

XL C/C++ Enterprise Edition for AIX



XL C/C++ ランゲージ・リファレンス

バージョン 7.0

XL C/C++ Enterprise Edition for AIX



XL C/C++ ランゲージ・リファレンス

バージョン 7.0

ご注意

本書の情報およびそれによってサポートされる製品を使用する前に、 421 ページの『特記事項』に記載する一般情報をお読みください。

本書は、IBM XL C/C++ Enterprise Edition for AIX® (プロダクト番号5724-I11) のバージョン 7.0.0、および新しい版で明記されていない限り、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。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

<http://www.ibm.com/jp/manuals/main/mail.html>

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

<http://www.ibm.com/jp/manuals/> の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

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： SC09-7890-00
XL C/C++ Enterprise Edition for AIX
XL C/C++ Language Reference
Version 7.0

発 行： 日本アイ・ビー・エム株式会社

担 当： ナショナル・ランゲージ・サポート

第1刷 2004.7

この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財)日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

© Copyright International Business Machines Corporation 1998, 2004. All rights reserved.

© Copyright IBM Japan 2004

目次

本書について	vii
IBM 言語拡張機能	viii
GNU Cおよび C++ に関連するフィーチャー	ix
強調表示の規則	ix
構文図の読み方	x

第 1 章 スコープおよびリンケージ

スコープ	2
ローカル・スコープ	3
関数スコープ	3
関数プロトタイプ・スコープ	4
グローバル・スコープ	4
クラス・スコープ	4
ID のネーム・スペース	5
名前の隠蔽	6
プログラム・リンケージ	7
内部結合	7
外部結合	8
リンケージなし	8
リンケージ指定 — C++ 以外のプログラムへのリンク	9
ネーム・マングリング	10

第 2 章 字句エレメント

トークン	13
区切り子	13
ソース・プログラムの文字セット	14
エスケープ・シーケンス	15
ユニコード規格	16
3 文字表記	18
マルチバイト文字	18
コメント	18
ID	20
予約済みの ID	21
ID での大/小文字の区別と特殊文字	21
事前定義済みの ID	22
キーワード	22
リテラル	24
ブール・リテラル	24
整数リテラル	25
浮動小数点リテラル	26
複素数リテラル	29
文字リテラル	29
ストリング・リテラル	31
複合リテラル	32

第 3 章 宣言

宣言概要	35
変数属性	36
__align 指定子	38
仮定義	39
オブジェクト	39

ストレージ・クラス指定子	40
auto ストレージ・クラス指定子	41
extern ストレージ・クラス指定子	42
mutable ストレージ・クラス指定子	44
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	45
typedef	47
型指定子	49
型名	49
型属性	50
互換型	53
単純型指定子	54
複合型	59
複素数型	77
型修飾子	79
const 型修飾子	80
volatile 型修飾子	81
restrict 型修飾子	82
asm 宣言	84
不完全型	85

第 4 章 宣言子

初期化指定子	88
ポインター	89
ポインターの宣言	90
ポインターの割り当て	90
ポインターの初期化	91
ポインターの使用	92
ポインター演算	92
配列	94
配列の宣言	95
配列の初期化	98
関数指定子	102
参照	103
参照の初期化	103

第 5 章 式と演算子

演算子優先順位と結合順序	106
左辺値と右辺値	109
1 次式	111
ID 式	112
整数定数式	112
括弧で囲んだ式 ()	113
C++ スコープ・レゾリューション演算子 ::	114
後置式	115
関数呼び出し演算子 ()	116
配列添え字演算子 []	117
ドット演算子	118
矢印演算子 ->	119
typeid 演算子	119
static_cast 演算子	120

reinterpret_cast 演算子	121
const_cast 演算子	123
dynamic_cast 演算子	124
単項式	126
増分 ++	127
減分 --	128
単項正 +	128
単項負 -	128
論理否定 !	128
ビット単位否定 ~	129
アドレス &	129
間接 *	130
alignof 演算子	131
sizeof 演算子	131
typeof 演算子	133
ラベル値演算子 &&	133
C++ の new 演算子	134
C++ の delete 演算子	138
キャスト式	139
共用体型へのキャスト	140
2 項式	141
乗算 *	142
除法 /	142
剰余 %	143
加法 +	143
減法 -	143
ビット単位左シフトと右シフト << >>	144
関係 < > <= >=	144
等価 == !=	146
ビット単位 AND &	147
ビット単位排他 OR ^	147
ビット単位包含 OR	148
論理 AND &&	149
論理 OR	149
メンバーを指す C++ ポインター演算子 (.* ->*)	150
条件式	151
C の条件式の型	152
C++ の条件式の型	152
条件式の例	152
代入式	153
単純代入 =	153
複合代入	154
コンマ式	155
C++ の throw 式	157

第 6 章 暗黙の型変換 159

整数および浮動小数点拡張	159
標準の型変換	160
左辺値から右辺値への変換	160
ブール変換	161
整数変換	161
浮動小数点の型変換	162
ポインター型変換	162
参照変換	164
メンバーを指すポインターの型変換	164

修飾変換	164
関数引き数変換	164
その他の変換	165
算術変換	166
explicit キーワード	167

第 7 章 関数 169

C 関数に対する C++ の拡張	169
関数宣言	170
C++ 関数の宣言	173
関数属性	174
関数宣言の例	179
関数定義	180
省略符号および void	184
関数定義の例	185
main() 関数	186
main への引き数	187
main への引き数の例	187
関数呼び出しおよび引き数の受け渡し	188
値による引き数の受け渡し	189
参照による引き数の受け渡し	190
C++ 関数におけるデフォルト引き数	192
デフォルト引き数に関する制約事項	193
デフォルト引き数の評価	193
関数からの戻り値	194
戻りの型としての参照の使用	195
割り振り関数および割り振り解除関数	195
関数へのポインター	197
インライン関数	197
ネストされた関数	199

第 8 章 ステートメント 201

ラベル	201
ローカルに宣言されたラベル	202
値としてのラベル	202
式ステートメント	203
C++ でのあいまいなステートメントの解決	203
ブロック・ステートメント	204
ステートメント式	205
if 文	206
switch 文	207
while ステートメント	211
do ステートメント	212
for ステートメント	213
break ステートメント	215
continue ステートメント	215
return ステートメント	217
戻り式の値および関数値	218
goto 文	219
計算済み goto	220
ヌル・ステートメント	220

第 9 章 プリプロセッサ・ディレクティブ 221

プリプロセッサの概要	221
プリプロセッサ・ディレクティブの形式	222

マクロの定義および展開 (#define)	222
オブジェクト類似マクロ.	223
関数類似マクロ.	224
マクロ名のスコープ (#undef)	228
# 演算子	228
## 演算子とのマクロ連結	229
プリプロセッサの Error ディレクティブ (#error)	230
プリプロセッサの warning ディレクティブ	
(#warning)	230
ファイルのインクルード (#include)	231
特殊化されたファイルのインクルード	
(#include_next)	232
ISO 規格事前定義マクロの名前	232
条件付きコンパイル・ディレクティブ	234
#if, #elif	235
#ifdef	236
#ifndef	236
#else	237
#endif	237
行制御 (#line)	238
ヌル・ディレクティブ (#)	239
プラグマ・ディレクティブ (#pragma)	239
標準プラグマ	240
_Pragma 演算子.	240
第 10 章 ネーム・スペース	241
ネーム・スペースの定義.	241
ネーム・スペースの宣言.	241
ネーム・スペース別名の作成	242
ネストされたネーム・スペースの別名の作成	242
ネーム・スペースの拡張.	242
ネーム・スペースおよび多重定義.	243
名前なしネーム・スペース	244
ネーム・スペース・メンバー定義.	245
ネーム・スペースおよびフレンド.	246
using ディレクティブ.	246
using 宣言およびネーム・スペース	247
明示的アクセス.	248
第 11 章 多重定義	249
関数の多重定義.	249
多重定義された関数の制約事項	250
演算子の多重定義	251
単項演算子の多重定義	253
増分と減分の多重定義	254
2 項演算子の多重定義	255
代入の多重定義.	256
関数呼び出しの多重定義.	257
添え字の多重定義	258
クラス・メンバー・アクセスの多重定義	259
多重定義解決	260
暗黙の変換シーケンス	261
多重定義された関数のアドレスの解決	262
第 12 章 クラス	265
クラス・タイプの宣言	265

クラス・オブジェクトの使用	266
クラスと構造体.	268
クラス名のスコープ	269
不完全なクラス宣言	269
ネスト・クラス.	270
ローカル・クラス	272
ローカル型名	273

第 13 章 クラス・メンバーとフレンド 275

クラス・メンバー・リスト	275
データ・メンバー	276
メンバー関数	277
const および volatile メンバー関数	278
仮想メンバー関数	278
特殊なメンバー関数	279
メンバー・スコープ	279
メンバーへのポインター.	281
this ポインター.	282
静的メンバー	285
静的メンバーでのクラス・アクセス演算子の使用	285
静的データ・メンバー	286
静的メンバー関数	288
メンバー・アクセス	290
フレンド.	292
フレンドのスコープ	294
フレンドのアクセス	297

第 14 章 継承 299

派生	301
継承されたメンバー・アクセス	304
protected メンバー.	304
基底クラス・メンバーのアクセス制御	306
using 宣言およびクラス・メンバー	307
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	308
クラス・メンバーのアクセスの変更	309
多重継承.	311
仮想基底クラス.	312
マルチアクセス.	313
あいまいな基底クラス	314
仮想関数.	318
あいまいな仮想関数呼び出し	322
仮想関数のアクセス	323
抽象クラス	324

第 15 章 特殊なメンバー関数 327

コンストラクターとデストラクターの概要.	327
コンストラクター	329
デフォルト・コンストラクター	329
コンストラクターでの明示的初期化	330
基底クラスおよびメンバーの初期化	332
派生クラス・オブジェクトの構築順序	336
デストラクター.	337
フリー・ストア.	339
一時オブジェクト	344
ユーザー定義の型変換	345

コンストラクターによる変換	347
型変換関数	348
コピー・コンストラクター	349
コピー代入演算子	351

第 16 章 テンプレート 353

テンプレート・パラメーター	354
「型」テンプレート・パラメーター	354
「非型」テンプレート・パラメーター	354
「テンプレート」テンプレート・パラメーター	355
テンプレート・パラメーターのデフォルトの引き数	355
テンプレート引き数	356
テンプレート型引き数	356
テンプレート非型引き数	357
「テンプレート」テンプレート引き数	359
クラス・テンプレート	360
クラス・テンプレートの宣言と定義	361
静的データ・メンバーとテンプレート	362
クラス・テンプレートのメンバー関数	362
フレンドとテンプレート	363
関数テンプレート	364
テンプレート引き数の推定	365
関数テンプレートの多重定義	371
関数テンプレートの部分選択	372
テンプレートのインスタンス化	372
暗黙のインスタンス化	373
明示的インスタンス生成	374
テンプレート特殊化	376
明示的特殊化	376
部分的特殊化	380
名前空間のバインディングおよび従属名	383
typename キーワード	384
修飾子としてのキーワード・テンプレート	384

第 17 章 例外処理 387

try キーワード	387
ネストされた try ブロック	389
catch ブロック	389
関数 try ブロック・ハンドラー	390
catch ブロックの引き数	393
スローされた例外とキャッチされた例外とのマッチング	394
キャッチの順序	394

throw 式	396
例外の rethrow	396
スタック・アンwind	398
例外の指定	399
特殊な例外処理関数	402
unexpected()	402
terminate()	403
set_unexpected() と set_terminate()	404
例外処理関数の使用例	405

付録 A. IBM C 言語拡張機能 409

直交拡張機能	409
個々のオプション・コントロールが備わった既存の IBM C 拡張機能	409
IBM C 拡張機能: C89 の拡張機能としての C99 フィーチャー	410
GNU C に関連する IBM C 拡張機能	411
非直交拡張機能	411
個々のオプション・コントロールが備わった既存の IBM C 拡張機能	412
IBM C 拡張機能: C89 の拡張機能としての C99 フィーチャー	412
GNU C に関連する IBM C 拡張機能	412

付録 B. IBM C++ 言語拡張機能 413

直交拡張機能	413
C99 との互換性のための IBM C++ 拡張機能	413
GNU C に関連する IBM C++ 拡張機能	414
GNU C++ に関連する IBM C++ 拡張機能	415
非直交拡張機能	415
C99 との互換性のための IBM C++ 拡張機能	415
GNU C に関連する IBM C++ 拡張機能	416

付録 C. 言語機能に関連した定義済みマクロ 417

特記事項 421

プログラミング・インターフェース情報	423
商標	423
業界標準	423

索引 425

本書について

本書、「C/C++ ランゲージ・リファレンス」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。プログラム言語の完全な仕様は構文とセマンティクスから構成されますが、言語仕様の準拠するインプリメンテーションは、言語拡張のため、それぞれ異なる可能性があります。C および C++ の IBM インプリメンテーションは、プログラム言語の基本的な性質を立証するものであると同時に、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++ の言語拡張機能は、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。

このリファレンスの目的は、C 言語および C++ 言語について説明し、また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、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固有の用語です。同じ命名規則が、C++ 言語にもあります。このリファレンスでは、*Standard C* および *Standard C++* に準拠するインプリメンテーションについて説明します。また、コンパイラーでは以前の言語レベルもサポートされます。

あいまいさと K&R C との混同を避けるために、本書では、C 言語と、C についての ISO 標準化よりも以前に使用されていて、Brian Kernighan および Dennis Ritchie (K&R C) によって作られ、かつ一般に受け入れられている拡張とを表す場合、*Classic C* という用語を使用しています。

本書は、C 言語および C++ 言語の基礎および応用に焦点をあてています。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、「XL C/C++ コンパイラー・リファレンス」で説明されています。

