することができます。

:: (スコープ・レゾリューション) 演算子を使用してクラス名を修飾して、静的メンバーにアクセスすることができます。次の例では、型 X のオブジェクトが宣言されていなくても、クラス型 X の静的メンバー f() を、X::f(): として参照することができます。

```
struct X {
  static int f();
};

int main() {
  X::f();
}
```

関連情報

- 232 ページの『定数および volatile メンバー関数』
- 40 ページの『static ストレージ・クラス指定子』
- 229 ページの『クラス・メンバー・リスト』

静的メンバーでのクラス・アクセス演算子の使用

静的メンバーを参照するのに、クラス・メンバー・アクセス構文を使用する必要はありません。つまり、クラス X の静的メンバー s にアクセスするために、式 X::s が使用できます。以下の例は、静的メンバーへのアクセスを説明しています。

```
#include <iostream>
using namespace std;

struct A {
  static void f() { cout << "In static function A::f()" << endl; }
};

int main() {

  // no object required for static member
  A::f();

  A a;
  A* ap = &a;
  a.f();
  ap->f();
}
```

ステートメント A::f()、a.f()、および ap->f() の 3 つはすべて、同じ静的メンバー関数 A::f() を呼び出します。

そのクラスの同じスコープ内、または静的メンバーのクラスから派生したクラスのスコープ内にある、静的メンバーを直接参照することができます。次の例は、後者のケースを説明しています (静的メンバーのクラスから派生したクラスのスコープ内にある静的メンバーを直接参照する)。

```
#include <iostream>
using namespace std;

int g() {
   cout << "In function g()" << endl;
   return 0;
}

class X {
   public:
      static int g() {
         cout << "In static member function X::g()" << endl;
         return 1;
      }
};

class Y: public X {
   public:
      static int i;
};

int Y::i = g();

int main() { }
```

次に、上記のコード出力を示します。

```
In static member function X::g()
```

初期設定 int Y::i = g() は、X::g() を呼び出しますが、グローバル・ネーム・スペースに宣言されている関数、g() は呼び出しません。

関連情報

- 40 ページの『static ストレージ・クラス指定子』
- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』
- 109 ページの『ドット演算子 .』
- 110 ページの『矢印演算子 ->』

静的データ・メンバー

クラスのメンバー・リストにおける静的データ・メンバーの宣言は、定義ではありません。静的メンバーは、ネーム・スペース・スコープのクラス宣言の外側で定義することが必要です。次に例を示します。

```
class X
{
public:
      static int i;
};
int X::i = 0; // definition outside class declaration
```

静的データ・メンバーを一度定義してしまえば、そのメンバーは、静的データ・メンバー・クラスのオブジェクトがなくても存在します。上記の例では、静的データ・メンバー X::i が定義されていても、クラス X のオブジェクトは、存在しません。

ネーム・スペース・スコープのクラスの静的データ・メンバーには、外部リンケージがあります。 静的データ・メンバー用の初期化指定子は、メンバーを宣言するクラスのスコープ内にあります。

静的データ・メンバーは、void、あるいは const または volatile で修飾された void を除く、あらゆる型に指定できます。静的データ・メンバーを mutable として宣言できません。

プログラム内には、静的メンバーの定義は 1 つしか入れられません。名前のないクラス、名前のないクラスに含まれるクラス、およびローカル・クラスは、静的データ・メンバーを持つことができません。

静的データ・メンバーとその初期化指定子は、そのクラスの、他の静的の private メンバーおよび保護メンバーにアクセスすることができます。以下の例は、他の静的メンバー (そのメンバーが private であっても) を使用して、どのように静的メンバーを初期化できるかを示しています。

```
class C {
      static int i;
      static int j;
      static int k;
      static int l;
      static int m;
      static int n;
      static int p;
      static int q;
      static int r;
      static int s;
      static int f() { return 0; }
      int a;
public:
      C() { a = 0; }
      };

C c;
int C::i = C::f();    // initialize with static member function
int C::j = C::i;      // initialize with another static data member
int C::k = c.f();     // initialize with member function from an object
int C::l = c.j;       // initialize with data member from an object
int C::s = c.a;       // initialize with nonstatic data member
int C::r = 1;         // initialize with a constant value

class Y : private C {} y;

int C::m = Y::f();
int C::n = Y::r;
int C::p = y.r;       // error
int C::q = y.f();     // error
```

y は、C から private に派生したクラスのオブジェクトであるため、C::p および C::q の初期化は、エラーとなり、このオブジェクトのメンバーは、C のメンバーからはアクセスできません。

静的データ・メンバーが const 整数型、または const 列挙型のメンバーである場合、*定数初期化指定子*を静的データ・メンバーの宣言で指定できます。この定数初期化指定子は、整数定数式でなければなりません。定数初期化指定子は、定義ではないことに注意してください。まだ、囲みネーム・スペースに静的メンバーを定義する必要があります。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct X {
  static const int a = 76;
};

const int X::a;
```

```
int main() {
  cout << X::a << endl;
}
```

静的データ・メンバー a の最後に宣言されているトークン = 76 が、定数初期化指定子です。

関連情報

- 8 ページの『外部結合』
- 243 ページの『メンバー・アクセス』
- 226 ページの『ローカル・クラス』

静的メンバー関数

同じ名前、および引数の数と型が同じである、静的メンバー関数と非静的メンバー関数を持つことはできません。

静的データ・メンバーのように、クラス A のオブジェクトを使用しないで、クラス A の静的メンバー関数 f() にアクセスできます。

静的メンバー関数は、this ポインターを持ちません。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct X {
private:
  int i;
  static int si;
public:
  void set_i(int arg) { i = arg; }
  static void set_si(int arg) { si = arg; }

  void print_i() {
    cout << "Value of i = " << i << endl;
    cout << "Again, value of i = " << this->i << endl;
  }

  static void print_si() {
    cout << "Value of si = " << si << endl;
//    cout << "Again, value of si = " << this->si << endl;
  }

};

int X::si = 77;       // Initialize static data member

int main() {
  X xobj;
  xobj.set_i(11);
  xobj.print_i();

  // static data members and functions belong to the class and
  // can be accessed without using an object of class X
  X::print_si();
  X::set_si(22);
  X::print_si();
}
```

上記の例の出力は、以下のとおりです。

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

コンパイラーは、このメンバー関数が静的として宣言されていて、そのためメンバー関数が this ポインターを持っていないので、関数 A::print_si() で、メンバー・アクセス操作 this->si を認めません。

非静的メンバー関数の this ポインターを使用して、静的メンバー関数を呼び出すことができます。以下の例では、非静的メンバー関数の printall() が、this ポインターを使用して静的メンバー関数の f() を呼び出します。

```
#include <iostream>
using namespace std;

class C {
  static void f() {
    cout << "Here is i: " << i << endl;
  }
  static int i;
  int j;
public:
  C(int firstj): j(firstj) { }
  void printall();
};

void C::printall() {
  cout << "Here is j: " << this->j << endl;
  this->f();
}

int C::i = 3;

int main() {
  C obj_C(0);
  obj_C.printall();
}
```

上記の例の出力は、以下のとおりです。

```
Here is j: 0
Here is i: 3
```

キーワード virtual、const、volatile、または const volatile を使用した静的メンバー関数の宣言はできません。

静的メンバー関数がアクセスできるのは、その関数が宣言されているクラスの静的メンバー、列挙子、およびネストされた型の名前だけです。静的メンバー関数 f() が、クラス X のメンバーであるとします。静的メンバー関数 f() は、非静的メンバー X または基底クラス X の非静的メンバーにアクセスできません。

関連情報

- 236 ページの『this ポインター』

メンバー・アクセス

メンバー・アクセスは、式または宣言内で、クラス・メンバーがアクセス可能かどうかを判別します。 x がクラス A のメンバーであるとします。このクラス・メンバー x は、以下のいずれかのレベルのアクセス可能性を持つものとして宣言することができます。

- public: x は、private や protected で定義したアクセス制限以外の場所なら、どこででも使用できます。

- private: x は、クラス A のメンバーとフレンドだけが使用できます。
- protected: x は、クラス A のメンバーとフレンド、およびクラス A から派生したクラスのメンバーとフレンドだけが使用できます。

キーワード class を指定して宣言されたクラスのメンバーのデフォルトは、private です。キーワード struct または union を指定して宣言されたクラスのメンバーのデフォルトは、public です。

クラス・メンバーのアクセスを制御するには、アクセス*指定子* public、private、または protected をクラス・メンバー・リストのラベルとして使用します。次の例はこれらのアクセス指定子を示しています。

```
struct A {
  friend class C;
private:
  int a;
public:
  int b;
protected:
  int c;
};

struct B : A {
void f() {
    // a = 1;
    b = 2;
    c = 3;
  }
};

struct C {
  void f(A x) {
    x.a = 4;
    x.b = 5;
    x.c = 6;
  }
};

int main() {
  A y;
//  y.a = 7;
  y.b = 8;
//  y.c = 9;

  B z;
//  z.a = 10;
  z.b = 11;
//  z.c = 12;
}
```

次の表は、上記の例のようなさまざまなスコープ内のデータ・メンバー A::a、A::b、および A::c へのアクセスについて示しています。

スコープ	A::a	A::b	A::c
関数 B::f()	アクセス不可。メンバー A::a は、private です。	アクセス可能。メンバー A::b は、public です。	アクセス可能。クラス B は、A から継承します。
関数 C::f()	アクセス可能。クラス C は、A のフレンドです。	アクセス可能。メンバー A::b は、public です。	アクセス可能。クラス C は、A のフレンドです。
main() 内のオブジェクト y	アクセス不可。メンバー y.a は、private です。	アクセス可能。メンバー y.a は、public です。	アクセス不可。メンバー y.c は、protected です。

スコープ	A::a	A::b	A::c
main() のオブジェクト z	アクセス不可。 メンバー z.a は、private です。	アクセス可能。メンバー z.a は、public です。	アクセス不可。メンバー z.c は、protected です。

アクセス指定子は、次のアクセス指定子まで、またはクラス定義の終わりまでその後に続くメンバーのアクセス可能性を指定します。任意の数のアクセス指定子を、任意の順序で使用することができます。クラス定義内に後からクラス・メンバーを定義する場合、そのアクセス指定は、その宣言と同一にする必要があります。次の例は、このことを示しています。

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

コンパイラーは、クラス B がすでに private として宣言されているため、このクラスの定義を許可しません。

クラス・メンバーには、それがそのクラス内、またはクラスの外側に定義されたかどうかには関係なく、同じアクセス制御があります。

アクセス制御は、名前に対応しています。特に、アクセス制御を typedef 名に追加する場合、それは typedef 名だけに影響します。次の例は、このことを示しています。

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

int main() {
  A::C x;
//  A::B y;
}
```

コンパイラーは、typedef 名 A::C が public なので、宣言 A::C x を許可します。コンパイラーは、A::B は、private なので、宣言 A::B y を認めません。

アクセス可能性と可視性は、別々のものであることに注意してください。可視性は、C++ のスコープ規則に基づきます。クラス・メンバーが、可視であり、同時にアクセス不能ということはあり得ます。

関連情報

- 1 ページの『スコープ』
- 229 ページの『クラス・メンバー・リスト』
- 256 ページの『継承されたメンバー・アクセス』

フレンド

クラス X のフレンドとは、X のメンバーではないが、X のメンバーと同じ X へのアクセスを認可されている関数またはクラスです。クラス・メンバー・リスト内で、friend 指定子を使用して宣言された関数は、そのクラスの*フレンド関数* と呼ばれます。別のクラスのメンバー・リスト内で friend 指定子を用いて宣言されたクラスは、そのクラスの*フレンド・クラス* と呼ばれます。

クラス Y を定義してからでなければ、Y の任意のメンバーを、別のクラスのフレンドとして宣言することはできません。

以下の例では、フレンド関数 print は、クラス Y のメンバーであり、クラス X の private データ・メンバー a および b にアクセスします。

```
#include <iostream>
using namespace std;

class X;

class Y {
public:
  void print(X& x);
};

class X {
  int a, b;
  friend void Y::print(X& x);
public:
  X() : a(1), b(2) { }
};

void Y::print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
}

int main() {
  X xobj;
  Y yobj;
  yobj.print(xobj);
}
```

上記の例の出力は、以下のとおりです。

```
a is 1
b is 2
```

クラス全体をフレンドとして宣言することができます。クラス F が、クラス A のフレンドだとします。メンバー関数、およびクラス F の静的データ・メンバー定義はいずれも、クラス A へのアクセスを行えます。

次の例では、フレンド・クラス F には、クラス X の private データ・メンバー a および b にアクセスする、メンバー関数 print があります。これは、前述の例のフレンド関数 print と同じタスクを実行します。また、クラス F に宣言されたその他のメンバーも、クラス X のメンバーすべてにアクセスできます。

```
#include <iostream>
using namespace std;

class X {
  int a, b;
  friend class F;
public:
  X() : a(1), b(2) { }
};

class F {
public:
  void print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
  }
};
```

```
int main() {
  X xobj;
  F fobj;
  fobj.print(xobj);
}
```

上記の例の出力は、以下のとおりです。

```
a is 1
b is 2
```

クラスをフレンドとして宣言する場合は、詳述型指定子を使用する必要があります。次の例は、このことを示しています。

```
class F;
class G;
class X {
  friend class F;
  friend G;
};
```

コンパイラーは、G のフレンド宣言が、詳述クラス名でなければならないことを警告します。

フレンド宣言では、クラスを定義できません。例えば、コンパイラーは次のコードを許可しません。

```
class F;
class X {
  friend class F { };
};
```

しかし、フレンド宣言で関数を定義できます。クラスは、非ローカル・クラスの関数で、関数名が非修飾であり、ネーム・スペース・スコープを持っている必要があります。次の例は、このことを示しています。

```
class A {
  void g();
};

void z() {
  class B {
//    friend void f() { };
  };
}

class C {
//  friend void A::g() { }
  friend void h() { }
};
```

コンパイラーは、f() または g() の関数定義を許可しません。コンパイラーは、h() の定義は認めます。

ストレージ・クラス指定子を使用して、フレンドを宣言することはできません。

関連情報

- 243 ページの『メンバー・アクセス』
- 256 ページの『継承されたメンバー・アクセス』

フレンド・スコープ

フレンド宣言で最初に導入されるフレンド関数またはフレンド・クラスの名前は、フレンド関係を認可するクラス (*囲みクラスとも呼ばれます*) のスコープ内にはなく、フレンド関係を認可するクラスのメンバーでもありません。

フレンド宣言で最初に導入された関数の名前は、囲みクラスが含まれている最初の非クラス・スコープのスコープ内にあります。フレンド宣言で与えられる関数の本体は、クラス内で定義されるメンバー関数と同じ方法で処理されます。定義の処理は、最外部の囲みクラスの終わりまで開始されません。さらに、関数定義の本体内の修飾されない名前による検索は、その関数定義が入っているクラスから始められます。

フレンド・クラスの名前が、フレンド宣言よりも前に導入されている場合、コンパイラーは、そのフレンド・クラスの名前と一致するクラス名の検索を、フレンド宣言のスコープの先頭から開始します。ネスト・クラスの宣言の後に同じ名前のフレンド・クラスの宣言が続いている場合、そのネスト・クラスは、囲みクラスのフレンドです。

フレンド・クラス名のスコープは、最初の非クラスの囲みスコープです。次に例を示します。

```
class A {
   class B { // arbitrary nested class definitions
      friend class C;
   };
};
```

は、以下と同等です。

```
class C;
class A {
   class B { // arbitrary nested class definitions
      friend class C;
   };
};
```

フレンド関数が別のクラスのメンバーである場合には、スコープ・レゾリューション演算子 (::) を使用することが必要です。次に例を示します。

```
class A {
public:
  int f() { }
};

class B {
  friend int A::f();
};
```

基底クラスのフレンドは、その基底クラスから派生したクラスには、継承されません。次の例は、このことを示しています。

```
class A {
  friend class B;
  int a;
};

class B { };

class C : public B {
  void f(A* p) {
//    p->a = 2;
  }
};
```

C は A のフレンドから継承していますが、クラス C がクラス A のフレンドではないので、コンパイラーは、ステートメント p->a = 2 を許可しません。

フレンド関係は、移行できません。次の例は、このことを示しています。

```
class A {
  friend class B;
  int a;
```

```
};

class B {
  friend class C;
};

class C {
  void f(A* p) {
//    p->a = 2;
  }
};
```

C は A のフレンドのフレンドですが、クラス C がクラス A のフレンドではないので、コンパイラーは、ステートメント p->a = 2 を許可しません。

ローカル・クラスにフレンドを宣言し、フレンドの名前が非修飾である場合、コンパイラーは、最も内側にある囲みの非クラス・スコープ内でのみ名前を検索します。関数を宣言してから、ローカル・スコープのフレンドとして関数を宣言する必要があります。クラスでこれを実行する必要はありません。しかし、フレンド・クラスの宣言は、囲みスコープ内の、同じ名前のクラスを隠します。次の例は、このことを示しています。

```
class X { };
void a();

void f() {
  class Y { };
  void b();
  class A {
    friend class X;
    friend class Y;
    friend class Z;
//    friend void a();
    friend void b();
//    friend void c();
  };
  ::X moocow;
//  X moocow2;
}
```

上記の例では、コンパイラーは、次のステートメントを認めます。

- friend class X: このステートメントは、このクラスが別の方法で宣言されていなくても、::X を A のフレンドとしてではなく、ローカル・クラス X をフレンドとして宣言しています。
- friend class Y: ローカル・クラス Y は、 f() のスコープに宣言されました。
- friend class Z: このステートメントは、Z が別の方法で宣言されていなくても、ローカル・クラス Z を A のフレンドとして宣言しています。
- friend void b(): 関数 b() は、f() のスコープに宣言されました。
- ::X moocow: この宣言は、非ローカル・クラス ::X のオブジェクトを作成します。

コンパイラーは、次のステートメントを許可しません。

- friend void a(): このステートメントは、関数 a() を、ネーム・スペース・スコープに宣言されたと認めません。関数 a() が、f() に宣言されていないので、コンパイラーはこのステートメントを認めません。
- friend void c(): 関数 c() が、f() のスコープに宣言されていないので、コンパイラーはこのステートメントを認めません。

- `X moocow2`: この宣言は、非ローカル・クラス `::X` ではなく、ローカル・クラス `X` のオブジェクトを作成しようとします。ローカル・クラス `X` が定義されていないので、コンパイラーは、このステートメントを認めません。

関連情報

- 223 ページの『クラス名のスコープ』
- 224 ページの『ネスト・クラス』
- 226 ページの『ローカル・クラス』

フレンド・アクセス

クラスのフレンドは、そのクラスの private メンバーおよび保護メンバーにアクセスすることができます。通常、クラスの private メンバーには、そのクラスのメンバー関数を介してのみアクセスでき、クラスの保護メンバーにはクラスのメンバー関数、またはクラスから派生したクラスを介してのみ、アクセスすることができます。

フレンド宣言は、アクセス指定子には影響されません。

関連情報

- 243 ページの『メンバー・アクセス』

第 13 章 継承 (C++ のみ)

継承 とは、既存のクラスを変更しないで、再利用したり拡張したりするメカニズムのことです。

継承は、オブジェクトをクラスに組み込むのとほぼ同じです。クラス A のオブジェクト x を、B のクラス定義に宣言するとします。その結果、クラス B は、クラス A の public データ・メンバーおよびメンバー関数のすべてにアクセスできます。しかし、クラス B では、クラス A のデータ・メンバーおよびメンバー関数にアクセスするのに、オブジェクト x を介してアクセスする必要があります。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};

class B {
public:
   A x;
};

int main() {
   B obj;
   obj.x.f(20);
   cout << obj.x.g() << endl;
//   cout << obj.g() << endl;
}
```

main 関数のオブジェクト obj は、ステートメント obj.x.f(20) を使用したデータ・メンバー B::x を介して、関数 A::f() にアクセスします。オブジェクト obj は、同様な方法で、ステートメント obj.x.g() で A::g() にアクセスします。コンパイラーは、g() がクラス A のメンバー関数であり、クラス B のメンバー関数ではないので、ステートメント obj.g() を許可しません。

継承メカニズムにより、上記の例で示した obj.g() のようなステートメントを使用できます。ステートメントを有効にするには、 g() が、クラス B のメンバー関数である必要があります。

継承により、別のクラスのメンバーの名前および定義を、新規クラスの一部としてインクルードできます。新規クラスにインクルードしたいメンバーを持つ元のクラスは、*基底クラス* と呼ばれます。新規クラスは、基底クラスから*派生*します。新規クラスは、基底クラスの型の*サブオブジェクト* を含みます。次の例は、継承メカニズムを使用してクラス B にクラス A のメンバーへのアクセスを与える点を除いては、直前の例と同じです。

```
#include <iostream>
using namespace std;

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};

class B : public A { };
```

© Copyright IBM Corp. 1998, 2010

```
int main() {
   B obj;
   obj.f(20);
   cout << obj.g() << endl;
}
```

クラス A は、クラス B の基底クラスです。クラス A のメンバーの名前、および定義は、クラス B の定義にインクルードされます。つまり、クラス B は、クラス A のメンバーを継承します。クラス B は、クラス A から派生します。クラス　B には、型 A のサブオブジェクトが含まれます。

派生クラスに新たにデータ・メンバーやメンバー関数を追加することもできます。新たに派生させたクラスで基底クラスのメンバー関数やデータをオーバーライドすることによって、既存メンバー関数やデータのインプリメンテーションを変更することができます。

別の派生クラスからもクラスを派生できます。その結果、別のレベルの継承を作成します。次の例は、このことを示しています。

```
struct A { };
struct B : A { };
struct C : B { };
```

クラス B は、A の派生クラスでもあり、同時に、C の基底クラスでもあります。継承のレベル数は、リソースによってのみ限定されます。

多重継承 を使用すると、複数の基底クラスの属性を継承する派生クラスを作成することができます。派生クラスは、その全基底クラスからメンバーを継承するので、その結果あいまいさが生じる可能性があります。例えば、2 つの基底クラスに同じ名前のメンバーがある場合、派生クラスでは 2 つのメンバーを暗黙的に区別することができません。多重継承を使用するときは、基底クラスの名前へのアクセスがあいまいにならないように注意してください。詳しくは、 262 ページの『多重継承』を参照してください。

直接基底クラス とは、その派生クラスの宣言の中に、基底指定子として直接現れる基底クラスのことです。

間接基底クラス とは、派生クラスの宣言の中には直接出てこないが、その基底クラスの 1 つを介して派生クラスで使用できる基底クラスのことです。あるクラスについて、直接基底クラスでない基底クラスは、すべて間接基底クラスです。次の例は、直接基底クラスおよび間接基底クラスを示しています。

```
class A {
  public:
    int x;
};
class B : public A {
  public:
    int y;
};
class C : public B { };
```

クラス B は、C の直接基底クラスです。クラス A は、B の直接基底クラスです。クラス A は、C の間接基底クラスです。 (x および y は、クラス C のデータ・メンバーになります。)

ポリモアフィック関数 は、複数の型のオブジェクトに適用できる関数です。 C++ では、ポリモアフィック関数は、2 つの方法でインプリメントできます。

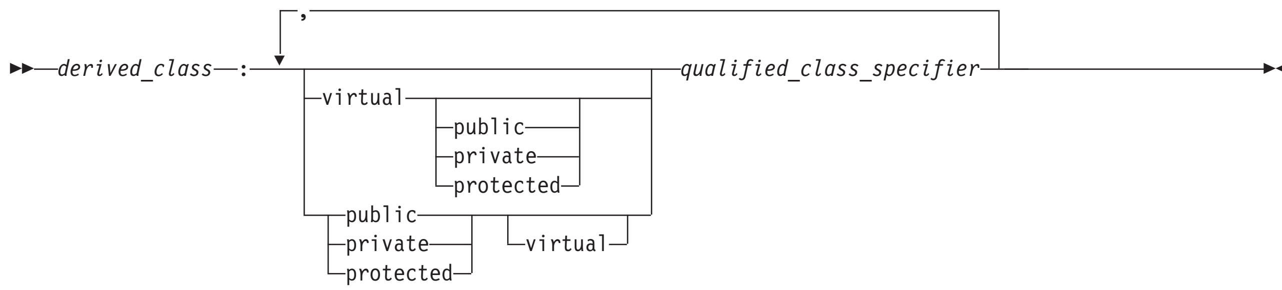
- 多重定義された関数は、コンパイル時に静的にバインドされます。

- C++ が、仮想関数を提供します。*仮想関数* は、派生を介して関連付けられている、いくつかの様々なユーザー定義の型について呼び出すことができる関数です。仮想関数は、実行時に動的にバインドされます。 269 ページの『仮想関数』で詳しく説明します。

派生

C++ では、継承は、派生のメカニズムによって実現されます。派生を使用すると、*派生クラス* と呼ばれるクラスを、*基底クラス* と呼ばれる別のクラスから派生させることができます。

派生クラスの構文



派生クラスの宣言の中で、派生クラスの基底クラスをリストします。派生クラスは、これらの基底クラスからそのメンバーを継承します。

qualified_class_specifier は、クラス宣言で事前に宣言されているクラスである必要があります。

アクセス指定子 は、public、private、または protected のいずれかです。

virtual キーワードは、仮想基底クラスの宣言に使用できます。

次の例は、派生クラス D と、基底クラス V、B1、および B2 の宣言を示しています。クラス B1 は、クラス V から派生し、D の基底クラスなので、基底クラスでもあり派生クラスでもあります。

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

宣言されているが定義されていないクラスは、基底リストに入れることができません。

次に例を示します。

```
class X;

// error
class Y: public X { };
```

コンパイラーは、X が定義されていないので、クラス Y の宣言を許可しません。

クラスを派生させると、派生クラスは、基底クラスのクラス・メンバーを継承します。継承されたメンバー (基底クラスのメンバー) は、派生クラスのメンバーと同様に参照することができます。次に例を示します。

```
class Base {
public:
  int a,b;
};
```

```
class Derived : public Base {
public:
  int c;
};

int main() {
  Derived d;
  d.a = 1;    // Base::a
  d.b = 2;    // Base::b
  d.c = 3;    // Derived::c
}
```

派生クラスには新しいクラス・メンバーを追加したり、既存の基底クラス・メンバーを再定義することもできます。上記の例では、派生クラスのメンバー c に加えて、派生クラス d の 2 つの継承されたメンバー a および b に値が代入されます。派生クラスの中で基底クラスのメンバーを再定義しても、:: (スコープ・レゾリューション) 演算子を使用すれば、依然として基底クラスのメンバーを参照することができます。次に例を示します。

```
#include <iostream>
using namespace std;

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  // call Derived::display()
  d.display();

  // call Base::display()
  d.Base::display();
}
```

上記の例の出力は、以下のとおりです。

```
Derived Class, Base Class
Base Class
```

派生クラスのオブジェクトは、基底クラスのオブジェクトと同様に操作できます。派生クラスのオブジェクトに対するポインターや参照を、その基底クラスに対するポインターや参照の代わりに使用することができます。例えば、派生クラスのオブジェクト D に対するポインターまたは参照を、D の基底クラスに対するポインターまたは参照を予期している関数に渡すことができます。これを実行するために明示的キャストを使用する必要はありません。標準型変換が実行されます。派生クラスに対するポインターを、明らかにアクセス可能な基底クラスを指すように、暗黙的に変換することができます。また派生クラスに対する参照を、基底クラスに対する参照に暗黙的に変換することもできます。

次の例では、派生クラスを指すポインターを基底クラスを指すポインターに変換する、標準型変換を示します。

```
#include <iostream>
using namespace std;

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  Derived* dptr = &d;

  // standard conversion from Derived* to Base*
  Base* bptr = dptr;

  // call Base::display()
  bptr->display();
}
```

上記の例の出力は、以下のとおりです。

```
Base Class
```

ステートメント `Base* bptr = dptr` は、`Derived` 型のポインターを `Base` 型のポインターに変換します。

この逆は認められていません。基底クラスのオブジェクトへのポインターや参照を、派生クラスへのポインターや参照に暗黙的に変換することはできません。例えば、クラス `Base` および `Class` が上記の例のように定義されている場合、コンパイラーは、次のコードを許可しません。

```
int main() {
  Base b;
  b.name = "Base class";

  Derived* dptr = &b;
}
```

コンパイラーは、ステートメントが暗黙的に `Base` 型のポインターを `Derived` 型のポインターに変換するので、ステートメント `Derived* dptr = &b` を許可しません。

派生クラスのメンバーと基底クラスのメンバーが同じ名前を持っている場合、基底クラス・メンバーは、派生クラスの中で隠されます。派生クラスのメンバーが基底クラスと同じ名前を持っている場合、基底クラス名は、派生クラスの中で隠されます。

関連情報

- 263 ページの『仮想基底クラス』

- 224 ページの『不完全なクラス宣言』
- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』

継承されたメンバー・アクセス

以下のセクションでは、保護非静的基底クラス・メンバーに影響を与えるアクセス規則と、アクセス指定子を使用して派生クラスを宣言する方法について説明します。

- 『protected メンバー』
- 257 ページの『基底クラス・メンバーのアクセス制御』

関連情報

- 243 ページの『メンバー・アクセス』

protected メンバー

基底クラスから派生したどのクラスのメンバーおよびフレンドも、次のいずれかの方法を使用して、protected 非静的基底クラス・メンバーにアクセスすることができます。

- 直接または間接の派生クラスへのポインター
- 直接または間接の派生クラスへの参照
- 直接または間接の派生クラスのオブジェクト

基底クラスから private にクラスを派生させた場合、基底クラスの全 protected メンバーは、派生クラスの private メンバーになります。

派生クラス B のフレンドまたはメンバー関数で、基底クラスの A の protected 非静的メンバー x を参照する場合、A から派生したクラスに対するポインター、参照、またはオブジェクトを介して x にアクセスする必要があります。しかし、x にアクセスし、メンバーに対するポインターを作成している場合は、派生クラス B の名前を付けるネスト名前指定子で x を修飾する必要があります。

```
class A {
public:
protected:
  int i;
};


class B : public A {
  friend void f(A*, B*);
  void g(A*);
};

void f(A* pa, B* pb) {
//  pa->i = 1;
  pb->i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}

void B::g(A* pa) {
//  pa->i = 1;
  i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}
```

```
void h(A* pa, B* pb) {
//  pa->i = 1;
//  pb->i = 2;
}

int main() { }
```

クラス A には、protected データ・メンバーである、整数 i が入っています。 B は A から派生するので、B のメンバーは、 A の protected メンバーへのアクセスが可能です。関数 f() は、クラス B のフレンドです。

- コンパイラーは、pa が派生クラス B に対するポインターでないので、pa->i = 1 を許可しません。
- コンパイラーは、i が派生クラス B の名前で修飾されていないので、int A::* point_i = &A::i を許可しません。

関数 g() は、クラス B のメンバー関数です。コンパイラーが許可するあるいは許可しないステートメントについての直前の注釈のリストは、次の点を除き、g() にも適用できます。

- コンパイラーは、i = 2 は、this->i = 2 と同等なので、認めます。

関数 h() は、A の派生クラスのフレンドでもメンバーでもないので、h() は、A のどの protected メンバーにもアクセスすることはできません。

基底クラス・メンバーのアクセス制御

派生クラスの宣言においては、派生クラスの基底リストの中の各基底クラスの前に、アクセス指定子を置くことができます。これによって、基底クラスから見たときの基底クラスの各メンバーのアクセス属性は、変更されませんが、派生クラスが、基底クラスのメンバーへのアクセス制御を制限できるようになります。

3 つのアクセス指定子のいずれかを使用して、クラスを派生させることができます。

- public 基底クラスでは、基底クラスの public および protected メンバーは、派生クラスにおいても public および protected メンバーです。
- protected 基底クラスにおいては、基底クラスの public および protected メンバーは、派生クラスの protected メンバーになります。
- private 基底クラスにおいては、基底クラスの public および protected メンバーは、派生クラスでは private メンバーになります。

すべての場合において、基底クラスの private メンバーは private のままです。基底クラスの private メンバーは、基底クラス内のフレンド宣言において、明示的にアクセスを認可されている場合でなければ、派生クラスから使用することはできません。

次の例では、クラス d は、クラス b から public に派生します。クラス b は、この宣言により、public 基底クラスに宣言されます。

```
class b { };
class d : public b // public derivation
{ };
```

構造体とクラスの両方を、派生クラス宣言の基底リストの中の基底クラスとして使用することができます。

- 派生クラスがキーワード class で宣言される場合、その基本リスト指定子にあるデフォルトのアクセス指定子は、private です。
- 派生クラスがキーワード struct で宣言される場合、その基本リスト指定子にあるデフォルトのアクセス指定子は、public です。

次の例では、基本リストで使用されるアクセス指定子がなく、派生クラスがキーワード class で宣言されているので、デフォルトで private の派生が使用されます。

```
struct B
{ };
class D : B // private derivation
{ };
```

クラスのメンバーおよびフレンドは、そのクラスのオブジェクトに対するポインターを暗黙的に次のいずれかに対するポインターに変換することができます。

- 直接 private 基底クラス
- protected 基底クラス (直接または間接)

関連情報

- 243 ページの『メンバー・アクセス』
- 233 ページの『メンバー・スコープ』

using 宣言とクラス・メンバー

クラス A の定義での using 宣言により、データ・メンバーまたはメンバー関数の*名前* を、A の基底クラスから A のスコープに導入できます。

規定および派生クラスからメンバー関数の多重定義セットを作成したい場合、またはクラス・メンバーのアクセスを変更したい場合は、クラス定義で using 宣言が必要です。

using 宣言の構文

```
►►──using──┬─┬──────────┬──┬────┬──nested_name_specifier──unqualified_id──;─┬──────────────────►◄
           │ └─typename─┘  └─::─┘                                            │
           └─::──unqualified_id──;───────────────────────────────────────────┘
```

クラス A の using 宣言は、次のいずれかに名前を付けます。

- A の基底クラスのメンバー
- A の基底クラスのメンバーである無名共用体のメンバー
- A の基底クラスのメンバーである列挙型の列挙子

次の例は、このことを示しています。

```
struct Z {
  int g();
};

struct A {
  void f();
  enum E { e };
  union { int u; };
};

struct B : A {
  using A::f;
  using A::e;
  using A::u;
//  using Z::g;
};
```

コンパイラーは、Z が A の基底クラスでないので、using 宣言 using Z::g を許可しません。

using 宣言は、テンプレートに名前を付けることはできません。例えば、コンパイラーは次のコードを許可しません。

```
struct A {
  template<class T> void f(T);
};

struct B : A {
  using A::f<int>;
};
```

using 宣言で示されている名前のインスタンスは、いずれもアクセス可能でなければなりません。次の例は、このことを示しています。

```
struct A {
private:
  void f(int);
public:
  int f();
protected:
  void g();
};

struct B : A {
//  using A::f;
  using A::g;
};
```

コンパイラーは、`int A::f()` はアクセス可能ですが、`void A::f(int)` が `B` からアクセス不可能なので、using 宣言 `using A::f` を許可しません。

関連情報

- 223 ページの『クラス名のスコープ』
- 203 ページの『using 宣言とネーム・スペース』

基底クラスおよび派生クラスからのメンバー関数の多重定義

クラス `A` で `f` と名付けられたメンバー関数は、戻りの型や引数に関係なく、`A` の基底クラスで、他の `f` という名前のメンバーをすべて隠します。次の例は、このことを示しています。