本書で説明する C および C++ 言語は、以下の規格に基づいています。

- *Programming languages - C* (ISO/IEC 9899:1990) に記述されている C 言語。本書では C89 と呼びます。これは、最初の ISO C 規格です。
- *Programming languages - C* (ISO/IEC 9899:1999) に記述されている C 言語。本書では C99 と呼びます。これは、C89 規格の更新版です。
- *Programming languages - C++* (ISO/IEC 14882:1998) に記述されている C++ 言語。これは、最初の正式な言語定義です。本書では C++98 と呼びます。
- *Programming languages - C++* (ISO/IEC 14882:2003 (E)) に記述されている C++ 言語。これは、*Standard C++* という語の現在の意味です。

本書で説明する C 言語は、C99 と整合性があり、XL C/C++ がサポートするフィーチャーを文書化しています。コンパイラーは、規格で指定されるすべての言語フィーチャーをサポートします。また、この規格では、ランタイム・ライブラリーのフィーチャーも指定していることに注意してください。これらのフィーチャーは、現行のランタイム・ライブラリーおよびオペレーティング環境ではサポートされないことがあります。

本書で説明する C++ 言語は、Standard C++ と整合性があり、IBM C++ コンパイラがサポートするフィーチャーを文書化しています。本書では、GNU C および C++ コンパイラのバージョン 3.2 以降で開発されたプログラムを移植する場合に役立つ、言語拡張についても説明します。

説明の範囲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機能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。

IBM 言語拡張機能

IBM C および C++ コンパイラは、さまざまな言語標準に基づき、使用可能度を高めて、異なるプラットフォームへのプログラムの移植を容易にする言語拡張機能をサポートします。

ベース部分に対する拡張の概念を紹介するために、以下の言語仕様を “ベース言語レベル” として参照します。

- Standard C++
- C++98
- C99
- C89

さらに、C89 が標準化される前に C インプリメンテーションによって一般に使用された事実上の K&R 業界標準について言及する目的で *Classic C* も使用します。

直交拡張機能 とは、既存の言語フィーチャーの振る舞いを変更することなく、ベース部分の上に追加されるフィーチャーのことです。ベース・レベルに準拠している有効なプログラムは、このような拡張機能を使用した場合も、依然として正しくコンパイルされ、実行されます。プログラムはなおも有効で、その振る舞いは直交拡張機能が存在しても以前と変わりません。このような拡張機能は、対応する基本標準レベルとの整合性を持っています。無効なプログラムは、実行時およびコンパイラが出す診断において、異なった振る舞いをする可能性があります。

一方、**非直交拡張機能** とは、既存構成のセマンティクスを変更したり、ベースと競合する構文を導入したりすることができる拡張機能のことです。ベースに準拠している有効なプログラムであっても、非直交拡張機能を使用した場合に正しくコンパイルおよび実行されることが保証されません。このため、このような拡張機能を使用可能にするために、個々のコンパイラ・オプションが提供されます。

Standard C、C++98、C99 および C89 の言語レベルは、標準への準拠について厳格に規定しています。Classic C は、概して 事実上の K&R 業界標準に準拠しています。これらの言語レベルは、`-qlanglvl` コンパイラー・オプションを使用して選択することができます。標準レベルに対する拡張も、このオプションを使用して指定することができます。例えば、`-qlanglvl=stdc99` は Standard C99 を指定し、`-qlanglvl=extc99` は C99 および直交拡張を指定します。`-qlanglvl` およびそのサブオプションについての詳細は、「*XL C/C++ コンパイラー・リファレンス*」を参照してください。

C89 に対する直交拡張であるコンパイラー・オプション `digraph`、UCS 文字、`long long`、および `dollar` など、以前から存在するオプションは引き続きサポートされます。C++ は、拡張フィーチャーに対する個々のオプション制御も提供します。

GNU C および C++ に関連するフィーチャー

移植を容易にするために、GNU C および C++ フィーチャーに対応する特定の言語拡張機能がインプリメントされました。これらの拡張機能として、C89、C99、および C++98 に対する直交および非直交の拡張があります。これらは、前のセクションで説明したとおり、`-qlanglvl` コンパイラー・オプションによって制御されます。

GNU C に関連する直交拡張機能の例は、関数宣言および定義で `noreturn` 関数属性を指定しています。コンパイラーには、関数が戻らないことが伝えられ、これによってパフォーマンスがよくなることもありますが、規格合致プログラムは、このフィーチャーによる影響を受けることはありません。`noreturn` 関数属性のセマンティクスは、直交 です。

非直交拡張機能の例は、**`inline`** キーワードです。これが非直交である理由は、現行の GNU C セマンティクスが C99 のセマンティクスと異なるからです。

強調表示の規則

太字	コマンド、キーワード、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、新規用語を最初に言及する際にも使用されます。
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。例では、使用しうるすべての言語構成の使用法を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。

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++ のコマンド、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。

▶▶—statement—required_item—▶◀

- オプション項目は、メインパスの下に記述されます。

▶▶—statement—
| 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。

▶▶—statement—
| 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。

▶▶—statement—
| optional_choice1 | —▶◀
| optional_choice2 |

デフォルト項目は、メインパスの上に記述されます。

▶▶—statement—
| default_item | —▶◀
| alternate_item |

- 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。

▶▶—statement—
| repeatable_item | —▶◀

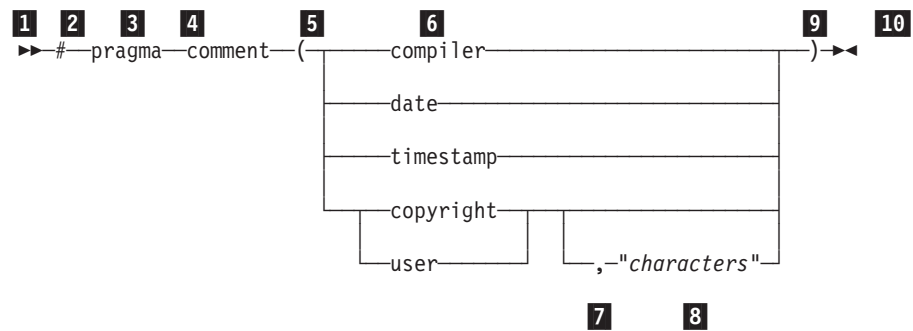
スタックの上の繰り返し矢印は、スタック内の項目から複数の項目を選択するか、1 つの項目を繰り返し選択できることを示しています。

- 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*)。変数は、ユーザー提供の名前または値を表します。

- 構文図に、句読記号、小括弧、算術演算子、または、ほかの同様な記号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、239 ページの『プラグマ・ディレクティブ (#pragma)』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、記号 # 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")
```

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、つまり名前が有効な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、関連していますが、概念は異なります。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。

したがって、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、つまりオブジェクトがストレージの指定領域に存続する期間と相互に関連します。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・ディレクティブ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。

関連参照

- 7 ページの『プログラム・リンケージ』

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である必要があります。ただし、このオブジェクトにアクセスするための名前、つまり ID 自体は再使用することができます。ID の意味は、ID が使用されるコンテキストによって変わります。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。

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。不連続スコープの可能性を活用することは、情報隠蔽 技術の基礎です。

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。

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)	グローバル・ネーム・スペース
	ネーム・スペース
	class

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・ネーム・スペース・スコープではありません)。

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

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。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。

クラスのフレンドとして最初に宣言された関数名は、クラスを囲む、最内部の非クラスのスコープに入っています。外部ネーム・スペースで最初に宣言される関数名は、フレンド宣言として使用されません。以下に例を示します。


```
int f();

namespace A {
    class X {
        friend f(); // refers to A::f() not to ::f();
    }
    f() { /* definition of f() */ }
}
```

フレンド関数が別のクラスのメンバーの場合は、この関数にはその別のクラスのスコープがあります。クラスのフレンドとして最初に宣言された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。

クラスの暗黙宣言は、同じクラスの別の宣言が見えるまでは可視になりません。

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。

関連参照

- 204 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** 文の中で、そのラベルを使用することができません。

関連参照

- 201 ページの『ラベル』

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。

関連参照

- 170 ページの『関数宣言』

グローバル・スコープ

C 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、グローバル・スコープがあります。グローバル・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++ 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。

関連参照

- 241 ページの『第 10 章 ネーム・スペース』
- 7 ページの『内部結合』

クラス・スコープ

C++ 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。また、宣言のスコープには、メンバー定義の ID に続く宣言子の一部も含まれます。

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

関連参照

- 265 ページの『第 12 章 クラス』
- 269 ページの『クラス名のスコープ』
- 306 ページの『基底クラス・メンバーのアクセス制御』

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を一意的に識別する必要があります。ここで使用するネーム・スペース という用語は、C および C++ に適用されるものであり、C++ のネーム・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。
- 次を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称
 - 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 name[20]; /* this structure member may not be named student */
    }
}
```

ID のネーム・スペース

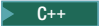
```
    int section;
    int id;
} sam;          /* this structure variable should not be named student */

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

student fred;   /* legal struct declaration in C++ */
}
```

コンパイラーは `student` が発生するたびに、プログラム内のコンテキストに基づいて解釈します。例えば、キーワード `struct` の後に `student` が表示された場合は、これは構造体タグを意味します。名前 `student` は `struct student` の構造体メンバーに対しては使用できません。 `goto` 文の後に `student` が現れたときは、コンパイラーは、ヌル・ステートメント・ラベルに制御を渡します。ほかのコンテキストでは、ID `student` は、構造変数を参照します。

名前の隠蔽

 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。このことを以下の例で示します。

```
#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。

関連参照

- 265 ページの『第 12 章 クラス』
- 277 ページの『メンバー関数』
- 279 ページの『メンバー・スコープ』
- 241 ページの『第 10 章 ネーム・スペース』

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。リンケージには外部結合、内部結合、リンケージなしの 3 つのタイプがあります。

- 外部結合 を持つ ID は、他の変換単位内で参照することができます。
- 内部結合 を持つ ID は、変換単位内でのみ参照することができます。
- リンケージなし の 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++** 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もできます。言語リンケージを使用すると、C で記述されたコードに C++ コードをリンクできるようにして、C++ と C の間の関係をクローズの状態にすることができます。すべての ID には、デフォルトで C++ の言語リンケージが設定されています。言語リンケージは変換単位間において一貫性がある必要があります。C++ 以外の言語リンケージの場合は、ID には外部結合が設定されます。

内部結合

次の種類の ID は、内部結合を持っています。

- 明示的に **static** を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** を宣言された関数テンプレート。
- ▶ **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 グローバル・スコープでは、**static** ストレージ・クラスの指定子なしで宣言された、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。

- オブジェクト。
- 関数。

C の ID が **extern** キーワードで宣言され、同じ ID を持つ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、ID は、最初の宣言と同じリンケージを持っています。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++ 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。

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を指すクラスまたは列挙の名前、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。


```
        ++str;
    }
    putchar('\n');
}
```

ネーム・マングリング

C++ ネーム・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。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き数の型をエンコードして、関数名を生成します。ネーム・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。ネーム・マングリングは、変数名にも適用されます。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、ネーム・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リング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のマングリングが行われないようにして、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 */
    }
};
```

複数レベルのネストした **extern** 宣言では、宣言内部で **extern** の指定が一般的に用いられます。


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

この例では、func() に C++ リンケージが含まれています。

第 2 章 字句エレメント

字句エレメント とは、ソース・ファイルで字句として使用してもよい 1 つの文字、あるいは複数の文字よりなるグループのことをいいます。本節では、基本字句エレメントならびに C および C++ プログラム言語の規則、すなわち、トークン、文字セット、コメント、ID、およびリテラルについて説明します。

トークン

プリプロセッサ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、コンパイラによって定義されている、プログラムの意味の最小独立単位です。トークンには、次の 5 つの型があります。

- ID
- キーワード
- リテラル
- 演算子
- 区切り子

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。

区切り子

区切り子 は、コンパイラに対して構文上意味を持つトークンですが、正確な重要度はコンテキストによって異なります。区切り子は、プリプロセッサの構文で使用するトークンとすることもできます。C89 言語レベルで、区切り子はアクションを起こしません。例えば、コンマは、引き数リストまたは初期化指定子リストの区切り子ですが、括弧で囲んだ式の中で使用される場合は演算子です。

C89 言語レベルで、区切り子は、以下のようなトークンを分離する文字とすることができます。

[] () { } , ; ;

または、以下のもの。


* = ... #

C89 では、番号記号 # の使用は、プリプロセッサ・ディレクティブのみに制限されています。

C99 言語レベルでは、区切り子の適正なトークンまたはプリプロセッサ・トークンの数は増加し、C 演算子を包含します。実行するオペレーションを指定する区切り子は、演算子 として知られます。区切り子と演算子の区別は、C++ によっても使用されます。C89 区切り子に加え、C99 は以下のトークンを区切り子、演算子、またはプリプロセッサ・トークンとして定義します。

トークン

.	->	++	--	##
&	+	-	~	!
/	%	<<	>>	!=
<	>	<=	>=	==
^		&&		?
*=	/=	%=	+=	-=
<<=	>>=	&=	^=	=
<:	:>	<%	%>	%;
				%;%:

 C99 プリプロセス・トークン、演算子、および区切り子に加え、C++ は以下のトークンを区切り子として許可します。

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

代替トークン

C および C++ は、いくつかの演算子および区切り子に代替表記を提供します。次の表には、演算子および区切り子と、それらの代替表記がリストされています。

演算子または区切り子	代替表記
------------	------

{	<%
}	%>
[<:
]	:>
#	%;
##	%;%:
&&	and
	bitor
	or
^	xor
~	compl
&	bitand
&=	and_eq
=	or_eq
^=	xor_eq
!	not
!=	not_eq

ソース・プログラムの文字セット

コンパイル時および実行時の両方に使用可能にする必要がある、基本ソース文字セット を次にリストします。

- 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

- 0 から 9 までの 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)。
- ASCII の分割垂直バー (|) 文字。
- 空白文字
- 改行、水平タブ、垂直タブ、改ページ、およびストリングの終了 (NULL 文字) を表す制御文字

インプリメンテーションおよびコンパイラー・オプションに応じて、ドル記号 (\$) や国別文字セットの文字など、それ以外の特殊な識別文字を ID で使用できる場合もあります。

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・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内で使用され、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 文字コードを使用しているシステムでは、エスケープ・シーケンス ¥x56 の値は、

大文字の V です。EBCDIC 文字コードを使用しているシステムでは、エスケープ・シーケンス `¥xE5` の値は、大文字の 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、筆記文字およびテキストのためのエンコード・スキームの仕様です。ユニコード規格は、多言語テキストの整合したエンコードを可能にし、テキスト・データを矛盾を起こすことなく国際間で交換できるようにする汎用標準です。C および C++ の ISO 規格とは、*ISO/IEC 10646-1:2000, Information Technology—Universal Multiple-Octet Coded Character Set (UCS)* を指します。(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、エンコード形式の数においては、ユニコード規格より制限があります。したがって、ISO/IEC 10646 に準拠する文字セットは、ユニコード規格にも準拠します。

ユニコード規格は、文字ごとに固有の数値と名前を指定しており、数値のビット表記に対して 3 つのエンコード形式を定義しています。名前と値のペアで 1 つの文字の識別が作成されます。1 つの文字を表す 16 進数値をコード・ポイントと呼びます。ユニコード仕様は、大/小文字、文字記述の方向、英字の特性、その他のセマンティック情報など、文字ごとに文字の特性全体についても記述しています。ユニコード規格は ASCII をモデルとして、英字、漢字、および記号に対応し、予約済みのコード・ポイントの範囲内でインプリメンテーションの定義による文字コードを許可しています。したがって、ユニコード規格のエンコード・スキームは、世界中のあらゆる国の歴史的な筆記体も含めて、既知のすべての文字エンコードに対応できるだけの柔軟性を十分に持っています。

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。C99 言語と C++ 言語のどちらも ID、文字定数、およびストリング・リテラルに汎用文字名を使用することを許可しています。C++ では、この言語フィーチャーはコンパイル時に指定された言語レベルには影響されません。

次の表に、一般的な汎用文字名構成および ISO/IEC 10646 の短縮名との対応を示します。

汎用文字名	ISO/IEC 10646 の短縮名
¥UNNNNNNN	NNNNNNNN
¥unNNNN	0000NNNN

ここでは、N は 16 進数字です。

C99 および C++ では、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 進数値、および ISO/IEC 10646 で制御文字用に予約されているコード・ポイントは許可されません。 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。

XL C/C++ は、データ型 `uint_least16_t` および `uint_least32_t` をインプリメントして、ユニコード規格に準拠して C および C++ の UTF-16 文字および UTF-32 文字を処理します。データ型はまた、それぞれ *u-literals* および *U-literals* として参照される、 UTF-16 または UTF-32 文字を指定するためにユニコード規格に必要なストリング・リテラルであり、C Standards Committee に承認されています。以前は、UTF-16 文字は、**unsigned short** で、 UTF-32 文字は **unsigned int** で表されました。

u-literals および *U-literals* のサポートは、ワイド文字リテラルのサポートに似ています。

`u"s-char-sequence"`
`uint_least16_t` の配列を示します。対応する文字リテラルは、以下によって示されます。

`U'c-char-sequence'`

`U"s-char-sequence"`
`uint_least32_t` の配列を示します。対応する文字リテラルは、以下によって示されます。

`U'c-char-sequence'`

例えば、次のようにします。

```
uint_least16_t msg[] = u"ucs characters ¥u1234 and ¥U81801234 ";
```

ストリング連結

u-literals および *U-literals* は、ワイド文字リテラルと同じ連結規則に従います。通常の文字ストリングが存在する場合、広げられます。以下に、許可された組み合わせを示します。他のすべての組み合わせは無効です。

組み合わせ	結果
<code>u"a" u"b"</code>	<code>u"ab"</code>
<code>u"a" "b"</code>	<code>u"ab"</code>
<code>"a" u"b"</code>	<code>u"ab"</code>
<code>U"a" U"b"</code>	<code>U"ab"</code>
<code>U"a" "b"</code>	<code>U"ab"</code>
<code>"a" U"b"</code>	<code>U"ab"</code>

これらの規則が再帰的に適用され、複数の連結が許可されます。

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 文字表記文字 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	1 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)

プリプロセッサが、3 文字表記を対応する単一文字表示に置換します。

マルチバイト文字

マルチバイト文字 とは、そのビット表記が 1 バイトまたは複数バイトに適合し、かつ拡張文字セット のメンバーである文字のことです。拡張文字セットは、基本文字セットのスーパーセットです。ワイド文字 とは、そのビット表記が `wchar_t` 型のオブジェクトに対応し、現行ロケールの任意の文字を表示できる文字のことです。

関連参照

- 55 ページの『`char` および `wchar_t` 型指定子』

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。

以下のとおり、2 種類のコメントがあります。

- `/*` (スラッシュ、アスタリスク) 文字があり、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はネストできません。`*/` が最初に現れた位置で、各コメントは終了します。

マルチバイト文字は、コメントにも含めることができます。

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ので、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 `*/` までは、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

コメント

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・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う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;
}
```

関連参照

- 18 ページの『3 文字表記』

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
- Typedef
- 列挙型および列挙子
- 構造体および共用体の名前
-  クラスおよびクラス・メンバー
-  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、次の形式の任意の数を持つ文字、数字、あるいは下線文字で構成されます。



基本ソース文字集合外の文字および数字の汎用文字名は、C++ および C99 言語レベルで使用できます。

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 1 文字の下線で始まる ID は、標準のネーム・スペースおよびタグのネーム・スペースのどちらにおいても、ファイル・スコープを持つ ID として予約されています。

C++ C++ は、より大きなネーム・スペースにさらに多くの ID を組み込むために、C の予約を拡張しています。二重下線を任意の位置に含む名前はずべて予約されています。1 つの下線で始まる ID はすべて、グローバル・ネーム・スペースで予約されています。

ID での大/小文字の区別と特殊文字

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。

関数名や変数名に、下線 (`_`) で始まる ID を使用しないでください。

ID の先頭文字は、文字でなければなりません。 `_` (下線) は文字と見なされます。ただし、下線で始まる ID は、グローバル・ネーム・スペース・スコープでの ID 用に、コンパイラーによって予約されています。

2 つの連続する下線を含んでいる ID、または後に 1 つの大文字が続く 1 つの下線で始まる ID は、すべてのコンテキストで予約されています。

ドル記号は、`-qdollars` コンパイラー・オプションを使用してコンパイルされる場合、あるいはこのオプションを含む拡張言語レベルの 1 つでコンパイルされる場合は、ID に使用することができます。

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。

システム呼び出しおよびライブラリー関数の名前は、該当する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。


事前定義済みの ID

事前定義済みの ID `__func__` を使用すると、関数名をその関数内で使用することができます。各関数定義の左中括弧の直後に、`__func__` が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。

この ID が `assert` マクロとともに使用されると、このマクロは標準エラー・ストリームにこの ID を含んでいる関数の名前を追加します。

 C++ は、C99 と互換性を持たせるための言語拡張として、事前定義済みの ID `__func__` をサポートしています。

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、`C` クラスのメンバー関数です。`foo` の事前定義済み ID は `C::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::C::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、インスタンス化された型を反映しています。例えば、`int` でインスタンス化されたテンプレート関数 `foo` の事前定義済み ID です。

```
template<class T> void foo()
```

は、`foo<int>` です。


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ありません。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。下記に、`C` および `C++` 言語の両方に共通するキーワードをリストします。

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


 C 言語では、次のキーワードも予約されています。

restrict	_Bool	_Complex	uint_least16_t
		_Imaginary	uint_least32_t


 C++ 言語では、次のキーワードも予約されています。

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

言語拡張用のキーワード

 XL C/C++ は、標準言語キーワードのほかに、言語拡張のため、GNU C コンパイラで開発されたアプリケーションを移植しやすくするため、および将来の使用のために、ID を予約しています。言語拡張での使用のために、次のキーワードが予約されています。

typeof	_attribute__	_imag__	_restrict
_align	_complex__	_inline__	_signed__
_alignof__	_const__	_label__	_typeof__
_asm	_extension__	_real__	_volatile__
_asm__			

 C++ の IBM インプリメンテーションは、C99 との互換性を保つための言語拡張として、次のキーワードを予約しています。

restrict	_Complex	uint_least16_t
_restrict__	_Imaginary	uint_least32_t

演算子および区切り子の代替表記

予約済み言語キーワードのほかに、以下の演算子および区切り子の代替表記も C および C++ で予約されています。

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

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 言語では、リテラル という名詞の代わりに、定数 という用語が使用されます。リテラルの という形容詞が付くと、定数の概念に、その定数を値の観点からしか取り扱うことができない、という考え方が追加されます。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。

すべてのリテラル は値とデータ型を持っています。リテラルの値は、プログラムの実行中に変わることはなく、その型で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。

- ブール
- 整数
- 文字
- 浮動小数点
- スtring
- 複合リテラル

C C99 では、複合リテラルを後置式として追加します。この言語フィーチャーを使用すると、集合体または共用体型の定数を指定することができます。

ブール・リテラル

C C 言語ではブール・リテラルは定義されていませんが、その代わりにブール値を表すために整数値の 0 および 1 が使用されます。0 は「false (偽)」を、ゼロ以外のすべての値は「true (真)」を表します。

C では、ヘッダー・ファイル `<stdbool.h>` 内にマクロとして `"true"` および `"false"` が定義されています。これらのマクロが定義されていると、マクロ `__bool_true_false_are_defined` が整数定数 1 に展開されます。

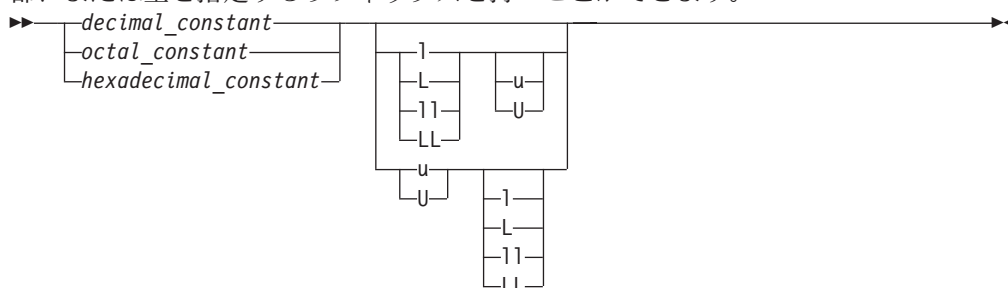
C++ ブール・リテラルには、**true** と **false** の 2 つしかありません。これらのリテラルは、型 **bool** を持っていて、左辺値ではありません。

関連参照

- 55 ページの『ブール変数』
- 109 ページの『左辺値と右辺値』

整数リテラル

整数リテラル は、10 進、8 進、または 16 進の値を表します。これらは、小数点または指数部を持たない数値です。ただし、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。



整数リテラルのデータ型は、定数の形式、値、およびサフィックスにより決まります。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, unsigned long int, long long int
サフィックスがない 8 進数	int, unsigned int, long int, unsigned long int, long long int, unsigned long long int
サフィックスがない 16 進数	int, unsigned int, long int, unsigned long int, long long int, unsigned long long int
u または U のサフィックスが付く 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 か U 、または l か L の両方のサフィックスが付く 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
サフィックスとして ll または LL が付く 8 進数または 16 進数	long long int, unsigned long long int
u か U 、または ll か LL の両方のサフィックスが付く 10 進数、8 進数、または 16 進数	unsigned long long int

プラス (+) またはマイナス (-) 記号は、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 進整数リテラルの例を次に示します。

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

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。

16 進整数リテラル

16 進整数リテラル は、数字 0 で始まり、その後に x または X が続き、その後、0 から 9 までの数字と a から f または A から F までの文字の組み合わせが続きます。A (または a) から F (または f) までの文字は、それぞれ 10 から 15 までの値を表します。



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、0 から 7 までの数字が含まれます。



8 進整数リテラルの例を次に示します。