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

struct B : A {
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
//  obj_B.f();
}
```

コンパイラーは、`void B::f(int)` の宣言が `A::f()` を隠しているので、関数呼び出し `obj_B.f()` を許可しません。

基底クラス `A` の関数を、派生クラス `B` で、隠蔽ではなく多重定義するには、using 宣言を使用して、関数の名前を `B` のスコープに導入します。以下の例は、using 宣言 `using A::f` を除いては、直前の例と同じです。

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

struct B : A {
  using A::f;
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
  obj_B.f();
}
```

クラス B に using 宣言があるので、名前 f は 2 つの関数で多重定義されます。これで、コンパイラーは、関数呼び出し obj_B.f() を許可します。

同じ方法で仮想関数を多重定義できます。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
  B obj_B;
  B* pb = &obj_B;
  pb->f(3);
  pb->f();
}
```

上記の例の出力は、以下のとおりです。

```
void B::f(int)
void A::f()
```

関数 f を using 宣言を指定して、基底クラス A から派生クラス B に導入し、さらに A::f として同じパラメーター型を持つ B::f という名前の関数が存在するとします。関数 B::f は、関数 A::f と競合するというより、むしろそれを隠蔽します。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
  void f() { }
  void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
  B obj_B;
  obj_B.f(3);
}
```

上記の例の出力は、以下のとおりです。

```
void B::f(int)
```

関連情報

- 205 ページの『第 10 章 多重定義 (C++ のみ)』
- 6 ページの『名前の隠蔽 (C++ のみ)』
- 258 ページの『using 宣言とクラス・メンバー』

クラス・メンバーのアクセスの変更

クラス B がクラス A の直接基底クラスであるとします。クラス B が、クラス A のメンバーにアクセスすることを制限するには、アクセス指定子 protected または private のどちらかを使用して、B を A から派生させてください。

クラス B から継承された、クラス A のメンバー x のアクセスを増やすには、using 宣言を使用してください。 using 宣言を指定して x へのアクセスを制限することはできません。次のメンバーのアクセスを増やすことができます。

- private として継承されたメンバー。 (using 宣言は、メンバーの名前にアクセスするはずなので、private として宣言されたメンバーのアクセスを増やすことはできません。)
- protected として継承、または宣言されたメンバー。

次の例は、このことを示しています。

```
struct A {
protected:
   int y;
public:
  int z;
};

struct B : private A { };

struct C : private A {
public:
  using A::y;
  using A::z;
};

struct D : private A {
protected:
  using A::y;
  using A::z;
};

struct E : D {
   void f() {
    y = 1;
    z = 2;
  }
};

struct F : A {
public:
  using A::y;
private:
  using A::z;
};

int main() {
  B obj_B;
```

```
//  obj_B.y = 3;
//  obj_B.z = 4;

  C obj_C;
  obj_C.y = 5;
  obj_C.z = 6;

  D obj_D;
//  obj_D.y = 7;
//  obj_D.z = 8;

  F obj_F;
  obj_F.y = 9;
  obj_F.z = 10;
}
```

コンパイラーは、上記の例から、次の割り当てを許可しません。

- obj_B.y = 3 および obj_B.z = 4: メンバー y および z は、private として継承されました。
- obj_D.y = 7 および obj_D.z = 8: メンバー y および z は、private として継承されましたが、それらのアクセスは protected に変更されました。

コンパイラーは、上記の例から、次のステートメントを許可します。

- D::f() の y = 1 および z = 2: メンバー y および z は、private として継承されましたが、それらのアクセスは protected に変更されました。
- obj_C.y = 5 および obj_C.z = 6: メンバー y および z は private として継承されましたが、それらのアクセスは public に変更されました。
- obj_F.y = 9: メンバー y のアクセスは、protected から public に変更されました。
- obj_F.z = 10: メンバー z のアクセスは、public のままです。 private の using 宣言 using A::z は、z のアクセスに影響を与えません。

関連情報

- 243 ページの『メンバー・アクセス』
- 256 ページの『継承されたメンバー・アクセス』

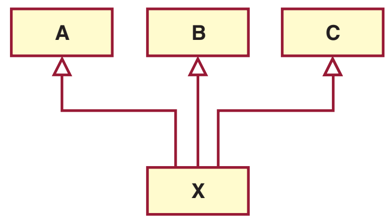
多重継承

1 つのクラスを複数の基底クラスから派生できます。複数の直接基底クラスから一つのクラスが派生することを、*多重継承* と呼びます。

次の例では、クラス A、B、および C は、派生クラス X の直接の基底クラスです。

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

次の*継承グラフ* は、上記の例で示した継承の関係を説明しています。矢印は、矢印の尾部の地点にあるクラスの直接基底クラスを指し示します。

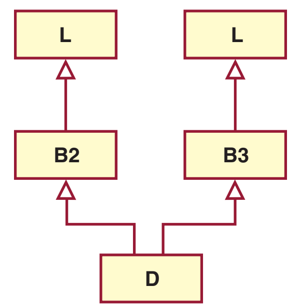


派生の順序は、コンストラクターによるデフォルト初期化およびデストラクターによる終結処理の順序の決定にのみ関係します。

直接の基底クラスは、派生クラスの基底リストに 2 回以上現れることはできません。

```
class B1 { /* ... */ };                     // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

ただし、次の例に示すように、派生クラスは間接の基底クラスを複数回継承することができます。




```
class L { /* ... */ }; // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

上記の例では、クラス D が、クラス B2 を介して、間接基底クラス L を 1 回継承し、クラス B3 を介して 1 回継承します。ただし、クラス L の 2 つのサブオブジェクトが存在し、両方ともクラス D を介してアクセスできるので、あいまいになる可能性があります。これは、修飾されたクラス名を使用して、クラス L を参照することによって避けることができます。次に例を示します。

```
B2::L
```

または

```
B3::L
```

また、253 ページの『派生』 に示すように、基底クラスの宣言に基底指定子 virtual を使用しても、このようなあいまいさを避けることができます。

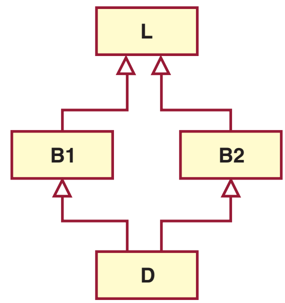
仮想基底クラス

共通の基底クラス A を持つ 2 つの派生クラス B および C があり、さらに B および C から継承した別のクラス D があるとします。基底クラス A を*仮想* として宣言することで、B および C が、同じ A のサブオブジェクトを共用していることを保証できます。

次の例では、クラス D のオブジェクトには、クラス L の 2 つの別個のサブオブジェクトがあり、一方はクラス B1 を介し、もう一方はクラス B2 を介しています。クラス B1 および B2 の*基底リスト* の基底ク

ラス指定子に、キーワード virtual を使用することで、クラス B1 およびクラス B2 が共用している、型 L のただ一つのサブオブジェクトが存在することを指示できます。

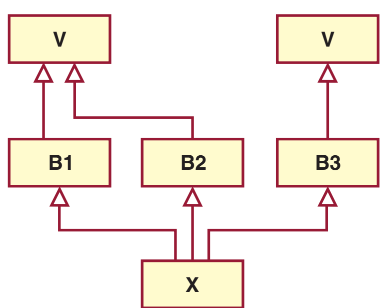
次に例を示します。




```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

この例で、キーワード virtual を使用すると、クラス D のオブジェクトが、クラス L のサブオブジェクトを 1 つだけ継承するようにすることができます。

派生クラスが仮想基底クラスと非仮想基底クラスを両方とも持つ場合があります。次に例を示します。




```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class X : public B1, public B2, public B3 { /* ... */
};
```

上記の例では、クラス x にクラス V のサブオブジェクトが 2 個あり、一つはクラス B1 とクラス B2 で共用され、もう一方はクラス B3 を介して共用されます。

関連情報

- 253 ページの『派生』

マルチアクセス

仮想基底クラスを含む継承グラフでは、複数のパスを経由して到達できる名前は、最大広範囲のアクセスを提供するパスを介してアクセスされます。

次に例を示します。

```
class L {
public:
  void f();
};

class B1 : private virtual L { };

class B2 : public virtual L { };

class D : public B1, public B2 {
public:
   void f() {
    // L::f() is accessed through B2
    // and is public
    L::f();
  }
};
```

上記の例では、関数 f() はクラス B2 を介してアクセスされます。クラス B2 は公的に継承され、クラス B1 は私的に継承されているので、クラス B2 の方がより多くのアクセスを提供します。

関連情報

- 243 ページの『メンバー・アクセス』
- 256 ページの『protected メンバー』
- 257 ページの『基底クラス・メンバーのアクセス制御』

あいまいな基底クラス

クラスを派生させるとき、基底クラスと派生クラスとに同じ名前のメンバーがあると、あいまいさが生じる可能性があります。固有な関数、またはオブジェクトを参照しない名前または修飾名を使用すると、基底クラス・メンバーへのアクセスがあいまいになります。派生クラス内であいまいな名前のメンバーを宣言してもエラーではありません。あいまいなメンバー名を使用すると、そのあいまいさにエラーとしてフラグを付けます。

例えば、A と B という名前の 2 つのクラスが、両方とも x という名前のメンバーを持ち、C という名前のクラスは、A と B の両方から継承するとします。クラス C から x にアクセスする試みは、あいまいになります。スコープ・レゾリューション (::) 演算子を使用して、そのクラス名でメンバーを修飾することによって、あいまいさを解消することができます。

```
class B1 {
public:
  int i;
  int j;
  void g(int) { }
};

class B2 {
public:
  int j;
  void g() { }
};

class D : public B1, public B2 {
public:
  int i;
};

int main() {
  D dobj;
  D *dptr = &dobj;
```

```
  dptr->i = 5;
//  dptr->j = 10;
  dptr->B1::j = 10;
//  dobj.g();
  dobj.B2::g();
}
```

ステートメント dptr->j = 10 は、B1 と B2 の両方に名前 j が現れるので、あいまいになります。B1::g(int) および B2::g() が異なるパラメーターを持っていますが、名前 g が B1 および B2 の両方に現れるので、ステートメント dobj.g() は、あいまいになります。

コンパイラーはコンパイル時にあいまいさを検査します。あいまいさの検査は、アクセス制御や型検査の前に行われるので、同じ名前の複数のメンバーの中の 1 つだけが派生クラスからアクセス可能である場合でも、あいまいさが検出される可能性があります。

名前の隠蔽

A と B という名前の 2 つのサブオブジェクトには、両方とも x という名前のメンバーがあるとします。A が B の基底クラスである場合、サブオブジェクト B のメンバー名 x は、サブオブジェクト A のメンバー名 x を隠します。次の例は、このことを示しています。

```
struct A {
   int x;
};

struct B: A {
   int x;
};

struct C: A, B {
   void f() { x = 0; }
};

int main() {
   C i;
   i.f();
}
```

関数 C::f() の割り当て x = 0 は、宣言 B::x が A::x を隠しているので、あいまいになりません。しかし、コンパイラーは、B を介してサブオブジェクト A にすでにアクセスしているので、コンパイラーは、A からの C の派生が冗長になっていることを警告します。

基底クラス宣言は、継承グラフの 1 つのパスに従っては、隠すことができますが、別のパスでは隠されません。次の例は、このことを示しています。

```
struct A { int x; };
struct B { int y; };
struct C: A, virtual B { };
struct D: A, virtual B {
   int x;
   int y;
};
struct E: C, D { };

int main() {
   E e;
//   e.x = 1;
   e.y = 2;
}
```

割り当て e.x = 1 は、あいまいです。宣言 D::x は、パス D::A::x に従って A::x を隠しますが、パス C::A::x では、A::x を隠しません。したがって、変数 x は、D::x または A::x のいずれかを参照できま

す。割り当て e.y = 2 は、あいまいではありません。宣言 D::y は、B が仮想基底クラスなので、パス D::B::y および C::B::y の両方で B::y を隠します。

あいまいさと using 宣言

A という名前のクラスから継承している C という名前のクラスがあり、さらに x が A のメンバー名だとします。using 宣言を使用して A::x を C に宣言し、x も C のメンバーであれば、C::x は、A::x を隠しません。したがって、using 宣言は、継承メンバーによるあいまいさを解決することはできません。次の例は、このことを示しています。

```
struct A {
   int x;
};

struct B: A { };

struct C: A {
   using A::x;
};

struct D: B, C {
   void f() { x = 0; }
};

int main() {
   D i;
   i.f();
}
```

コンパイラーは、割り当てがあいまいなので、関数 D::f() の割り当て x = 0 を許可しません。コンパイラーは、x を 2 つの方法で検索でき、B::x として、または C::x として見つけ出します。

明確なクラス・メンバー

コンパイラーは、オブジェクトが持っている型 A のサブオブジェクトの数に関係なく、基底クラス A に定義された、静的メンバー、ネスト型、および列挙子をあいまいさなく検出することができます。次の例は、このことを示しています。

```
struct A {
   int x;
   static int s;
   typedef A* Pointer_A;
   enum { e };
};

int A::s;

struct B: A { };

struct C: A { };

struct D: B, C {
   void f() {
      s = 1;
      Pointer_A pa;
      int i = e;
//      x = 1;
   }
};

int main() {
   D i;
   i.f();
}
```

コンパイラーは、割り当て s = 1、宣言 Pointer_A pa、およびステートメント int i = e を許可します。静的変数 s、typedef Pointer_A、および列挙子 e は、それぞれ 1 つだけあります。コンパイラーは、x がクラス B またはクラス C からアクセスされるので、割り当て x = 1 を許可しません。

ポインター型変換

派生クラスのポインターまたは参照から基底クラスのポインターまたは参照への変換 (明示的または暗黙の) は、同一のアクセス可能基底クラス・オブジェクトを一義的に参照しなければなりません。 (*アクセス可能基底クラス* とは、継承の階層の中で隠蔽されておらず、またあいまいでもない、public に派生された基底クラスです。) 次に例を示します。

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
      Z z;
      X* xptr = &z;      // valid
      Y* yptr = &z;      // valid
      W* wptr = &z;       // error, ambiguous reference to class W
                          // X's W or Y's W ?
}
```

仮想基底クラスを使用すれば、あいまいな参照を避けることができます。たとえば、次のとおりです。

```
class W { /* ... */ };
class X : public virtual W { /* ... */ };
class Y : public virtual W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
      Z z;
      X* xptr = &z;      // valid
      Y* yptr = &z;      // valid
      W* wptr = &z;      // valid, W is virtual therefore only one
                         // W subobject exists
}
```

以下の条件が真である場合、基底クラスのメンバーへのポインターは、派生クラスのメンバーへのポインターに変換することができます。

- 型変換が、あいまいではない。基底クラスの複数インスタンスが派生クラス内にあると、変換はあいまいになります。
- 派生クラスへのポインターは、基底クラスへのポインターに変換することができる。その場合、その基底クラスは、*アクセス可能* であるといいます。
- メンバー型が一致する。例えば、クラス A は、クラス B の基底クラスであると想定します。型 int の A のメンバーを指すポインターを、型 float の型 B のメンバーを指すポインターには、変換できません。
- 基底クラスが、仮想ではない。

多重定義解決

多重定義解決は、コンパイラーが任意の関数名をあいまいさなく検出した後で 行われます。次の例は、このことを示しています。

```
struct A {
   int f() { return 1; }
};

struct B {
```

```
   int f(int arg) { return arg; }
};

struct C: A, B {
   int g() { return f(); }
};
```

コンパイラーは、名前 f が A および B の両方で宣言されているので、C::g() の f() の関数呼び出しを許可しません。コンパイラーは、多重定義解決が基底一致 A::f() を選択する前に、あいまいさエラーを検出します。

関連情報

- 108 ページの『スコープ・レゾリューション演算子 :: (C++ のみ)』
- 263 ページの『仮想基底クラス』

仮想関数

C++ は、デフォルトによりコンパイル時に、関数呼び出しを正しい関数定義とマッチングさせせます。これは*静的バインディング* と呼ばれます。コンパイラーに、関数呼び出しと正しい関数定義を、実行時にマッチングさせることを指定できます。これは、*動的バインディング* と呼ばれます。特定の関数に対して、コンパイラーに動的バインディングを使用させたい場合、キーワード virtual を指定して関数を宣言します。

次の例は、静的バインディングと動的バインディングの違いを示しています。最初の例は、静的バインディングを示しています。

```
#include <iostream>
using namespace std;

struct A {
   void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
   arg.f();
}

int main() {
   B x;
   g(x);
}
```

上記の例の出力は、以下のとおりです。

```
Class A
```

関数 g() が呼び出されると、引数は、型 B のオブジェクトを参照しますが、関数 A::f() がコールされます。コンパイル時にコンパイラーが唯一認知できるのは、関数 g() の引数が、A から派生したオブジェクトの参照である可能性があるということです。引数が、型 A のオブジェクトへの参照なのか、または型 B のオブジェクトへのものなのかどうかを判別することはできません。しかし、これは実行時に判別されます。次の例は、A::f() が virtual キーワードで宣言されている点を除いては、直前の例と同じです。

```
#include <iostream>
using namespace std;
```

```
struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
   arg.f();
}

int main() {
   B x;
   g(x);
}
```

上記の例の出力は、以下のとおりです。

```
Class B
```

virtual キーワードは、参照の型ではなく、参照先のオブジェクト型を使用して、f() のための適切な定義を選択する必要があることをコンパイラーに指示します。

したがって、*仮想関数* とは、別の派生クラスのために再定義できるメンバー関数です。また、たとえオブジェクトの基底クラスに対するポインター、または参照を使用して関数を呼び出したとしても、対応する派生クラスのオブジェクト向けに再定義された仮想関数を、コンパイラーが呼び出せることも保証できます。

仮想関数を宣言するクラス、または継承するクラスは、*ポリモアフィック* と呼ばれます。

仮想メンバー関数は、任意の派生クラスにおいて、どのメンバー関数とも同じように、再定義することができます。クラス A で f という名前の仮想関数を宣言し、A から直接的、または間接的に B という名前のクラスを派生させたとします。 A::f と同じ名前、および同じパラメーター・リストを指定して、クラス B で f という名前の関数を宣言する場合、B::f もまた仮想であり (virtual キーワードを使用して B::f を宣言しているかどうかには関係なく)、それは A::f を*オーバーライド* します。しかし、A::f と B::f のパラメーター・リストが異なり、A::f と B::f は違うものであると見なされる場合、B::f は A::f をオーバーライドしませんし、B::f は、仮想ではありません (virtual キーワードを使用してこれを宣言していない場合)。代わりに、B::f は、A::f を*隠します*。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f(int) { cout << "Class B" << endl; }
};

struct C: B {
   void f() { cout << "Class C" << endl; }
};

int main() {
   B b; C c;
   A* pa1 = &b;
   A* pa2 = &c;
//   b.f();
   pa1->f();
   pa2->f();
}
```

上記の例の出力は、以下のとおりです。

```
Class A
Class C
```

関数 B::f は、仮想ではありません。これは A::f を隠します。つまり、コンパイラーは、関数呼び出し b.f() を許可しません。関数 C::f は、仮想です。A::f は C で可視ではありませんが、これは A::f をオーバーライドします。

基底クラス・デストラクターを仮想として宣言する場合、派生クラス・デストラクターは、デストラクターが継承されていなくても、基底クラス・デストラクターをオーバーライドします。

オーバーライドする仮想関数の戻りの型は、オーバーライドされる仮想関数の戻りの型とは異なる場合があります。このオーバーライド関数は、*共変仮想関数* と呼ばれます。 B::f が、仮想関数 A::f をオーバーライドするとします。 A::f と B::f の戻りの型は、次の条件のすべてが満たされるかどうかで変わります。

- 関数 B::f は、型 T のクラスに対する参照、またはポインターを戻し、A::f は、T の明確な直接、あるいは間接基底クラスに対するポインター、または参照を戻す。
- B::f が戻すポインター、あるいは参照における const または volatile 修飾は、A::f が戻すポインター、または参照と同等な、あるいはより低い const または volatile 修飾を持っている。
- B::f の戻りの型は、B::f の宣言ポイントで完了する必要がある。または、型 B となる。

次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A { };

class B : private A {
   friend class D;
   friend class F;
};

A global_A;
B global_B;

struct C {
   virtual A* f() {
      cout << "A* C::f()" << endl;
      return &global_A;
   }
};

struct D : C {
   B* f() {
      cout << "B* D::f()" << endl;
      return &global_B;
   }
};

struct E;

struct F : C {

// Error:
//   E is incomplete
//   E* f();
};

struct G : C {
```

```
// Error:
//   A is an inaccessible base class of B
//   B* f();
};

int main() {
   D d;
   C* cp = &d;
   D* dp = &d;

   A* ap = cp->f();
   B* bp = dp->f();
};
```

上記の例の出力は、以下のとおりです。

```
B* D::f()
B* D::f()
```

ステートメント A* ap = cp->f() は、D::f() を呼び出して、戻されるポインターを型 A* に変換します。ステートメント B* bp = dp->f() は、D::f() も呼び出しますが、戻されるポインターを変換しません。戻りの型は B* となります。コンパイラーは、E が完全なクラスではないので、仮想関数 F::f() の宣言を許可しません。コンパイラーは、クラス A が B にアクセス可能な基底クラスではないので、仮想関数 G::f() の宣言を許可しません (フレンド・クラス D および F と異なり、B の定義は、クラス G のメンバーにアクセス権を与えません)。

定義によると、仮想関数は、基底クラスのメンバー関数であり、特定のオブジェクトに従って、関数のどのインプリメンテーションを呼び出すかを決めるので、仮想関数をグローバルにも静的にもすることはできません。仮想関数を別のクラスのフレンドとして宣言することができます。

基底クラスにおいて仮想と宣言した関数の場合でも、スコープ・レゾリューション (::) 演算子を使用すれば、それを直接にアクセスすることができます。この場合、仮想関数呼び出しのメカニズムを抑止し、基底クラスで定義された関数インプリメンテーションが使用されます。さらに、派生クラスで仮想メンバー関数を再オーバーライドしなければ、その関数に対する呼び出しでは、基底クラスで定義された関数インプリメンテーションが使用されます。

仮想関数は次のいずれかでなければなりません。

- 定義された
- 宣言された純粋
- 定義され宣言された純粋

1 つまたは複数の純粋仮想メンバー関数を含む基底クラスは、*抽象クラス* と呼ばれます。

関連情報

- 274 ページの『抽象クラス』

あいまいな仮想関数呼び出し

1 つの仮想関数を、2 つ以上のあいまいな仮想関数でオーバーライドすることはできません。これは、仮想基底クラスから派生した 2 つの非仮想基底から継承する派生クラスで発生する可能性があります。

次に例を示します。

```
class V {
public:
 virtual void f() { }
```

```
};

class A : virtual public V {
  void f() { }
};

class B : virtual public V {
 void f() { }
};

// Error:
// Both A::f() and B::f() try to override V::f()
class D : public A, public B { };

int main() {
   D d;
  V* vptr = &d;

  // which f(), A::f() or B::f()?
 vptr->f();
}
```

コンパイラーは、クラス D の定義を許可しません。クラス A では、A::f() のみが V::f() をオーバーライドします。同様にクラス B でも、B::f() のみが V::f() をオーバーライドします。ただし、クラス D では、A::f() と B::f() の両方が、V::f() をオーバーライドしようとします。上記の例で示すように、D オブジェクトが、クラス V を指すポインターを使用して参照される場合、コンパイラーは、どちらの関数を呼び出すべきかを決めることができないので、このような試みは許されません。 1 つの関数のみが仮想関数をオーバーライドできます。

同じクラス型の別個のインスタンスがあることによって、仮想関数のあいまいなオーバーライドが起きた場合、特殊なケースになります。次の例では、クラス D には、クラス A の 2 つの別々のサブオブジェクトがあります。

```
#include <iostream>
using namespace std;

struct A {
   virtual void f() { cout << "A::f()" << endl; };
};

struct B : A {
   void f() { cout << "B::f()" << endl;};
};

struct C : A {
   void f() { cout << "C::f()" << endl;};
};

struct D : B, C { };

int main() {
   D d;

   B* bp = &d;
   A* ap = bp;
   D* dp = &d;

   ap->f();
//   dp->f();
}
```

クラス D には、クラス A のオカレンスが 2 つあり、一つは B から継承されたもので、もう一つは C から継承されたものです。したがって、仮想関数 A::f にも 2 つのオカレンスがあります。ステートメント

ap->f() は、D::B::f を呼び出します。しかし、コンパイラーは、D::B::f または D::C::f のどちらも呼び出すことができるので、ステートメント dp->f() を許可しません。

仮想関数アクセス

仮想関数へのアクセスは、宣言時に指定されます。後で仮想関数をオーバーライドする関数のアクセス規則が、仮想関数のアクセス規則に影響を与えることはありません。一般に、オーバーライドするメンバー関数のアクセスは未知です。

クラス・オブジェクトを指すポインターまたは参照で仮想関数を呼び出す場合は、仮想関数のアクセスの判別に、クラス・オブジェクトの型は使用されません。代わりに、使用されるのは、クラス・オブジェクトを指すポインターまたは参照の型です。

次の例では、型 B* を持つポインターを使用して関数 f() を呼び出すと、関数 f() へのアクセスの判別に bptr が使用されます。クラス D で定義された f() の定義が実行されますが、クラス B の中のメンバー関数 f() のアクセスが、使用されます。型 D* を持つポインターを使用して関数 f() を呼び出すと、関数 f() へのアクセスの判別に dptr が使用されます。 f() はクラス D で private と宣言されているので、この呼び出しはエラーになります。

```
class B {
public:
  virtual void f();
};

class D : public B {
private:
  void f();
};

int main() {
  D dobj;
  B* bptr = &dobj;
  D* dptr = &dobj;

  // valid, virtual B::f() is public,
  // D::f() is called
  bptr->f();

  // error, D::f() is private
  dptr->f();
}
```

抽象クラス

抽象クラス とは、特に基底クラスとして使用するように設計されたクラスです。抽象クラスには、少なくとも 1 つの*純粋仮想関数* が含まれています。クラス宣言の中の仮想メンバー関数の宣言で、*純粋指定子* (= 0) を使用することによって、純粋仮想関数を宣言することができます。

次に抽象クラスの例を示します。

```
class AB {
public:
  virtual void f() = 0;
};
```

関数 AB::f は、純粋仮想関数です。関数宣言に、純粋指定子と定義の両方を入れることはできません。例えば、コンパイラーは、次の表記を許可しません。

```
struct A {
  virtual void g() { } = 0;
};
```

抽象クラスをパラメーター型、関数からの戻りの型、または明示型変換の型として使用することはできませんし、抽象クラスのオブジェクトも宣言することはできません。ただし、抽象クラスを指すポインター、および参照を宣言することは可能です。次の例は、このことを示しています。

```
struct A {
  virtual void f() = 0;
};

struct B : A {
  virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

// Error:
// Class A is an abstract class
//   A a;

   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

クラス A は、抽象クラスです。コンパイラーは、関数宣言 A g() または void h(A)、オブジェクト a の宣言、そして b の型 A への静的キャストも許可しません。

仮想メンバー関数は、継承されます。派生クラスにある各純粋仮想関数をオーバーライドしない限り、抽象基底クラスから派生するクラスも抽象になります。

次に例を示します。

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
  D2 d;
}
```

コンパイラーは、オブジェクト d の宣言を許可しません。 D2 が、AB から純粋仮想関数 f() を継承した抽象クラスだからです。関数 D2::g() を定義すれば、コンパイラーは、オブジェクト d の宣言を行うことができます。

非抽象クラスから抽象クラスを派生させたり、非純粋仮想関数を純粋仮想関数でオーバーライドできることに注意してください。

抽象クラスのコンストラクターまたはデストラクターから、メンバー関数を呼び出すことができます。ただし、コンストラクターから純粋仮想関数を呼び出した (直接または間接) 結果は、未定義です。次の例は、このことを示しています。

```
struct A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

A のデフォルトのコンストラクターは、直接的、および間接的 (indirect() を介して) の両方で、純粋仮想関数 direct() を呼び出します。

コンパイラーは、純粋仮想関数の直接呼び出しに警告を出しますが、間接呼び出しには警告を出しません。

関連情報

- 269 ページの『仮想関数』
- 274 ページの『仮想関数アクセス』

第 14 章 特殊メンバー関数 (C++ のみ)

デフォルトのコンストラクター、デストラクター、コピー・コンストラクター、およびコピー代入演算子は、*特殊なメンバー関数* です。これらの関数は、クラス・オブジェクトを作成、破棄、変換、初期化、およびコピーします。これらの関数については以下のセクションで説明します。

- 『コンストラクターとデストラクターの概要』
- 279 ページの『コンストラクター』
- 286 ページの『デストラクター』
- 290 ページの『変換コンストラクター』
- 292 ページの『変換関数』
- 293 ページの『コピー・コンストラクター』

コンストラクターとデストラクターの概要

データや関数を含むクラスの内部構造は複雑なので、オブジェクトの初期化やクラス終結処理は、単純なデータ構造の場合より格段に複雑です。コンストラクターやデストラクターは、クラス・オブジェクトの構成や破棄に使用されるクラスの特殊なメンバー関数です。構成では、オブジェクトのストレージ割り当てや初期化もあわせて行われる場合があります。破棄に際しては、オブジェクトのストレージの終結処理や割り当て解除も行われる場合があります。

他のメンバー関数と同様、コンストラクターとデストラクターは、クラス宣言の中で宣言されます。これらは、インラインまたはクラス宣言の外で定義することができます。コンストラクターは、デフォルトの引数を持つことができます。コンストラクターは、他のメンバー関数と異なり、メンバー初期化リストを持つことができます。コンストラクターとデストラクターには、次の制約事項が適用されます。

- コンストラクターとデストラクターには戻りの型がなく、また値を戻すこともできません。
- 参照とポインターは、コンストラクターとデストラクターには、そのアドレスを取得できないので、使用することはできません。
- コンストラクターは、`virtual` というキーワードでは、宣言できません。
- コンストラクターおよびデストラクターは `static`、`const`、または `volatile` として宣言することができません。
- 共用体には、コンストラクターやデストラクターのあるクラス・オブジェクトを入れることができません。

コンストラクターとデストラクターは、メンバー関数と同じアクセス規則に従います。例えば、protected を使用してコンストラクターを宣言した場合、それを使用してクラス・オブジェクトを使用できるのは、派生クラスとフレンドだけです。

コンパイラーは、クラス・オブジェクトを定義するときはコンストラクターを、クラス・オブジェクトがスコープ外に出るときはデストラクターを、それぞれ自動的に呼び出します。コンストラクターは、その `this` ポインターが参照するクラス・オブジェクトにメモリーを割り振ることはありませんが、そのクラス・オブジェクトが参照するオブジェクトより多くのオブジェクトにストレージを割り振る場合があります。オブジェクトのためにメモリー割り振りが必要な場合は、コンストラクターが明示的に `new` 演算子を呼び出すことができます。終結処理時に、デストラクターは、対応するコンストラクターによって割り当てられたオブジェクトを解放します。オブジェクトを解放するには、`delete` 演算子を使用します。

© Copyright IBM Corp. 1998, 2010

派生クラスはその基底クラスのコンストラクターやデストラクターを継承または多重定義しませんが、基底クラスのコンストラクターやデストラクターを呼び出します。デストラクターは、virtual というキーワードで宣言できます。

コンストラクターは、ローカルまたは一時クラス・オブジェクトが作成されるときにも呼び出され、デストラクターは、ローカルまたは一時オブジェクトがスコープを超えたときに呼び出されます。

コンストラクターまたはデストラクターから、メンバー関数を呼び出すことができます。クラス A のコンストラクター、またはデストラクターから、直接的に、あるいは間接的に仮想関数を呼び出すことができます。この場合、呼び出される関数は A で定義されているものか、A の基底クラスであり、A から派生したクラスでオーバーライドされた関数ではありません。これにより、コンストラクター、またはデストラクターから、非構成オブジェクトにアクセスする可能性を回避します。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void g() { cout << "void A::g()" << endl; }
  virtual void h() { cout << "void A::h()" << endl; }
};

struct B : A {
  virtual void f() { cout << "void B::f()" << endl; }
  B() {
    f();
    g();
    h();
  }
};

struct C : B {
  virtual void f() { cout << "void C::f()" << endl; }
  virtual void g() { cout << "void C::g()" << endl; }
  virtual void h() { cout << "void C::h()" << endl; }
};

int main() {
  C obj;
}
```

次に、上記の例の出力を示します。

```
void B::f()
void A::g()
void A::h()
```

例では obj いう名前で型 C のオブジェクトを作成しますが、B のコンストラクターは、C が B から派生しているので、C でオーバーライドされた関数は、どれも呼び出しません。

コンストラクター、またはデストラクターで typeid または dynamic_cast 演算子を使用でき、同様に、コンストラクターのメンバー初期化指定子も使用できます。

関連情報

- 141 ページの『new 式 (C++ のみ)』
- 145 ページの『delete 式 (C++ のみ)』

コンストラクター

コンストラクター は、そのクラスと同じ名前を持つメンバー関数です。次に例を示します。

```
class X {
public:
  X();      // constructor for class X
};
```

コンストラクターは、そのクラス型のオブジェクトの作成に使用され、オブジェクトを初期化できます。

コンストラクターは、virtual または static として宣言できませんし、const、volatile、または const volatile として宣言することもできません。

コンストラクターの戻りの型は、指定しません。コンストラクター本体にあるリターン・ステートメントは、戻り値を持てません。

デフォルト・コンストラクター

デフォルト・コンストラクター とは、パラメーターがないか、ある場合でも、すべての パラメーターにデフォルト値があるコンストラクターです。

クラス A にユーザー定義のコンストラクターが必要であるが、それが存在しない場合、コンパイラーは、デフォルトのパラメーターなしコンストラクター A::A() を暗黙的に*宣言*します。このコンストラクターは、そのクラスのインライン・パブリック・メンバーです。コンパイラーが、コンストラクターを使用して、型 A のオブジェクトを作成する時に、コンパイラーは、A::A() を暗黙的に*定義* します。コンストラクターは、コンストラクター初期化指定子もヌル・ボディも持つようにはなりません。

コンパイラーは、まず最初に暗黙的に宣言された基底クラスのコンストラクターと、クラス A の非静的データ・メンバーを暗黙的に定義してから、暗黙的に宣言された A のコンストラクターを定義します。定数や参照型メンバーを持つクラスに対して、デフォルトのコンストラクターは作成されません。

クラス A のコンストラクターは、次のことがすべて真であれば、*単純* です。

- 暗黙的に定義される
- A に仮想関数がなく、仮想基底クラスもない
- A の直接基底クラスが、すべて単純コンストラクターを持っている
- A のすべての非静的データ・メンバーに関するクラスが、単純コンストラクターを持っている

上記のいずれかが偽であれば、コンストラクターは、*非単純* です。

共用体メンバーは、非単純コンストラクターを持つクラス型にはできません。

すべての関数と同様、コンストラクターは、デフォルト引数を持つことができます。これらは、メンバー・オブジェクトの初期化に使用されます。デフォルト値が提供される場合、末尾引数は、コンストラクターの式リストで省略できます。コンストラクターにデフォルト値を持たない引数がある場合、それはデフォルト・コンストラクターではないことに注意してください。