```
0
0125
034673
03245
```

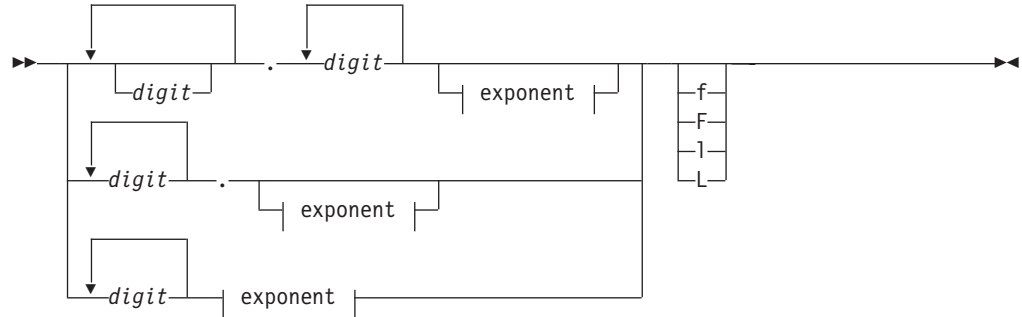
浮動小数点リテラル

浮動小数点リテラル 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
- オプションのサフィックス

整数部と小数部は、両方とも 10 進数で作成されます。整数部か小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。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。



Exponent:



float の大きさの範囲は、おおよそ $1.2\text{e-}38$ から $3.4\text{e}38$ までです。**double** または **long double** の大きさの範囲は、おおよそ $2.2\text{e-}308$ から $1.8\text{e}308$ までです。浮動小数点定数が長すぎたり、短すぎたりする場合は、結果は言語によって定義されていません。

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	0.000000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

C `printf` 関数を使用して、浮動小数点定数値を表示する場合は、指定する `printf` 型変換コード修飾子が、その浮動小数点定数値に対して、十分な大きさであることを確認してください。

関連参照

- 56 ページの『浮動小数点変数』
- 126 ページの『単項式』

16 進浮動小数点定数

16 進浮動小数点定数は、以下のもので構成されます。

- 16 進接頭部
- 有効部分
- 2 進指数部
- オプションのサフィックス

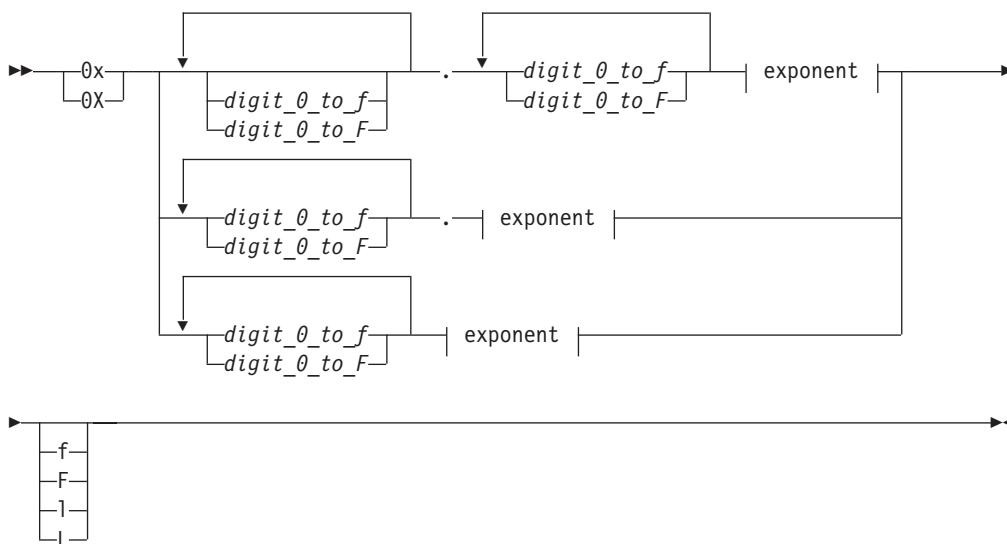
C++ C++ は、C99 との互換性のために、16 進浮動小数点定数をサポートするように Standard C++ および C++98 を拡張します。

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

オプションの小数部は、ピリオドの後に 16 進数字が続きます。

指数部は、有効部分を 2 のべき乗することを示し、オプションで符号付きの 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。



Exponent:



整数部または小数部のいずれか一方は省略できますが、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 進数字と間違われるのを避けるためです。

複素数リテラル

複素数リテラル型は、複素数を表します。事前定義マクロ `_Complex_I` は、虚数単位の値を使用して、型 `const float _Complex` の定数式を表します。例えば、次のようにします。

```
float _Complex varComplex = 2.0f + 2.0f*_Complex_I;
```

は、型 `float _Complex` に対する変数 `varComplex` を初期化します。

複素数型は、サフィックス `i`、`I`、`j`、または `J` のいずれかで示すこともできます。複素数の実数部は、サフィックス `f`、`F`、`l`、または `L` の 1 つで示すことができます。これらのサフィックスは、GNU C で開発されたアプリケーションを移植しやすくするための C99 の拡張機能です。

複素数リテラルを単純化した構文を次に示します。

```
▶▶ floating-constant | complex-suffix ▶▶
```

floating-constant:

```
decimal-floating-constant
hexadecimal-floating-constant
```

complex-suffix:

```
floating-suffix suffixij
suffixij floating-suffix
```

ここで、

floating-suffix `f`、`F`、`l` (小文字の `L`) または `L` の 1 つ。

サフィックス `f` または `F` は、型 `float _Complex` の複素数リテラルを示します。サフィックス `l` または `L` は、型 `long double _Complex` の複素数リテラルを示します。複素数リテラルは、サフィックスがない型 `double _Complex` です。

suffixij `i`、`I`、`j`、`J` の 1 つ。

関連参照

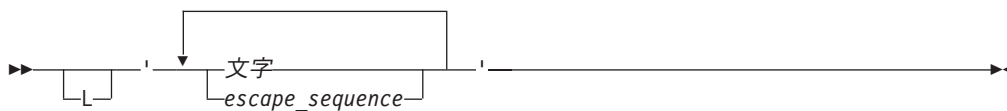
- 127 ページの複素数型の単項演算子

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。

文字リテラルの形式は、次のとおりです。



文字リテラルには、1 つ以上の文字またはエスケープ・シーケンスがなければなりません。この文字には、ソース・プログラムの文字セットのどの文字でも使用できます。ただし、一重引用符、円記号、および改行記号は除きます。基本ソース文字集合外の文字の汎用文字名を使用できます。文字リテラルは、1 行の論理ソース行に表す必要があります。

C 文字リテラルは、**int** 型を持ちます。

C++ 文字を 1 つだけ含んでいる文字リテラルは、整数型である型 **char** を持っています。

C と **C++** のどちらにおいても、ワイド文字リテラルは **wchar_t** 型を持っており、複数文字リテラルは **int** 型を持っています。

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。複数の文字またはエスケープ・シーケンスを含む狭幅の文字リテラルまたはワイド文字リテラルの値は、インプリメンテーション定義されています。

二重引用符記号は、それ自体で二重引用符記号を表すことができます。しかし、一重引用符記号を表すには、円記号の後に一重引用符記号の付いたもの (`¥'` エスケープ・シーケンス) を使用する必要があります。

改行文字は、`¥n` 改行エスケープ・シーケンスによって表すことができます。

円記号文字は、`¥¥` 円記号エスケープ・シーケンスによって、表すことができます。

文字リテラルの例を次に示します。

```
'a'
'¥'
L'0'
'('
```

関連参照

- 55 ページの『`char` および `wchar_t` 型指定子』
- 16 ページの『ユニコード規格』

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。



基本ソース文字集合外の文字の汎用文字名を使用できます。

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** 型の配列で、ワイド・ストリング・リテラルの型は、**wchar_t** 型の配列です。

C++ 狭幅ストリング・リテラルの型は **const char** の配列で、ワイド・ストリング・リテラルの型は **const wchar_t** の配列です。両方の型には、静的ストレージ期間が指定されています。

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 *mail_addr = "Last Name   First Name   MI   Street Address ¥
                  City   Province   Postal code ";
char *temp_string = "abc" "def" "ghi"; /* *temp_string = "abcdefghi¥0" */
wchar_t *wide_string = L"longstring";
```

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・リテラル内に含まれている複数のスペースは保存されます。

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および改行文字 (必要) を使用します。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. */
```

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。ワイド・ストリング・リテラルと狭幅ストリング・リテラルが相互に隣接する場合、その結果の振る舞いについては未定義です。次の例は、このことを示しています。

```
"hello " "there" /* is equivalent to "hello there" */
"hello " L"there" /* the behavior at the C89 language level is undefined */
"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` を形成するためにマージされることはありません。

C ワイド・ストリング・リテラルと狭幅ストリング・リテラルが隣接すると、その結果はワイド・ストリング・リテラルになります。

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。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。ワイド・ストリング・リテラルに対して、**wchar_t** の `'¥0'` が付加されます。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" */
```

複合リテラル

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。初期化指定子リストの式は定数であってもかまいません。C99 言語フィーチャーを使用すると、初期化指定子および式で複合定数を使用して、集合体型または共用体型の定数を指定することができます。これらの型の内の 1 つのインスタンスが一度しか使用されない場合は、複合リテラルを使用することで、一時変数を使用する必要がなくなります。C++ は、C と互換性を持たせるための、Standard C++ への拡張機能として、このフィーチャーをサポートします。

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。構文は以下のとおりです。

```
▶▶ (—type_name—) { —initializer_list— }
                        └──initializer_list──┘
```

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。

複合リテラルは、関数の本体の外で使用されている場合は、**静的**ストレージ期間を持ちます。初期化指定子リストは定数式で構成されます。それ以外の場合は、複合リテラルはそれを含むブロックに関連付けられている自動ストレージ期間を持ちます。次の式の意味はそれぞれ違います。複合リテラルは、関数の本体内で使用されると、自動ストレージ期間を持ちます。

```
"string"           /* an array of char with static storage duration */
(char[]){ "string" } /* modifiable */
(const char[]){ "string" } /* not modifiable */
```

const 修飾された複合リテラルは、読み取り専用メモリー内に配置できます。

const 修飾された型を持つ複合リテラルは、同じ表記または重複表記を持つストリング・リテラルと、ストレージを共用できます。例えば、次のような場合です。

```
(const char[]){ "string" } == "string"
```

ストレージを共有している場合、1 が生成される可能性があります。ただし、**const** 修飾された型を持つ複合リテラルは必ずしも共用されません。次の式は、`struct s` 型の 2 つの異なるオブジェクトになります。

```
(const struct s){1,2,3}  
(const struct s){1,2,3}
```

リテラル

第 3 章 宣言

宣言 では、プログラムで使われるデータ・オブジェクトや関数の名前および特性が確立されます。定義 は、データ・オブジェクトにストレージを割り振るか、あるいは関数の本体を指定し、ID をオブジェクトまたは関数に関連付けます。型を宣言または定義すると、ストレージは割り振られません。

宣言によって、オブジェクトの内部関連属性 (ストレージ・クラス、型、スコープ、可視性、ストレージ期間、およびリンケージ) がさまざまな方法で決定されます。

宣言概要

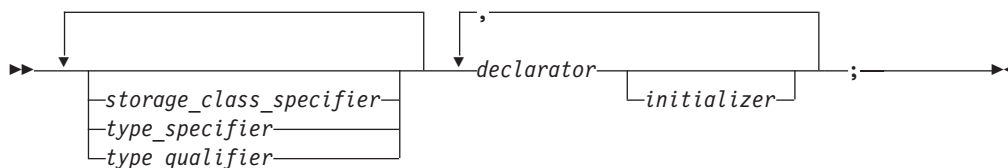
宣言は、データ・オブジェクトおよびそれらの ID の以下の属性を決定します。

- 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振り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の字句の順序は、次のとおりです。

- ストレージ期間およびリンケージ指定
- タイプ指定
- 宣言子。ID を定義し、型修飾子およびストレージ修飾子を使用します
- 初期化指定子。ストレージを初期値で初期化します

すべてのデータ宣言の形式は、次のとおりです。



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、ストレージを割り振りません。これらの ID は、対応する定義を参照します。関数の場合は、対応する定義は、関数のコードまたは本体です。2 番目の列に宣言されている ID は、ストレージを割り振ります。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
<code>extern double pi;</code>	<code>double pi = 3.14159265;</code>
<code>float square(float x);</code>	<code>float square(float x) { return x*x; }</code>
<code>struct payroll;</code>	<pre>struct payroll { char *name; float salary; } employee;</pre>

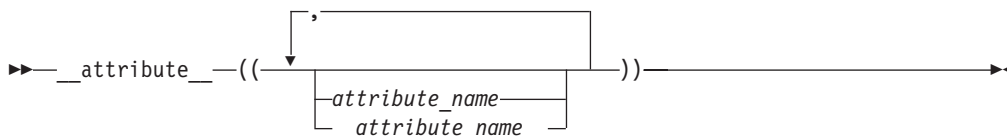
関連参照

- 87 ページの『第 4 章 宣言子』

変数属性

変数属性は、GNU C/C++ コンパイラーを使用して開発されたプログラムの処理を容易にするために提供される直交言語拡張です。この言語フィーチャーを使うと、名前付き属性を使用して変数の宣言を変更できます。本節では、構文およびサポートされる変数属性について説明します。サポートされない属性名を指定した場合、IBM C/C++ コンパイラーは診断を実行し、その属性指定を無視します。

キーワード `__attribute__` は、変数属性を指定します。属性構文は、次のような一般的な形式です。



属性指定子は、宣言指定子であるため、宣言内で宣言子の前に表示できます。属性指定子は、宣言子の後に表示することもできます。この場合、宣言子のコンマで区切られたリスト内にある特定宣言子だけに適用されます。

`__attribute_name__` 形式（つまり、2 つの先導および後続する下線文字が付いた変数属性キーワード）を使用する変数属性指定により、同じ名前のマクロと名前が競合する可能性が減ります。

関連参照

- 174 ページの『関数属性』
- 50 ページの『型属性』

aligned 変数属性

変数属性 **aligned** を使用すると、変数または構造体メンバーに対して、バイト単位で最小限の位置合わせを指定することができます。位置合わせを指定することによって、コンパイラーは、この方法で位置合わせされた変数または構造体メンバーとの間でコピーを行うときに、メモリーの最大量をコピーする命令を使用することができるため、コピー操作の効率が上がります。

aligned 変数属性が自動変数に適用されると、位置合わせは、スタックの最大の位置合わせによって限定されます。属性 **aligned** がビット・フィールド構造体メンバーに適用されると、コンテナの位置合わせが位置合わせ係数より大きい場合を除き、ビット・フィールド・コンテナは位置合わせ仕様に従って調整されます。この場合、属性 **aligned** は無視されます。