クラス A の*コピー・コンストラクター* とは、その第 1 パラメーターが、型 A&、const A&、volatile A&、または const volatile A& のいずれかであるコンストラクターです。コピー・コンストラクターは、あるクラス・オブジェクトを、同じクラス型の別のクラス・オブジェクトからコピーするために使用されます。そのクラスと同じタイプの引数でコピー・コンストラクターを使用することはできません。参照を使用する必要があります。すべてがデフォルト引数である限り、追加デフォルト引数を持つコピー・コンストラクターを提供することはできます。あるクラスにユーザー定義のコンストラクターが必要であるにもかかわ

らず、それが存在しない場合、コンパイラーは、そのクラス用に、パブリック・アクセスを持つコピー・コンストラクターを作成します。コンパイラーは、メンバーまたは基底クラスにアクセス不能なコピー・コンストラクターがあるクラスに対しては、コピー・コンストラクターを作成しません。

次のコードは、デフォルト・コンストラクターとコピー・コンストラクターがある 2 つのクラスを示しています。

```
class X {
public:

  // default constructor, no arguments
  X();

  // constructor
  X(int, int , int = 0);

  // copy constructor
  X(const X&);

  // error, incorrect argument type
  X(X);
};

class Y {
public:

  // default constructor with one
  // default argument
  Y( int = 0);

  // default argument
  // copy constructor
  Y(const Y&, int = 0);
};
```

関連情報

- 293 ページの『コピー・コンストラクター』

コンストラクターによる明示的初期化

クラス・オブジェクトは、コンストラクターを用いて明示的に初期化されるか、またはデフォルト・コンストラクターを持っていなければなりません。コンストラクターを使用する明示的初期化は、集合体初期化の場合を除き、非静的定数および参照クラス・メンバーを初期化する唯一の方法です。

ユーザー宣言のコンストラクター、仮想関数、private メンバーまたは protected 非静的データ・メンバー、および基底クラスのいずれも持たないクラス・オブジェクトは、*集合体* と呼ばれます。 集合体の例としては、C 形式の構造体および共用体があります。

クラス・オブジェクトを作成する場合、そのオブジェクトを明示的に初期化します。クラス・オブジェクトを初期化するには、次の 2 つの方法があります。

- 括弧で囲んだ式のリストの使用。コンパイラーは、このリストをコンストラクターの引数リストとして使用し、クラスのコンストラクターを呼び出します。
- 単一初期化値、および = 演算子の使用。このような型の式は、代入でなく初期化なので、代入演算子関数 (これが存在する場合でも) は、呼び出されません。単一引数の型は、コンストラクターに対する最初の引数の型と一致していなければなりません。コンストラクターに残りの引数がある場合、これらの引数はデフォルト値を持っている必要があります。

初期化指定子の構文

次の例は、クラス・オブジェクトを明示的に初期化するいくつかのコンストラクターの宣言および使用の方法を示します。

```
// This example illustrates explicit initialization
// by constructor.
#include <iostream>
using namespace std;

class complx {
  double re, im;
public:

  // default constructor
  complx() : re(0), im(0) { }

  // copy constructor
  complx(const complx& c) { re = c.re; im = c.im; }

  // constructor with default trailing argument
  complx( double r, double i = 0.0) { re = r; im = i; }

  void display() {
    cout << "re = "<< re << " im = " << im << endl;
  }
};

int main() {

  // initialize with complx(double, double)
  complx one(1);

  // initialize with a copy of one
  // using complx::complx(const complx&)
  complx two = one;

  // construct complx(3,4)
  // directly into three
  complx three = complx(3,4);

  // initialize with default constructor
  complx four;

  // complx(double, double) and construct
  // directly into five
  complx five = 5;

  one.display();
  two.display();
  three.display();
  four.display();
  five.display();
}
```

上記の例で作成される出力は次のようになります。

```
re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0
```

関連情報

- 81 ページの『初期化指定子』

基底クラスおよびメンバーの初期化

コンストラクターは、次に示す 2 とおりの異なった方法でメンバーを初期化できます。コンストラクターは渡された引数を使用して、コンストラクター定義内のメンバー変数を初期化することができます。

```
complx(double r, double i = 0.0) { re = r; im = i; }
```

またはコンストラクターは、定義の中に*初期化指定子リスト* を含めることができますが、それらは、コンストラクター本体の前に置く必要があります。

```
complx(double r, double i = 0) : re(r), im(i) { /* ... */ }
```

どちらの方法でも、引数値は適切なクラスのデータ・メンバーに代入されます。

初期化指定子リストの構文

コンストラクター宣言の一部ではなく、コンストラクター定義の一部として初期化リストをインクルードします。次に例を示します。

```
#include <iostream>
using namespace std;

class B1 {
  int b;
public:
  B1() { cout << "B1::B1()" << endl; };

  // inline constructor
  B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};
class B2 {
  int b;
protected:
  B2() { cout << "B2::B2()" << endl; }

  // noninline constructor
  B2(int i);
};

// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {
  int d1, d2;
public:
  D(int i, int j) : B1(i+1), B2(), d1(i) {
    cout << "D1::D1(int, int)" << endl;
```

```
    d2 = j;}
};

int main() {
  D obj(1, 2);
}
```

上記の例の出力は、以下のとおりです。

```
B1::B1(int)
B1::B1()
D1::D1(int, int)
```

コンストラクターのある基底クラスまたはメンバーをコンストラクターを呼び出すことによって、明示的に初期化するのでない場合、コンパイラーは、自動的にデフォルト・コンストラクターのある基底クラスまたはメンバーを初期化します。上記の例では、クラス D のコンストラクター内の呼び出し B2() を除外すると (後に示すように)、空の式リストのあるコンストラクター初期化指定子が自動的に作成されて、B2 を初期化します。クラス D のコンストラクター (上記と下記に示されています) は、クラス D のオブジェクトと同じ構造体になります。

```
class D : public B1, public B2 {
  int d1, d2;
public:

  // call B2() generated by compiler
  D(int i, int j) : B1(i+1), d1(i) {
    cout << "D1::D1(int, int)" << endl;
    d2 = j;}
};
```

上記の例では、コンパイラーは、B2() のデフォルトのコンストラクターを自動的に呼び出します。

派生クラスがコンストラクターを呼び出すことができるようにするには、コンストラクターを public または protected 付きで宣言する必要があります。次に例を示します。

```
class B {
  B() { }
};

class D : public B {

  // error: implicit call to private B() not allowed
  D() { }
};
```

コンパイラーは、コンストラクターが private コンストラクター B::B() にアクセスできないので、D::D() の定義を許可しません。

初期化指定子リストを指定して、次のことを初期化する必要があります。それらは、デフォルト・コンストラクターのない基底クラス、参照データ・メンバー、非静的 const データ・メンバー、または定数データ・メンバーを含むクラス・タイプです。次の例は、このことを示しています。

```
class A {
public:
  A(int) { }
};

class B : public A {
  static const int i;
  const int j;
  int &k;
public:
  B(int& arg) : A(0), j(1), k(arg) { }
```

```
};

int main() {
  int x = 0;
  B obj(x);
};
```

データ・メンバー j および k、さらに基底クラス A は、B のコンストラクターの初期化指定子リストで初期化される必要があります。

クラスのメンバーを初期化する際、データ・メンバーを使用できます。次の例は、このことを示しています。

```
struct A {
  int k;
  A(int i) : k(i) { }
};
struct B: A {
  int x;
  int i;
  int j;
  int& r;
  B(int i): r(x), A(i), j(this->i), i(i) { }
};
```

コンストラクター B(int i) は、次のことを初期化します。

- B::x を参照するための B::r
- B(int i) への引数の値を指定したクラス A
- B::i の値を指定した B::j
- B(int i) への引数の値を指定した B::i

クラスのメンバーを初期化する場合、メンバー関数 (仮想メンバー関数を含む) を呼び出したり、あるいは演算子 typeid または dynamic_cast を使用することもできます。しかし、すべての基底クラスが初期化される前に、メンバー初期化リストにあるこれらの演算を実行する場合、その振る舞いは予期できません。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(f()), j(1234) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

上記の例の出力は、次の出力に類似しています。

```
Value of i: 8
Value of j: 1234
```

B のコンストラクターの初期化指定子 A(f()) の振る舞いは、未定義です。ランタイムは、B::f() を呼び出し、基底 A が初期化されていなくても、A::i にアクセスしようとします。

次の例は、B::B() の初期化指定子が異なる引数を持つ点を除いては、直前の例と同じです。

```
#include <iostream>
using namespace std;

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(5678), j(f()) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

次に、上記の例の出力を示します。

```
Value of i: 5678
Value of j: 5678
```

B のコンストラクターでは、初期化指定子 j(f()) の振る舞いは、明確に定義されています。 B::j が初期化される時、基底クラス A も初期化済みです。

関連情報

- 114 ページの『typeid 演算子 (C++ のみ)』
- 139 ページの『dynamic_cast 演算子 (C++ のみ)』

派生クラス・オブジェクトの構築順序

コンストラクターを使用して派生クラス・オブジェクトを作成する場合、そのオブジェクトは、次の順番で作成されます。

1. 基底リストに示されている順序にしたがって、仮想基底クラスが初期化される。
2. 宣言に示されている順序にしたがって、非仮想基底クラスが初期化される。
3. クラス・メンバーは、宣言の順番に (初期化リストでの順番には関係なく) 初期化される。
4. コンストラクターの本体が、実行される。

次の例は、このことを示しています。

```
#include <iostream>
using namespace std;
struct V {
  V() { cout << "V()" << endl; }
};
```

```
struct V2 {
  V2() { cout << "V2()" << endl; }
};
struct A {
  A() { cout << "A()" << endl; }
};
struct B : virtual V {
  B() { cout << "B()" << endl; }
};
struct C : B, virtual V2 {
  C() { cout << "C()" << endl; }
};
struct D : C, virtual V {
  A obj_A;
  D() { cout << "D()" << endl; }
};
int main() {
  D c;
}
```

上記の例の出力は、以下のとおりです。

```
V()
V2()
B()
C()
A()
D()
```

上記の出力は、型 D のオブジェクトを作成するために、C++ ランタイムがコンストラクターを呼び出す順序をリストしています。

関連情報

- 263 ページの『仮想基底クラス』

デストラクター

デストラクター は、オブジェクトを破棄するときに、ストレージの割り当て解除、およびクラス・オブジェクトとそのクラス・メンバーに関するその他の終結処理を行うために使用されます。オブジェクトがスコープを超えるか、明示的にオブジェクトを削除するときに、そのクラス・オブジェクトのデストラクターを呼び出します。

デストラクターは、そのクラスと同じ名前を持つメンバー関数で、接頭部として ~ (波形記号) が付きます。次に例を示します。

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

デストラクターは、引数を使用せず、戻りの型もありません。そのアドレスを取得することはできません。デストラクターを const、volatile、const volatile、または static で宣言できません。デストラクターは、virtual、または純粋 virtual で宣言できます。

あるクラスにユーザー定義のデストラクターが必要であるが、それが存在しない場合、コンパイラーは、デストラクターを暗黙的に宣言します。この暗黙的に宣言されたデストラクターは、そのクラスのインライン・パブリック・メンバーです。

コンパイラーがデストラクターを使用して、デストラクターのクラス型のオブジェクトを破棄する場合、コンパイラーは、暗黙的に宣言されたデストラクターを暗黙的に定義します。クラス A が、暗黙的に宣言されたデストラクターを持っているとします。次は、コンパイラーが A に対して暗黙的に定義を行う関数と同等です。

```
A::~A() { }
```

コンパイラーは、まず最初に暗黙的に宣言された基底クラスのデストラクターと、クラス A の非静的データ・メンバーを暗黙的に定義してから、暗黙的に宣言された A のデストラクターを定義します。

クラス A のデストラクターは、次のことがすべて真であれば*単純* です。

- 暗黙的に定義される
- A の直接基底クラスは、すべて単純デストラクターを持っている
- A のすべての非静的データ・メンバーに関するクラスが、単純デストラクターを持っている

上記のいずれかが偽であれば、デストラクターは*非単純* です。

共用体メンバーは、非単純デストラクターを持つクラス型にはできません。

クラス型であるクラス・メンバーは、独自のデストラクターを持つことができます。基底クラスと派生クラスは、両方ともデストラクターを持つことができますが、デストラクターは継承されません。基底クラス A または A のメンバーがデストラクターを持っており、A から派生したクラスがデストラクターを宣言しない場合、デフォルトのデストラクターが生成されます。

デフォルト・デストラクターは、基底クラスおよび派生クラスのメンバーのデストラクターを呼び出します。

基底クラスおよびメンバーのデストラクターは、それらのコンストラクターが完了する順番の逆順で呼び出されます。

1. クラス・オブジェクト用のデストラクターは、メンバーおよび基底用のデストラクターより前に呼び出されます。
2. 非静的基底クラスのデストラクターは、仮想基底クラスのデストラクターより前に、呼び出されます。
3. 非仮想基底クラスのデストラクターは、仮想基底クラスのデストラクターより前に、呼び出されます。

デストラクターを持つクラス・オブジェクトの例外をスロー (throw) すると、プログラムが catch ブロックの外に制御を渡すまで、スローする一時オブジェクトのデストラクターを呼び出しません。

自動オブジェクト (auto または register を宣言されたローカル・オブジェクト、あるいは static または extern として宣言されていないローカル・オブジェクト) または一時オブジェクトがスコープ外に渡されると、デストラクターが暗黙的に呼び出されます。構築された外部オブジェクトや静的オブジェクトのプログラム終了処理時にも、暗黙的にデストラクターを呼び出します。デストラクターは、new 演算子によって作成されたオブジェクトに対してユーザーが delete 演算子を使用すると呼び出されます。

次に例を示します。

```
#include <string>

class Y {
private:
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
```

```
  // Destructor
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // ...

  // Destructor ~Y is called before
  // control returns from main()
}
```

デストラクターを明示的に使用してオブジェクトを破棄することもできますが、この方法はお勧めできません。しかし、配置 new 演算子を使用して作成されたオブジェクトを破棄するために、オブジェクトのデストラクターを明示的に呼び出すことができます。次の例は、このことを示しています。

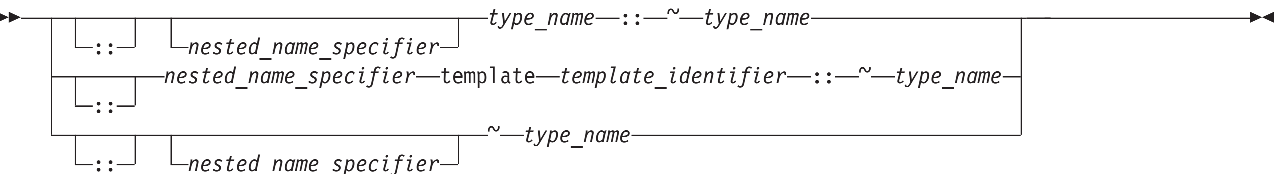
```
#include <new>
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
  char* p = new char[sizeof(A)];
  A* ap = new (p) A;
  ap->A::~A();
  delete [] p;
}
```

ステートメント A* ap = new (p) A は、型 A の 新規のオブジェクトを、フリー・ストレージにではなく、p が割り振ったメモリーに動的に作成します。ステートメント delete [] p は、p が割り振ったストレージを削除します。しかし、ランタイムは、A のデストラクターを明示的に呼び出すまでは (ステートメント ap->A::~A() と指定して)、ap が指すオブジェクトはまだ存在していると判断しています。

疑似デストラクター

疑似デストラクター は、非クラス型のデストラクターです。

疑似デストラクターの構文



次の例は、整数型の疑似デストラクターを呼び出します。

```
typedef int I;
int main() {
  I x = 10;
  x.I::~I();
  x = 20;
}
```

疑似デストラクターの呼び出し x.I::~I() は、まったく効果はありません。オブジェクト x は、破棄されておらず、代入 x = 20 はまだ有効です。疑似デストラクターは、非クラス型を有効にするためにデストラクターを明示的に呼び出すための構文を必要とするので、任意の型に対するデストラクターが存在するかどうかを把握しなくても、コードの書き込みができます。

関連情報

- 229 ページの『第 12 章 クラスのメンバーおよびフレンド (C++ のみ)』
- 223 ページの『クラス名のスコープ』

ユーザー定義の型変換

ユーザー定義の型変換 を使用して、コンストラクターまたは変換関数を使用したオブジェクト変換を指定することができます。初期化指定子、関数引数、関数からの戻り値、式のオペランド、式の反復制御、選択文の変換、および明示的型変換の標準型変換の他に、ユーザー定義の型変換が暗黙的に使用されます。

ユーザー定義の型変換には次の 2 つのタイプがあります。

- 変換コンストラクター
- 変換関数

単一の値を暗黙的に変換する場合、コンパイラーは、ユーザー定義の型変換を 1 つだけ (型変換コンストラクターまたは型変換関数のどちらか) を使用することができます。次の例は、このことを示しています。

```
class A {
  int x;
public:
  operator int() { return x; };
};

class B {
  A y;
public:
  operator A() { return y; };
};

int main () {
  B b_obj;
//  int i = b_obj;
  int j = A(b_obj);
}
```

コンパイラーは、ステートメント int i = b_obj を許可しません。コンパイラーは、暗黙的に b_obj を型 A のオブジェクトに変換してから (B::operator A() を使用して)、暗黙的にそのオブジェクトを整数に変換しなければなりません (A::operator int() を使用して)。ステートメント int j = A(b_obj) は、暗黙的に b_obj を型 A のオブジェクトに変換してから、暗黙的にそのオブジェクトを整数に変換します。

ユーザー定義の型変換は、明確でなければなりません。さもないとそれらは呼び出されません。派生クラスの型変換関数は、両方の型変換関数が同じ型に変換されない限り、基底クラスにある別の型変換関数を隠しません。関数の多重定義解決は、最適な型変換関数を選択します。次の例は、このことを示しています。

```
class A {
  int a_int;
  char* a_carp;
public:
  operator int() { return a_int; }
  operator char*() { return a_carp; }
};

class B : public A {
  float b_float;
  char* b_carp;
public:
  operator float() { return b_float; }
  operator char*() { return b_carp; }
};

int main () {
  B b_obj;
//  long a = b_obj;
  char* c_p = b_obj;
}
```

コンパイラーは、ステートメント long a = b_obj を許可しません。コンパイラーは、A::operator int() または B::operator float() のどちらかを使用して、b_obj を long に変換できます。 B::operator char*() が A::operator char*() を隠すので、ステートメント char* c_p = b_obj は、B::operator char*() を使用して、b_obj を char* に変換します。

引数を持つコンストラクターを呼び出し、その引数型を受け入れるコンストラクターが定義されていない場合、標準型変換だけを使用して、その引数を、そのクラスのコンストラクターによって受け入れ可能な別の引数型に変換します。そのクラス用に定義されたコンストラクターに受け入れられる型に引数を変換するために、他のコンストラクターや変換関数を呼び出すことはありません。次の例は、このことを示しています。

```
class A {
public:
  A() { }
  A(int) { }
};

int main() {
   A a1 = 1.234;
//   A moocow = "text string";
}
```

コンパイラーは、ステートメント A a1 = 1.234 を認めます。コンパイラーは、標準型変換を使用して、1.234 を int に変換してから、暗黙的に変換コンストラクター A(int) を呼び出します。コンパイラーは、ステートメント A moocow = "text string" を許可しません。テキスト・ストリングを整数に変換するのは、標準の型変換ではありません。

関連情報

- 95 ページの『第 5 章 型変換』

変換コンストラクター

変換コンストラクター は、関数指定子 explicit を指定せずに宣言される単一パラメーター・コンストラクターです。 コンパイラーは、変換コンストラクターを使用して、オブジェクトを第 1 パラメーターの型から、変換コンストラクターのクラスの型に変換します。次の例は、このことを示しています。

```
class Y {
  int a, b;
  char* name;
public:
  Y(int i) { };
  Y(const char* n, int j = 0) { };
};

void add(Y) { };

int main() {

  // equivalent to
  // obj1 = Y(2)
  Y obj1 = 2;

  // equivalent to
  // obj2 = Y("somestring",0)
  Y obj2 = "somestring";

  // equivalent to
  // obj1 = Y(10)
  obj1 = 10;

  // equivalent to
  // add(Y(5))
  add(5);
}
```

上記の例には、次の 2 つの変換コンストラクターがあります。

- Y(int i) は、整数をクラス Y のオブジェクトに変換するために使用。
- Y(const char* n, int j = 0) は、ストリングを指すポインターを、クラス Y のオブジェクトに変換するために使用。

コンパイラーは、上記で説明したような型を、explicit キーワードを使用して宣言されたコンストラクターで暗黙的に変換しません。コンパイラーは、new 式、static_cast 式と明示キャスト、および基底とメンバーの初期化で明示的に宣言されたコンストラクターだけを使用します。次の例は、このことを示しています。

```
class A {
public:
  explicit A() { };
  explicit A(int) { };
};

int main() {
  A z;
//  A y = 1;
  A x = A(1);
  A w(1);
  A* v = new A(1);
  A u = (A)1;
  A t = static_cast<A>(1);
}
```

コンパイラーは、これが暗黙の型変換なので、ステートメント A y = 1 を許可しません。クラス A は、型変換コンストラクターを持っていません。

コピー・コンストラクターは、変換コンストラクターです。

関連情報

- 141 ページの『new 式 (C++ のみ)』
- 135 ページの『static_cast 演算子 (C++ のみ)』

明示的指定子

明示的関数指定子は、望ましくない暗黙的型変換を制御します。それは、クラス宣言内のコンストラクターの宣言にだけ使用されます。例えば、デフォルトのコンストラクターを除いて、以下のクラスのコンストラクターは、変換コンストラクターです。

```
class A
{  public:
   A();
   A(int);
   A(const char*, int = 0);
};
```

以下の宣言は、正しい宣言です。

```
A c = 1;
A d = "Venditti";
```

最初の宣言は `A c = A(1)` に等価です。

明示的 キーワードを使用してこのクラスのコンストラクターを宣言すると、前の宣言は正しくなくなります。

例えば、クラスを以下のようなクラスとして宣言する場合、

```
class A
{  public:
   explicit A();
   explicit A(int);
   explicit A(const char*, int = 0);
};
```

クラス型の値に一致する値だけを代入できます。

例えば、以下のステートメントは正しくありません。

```
    A a1;
  A a2 = A(1);
  A a3(1);
  A a4 = A("Venditti");
  A* p = new A(1);
  A a5 = (A)1;
  A a6 = static_cast<A>(1);
```

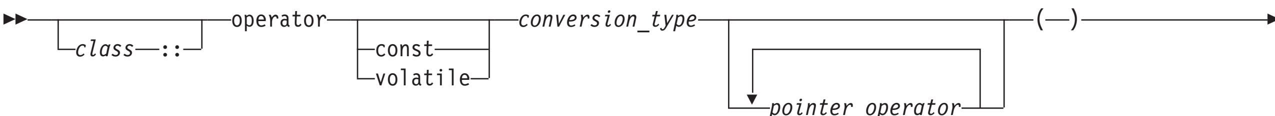
関連情報

- 290 ページの『変換コンストラクター』

変換関数

変換関数 と呼ばれる、クラスのメンバー関数を定義することができ、これは、そのクラスの型を指定された別の型に変換するものです。

変換関数の構文



```
►─┬──────────────────────────────────────────────────────────────►◄
  └─{──function_body──}─┘
```

クラス X に所属する型変換関数は、クラス型 X から *conversion_type* で指定する型に変換を指定します。次のコードは operator int() という変換関数を示しています。

```
class Y {
  int b;
public:
  operator int();
};
Y::operator int() {
  return b;
}
void f(Y obj) {
  int i = int(obj);
  int j = (int)obj;
  int k = i + obj;
}
```

関数 f(Y) にある 3 つのステートメントはすべて、型変換関数 Y::operator int() を使用します。

クラス、列挙型、typedef 名、関数型、または配列型は、*conversion_type* で宣言、または定義することはできません。型変換関数は、型 A のオブジェクトを、型 A、A の基底クラス、または void に変換するためには使用できません。

変換関数には引数がなく、戻りの型が暗黙的に変換の型になります。変換関数は継承することができます。仮想変換関数は使用できますが、静的変換関数は使用できません。

関連情報

- 95 ページの『第 5 章 型変換』
- 289 ページの『ユーザー定義の型変換』
- 290 ページの『変換コンストラクター』
- 95 ページの『第 5 章 型変換』

コピー・コンストラクター

コピー・コンストラクター により、初期化をすることで、既存オブジェクトから新規オブジェクトを作成できます。クラス A のコピー・コンストラクターは、第 1 パラメーターが、型 A&、const A&、volatile A&、または const volatile A& である非テンプレート・コンストラクターであり、そのパラメーターの残りは (他にあれば)、デフォルト値を持っています。

クラス A に対してコピー・コンストラクターを宣言しない場合、コンパイラーは、コピー・コンストラクターを暗黙的に宣言し、それはインライン・パブリック・メンバーとなります。

次の例は、暗黙的に定義されたコピー・コンストラクターと、暗黙的なユーザー定義のコピー・コンストラクターを示しています。

```
#include <iostream>
using namespace std;

struct A {
  int i;
  A() : i(10) { }
};
```

```
struct B {
  int j;
  B() : j(20) {
    cout << "Constructor B(), j = " << j << endl;
  }

  B(B& arg) : j(arg.j) {
    cout << "Copy constructor B(B&), j = " << j << endl;
  }

  B(const B&, int val = 30) : j(val) {
    cout << "Copy constructor B(const B&, int), j = " << j << endl;
  }
};

struct C {
  C() { }
  C(C&) { }
};

int main() {
  A a;
  A a1(a);
  B b;
  const B b_const;
  B b1(b);
  B b2(b_const);
  const C c_const;
//  C c1(c_const);
}
```

上記の例の出力は、以下のとおりです。

```
Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30
```

ステートメント A a1(a) は、暗黙的定義のコピー・コンストラクターを使用して、a から新規オブジェクトを作成します。ステートメント B b1(b) は、ユーザー定義のコピー・コンストラクター B::B(B&) を使用して、b から新規オブジェクトを作成します。ステートメント B b2(b_const) は、コピー・コンストラクター B::B(const B&, int) を使用して、新規オブジェクトを作成します。コンパイラーは、第 1 パラメーターとして型 const C& のオブジェクトを取得するコピー・コンストラクターが定義されていないので、ステートメント C c1(c_const) を許可しません。

次のことが真の場合、暗黙的に宣言されたクラス A のコピー・コンストラクターは、A::A(const A&) の書式を持ちます。

- A の直接基底および仮想基底は、第 1 パラメーターが、const または const volatile で修飾されたコピー・コンストラクターを持っている。
- A の非静的クラス型、またはクラス型データ・メンバーの配列は、第 1 パラメーターが、const または const volatile で修飾されたコピー・コンストラクターを持っている。

クラス A に対して上記のことが真でない場合、コンパイラーは、A::A(A&) の書式のコピー・コンストラクターを暗黙的に宣言します。

コンパイラーは、コンパイラーがクラス A のコピー・コンストラクターを暗黙的に定義する必要があり、次の中で 1 つまたは複数が真になるプログラムは、許可しません。

- クラス A が、非静的データ・メンバーを持ち、その型が、アクセス不能またはあいまいなコピー・コンストラクターを持っている。

- クラス A は、アクセス不能またはあいまいなコピー・コンストラクターを持つクラスから派生している。

型 A のオブジェクト、またはクラス A から派生したオブジェクトを初期化する場合、コンパイラーは、暗黙的に宣言されたクラス A のコンストラクターを暗黙的に定義します。

暗黙的に定義されたコピー・コンストラクターは、コンストラクターがオブジェクトの基底およびメンバーを初期化する順序と同じ順序で、オブジェクトの基底およびメンバーをコピーします。

関連情報

- 277 ページの『コンストラクターとデストラクターの概要』

コピー代入演算子

コピー代入演算子 により、初期化をすることで、既存オブジェクトから新規オブジェクトを作成できます。クラス A のコピー代入演算子は、次の書式のいずれかを持つ非静的、非テンプレートのメンバー関数です。

- `A::operator=(A)`
- `A::operator=(A&)`
- `A::operator=(const A&)`
- `A::operator=(volatile A&)`
- `A::operator=(const volatile A&)`

クラス A に対してコピー代入演算子を宣言しない場合、コンパイラーは、コピー代入演算子を暗黙的に宣言し、それはインライン・パブリックとなります。

次の例は、暗黙的に定義された代入演算子と、暗黙的なユーザー定義の代入演算子を示しています。

```
#include <iostream>
using namespace std;

struct A {
  A& operator=(const A&) {
    cout << "A::operator=(const A&)" << endl;
    return *this;
  }

  A& operator=(A&) {
    cout << "A::operator=(A&)" << endl;
    return *this;
  }

};
class B {
  A a;
};

struct C {
  C& operator=(C&) {
    cout << "C::operator=(C&)" << endl;
    return *this;
  }
  C() { }
};

int main() {
  B x, y;
```

```
  x = y;

  A w, z;
  w = z;

  C i;
  const C j();
//  i = j;
}
```

上記の例の出力は、以下のとおりです。

```
A::operator=(const A&)
A::operator=(A&)
```

代入 x = y は、暗黙的に定義された B のコピー代入演算子を呼び出します。つまり、ユーザー定義のコピー代入演算子 A::operator=(const A&) を呼び出します。代入 w = z は、ユーザー定義の演算子 A::operator=(A&) を呼び出します。コンパイラーは、演算子 C::operator=(const C&) が定義されていないので、代入 i = j を許可しません。

次のことが真の場合、暗黙的に宣言されたクラス A のコピー代入演算子は、 A& A::operator=(const A&) の書式を持ちます。

- クラス A の直接または仮想基底 B が、パラメーターが、型 const B&、const volatile B&、または B であるコピー代入演算子を持っている。
- クラス A に属する、型 X の非静的クラス型データ・メンバーが、パラメーターが、型 const X&、const volatile X&、または X であるコピー・コンストラクターを持っている。

クラス A に対して上記のことが真でない場合、コンパイラーは、A& A::operator=(A&) の書式を使用したコピー代入演算子を暗黙的に宣言します。

暗黙的に宣言されたコピー代入演算子は、演算子の引数への参照を戻します。

派生クラスのコピー代入演算子は、その基底クラスのコピー代入演算子を隠します。

コンパイラーは、クラス A に対してコピー代入演算子を暗黙的に定義しなければならず、次のなかで 1 つまたは複数が真になっているようなプログラムは、許可しません。

- クラス A は、const 型、または参照型の非静的データ・メンバーを持つ。
- クラス A は、型が非静的データ・メンバーで、アクセス不能のコピー代入演算子を持つ。
- クラス A は、アクセス不能なコピー代入演算子を使用した基底クラスから派生する。

暗黙的に定義されたクラス A のコピー代入演算子は、まず最初に、A の定義にそれらが現れる順序で、A の直接基底クラスを割り当てます。次に、暗黙的に定義されるコピー代入演算子は、A の定義でそれらが宣言されている順序で、A の非静的データ・メンバーを割り当てます。

関連情報

- 119 ページの『代入演算子』

第 15 章 テンプレート (C++ のみ)

テンプレート では、関連するクラスのセット、または関連する関数のセットについて記述し、その宣言のパラメーターのリストでは、そのセットのメンバーが、どのように異なるかを記述します。これらのパラメーターに引数を提供すると、コンパイラーは、新規のクラスまたは関数を生成します。このプロセスは、テンプレートのインスタンス化 と呼ばれ、 316 ページの『テンプレートのインスタンス化』 で詳しく説明しています。テンプレート、およびテンプレート・パラメーターのセットから生成されたこのクラスや関数定義は、 319 ページの『テンプレートの特殊化』 に示すように *特殊化* と呼ばれます。

400 IBM i 固有の使用法については、「*ILE C/C++ プログラマーの手引き*」の第 28 章『C++ プログラム内でのテンプレートの使用』を参照してください。

テンプレート宣言の構文

```
>>-+--------+--template--<--template_parameter_list-->--declaration-><
   '-export-'
```

コンパイラーは、テンプレートで export キーワードを受諾し、暗黙に無視します。

template_parameter_list は、テンプレート・パラメーターのコンマで分離したリストで、 298 ページの『テンプレート・パラメーター』 で説明しています。

declaration は、次のいずれかです。

- 関数またはクラスの宣言または定義
- メンバー関数またはクラス・テンプレートのメンバー・クラスの定義
- クラス・テンプレートの静的データ・メンバーの定義
- クラス・テンプレート内のネスト・クラスの静的データ・メンバーの定義
- クラスまたはクラス・テンプレートのメンバー・テンプレートの定義

型 の ID が、テンプレート宣言のスコープ内の *type_name* であると定義されます。テンプレート宣言は、ネーム・スペース・スコープ宣言またはクラス・スコープ宣言として現れます。

次の例は、クラス・テンプレートの使用法を示しています。

```
template<class T> class Key
{
    T k;
    T* kptr;
    int length;
public:
    Key(T);
    // ...
};
```

その後、次の宣言が現れるとします。

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

© Copyright IBM Corp. 1998, 2010

コンパイラーは class Key の 3 つのインスタンスを作成します。次の表は、3 つのクラス・インスタンスが、ソース・コードの書式で、テンプレートとしてではなく通常のクラスとして書き出された場合の、それらオブジェクトの定義を示しています。

class Key<int> i;	**class Key<char*> c;**	**class Key<mytype> m;**

class Key
{
      int k;
      int * kptr;
      int length;
public:
      Key(int);
      // ...
};
	class Key
	{
	      char* k;
	      char** kptr;
	      int length;
	public:
	      Key(char*);
	      // ...
	};
		class Key
		{
		      mytype k;
		      mytype* kptr;
		      int length;
		public:
		      Key(mytype);
		      // ...
		};

これらの 3 つのクラスには、それぞれ名前があることに注意してください。不等号中括弧の中に含まれている引数は、単にクラス名に対する引数ではなく、クラス名自体の一部です。 Key<int> と Key<char*> は、クラス名です。

テンプレート・パラメーター

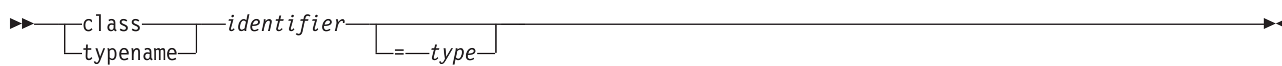
テンプレート・パラメーターには、下記の 3 種類があります。

- 型テンプレート・パラメーター
- 「非型」テンプレート・パラメーター
- 「テンプレート」テンプレート・パラメーター

テンプレート・パラメーター宣言で、キーワード class と typename は交換できます。テンプレート・パラメーター宣言内では、ストレージ・クラス指定子 (static および auto) は使用できません。

型テンプレート・パラメーター

型テンプレート・パラメーター宣言の構文



identifier は、型の名前です。

関連情報

- 327 ページの『型名キーワード』

「非型」テンプレート・パラメーター

「非型」テンプレート・パラメーターの構文は、次のいずれかの型の宣言と同じです。

- 整数または列挙型
- オブジェクトへのポインターまたは関数を指すポインター
- オブジェクトへの参照または関数への参照
- メンバーへのポインター

配列、または関数として宣言された「非型」テンプレート・パラメーターは、ポインター、または関数を指すポインターにそれぞれ変換されます。次の例は、このことを示しています。

```
template<int a[4]> struct A { };
template<int f(int)> struct B { };

int i;
int g(int) { return 0;}

A<&i> x;
B<&g> y;
```

&i から推定される型は int * で、 &g から推定される型は int (*)(int) です。

「非型」テンプレート・パラメーターを const または volatile で修飾できます。

「非型」テンプレート・パラメーターを、浮動小数点、クラス、または void 型として宣言することはできません。

「非型」テンプレート・パラメーターは、左辺値ではありません。

関連情報

- 62 ページの『型修飾子』
- 103 ページの『左辺値と右辺値』

「テンプレート」テンプレート・パラメーター

「テンプレート」テンプレート・パラメーター宣言の構文

```
►►—template—<—template-parameter-list—>—class—┬────────────┬──┬─────────────────┬──────────►◄
                                              └─identifier─┘  └─=—id-expression─┘
```

次の例は、「テンプレート」テンプレート・パラメーターの宣言と使い方を示しています。

```
template<template <class T> class X> class A { };
template<class T> class B { };

A<B> a;
```

関連情報

- 298 ページの『テンプレート・パラメーター』

テンプレート・パラメーターのデフォルト引数

テンプレート・パラメーターは、デフォルトの引数を持つことができます。デフォルトのテンプレート引数のセットは、任意のテンプレートの宣言すべてに累積していきます。次の例は、このことを示しています。

```
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
   public:
      T x;
      U y;
};

A<> a;
```

メンバー a.x の型は float で、a.y の型は int です。

デフォルトの引数を、同じスコープ内の異なる宣言にある、同じテンプレート・パラメーターに与えることはできません。例えば、コンパイラーは次のことを許可しません。

```
template<class T = char> class X;
template<class T = char> class X { };
```

あるテンプレート・パラメーターが、デフォルトの引数を持つ場合、それに続くテンプレート・パラメーターも、すべてデフォルトの引数を持つはずです。例えば、コンパイラーは次のコードを許可しません。

```
template<class T = char, class U, class V = int> class X { };
```

テンプレート・パラメーター U は、デフォルトの引数が必要です。あるいは T のデフォルトを除去する必要があります。

テンプレート・パラメーターのスコープは、その宣言のポイントからそのテンプレート定義の終了までです。つまり、他のテンプレート・パラメーター宣言内のテンプレート・パラメーターの名前、およびそれらのデフォルトの引数を使用できるということです。次の例は、このことを示しています。

```
template<class T = int> class A;
template<class T = float> class B;
template<class V, V obj> class C;
// a template parameter (T) used as the default argument
// to another template parameter (U)
template<class T, class U = T> class D { };
```

関連情報

- 298 ページの『テンプレート・パラメーター』

テンプレート引数

下記の 3 つのテンプレート・パラメーターの型に対応する、3 種類のテンプレート引数があります。

- テンプレート型引数
- テンプレート非型引数
- 「テンプレート」テンプレート引数

テンプレート引数は、テンプレートに宣言された対応パラメーターが指定する型、およびフォームと一致しなければなりません。

テンプレート・パラメーターのデフォルト値を使用するには、対応するテンプレート引数を省略します。しかし、たとえすべてのテンプレート・パラメーターがデフォルトを持っていても、<> 大括弧を使用する必要があります。例えば、次は構文エラーが発生します。

```
template<class T = int> class X { };
X<> a;
X b;
```

最後の宣言 X b は、エラーになります。

テンプレート型引数

次のいずれかを、「型」テンプレート・パラメーターのテンプレート引数として使用することはできません。

- ローカル型
- リンケージなしの型
- 無名型

- 上記の型のいずれかを複合した型

テンプレート引数が、型なのか、式なのかあいまいな場合は、テンプレート引数は、型であると見なされます。次の例は、このことを示しています。

```
template<class T> void f() { };
template<int i> void f() { };

int main() {
  f<int()>();
}
```

関数呼び出し `f<int()>()` は、T をテンプレート引数として、関数を呼び出します。このときコンパイラーは、`int()` を型として扱い、したがって暗黙的にインスタンスを作成し、最初の `f()` を呼び出します。

関連情報

- 2 ページの『ブロック/ローカル・スコープ』
- 9 ページの『リンケージなし』
- 53 ページの『ビット・フィールド・メンバー』
- 60 ページの『typedef 定義』

テンプレート非型引数

テンプレート引数リストに指定されている「非型」テンプレート引数は、コンパイル時に値が決められる式です。このような引数は、定数式、関数のアドレス、外部リンケージのあるオブジェクト、または静的クラス・メンバーのアドレスでなければなりません。通常は、「非型」テンプレート引数を使用して、クラスの初期化またはクラス・メンバーのサイズを指定します。

非型整数引数の場合、インスタンス引数は、そのパラメーター型に適した値と符号がある限りは、対応するテンプレート・パラメーターと一致します。

非型アドレス引数の場合、インスタンス引数の型は、*identifier* または *&identifier* の形式でなければなりません。また、インスタンス引数の型は、マッチングの前に、関数名が関数型を指すポインターに変更される点以外は、正確にテンプレート・パラメーターと一致していなければなりません。

テンプレート引数リスト内に「非型」テンプレート引数がある場合、結果として得られる値は、そのテンプレート・クラス型の一部を形成します。 2 つのテンプレート・クラス名が同じテンプレート名を持っており、それらの引数の値が同じ場合、それらは同じクラスであるといいます。

次の例では、クラス・テンプレートが、型引数だけでなく、「非型」テンプレート `int` 引数も必要であると定義されています。

```
template<class T, int size> class Myfilebuf
{
      T* filepos;
      static int array[size];
public:
      Myfilebuf() { /* ... */ }
      ~Myfilebuf();
      advance(); // function defined elsewhere in program
};
```

この例では、テンプレート引数 `size` が、テンプレート・クラス名の一部になります。このようなテンプレート・クラスのオブジェクトは、クラス型引数 `T` と「非型」テンプレート引数 `size` の値の両方を指定して作成されます。

オブジェクト x および引数 double と size=200 を持つ、その対応するテンプレート・クラスは、その 2 番目のテンプレート引数として値を持ったこのテンプレートから作成することができます。

```
Myfilebuf<double,200> x;
```

x も、演算式を使用して作成できます。

```
Myfilebuf<double,10*20> x;
```

これらの式によって作成されるオブジェクトは、テンプレート引数の評価が同じになるので、同一になります。最初の式の中の値 200 は、2 番目の構成に示すように、コンパイル時の結果が 200 に等しいということがわかっている式によって表すこともできます。

注: < 記号または > 記号を含む引数は、テンプレート引数リスト区切り文字が関係演算子として実際に使用されているときにどちらの記号もテンプレート引数リスト区切り文字として解析されないようにするため、括弧で囲む必要があります。例えば、次の定義の中の引数は有効です。

```
Myfilebuf<double, (75>25)> x;       // valid
```

ただし次の定義では、より大の演算子 (>) がテンプレート引数リストの終了区切り文字と解釈されるので、有効ではありません。

```
Myfilebuf<double, 75>25> x;         // error
```

テンプレート引数の評価結果が同一でなければ、作成されたオブジェクトは異なる型になります。

```
Myfilebuf<double,200> x;            // create object x of class
                                    // Myfilebuf<double,200>
Myfilebuf<double,200.0> y;          // error, 200.0 is a double,
                                    // not an int
```

y のインスタンス化は、値 200.0 の型が double で、テンプレート引数の型が int であるために失敗します。

次の 2 つのオブジェクトは、

```
Myfilebuf<double, 128> x
Myfilebuf<double, 512> y
```

分離テンプレート特殊化のオブジェクトです。後でこれらのオブジェクトのいずれかを Myfilebuf<double> で参照するとエラーになります。

クラス・テンプレートは、非型引数を持つ場合は、型引数を持つ必要がありません。例えば、次のテンプレートは有効なクラス・テンプレートです。

```
template<int i> class C
{
public:
            int k;
            C() { k = i; }
};
```

このクラス・テンプレートは、次のような宣言でインスタンスを生成できます。

```
class C<100>;
class C<200>;
```

繰り返しですが、これら 2 つの宣言は、これらの非型引数の値が異なるので、別個のクラスを参照しています。

関連情報

- 105 ページの『整数定数式』
- 80 ページの『参照 (C++ のみ)』
- 8 ページの『外部結合』
- 239 ページの『静的メンバー』

「テンプレート」テンプレート引数

「テンプレート」テンプレート・パラメーターのテンプレート引数は、クラス・テンプレートの名前です。

コンパイラーが「テンプレート」テンプレート引数と一致するテンプレートの検索を試みる場合、それは主クラス・テンプレートだけを検索します。 (主テンプレート は、特殊化しようとしているテンプレートのことです。) コンパイラーは、たとえそれらのパラメーター・リストが、「テンプレート」テンプレート・パラメーターのリストと一致していても、部分的な特殊化は考慮に入れません。例えば、コンパイラーは次のコードを許可しません。

```
template<class T, int i> class A {
   int x;
};

template<class T> class A<T, 5> {
   short x;
};

template<template<class T> class U> class B1 { };

B1<A> c;
```

コンパイラーは、宣言 `B1<A> c` を許可しません。`A` の部分的な特殊化は、`B1` の「テンプレート」テンプレート・パラメーター `U` と一致しているように見えますが、コンパイラーは、`U` とは異なるテンプレート・パラメーターを持つ、主テンプレート `A` だけを考慮します。

「テンプレート」テンプレート・パラメーターを基にした特殊化のインスタンスをいったん作成すると、コンパイラーは、それに対応する「テンプレート」テンプレート引数に基づく部分的な特殊化を考慮します。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

template<class T, class U> class A {
   int x;
};

template<class U> class A<int, U> {
   short x;
};

template<template<class T, class U> class V> class B {
   V<int, char> i;
   V<char, char> j;
};

B<A> c;

int main() {
   cout << typeid(c.i.x).name() << endl;
   cout << typeid(c.j.x).name() << endl;
}
```

上記の例の出力は、以下のとおりです。

```
short
int
```

宣言 `V<int, char> i` は、部分的な特殊化を使用しますが、宣言 `V<char, char> j` は、主テンプレートを使用します。

関連情報

- 324 ページの『部分的特殊化』
- 316 ページの『テンプレートのインスタンス化』

クラス・テンプレート

クラス・テンプレートと個々のクラスとの間の関係は、クラスと個々のオブジェクトとの間の関係に似ています。個々のクラスがオブジェクトのグループの構成方法を定義し、一方、クラス・テンプレートがクラスのグループの生成方法を定義します。

クラス・テンプレート とテンプレート・クラス という用語の間の区別に注意してください。

クラス・テンプレート
: これは、テンプレート・クラスの生成に使用されるテンプレートです。クラス・テンプレートのオブジェクトは、宣言できません。

テンプレート・クラス
: クラス・テンプレートのインスタンスです。

テンプレート定義は、下記の点を除いて、テンプレートが生成し得る有効なクラス定義のいずれとも同一です。

- クラス・テンプレート定義には、次の語が先行します。

  `template<` *`template-parameter-list`* `>`

  ここで、*template-parameter-list* は、次に示す種類のテンプレート・パラメーターの 1 つまたは複数を、コンマで分離したリストです。
  - 型
  - 非型
  - template
- クラス・テンプレート内の型、変数、定数およびオブジェクトを、テンプレート・パラメーターおよび明示型 (例えば、`int` や `char`) を使用して宣言することができます。

詳述型指定子を使用して定義しなくても、クラス・テンプレートを宣言することができます。次に例を示します。

```
template<class L, class T> class Key;
```

これにより、名前がクラス・テンプレート名として予約されます。クラス・テンプレートのテンプレート宣言は、すべてが、同じ型と同じ数のテンプレート引数を持っていなければなりません。クラス定義を含む 1 つのテンプレート宣言だけが許可されます。

注: テンプレート引数リストがネストされている場合は、内側のリストの終わりの `>` と外側のリストの終わりの `>` の間に分離スペースが必要です。これがなければ、抽出演算子 `>>` と 2 つのテンプレート・リスト区切り文字 `>` との間があいまいになります。

通常のクラス・メンバーのオブジェクトや関数のアクセスに使用されるどの手法でも、個々のテンプレート・クラスのオブジェクトや関数メンバーをアクセスすることができます。次のクラス・テンプレートがあるとします。

```
template<class T> class Vehicle
{
public:
    Vehicle() { /* ... */ }    // constructor
    ~Vehicle() {};             // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

そして、次の宣言を行います。

```
Vehicle<char> bicycle; // instantiates the template
```

コンストラクター、構成オブジェクト、およびメンバー関数 `drive()` は、次のいずれかを指定してアクセスできます (標準ヘッダー・ファイル `string.h` が、プログラム・ファイルに含まれているとします)。

コンストラクター	`Vehicle<char> bicycle;`
	
	`// constructor called automatically,`
	`// object bicycle created`

オブジェクト `bicycle`	`strcpy (bicycle.kind, "10 speed");` `bicycle.kind[0] = '2';`
関数 `drive()`	`char* n = bicycle.drive();`
関数 `roadmap()`	`Vehicle<char>::roadmap();`

関連情報

- 220 ページの『クラス型の宣言』
- 223 ページの『クラス名のスコープ』

クラス・テンプレートの宣言と定義

クラス・テンプレートを宣言してから、対応するテンプレート・クラスのインスタンス化を行う必要があります。クラス・テンプレートの定義は、1 つの変換単位内で 1 回しか使用できません。クラス・テンプレートは、クラスのサイズを必要とする、またはクラスのメンバーを参照する、テンプレート・クラスを使用する前に定義する必要があります。

次の例では、クラス・テンプレート `Key` は、定義の前に宣言されます。クラスのサイズは必要ないので、ポインター `keyiptr` の宣言は有効です。ただし、`keyi` の宣言はエラーになります。

```
template  <class L> class Key;      // class template declared,
                                    // not defined yet
                                    //
class Key<int> *keyiptr;            // declaration of pointer
                                    //
class Key<int> keyi;                // error, cannot declare keyi
                                    // without knowing size
                                    //
template <class L> class Key        // now class template defined
{ /* ... */ };
```

クラス・テンプレートを定義する前に、対応するテンプレート・クラスを使用すると、コンパイラーはエラーを発行します。テンプレート・クラス名の形式をもつクラス名は、テンプレート・クラスであると見なされます。言い換えれば、テンプレート・クラスの場合は、不等号括弧が有効なのは、クラス名の中だけです。

前の例では、詳述型指定子 `class` を使用してクラス・テンプレート `key` とポインター `keyiptr` を宣言しています。 `keyiptr` の宣言は詳述型指定子なしでも行うことができます。

```
template  <class L> class Key;           // class template declared,
                                   // not defined yet
                                         //
Key<int> *keyiptr;                       // declaration of pointer
                                         //
Key<int> keyi;                           // error, cannot declare keyi
                                   // without knowing size
                                         //
template <class L> class Key             // now class template defined
{ /* ... */ };
```

関連情報

- 304 ページの『クラス・テンプレート』

静的データ・メンバーとテンプレート

どのクラス・テンプレートのインスタンス化も、静的データ・メンバーの専用コピーを所有します。静的宣言は、テンプレート引数型または任意の定義された型です。

別々に静的メンバーを定義する必要があります。次の例は、このことを示しています。

```
template <class T> class K
{
public:
      static T x;
};
template <class T> T K<T> ::x;

int main()
{
      K<int>::x = 0;
}
```

ステートメント `template T K::x` は、クラス `K` の静的メンバーを定義しますが、`main()` 関数のステートメントは、`K <int>` のデータ・メンバーに値を代入します。

関連情報

- 239 ページの『静的メンバー』

クラス・テンプレートのメンバー関数

テンプレートのメンバー関数を、そのクラス・テンプレート定義の外側に定義できます。

クラス・テンプレート特殊化のメンバー関数を呼び出す場合、コンパイラーは、以前、クラス・テンプレートの作成に使用したテンプレート引数を使用します。次の例は、このことを示しています。

```
template<class T> class X {
   public:
      T operator+(T);
};

template<class T> T X<T>::operator+(T arg1) {
```

```
   return arg1;
};

int main() {
   X<char> a;
   X<int> b;
   a +'z';
   b + 4;
}
```

多重定義された加法演算子は、クラス X の外側で定義されています。ステートメント a + 'z' は、a.operator+('z') と同等です。ステートメント b + 4 は、b.operator+(4) と同等です。

関連情報

- 231 ページの『メンバー関数』

フレンドとテンプレート

テンプレートを含める場合、クラスとそれらのフレンドとの間には 4 種類の関係があります。

- *1 対多*: 非テンプレート関数は、すべてのテンプレート・クラスのインスタンス生成へのフレンドです。
- *多対 1*: テンプレート関数のすべてのインスタンス生成は、通常の非テンプレート・クラスへのフレンドです。
- *1 対 1*: テンプレート引数の 1 セットを使用したテンプレート関数のインスタンス生成は、同じテンプレート引数のセットを使用してインスタンス生成された 1 つのテンプレート・クラスのフレンドです。これは、通常の非テンプレート・クラスと通常の非テンプレート・フレンド関数との間の関係でもあります。
- *多対多*: テンプレート関数のすべてのインスタンス生成は、テンプレート・クラスのすべてのインスタンス生成へのフレンドです。

次の例は、これらの関係を示しています。

```
class B{
   template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<class U> friend int h();
};
```

- 関数 e() は、クラス A と 1 対多の関係を持ちます。関数 e() は、クラス A のすべてのインスタンス生成のフレンドです。
- 関数 f() は、クラス A と 1 対 1 の関係を持ちます。コンパイラーは、この種の宣言に対して、以下に類似する警告を出します。

  ```
  The friend function declaration "f" will cause an error when the enclosing
  template class is instantiated with arguments that declare a friend function
  that does not match an existing definition. The function declares only one
  function because it is not a template but the function type depends on
  one or more template parameters.
  ```

- 関数 g() は、クラス A と 1 対 1 の関係を持ちます。関数 g() は、関数テンプレートです。この前に宣言する必要があります。そうしないと、コンパイラーは g<T> をテンプレート名として認識しません。 A のインスタンス生成ごとに、g() にマッチングするインスタンス生成が 1 つあります。例えば、g<int> は、A<int> のフレンドです。

- 関数 h() は、クラス A と多対多の関係を持ちます。関数 h() は、関数テンプレートです。 A のすべてのインスタンス生成にとって、h() のインスタンス生成は、すべてフレンドです。
- 関数 j() は、クラス B と多対 1 の関係を持ちます。

これらの関係は、フレンド・クラスにも適用します。

関連情報

- 245 ページの『フレンド』

関数テンプレート

関数テンプレート は、関数のグループの生成方法を定義します。

非テンプレート関数は、テンプレートから生成された特殊化の関数と同じ名前、およびパラメーター・プロファイルを持っていたとしても、非テンプレート関数は、関数テンプレートとは関連がありません。非テンプレート関数は、関数テンプレートの特殊化と見なされることは、ありません。

次の例は、quicksort という名前の関数テンプレートを使用した、quicksort アルゴリズムをインプリメントします。

```
#include <iostream>
#include <cstdlib>
using namespace std;

template<class T> void quicksort(T a[], const int& leftarg, const int& rightarg)
{
  if (leftarg < rightarg) {

    T pivotvalue = a[leftarg];
    int left = leftarg - 1;
    int right = rightarg + 1;

  for(;;) {

    while (a[--right] > pivotvalue);
    while (a[++left] < pivotvalue);

    if (left >= right) break;

    T temp = a[right];
    a[right] = a[left];
    a[left] = temp;
  }

  int pivot = right;
  quicksort(a, leftarg, pivot);
  quicksort(a, pivot + 1, rightarg);
  }
}

int main(void) {
  int sortme[10];

  for (int i = 0; i < 10; i++) {
    sortme[i] = rand();
    cout << sortme[i] << " ";
  };
  cout << endl;

  quicksort<int>(sortme, 0, 10 - 1);

  for (int i = 0; i < 10; i++) cout << sortme[i] << "
```

```
  ";
  cout << endl;
  return 0;
}
```

上記の例は、次に類似する出力を行います。

```
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
4086 5627 5758 7419 10113 16212 16838 17515 23010 31051
```

quickSort アルゴリズムは、型 T の配列 (これの関係演算子および代入演算子は、定義されている) をソートします。テンプレート関数は、1 つのテンプレート引数と 3 つの関数引数を取ります。

- ソートされる配列の型、T
- ソートされる配列の名前、a
- 配列の下限、leftarg
- 配列の上限、rightarg

上記の例では、次のステートメントを使用して、quicksort() テンプレート関数を呼び出すこともできます。

```
  quicksort(sortme, 0, 10 - 1);
```

コンパイラーが、テンプレート関数呼び出しの使い方とコンテキストにより、テンプレート引数を推定できる場合合、テンプレート引数を省略できます。ここでは、コンパイラーは、sortme が、型 int の配列であると推定します。

テンプレート引数の推定

テンプレート関数を呼び出す場合、テンプレート関数呼び出しの使い方とそのコンテキストによって、コンパイラーが決定または*推定* できるテンプレート引数は、どれでも省略できます。

コンパイラーは、対応するテンプレート・パラメーターの型と、関数呼び出しで使用される引数の型を比較することにより、テンプレート引数を推定しようとします。テンプレート引数の推定を行うためには、コンパイラーが比較する 2 つの型 (テンプレート・パラメーターと関数呼び出しで使用される引数) は、ある特定の構造体でなければなりません。下記に、これらの型構造体をリストします。

```
T
const T
volatile T
T&
T*
T[10]
A<T>
C(*)(T)
T(*)()
T(*)(U)
T C::*
C T::*
T U::*
T (C::*)()
C (T::*)()
D (C::*)(T)
C (T::*)(U)
T (C::*)(U)
T (U::*)()
T (U::*)(V)
E[10][i]
```

```
B<i>
TT<T>
TT<i>
TT<C>
```

- T、U、および V は、テンプレート型引数を表す
- 10 は、整数定数を示す
- i は、テンプレート非型引数を示す
- [i] は、参照またはポインター型の配列境界、あるいは標準配列の非主配列の境界を示す
- TT は、「テンプレート」テンプレート引数を示す
- (T)、(U)、および (V) は、少なくとも 1 つのテンプレート型引数を持つ引数リストを示す
- () は、テンプレート引数を持っていない引数リストを示す
- <T> は、少なくとも 1 つのテンプレート型引数を持つテンプレート引数リストを示す
- <i> は、少なくとも 1 つのテンプレート非型引数を持つテンプレート引数リストを示す
- <C> は、テンプレート・パラメーターに従属するテンプレート引数を持っていない、テンプレート引数リストを示す

次の例は、これら型構造体のそれぞれの使用法を示しています。例では、引数として上記の各構造体を使用して、テンプレート関数を宣言しています。そして、これらの関数が、宣言順に (テンプレート引数を使用せずに) 呼び出されます。この例は、型構造体のリストと同様なものを出力します。

```
#include <iostream>
using namespace std;

template<class T> class A { };
template<int i> class B { };

class C {
   public:
      int x;
};

class D {
   public:
      C y;
      int z;
};

template<class T> void f (T)          { cout << "T" << endl; };
template<class T> void f1(const T)    { cout << "const T" << endl; };
template<class T> void f2(volatile T) { cout << "volatile T" << endl;  };
template<class T> void g (T*)         { cout << "T*" << endl; };
template<class T> void g (T&)         { cout << "T&" << endl; };
template<class T> void g1(T[10])      { cout << "T[10]" << endl;};
template<class T> void h1(A<T>)       { cout << "A<T>" << endl; };

void test_1() {
   A<char> a;
   C c;

   f(c);   f1(c);   f2(c);
   g(c);   g(&c);   g1(&c);
   h1(a);
}

template<class T>          void j(C(*)(T)) { cout << "C(*) (T)" << endl; };
template<class T>          void j(T(*)())  { cout << "T(*) ()" << endl; }
template<class T, class U> void j(T(*)(U)) { cout << "T(*) (U)" << endl; };

void test_2() {
```

```
   C (*c_pfunct1)(int);
   C (*c_pfunct2)(void);
   int (*c_pfunct3)(int);
   j(c_pfunct1);
   j(c_pfunct2);
   j(c_pfunct3);
}

template<class T>          void k(T C::*) { cout << "T C::*" << endl; };
template<class T>          void k(C T::*) { cout << "C T::*" << endl; };
template<class T, class U> void k(T U::*) { cout << "T U::*" << endl; };

void test_3() {
   k(&C::x);
   k(&D::y);
   k(&D::z);
}

template<class T>     void m(T (C::*)() )
   { cout << "T (C::*)()" << endl; };
template<class T>     void m(C (T::*)() )
   { cout << "C (T::*)()" << endl; };
template<class T>     void m(D (C::*)(T))
   { cout << "D (C::*)(T)" << endl; };
template<class T, class U>  void m(C (T::*)(U))
   { cout << "C (T::*)(U)" << endl; };
template<class T, class U>  void m(T (C::*)(U))
   { cout << "T (C::*)(U)" << endl; };
template<class T, class U>  void m(T (U::*)() )
   { cout << "T (U::*)()" << endl; };
template<class T, class U, class V> void m(T (U::*)(V))
   { cout << "T (U::*)(V)" << endl; };

void test_4() {
   int (C::*f_membp1)(void);
   C (D::*f_membp2)(void);
   D (C::*f_membp3)(int);
   m(f_membp1);
   m(f_membp2);
   m(f_membp3);

   C (D::*f_membp4)(int);
   int (C::*f_membp5)(int);
   int (D::*f_membp6)(void);
   m(f_membp4);
   m(f_membp5);
   m(f_membp6);

   int (D::*f_membp7)(int);
   m(f_membp7);
}

template<int i> void n(C[10][i]) { cout << "E[10][i]" << endl; };
template<int i> void n(B<i>)     { cout << "B<i>" << endl; };

void test_5() {
   C array[10][20];
   n(array);
   B<20> b;
   n(b);
}

template<template<class> class TT, class T> void p1(TT<T>)
   { cout << "TT<T>" << endl; };
template<template<int> class TT, int i>     void p2(TT<i>)
   { cout << "TT<i>" << endl; };
template<template<class> class TT>          void p3(TT<C>)
```

```
   { cout << "TT<C>" << endl; };

void test_6() {
   A<char> a;
   B<20> b;
   A<C> c;
   p1(a);
   p2(b);
   p3(c);
}

int main() { test_1(); test_2(); test_3(); test_4(); test_5(); test_6(); }
```

「型」テンプレート引数の推定

コンパイラーは、リストされたいくつかの型構造体で構成される型から、テンプレート引数を推定できます。次の例は、いくつかの型構造体で構成される型からのテンプレート引数の推定を示しています。

```
template<class T> class Y { };

template<class T, int i> class X {
   public:
      Y<T> f(char[20][i]) { return x; };
      Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
   void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
   Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
   g(p);
}
```

型 Y<int> (X<int, 20>;::*p)(char[20][20]) T<U> (V::*)(W[20][i]) は、型構造体 T (U::*)(V) に基づいています。

- T は、Y<int> です
- U は、X<int, 20> です。
- V は、char[20][20] です

型が属するクラスを使用してその型を修飾し、そのクラス (*ネストされた名前指定子*) がテンプレート・パラメーターに依存する場合、コンパイラーは、そのパラメーターのテンプレート引数を推定できません。型が、この理由により推定できないテンプレート引数を含んでいる場合、その型にあるすべてのテンプレート引数は、推定されません。次の例は、このことを示しています。

```
template<class T, class U, class V>
  void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
  Y<int>::Z<char> a;
  Y<int> b;
  Y<float> c;

  h<int, char, float>(a, b, c);
  h<int, char>(a, b, c);
  // h<int>(a, b, c);
}
```

コンパイラーは、typename Y<T>::template Z<U> のテンプレート引数 T および U を推定できません (しかし、Y<T> の T は推定します)。コンパイラーは、U がそのコンパイラーによって推定されないので、テンプレート関数呼び出し h<int>(a, b, c) を許可しません。

コンパイラーは、関数を指すポインターから、またはいくつかの多重定義関数名を与えられたメンバー関数引数を指すポインターから、関数テンプレート引数を推定できます。しかし、多重定義関数が、いずれも関数テンプレートではないこともあり、複数の多重定義関数が、要求される型と一致しないこともあります。次の例は、このことを示しています。

```
template<class T> void f(void(*) (T,int)) { };

template<class T> void g1(T, int) { };

void g2(int, int) { };
void g2(char, int) { };

void g3(int, int, int) { };
void g3(float, int) { };

int main() {
//   f(&g1);
//   f(&g2);
   f(&g3);
}
```

コンパイラーは、g1() が関数テンプレートなので、呼び出し f(&g1) を許可しません。コンパイラーは、g2() という名前の関数が、両方とも f() が必要とする型と一致するので、呼び出し f(&g2) を許可しません。

コンパイラーは、デフォルト引数の型からテンプレート引数を推定できません。次の例は、このことを示しています。

```
template<class T> void f(T = 2, T = 3) { };

int main() {
   f(6);
//   f();
   f<int>();
}
```

コンパイラーは、関数呼び出しの引数値からテンプレート引数 (int) を推定できるので、呼び出し f(6) を許可します。コンパイラーは、f() のデフォルトの引数からテンプレート引数を推定できないので、呼び出し f() を許可しません。

コンパイラーは、「非型」テンプレート引数の型からテンプレート型引数を推定できません。例えば、コンパイラーは次のコードを許可しません。

```
template<class T, T i> void f(int[20][i]) { };

int main() {
   int a[20][30];
   f(a);
}
```

コンパイラーは、テンプレート・パラメーター T の型を推定できません。

「非型」テンプレート引数の推定

コンパイラーは、境界が参照またはポインター型を参照しない限り、主配列の境界の値を推定できません。主配列の境界は、関数仮パラメーター型の一部ではありません。次のコードは、このことを示しています。

```
template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
```

```
   int b[10][20];
   int c[10];
   f(b);
   // g(c);
   h(c);
}
```

コンパイラーは、呼び出し g(c) を許可しません。コンパイラーは、テンプレート引数 i を推定できません。

コンパイラーは、テンプレート関数のパラメーター・リストの式で使用されている、「非型」テンプレート引数の値を推定できません。次の例は、このことを示しています。

```
template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
  X<0> a;
  f<1>(a);
  // f(a);
}
```

関数 f() をオブジェクト a で呼び出すためには、関数が、型 X<0> の引数を受諾する必要があります。しかし、コンパイラーは、X<i - 1> が X<0> と同等であるためには、テンプレート引数 i が、1 と等しい必要があるということを推定できません。したがって、コンパイラーは、関数呼び出し f(a) を許可しません。

コンパイラーに「非型」テンプレート引数を推定させたい場合、パラメーターの型が、関数呼び出しで使用される値の型と正確に一致しなければなりません。例えば、コンパイラーは次のことを許可しません。

```
template<int i> class A { };
template<short d> void f(A<d>) { };

int main() {
   A<1> a;
   f(a);
}
```

例で f() を呼び出す場合、コンパイラーは、int を short に変換しません。

しかし、推定された配列の境界は、整数型になります。

多重定義関数テンプレート

非テンプレート関数、または別の関数テンプレートのどちらかを使用して、関数テンプレートを多重定義できます。

多重定義関数テンプレートの名前を呼び出す場合、コンパイラーは、そのテンプレート引数の推定を試み、明示的に宣言されたテンプレート引数をチェックします。成功すれば、コンパイラーは、関数テンプレート特殊化のインスタンスを作成してから、この特殊化を多重定義解決で使用する*候補関数* のセットに追加します。コンパイラーは、多重定義解決を続け、候補関数のセットから最も適切な関数を選択します。非テンプレート関数は、テンプレート関数より優先順位があります。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }
```

```
int main() {
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');
}
```

上記の例の出力は、以下のとおりです。

```
Non-template
Template
Non-template
```

関数呼び出し f(1, 2) は、テンプレート関数と非テンプレート関数の両方の引数型と一致します。多重定義の解決では非テンプレート関数の方が優先順位が高いと見なされるため、非テンプレート関数が呼び出されます。

関数呼び出し f('a', 'b') は、テンプレート関数の引数型とだけ一致します。テンプレート関数が呼び出されます。

関数呼び出し f(1, 'b') では、引数の推定は失敗します。コンパイラーは、テンプレート関数特殊化を生成せず、また、多重定義解決も生じません。非テンプレート関数は、関数引数 'b' に、標準型変換を使用して char から int に変換した後で、この関数呼び出しを解決します。

関連情報

- 215 ページの『多重定義解決』

関数テンプレートの部分選択

関数テンプレートの特殊化は、テンプレート引数の推定で、特殊化が複数の多重定義と関連付けられるので、あいまいになります。そのため、コンパイラーは、最も特殊化された定義を選択します。関数テンプレート定義を選択するこの処理は、*部分選択* と呼ばれます。

X からの特殊化と一致する引数リストは、いずれも Y からの特殊化と一致するが、その逆では一致しないという場合は、テンプレート X は、テンプレート Y よりもさらに特殊化されています。次の例は、部分選択を示しています。

```
template<class T> void f(T)  { }
template<class T> void f(T*) { }
template<class T> void f(const T*) { }

template<class T> void g(T) { }
template<class T> void g(T&) { }

template<class T> void h(T) { }
template<class T> void h(T, ...) { }

int main() {
   const int *p;
   f(p);

   int q;
//  g(q);
//  h(q);
}
```

宣言 `template<class T> void f(const T*)` は、 `template<class T> void f(T*)` よりもさらに特殊化されています。したがって、関数呼び出し `f(p)` は、 `template<class T> void f(const T*)` を呼び出します。しかし、`void g(T)` および `void g(T&)` はどちらも、特殊化の程度には差はありません。したがって、関数呼び出し `g(q)` はあいまいになります。

省略符号は、部分選択に影響を与えません。したがって、関数呼び出し `h(q)` もあいまいです。

コンパイラーは、次の場合に、部分選択を使用します。

- 多重定義解決を必要とする関数テンプレート特殊化の呼び出し
- 関数テンプレート特殊化のアドレスの取得
- フレンド関数宣言、明示的インスタンス生成、または明示的特殊化が、関数テンプレート特殊化を参照するとき
- 任意の割り振り解除 new の関数テンプレートでもある適切な配置解除関数の決定

関連情報

- 319 ページの『テンプレートの特殊化』
- 141 ページの『new 式 (C++ のみ)』

テンプレートのインスタンス化

関数、クラス、またはクラスのメンバーの新規定義を、テンプレート宣言と 1 つ以上のテンプレート引数から作成する処理は、*テンプレートのインスタンス化* と呼ばれます。特定のテンプレート引数のセットを処理するためにテンプレートのインスタンス化から作成された定義は、*特殊化* と呼ばれます。

IBM 拡張

テンプレート・インスタンス生成のフォワード宣言の形式は、前に `extern` キーワードが付いた明示的テンプレート・インスタンス生成です。

IBM 拡張 の終り

テンプレート・インスタンス化宣言の構文

```
▶▶—extern—  template—template_declaration——————————————————————▶◀
```

この言語フィーチャーは、GNU C++ との互換性のための Standard C++ に対する直交拡張で、 318 ページの『明示的インスタンス生成』 で詳しく説明しています。

関連情報

- 319 ページの『テンプレートの特殊化』

暗黙のインスタンス生成

テンプレートの特殊化が明示的にインスタンス化されない限り、または明示的に特殊化されない限り、コンパイラーは、定義が必要とされる場合にのみ、テンプレートの特殊化を生成します。これは、*暗黙のインスタンス化* と呼ばれます。