```

▶▶ __attribute__((aligned | __aligned__ | (alignment_factor)))▶▶

```

ここで、*alignment_factor* は、1 または 2 の正の累乗値を求める定数式です。

位置合わせ係数（およびそれを囲む括弧）を省略すると、コンパイラーが位置合わせを決定することができます。この場合の位置合わせは、ターゲット・マシン上で処理できる任意の普通型（つまり、整数または実数）に対する最大の位置合わせ、そのものとなります。

aligned 属性は、位置合わせを増加させるだけです。**packed** 属性を使うと、それを減少できます。プラットフォームの最大量より大きな位置合わせ係数は、警告とともに無視され、その結果は予測できません。

init_priority 変数属性

▶ **C++** 変数属性 **init_priority** は、単一コンパイル単位内のネーム・スペース・スコープで定義されたオブジェクトの初期化の順番を制御できる C++ への直交拡張です。属性は、初期化の相対優先順位を示すパラメーターを取ります。数字が低いほど、優先順位が高いことが示されます。

構文は以下のとおりです。

```

▶▶ type_specifier declarator __attribute__((init_priority | __init_priority__ | (relative_priority)))▶▶

```

ここで、*relative_priority* は包括的な 101 から 65535 までの定数整数式です。

mode 変数属性

変数属性 **mode** を使用すると、変数宣言の型指定子をオーバーライドできます。型指定子で指定された元の型は、特定サイズの整数型によってオーバーライドされます。サイズは、モード・パラメーターの値によって示されます。例えば、**__word__** のモード値は、サイズが 4 バイトである整変数になります。元の型指定子の符号は保存されます。

属性 **mode** の有効な引き数は、**byte**、**word**、**pointer**、およびこれら 3 つのモードに先行および後続する 2 つの下線が付いた形式です。

- **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、32 ビット・モードでは 4 バイトの整数型、64 ビット・モードでは 8 バイトの整数型です。

構文は以下のとおりです。

```

▶▶ __attribute__((mode | __mode__ | (byte | word | pointer | __byte__ | __word__ | __pointer__)))▶▶

```

ここで、*mode* は幅の指示を含む型指定子です。

packed 変数属性

変数属性 **packed** を使用すると、構造体メンバーまたはビット・フィールド構造体メンバーが可能最小限度の位置合わせ、つまり、メンバー用に 1 バイト、フィールド用に 1 ビット (**aligned** 変数属性でより大きな値が指定されていない場合) を持つように指定することができます。



構文は以下のとおりです。

```

▶▶ __attribute__((__packed__))
    |         |
    |         |__packed__

```

weak 変数属性

  変数属性 **weak** および関数属性 **weak** は、同じ振る舞いと規則を持ちます。属性指定子を変数宣言へ適用する構文を使用すると、変数属性指定子を宣言子の前か後に表示することができます。以下のダイアグラムは、有効な 2 つの宣言構文の形式を示しています。

```

▶▶ __type_specifier__ __attribute__((__weak__)) __variable_name__
    |         |         |
    |         |         |__weak__

```

上記の構文は、**weak** 関数を宣言および定義するときと同じ構文です。**weak** 変数を宣言する他の有効な構文は、**weak** 関数宣言の構文と同じですが、構文定義は異なります。

```

▶▶ __type_specifier__ __variable_name__ __attribute__((__weak__))
    |         |         |
    |         |         |__weak__

```

__align 指定子

__align キーワードを使用すると、データ構造の明示的位置合わせを指定することができます。キーワードは、集合体型の定義または第 1 レベル変数の宣言において使用するための直交言語拡張です。指定されたバイト境界は、メンバーの位置合わせではなく、集合体の位置合わせに全体として影響します。**__align** 指定子は、別の集合体定義内でネストされた集合体定義に適用することができますが、集合体またはクラスの個々のエレメントには適用できません。位置合わせ指定は、パラメータおよび自動変数に対しては無視されます。

宣言の形式は次のいずれかです。

```

▶▶ __declarator__ __align__ (__int_constant__) __identifier__ ;

```

構造体または共用体の構文:

```

▶▶ __align__ (__int_constant__) __struct_or_union_specifier__ {__struct_declaration_list__};
    |         |
    |         |__tag__

```

ここで、

int_constant

バイト位置合わせ境界を示す正整数値です。正しい値は、1 または 2 の正の累乗値です。

struct_or_union_specifier

構造体または共用体の指定子です。

struct_declaration_list

構造体宣言リストです。

tag

構造体または共用体の ID です。

制約および制限事項

`__align` 指定子は、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使用することができません。

すべての位置合わせがオブジェクト・ファイル内で表示可能であるとは限りません。

`__align` 指定子を次の項目に適用することはできません。

- 集合体定義内の個々のエレメント。
- 配列の個々のエレメント。
- 不完全型の変数。
- 宣言されたが、定義されていない集合体。
- 型定義、関数、または列挙型などの他のタイプの宣言または定義。

仮定義

C 仮定義 は、ストレージ・クラス指定子も初期化指定子も持たない任意の外部データ宣言です。仮定義は、変換単位の最後に到達しても ID の定義が初期化指定子とともに表示されない場合は、完全定義になります。この場合、コンパイラは、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。

次のステートメントは、標準定義と仮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++ C++ は、仮定義の概念をサポートしていません。つまり、ストレージ・クラス指定子のない外部データ宣言は常に 1 つの定義と見なされます。

関連参照

- 35 ページの『宣言概要』
- 40 ページの『ストレージ・クラス指定子』

オブジェクト

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 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。

オブジェクトのデータ型によって、そのオブジェクトの初期ストレージ割り振り、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。

C および C++ の両方には、組み込みデータ型すなわち基本 データ型と、ユーザー定義のデータ型があります。標準データ型には、符号付き整数および符号なし整数、浮動小数点数、および文字が含まれます。ユーザー定義データ型には、列挙、構造体、共用体が含まれます。すべての C++ クラスは、ユーザー定義の型として見なされます。

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。

関連参照

- 109 ページの『左辺値と右辺値』
- 265 ページの『第 12 章 クラス』


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。宣言で使用するストレージ・クラス指定子によって、次のことが決まります。

- オブジェクトに、内部結合または外部結合があるか、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は、静的 (ストレージは、プログラムの実行中は保持される) または自動 (オブジェクトが定義されているブロックが実行される間のみ、ストレージは保持される) のいずれかです。

変数の場合、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、それが宣言される場所がブロック・ステートメントまたは関数本文の内側か外側かによって異なります。これらのデフォルトでは十分でない場合、明示的なストレージ・クラス **auto**、**static**、**extern**、または **register** を指定することができます。C++ では、追加のオプションを使うことによって、クラス・データ・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、そのメンバーにストレージ・クラス **mutable** を指定し、それを変更可能にすることができます。

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。唯一のオプションは、**extern** および **static** です。**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。**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はしか呼び出しできないことを意味します。関数のデフォルトは、外部結合です。


 関数仮パラメーターに指定できる唯一のストレージ・クラスは、**register** です。この理由は、関数仮パラメーターが **auto** 変数と同じプロパティ（自動ストレージ期間、ブロック・スコープ、およびリンケージなし）を持つためです。

auto または **register** ストレージ・クラス指定子が指定された宣言は、自動ストレージになります。**static** ストレージ・クラス指定子が指定された宣言は、静的ストレージになります。

extern ストレージ・クラス指定子が指定されていないローカルな宣言のほとんどは、ストレージを割り振られます。ただし、関数宣言および型宣言は、ストレージを割り振られません。

ネーム・スペース・スコープ宣言またはグローバル・スコープ宣言で許可されるストレージ・クラス指定子は、**static** と **extern** のみです。C++ では、内部結合の指定に **static** は使用すべきではありません。代わりに、名前のないネーム・スペースを使用します。

C および C++ におけるストレージ・クラス指定子は、以下のとおりです。

- **auto**
- **extern**
-  **mutable**
- **register**
- **static**
- **typedef**

typedef は、機能よりも構文がストレージ・クラス指定子に類似しているという理由と **typedef** 宣言はストレージを割り振らないという理由で、ストレージ・クラス指定子に分類されます。

auto 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宣言できます。**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。

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 文を使用し、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。

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振られます。

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。

関連参照

- 204 ページの『ブロック・ステートメント』
- 219 ページの『goto 文』
- 170 ページの『関数宣言』

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と関数を宣言できます。 **extern** 変数、関数定義、または宣言は、現行ソース・ファイルの以降の部分によって、記述された変数または関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。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。

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。

ストレージ・クラス指定子がない宣言の前に、ストレージ・クラス指定子 **static** がある同じ関数に対して宣言をインクルードすると、宣言が非互換であるために、エラーになります。オリジナルの宣言に **extern** ストレージ・クラス指定子を含めることは有効であり、その関数は内部結合を持ちます。

インライン関数用の GNU C セマンティクスが要求され、ソース・コードがそれに応じてコンパイルされると、キーワード **extern** はキーワード **inline** と結合し、単一キーワードとして振る舞います。

関連参照

- 198 ページの **extern inline** キーワード

 以下の注釈は、C++ にのみ適用されます。

- 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。
- C++ では、**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。

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、グローバル・スコープ (C) またはネーム・スペース・スコープ (C++) 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。

- 定数式によって初期値が記述され、定義の一部として発生する。または、
- 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。

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。メモリは、**main** 関数が実行される前に **extern** オブジェクトに割り振られ、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、プログラム・テキスト内の宣言の場所によって決定されます。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。

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・テキスト内の宣言がどこに配置されたかによって決まります。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、すべて外部結合を持つ ID を宣言します。ストレージ・クラスを指定しない関数定義では、すべて外部結合を持つ関数を定義します。

C++ 名前なしネーム・スペース内のオブジェクトの場合、リンケージは外部の可能性がありますが、名前は固有のため、他の変換単位の状況から判断されて、名前は実際上、内部結合を持ちます。

関連参照

- 8 ページの『外部結合』
- 7 ページの『内部結合』
- 45 ページの『static ストレージ・クラス指定子』
- 4 ページの『クラス・スコープ』
- 241 ページの『第 10 章 ネーム・スペース』
- 197 ページの『インライン関数』

mutable ストレージ・クラス指定子

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。これは、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、変更可能にするために使用されます。**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。

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

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

この例では、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。

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、オブジェクトの値をマシン・レジスターに常駐させるようにコンパイラーに指示します。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・クラス指定子を使用することで、ループ制御変数などのオブジェクトが頻繁に使用されているので、アクセス時間を最小化してパフォーマンスを上げるようプログラマーが望んでいることを知らせます。

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。

初期化

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。

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振ります。

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。

C 制約事項

- **register** ストレージ・クラス指定子は、ブロック内で宣言された変数についてのみ使用が許されます。これを、グローバルなスコープ・データ宣言で使用することはできません。
- **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++ 制約事項

- **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 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
```

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、内部結合を持つオブジェクトまたは関数を定義することができます。つまり、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または関数を表します。さらに、**static** と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。つまり、これらのオブジェクトのメモリーは、プログラムの実行が開始したときに割り振られ、プログラムが終了すると解放されます。

オブジェクト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。オブジェクト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。一方、**static** ストレージ・クラス指定子は、ファイル・スコープにある場合に限り、関数宣言で使用することができます。

static ストレージ・クラス指定子は、以下の名前だけに適用できます。

- オブジェクト
- 関数
- クラス・メンバー
- 無名共用体

以下のものは、**static** として宣言できません。

- 型宣言

- ブロック内の関数宣言
- 関数仮パラメーター

▶ **C** キーワード **static** は、情報隠蔽を強化するための、C の主要なメカニズムです。C++ は、ネーム・スペース言語フィーチャーおよびクラスのアクセス制御を使用して、情報隠蔽を強化します。

C99 言語レベルで、**static** キーワードを関数に対する配列パラメーターの宣言で使用することができます。**static** キーワードは、関数へ渡された引き数が少なくとも指定サイズの配列を指すポインターであることを知らせます。このようにして、ポインターの引き数は絶対にヌルではないことがコンパイラーに伝えられます。

▶ **C++** 外部変数のスコープを制限するキーワード **static** は、ネーム・スペース・スコープ内のオブジェクトを宣言するためには使用しないようにしてください。

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** (または外部) 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。

▶ **C** C における詳しい説明は、次のとおりです。

- 変数がポインター型の場合、変数はヌル・ポインターに初期化されます。
- 変数が算術型を持つ場合、正または符号なしのゼロに初期化されます。
- 変数が集合体の場合、最初の名前付きメンバーはこれらの規則に従って再帰的に初期化されます。
- 変数が共用体の場合、最初の名前付きメンバーはこれらの規則に従って再帰的に初期化されます。

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。

再帰的関数は、呼び出されるごとに **auto** 変数の新規セットを入手します。ただし、関数が **static** 変数を持つ場合は、関数のすべての呼び出しで同じストレージ・ロケーションが使用されます。

▶ **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。

C++ では、非定数式を使用して **static** オブジェクトを初期化できますが、以下の例は使用しないようにしてください。

```
static int staticInt = 5;
int main()
{
    // . . .
}
```

C++ では、外部変数のスコープを制限するネーム・スペース言語フィーチャーが提供されます。

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またはファイルの宣言は、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または関数を表します。

例

例えば、静的変数 `x` が関数 `f()` で宣言されていると想定します。プログラムが `f()` のスコープを終了するとき、`x` は破棄されません。次の例は、このことを示しています。

```
#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;
}
```

次に、上記の例の出力を示します。

```
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。

typedef

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、ストレージを予約しません。 **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。

typedef 宣言の例

ストレージ・クラス指定子

次のステートメントは、LENGTH を **int** の同義語として宣言し、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++ のクラス) 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;
```

次の例では、yds の型は、「int を戻す、パラメーターが指定されていない関数へのポインター」です。

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

次の typedef では、トークン struct は型名の一部です。ex1 の型名は struct a、ex2 は struct b です。

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

型 ex1 は、型 struct a、および ptr1 によって指示されるオブジェクトの型と互換性があります。型 ex1 は、char、ex2、または struct b とは互換性がありません。