コンパイラーが、クラス・テンプレート特殊化のインスタンスを生成する必要があり、テンプレートが宣言される場合、テンプレートも定義する必要があります。

例えば、クラスを指すポインターを宣言する場合、そのクラスの定義は、必要とされず、そのクラスは、暗黙的にインスタンス作成されません。次は、コンパイラーが、テンプレート・クラスのインスタンスを作成する例を示しています。

```
template<class T> class X {
  public:
    X* p;
    void f();
    void g();
};

X<int>* q;
X<int> r;
X<float>* s;
r.f();
s->g();
```

コンパイラーは、次のクラスおよび関数のインスタンス生成を必要とします。

- オブジェクト r が宣言されたときの X<int>
- メンバー関数呼び出し r.f() での X<int>::f()
- クラス・メンバー・アクセス関数呼び出し s->g() での X<float> および X<float>::g()

したがって、上記の例をコンパイルするには、関数 X<T>::f() および X<T>::g() を定義する必要があります。 (コンパイラーは、オブジェクト r を作成する場合、クラス X のデフォルトのコンストラクターを使用します。) コンパイラーは、次の定義のインスタンス生成を必要としません。

- ポインター p が宣言されたときの クラス X
- ポインター q が宣言されたときの X<int>
- ポインター s が宣言されたときの X<float>

コンパイラーが、ポインター型変換、またはメンバー型変換を指すポインターに関係する場合、それはクラス・テンプレート特殊化のインスタンスを暗黙的に生成します。次の例は、このことを示しています。

```
template<class T> class B { };
template<class T> class D : public B<T> { };

void g(D<double>* p, D<int>* q)
{
  B<double>* r = p;
  delete q;
}
```

割り当て B<double>* r = p は、型 D<double>* の p を B<double>* の型に変換します。コンパイラーは、D<double> のインスタンスを生成する必要があります。コンパイラーは、q の削除を試みる際に、D<int> のインスタンスを生成しなければなりません。

コンパイラーが、静的メンバーを含むクラス・テンプレートのインスタンスを暗黙的に生成する場合、それらの静的メンバーのインスタンスは、暗黙的には生成されません。コンパイラーは、静的メンバーの定義を必要とする場合のみ、静的メンバーのインスタンスを生成します。インスタンスを生成されたクラス・テンプレートは、いずれも静的メンバーのそれ自身のコピーを所有しています。次の例は、このことを示しています。

```
template<class T> class X {
public:
   static T v;
};

template<class T> T X<T>::v = 0;
```

```
X<char*> a;
X<float> b;
X<float> c;
```

オブジェクト a には、型 char* の静的メンバー変数 v があります。オブジェクト b には、型 float の静的変数 v があります。オブジェクト b と c は、単一静的データ・メンバー v を共用します。

暗黙的にインスタンスを生成されたテンプレートは、テンプレートを定義した場所と同じネーム・スペースにあります。

関数テンプレート、またはメンバー関数テンプレート特殊化が、多重定義解決にかかわってくる場合、コンパイラーは、特殊化の宣言のインスタンスを暗黙的に生成します。

関連情報

- 316 ページの『テンプレートのインスタンス化』

明示的インスタンス生成

コンパイラーに、テンプレートから定義をいつ生成するのかを明示的に指示できます。これは、*明示的インスタンス化* と呼ばれます。

明示的インスタンス化宣言の構文

```
►►──template──template_declaration────────────────────────────────►◄
```

下記は、明示的インスタンス生成の例です。

```
template<class T> class Array { void mf(); };
template class Array<char>;      // explicit instantiation
template void Array<int>::mf();  // explicit instantiation

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); // explicit instantiation

namespace N {
   template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);                  // explicit instantiation

template <class T> class X {
   private:
      T v(T arg) { return arg; };
};

template int X<int>::v(int);    // explicit instantiation

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

テンプレート宣言は、テンプレートの明示的インスタンス生成のインスタンス化のポイントのスコープ内に存在する必要があります。テンプレート特殊化の明示的インスタンス生成は、テンプレートを定義した場所と同じネーム・スペースにあります。

アクセス検査規則は、明示的インスタンス生成には適用されません。明示的インスタンス生成の宣言にあるテンプレート引数および名前は、private 型、またはオブジェクトになります。上記の例では、コンパイラーは、メンバー関数が private で宣言されていても、`template int X<int>::v(int)` を許可します。

テンプレートのインスタンスを明示的に生成する場合、コンパイラーは、デフォルトの引数を使用しません。上記の例では、コンパイラーは、デフォルトの引数が型 `int` のアドレスであっても、明示的インスタンス化 `template char g(char)` を許可します。

IBM 拡張

`extern` 修飾されたテンプレート宣言は、クラスまたは関数のインスタンスを生成しません。クラスと関数の両方において、`extern` テンプレートのインスタンス化は、先行するコードでまだインスタンス化でトリガーされていなければ、テンプレートのパーツをインスタンス化しません。クラスについては、そのメンバー (静的と非静的の両方) のインスタンスは生成されません。クラスをマップするために必要であれば、クラスそのもののインスタンスが生成されます。関数については、プロトタイプはインスタンスが生成されますが、テンプレート関数の本体のインスタンスは生成されません。

次の例では、`extern` を使用したテンプレートのインスタンス化を示します。

```
template<class T>class C {
   static int i;
   void f(T) { }
};
template<class U>int C<U>::i = 0;
extern template C<int>;  // extern explicit template instantiation
C<int>c;   // does not cause instantiation of C<int>::i
           // or C<int>::f(int) in this file,
           // but the class is instantiated for mapping
C<char>d;  // normal instantiations

template<class C> C foo(C c) { return c; }
extern template int foo<int>(int);  // extern explicit template instantiation
int i = foo(1);  // does not cause instantiation of the body of foo<int>
```

IBM 拡張 の終り

テンプレートの特殊化

関数、クラス、またはクラスのメンバーの新規定義を、テンプレート宣言と 1 つ以上のテンプレート引数から作成する処理は、*テンプレートのインスタンス化* と呼ばれます。テンプレートのインスタンス化から作成された定義は、*特殊化* と呼ばれます。*主テンプレート* とは、特殊化しようとしているテンプレートのことです。

関連情報

- 316 ページの『テンプレートのインスタンス化』

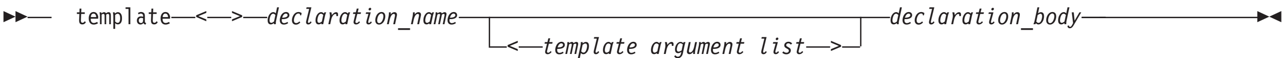
明示的特殊化

テンプレート引数の特定のセットでテンプレートをインスタンス化する場合、コンパイラーは、それらのテンプレート引数に基づいて新規の定義を生成します。この定義生成の振る舞いをオーバーライドできます。その代わりに、コンパイラーがテンプレート引数の任意のセットで使用する定義を、指定することができます。これは、*明示的特殊化* と呼ばれます。次のものを明示的に特殊化できます。

- 関数またはクラス・テンプレート
- クラス・テンプレートのメンバー関数

- クラス・テンプレートの静的データ・メンバー
- クラス・テンプレートのメンバー・クラス
- クラス・テンプレートのメンバー関数テンプレート
- クラス・テンプレートのメンバー・クラス・テンプレート

明示的特殊化宣言の構文



`template<>` 接頭部は、次のテンプレート宣言が、テンプレート・パラメーターを取得しないということを示しています。 *declaration_name* は、以前宣言されたテンプレートの名前です。少なくとも特殊化が参照されているまでは、明示的特殊化を前もって宣言できること、その場合 *declaration_body* は、オプションであることに注意してください。

次の例は、明示的特殊化を示しています。

```
using namespace std;

template<class T = float, int i = 5> class A
{
   public:
      A();
      int value;
};

template<> class A<> { public: A(); };
template<> class A<double, 10> { public: A(); };

template<class T, int i> A<T, i>::A() : value(i) {
   cout << "Primary template, "
        << "non-type argument is " << value << endl;
}

A<>::A() {
   cout << "Explicit specialization "
        << "default arguments" << endl;
}

A<double, 10>::A() {
   cout << "Explicit specialization "
        << "<double, 10>" << endl;
}

int main() {
   A<int,6> x;
   A<> y;
   A<double, 10> z;
}
```

上記の例の出力は、以下のとおりです。

```
Primary template non-type argument is: 6
Explicit specialization default arguments
Explicit specialization <double, 10>
```

この例では、主テンプレート (特殊化しようとしているテンプレート) クラス `A` に対して、2 つの明示的特殊化を宣言しました。オブジェクト `x` は、主テンプレートのコンストラクターを使用します。オブジェクト `y` は、明示的特殊化 `A<>::A()` を使用します。オブジェクト `z` は、明示的特殊化 `A<double, 10>::A()` を使用します。

関連情報

- 308 ページの『関数テンプレート』
- 304 ページの『クラス・テンプレート』
- 306 ページの『クラス・テンプレートのメンバー関数』
- 306 ページの『静的データ・メンバーとテンプレート』

明示的特殊化の定義と宣言

明示的特殊化クラスの定義は、主テンプレートの定義とは無関係です。特殊化を定義するために、主テンプレートを定義する必要はありません (または、主テンプレートを定義するために、特殊化を定義する必要もありません)。例えば、コンパイラーは、次のコードを許可します。

```
template<class T> class A;
template<> class A<int>;

template<> class A<int> { /* ... */ };
```

主テンプレートは定義されませんが、明示的特殊化は定義されます。

宣言はされているが、不完全なクラスと同じように定義されていない、明示的特殊化の名前を使用できます。次の例は、このことを示しています。

```
template<class T> class X { };
template<>  class X<char>;
X<char>* p;
X<int> i;
// X<char> j;
```

コンパイラーは、宣言 `X<char>` の明示的特殊化が定義されていないので、`X<char> j` を許可しません。

明示的特殊化とスコープ

主テンプレートの宣言は、明示的特殊化を宣言するポイント のあるスコープ内になければなりません。言い換えれば、明示的特殊化宣言は、主テンプレートの宣言の後に入れなければなりません。例えば、コンパイラーは次のことを許可しません。

```
template<> class A<int>;
template<class T> class A;
```

明示的特殊化は、主テンプレートの定義と同じネーム・スペースに存在します。

明示的特殊化のクラス・メンバー

明示的特殊化クラスのメンバーは、主テンプレートのメンバー宣言から暗黙的にインスタンス化されることはありません。クラス・テンプレート特殊化のメンバーを明示的に定義する必要があります。明示的特殊化テンプレート・クラスのメンバーを、`template<>` 接頭部を使用せずに、標準クラスのメンバーを定義するのと同じように定義します。さらに、明示的特殊化のメンバーをインラインに定義できます。ここでは、特別なテンプレート構文は使用されていません。次の例は、クラス・テンプレート特殊化を示しています。

```
template<class T> class A {
   public:
      void f(T);
};

template<> class A<int> {
   public:
      int g(int);
};

int A<int>::g(int arg) { return 0; }
```

```
int main() {
   A<int> a;
   a.g(1234);
}
```

明示的特殊化 A<int> は、メンバー関数 g() を含んでいますが、主テンプレートは、これを含んでいません。

テンプレート、メンバー・テンプレート、またはクラス・テンプレートのメンバーを明示的に特殊化する場合、この特殊化を宣言してから、特殊化のインスタンスを暗黙的に生成する必要があります。例えば、コンパイラーは次のコードを許可しません。

```
template<class T> class A { };

void f() { A<int> x; }
template<> class A<int> { };

int main() { f(); }
```

コンパイラーは、関数 f() が、特殊化の前に、この特殊化 (x の構造体にある) を使用するので、明示的特殊化 template<> class A<int> { }; を許可しません。

関数テンプレートの明示的特殊化

関数テンプレート特殊化では、コンパイラーが関数引数の型からテンプレート引数を推定できるのであれば、テンプレート引数はオプションです。次の例は、このことを示しています。

```
template<class T> class X { };
template<class T> void f(X<T>);
template<> void f(X<int>);
```

明示的特殊化 template<> void f(X<int>) は、 template<> void f<int>(X<int>) と同等です。

次に関する宣言および定義に対しては、デフォルトの関数引数は、指定できません。

- 関数テンプレートの明示的特殊化
- メンバー関数テンプレートの明示的特殊化

例えば、コンパイラーは次のコードを許可しません。

```
template<class T> void f(T a) { };
template<> void f<int>(int a = 5) { };

template<class T> class X {
  void f(T a) { }
};
template<> void X<int>::f(int a = 10) { };
```

関連情報

- 308 ページの『関数テンプレート』

クラス・テンプレートのメンバーの明示的特殊化

インスタンス化された各クラス・テンプレート特殊化は、静的メンバーの専用コピーを所有します。静的メンバーを明示的に特殊化できます。次の例は、このことを示しています。

```
template<class T> class X {
public:
   static T v;
   static void f(T);
};
```

```
template<class T> T X<T>::v = 0;
template<class T> void X<T>::f(T arg) { v = arg; }

template<> char* X<char*>::v = "Hello";
template<> void X<float>::f(float arg) { v = arg * 2; }

int main() {
   X<char*> a, b;
   X<float> c;
   c.f(10);
}
```

このコードは、テンプレート引数 char* がストリング "Hello" を指すようにする、静的データ・メンバー X::v の初期化を明示的に特殊化します。関数 X::f() は、テンプレート引数 float に対して明示的に特殊化されます。オブジェクト a および b の静的データ・メンバー v は、同じストリング、つまり "Hello" を指します。 c.v の値は、関数呼び出し c.f(10) の後で 20 になります。

メンバー・テンプレートを、複数の囲みクラス・テンプレート内でネストできます。いくつかの囲みクラス・テンプレート内でネストされたテンプレートを明示的に特殊化する場合、特殊化するすべての囲みクラス・テンプレートに、その宣言の前に template<> を付ける必要があります。特殊化されていない囲みクラス・テンプレートも残すことはできますが、その囲みクラス・テンプレートを明示的に特殊化しない限り、ネスト・クラス・テンプレートを明示的に特殊化できません。次の例は、ネストされたメンバー・テンプレートの明示的特殊化を示しています。

```
#include <iostream>
using namespace std;

template<class T> class X {
public:
  template<class U> class Y {
  public:
    template<class V> void f(U,V);
    void g(U);
  };
};

template<class T> template<class U> template<class V>
  void X<T>::Y<U>::f(U, V) { cout << "Template 1" <<   endl; }

template<class T> template<class U>
  void X<T>::Y<U>::g(U) { cout << "Template 2" <<   endl; }

template<> template<>
  void X<int>::Y<int>::g(int) { cout << "Template 3"   << endl; }

template<> template<> template<class V>
  void X<int>::Y<int>::f(int, V) { cout << "Template 4" << endl; }

template<> template<> template<>
  void X<int>::Y<int>::f<int>(int, int) { cout << "Template 5" << endl; }

// template<> template<class U> template<class V>
//    void X<char>::Y<U>::f(U, V) { cout << "Template 6" << endl; }

// template<class T> template<>
//    void X<T>::Y<float>::g(float) { cout << "Template 7" << endl; }

int main() {
  X<int>::Y<int> a;
  X<char>::Y<char> b;
  a.f(1, 2);
  a.f(3, 'x');
```

```
  a.g(3);
  b.f('x', 'y');
  b.g('z');
}
```

下記は、上記のプログラムの出力です。

```
Template 5
Template 4
Template 3
Template 1
Template 2
```

- コンパイラーは、メンバー (関数 f()) を、それが含んでいるクラス (Y) を特殊化せずに特殊化しようとしているので、"Template 6" を出力するテンプレート特殊化定義を許可しません。
- コンパイラーは、クラス Y の囲みクラス (クラス X) が明示的に特殊化されていないので、"Template 7" を出力するテンプレート特殊化定義を許可しません。

フレンド宣言は、明示的特殊化を宣言できません。

関連情報

- 306 ページの『静的データ・メンバーとテンプレート』

部分的特殊化

クラス・テンプレートをインスタンス化する場合、コンパイラーは、受け渡したテンプレート引数に基づいて定義を作成します。代替として、それらすべてのテンプレート引数が、明示的特殊化のものと一致する場合、コンパイラーは、明示的特殊化が定義した定義を使用します。

部分的特殊化 は、明示的特殊化に汎用性を持たせたものです。明示的特殊化は、テンプレート引数リストだけを持っています。部分的特殊化は、テンプレート引数リストとテンプレート・パラメーター・リストの両方を持っています。コンパイラーは、テンプレート引数リストが、テンプレートのインスタンス生成のテンプレート引数のサブセットと一致する場合、部分的特殊化を使用します。そして、コンパイラーは、テンプレートのインスタンス生成の一致しない残りのテンプレート引数を使用して、部分的特殊化から新規定義を生成します。

関数テンプレートは、部分的に特殊化することはできません。

部分的特殊化の構文

```
►►──template──<template_parameter_list>──declaration_name──<template_argument_list>──────────►

►─declaration_body──────────────────────────────────────────────────────────────────────────►◄
```

declaration_name は、以前宣言されたテンプレートの名前です。 *declaration_body* はオプションなので、部分的特殊化を前もって宣言できることに注意してください。

以下は、部分的特殊化の使用法を示したものです。

```
#include <iostream>
using namespace std;

template<class T, class U, int I> struct X
  { void f() { cout << "Primary template" << endl; } };

template<class T, int I> struct X<T, T*, I>
  { void f() { cout << "Partial specialization 1" << endl;
  } };
```

```
template<class T, class U, int I> struct X<T*, U, I>
  { void f() { cout << "Partial specialization 2" << endl;
  } };

template<class T> struct X<int, T*, 10>
  { void f() { cout << "Partial specialization 3" << endl;
  } };

template<class T, class U, int I> struct X<T, U*, I>
  { void f() { cout << "Partial specialization 4" << endl;
  } };

int main() {
   X<int, int, 10> a;
   X<int, int*, 5> b;
   X<int*, float, 10> c;
   X<int, char*, 10> d;
   X<float, int*, 10> e;
//   X<int, int*, 10> f;
   a.f(); b.f(); c.f(); d.f(); e.f();
}
```

上記の例の出力は、以下のとおりです。

```
Primary template
Partial specialization 1
Partial specialization 2
Partial specialization 3
Partial specialization 4
```

コンパイラーは、宣言 X<int, int*, 10> f が、 template struct X<T, T*, I>、template struct X<int, T*, 10>、または template struct X<T, U*, I> と一致し、どれも他よりうまく一致する訳ではないので、この宣言を許可しません。

各クラス・テンプレートの部分的特殊化は、別々のテンプレートです。クラス・テンプレートの部分的特殊化のメンバーごとに、定義が必要です。

部分的特殊化のテンプレート・パラメーターと引数リスト

主テンプレートは、テンプレート引数リストを持っていません。このリストは、テンプレート・パラメーター・リストに含まれています。

テンプレート・パラメーターを主テンプレートでは指定しているが、部分的特殊化で使用していなければ、それを部分的特殊化のテンプレート・パラメーター・リストから省略できます。部分的特殊化の引数リストの順序は、主テンプレートの暗黙の引数リストの順序と同じです。

部分的テンプレート・パラメーターのテンプレート引数リストでは、式が ID のみとなる場合を除いて、非型引数を含む式を持つことはできません。次の例では、コンパイラーは、最初の部分的特殊化を許可しませんが、2 番目のものは許可します。

```
template<int I, int J> class X { };

// Invalid partial specialization
template<int I> class X <I * 4, I + 3> { };

// Valid partial specialization
template <int I> class X <I, I> { };
```

「非型」テンプレート引数の型は、部分的特殊化のテンプレート・パラメーターに依存できません。コンパイラーは、次の部分的特殊化を許可しません。

```
template<class T, T i> class X { };

// Invalid partial specialization
template<class T> class X<T, 25> { };
```

部分的特殊化のテンプレート引数リストは、主テンプレートによる暗黙のリストと同じにすることはできません。

部分的特殊化のテンプレート・パラメーター・リストに、デフォルト値を持つことができません。

関連情報

- 298 ページの『テンプレート・パラメーター』
- 300 ページの『テンプレート引数』

クラス・テンプレートの部分的特殊化のマッチング

コンパイラーは、クラス・テンプレート特殊化のテンプレート引数と、主テンプレートおよび部分的特殊化のテンプレート引数リストを突き合わせて、主テンプレートを使用するのか、その部分的特殊化の 1 つを使用するのかを判別します。

- コンパイラーが特殊化を 1 つだけ検出する場合、コンパイラーは、その特殊化から定義を生成します。
- コンパイラーが複数の特殊化を検出する場合、コンパイラーは、どの特殊化が最も特殊化されているのかを判別します。 X からの特殊化と一致する引数リストは、いずれも Y からの特殊化と一致するが、その逆では一致しないという場合は、テンプレート X は、テンプレート Y よりもさらに特殊化されています。コンパイラーが最も特殊化された特殊化を検出できない場合は、クラス・テンプレートの使用は、あいまいになります。つまり、コンパイラーは、プログラムを許可しません。
- コンパイラーがどのような一致も検出しない場合、コンパイラーは、主テンプレートから定義を生成します。

名前のバインディングと従属名

名前のバインディング は、テンプレートで明示的に、または暗黙的に使用されている名前ごとに宣言を検出する処理です。コンパイラーは、テンプレートの定義で名前をバインドしたり、またはテンプレートのインスタンス生成において名前をバインドします。

従属名 は、テンプレート・パラメーターの型、または値に依存する名前です。次に例を示します。

```
template<class T> class U : A<T>
{
  typename T::B x;
  void f(A<T>& y)
  {
    *y++;
  }
};
```

この例では、従属名は、基底クラス A<T>、型名 T::B、および変数 y です。

テンプレートのインスタンスが作成されると、コンパイラーは、従属名をバインドします。テンプレートが定義されると、コンパイラーは、非従属名をバインドします。次に例を示します。

```
void f(double) { cout << "Function f(double)" << endl; }

template<class T> void g(T a) {
  f(123);
  h(a);
}
```

```
void f(int) { cout << "Function f(int)" << endl; }
void h(double) { cout << "Function h(double)" << endl; }

void i() {
  extern void h(int);
  g<int>(234);
}

void h(int) { cout << "Function h(int)" << endl; }
```

関数 i() を呼び出す場合、以下が出力されます。

```
Function f(double)
Function h(double)
```

テンプレートの*定義のポイント* は、それの定義の直前に配置されます。この例では、関数テンプレート g(T) の定義のポイントは、キーワード template の直前に配置されます。関数呼び出し f(123) は、テンプレート引数に依存しないので、コンパイラーは、関数テンプレート g(T) の定義の前に宣言された名前を考慮します。したがって、呼び出し f(123) は、f(double) を呼び出します。 f(int) のほうが的確ですが、それは、g(T) の定義のポイントのあるスコープ内に存在していません。

テンプレートの*インスタンス生成のポイント* は、その使用を囲む宣言の直前に配置されます。この例では、g<int>(234) 呼び出しのインスタンス生成のポイントは、i() の直前に配置されます。関数呼び出し h(a) がテンプレート引数に依存するので、コンパイラーは、関数テンプレート g(T) のインスタンス生成の前に、宣言された名前を考慮します。したがって、h(a) の呼び出しは、h(double) を呼び出します。この関数が g<int>(234) のインスタンス生成のポイントのあるスコープ内に存在しないので、コンパイラーは、h(int) を考慮しません。

インスタンス生成バインディングのポイントは、次のことを意味しています。

- テンプレート・パラメーターは、ローカル名、またはクラス・メンバーに依存できない。
- テンプレートの修飾名は、ローカル名、またはクラス・メンバーに依存できない。

関連情報

- 316 ページの『テンプレートのインスタンス化』

型名キーワード

型を参照する修飾名やテンプレート・パラメーターに依存する修飾名がある場合は、キーワード typename を使用してください。キーワード typename のみを、テンプレート宣言または定義で使用してください。次の例は、キーワード typename の使用法を示しています。

```
template<class T> class A
{
  T::x(y);
  typedef char C;
  A::C d;
}
```

ステートメント T::x(y) は、あいまいです。そのステートメントは、非ローカル引数 y を使用した関数 x() の呼び出しや、または、型 T::x を使用して変数 y の宣言にすることができます。 C++ は、このステートメントを関数呼び出しとして解釈します。コンパイラーにこのステートメントを宣言として解釈させるには、キーワード typename をそのステートメントの開始位置に追加します。ステートメント A::C d; は、不適格です。クラス A は、A<T> も参照するので、テンプレート・パラメーターに依存します。キーワード typename をこの宣言の開始位置に追加する必要があります。

```
    typename A::C d;
```

テンプレート・パラメーター宣言で、キーワード class の代わりに、キーワード typename も使用できます。

関連情報

- 298 ページの『テンプレート・パラメーター』

修飾子としての template キーワード

メンバー・テンプレートと他の名前を区別するために、キーワード template を修飾子として使用してください。次の例は、template を修飾子として使用しなければならない状況を示しています。

```
class A
{
  public:
    template<class T> T function_m() { };
};

template<class U> void function_n(U argument)
{
  char object_x = argument.function_m<char>();
}
```

宣言 char object_x = argument.function_m<char>(); は、不適格です。コンパイラーは、< が「より小演算子」であると見なします。コンパイラーに関数テンプレート呼び出しを認識させるには、template 修飾子を追加する必要があります。

```
char object_x = argument.template function_m<char>();
```

メンバー・テンプレート特殊化の名前が . 演算子、-> 演算子、または :: 演算子の後にあって、かつその名前が明示的に修飾されたテンプレート・パラメーターを持っている場合は、メンバー・テンプレート名の前にキーワード template を付けてください。次の例は、このキーワード template の使用法を示しています。

```
#include <iostream>
using namespace std;

class X {
   public:
      template <int j> struct S {
         void h() {
            cout << "member template's member function: " << j << endl;
         }
      };
      template <int i> void f() {
        cout << "Primary: " << i << endl;
      }
};

template<> void X::f<20>() {
   cout << "Specialized, non-type argument = 20" << endl;
}

template<class T> void g(T* p) {
   p->template f<100>();
   p->template f<20>();
   typename T::template S<40> s; // use of scope operator on a member template
   s.h();
}

int main()
```

```
{
   X temp;
   g(&temp);
}
```

上記の例の出力は、以下のとおりです。

```
Primary: 100
Specialized, non-type argument = 20
member template's member function: 40
```

これらのケースでキーワード template を使用しない場合、コンパイラーは、< を「より小演算子」として解釈します。例えば、次のコード行は、不適格です。

```
p->f<100>();
```

コンパイラーは、f を非テンプレート・メンバーとして、 < を「より小演算子」として解釈します。

第 16 章 例外処理 (C++ のみ)

例外処理 は、例外的な状況を検出したり、処理するコードをプログラムの他の部分から分離するメカニズムです。例外的な状況は、必ずしもエラーではないことに注意してください。

関数が例外的な状況を検出する場合、オブジェクトでこれを表します。このオブジェクトを*例外オブジェクト* と呼びます。 例外的な状況を処理するには、*例外*をスローします。これによって、例外をスローした関数を直接的または間接的に呼び出した元のコードの指定ブロックに、制御だけでなく例外も渡されます。コードのこのブロックは、ハンドラー と呼ばれます。 処理させる例外のタイプを、ハンドラーに指定します。 C++ ランタイムは、生成コードとともに、スローされた例外を処理できる最初の適切なハンドラーに制御を渡します。これが起きる場合、例外はキャッチ されました。ハンドラーは、別のハンドラーが例外をキャッチできるように、それを*再スロー* します。

400 IBM i 固有の使用情報については、「*ILE C/C++ Programmer's Guide*」の第 21 章「Handling Exceptions in a Program」を参照してください。

例外処理のメカニズムは、次の要素で構成されます。

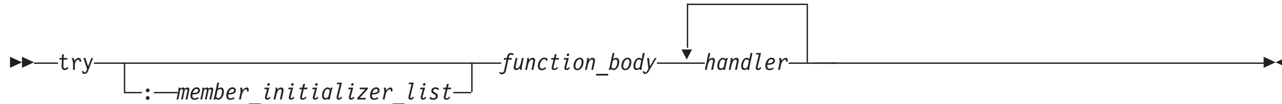
- `try` ブロック
- `catch` ブロック
- `throw` 式
- 例外指定

try ブロック

try ブロック を使用して、すぐに処理する例外をスローする可能性のあるプログラムのエリアを示します。*関数 try ブロック* を使用して、関数本体の全体で例外を検出することを指示します。

try ブロック構文

関数 try ブロック構文



以下は、メンバー初期化指定子、関数 try ブロック、および try ブロックを持つ関数 try ブロックの例です。

```
#include <iostream>
using namespace std;

class E {
   public:
```

```
      const char* error;
      E(const char* arg) : error(arg) { }
};

class A {
   public:
      int i;

      // A function try block with a member
      // initializer
      A() try : i(0) {
         throw E("Exception thrown in A()");
      }
      catch (E& e) {
         cout << e.error << endl;
      }
};

// A function try block
void f() try {
   throw E("Exception thrown in f()");
}
catch (E& e) {
   cout << e.error << endl;
}

void g() {
   throw E("Exception thrown in g()");
}

int main() {
   f();

   // A try block
   try {
      g();
   }
   catch (E& e) {
      cout << e.error << endl;
   }
   try {
      A x;
   }
   catch(...) { }
}
```

上記の例の出力は、以下のとおりです。

```
Exception thrown in f()
Exception thrown in g()
Exception thrown in A()
```

クラス A のコンストラクターには、メンバー初期化指定子を持つ関数 try ブロックがあります。関数 f() には、関数 try ブロックがあります。 main() 関数は、try ブロックを含んでいます。

関連情報

- 282 ページの『基底クラスおよびメンバーの初期化』

ネストされた try ブロック

try ブロックがネストされており、内側の try ブロックによって呼び出された関数内で throw が生じる場合は、制御は、引数が throw 式の引数と一致する最初の catch ブロックが見つかるまで、ネストされた try ブロック間を外側に向けて渡されて行きます。

次に例を示します。

```
try
{
      func1();
      try
      {
             func2();
      }
      catch (spec_err) { /* ... */ }
      func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

上記の例で、spec_err が内側の try ブロック (この場合は、func2() から) 内でスローされる場合、内側の catch ブロックがこの例外をキャッチします。この catch ブロックが制御権移動を行わない場合は、func3() が呼び出されます。内側の try ブロックの後で spec_err がスローされた場合 (例えば func3() によって)、この例外はキャッチされず、関数 terminate() が呼び出されます。内側の try ブロックの中の func2() からスローされた例外が、type_err である場合、プログラムは、func3() を呼び出さずに、両方の try ブロックから出て 2 番目の catch ブロックにスキップします。内側の try ブロックの後には、適切な catch ブロックがないためです。

catch ブロック内で try ブロックをネストすることもできます。

catch ブロック

catch ブロック構文

```
▶▶—catch—(—exception_declaration—)—{—ステートメント—}——————————▶◀
```

ハンドラーが、多くの型の例外をキャッチできるように宣言することができます。関数がキャッチできる許容オブジェクトは、catch キーワードの後に続く括弧の中 (*exception_declaration*) に宣言します。 基本的な型のオブジェクト、基底および派生クラス・オブジェクト、参照、およびこれらすべての型を指すポインターをキャッチすることができます。 const 型と volatile 型もキャッチすることができます。 *exception_declaration* を、非完了型、または以下を除く非完了型を指す参照、またはポインターにすることはできません。

- void*
- const void*
- volatile void*
- const volatile void*

exception_declaration では、型を定義できません。

catch(...) 形式のハンドラーを使用して、以前の catch ブロックでキャッチされなかった、スローされた例外をすべてキャッチすることができます。catch 引数の中の省略符号は、このハンドラーが、スローされたどの例外も処理できることを示しています。

catch(...) ブロックによって例外がキャッチされた場合には、スローされたオブジェクトにアクセスする直接的方法はありません。 catch(...) によって、キャッチされた例外に関する情報は、非常に限られています。

catch ブロック内にあるスローされたオブジェクトにアクセスしたい場合は、オプションの変数名を宣言することができます。

catch ブロックはアクセス可能オブジェクトしかキャッチできません。キャッチされたオブジェクトには、アクセス可能なコピー・コンストラクターがあるはずです。

関連情報

- 62 ページの『型修飾子』
- 243 ページの『メンバー・アクセス』

関数 try ブロック・ハンドラー

関数またはコンストラクターのパラメーターのスコープおよび存続期間が、関数 try ブロックのハンドラーにまで適用されます。次の例は、このことを示しています。

```
void f(int &x) try {
   throw 10;
}
catch (const int &i)
{
   x = i;
}

int main() {
   int v = 0;
   f(v);
}
```

f() が呼び出された後は、v の値は、10 になります。

main() の関数 try ブロックは、静的ストレージ期間を持つオブジェクトのデストラクター、またはネーム・スペース・スコープ・オブジェクトのコンストラクターで、スローされる例外をキャッチしません。

次の例は、静的オブジェクトのデストラクターから例外をスローします。

```
#include <iostream>
using namespace std;

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};

class A {
  public: ~A() { throw E("Exception in ~A()"); }
};

class B {
  public: ~B() { throw E("Exception in ~B()"); }
};

int main() try {
  cout << "In main" << endl;
  static A cow;
  B bull;
}
catch (E& e) {
  cout << e.error << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In main
Exception in ~B()
```

ランタイムは、オブジェクト cow が、プログラムの終わりに破棄される時にスローされる例外をキャッチできません。

次の例は、ネーム・スペース・スコープ・オブジェクトのコンストラクターから例外をスローします。

```
#include <iostream>
using namespace std;

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};

namespace N {
  class C {
  public:
    C() {
      cout << "In C()" << endl;
      throw E("Exception in C()");
    }
  };

  C calf;
};

int main() try {
  cout << "In main" << endl;
}
catch (E& e) {
  cout << e.error << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In C()
```

コンパイラーは、オブジェクト calf の作成時にスローされる例外をキャッチできません。

関数 try ブロックのハンドラーでは、コンストラクター本体、またはデストラクター本体にジャンプすることはできません。

リターン・ステートメントは、コンストラクターの関数 try ブロック・ハンドラー内に置くことはできません。

オブジェクトのコンストラクター、またはデストラクターの関数 try ブロック・ハンドラーに入ると、そのオブジェクトの完全な構成の基底クラスおよびメンバーは破棄されます。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class B {
   public:
      B() { };
```

```
      ~B() { cout << "~B() called" << endl; };
};

class D : public B {
   public:
      D();
      ~D() { cout << "~D() called" << endl; };
};

D::D() try : B() {
   throw E("Exception in D()");
}
catch(E& e) {
   cout << "Handler of function try block of D(): " << e.error << endl;
};

int main() {
   try {
      D val;
   }
   catch(...) { }
}
```

上記の例の出力は、以下のとおりです。

```
~B() called
Handler of function try block of D(): Exception in D()
```

D() の関数 try ブロックのハンドラーに入ると、ランタイムは、まず最初に D の基底クラスのデストラクター、つまり B を呼び出します。 val が完全に構成されていないので、D のデストラクターは呼び出されません。

ランタイムは、コンストラクターまたはデストラクターの関数 try ブロックのハンドラーの終了時に、例外を再スローします。その他のすべての関数は、関数 try ブロックのハンドラーの終了に到達した時点でリターンします。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class A {
   public:
      A() try { throw E("Exception in A()"); }
      catch(E& e) { cout << "Handler in A(): " << e.error << endl; }
};

int f() try {
   throw E("Exception in f()");
   return 0;
}
catch(E& e) {
   cout << "Handler in f(): " << e.error << endl;
   return 1;
}

int main() {
   int i = 0;
   try { A cow; }
   catch(E& e) {
      cout << "Handler in main(): " << e.error << endl;
```

```
   }

   try { i = f(); }
   catch(E& e) {
      cout << "Another handler in main(): " << e.error << endl;
   }

   cout << "Returned value of f(): " << i << endl;
}
```

上記の例の出力は、以下のとおりです。

```
Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1
```

関連情報

- 188 ページの『main() 関数』
- 40 ページの『static ストレージ・クラス指定子』
- 197 ページの『第 9 章 ネーム・スペース (C++ のみ)』
- 286 ページの『デストラクター』

catch ブロックの引数

catch ブロックの引数に対してクラス型を指定する場合 (*exception_declaration*)、コンパイラーはコピー・コンストラクターを使用して、その引数を初期化します。その引数に名前が入っていなければ、コンパイラーは、一時オブジェクトを初期化し、ハンドラーが終了するとき、それを破棄します。

冗長と思われる場合、ISO C++ 仕様に一時オブジェクトを作成するコンパイラーは不要です。コンパイラーは、この規則を利用して、より効率的な最適化コードを生成します。プログラムをデバッグする場合、特にメモリー問題のデバッグにおいては、このことを考慮してください。

スローされる例外とキャッチされる例外のマッチング

ハンドラーの catch 引数の中の引数は、次のいずれかの条件が満たされる場合、throw 式 (throw 引数) の *assignment_expression* の引数と一致します。

- catch 引数の型が、スローされたオブジェクトの型と一致する。
- catch 引数が、スローされたクラス・オブジェクトのパブリック基底クラスである。
- catch がポインターの型を指定し、スローされたオブジェクトが、標準ポインター型変換によって catch 引数のポインター型に変換できるポインター型である。

注: スローされたオブジェクトの型が `const` または `volatile` である場合、一致するには、catch 引数も `const` または `volatile` であることが必要です。ただし、`const`、`volatile`、または参照型の catch 引数が、非定数、非 volatile、または非参照オブジェクト型と一致することがあります。非参照 catch 引数型は、同じ型のオブジェクトへの参照と一致します。

関連情報

- 99 ページの『ポインター型変換』
- 62 ページの『型修飾子』
- 80 ページの『参照 (C++ のみ)』

キャッチの順序

コンパイラーが try ブロックで例外を検出すると、その出現の順に各ハンドラーを試行します。

基底クラスのオブジェクト用の catch ブロックが、その基底クラスから派生するクラスのオブジェクト用の catch ブロックより前にある場合は、コンパイラーは警告を発行し、派生クラス・ハンドラー内に到達不能コードがあっても、プログラムのコンパイルを続行します。

`catch(...)` の書式の catch ブロックは、try ブロックの後に続く最後の catch ブロックでなければならず、そうでなければエラーが起こります。このように配置することによって、`catch(...)` ブロックによって、さらに特定の catch ブロックが、本来キャッチすることになっている例外をキャッチすることを防ぐことができなくなります。

ランタイムが、現行スコープ内に一致するハンドラーを検出できない場合、ランタイムは、動的な囲み try ブロック内で一致するハンドラーの検出を継続します。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { };
};

class F : public E {
public:
  F(const char* arg) : E(arg) { };
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    throw E("Class E exception");
  }
  catch (F& e) {
    cout << "In handler of f()";
    cout << e.error << endl;
  }
};

int main() {
  try {
    cout << "In main" << endl;
    f();
  }
  catch (E& e) {
    cout << "In handler of main: ";
    cout << e.error << endl;
  };
  cout << "Resume execution in main" << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In main
In try block of f()
In handler of main: Class E exception
Resume execution in main
```

関数 `f()` では、ランタイムは、スローされた型 `E` の例外を処理するハンドラーを検出できません。ランタイムは、動的な囲み try ブロック内、つまり `main()` 関数の try ブロック内で、一致するハンドラーを検出します。

ランタイムが、プログラムで一致するハンドラーを検出できない場合、`terminate()` 関数を呼び出します。

関連情報

- 331 ページの『try ブロック』

throw 式

throw 式 は、プログラムで例外が生じたことを示すために使用します。

throw 例外構文

assignment_expression の型は、非完了型、または以下を除く非完了型を指すポインターにすることはできません。

- `void*`
- `const void*`
- `volatile void*`
- `const volatile void*`

assignment_expression は、呼び出しでの関数引数、またはリターン・ステートメントのオペランドと同じように扱われます。

assignment_expression が、クラス・オブジェクトの場合、そのオブジェクトのコピー・コンストラクターおよびデストラクターは、アクセス可能でなければなりません。例えば、プライベートとして宣言されたコピー・コンストラクターを持つクラス・オブジェクトをスローすることはできません。

400 IBM i では、スローされるオブジェクトのサイズは、16MB を超えることができないという制限があります。

関連情報

- 36 ページの『不完全型』

例外の再スロー

catch ブロックが、キャッチした特定の例外を処理できない場合、その例外を再スローする (rethrow) ことができます。rethrow 式 (*assignment_expression* がない `throw`) は、最初にスローされたオブジェクトを再びスローします。

例外が、rethrow 式が発生するスコープですでにキャッチされているので、その例外は、次の動的な囲み try ブロックへ再びスローされます。したがって、rethrow 式の発生したスコープの catch ブロックは、その例外を処理できなくなります。動的な囲み try ブロックの catch ブロックは、いずれも、例外をキャッチする機会があります。

次の例は、例外の再スローを示しています。

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
```

```
  E() : message("Class E") { }
};

struct E1 : E {
  const char* message;
  E1() : message("Class E1") { }
};

struct E2 : E {
  const char* message;
  E2() : message("Class E2") { }
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    cout << "Throwing exception of type E1" << endl;
    E1 myException;
    throw myException;
  }
  catch (E2& e) {
    cout << "In handler of f(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E1& e) {
    cout << "In handler of f(), catch (E1& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E& e) {
    cout << "In handler of f(), catch (E& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
}

int main() {
  try {
    cout << "In try block of main()" << endl;
    f();
  }
  catch (E2& e) {
    cout << "In handler of main(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
  }
  catch (...) {
    cout << "In handler of main(), catch (...)" << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In try block of main()
In try block of f()
Throwing exception of type E1
In handler of f(), catch (E1& e)
Exception: Class E1
In handler of main(), catch (...)
```

main() 関数の try ブロックは、関数 f() を呼び出します。関数 f() の try ブロックは、myException という名前の、型 E1 のオブジェクトをスローします。ハンドラー catch (E1 &e) が、myException キャッチします。それからハンドラーは、ステートメント throw を使用して、myException を、次の動的な囲み try ブロック、つまり main() 関数の try ブロックに再びスローします。ハンドラー catch(...) が、myException をキャッチします。

スタックのアンワインド

例外がスローされ、制御が try ブロックからハンドラーに渡されると、C++ ランタイムは、 try ブロックの開始以降に作成されたすべての自動オブジェクトに対して、デストラクターを呼び出します。この処理は、スタック・アンワインド と呼ばれます。 自動オブジェクトは、その作成の逆順で破棄されます。 (自動オブジェクトは、auto または register と宣言されているか、あるいは static または extern と宣言されていない、ローカル・オブジェクトです。自動オブジェクト x は、x が宣言されているブロックのプログラムの終了時に、必ず削除されます。)

例外が、サブオブジェクトまたは配列エレメントを含むオブジェクトの作成中にスローされる場合、デストラクターは、例外がスローされる前に正常に作成されたサブオブジェクトまたは配列エレメントに対してのみ、呼び出されます。ローカル静的オブジェクトに対するデストラクターは、オブジェクトが正常に作成された場合にのみ呼び出されます。

スタック・アンワインド中にデストラクターが例外をスローし、その例外が処理されない場合、terminate() 関数が呼び出されます。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_terminate() {
  cout << "Call to my_terminate" << endl;
};

struct A {
  A() { cout << "In constructor of A" << endl; }
  ~A() {
    cout << "In destructor of A" << endl;
    throw E("Exception thrown in ~A()");
  }
};

struct B {
  B() { cout << "In constructor of B" << endl; }
  ~B() { cout << "In destructor of B" << endl; }
};

int main() {
  set_terminate(my_terminate);

  try {
    cout << "In try block" << endl;
    A a;
    B b;
    throw("Exception thrown in try block of main()");
  }
  catch (const char* e) {
    cout << "Exception: " << e << endl;
  }
  catch (...) {
    cout << "Some exception caught in main()" << endl;
  }

  cout << "Resume execution of main()" << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate
```

try ブロックでは、2 つの自動オブジェクト、a と b が作成されます。 try ブロックは、型 const char* の例外をスローします。ハンドラー catch (const char* e) が、この例外をキャッチします。 C++ ランタイムは、スタックをアンワインドし、a および b のデストラクターを、それらが構築された逆順で呼び出します。 a のデストラクターが、例外をスローします。プログラムにこの例外を処理できるハンドラーが存在しないので、C++ ランタイムは terminate() を呼び出します。 (関数 terminate() は、set_terminate() に引数として指定した関数を呼び出します。この例では、terminate() は、my_terminate() を呼び出すよう指定されています。)

例外指定

C++ には、特定の関数が、指定されたリストの例外だけをスローするように限定するメカニズムがあります。任意の関数の先頭に例外を指定すると、関数の呼び出し元に対して、その関数が例外の指定に含まれていない例外を直接にも間接にもスローしないことを保証することができます。

例えば、次の関数を考えます。

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

これは、型が unknown_word または bad_grammar である例外オブジェクト、あるいは unknown_word または bad_grammar から派生した型の例外オブジェクトのみをスローすることを明示的に示しています。

例外指定構文

type_id_list は、コンマで区切られた型のリストです。このリストでは、オプションで const または volatile、あるいはその両方で修飾した、非完了型、および非完了型を指すポインターまたは参照 (void を指すポインターを除く) を指定することはできません。例外指定では、型を定義できません。

例外指定のない関数は、すべての例外のスローを認めます。空の *type_id_list* を持つ例外指定を使用する関数、throw() は、例外のスローを許可しません。

例外指定は関数の型の一部ではありません。

例外指定は、関数、関数を指すポインター、関数への参照、メンバー関数宣言を指すポインター、またはメンバー関数定義を指すポインターの関数宣言子の終了にのみ現れます。例外指定を typedef 宣言に入れることはできません。次の宣言は、このことを示しています。

```
void f() throw(int);
void (*g)() throw(int);
void h(void i() throw(int));
// typedef int (*j)() throw(int);  This is an error.
```

コンパイラーは、最後の宣言、typedef int (*j)() throw(int) を許可しません。

クラス A が、関数の例外指定の *type_id_list* に入っている型の一つだとします。その関数は、クラス A またはクラス A から public に派生したクラスの例外オブジェクトをスローします。次の例は、このことを示しています。

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

void f(int i) throw (A) {
   switch (i) {
      case 0: throw A();
      case 1: throw B();
      default: throw C();
   }
}

void g(int i) throw (A*) {
   A* a = new A();
   B* b = new B();
   C* c = new C();
   switch (i) {
      case 0: throw a;
      case 1: throw b;
      default: throw c;
   }
}
```

関数 f() は、型 A または B のオブジェクトをスローできます。関数が型 C のオブジェクトをスローしようとする場合、コンパイラーは、C が関数の例外指定に指定されてもいないし、それが A から public に派生したものでもないので、unexpected() を呼び出します。同様に、関数 g() は、型 C のオブジェクトを指すポインターをスローできません。関数は、型 A のポインター、または A から public に派生するオブジェクトのポインターをスローします。

仮想関数をオーバーライドする関数は、その仮想関数が指定する例外のみをスローすることができます。次の例は、このことを示しています。

```
class A {
   public:
      virtual void f() throw (int, char);
};

class B : public A{
   public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
   class C : public A {
      public: void f() { }
   };

   class D : public A {
      public: void f() throw (int, char, double) { }
   };
*/
```

コンパイラーは、メンバー関数が、型 int の例外のみをスローするので、B::f() を認めます。コンパイラーは、メンバー関数が、どの種類の例外もスローするので、C::f() を許可しません。コンパイラーは、メンバー関数が、A::f() よりも多くの型の例外 (int、char、および double) をスローするので、D::f() を許可しません。

x という名前の関数を指すポインター、または y という名前の関数を指すポインターの割り当て、または初期化を行うとします。関数 x を指すポインターは、y の例外指定が指定する例外のみをスローできます。次の例は、このことを示しています。

```
void (*f)();
void (*g)();
void (*h)() throw (int);

void i() {
   f = h;
//   h = g;  This is an error.
}
```

コンパイラーは、f がどのような種類の例外もスローできるので、割り当て f = h を認めます。 g は、どのような種類の例外もスローできますが、h が、型 int のオブジェクトしかスローできないので、コンパイラーは、割り当て h = g を許可しません。

暗黙的に宣言された特殊メンバー関数 (デフォルトのコンストラクター、コピー・コンストラクター、デストラクター、およびコピー代入演算子) には、例外指定があります。暗黙的に宣言された特殊メンバー関数の例外指定の中には、その特殊関数が起動する対象の関数の例外指定の中で宣言されている型が含められます。特別な関数を起動する関数が、例外をすべて許可する場合は、特別な関数も例外をすべて許可します。特別な関数が呼び出す関数のすべてが、例外を許可しない場合は、特別な関数も例外を許可しません。次の例は、このことを示しています。

```
class A {
   public:
      A() throw (int);
      A(const A&) throw (float);
      ~A() throw();
};

class B {
   public:
      B() throw (char);
      B(const A&);
      ~B() throw();
};

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

上記の例で示した次の特別な関数は、暗黙的に宣言されています。

```
C::C() throw (int, char);
C::C(const C&);   // Can throw any type of exception, including float
C::~C() throw();
```

デフォルトの C のコンストラクターは、型 int または char の例外をスローできます。 C のコピー・コンストラクターは、どのような種類の例外もスローできます。 C のデストラクターは、いかなる例外もスローできません。

関連情報

- 36 ページの『不完全型』
- 169 ページの『関数宣言および関数定義』
- 195 ページの『関数へのポインター』
- 277 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

特殊例外処理関数

スローされたすべてのエラーが catch ブロックによってキャッチされ、正常に処理されるというわけではありません。ある状況においては、例外を処理する最良の方法は、プログラムを終了することです。 C++ には、catch ブロックによって正しく処理できない例外、または有効な try ブロックの外部にスローされる例外の処理のために、 2 つの特殊なライブラリー関数がインプリメントされています。自動的に実行される機能には次のものがあります。

- unexpected() 関数
- terminate() 関数

unexpected() 関数

例外指定を持つ関数が、例外指定にリストされていない例外をスローすると、C++ ランタイムは、以下を行います。

1. `unexpected()` 関数が呼び出されます。
2. `unexpected()` 関数は、`unexpected_handler` によって指定された関数を呼び出します。デフォルトでは、`unexpected_handler` は、関数 `terminate()` を指します。

`unexpected_handler` のデフォルト値を、関数 `set_unexpected()` を使用して置き換えることができます。

`unexpected()` は、リターンできませんが、例外をスロー (または再スロー) することはできます。関数 `f()` の例外指定が、違反されていたとします。 `unexpected()` が `f()` の例外指定で許可された例外をスローする場合は、C++ ランタイムは、`f()` の呼び出しで別のハンドラーを検索します。次の例は、このことを示しています。

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_unexpected() {
  cout << "Call to my_unexpected" << endl;
  throw E("Exception thrown from my_unexpected");
}

void f() throw(E) {
  cout << "In function f(), throw const char* object" << endl;
  throw("Exception, type const char*, thrown from f()");
}

int main() {
  set_unexpected(my_unexpected);
  try {
    f();
  }
  catch (E& e) {
    cout << "Exception in main(): " << e.message << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In function f(), throw const char* object
Call to my_unexpected
Exception in main(): Exception thrown from my_unexpected
```

main() 関数の try ブロックは、関数 f() を呼び出します。関数 f() は、型 const char* のオブジェクトをスローします。しかし、f() の例外指定は、型 E のオブジェクトのみをスローすることを許可します。関数 unexpected() が呼び出されます。関数 unexpected() は、my_unexpected() を呼び出します。関数 my_unexpected() は、型 E のオブジェクトをスローします。 unexpected() は、f() の例外指定で許可されたオブジェクトをスローするので、 main() 関数にあるハンドラーは、その例外を処理できます。

unexpected() が、f() の例外指定で許可されたオブジェクトをスロー (または再スロー) しない場合は、C++ ランタイムは、2 つのことを行います。

- f() の例外指定にクラス std::bad_exception がある場合、unexpected() は、型 std::bad_exception のオブジェクトをスローし、C++ ランタイムは、f() の呼び出しで別のハンドラーを検索します。
- f() の例外指定にクラス std::bad_exception がない場合、関数 terminate() が呼び出されます。

関連情報

- 345 ページの『特殊例外処理関数』
- 347 ページの『set_unexpected() 関数および set_terminate() 関数』

terminate() 関数

例外処理メカニズムが正しく機能せず、void terminate() の呼び出しが起きるケースもあります。この terminate() の呼び出しは、次のいずれかの状況で行われます。

- 例外処理メカニズムが、スローされた例外のハンドラーを検出できません。以下に、上記のさらに詳しい事例を示します。
  - スタック・アンワインド中に、デストラクターが例外をスローし、その例外が処理されません。
  - スローされた例外が、また例外をスローし、その例外が処理されません。
  - 非ローカル静的オブジェクトのコンストラクターまたはデストラクターが例外をスローし、その例外が処理されません。
  - atexit() で登録された関数が例外をスローし、その例外が処理されません。以下に、このことを示します。

```
extern "C" printf(char* ...);
#include <exception>
#include <cstdlib>
using namespace std;

void f() {
  printf("Function f()¥n");
  throw "Exception thrown from f()";
}

void g() { printf("Function g()¥n"); }
void h() { printf("Function h()¥n"); }

void my_terminate() {
  printf("Call to my_terminate¥n");
  abort();
}

int main() {
  set_terminate(my_terminate);
  atexit(f);
  atexit(g);
  atexit(h);
  printf("In main¥n");
}
```

上記の例の出力は、以下のとおりです。

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

atexit() で関数を登録するには、登録したい関数を指すポインターに、atexit() へのパラメーターを渡します。標準プログラム終了処理で atexit() は、引数のない登録済み関数を逆順で呼び出します。 atexit() 関数は、<cstdlib> ライブラリーに入っています。

- オペランドを指定しない throw 式が、例外を再びスローしようとし、現在処理されている例外がありません。
- 関数 f() が、その例外指定に違反する例外をスローします。 unexpected() 関数が、f() の例外指定に違反する例外をスローし、f() の例外指定が、クラス std::bad_exception を含んでいませんでした。
- unexpected_handler のデフォルト値が呼び出されます。

terminate() 関数は、terminate_handler 関数によって指定された関数を呼び出します。デフォルトで terminate_handler は、プログラムから終了する関数 abort() を指します。 terminate_handler のデフォルト値を、関数 set_terminate() に置き換えることができます。

終了関数は、return を使用するか例外をスローすることによって呼び出し元に戻ることはできません。

関連情報

- 『set_unexpected() 関数および set_terminate() 関数』

set_unexpected() 関数および set_terminate() 関数

関数 unexpected() は、起動されたときに、set_unexpected() に、最後に引数として渡された関数を呼び出します。 set_unexpected() がまだ呼び出されていない場合、unexpected() は terminate() を呼び出します。

関数 terminate() は、起動されると、set_terminate() に、一番最近に引数として供給された関数を呼び出します。 set_terminate() がまだ呼び出されていない場合、terminate() は abort() を呼び出し、これによってプログラムが終了します。

set_unexpected() および set_terminate() を使用して、unexpected() および terminate() によって呼び出される、ユーザー定義関数を登録することができます。 set_unexpected() と set_terminate() は、標準ヘッダー・ファイルに入っています。これらの各関数は、その戻りの型およびその引数の型として、戻りの型が void で引数なしの関数を指すポインターを持っています。引数として提供する関数を指すポインターは、対応する特殊な関数によって呼び出される関数になります。 set_unexpected() への引数が unexpected() によって呼び出される関数になり、set_terminate() への引数が terminate() によって呼び出される関数になります。

set_unexpected() および set_terminate() は、前にそれぞれの特殊な関数 (unexpected() および terminate()) によって呼び出された関数を指すポインターを戻します。戻り値を保管することによって、後で元の特殊な関数を復元して、unexpected() と terminate() が、再び terminate() と abort() を呼び出すようにすることができます。

set_terminate() を使用して、ユーザー自身の関数を登録する場合、関数は、呼び出し元にはリターンせず、プログラムの実行を終了する必要があります。

例外処理関数を使用した例

次の例は、制御の流れと、例外処理で使用する特殊な関数を示しています。

```
#include <iostream>
#include <exception>
using namespace std;

class X { };
class Y { };
class A { };

// pfv type is pointer to function returning void
typedef void (*pfv)();

void my_terminate() {
  cout << "Call to my terminate" << endl;
  abort();
}

void my_unexpected() {
  cout << "Call to my_unexpected()" << endl;
  throw;
}

void f() throw(X,Y, bad_exception) {
  throw A();
}

void g() throw(X,Y) {
  throw A();
}

int main()
{
  pfv old_term = set_terminate(my_terminate);
  pfv old_unex = set_unexpected(my_unexpected);
  try {
    cout << "In first try block" << endl;
    f();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e1) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
    cout << "Caught some exception" << endl;
  }

  cout << endl;

  try {
    cout << "In second try block" << endl;
    g();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e2) {
    cout << "Caught bad_exception" << endl;
  }
```

```
  catch (...) {
    cout << "Caught some exception" << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In first try block
Call to my_unexpected()
Caught bad_exception

In second try block
Call to my_unexpected()
Call to my terminate
```

実行時に、このプログラムは次のように振る舞います。

1. set_terminate() を呼び出すと、old_term に、 set_terminate() が前に呼び出されたときに最後に set_terminate() に渡された関数のアドレスが割り当てられます。
2. set_unexpected() を呼び出すと、old_unex に、 set_unexpected() が前に呼び出されたときに、最後に set_unexpected() に渡された関数のアドレスが割り当てられます。
3. 最初の try ブロックの中では、関数 f() が呼び出されます。 f() が、予期しない例外をスローするので、unexpected() への呼び出しが行われます。次に、unexpected() は、my_unexpected() を呼び出し、標準出力に対してメッセージを印刷します。関数 my_unexpected() は、型 A の例外を再びスローしようとします。クラス A が、関数 f() の例外指定で指定されていないので、 my_unexpected() は、型 bad_exception の例外をスローします。
4. bad_exception が、関数 f() の例外指定で指定されているので、ハンドラー catch (bad_exception& e1) は、例外を処理できます。
5. 2 番目の try ブロックの中では、関数 g() が呼び出されます。 g() が、予期しない例外をスローするので、unexpected() への呼び出しが行われます。 unexpected() は、型 bad_exception の例外をスローします。 bad_exception は、g() の例外指定に指定されていないので、 unexpected() は、terminate() を呼び出し、これが関数 my_terminate() を呼び出します。
6. my_terminate() は、メッセージを表示してから、プログラムを終了する abort() を呼び出します。

例外が、my_unexpected() によって、有効な例外としてではなく、予期しないスロー (throw) として処理されたので、2 番目の try ブロックに続く catch ブロックには、入らないことに留意してください。

第 17 章 プリプロセッサー・ディレクティブ

プリプロセッサーは、コンパイルの前にコードを処理するために、コンパイラーによって起動されるプログラムです。このプログラムのためのコマンドはディレクティブ と呼ばれ、ソース・ファイル内の、# 文字で始まる行がこれにあたります。この文字により、このようなコマンド行とソース・プログラムのテキストが区別されます。各プリプロセッサー・ディレクティブを使用すると、ソース・コードのテキストに変更され、その結果、ディレクティブを含まない新しいソース・コード・ファイルが生成されます。プリプロセスされたソース・コードは中間ファイルですが、これがコンパイラーへの入力になるので、このソース・コードは有効な C または C++ プログラムでなければなりません。

プリプロセッサー・ディレクティブは次のもので構成されます。

- マクロ定義ディレクティブ。現在のファイル内のトークンを指定された置換トークンと置き換える。
- ファイル・インクルード・ディレクティブ。現在のファイル内にファイルを組み込む。
- 条件付きコンパイル・ディレクティブ。現在のファイルのセクションを条件によりコンパイルする。
- メッセージ生成ディレクティブ。診断メッセージの生成を制御する。
- C++ アサーション・ディレクティブ。プログラムを実行するシステムの属性を指定する。
- ヌル・ディレクティブ (#)。アクションを実行しない。
- プラグマ・ディレクティブ。コンパイラー特有の規則を、コードの指定されたセクションに適用する。

プリプロセッサー・ディレクティブは、# トークンで始まり、その後にプリプロセッサー・キーワードが続きます。# トークンは、空白でない行の先頭文字として存在しなければなりません。# はディレクティブ名の一部ではなく、空白で名前から分離することができます。

行の最後の文字が ¥ (円記号) 文字でない限り、プリプロセッサー・ディレクティブは改行文字で終了します。¥ 文字がプリプロセッサー行の最後の文字として現れると、プリプロセッサーは ¥ と改行文字を継続マーク文字として解釈します。プリプロセッサーは、¥ (およびそれに続く改行文字) を削除して、物理ソース行を継続する論理行に継ぎます。円記号と、行末文字またはレコードの物理的な終わりとの間に、空白文字があっても構いません。ただし、通常、この空白文字は編集の際には見えません。

一部の #pragma ディレクティブを除いて、プリプロセッサー・ディレクティブはプログラム内の任意の場所に入れることができます。

マクロ定義ディレクティブ

マクロ定義ディレクティブには、次のディレクティブと演算子が含まれます。

- #define ディレクティブ。マクロを定義します
- #undef dディレクティブ。マクロ定義を除去します

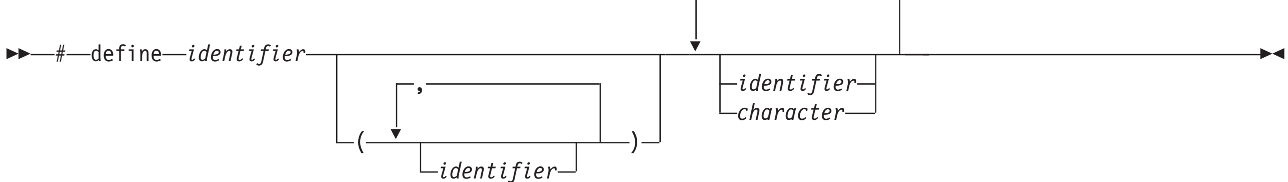
標準事前定義マクロと IBM i 用に事前定義されたマクロについては、「*ILE C/C++ コンパイラー参照*」にある『事前定義マクロ』に説明されています。

#define ディレクティブ

プリプロセッサー定義ディレクティブ は、これ以降のマクロの出現を、指定された置換トークンに置き換えるようプリプロセッサーに指示します。

© Copyright IBM Corp. 1998, 2010

#define ディレクティブの構文



#define ディレクティブは次のものを含むことができます。

- オブジェクト類似マクロ
- 関数類似マクロ

#define と const 型修飾子との違いをいくつか次に示します。

- #define ディレクティブは、数値、文字、あるいはストリング定数の名前の作成に使用できるのに対し、const オブジェクトの場合は、任意の型を宣言することができます。
- const オブジェクトには変数に対するスコープ規則が適用されるのに対し、#define を使用して作成された定数には適用されません。
- const オブジェクトと違って、マクロはインラインで展開されるので、マクロの値は、コンパイラーが使用する中間ソース・コードには含まれません。インライン展開をすると、デバッガーはマクロ値を使用できなくなります。
- マクロは、バインドされた配列などの定数式の中で使用できますが、 const オブジェクトはできません。
- C++ コンパイラーは、マクロ引数を含めて、マクロの型検査は行いません。

関連情報

- 65 ページの『const 型修飾子』

オブジェクト類似マクロ

オブジェクト類似マクロ定義 は、単一の ID を指定された置換トークンに置き換えます。以下のオブジェクト類似の定義を使用すると、プリプロセッサーは、ID COUNT のこれ以降のすべてのインスタンスを、定数 1000 に置き換えます。

```
#define COUNT 1000
```

次のステートメント

```
int arry[COUNT];
```

が、この定義の後、かつ定義と同じファイル内に現れると、プリプロセッサーは、このステートメントをプリプロセッサーの出力で、以下のステートメントのように変更します。

```
int arry[1000];
```

他の定義が ID COUNT を参照することができます。

```
#define MAX_COUNT COUNT + 100
```

プリプロセッサーは、MAX_COUNT のこれ以降の出現を COUNT + 100 に置き換えます。これを、プリプロセッサーは、さらに 1000 + 100 に置き換えます。

マクロ展開によって部分的に構築された番号が作成された場合、プリプロセッサーは、その結果を単一の値であるとは見なしません。例えば、以下の結果は 10.2 という値にはならず、構文エラーになります。

```
#define a 10
a.2
```

マクロ展開によって部分的に構築される ID は、作成されない場合があります。したがって、以下の例は、2 つの ID を含んでいて、結果は構文エラーとなります。

```
#define d efg
abcd
```

関数類似マクロ

関数類似マクロは、オブジェクト類似マクロより複雑であって、その定義は、仮パラメーターの名前をコンマで区切って、括弧で囲んで宣言します。空の仮パラメーター・リストは有効です。そのようなマクロを使って、引数を取らない関数をシミュレートすることができます。 C99 では、数が可変の引数を持つ関数類似マクロがサポートされています。ILE C++ は、可変個の引数を持つ関数類似マクロを、C との互換性のための言語拡張機能としてサポートします。

関数類似マクロの定義

小括弧に囲まれたパラメーター・リストおよび置換トークンが後ろに続く ID。パラメーターを置換コード内に組み込みます。空白文字で、ID (マクロの名前) とパラメーター・リストの左括弧とを分離することはできません。コンマで各パラメーターを区切る必要があります。

移植性のため、1 つのマクロには、パラメーターが 31 を超えないようにする必要があります。パラメーター・リストは、省略符号 (...) で終了することができます。 この場合、ID __VA_ARGS__ を置換リストに挿入することができます。

関数類似マクロの呼び出し

小括弧に入れられた、コンマで区切られた引数のリストが後に続く ID。引数の数は、定義内のパラメーター・リストが省略記号で終了していない限り、マクロ定義内のパラメーターの数に一致している必要があります。定義内のパラメーター・リストが省略符号で終了する場合、マクロ呼び出しでの引数の数は、定義内のパラメーターの数を超えるはずです。 この過剰分の引数は*後続引数*と呼ばれます。プリプロセッサーは、関数類似マクロの呼び出しを確認すると、引数の置換を行います。置換コード内のパラメーターは、対応する引数に置き換えられます。後続の引数がマクロ定義によって許可されている場合は、それらの引数は、その間にコンマを挟んでマージされ、それら後続引数があたかも 1 つの引数であるかのように、__VA_ARGS__ で置き換えられます。引数自体に含まれるマクロの呼び出しはすべて、引数が置換コード内の対応するパラメーターと置き換わる前に、完全に置き換えられます。

ID リストが省略符号で終了していない場合、マクロ呼び出しの中の引数の数は、対応するマクロ定義の中のパラメーターの数と同じでなければなりません。パラメーターの置換時、指定されたすべての引数 (区切り文字のコンマも含む) が置換された後に残っている引数は、変数引数と呼ばれる 1 つの引数にまとめられます。変数引数は、置換リストの中の ID __VA_ARGS__ のすべてのオカレンスを置き換えます。次の例は、このことを示しています。

```
#define debug(...)   fprintf(stderr, __VA_ARGS__)

debug("flag");     /*   Becomes fprintf(stderr, "flag");   */
```

マクロ呼び出し引数リストにおけるコンマは、以下の場合には、分離文字として作用しません。

- 文字定数内にある。
- ストリング・リテラル内にある。
- 小括弧で囲まれている。

以下の行は、a と b という 2 つのパラメーターと置換トークン (a + b) を持つものとしてマクロ SUM を定義します。

```
#define SUM(a,b) (a + b)
```

この定義により、プリプロセッサーは以下のステートメントを変更することになります (そのステートメントが前の定義の後に現れる場合)。

```
c = SUM(x,y);
c = d * SUM(x,y);
```

プリプロセッサーの出力においては、これらのステートメントは次のように表示されます。

```
c = (x + y);
c = d * (x + y);
```

置換テキストが正しく評価されるようにするためには、小括弧を使用してください。例えば、

```
#define SQR(c)  ((c) * (c))
```

上記の定義では、定義内の各パラメーター c の周りに小括弧を必要とします。以下のような表現も正しく評価することができます。

```
y = SQR(a + b);
```

プリプロセッサーはこのステートメントを次のように展開します。

```
y = ((a + b) * (a + b));
```

定義内に小括弧がないと、プリプロセッサーは評価の正しい順序を保てず、プリプロセッサーの出力は次のようになります。

```
y = (a + b * a + b);
```

および ## 演算子の引数は、関数類似マクロのパラメーターの置換の*前に* 変換されます。

プリプロセッサー ID は、いったん定義されると定義されたままとなり、言語のスコープ決定規則とは関係なく、有効となります。マクロ定義のスコープは定義から始まり、対応する #undef ディレクティブに遭遇するまで終了しません。対応する #undef ディレクティブがない場合、そのマクロ定義のスコープは、変換単位の終わりまで続きます。

再帰マクロは、完全には展開されません。例えば、以下の定義

```
   #define x(a,b) x(a+1,b+1) + 4
```

は、

```
   x(20,10)
```

を、以下のように展開します。

```
   x(20+1,10+1) + 4
```

マクロ x を、それ自体の中で繰り返し展開しようとするよりも、上述の展開を行います。マクロ x が展開された後で、そのマクロは、関数 x() の呼び出しとなります。

置換トークンを指定するのに、定義は必須ではありません。以下の定義は、現在のファイル内のこれ以降の行から、トークン debug のすべてのインスタンスを除去します。

```
#define debug
```

2 番目のプリプロセッサー #define ディレクティブを用いて、定義済みの ID またはマクロの定義を変更することができます。ただし、2 番目のプリプロセッサー #define ディレクティブの前に、プリプロセッサー #undef ディレクティブがある場合に限ります。#undef ディレクティブは、最初の定義を無効にして、同じ ID を再定義で使用できるようにします。

プログラムのテキストの中については、プリプロセッサーはマクロ呼び出しのための文字定数またはストリング定数のスキャンを行いません。

次のプログラム例には、2 つのマクロ定義と、定義されている両方のマクロを参照するマクロ呼び出しが含まれています。

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d¥n", a); ¥
  printf("value 2 = %d¥n", b) ;

int main(void)
{
  int x = 2;
  int y = 3;

     PRNT(SQR(x),y);

   return(0);
}
```

プリプロセッサーによって解釈された後、このプログラムは、以下のものに等価のコードによって置き換えられます。

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

     printf("value 1 = %d¥n", ( (x) * (x) ) );
     printf("value 2 = %d¥n", y);

   return(0);
}
```