 このセクションでのここから先の説明は、C++ だけに適用されます。

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せん。これは、C の場合には当てはまりません。標準の C ヘッダーには、以下のものが含まれています。

```
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、リンクージ用の **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、それ自体はクラスの型名ではありません。したがって、Trees() が、そのクラスのコンストラクターになることはできません。

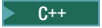
型指定子

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

- 単純型指定子
- 列挙指定子
- **const** および **volatile** 修飾子
-  クラス指定子
-  詳述型指定子

用語 **スカラー型** は、C においては、算術型またはポインター型の総称として使用されます。C++ では、スカラー型は、C スカラー型のすべての **cv-qualified** バージョン、および列挙型とポインター・メンバー間型のすべての **cv-qualified** バージョンが含まれます。

用語 **集合体型** は、C と C++ の両方において、配列型および構造体型のことをいいます。

 C++ では、型は、宣言の中で宣言する必要があります。式の中で宣言してはなりません。

型名

データ型、つまり **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、常に添え字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。添え字演算子、関数呼び出し演算子、および間接演算子のみを、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添え字演算子または関数呼び出し演算子よりも、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。

型指定子

```
int *[]      /* array of 5 pointers to int */
int (*)[5]   /* pointer to an array of 5 ints */
int (*)[*]   /* pointer to an variable length array of
              an unspecified number of ints */
int *()      /* function with no parameter specification
              returning a pointer to int */
int *(void)  /* function with no parameters returning an int */
int (*const [])(unsigned int, ...)
              /* array of an unspecified number of
              constant pointers to functions returning an int
              Each function takes one parameter of type unsigned int
              and an unspecified number of other parameters */
```

コンパイラーは、任意の関数指定機能を、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、関数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: the compiler turns foo into a function pointer */
```

C++ C++ では、キーワード **typename** および **class** (交換可能) は、型名を示します。

型属性

型属性は、キーワード **__attribute__** および付随する構文を使用して、構造体、共用体、列挙型、またはクラスの特別なプロパティを指定する宣言指定子です。型属性は C および C++ に対する直交拡張機能で、GNU C および C++ を使用して開発されたプログラムの移植を容易にするためにインプリメントされています。

型属性の構文の一般的な形式は次のとおりです。

```
▶▶ __attribute__((__attribute_name__))
```

型属性 aligned

型属性 **aligned** により、構造体、クラス、共用体、または列挙型の位置合わせを指定することができます。指定する位置合わせ係数の構文および考慮事項は、変数属性 **aligned** と同じです。変数属性 **aligned** と同様に、型属性 **aligned** は位置合わせを増加させるだけです。型属性 **packed** は、位置合わせを減少させるために使用します。

属性が、クラス、構造体、共用体、または列挙型トークンの直後、または右中括弧の直後に表示される場合、型 ID に適用されます。**typedef** 宣言で指定することもできます。以下のような変数宣言では、

```
class A { } a;
```

型属性の配置が混乱する可能性があります。

次の定義で、属性は A に適用されます。

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



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

次の定義で、属性は `a` に適用されます。

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

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

型属性 `packed`

構造体、クラス、共用体、または列挙型で `packed` 型属性を指定することは、必要最少量のメモリーがその型に使用されることを示します。型属性 `packed` の配置は、型属性 `packed` が `typedef` 宣言で黙って無視されることを除いては、型属性 `aligned` と同じです。

型属性 `transparent_union`

  共用体定義または共用体 `typedef` に適用される

`transparent_union` 属性は、透過共用体として共用体を使用できることを示します。透過共用体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きは常に、その透過共用体は、明示的キャストのない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き数は、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。この言語フィーチャーは、C89、C99、および Standard C++ および C++98 に対する直交拡張で、もともとは GNU C で開発されたプログラムの移植を容易にするためにインプリメントされています。

この型属性は、共用体または `typedef` 定義の右中括弧の後に指定しなければなりません。

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

typedef union {
    int *iptr;
    union u2_t *u2ptr;
} status_ptr_t __attribute__((__transparent_union__));
```

型属性 `transparent_union` は、タグ名を持つ無名共用体に適用することができます。

型属性 `transparent_union` が、ネストされた共用体の外部共用体に適用される場合、内部共用体 (その最大メンバー) のサイズを使用して、外部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 u_t {
    union u2_t {
        char a;
        short b;
        char c;
        char d;
    };
    int a;
} __attribute__((transparent_union));
```

属性 `transparent_union` は無視されます。これは、そのものも共用体である、共用体 `u_t` の先頭メンバーが 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` の他のメンバーの型は `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評価されます。

制約事項

共用体は、完全な共用体型でなければなりません。

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、多くても 4 バイトで表現可能でなければなりません。この透過共用体の場合、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見なされます。

共用体の先頭メンバーが浮動小数点型 (`float`、`double`、`float _Complex`、または `double _Complex`) であることはできません。ただし、`float _Complex` 型と `double _Complex` 型は、それらが先頭メンバーでないかぎり、透過共用体のメンバーであることができます。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約事項も適用されます。

例

この例は、属性 `transparent_union` を関数仮パラメーター宣言にどのように適用できるかを示しています。

```
void foo( union u_t {
    int a;
    short b;
    char c;
} __attribute__((transparent_union)) uu
) {
    printf("uu.b is %d\n",uu.b);
}

int main() {
    short s = 99;
    foo(s);
    return 0;
}
```

この例は、Complex 型がどのように透過共用体のメンバーであることができるかを示しています。

```
union u_t {
    int i[2];        // This member must has a machine representation of 8 bytes.
    float _Complex cf;
} __attribute__((__transparent_union__)) U;

void foo(union u_t uu) {
    printf("uu.cf is %f\n",uu.cf);
}

int main() {
    float _Complex my_cf = 5.0f + 1.0f * __I;
    foo(my_cf);
    return 0;
}
```

互換型

C 互換型の概念は、変更することなく 2 つの型を一緒に使用できるという考え方 (代入式のように)、変更することなく一方から他方へ置換できるという考え方、およびそれらを複合型へ結び付けるという考え方を結合したものです。複合型は、2 つの互換型を結合した結果生まれるものです。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、関数プロトタイプ、および型修飾型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。typedef 定義での名前は単なる型の同義語で、そのため typedef の名前が同じもの、ゆえに互換タイプを示す可能性があります。特定のプロパティを持つポインター、関数、および配列もまた、互換タイプである可能性があります。

同一型

算術型の場合には、さまざまな結合の型指定子があることによって、異なる型が表示されたり、表示されなかったりします。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。

型修飾子があることによって、型が変更します。つまり、**const int** は **int** と同じ型でないため、この 2 つの型には互換性はありません。

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。

別々にコンパイルされたソース・ファイル間の互換性

構造体、共用体、または列挙型の定義によって、新規の型が生まれます。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・ファイルに定義されると、理論上、各ファイルは同じ名前を持つ型のオブジェクトに異なる定義を含めることができます。2 つの宣言は互換性がある必要があり、互換性がなければプログラムのランタイム振る舞いは定義されません。したがって、互換性規則は、同じソース・

ファイル内での互換性規則よりも制限が厳しくて具体的です。別々にコンパイルされたファイルで定義された構造体、共用体、および列挙型の場合、その複合型は現行ソース・ファイル内の型です。

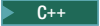
別々のソース・ファイルで宣言された 2 つの構造体、共用体、または列挙型の間の互換性要件は、以下のとおりです。

- 一方をタグを使用して宣言した場合、もう一方も同じタグで宣言する必要があります。
- 両方が完了型の場合、それらのメンバーは、数が正確に一致し、互換タイプを使用して宣言し、一致する名前を持つ必要があります。

列挙型の場合はまた、一致するメンバーが同じ値を持つ必要があります。



構造体および共用体の場合、型の互換性を保つために、以下の追加の要件を満たさなければなりません。

- 一致するメンバーは、同じ順番で宣言する必要があります (構造体のみに適用)。
- 一致するビット・フィールドは、同じ幅を持つ必要があります。

 同一型であることを型互換性の条件とする考え方とは異なる考え方は、C++ では存在しません。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。

単純型指定子

単純型指定子 は、(以前に宣言された) ユーザー定義の型または基本型 のいずれかを指定します。基本型は、言語にビルドされる型です。以下に、基本型のカテゴリーの概要を示します。

- 算術型
 - 整数型
 -  **bool**
 - **char**
 - **wchar_t**
 - 符号付き整数型
 - **signed char**
 - **short int**
 - **int**
 - **long int**
 - **long long int**
 - 符号なし整数型
 -  **_Bool**
 - **unsigned char**
 - **unsigned short int**
 - **unsigned int**
 - **unsigned long int**
 - **unsigned long long int**
 - 浮動小数点型
 - **float**
 - **double**

- long double
- void

浮動小数点型は、複素数型 `float _Complex`、`double _Complex`、および `long double _Complex` と区別する必要がある場合、**実浮動小数点型** として参照されます。実浮動小数点型および複素数型は、総称して、**浮動小数点型** と呼ばれます。

bool 変数

bool 変数は、整数値 0 または 1、もしくは C++ リテラル `true` および `false` を保持する場合に使用できます。これらのリテラルは、算術値が必要なときにいつでも整数 0 および 1 へ暗黙的にプロモートされます。bool 変数を宣言する型指定子は、C++ では **bool** です。C で bool 変数を宣言するには、ヘッダー・ファイル `<stdbool.h>` で定義される `bool` マクロを使用します。bool 変数は、指定子 `signed`、`unsigned`、`short`、または `long` によってさらに修飾できません。

C bool 型は、符号がなく、標準符号なし整数型のカテゴリーにおいて最低にランク付けされます。単純な割り当てでは、左方オペランドが bool 型であれば、右方オペランドは算術型またはポインタのいずれかである必要があります。bool 型として宣言されたオブジェクトは、1 バイトのストレージ・スペースを使用します。これは値 0 または 1 を保持するのに十分な大きさです。

C では、bool 型をビット・フィールド型として使用することができます。bool 型のゼロ以外の幅のビット・フィールドが値 0 または 1 を保持する場合、ビット・フィールドの値は、それぞれ 0 または 1 と同等比較されます。

C++ 型 **bool** の変数は 2 つの値のどちらか、つまり `true` または `false` を保持できます。型 **bool** の右辺値は、整数型へプロモートできます。`false` の **bool** 右辺値は、値 0 へプロモートでき、`true` の **bool** 右辺値は、値 1 へプロモートされます。

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 bool 定数 `true` または `false` のいずれになります。

型指定子 **bool** およびリテラル `true` および `false` を使用して、bool 論理テストを行います。bool 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` を返します。そうでなければ、`f()` は `false` を返します。

char および wchar_t 型指定子

char 指定子の構文は、次のとおりです。




char 指定子は、整数型です。

char は、基本文字セットからの文字を表すのに十分なストレージを持っています。**char** に割り振られるストレージの量は、インプリメンテーションによって異なります。

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、型 **char** の変数を初期化します。

単一バイトを占有する数値変数を宣言するには、**signed char** または **unsigned char** を使用します。

 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 型指定子の例

次の例では、ID `end_of_string` を、初期値 `¥0` (ヌル文字) を持つ、型 **char** の定数オブジェクトとして定義しています。

```
const char end_of_string = '¥0';
```

次の例では、**unsigned char** 変数である `switches` を、初期値 `3` を持つものとして定義します。

```
unsigned char switches = 3;
```

次の例では、`string_pointer` を文字を指すポインターとして定義します。

```
char *string_pointer;
```

次の例では、`name` を文字を指すポインターとして定義します。初期化の後で、`name` は、文字ストリング "Johnny" の最初の文字を指します。

```
char *name = "Johnny";
```

次の例では、文字を指す 1 次元配列のポインターを定義します。配列には、3 つのエレメントがあります。最初は、これらのエレメントは、ストリング "Venus" を指すポインター、"Jupiter" を指すポインター、および "Saturn" を指すポインターです。

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

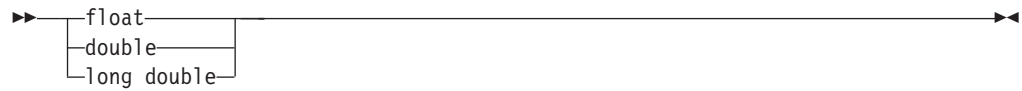
wchar_t 型指定子: **wchar_t** 型指定子は、ワイド文字リテラルを表すのに十分なストレージを持つ整数型です。(ワイド文字リテラルは、`L'x'` のように、文字 `L` を接頭部として付けた文字リテラルです。)

浮動小数点変数

浮動小数点変数には、次の 3 つの型があります。

- **float**
- **double**
- **long double**

浮動小数点型のデータ・オブジェクトを宣言するには、次の**浮動**指定子を使用します。



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評価される変数または式を使用して行います。変数のストレージ・クラスによって、変数を初期化する方法が決まります。

浮動小数点データ型の例

次の例では、ID `pi` を **double** 型のオブジェクトとして定義します。

```
double pi;
```

次の例では、**float** 変数である `real_number` を初期値 100.55 で定義します。

```
static float real_number = 100.55f;
```

注: **f** サフィックスを浮動小数点リテラルに追加しない場合は、その数は、型 **double** になります。型 **float** のオブジェクトを、型 **double** のオブジェクトで初期化する場合、コンパイラは暗黙的に、型 **double** のオブジェクトを型 **float** のオブジェクトに変換します。

次の例では、**float** 変数である `float_var` を初期値 0.0143 で定義します。

```
float float_var = 1.43e-2f;
```

次の例では、**long double** 変数である `maximum` を宣言します。

```
extern long double maximum;
```