このプログラムの出力は次のようになります。

```
value 1 = 4
value 2 = 3
```

関連情報

- 146 ページの『演算子優先順位と結合順序』
- 107 ページの『括弧で囲んだ式 ( )』

可変個引数マクロ拡張:

C++ のみ。

可変個引数マクロ拡張は、C99 と Standard C++ に対する 2 つの拡張で、可変個の引数を持つマクロに関係するものです。 1 つの拡張は、変数引数 ID を__VA_ARGS__ からユーザー定義 ID に名前変更するためのメカニズムです。もう 1 つの拡張は、変数引数が指定されていないとき可変個引数マクロ内の中ぶらりんのコンマを除去する方法を提供するものです。

以下の例は、__VA_ARGS__ の代わりに ID を使用する方法を示しています。マクロ debug の最初の定義は、__VA_ARGS__ の通常の使用法を示しています。 2 番目の定義は、__VA_ARGS__ の代わりに ID args を使用する方法を示しています。

```
#define debug1(format, ...)  printf(format, ## __VA_ARGS__)
#define debug2(format, args ...)  printf(format, ## args)
```

呼び出し	マクロ展開の結果
debug1("Hello %s\n","World");	printf("Hello %s\n","World");
debug2("Hello %s\n","World");	printf("Hello %s\n","World");

関数マクロへの変数引数が省略されたり空であったりして、かつ関数マクロ定義内でコンマとその後に続く ## が変数引数 ID に先行している場合は、プリプロセッサーが末尾のコンマを除去します。

C++ のみ。 の終り

#undef dディレクティブ

プリプロセッサーの undef ディレクティブ により、プリプロセッサーはプリプロセッサー定義のスコープを終わらせます。

#undef ディレクティブの構文

```
▶▶─#─undef─identifier──────────────────────────────────────────────▶◀
```

identifier が現在マクロとして定義されていなければ、#undef は無視されます。

以下のディレクティブは BUFFER および SQR を定義します。

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

以下のディレクティブはその定義を無効にします。

```
#undef BUFFER
#undef SQR
```

これらの #undef ディレクティブの後に続いて ID BUFFER および SQR が現れても、それらは、いかなる置換トークンにも置き換えられません。 #undef ディレクティブによってマクロの定義が除去されてしまえば、新しい #define ディレクティブでその ID を使用することができます。

演算子

(単一番号記号) 演算子は、関数類似マクロのパラメーターを文字ストリング・リテラルに変換します。例えば、以下のディレクティブを使用してマクロ ABC が定義される場合、

```
#define ABC(x)   #x
```

これ以降のマクロ ABC の呼び出しはすべて、ABC に渡された引数を含む文字ストリング・リテラルに展開されます。次に例を示します。

呼び出し	マクロ展開の結果
ABC(1)	"1"
ABC(Hello there)	"Hello there"

演算子を、ヌル・ディレクティブと混同してはなりません。

演算子は、下記の規則に従って、関数類似マクロ定義で使用してください。

- 関数類似マクロの中の # 演算子に続くパラメーターは、マクロに渡された引数を含む文字ストリング・リテラルに変換されます。
- プリプロセッサーは、マクロに渡された引数の前または後ろにある空白文字を削除します。
- マクロに渡された引数内に組み込まれた複数の空白文字は、単一のスペース文字に置き換えられます。
- マクロに渡された引数にストリング・リテラルがある場合、およびそのリテラル内に ¥ (円記号) 文字がある場合には、マクロ展開時に、元の ¥ の前に追加の ¥ 文字が挿入されます。
- マクロに渡された引数に " (二重引用符) 文字がある場合、マクロ展開時に、" の前に ¥ 文字が挿入されます。
- 引数のストリング・リテラルへの変換は、その引数でマクロが展開される前に行われます。
- マクロ定義の置換リスト内に複数の ## 演算子または # 演算子がある場合、その演算子の評価の順序は定義されていません。
- マクロ展開の結果が有効な文字ストリング・リテラルでない場合、その振る舞いは予期できません。

以下の例は、# 演算子の使用法を示したものです。

```
#define STR(x)        #x
#define XSTR(x)       STR(x)
#define ONE           1
```

呼び出し	マクロ展開の結果
STR(¥n "¥n" '¥n')	"¥n ¥"¥¥n¥" '¥¥n'"
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"¥"hello¥""

関連情報

- 369 ページの『ヌル・ディレクティブ (#)』

演算子

(二重番号記号) 演算子は、マクロ定義に含まれるマクロの呼び出し (テキストまたは引数、あるいはその両方) における 2 つのトークンを連結します。

以下のディレクティブを使用して、マクロ XY が定義された場合、

```
#define XY(x,y)    x##y
```

x に対する引数の最後のトークンは、y に対する引数の最初のトークンと連結されます。

演算子は、以下の規則に従って使用します。

- ## 演算子を、マクロ定義の置換リスト内の最初の項目または最後の項目にすることはできません。
- ## 演算子の前にある項目の最後のトークンは、 ## 演算子の後ろにある項目の最初のトークンに連結されます。
- 連結は、引数の中のマクロのいずれかが展開される前に行われます。
- 連結の結果が有効なマクロ名になった場合には、たとえその名前が、通常はその中では使用できないコンテキストの中に現れたとしても、その名前を、その後の置換に使用することができます。
- マクロ定義の置換リスト内に複数の ## 演算子、または # 演算子、またはその両方がある場合、その演算子の評価の順序は定義されていません。

以下の例は、## 演算子の使用法を示したものです。

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

呼び出し	マクロ展開の結果
ArgArg(lady, bug)	"ladybug"
ArgText(con)	"conTEXT"
TextArg(book)	"TEXTbook"
TextText	"TEXTtext"
ArgArg(Jitter, bug)	3

ファイル・インクルード・ディレクティブ

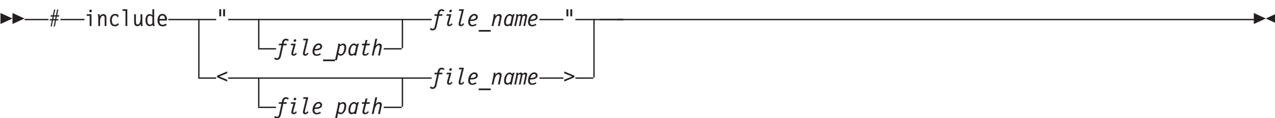
ファイル・インクルード・ディレクティブは次のもので構成されます。

- #include ディレクティブ。別のソース・ファイルからテキストを挿入します。
- 400 #include_next ディレクティブ。インクルード・ファイルの検索時にコンパイラーが検索パスからインクルード・ファイルのディレクトリーを除外するようにします。

#include ディレクティブ

プリプロセッサーの *include ディレクティブ* を使用すると、プリプロセッサーは、そのディレクティブを指定されたファイルの内容に置き換えます。

#include ディレクティブの構文



データ管理ファイル内のソースのコンパイル時における #include ディレクティブの使用

次の表は、コンパイラーがソース物理ファイルのために取る検索パスを示しています。下記のデフォルト・ファイル名と検索パスを参照してください。

表 24. コンパイラーがソース物理ファイルのために取る検索パス

ファイル名	メンバー	ファイル	ライブラリー
mbr	*mbr*	デフォルト・ファイル	デフォルト検索
file/mbr[1]	*mbr*	*file*	デフォルト検索
mbr.file	*mbr*	*file*	デフォルト検索
lib/file/mbr	*mbr*	*file*	*lib*
lib/file(mbr)	*mbr*	*file*	*lib*

注: [1] インクルード・ファイル形式 <file/mbr.h> が使用されると、コンパイラーはまずライブラリー・リスト内のそのファイルで *mbr* を検索します。 *mbr* が見つからないと、コンパイラーはライブラリー・リスト内の同じファイルで *mbr.h* を検索します。メンバー名の拡張子として有効なのは「h」または「H」のみです。

ライブラリーとファイルが指定されていないと、プリプロセッサーは *filename* を囲む区切り文字に応じて特定の検索パスを使用します。 < > 区切り文字は、名前をシステム・インクルード・ファイルとして指定します。 " " 区切り文字は、名前をユーザー・インクルード・ファイルとして指定します。

コンパイラーが使用する #include ディレクティブの検索パスについて次に説明します。

- ライブラリーとファイルが指定されていないときのデフォルト・ファイル名 (メンバー名のみ):

  インクルード・タイプ
  デフォルト・ファイル名

  <> C コンパイラーの場合は QCSRC、C++ コンパイラーの場合は STD

  ” " ルート・ソース・メンバーのソース・ファイル (ここでルート・ソース・メンバーは、モジュール作成コマンドまたはバインド済みプログラム作成コマンドの SRCFILE オプションによって決定されるライブラリー、ファイル、およびメンバーです)

- ファイル名がライブラリー修飾でないときのデフォルト検索パス

  インクルード・タイプ
  検索パス

  <> 現行ライブラリー・リスト (*LIBL) を検索します。

  " " ルート・ソース・メンバーが入っているライブラリーを検査します。そこに存在しないと、コンパイラーは指定されたファイル名またはルート・ソース・メンバーのファイル名 (ファイル名が指定されていない場合) を使用してライブラリー・リストのユーザー部分を検索します。見つからないと、コンパイラーは指定されたファイル名を使用してライブラリー・リスト (*LIBL) を検索します。

- ファイル名がライブラリー修飾であるときの検索パス (lib/file/mbr)

  インクルード・タイプ
  検索パス

  <> lib/file/mbr のみを検索します。

  " " 指定されたライブラリーとファイルでメンバーを検索します。見つからない場合は、指定されたファイル名とメンバー名を使用してライブラリー・リストのユーザー部分を検索します。

モジュール作成コマンドまたはバインド済みプログラム作成コマンドで *SYSINCPATH オプションが指定されていると、ユーザー・インクルードはシステム・インクルードと同じに扱われます。

プリプロセッサーが #include ディレクティブ上のマクロを解決します。マクロ置き換え後に結果として得られるトークン・シーケンスは、二重引用符または文字 < および > で囲まれたファイル名で構成されます。次に例を示します。

```
#define MONTH <july.h>
#include MONTH
```

使用法

いくつかのファイルによって使用される宣言が多数存在する場合は、これらのすべての定義を、これらの定義を使用する各ファイルを保管する 1 つのファイルと #include に入れることができます。例えば、以下のファイル defs.h には、いくつかの定義と、宣言の追加ファイルのインクルードが 1 つ入っています。

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

以下のディレクティブを用いて、defs.h 内にある定義を組み込むことができます。

```
#include "defs.h"
```

次の例には、プリプロセッサー・ディレクティブの使用を結合できる方法の 1 つが示されています。 C または C++ の標準入出力ヘッダー・ファイルの名前を表すマクロを定義するために #define が使用されます。次に、ヘッダー・ファイルを C プログラムまたは C++ プログラムが使用できるようにするために #includeが使用されます。

```
#define IO_HEADER   <stdio.h>
      .
      .
      .
#include IO_HEADER   /* equivalent to specifying #include <stdio.h>  */
      .
      .
      .
```

統合ファイル・システム・ファイル内のソースのコンパイル時における #include ディレクティブの使用

SRCSTMF キーワードを使用してコンパイル時に統合ファイル・システム・ファイルを指定することができます。 #include 処理は、ライブラリー・リストが検索されないという点でソース物理ファイル処理とは異なります。ヘッダー・ファイルを解決するために、INCLUDE 環境変数 (存在する場合) で指定された検索パスとコンパイラーのデフォルト検索パスが使用されます。

コンパイラーのデフォルト・インクルード・パスは /QIBM/include です。

#include ファイルは区切り文字 " " または <> を使用します。

コンパイラーは、インクルード・ファイルのオープンを試みるとき、ファイルが見つかるまで、検索パス内のすべてのディレクトリーを順に検索します。

インクルード・ファイルを検索するアルゴリズムは次のとおりです。

```
if file is fully qualified (a slash / starts the name) then
  attempt to open the fully qualified file
else
  if "" is delimeter, check job's current directory
  if not found:
     loop through the list of directories specified in the INCLUDE
          environment variable and then the default include path
     until the file is found or the end of the include path is encountered
  endif
```

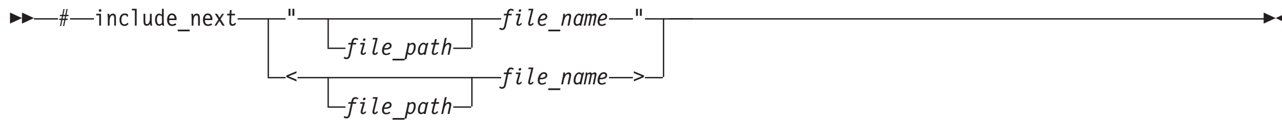
詳しくは、「*IBM Rational Development Studio for i: ILE C/C++ プログラマーの手引き*」にある『*IBM i 統合ファイル・システムでの ILE C/C++ ストリーム機能の使用*』を参照してください。

#include_next ディレクティブ

IBM 拡張

プリプロセッサー・ディレクティブ #include_next の動作は、指定されたファイルを検索するパスからインクルード・ファイルのディレクトリーを特に除外することを除けば、#include ディレクティブの動作と類似しています。インクルード・ファイルのディレクトリーまでを含むすべての検索パスが、インクルード・ファイルを検索するパスのリストから除外されます。これにより、同じ名前を持つファイルの複数のバージョンをアプリケーションのさまざまな部分に組み込んだり、1 つのヘッダー・ファイルを同じ名前の別のヘッダー・ファイルに (ヘッダー自体を再帰的に組み込むヘッダーなしで) 組み込んだりすることができます。異なるファイル・バージョンが異なるディレクトリーに保管されていると、このディレクティブにより、絶対パスを使用してファイル名を指定しなくてもファイルの各バージョンにアクセスできるようになります。

#include_next ディレクティブの構文



このディレクティブはヘッダー・ファイルの中でしか使用できず、*file_name* で指定するファイルはヘッダー・ファイルでなければなりません。ファイル名を囲むのに二重引用符の使用と不等号括弧の使用は区別がありません。

#include_next ディレクティブを使用した検索パスの解決の例として、2 つのバージョンを持つファイル t.h が存在するものとします。1 つ目のバージョンはソース・ファイル t.c に組み込まれ、サブディレクトリー path1 に入っています。2 つ目のバージョンは 1 つ目のバージョンに組み込まれ、サブディレクトリー path2 に入っています。 t.c がコンパイルされると、両方のディレクトリーがインクルード・ファイルの検索パスとして指定されます。

```
/* t.c  */

#include "t.h"

int main()
{
printf("%d", ret_val);
}

/* t.h in path1 */

#include_next "t.h"
```

```
int ret_val = RET;

/* t.h in path2 */

#define RET 55;
```

#include_next ディレクティブは、プリプロセッサーに対して、path1 ディレクトリーをスキップして、path2 ディレクトリーからインクルード・ファイルの検索を開始するように指示します。このディレクティブにより、t.h の 2 つの異なるバージョンを使用でき、t.h が再帰的に組み込まれないようにすることができます。

IBM 拡張 の終り

条件付きコンパイル・ディレクティブ

プリプロセッサーの条件付きコンパイル・ディレクティブ を使用すると、プリプロセッサーは、ソース・コードのコンパイルの一部を条件付きで抑止します。これらのディレクティブは、定数式または ID をテストして、プリプロセッサーがコンパイラーに渡すべきトークン、およびプリプロセス時にう回すべきトークンを判別します。この条件付きディレクティブには、次のものがあります。

- #if および #elif ディレクティブ。定数式の結果に基づいて、ソース・コードの部分を条件によりインクルードまたは抑止します。
- #ifdef ディレクティブ。マクロ名が定義されている場合に、ソース・テキストを条件によりインクルードします。
- #ifndef ディレクティブ。マクロ名が定義されていない場合に、ソース・テキストを条件によりインクルードします。
- #else ディレクティブ。直前の #if、#ifdef、#ifndef、または #elif テストが失敗した場合に、条件によりソース・テキストをインクルードします。
- #endif ディレクティブ。条件付きテキストを終了します。

プリプロセッサーの条件付きコンパイル・ディレクティブは、下記のいくつかの行に及びます。

- 条件指定行 (#if、#ifdef、または #ifndef で始まる)
- 条件の評価が非ゼロ値になった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- #elif 行 (オプション)
- 条件の評価が非ゼロ値になった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- #else 行 (オプション)
- 条件の評価がゼロになった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- プリプロセッサーの #endif ディレクティブ

#if、#ifdef、および #ifndef の各ディレクティブのそれぞれに対して、ゼロまたは複数の #elif ディレクティブ、ゼロまたは 1 つの #else ディレクティブ、および一致する 1 つの #endif ディレクティブがあります。一致するディレクティブは、すべて同じネスト・レベルにあるものと見なします。

条件付きコンパイル・ディレクティブをネストすることができます。以下のディレクティブでは、最初の #else は、#if ディレクティブに突き合わせられます。

```
#ifdef MACNAME
                 /*  tokens added if MACNAME is defined */
#   if TEST <=10
                 /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                 /* tokens added if MACNAME is defined and TEST >  10 */
#   endif
#else
                 /*  tokens added if MACNAME is not defined */
#endif
```

各ディレクティブは、その直後のブロックを制御します。ブロックは、ディレクティブの後の行から始まって、同じネスト・レベルにある次の条件付きコンパイル・ディレクティブで終了する、すべてのトークンで構成されます。

各ディレクティブは、検出された順序で処理されます。式の評価がゼロの場合、ディレクティブの後に続くブロックは無視されます。

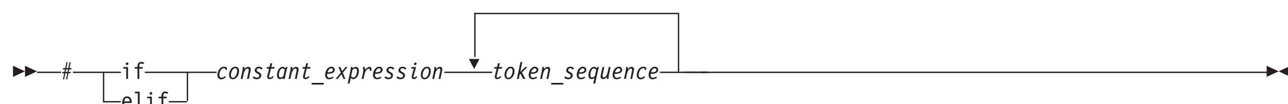
プリプロセッサー・ディレクティブの後に続くブロックを無視することになっているとき、条件付きネスト・レベルが判別できるように、そのブロック内のプリプロセッサー・ディレクティブを識別するだけのために、トークンが検査されます。ディレクティブの名前以外のトークンは、すべて無視されます。

式が非ゼロとなる最初のブロックのみを処理します。そのネスト・レベルにある残りのブロックは無視します。そのネスト・レベルにあるブロックのどれも処理されていなくて、`#else` ディレクティブがある場合、`#else` ディレクティブに続くブロックが処理されます。そのネスト・レベルにあるブロックのどれも処理されていなくて、`#else` ディレクティブがない場合、ネスト・レベル全体が無視されます。

#if および #elif ディレクティブ

`#if` および `#elif` ディレクティブは、 *constant_expression* の値をゼロと比較します。

#if ディレクティブと #elif ディレクティブの構文



定数式が非ゼロ値に評価される場合、条件の直後にあるコードの行をコンパイラーに渡します。

式がゼロに評価され、条件付きコンパイル・ディレクティブが、プリプロセッサー `#elif` ディレクティブを含んでいる場合、`#elif` および次の `#elif` またはプリプロセッサー `#else` ディレクティブとの間にあるソース・テキストが、プリプロセッサーによって選択され、コンパイラーに渡されます。 `#elif` ディレクティブをプリプロセッサーの `#else` ディレクティブの後ろに入れることはできません。

すべてのマクロが展開され、`defined()` の式はすべて処理され、残りのすべての ID は、トークン `0` に置き換えられます。

テストされる *constant_expression* は、以下の属性を持つ整定数式でなければなりません。

- キャストは実行されません。
- `long int` 値を使用して演算を実行します。
- 定義済みマクロを *constant_expression* に入れることはできます。それ以外の ID は式の中には入れられません。

- *constant_expression* には、単項演算子 defined を入れることができます。この演算子は、プリプロセッサー・キーワードの #if または #elif を用いた場合にのみ使用できます。以下の式の評価は、プリプロセッサーに *identifier (ID)* が定義されている場合は、1 に、それ以外の場合は、0 になります。

  ```
  defined identifier
  defined(identifier)
  ```

  次に例を示します。

  ```
  #if defined(TEST1) || defined(TEST2)
  ```

注: マクロが定義されていない場合、0 (ゼロ) の値がそれに代入されます。以下の例では、TEST がマクロ ID であることが必要です。

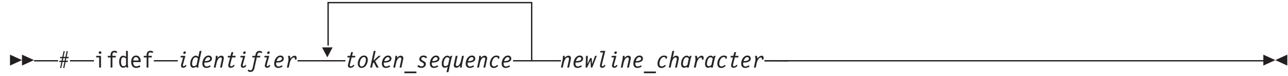
```
#if TEST >= 1
   printf("i = %d¥n", i);
   printf("array[i] = %d¥n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds ¥n");
#endif
```

#ifdef ディレクティブ

#ifdef ディレクティブは、マクロ定義の存在を検査します。

指定された ID がマクロとして定義されている場合、条件の直後にあるコードの行がコンパイラーに渡されます。

#ifdef ディレクティブの構文



以下の例は、プリプロセッサーに対して EXTENDED が定義されている場合に、MAX_LEN を 75 として定義します。定義されていない場合には、MAX_LEN を 50 として定義します。

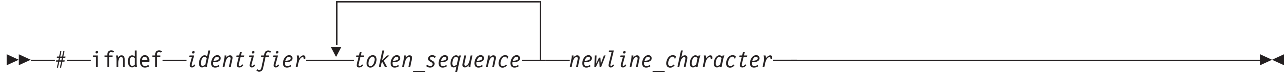
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef ディレクティブ

#ifndef ディレクティブは、マクロが定義されていないかどうかをチェックします。

指定された ID がマクロとして定義されていない場合、条件の直後にあるコードの行がコンパイラーに渡されます。

#ifndef ディレクティブの構文



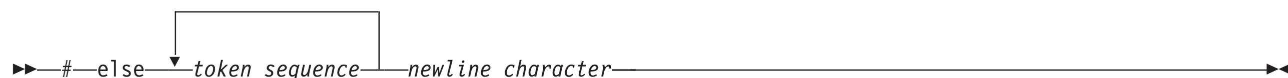
ID は、#ifndef キーワードの後に続いていなければなりません。以下の例は、プリプロセッサーに対して EXTENDED が定義されていない場合に、MAX_LEN を 50 として定義します。定義されていない場合、MAX_LEN を 75 として定義します。

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

#else ディレクティブ

#if、#ifdef、または #ifndef ディレクティブで指定された条件が 0 に評価され、条件付きコンパイル・ディレクティブが、プリプロセッサー #else ディレクティブを含んでいる場合、プリプロセッサー #else ディレクティブおよびプリプロセッサー #endif ディレクティブとの間にあるコードの行が、プリプロセッサーによって選択され、コンパイラーに渡されます。

#else ディレクティブの構文



#endif ディレクティブ

プリプロセッサー・ディレクティブの #endif は、条件付きコンパイル・ディレクティブを終了します。

#endif ディレクティブの構文

```
►►─#─endif─newline_character──────────────────────────────────►◄
```

条件付きコンパイル・ディレクティブの例

以下の例は、プリプロセッサーの条件付きコンパイル・ディレクティブをどのようにネストできるかを示しています。

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

以下のプログラムには、プリプロセッサーの条件付きコンパイル・ディレクティブが含まれています。

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>
```

```
int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d¥n", i);
   printf("array[i] = %d¥n",
   array[i]);
#endif

   }
   return(0);
}
```

メッセージ生成ディレクティブ

メッセージ生成ディレクティブには次のものが含まれています。

- #error ディレクティブ。コンパイル時エラー・メッセージ用のテキストを定義します。
- #warning ディレクティブ。コンパイル時警告メッセージ用のテキストを定義します。
- #line ディレクティブ。コンパイラー・メッセージの行番号を提供します。

関連情報

- 362 ページの『条件付きコンパイル・ディレクティブ』

#error ディレクティブ

プリプロセッサーの error ディレクティブ を使用すると、プリプロセッサーはエラー・メッセージを生成して、コンパイルを失敗させます。

#error ディレクティブの構文

`#error` ディレクティブは、コンパイル時の安全チェックとして、`#if-#elif-#else` 構造の `#else` 部でよく使用されます。例えば、`#error` ディレクティブをソース・ファイルで使用すると、プログラムのバイパスすべき部分に到達したら、コードが生成されないようにすることできます。

例えば、以下のディレクティブ

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

は、次のエラー・メッセージを生成します。

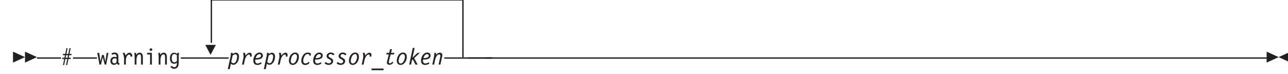
```
BUFFER_SIZE is too small.
```

#warning ディレクティブ

C++ のみ。

プリプロセッサーの *warning ディレクティブ* を使用すると、プリプロセッサーは警告メッセージを生成します。ただし、コンパイルは続行します。#warning に対する引数は、マクロ展開されません。

#warning ディレクティブの構文



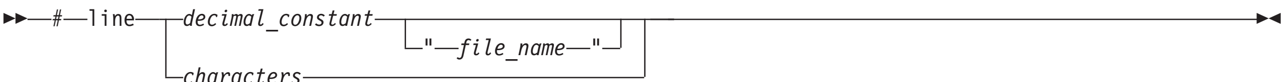
プリプロセッサー・ディレクティブ #warning は言語拡張です。インプリメンテーションにより複数の空白文字が保持されます。

C++ のみ。 の終り

#line ディレクティブ

プリプロセッサーの行制御ディレクティブ は、コンパイラー・メッセージに対して行番号を提供します。このディレクティブにより、コンパイラーは、次のソース行の行番号を指定された番号として表示します。

#line ディレクティブの構文



コンパイラーがプリプロセスされたソース内の行番号への参照をわかりやすく行えるようにするため、プリプロセッサーは必要な個所に (例えば、含まれているテキストの始めまたはテキストの終わりの後に)、#line ディレクティブを挿入します。

二重引用符で囲まれたファイル名の指定を行番号の後に続けることができます。ファイル名を指定すると、コンパイラーは指定されたファイルの一部として次の行を表示します。ファイル名を指定しないと、コンパイラーは現行ソース・ファイルの一部として次の行を表示します。

C99 言語レベルでは、#line プリプロセス指令の最大値は 2147483647 です。

すべての C および C++ インプリメンテーションにおいて、#line ディレクティブのトークン・シーケンスは、マクロ置き換えをすることがあります。マクロ置き換え後に結果として得られる文字シーケンスは、10 進定数 (オプションで、二重引用符で囲まれたファイル名が後に続く) で構成されます。

#line 制御ディレクティブを使用して、コンパイラーにもっとわかりやすいエラー・メッセージを提供させることができます。次のプログラム例は、#line 制御ディレクティブを使用して、認識しやすい行番号を各関数に提供します。

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200
```

```
int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d¥n",_ _LINE_ _);
}

#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d¥n",_ _LINE_ _);
}
```

このプログラムの出力は次のようになります。

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

アサーション・ディレクティブ

C++ のみ。

アサーション・ディレクティブはマクロ定義の代替手段で、コンパイルしたプログラムを実行するコンピューターまたはシステムを定義するために使用されます。アサーションは通常は事前定義されていますが、#assert プリプロセッサー・ディレクティブを使用して定義することができます。

#assert ディレクティブの構文

```
►►—#—assert—predicate—(—answer—)———————————————————————►◄
```

predicate は定義するアサーション・エンティティーを表します。 *answer* はアサーションに割り当てる値を表します。同じ述部と異なる応答を使用して複数のアサーションを作成することができます。与えられた任意の述部に対する応答はすべて同時に真です。例えば、以下のディレクティブはフォントのプロパティーに関するアサーションを作成します。

```
#assert font(arial)
#assert font(blue)
```

アサーションが定義されると、アサーション述部を条件付きディレクティブの中で使用して現行システムをテストすることができます。次のディレクティブは、font に対して arial が表明されているか、それとも blueが表明されているかをテストします。

```
#if #font(arial) || #font(blue)
```

条件付きディレクティブの中の応答を除外することによって、述部に対する応答を表明するかどうかをテストすることができます。

```
#if #font
```

#unassert ディレクティブを使用してアサーションを取り消すことができます。 #assert ディレクティブと同じ構文を使用すると、このディレクティブは指定された応答のみを取り消します。例えば、次のディレクティブは font 述部に対する arial 応答を取り消します。

```
#unassert font(arial)
```

#unassert ディレクティブから応答を除外すると、述部全体が取り消されます。次のディレクティブは font ディレクティブを完全に取り消します。

```
#unassert font
```

関連情報

- 362 ページの『条件付きコンパイル・ディレクティブ』

C++ のみ。 の終り

ヌル・ディレクティブ (#)

ヌル・ディレクティブ ではアクションは行われません。このディレクティブは、ディレクティブ自体の行上の単一の # で構成されます。

ヌル・ディレクティブを、# 演算子や、プリプロセッサー・ディレクティブの最初の文字と混同しないようにしてください。

以下の例では、MINVAL が定義済みマクロ名である場合、アクションを行いません。 MINVAL が定義済み ID ではない場合は、1 に定義します。

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

関連情報

- 356 ページの『# 演算子』

プラグマ・ディレクティブ

プラグマ は、コンパイラーに対するインプリメンテーション定義の命令です。一般的な形式は次のとおりです。

#pragma ディレクティブの構文

character_sequence は、特定のコンパイラー・ディレクティブおよび引数 (あれば) を指定する一連の文字です。 プラグマ・ディレクティブは改行 文字で終了する必要があります。

プラグマの *character_sequence* は、マクロ置換を受けることがあります。次に例を示します。

```
#define XX_ISO_DATA isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

1 つのプラグマ・ディレクティブで、複数のプラグマ構成を指定することができます。コンパイラーは、認識されないプラグマを無視します。

_Pragma プリプロセス演算子

C99 フィーチャーである単項演算子 `_Pragma` を使用して、プリプロセッサー・マクロをプラグマ・ディレクティブに含めることができます。

_Pragma 演算子の構文

```
►►—_Pragma—(—"—string_literal—"—)——————————————————————————►◄
```

string_literal の前に `L` を付加して、これをワイド・ストリング・リテラルにできます。

このストリング・リテラルは、ストリング化が解除され、トークン化されます。これにより生成されるトークンのシーケンスは、それがプラグマ・ディレクティブにあるかのように処理されます。次に例を示します。

```
_Pragma ( "pack(full)" )
```

これは、以下と同等です。

```
#pragma pack(full)
```

付録 A. ILE C 言語拡張

本付録では、以下のカテゴリーの ILE C 拡張を示します。

- 『C89 への拡張としての C99 フィーチャー』
- 『GNU C との互換性のための拡張』
- 372 ページの『10 進浮動小数点サポートのための拡張』

C89 への拡張としての C99 フィーチャー

以下のフィーチャーは、**LANGLVL(*EXTENDED)** オプション (デフォルト言語レベル) を使用してコンパイルした場合にデフォルトで使用可能になります。詳しくは、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** オプションを参照してください。

表 25. C89 への拡張としてのデフォルト C99 フィーチャー

言語フィーチャー	説明している場所
`long long` データ型	45 ページの『整数型』
構造体または共用体の最後の柔軟な配列メンバー	51 ページの『柔軟な配列メンバー 』
関数類似マクロの変数引数	353 ページの『関数類似マクロ』
C++ スタイルのコメント	32 ページの『コメント』

以下のフィーチャーは、指定されたコンパイル・オプションを使用してコンパイルした場合に使用可能になります。

表 26. C89 への拡張としてのデフォルト C99 フィーチャー、および個別オプション制御

言語フィーチャー	説明している場所	個別オプション制御
2 文字表記	31 ページの『2 文字表記文字』	OPTION(*DIGRAPH)

関連情報

- 「*ILE C/C++ コンパイラー参照*」の「コンパイラーの起動」

GNU C との互換性のための拡張

以下のフィーチャーは、すべての言語レベルでデフォルトで使用可能になっています。

表 27. GNU C との互換性のためのデフォルト ILE C 拡張

言語フィーチャー	説明している場所
`#include_next` プリプロセッサー・ディレクティブ	361 ページの『#include_next ディレクティブ』

以下のフィーチャーは、**LANGLVL(*EXTENDED)** オプション (デフォルト言語レベル) を使用してコンパイルした場合にデフォルトで使用可能になります。詳しくは、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** オプションを参照してください。

© Copyright IBM Corp. 1998, 2010

表 28. *GNU C との互換性のためのデフォルト ILE C 拡張*

言語フィーチャー	説明している場所
`__alignof__` 演算子	115 ページの『__alignof__ 演算子』
`__typeof__` 演算子	117 ページの『__typeof__ 演算子』
ID のドル記号	15 ページの『ID に使用される文字』
__thread ストレージ・クラス指定子	44 ページの『__thread ストレージ・クラス指定子』

関連情報

- 「*ILE C/C++ コンパイラー参照*」の「コンパイラーの起動」

10 進浮動小数点サポートのための拡張

以下のフィーチャーは、追加オプションを使用してコンパイルする必要があります。

言語フィーチャー	説明している場所	必要なコンパイル・オプション
10 進浮動小数点型	47 ページの『実数浮動小数点型』, 21 ページの『10 進浮動小数点リテラル』, 97 ページの『浮動小数点の型変換』	**LANGLVL(*EXTENDED)**

付録 B. ILE C++ 言語拡張

本付録では、以下のカテゴリーの標準 C++ への ILE C++ 拡張を示します。

- 『一般的な IBM® 拡張』
- 『C99 との互換性のための拡張』
- 374 ページの『GNU C との互換性のための拡張』
- 374 ページの『GNU C++ との互換性のための拡張』
- 375 ページの『10 進浮動小数点サポートのための拡張』

一般的な IBM® 拡張

以下のフィーチャーは、**LANGLVL(*EXTENDED)** オプション (デフォルト言語レベル) を使用した場合に使用可能になります。詳細については、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** オプションを参照してください。

言語フィーチャー	**説明している場所**
`long long` データ型	45 ページの『整数型』

C99 との互換性のための拡張

ILE C++ により、以下の C99 言語フィーチャーのサポートが追加されます。これらのフィーチャーは、**LANGLVL(*EXTENDED)** オプション (デフォルト言語レベル) を使用した場合に使用可能になります。詳しくは、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** オプションを参照してください。

表 29. 標準 C++ への拡張としてのデフォルト C99 フィーチャー

言語フィーチャー	**説明している場所**
構造体または共用体の最後の柔軟な配列メンバー	51 ページの『柔軟な配列メンバー 』
`_Pragma` 演算子	370 ページの『_Pragma プリプロセス演算子』
追加の事前定義マクロ名	「*ILE C/C++ コンパイラー参照*」
関数類似マクロの空引数	353 ページの『関数類似マクロ』
`__func__` 事前定義 ID	16 ページの『__func__ 事前定義 ID』
16 進浮動小数点リテラル	20 ページの『16 進浮動小数点リテラル』
`enum` 宣言で許可される末尾のコンマ	57 ページの『列挙型定義』
`restrict` 型修飾子	66 ページの『restrict 型修飾子』
可変長配列	79 ページの『可変長配列 (C++ のみ)』
複合リテラル	140 ページの『複合リテラル式』
関数類似マクロの変数引数	353 ページの『関数類似マクロ』

以下のフィーチャーは、特定のコンパイラー・オプションによってのみ使用可能になります。

© Copyright IBM Corp. 1998, 2010

表 30. 標準 C++ への拡張としての C99 フィーチャー

言語フィーチャー	説明している場所
汎用文字名	30 ページの『ユニコード規格』

GNU C との互換性のための拡張

以下のフィーチャーは、**LANGLVL(*EXTENDED)** オプション (デフォルト言語レベル) を使用した場合に使用可能になります。詳しくは、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** オプションを参照してください。

表 31. GNU C との互換性のためのデフォルト ILE C++ 拡張

言語フィーチャー	説明している場所
構造体または共用体の任意の場所への柔軟な配列メンバーの指定	51 ページの『柔軟な配列メンバー 』
集合体の柔軟な配列メンバーの静的初期化	51 ページの『柔軟な配列メンバー 』
`__alignof__` 演算子	115 ページの『__alignof__ 演算子』
`__typeof__` 演算子	117 ページの『__typeof__ 演算子』
汎用左辺値	103 ページの『左辺値と右辺値』
関数の属性	185 ページの『関数属性』
`#include_next` プリプロセッサー・ディレクティブ	361 ページの『#include_next ディレクティブ』
代替キーワード	14 ページの『言語拡張のキーワード』
`__extension__` キーワード	14 ページの『言語拡張のキーワード』
型属性	67 ページの『型属性』
変数属性	89 ページの『変数属性』
ゼロ長配列	52 ページの『ゼロ長配列メンバー』
可変個引数マクロ拡張	356 ページの『可変個引数マクロ拡張』
`#` 警告プリプロセッサー・ディレクティブ	367 ページの『#warning ディレクティブ』
`#assert` プリプロセッサー・ディレクティブ、`#unassert` プリプロセッサー・ディレクティブ	368 ページの『アサーション・ディレクティブ』

以下のフィーチャーは、追加オプションを使用してコンパイルする必要があります。

表 32. 追加のコンパイラー・オプションが必要な、GNU C との互換性のための ILE C++ 拡張

言語フィーチャー	説明している場所
ID のドル記号	15 ページの『ID に使用される文字』

GNU C++ との互換性のための拡張

表 33. GNU C++ との互換性のための ILE C++ 言語拡張

言語フィーチャー	説明している場所
`extern` として宣言されたテンプレートのインスタンス生成	316 ページの『テンプレートのインスタンス化』

関連情報

- 「ILE C/C++ コンパイラー参照」の **LANGLVL(*EXTENDED)**

10 進浮動小数点サポートのための拡張

以下のフィーチャーは、**LANGLVL(*EXTENDED)** オプション (デフォルト言語レベル) を使用してコンパイルした場合にデフォルトで使用可能になります。詳しくは、「*ILE C/C++ コンパイラー参照*」の **LANGLVL** オプションを参照してください。

言語フィーチャー	**説明している場所**
10 進浮動小数点型	47 ページの『実数浮動小数点型』, 21 ページの『10 進浮動小数点リテラル』, 97 ページの『浮動小数点の型変換』

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、米国以外の国においては本書で述べる製品、サービス、またはプログラムを提供しない場合があります。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901 U.S.A.

© Copyright IBM Corp. 1998, 2010

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、IBM 機械コードのご使用条件、またはそれと同等の条項に基づいて、 IBM より提供されます。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。サンプル・プログラムは特定物として現存するままの状態で提供されるものであり、いかなる保証も提供されません。 IBM は、サンプル・プログラムの使用から生ずるいかなる損害に対しても責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. 1998, 2010. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

プログラミング・インターフェース情報は、プログラムを使用してアプリケーション・ソフトウェアを作成する際に役立ちます。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

業界標準

次の規格がサポートされます。

- C 言語は、International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1990) に準拠しています。
- C 言語は、International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)) に準拠しています。
- C++ 言語は、International Standard for Information System-Programming Language C++ (ISO/IEC 14882:1998) に準拠しています。
- C++ 言語は、International Standard for Information System-Programming Language C++ (ISO/IEC 14882:2003(E)) に準拠しています。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

［ア行］

あいまいさ
　解決　152, 267
　仮想関数呼び出し　272
　基底クラス　263
　基底メンバー名と派生メンバー名　265
アクセス可能性　243, 264
アクセス規則
　仮想関数　274
　基底クラス　257
　クラス型　219, 244
　フレンド　250
　マルチアクセス　264
　メンバー　243
　protected メンバー　256
アクセス指定子　230, 243, 253, 261
　クラス派生の中の　257
値による受け渡し　189
アドレス演算子 (&)　113
暗黙的変換　95
　型　95
　基底クラスへのポインター　255
　左辺値 (lvalue)　103
　整数　96
　派生クラスへのポインター　255, 258
　ブール　96
　浮動小数点　97
暗黙のインスタンス生成
　テンプレート　316
位置合わせ　91, 92
　構造体　91
　構造体および共用体　63
　構造体メンバー　50
　ビット・フィールド　54
一時オブジェクト　337
インライン
　関数　176, 231
　関数指定子　176
打ち切り機能　347
エスケープ文字 ¥　29
エスケープ・シーケンス　29
　アラーム ¥a　30
　一重引用符 ¥'　30
　円記号 ¥¥　30
エスケープ・シーケンス *(続き)*
　改行 ¥n　30
　改ページ ¥f　30
　疑問符 ¥?　30
　垂直タブ ¥v　30
　水平タブ ¥t　30
　二重引用符 ¥"　30
　バックスペース ¥b　30
　復帰 ¥r　30
演算子　26
　結合順序　146
　スコープ・レゾリューション　254, 265, 272
　代替表記　27
　代入　119
　　コピー代入　295
　多重定義　207, 231
　　単項　208
　　2 項　210
　単項　110
　単項正演算子 (+)　112
　定義済み　363
　等価　124
　ビット単位否定演算子 (~)　112
　複合代入　120
　プリプロセッサー
　　プラグマ　370
　　#　356
　　##　357
　メンバーへのポインター　130, 235
　優先順位　146
　　型名　73
　　例　149
　リレーショナル　123
　2 項　118
　const_cast　137
　delete　145
　dynamic_cast　139
　new　141
　reinterpret_cast　136
　sizeof　116
　static_cast　135
　typeid　114
　! (論理否定)　112
　!= (非等価)　124
　& (アドレス)　113
　& (ビット単位 AND)　125
　&& (論理 AND)　127
　() (関数呼び出し)　109, 169
　* (間接)　113
　* (乗算)　121
演算子 *(続き)*
　+ (加法)　121
　++ (増分)　111
　, (コンマ)　129
　- (減法)　122
　- (単項負)　112
　-- (減分)　111
　-> (矢印)　110
　->* (メンバーを指すポインター)　130
　. (ドット)　109
　.* (メンバーを指すポインター)　130
　/ (除法)　121
　:: (スコープ・レゾリューション)　108
　= (単純代入)　119
　== (等価)　124
　? : (条件)　131
　> (より大きい)　123
　>= (より大きいまたは等しい)　123
　>> (右シフト)　122
　< (より小さい)　123
　<= (より小さいまたは等しい)　123
　<< (左シフト)　122
　| (ビット単位包含 OR (包含論理和))　126
　|| (論理 OR)　127
　% (剰余)　121
　[] (配列添え字)　128
　__typeof__　117
　^ (ビット単位排他 OR)　126
演算子関数　207
演算子の結合順序　146
演算子の優先順位　146
エントリー・ポイント
　プログラム　188
オーバーライド、仮想関数の　273
　共変仮想関数　271
オブジェクト　103
　クラス
　　宣言　220
　静的
　　デストラクターでスローされる例外　334
　説明　35
　存続時間　1
　ネーム・スペース・スコープ
　　コンストラクターでスローされる例外　334
　restrict で修飾されたポインター　66
オブジェクト類似マクロ　352

© Copyright IBM Corp. 1998, 2010

［カ行］

拡張
  整数および浮動小数点　97
隠れた名前　221, 223
囲みクラス　231, 247
可視性　1, 6
  クラス・メンバー　245
  ブロック　2
下線文字　14, 16
仮想
  基底クラス　253, 263, 268
仮想関数　232, 269
  あいまいな呼び出し　272
  アクセス　274
  オーバーライド　273
  純粋指定子　274
型
  型ベースの別名割り当て　76
  型変換　133
  クラス　220
型指定子　45
  オーバーライド　92
  クラス型　220
  詳述　223
  列挙型　57
  bool　46
  char　48
  double　47
  float　47
  int　45
  long　45
  long double　47
  long long　45
  short　45
  unsigned　45
  void　48
  wchar_t　45, 48
  _Bool　46
型修飾子
  関数仮パラメーター　206
  複製　63
  const　62, 65
  const および volatile　72
  restrict　62, 66
  volatile　62
型属性　67
型変換
  関数からポインターへ　99
  関数引数　101
  キャスト　133
  左辺値から右辺値への　98, 103, 216
  算術　95
  参照　100
  修飾　101
  整数　96
型変換 *(続き)*
  配列からポインターへ　99
  派生クラスへのポインター　268
  標準の　95
  ブール　96
  浮動小数点　97
  ポインター　99
  明示的キーワード　292
  メンバーへのポインター　235
  ユーザー定義の　289
  void ポインター　100
型名　73
  型名キーワード　327
  修飾された　108, 225
  ローカル　227
  __typeof__ 演算子　117
型名キーワード　327
括弧で囲んだ式　73, 107
可変長配列　36, 79, 167
  型名　74
  関数仮パラメーターとして　80, 189, 215
  sizeof　106
可変的に変更される型　79
可変変更型　158
加法演算子 (+)　121
間隔文字　351
関数　169
  インライン　176, 231
  仮想　232, 269, 272
  型名　74
  関数からポインターへの変換　99
  関数テンプレート　308
  関数呼び出し演算子　169
  クラス・テンプレート　306
  シグニチャー　182
  事前定義 ID　16
  指定可能な属性　185
  指定子　176
  宣言　169
    パラメーターの名前　184
    複数の　173
    例　171
    C++　232
  多重定義　205
  定義　169
    例　172
  デフォルト引数　193
    制約事項　194
    評価　194
  テンプレート関数
    テンプレート引数の推定　309
  名前
    診断　16
  パラメーター　189
  引数　169, 189
関数 *(続き)*
    型変換　101
  フレンド　245
  ブロック　169
  プロトタイプ　169
  へのポインター　195
  変換関数　292
  ポリモアフィック　252
  本体　169
  戻り値　169, 181
  戻りの型　169, 180, 181
  呼び出し　109, 189
    左辺値として　104
  ライブラリー関数　169
  例外指定　342
  例外処理　345
  割り振り　192
  割り振り解除　192
  main　188
  name　169
  return ステートメント　166
関数 try ブロック　331
  ハンドラー　334
関数指定子　104
  明示的　290, 292
関数テンプレート
  明示的特殊化　322
関数の属性　185
関数類似マクロ　353
間接演算子 (*)　113
間接基底クラス　252, 263
間接参照演算子　113
キーワード　13
  下線文字　14
  言語拡張　14
  テンプレート　297
  例外処理　331
  template　327, 328
疑似デストラクター　288
基底クラス
  あいまいさ　263, 265
  アクセス規則　257
  仮想　263, 268
  間接　252, 263
  初期化　282
  抽象　274
  直接　262
  へのポインター　255
  マルチアクセス　264
基底リスト　263
キャスト式　133
狭幅の文字リテラル　23
共変仮想関数　271
共用体　49
  クラス型として　219, 220
  互換性　59, 60

共用体 *(続き)*
- 指定子　50
- 初期化　85
- 名前なしメンバー　83

切り捨て
- 整数除法　121

空白文字　13, 32, 351, 357
区切り子　26
- 代替表記　27

組み込みデータ型　35
クラス　222
- アクセス規則　243
- 概説　219
- 仮想　263, 269
- キーワード　219
- 基底リスト　253
- クラス指定子　220
- クラス・オブジェクト　35
- クラス・テンプレート　304
- 継承　251
- 集合体　220
- 静的メンバー　239
- 宣言　220
  - 不完全な　224, 230
- 抽象　274
- 名前のスコープ　223
- ネストされた　224, 248
- 派生　253
- フレンド　245
- ポリモアフィック　219
- メンバー関数　231
- メンバー・スコープ　233
- メンバー・リスト　229
- ローカル　226
- base　253
- this ポインター　236
- using 宣言　258

クラス・テンプレート
- 静的データ・メンバー　306
- 宣言と定義　305
- 明示的特殊化　322
- メンバー関数　306
- テンプレート・クラス との区別　304

クラス・メンバー
- アクセス演算子　109
- アクセス規則　243
- クラス・メンバー・リスト　229
- 初期化　282
- 宣言　230
- 割り振りの順序　230

グローバル変数　3, 8
- 未初期化　83

警告プリプロセッサー・ディレクティブ　367
継承
- 概説　251

継承 *(続き)*
- 多重　252, 262

継続文字　25, 351
言語拡張　14
- C99　371
- GNU C　371

減分演算子 (--)　111
減法演算子 (-)　122
構造体　49, 222
- 位置合わせ　63
- 基底クラスとして　257
- クラス型として　219, 220
- 互換性　59, 60
- 柔軟な配列メンバー　50, 51
- 初期化　83
- 名前なしメンバー　83
- ネーム・スペース　6
- パック　50
- メンバー　50
  - 位置合わせ　50
  - 埋め込み　50
  - ゼロ長配列　50
  - パック　53
  - 不完全型　51
  - メモリー内のレイアウト　50, 83
- ID (タグ)　50

後置
- ++ と --　111

候補関数　205, 215
互換型
- 算術型　49
- 条件式における　131
- ソース・ファイル間　60
- 配列　80

互換性
- データ型　37
- ユーザー定義型　59
- C89 および C99　371
- XL C および GCC　371
- XL C++ および C99　373
- XL C++ および GCC　374

コピー代入演算子　295
コピー・コンストラクター　293
コメント　32
コンストラクター　279
- 概要　277
- コピー　293
- コンストラクターでスローされる例外　334
- 初期化
  - 明示的　280
- 単純　279, 287
- 非単純　279, 287
- 変換　290, 292
- 例外処理　341

コンマ　129

コンマ *(続き)*
- 列挙子リスト内　57

［サ行］

最良の実行可能関数　215
サブスクリプトされていない配列
- 説明　79, 184

サブスクリプト宣言子
- 配列の　78

算術型
- 型互換性　49

算術変換　95
参照
- 型変換　100
- 初期化　88
- 説明　80
- 宣言子　113
- バインディング　88
- 戻りの型として　181

参照による受け渡し　80, 190
暫定定義　38
式
- あいまいなステートメントの解決　152
- 括弧で囲んだ　107
- キャスト　133
- コンマ　129
- 条件付き　131
- ステートメント　152
- 整数定数　105
- 説明　103
- 代入　119
- 単項　110
- メンバーへのポインター　130
- 割り振り　141
- 割り振り解除　145
- 1 次　104
- 2 項　118
- new 初期化指定子　143
- throw　146, 339

字下げ、コードの　351
指数　20
事前定義 ID　16
指定子
- アクセス制御　257
- インライン　176
- 純粋　232
- ストレージ・クラス　39
- union　83

指定初期化指定子
- union　83

自動ストレージ・クラス指定子　40
シフト演算子 << および >>　122
集合体型　35, 280
- 初期化　83, 280

修飾子
　パラメーター型指定　206
　const　62
　restrict　66
　volatile　62, 67
修飾された名前　108, 225
修飾変換　101
従属名　326
柔軟な配列メンバー　51
純粋仮想関数　274
純粋指定子　230, 232, 272, 274
条件式 (? :)　121, 131
条件付きコンパイル・ディレクティブ　362
　例　365
　elif プリプロセッサー・ディレクティブ　363
　else プリプロセッサー・ディレクティブ　365
　endif プリプロセッサー・ディレクティブ　365
　if プリプロセッサー・ディレクティブ　363
　ifdef プリプロセッサー・ディレクティブ　364
　ifndef プリプロセッサー・ディレクティブ　364
乗算演算子 (*)　121
詳述型指定子　223
情報隠蔽　1, 3
情報の隠蔽　229, 255
剰余演算子 (%)　121
省略符号
　関数宣言で　183
　関数定義で　183
　変換シーケンス　217
　マクロ引数リストにおける　353
初期化
　基底クラス　282
　共用体メンバー　85
　クラス・メンバー　282
　参照　100
　自動オブジェクト　82
　集合体型　83
　静的オブジェクト　82
　静的データ・メンバー　242
　extern オブジェクト　83
　register オブジェクト　83
初期化指定子　81, 282
　共用体　85
　集合体型　83
　列挙型　85
除法演算子 (/)　121
スカラー型　35, 74
スコープ　1
　囲みおよびネスト　3
スコープ *(続き)*
　関数　3
　関数プロトタイプ　3
　クラス　5
　クラス名　223
　グローバル　3
　グローバル・ネーム・スペース　3
　説明　1
　ネスト・クラス　224
　フレンド　247
　マクロ名　356
　メンバー　233
　ローカル (ブロック)　2
　ローカル・クラス　226
　ID　5
スコープ・レゾリューション演算子
　あいまいな基底クラス　265
　仮想関数　272
　継承　254
　説明　108
スタック・アンワインド　341
ステートメント　151
　あいまいさの解決　152
　式　152
　選択　154, 156
　ヌル　168
　ブロック　153
　ラベル　151
　break　164
　continue　164
　do　161
　for　162
　goto　167
　if　154
　return　166, 181
　switch　156
　while　160
ストリング
　終止符　25
　リテラル　24
ストレージ期間　1
　自動ストレージ・クラス指定子　40
　静的
　　デストラクターでスローされる例外　334
　extern ストレージ・クラス指定子　42
　register ストレージ・クラス指定子　43
　static　40, 174
ストレージ・クラス指定子　39
　auto　40
　extern　41, 174
　mutable　42
　register　43
　static　40, 174
整数
　拡張　97
整数 *(続き)*
　型変換　96
　データ型　45
　定数式　57, 105
　リテラル　17
静的
　ストレージ・クラス指定子
　　リンケージ　41
　データ・メンバー　240
　データ・メンバーの初期化　242
　バインディング　269
　メンバー　239
　メンバー関数　242
接続プリプロセッサー・ディレクティブ
　#　356
接続プリプロセッサー・ディレクティブ
　##　357
接頭部
　10 進浮動小数点定数　21
　16 進整数リテラル　18
　16 進浮動小数点定数　20
　8 進整数リテラル　19
　++ と --　111
接尾部
　整数リテラル定数　17
　浮動小数点リテラル　19
　10 進浮動小数点定数　21
　16 進浮動小数点定数　20
ゼロ長配列　52
宣言　169
　あいまいなステートメントの解決　152
　クラス　220, 224
　構文　38, 73, 170
　サブスクリプトされていない配列　79
　説明　37
　重複型修飾子　63
　フレンド　250
　メンバーへのポインター　234
　メンバー・リスト内のフレンド指定子　245
宣言子
　参照　80
　説明　71
宣言領域　1
増分演算子 (++)　111
添え字演算子　78, 128
　型名内の　74

[タ行]

代入演算子 (=)
　単純　119
　複合　120
　ポインター　77
タグ
　構造体　50

タグ *(続き)*
列挙型　57
union　50
多次元配列　78
多重
アクセス　264
継承　252, 262
多重定義
演算子　207, 219
関数呼び出し　212
クラス・メンバー・アクセス　214
減分　209
増分　209
添え字　213
代入　211
単項　208
2 項　210
関数　205, 259
制約事項　206
関数テンプレート　314
説明　205
多重定義解決　215, 268
多重定義された関数のアドレスの解決　217
単項演算子　110
正 (+)　112
負 (-)　112
単項式　110
抽象クラス　272, 274
直接基底クラス　262
データ型
組み込み　35
互換　37
集合体　35
スカラー　35
整数　45
ブール　46
不完全な　36
複合　35, 37
浮動小数点　47
文字　48
ユーザー定義の　35, 49
列挙された　57
void　48
データ・メンバー
スコープ　233
静的　240
説明　230
定義
暫定　38
説明　37
マクロ　351
メンバー関数　231
定数式　57, 105
定数初期化指定子　230
デストラクター　286
デストラクター *(続き)*
概要　277
疑似　288
デストラクターでスローされる例外　334
例外処理　341
デフォルト・コンストラクター　279
テンプレート
インスタンス生成　297, 316, 319
暗黙の　316
フォワード宣言　316, 319
明示的　318
インスタンス生成のポイント　327
関数
多重定義　314
引数の推定　313
部分選択　315
関数テンプレート　308
「型」テンプレート引数の推定　312
クラス
静的データ・メンバー　306
宣言と定義　305
明示的特殊化　322
メンバー関数　306
テンプレート・クラス との区別　304
クラスとそのフレンド間の関係　307
従属名　326
スコープ　321
宣言　297
定義のポイント　327
特殊化　297, 316, 319
名前のバインディング　326
パラメーター　298
型　298
デフォルト引数　299
非型　298
template　299
引数
型　300
非型　301
部分的特殊化　324
パラメーターと引数リスト　325
マッチング　326
明示的特殊化　319, 321
関数テンプレート　322
クラス・メンバー　321
宣言　320
定義と宣言　321
テンプレート引数　300
型　300
推定　309
推定、型　312
推定、非型　313
非型　301
テンプレート引数 *(続き)*
template　303
テンプレート・キーワード　328
トークン　13, 351
演算子および区切り子の代替表記　27
等価演算子 (==)　124
動的バインディング　269
特殊メンバー関数　233
特殊文字　28
ドット演算子　109
ドル記号　15, 28

［ナ行］

名前
隠れた　108, 221, 223
レゾリューション　258, 267
ローカル型　227
名前なしネーム・スペース　200
名前の隠蔽　6, 108
あいまいさ　266
アクセス可能基底クラス　268
名前のバインディング　326
ヌル
ステートメント　168
プリプロセッサー・ディレクティブ　369
ポインター　86
ポインター定数　99
文字 ¥0　25
ネーム
競合　5
マングリング　10
レゾリューション　3
ネーム・スペース　197
拡張　198
宣言　197
多重定義　199
定義　197
名前なし　200
ネーム・スペース・スコープ・オブジェクト
コンストラクターでスローされる例外　334
フレンド　201
別名　197, 198
明示的アクセス　203
メンバー定義　201
ユーザー定義の　3
using 宣言　203
using ディレクティブ　202
ネスト・クラス
スコープ　224
フレンドのスコープ　248

[ハ行]

排他 OR 演算子、ビット単位 (^) 126
配置構文 142
配列
  型互換性 80
  可変長 74, 79
  関数仮パラメーターとして 184
  柔軟な配列メンバー 50, 51
  初期化 86
  説明 78
  ゼロ長 50, 52
  宣言 184, 230
  添え字演算子 128
  多次元 78
  配列からポインターへの変換 99
バインディング 88
  仮想関数 269
  静的 269
  直接 89
  動的 269
派生 (derivation) 253
  配列型 78
  public、protected、private 257
派生クラス
  構築順序 285
  へのポインター 255
  catch ブロック 338
パック
  構造体メンバー 53
番号記号 (#)
  プリプロセッサー演算子 356
  プリプロセッサー・ディレクティブの文字 351
ハンドラー 333
汎用文字名 15, 23, 30
引数
  値による受け渡し 189
  後書き 353, 356
  受け渡し 169, 189
  参照による受け渡し 190
  デフォルト 193
  評価 194
  マクロ 353
  catch ブロックの 337
  main 関数 188
左シフト演算子 (<<) 122
ビット単位否定演算子 (~) 112
ビット・フィールド 53
  型名 118
  構造体メンバーとして 50
非等価演算子 (!=) 124
評価順序点 130
標準の型変換 95
ブール
  型変換 96
ブール *(続き)*
  データ型 46
  リテラル 19
ファイルのインクルード 358, 361
ファイル・スコープ・データ宣言
  サブスクリプトされていない配列 79
不完全型 36, 78
  クラス宣言 224
  構造体メンバーとして 50, 51
複合
  型 35
  式 121
  ステートメント 153
  代入 120
  リテラル 140
複合型 37
  ソース・ファイル間 60
副次作用 67
複数文字リテラル 23
浮動小数点
  拡張 97
  型変換 97
  定数 20, 21
  リテラル 19
浮動小数点型 47
プラグマ
  プリプロセッサー・ディレクティブ 369
  _Pragma 370
プラグマ演算子 370
フリー・ストア
  delete 演算子 145
  new 演算子 141
プリプロセッサー演算子
  # 356
  ## 357
  _Pragma 370
プリプロセッサー・ディレクティブ 351
  条件付きコンパイル 362
  特殊文字 351
  プリプロセスの概要 351
  warning 367
プリプロセッサー・ディレクティブの定義 351
フレンド
  アクセス規則 250
  仮想関数 272
  指定子 245
  スコープ 247
  テンプレートを必要とする場合のクラスとの関係 307
  ネスト・クラス 248
  ポインターの暗黙的変換 258
  メンバー関数 231
ブロックの可視性 2
ブロック・ステートメント 153
別名 80
  型ベースの別名割り当て 76
変換
  暗黙的変換シーケンス 216
  関数 292
  コンストラクター 290
変換シーケンス
  暗黙の 216
  省略符号 217
  標準の 216
  ユーザー定義の 217
変換単位 1
変更可能な左辺値 103, 119
変数属性 89
ポインター
  型修飾 74
  型変換 99, 268
  関数への 195
  参照解除 76
  説明 74
  ヌル 86
  汎用 100
  ポインター演算 75
  メンバーへの 130, 234
  cv 修飾 74
  restrict で修飾された 66
  this 236
  void* 99
包含 OR 演算子、ビット単位 (|) 126
ポリモアフィズム
  ポリモアフィック関数 252
  ポリモアフィック・クラス 219, 270
ポンド記号 (#)
  プリプロセッサー演算子 356
  プリプロセッサー・ディレクティブの文字 351

[マ行]

マクロ
  オブジェクト類似 352
  関数類似 353
  定義 351
    __typeof__ 演算子 118
  変数引数 353, 356
  呼び出し 353
マルチバイト文字 28
  連結 26
右シフト演算子 (>>) 122
明示的
  インスタンス生成、テンプレート 318
  型変換 133
  キーワード 290, 292
  特殊化、テンプレート 319, 321
メンバー
  アクセス 243

メンバー *(続き)*
アクセス制御　261
仮想関数　232
クラス・メンバー・アクセス演算子　109
スコープ　233
静的　239
データ　230
へのポインター　130, 234
protected　256
static　225
メンバー関数
静的　242
定義　231
特殊　233
フレンド　231
const および volatile　232
this ポインター　236, 274
メンバーへのポインター
演算子　130, 235
型変換　235
宣言　234
メンバー・リスト　220, 229
文字
データ型　48
マルチバイト　26, 28
リテラル　23
文字セット
拡張　28
ソース　28
モジュロ演算子 (%)　121
戻りの型
として参照　181
size_t　116

[ヤ行]

ユーザー定義の型変換　289
ユーザー定義のデータ型　35, 49
ユニコード　30
より大きい演算子 (>)　123
より大きいまたは等しい演算子 (>=)　123
より小さい演算子 (<)　123
より小さいまたは等しい演算子 (<=)　123
弱いシンボル　92

[ラ行]

ラベル
暗黙宣言　3
ステートメント　151
switch ステートメントの中　157
リテラル　17
ストリング　24
整数　17
リテラル *(続き)*
10 進　18
16 進　18
8 進　19
ブール　19
複合　140
浮動小数点　19
文字　23
リンケージ　1, 7
インライン・メンバー関数　232
外部　8
関数定義で　174
関数ポインターで　196
言語　9
指定　9
自動ストレージ・クラス指定子　40
内部　8, 40, 174
なし　9
複数の関数宣言　173
弱いシンボル　92
const cv 修飾子　65
extern ストレージ・クラス指定子　11, 42
register ストレージ・クラス指定子　44
static ストレージ・クラス指定子　41
例外
関数 try ブロック・ハンドラー　334
指定　342
宣言　333
例外処理　331
関数 try ブロック　331
キャッチの順序　338
コンストラクター　341
試行例外　334
スタック・アンワインド　341
デストラクター　341
特殊な関数　345
ハンドラー　331, 333
引数のマッチング　337
例、C++　347
例外オブジェクト　331
例外の rethrow　339
catch ブロック　333
引数　337
set_terminate　347
set_unexpected　347
terminate 関数　346
throw 式　332, 339
try ブロック　331
unexpected 関数　345
列挙型　57
後書きコンマ　57
互換性　59, 60
初期化　85
宣言　57
連結
マクロ　357
マルチバイト文字　26
ローカル
型名　227
クラス　226
論理演算子
! (論理否定)　112
&& (論理 AND)　127
|| (論理 OR)　127

[ワ行]

ワイド文字
リテラル　23
ワイド・ストリング・リテラル　26
割り振り
関数　192
式　141
割り振り解除
関数　192
式　145

[数字]

1 次式　104
1 の補数演算子 (~)　112
10 進
浮動小数点定数　21
10 進整数リテラル　18
16 進
浮動小数点定数　20
16 進整数リテラル　18
2 項式と演算子　118
2 文字表記文字　31
3 文字表記　31
8 進整数リテラル　19

A

alignof 演算子　115
AND 演算子、ビット単位 (&)　125
AND 演算子、論理 (&&)　127
argc (引数カウント)　188
example　188
argv (引数ベクトル)　188
example　188
ASCII 文字コード　30
asm　13
atexit 関数　346

B

break ステートメント　164

C

case ラベル 156
catch ブロック 331, 333
  キャッチの順序 338
  引数のマッチング 337
char 型指定子 48
Classic C ix
const 65
  オブジェクト 103
  型名内に配置 74
  関数属性 186
  修飾子 62
  対 #define 352
  メンバー関数 232
  const 型をキャストする 191
const_cast 137, 191
continue ステートメント 164
cv 修飾子 62, 72
  構文 62
  パラメーター型指定 206
C++ 以外のプログラムへのリンク 9

D

default
  文節 156, 157
  ラベル 157
defined 単項演算子 363
delete 演算子 145
do ステートメント 161
double 型指定子 47
downcast 139
dynamic_cast 139

E

EBCDIC 文字コード 30
elif プリプロセッサー・ディレクティブ 363
else
  ステートメント 154
  プリプロセッサー・ディレクティブ 365
endif プリプロセッサー・ディレクティブ 365
enum
  キーワード 57
enumerator 57
error プリプロセッサー・ディレクティブ 366
extern ストレージ・クラス指定子 8, 11, 41, 174
  可変長配列の使用 79
  関数ポインターで 196
  テンプレート宣言で 316, 319

F

float 型指定子 47
for ステートメント 162

G

goto ステートメント 167
  制約事項 167

I

ID 15, 105
  大文字小文字の区別 15
  事前定義 16
  特殊文字 15, 28
  予約 13, 14, 16
  リンケージ 8
  id-expression 72, 106
  namespace 5
identifier
  ラベル 151
if
  ステートメント 154
  プリプロセッサー・ディレクティブ 363
ifdef プリプロセッサー・ディレクティブ 364
ifndef プリプロセッサー・ディレクティブ 364
include プリプロセッサー・ディレクティブ 358
include_next プリプロセッサー・ディレクティブ 361

K

K&R C ix

L

line プリプロセッサー・ディレクティブ 367
long double 型指定子 47
long long 型指定子 45
long 型指定子 45
lvalues 62, 103, 105
  型変換 98, 103, 216
  キャスト 133

M

main 関数 188
  引数 188
  example 188
mutable ストレージ・クラス指定子 42

N

namespace
  クラス名 223
  コンテキスト 6
  ID 5
new 演算子
  初期化指定子の式 143
  説明 141
  デフォルト引数 194
  配置構文 142
  set_new_handler 関数 144

O

OR 演算子、論理 (||) 127

P

packed
  変数属性 92

R

register ストレージ・クラス指定子 43
reinterpret_cast 136
restrict 66
  パラメーター型指定 206
return ステートメント 166, 181
RTTI サポート 114, 139
rvalues 103

S

set_new_handler 関数 144
set_terminate 関数 347
set_unexpected 関数 345, 347
short 型指定子 45
signed 型指定子
  char 48
  int 45
  long 45
  long long 45
sizeof 演算子 116
  可変長配列の使用 80
size_t 116
Standard C ix
Standard C++ ix
static
  可変長配列の使用 79
  ストレージ・クラス指定子 40, 174
  配列宣言で 184

static *(続き)*
　メンバー　225
static ストレージ・クラス指定子　8
static_cast　135
struct 型指定子　50
switch ステートメント　156

T

terminate 関数　331, 332, 338, 341, 345, 346
　set_terminate　347
this ポインター　236, 274
throw 式　146, 331, 339
　ネストされた try ブロック内　332
　引数のマッチング　337
　例外の rethrow　339
try キーワード　331
try ブロック　331
　ネストされた　332
typedef 指定子　60
　可変長配列の使用　80
　クラス宣言　227
　修飾された型名　225
　メンバーへのポインター　235
　ローカル型名　227
typeid 演算子　114

U

undef プリプロセッサー・ディレクティブ　356
unexpected 関数　331, 345, 346
　set_unexpected　347
unsigned 型指定子
　char　48
　int　45
　long　45
　long long　45
　short　45
using 宣言　203, 258, 267
　メンバー関数の多重定義　259
　メンバー・アクセスの変更　261
using ディレクティブ　202

V

void　48
　関数定義で　180, 183
　ポインター　99, 100
volatile
　修飾子　62, 67
　メンバー関数　232

W

wchar_t 型指定子　23, 45, 48
while ステートメント　160

［特殊文字］

! (論理否定演算子)　112
!= (非等価演算子)　124
プリプロセッサー演算子　356
プリプロセッサー・ディレクティブの文字　351
(マクロ連結)　357
$　15, 28
& (アドレス演算子)　113
& (参照宣言子)　80
& (ビット単位 AND 演算子)　125
&& (論理 AND 演算子)　127
&= (複合代入演算子)　120
* (間接演算子)　113
* (乗算演算子)　121
*= (複合代入演算子)　120
+ (加法演算子)　121
+ (単項正演算子)　112
++ (増分演算子)　111
+= (複合代入演算子)　120
, (コンマ演算子)　129
- (減法演算子)　122
- (単項負演算子)　112
-- (減分演算子)　111
-> (矢印演算子)　110
. (ドット演算子)　109
/ (除法演算子)　121
/= (複合代入演算子)　120
:: (スコープ・レゾリューション演算子)　108
= (単純代入演算子)　119
== (等価演算子)　124
? : (条件演算子)　131
[ ] (配列添え字演算子)　128
> (より大きい演算子)　123
>= (より大きいまたは等しい演算子)　123
>> (右シフト演算子)　122
>>= (複合代入演算子)　120
< (より小さい演算子)　123
<= (より小さいまたは等しい演算子)　123
<< (左シフト演算子)　122
<<= (複合代入演算子)　120
| (ビット単位包含 OR 演算子)　126
|| (論理 OR 演算子)　127
% (剰余)　121
_Pragma　370
__align　63
__cdecl　195
__func__　16
__typeof__ 演算子　117
__VA_ARGS__　353, 356
~ (ビット単位否定演算子)　112
^ (ビット単位排他 OR 演算子)　126
^= (複合代入演算子)　120
¥ エスケープ文字　29
¥ 継続文字　25, 351

IBM®

プログラム番号: 5761–WDS

Printed in USA

SC88-4026-02

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21