次の例では、配列 `table` を型 **double** の 20 個の要素で定義します。

```
double table[20];
```

関連参照

- 26 ページの『浮動小数点リテラル』
- 153 ページの『代入式』


整変数

整変数は、次のカテゴリーに分かれます。

- 整数型

-  **bool**
- **char**
- **wchar_t**
- 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**


注:  整数型の **long long int** および **unsigned long long int** は、Standard C++ および C++98 の直交拡張機能です。

ビット・フィールドのデフォルトの整数型は、**unsigned** です。

整数データに割り振られるストレージの量は、インプリメンテーションによって異なります。

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評価される式を使用して行うことができます。変数のストレージ・クラスは、変数を初期化する方法を決めます。

 多重定義関数および多重定義演算子の引き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** 引き数に対して、**int** 引き数は、多重定義できません。

整数データ型の例

次の例では、**short int** 変数である `flag` を定義します。

```
short int flag;
```

次の例では、**int** 変数である `result` を定義します。

```
int result;
```

次の例では、**unsigned long int** 変数である `ss_number` を、初期値 438888834 を持つように定義します。

```
unsigned long ss_number = 438888834ul;
```

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。

関数が値を戻さないときは、関数の定義および宣言で、型指定子として **void** を使用する必要があります。引き数を取らない関数の引き数リストは、**void** 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

void 型の例

次の例では、関数 `find_max` は、型 **void** を持つものとして宣言されています。

注:  `find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));` の行の `sizeof` 演算子の使用は、配列の要素の数を決める標準的な方法です。

```
/**
** Example of void type
**/
#include <stdio.h>

/* declaration of function find_max */
extern void find_max(int x[ ], int j);

int main(void)
{
    static int numbers[ ] = { 99, 54, -102, 89};


    find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));

    return(0);
}

void find_max(int x[ ], int j)
{ /* begin definition of function find_max */
    int i, temp = x[0];

    for (i = 1; i < j; i++)
    {
        if (x[i] > temp)
            temp = x[i];
    }
    printf("max number = %d\n", temp);
} /* end definition of function find_max */
```

複合型

 C++ では、複合型の概念および構成方法を正式に定義しています。多くの複合型は、C から生まれました。

複合型を使用するのは、以下のいずれかを作成するときです。

- 指定型のオブジェクト配列
- 指定型のパラメーターを持ち、指定型の `void` またはオブジェクトを戻す、任意の関数
- 指定型の **void**、オブジェクト、または関数を指すポインター
- 指定型のオブジェクトまたは関数への参照
- クラス
- 共用体
- 列挙型
- 非静的クラス・メンバーを指すポインター

構造体

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。配列の要素とは異なり、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。

C 構造体のメンバーは、可変的に変更される型を除き、任意のオブジェクト型を持ちます。最後のメンバーを除くすべてのメンバーは、完了型である必要があります。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要素は、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。

C++ C++ では、構造体メンバーは完了型である必要があります。

論理的に関連したオブジェクトをグループ化するには、構造体を使用します。例えば、あるアドレスの複数のコンポーネントにストレージを割り振るには、次の変数を定義します。

```
int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;
```

複数のアドレスにストレージを割り振るには、構造体データ型と、構造体データ型を保持するのに必要な数のみ変数を定義することによって、各アドレスのコンポーネントをグループ化します。

C++ C++ では、構造体は、そのメンバーおよび継承が、デフォルトにより `public` であるということを除けば、クラスと同じです。

次の例では、`int street_no;` から `char *postal_code;` までの行は、構造体タグ `address` を宣言しています。

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
struct address perm_address;
struct address temp_address;
struct address *p_perm_address = &perm_address;
```

変数 `perm_address` と `temp_address` は、構造体データ型 `address` のインスタンスです。これらの両方には、`address` の宣言に記述されたメンバーが含まれています。ポインター `p_perm_address` は、`address` という構造体を指し、`perm_address` を指すように初期化されます。

ドット演算子 (`.`) 付きの構造変数の名前か、矢印演算子 (`->`) 付きのポインターとメンバー名を指定することによって、構造体の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、文字列 "Ontario" を指すポインタを、構造体 perm_address にあるポインタ prov に代入します。

構造体への参照はすべて、完全修飾されている必要があります。例では、`prov` のみでは、4 番目のフィールドを参照することはできません。このフィールドは、`perm address.prov` によって参照する必要があります。

同一メンバーがあるが別の名前が付けられている構造体には、互換性はなく、互いに割り当てることはできません。

構造体は、ストレージを節約するためのものではありません。バイト・マッピングの直接制御を行う必要がある場合は、ポインターを使用します。

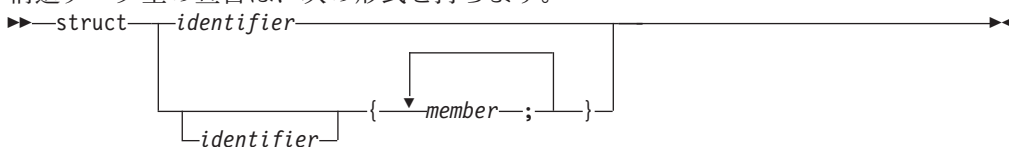
互換構造体

C 構造体の定義ごとに、同じソース・ファイルにある他の構造体型とは異なる、または互換性のない新規の構造体型を作成します。ただし、すでに定義された構造体型への参照である型指定子は同じ型です。構造体タグが、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型 `j` および `k` だけが同じ型になります。

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

構造体の宣言および定義: 構造体型の定義 では、構造体の一部になっているメンバーを記述します。構造体型の定義には、**struct** 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ています。

構造データ型の宣言は、次の形式を持ちます。



ID (タグ) が後に続くキーワード **struct** は、データ型に名前を付けます。タグの名前を与えない場合は、そのタグを参照するすべての変数定義を、データ型の宣言内に入れる必要があります。

構造体宣言 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、構造体定義と同じ形式です。構造体定義は、構造体データ型の宣言と同じ形式を持ちますが、セミコロンで終了します。

構造体メンバーの定義

メンバーのリストで、構造体に保管できる値の説明と構造体データ型が提供されます。C では、構造体メンバーは、「関数からの戻り T」(型によっては T)、不完全型、可変的に変更される型、および `void` を除く、任意の型をとります。不完全型

は構造体メンバーとして許可されないため、構造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。

構造体メンバーの定義は、変数宣言の形式になっています。構造体メンバーの名前は、単一構造体内で固有でなければなりません、同じメンバー名は、同じスコープ内で定義された別の構造体型で使用でき、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。ビット・フィールドを表さないメンバーは、任意のデータ型にすることができ、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。ただし、型修飾子のないビット・フィールドは、構造体メンバーとして宣言することができます。ビット・フィールドは、名前がない場合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。

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、ホールまたは埋め込みを構造体レイアウト内の連続メンバー間に置くことができます。

フレキシブルな配列メンバー

複数の名前付きメンバーを持つ構造体の最後のエレメントは、不完全な配列型となることができます。これを *柔軟な配列メンバー* と呼びます。

柔軟な配列メンバー は、複数の名前付きメンバーを持つ構造のエレメントです。柔軟な配列メンバーは、構造体などの最後のエレメントで、不完全な配列型である必要があります。

▶▶—array_identifier[]————▶▶

例えば、b は struct foo の柔軟な配列メンバーです。

```
struct foo{
    int a;
    char b[];
};
```

struct foo のサイズは 4 です。struct foo は、別の struct または配列のメンバーにすることはできません。

配列添え字がゼロの場合、配列メンバーは *ゼロ・エクステント配列* と見なされます。

▶▶—array_identifier[0]————▶▶

前記の例で b がゼロ・エクステント配列として宣言される場合、struct foo のサイズは 4 のままですが、次の例のように、struct foo は別の struct または配列のメンバーとすることができます。

```
struct bar{
    struct foo zarray;
};
```

普通、柔軟な配列メンバーは無視されます。ただし、これは以下の 2 つの場合に認識されます。

- 未指定の長さの配列が、柔軟な配列メンバーを置換するとしましょう。元の構造体の柔軟な配列メンバーが認識されるのは、元の構造体のサイズが、置換配列を使った構造体の最後のエレメントのオフセットと等しい場合です。


- 柔軟な配列メンバーを表すために、ドットまたは矢印の演算子を使用されている場合。

2 番目の場合、アクセスしているオブジェクトよりも構造体を大きくすることのない最長の配列に、メンバーが置き換えられたかのように振る舞います。配列のオフセットは、柔軟な配列メンバーのオフセットと同じまです。置換配列にエレメントがない場合、振る舞いはエレメントが 1 つある場合のように見えますが、そのエレメントにはアクセスできず、それを通過するポインターも作成できません。以下の例で、`d` は構造体 `struct s` の柔軟な配列メンバーです。

```
// Assuming the same alignment for all array members,
```

```
struct s { int n; double d[]; };
struct ss { int n; double d[1]; };
```


式 `offsetof(struct s, d)` および `offsetof(struct ss, d)` は、同じ値 `sizeof(struct s)` を持ちます。

構造変数の定義:  構造体変数の定義には、オプションのストレージ・クラス・キーワード、**struct** キーワード、構造体タグ、宣言子、およびオプションの ID が含まれています。構造体タグは、構造体変数のデータ型を指示します。

 キーワード **struct** は、C++ ではオプションです。

任意のストレージ・クラスを持つ構造体を宣言できます。 **register** ストレージ・クラス指定子で宣言された構造体は、自動構造体として扱われます。

構造体の初期化: 構造体の初期化指定子は、括弧で囲まれ、コンマで区切られた値のリストです。初期化指定子には、等号 (=) が前に付いています。指定がなければ、構造体メンバーのメモリーは宣言された順に割り振られます。メモリー・アドレスは昇順で割り当てられ、最初のコンポーネントは構造体名自身の開始アドレスで始まります。構造体のすべてのメンバーを初期化する必要はありません。 `static` ストレージを持つ構造体のデフォルトの初期化指定子は各コンポーネントの再帰的デフォルトです。 `automatic` ストレージを持つ構造体には何も含まれません。

 以下の小節は、C にのみ適用されます。

構造体の名前付きメンバーは、任意の順番で初期化できます。共用体の名前付きメンバーは、最初のメンバーでなくても初期化できます。指定機能 は、初期化する構造体または共用体のメンバーを識別します。構造体または共用体のメンバー用の指定機能は、ドットおよびその ID (*.fieldname*) で構成されます。指定機能リストは、任意の集合体型用の 1 つまたは複数の指定機能の組み合わせです。指定 は、等号 (=) が後に付いた指定機能リストです。

指定機能は、初期化の最初では構造体自身である現行オブジェクトの最初のサブオブジェクトを識別します。最初のサブオブジェクトを初期化した後は、次のサブオブジェクトが現行オブジェクトになり、その最初のサブオブジェクトが初期化されます。つまり、初期化は前方向の順番で進み、直前のサブオブジェクトの初期化はオーバーライドされます。

構造体または任意の集合体型の自動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。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るのは、C99 言語フィーチャーです。

構造体の以下の宣言は、明示的な初期化を提供することによってどのサブオブジェクトが初期化されるかについてのあいまいさを除去する指定機能を含む定義です。以下の宣言では、2 つの要素構造体を使用して配列を定義します。以下の引用では、`[0].a` および `[1].a[0]` が指定機能リストです。

```
struct { int a[5], b; } game[] =
    { [0].a = { 1 }, [1].a[0] = 2 };

/* game[0].a[0] is 1, game[1].a[0] is 2, and all other elements are zero. */
```

宣言構文は、初期化指定子リストを示すために中括弧を使用しますが、大括弧形式として参照されます。宣言の完全な大括弧形式は、簡潔な形式よりも誤解されにくくなります。以下の定義は、上記と同じことを実行し、適正で短いですが、大括弧が相互に一致していないので、誤解を招く可能性があります。2 つの `struct game` オブジェクトのいずれの `b` 構造体メンバーも 2 に初期化されません。

```
struct { int a[5], b; } game[] =
    { { 1 }, 2 };

/* game[0].a[0] is 1, game[1].a[0] is 2, and all other elements are zero. */
```

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。

例

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
<code>perm_address.street_no</code>	3
<code>perm_address.street_name</code>	ストリング "Savona Dr." のアドレス
<code>perm_address.city</code>	ストリング "Dundas" のアドレス
<code>perm_address.prov</code>	ストリング "Ontario" のアドレス
<code>perm_address.postal_code</code>	ストリング "L4B 2A1" のアドレス

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	ストレージ・クラスによって異なる値

注: temp_address.postal_code のような初期化されていない構造体メンバーの初期値は、メンバー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。

同じステートメントでの構造体型および変数の宣言:  1 つのステートメントで、構造体型および構造変数を定義するには、型定義の後に、宣言子とオプションの初期化指定子を置きます。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;
```

この例では、構造データ型に名前を付けないので、perm_address と temp_address は、このデータ型が指定された唯一の構造変数です。struct の後に ID を入れることによって、プログラムの中に、後で、このデータ型の変数定義を追加することができます。

構造型 (またはタグ) は **volatile** 修飾子を持つことはできませんが、メンバーまたは構造変数は、**volatile** 修飾子を持つように定義することができます。

次に例を示します。

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

この例では、構造体 file1 と file2 を修飾し、構造体メンバー subfile.code を **volatile** として修飾します。

構造体位置合わせ: 構造体は、align コンパイラ・オプションの設定に従って位置合わせされます。この設定では、構造体および共用体のストレージをレイアウトするときにコンパイラが使用する位置合わせ規則を指定します。サブオプションごとに、位置合わせに与える影響は異なります。構造体のマッピングは、構造体定義の左中括弧の個所において有効な位置合わせモードに基づきます。構造体メンバーは、型によって位置合わせされます。

位置合わせコンパイラー・オプションによって設定される位置合わせモードの他に、位置合わせモードを設定するために**#pragma・オプション位置合わせディレクティブ**が使用されます。構造体の左中括弧の個所において有効になっている位置合わせオプションによって、構造体をマップする方法が決定されるため、構造体内部でネストされた**#pragma・オプション位置合わせ**は、**pragma**の後に続く左中括弧を持つ構造体の定義にのみ影響を与えます。

異なる位置合わせを持つ構造体および共用体をネストできます。各構造体は、適用可能な位置合わせを使用してレイアウトされます。ネストされた構造体の開始位置は、それがネストされた構造体において有効になっている位置合わせ規則によって決定されます。

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は、型互換性はなく、互いに割り当ててはできません。

関連参照

- 位置合わせに影響する **align** コンパイラー・オプションおよび **#pragma** の詳しい説明については、「**XL C/C++ コンパイラー・リファレンス: データ・マッピングとストレージ**」を参照してください。

構造体でのビット・フィールドの宣言および使用: C および C++ では、整数メンバーを、コンパイラーが普通許可するサイズより小さなメモリー・スペースへ保管することができます。スペースを節約するこれらの構造体メンバーは、**ビット・フィールド** と呼ばれ、それらのビット単位の幅を明示的に宣言することができます。ビット・フィールドは、データ構造体を固定ハードウェア表記に強制的に調和させる必要がある移植不能なプログラムで使用されます。

ビット・フィールドを宣言する構文は、以下のとおりです。

```

▶▶ type_specifier [declarator] : constant_expression ; ▶▶

```

ビット・フィールド宣言には、型指定子と、その後に続く、オプションの宣言子、コロン、フィールド幅をビット単位で示す定数整数式、およびセミコロンが含まれます。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 C99 規格では、修飾および非修飾の **_Bool**、**signed int**、および **unsigned int** を組み込むために、ビット・フィールド用に許可されるデータ型を必要とします。さらに、このインプリメンテーションは次の型をサポートします。

- int**
- short**、**signed short**、**unsigned short**
- char**、**signed char**、**unsigned char**
- long**、**signed long**、**unsigned long**
- long long**、**signed long long**、**unsigned long long**

すべてのインプリメンテーションにおいて、ビット・フィールドのデフォルトの整数型は、**unsigned** です。

▶ **C++** C++ では、ビット・フィールドの許容型リストが拡張され、任意の整数型または列挙型が含まれます。

どちらの言語でも、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。

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。幅がゼロのビット・フィールドの場合は、次のフィールドが次のコンテナ境界に位置合わせされることになり、そこでは、コンテナは、ビット・フィールドの基礎となる型と同じサイズになります。

ビット・フィールドは、align コンパイラ・オプションも必要とします。align サブオプションごとに、異なるセットの位置合わせプロパティをビット・フィールドに与えます。位置合わせに影響する align コンパイラ・オプションおよび #pragma の詳しい説明については、「*XL C/C++ コンパイラ・リファレンス*」を参照してください。

▶ **AIX** ▶ **Linux** ▶ **C** 最大のビット・フィールド長は、64 ビットです。移植性を持たせるために、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。

▶ **AIX** ▶ **Linux** ▶ **C++** C++ Standard では、宣言でビット・フィールドの長さは制限されません。ただし、基礎となる型のサイズを超えるビットは埋め込みのように動作します。

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の保持
- ビット・フィールドへの参照の保持

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・フィールドの位置合わせ

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。

次の例は、埋め込みを示しています。**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
<code>light</code>	1 ビット
<code>toaster</code>	1 ビット
(埋め込み — 30 ビット)	次の <code>int</code> 境界まで
<code>count</code>	<code>int</code> のサイズ (4 バイト)
<code>ac</code>	4 ビット
(名前なしフィールド)	4 ビット
<code>clock</code>	1 ビット
(埋め込み — 23 ビット)	次の <code>int</code> 境界まで (名前なしフィールド)
<code>flag</code>	1 ビット
(埋め込み — 31 ビット)	次の <code>int</code> 境界まで

構造体フィールドに対するすべての参照は、完全修飾されている必要があります。例えば、`toaster` によって、2 番目のフィールドを参照することはできません。このフィールドを参照するには、`kitchen.toaster` を使用しなければなりません。

次の式では、`light` フィールドを 1 にセットします。

```
kitchen.light = 1;
```

ビット・フィールドに範囲外の値を割り当てると、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。次の式では、`kitchen` 構造体の `toaster` フィールドに 0 を割り当てます。これは、最下位桁のビットのみが `toaster` フィールドに割り当てられるからです。

```
kitchen.toaster = 2;
```

関連参照

- ・「*XL C/C++ プログラミング・ガイド*」の『集合体内のデータの位置合わせ』
- ・「*XL C/C++ コンパイラー・リファレンス*」の『`-qalign`』

構造体を使用したプログラムの例: 次のプログラムでは、リンク・リストの整数の合計を検出します。

```

/**
 ** Example program illustrating structures using linked lists
 **/

#include <stdio.h>

struct record {
    int number;
    struct record *next_num;
};

```

```

int main(void)
{
    struct record name1, name2, name3;
    struct record *recd_pointer = &name1;
    int sum = 0;

    name1.number = 144;
    name2.number = 203;
    name3.number = 488;

    name1.next_num = &name2;
    name2.next_num = &name3;
    name3.next_num = NULL;

    while (recd_pointer != NULL)
    {
        sum += recd_pointer->number;
        recd_pointer = recd_pointer->next_num;
    }
    printf("Sum = %d\n", sum);

    return(0);
}

```

構造体型 `record` には、整数 `number` と `next_num` の 2 つのメンバーが含まれます。これは、`record` 型の構造変数を指すポインターです。

`record` 型変数である `name1`、`name2`、および `name3` に、次の値が割り当てられます。

メンバー名	値
<code>name1.number</code>	144
<code>name1.next_num</code>	<code>name2</code> のアドレス
<code>name2.number</code>	203
<code>name2.next_num</code>	<code>name3</code> のアドレス
<code>name3.number</code>	488
<code>name3.next_num</code>	NULL (リンク・リストの終了を指示する)

変数 `recd_pointer` は、`record` 型の構造体を指すポインターです。この変数を `name1` のアドレス (リンク・リストの先頭) に初期化します。

while ループによって、`recd_pointer` が NULL になるまでリンク・リストがスキャンされます。ステートメント、

```
recd_pointer = recd_pointer->next_num;
```

では、ポインターをリスト内の次のオブジェクトに進めます。

関連参照

- 85 ページの『不完全型』

共用体

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ば、構造体に類似したオブジェクトです。共用体は、一度に、そのメンバーのうちの 1 つの値しか含むことができません。 `static` ストレージを持つ共用体の

デフォルトの初期化指定子は最初のコンポーネントのデフォルトであり、`automatic` ストレージを持つ共用体には何も含まれません。

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に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)。このため、可変的に変更される型は共用体メンバーとして宣言できません。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。

C 最初のメンバーだけでなく共用体の任意のメンバーを、指定機能を使用して初期化できます。共用体の指定された初期化指定子は、構造体と同じ構文を持ちます。以下の例において、指定機能は `.any_member`、初期化指定子は `{.any_member = 13}` です。

```
union { /* ... */ } caw = { .any_member = 13 };
```

互換共用体

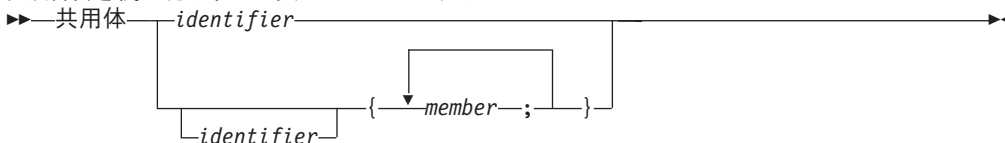
C 共用体の定義ごとに、同じソース・ファイルにある他の共用体型とは異なるか、または互換性のない新規の構造体型を作成します。ただし、すでに定義された共用体型への参照である型指定子は同じ型です。共用体タグが、参照と定義を関連付け、型名として効果的に機能します。

C++ C++ では、共用体は、クラス型の限定された形式です。それには、アクセス指定子 (`public`、`protected`、`private`)、メンバー・データ、メンバー関数 (コンストラクターおよびデストラクターを含む) を含めることができます。それには、仮想メンバー関数または静的データ・メンバーを含めることはできません。共用体のメンバーのデフォルトのアクセスは `public` です。共用体は、基底クラスとして使用できず、また基底クラスから派生できません。

C++ は、共用体メンバーの許容データ型に制限を追加します。C++ の共用体メンバーは、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。

共用体の宣言: 共用体型の定義 には、**`union`** キーワードと、その後続く、オプションの ID (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。

共用体定義の形式は、次のとおりです。



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、共用体定義と同じ形式です。

identifier は、メンバー・リストによって指定された共用体に与えられるタグです。タグを指定すると、以降の共用体の宣言 (同じスコープ内) では、タグを宣言し、メンバー・リストを省略することができます。タグを指定しない場合は、その共用体を参照する変数定義をすべて、データ型を定義するステートメント内に入れる必要があります。

メンバーのリストには、データ型と、共用体に保管できるオブジェクトの記述を指定します。

共用体メンバー定義の形式は、変数宣言の形式と同じです。

共用体のメンバーは、構造体のメンバーと同じ方法で参照することができます。

次に例を示します。

```
union {
    char birthday[9];
    int age;
    float weight;
} people;

people.birthday[0] = '\n';
```

'\n'を、共用体 `people` のメンバーである、文字配列 `birthday` の 1 番目のエレメントに代入します。

共用体は、一度に、そのメンバーのうちの 1 つのみしか表すことができません。この例では、共用体 `people` には、`age`、`birthday`、または `weight` のいずれか 1 つを含むことができますが、2 つ以上を含むことはできません。次の例では、`printf` ステートメントの結果は正しくありません。これは、`people.age` が、1 行目で `people.birthday` に代入された値を置換するからです。

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

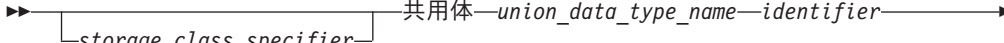
union {
    char birthday[9];
    int age;
    float weight;
} people;

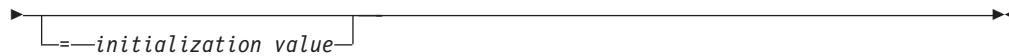
int main(void) {
    strcpy(people.birthday, "03/06/56");
    printf("%s\n", people.birthday);
    people.age = 38;
    printf("%s\n", people.birthday);
}
```

上記の例の出力は、次の出力に似ています。

```
03/06/56
&
```

共用体変数の定義:  共用体変数定義の形式は、次のとおりです。

→  共用体 `union_data_type_name` `identifier`



共用体データ型が指定された共用体を定義する前に、共用体データ型を宣言する必要があります。

共用体の名前付きメンバーは、最初のメンバーでなくても初期化できます。共用体型の自動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。非定数の集合体初期化指定子を許可することは、C99 の言語フィーチャーです。

次の例では、共用体変数 `people` の最初の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"};
```

データ型定義の後に、変数宣言子を配置することによって、共用体データ型とその型の共用体を、同じステートメント内に定義できます。変数のストレージ・クラス指定子は、ステートメントの先頭に表示される必要があります。

無名共用体: 無名共用体 は、クラス名が付けられていない共用体です。その後、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、メンバー関数を持つことはできません。

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` メンバーまたは `private` メンバーを持つことはできません。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。

共用体の例: 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。 `length` のメンバーは、**`long int`**、**`float`**、または **`double`** のいずれでもかまいません。

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

次の例では、共用体型 `data` を、1 つのメンバーを含むものとして定義します。メンバーには、`charctr`、`whole`、または `real` という名前が付けられます。2 番目のステートメントでは、2 つの `data` 型変数、`input` と `output` を定義します。

```
union data {
    char charctr;
    int whole;
    float real;
};
union data input, output;
```

次のステートメントでは、1 つの文字を `input` に代入します。

```
input.charctr = 'h';
```

次のステートメントでは、浮動小数点数をメンバー `output` に代入します。

```
output.real = 9.2;
```

次の例では、`records` という名前の構造体の配列を定義します。 `records` の各エレメントには、整数 `id_num`、整数 `type_of_input`、および `union` 変数である `input` の 3 つのメンバーが含まれます。 `input` には、前の例で定義された `union` データ型があります。

```
struct {
    int id_num;
    int type_of_input;
    union data input;
} records[10];
```

次のステートメントでは、文字を、`records` の 1 番目のエレメントの構造体メンバー **`input`** に代入します。

```
records[0].input.charctr = 'g';
```

列挙

列挙型 は、名前付き整数定数である値のセットから構成されているデータ型です。これは、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。列挙型の名前付き値は、*列挙型定数* と呼ばれます。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。

列挙型データ型と、あるステートメントに列挙型が指定されたすべての変数を定義できます。または、列挙型をその型の変数の定義とは別に宣言できます。データ型 (オブジェクトではなく) に関連した ID は、*列挙型タグ* と呼ばれます。列挙が異なれば、列挙型も異なります。

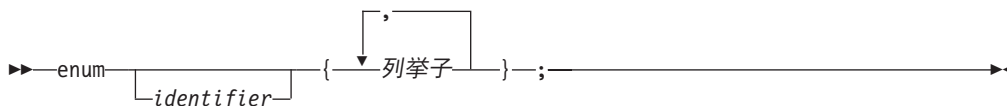
互換列挙型

C C では、列挙された型それぞれは、それを表す整数型と互換性がなければなりません。列挙型変数および定数は、コンパイラによって整数型のように取り扱われます。したがって、C では型互換性に関係なく、列挙された異なる型の値を自由に混合することができます。

C 表示される列挙型と整数型の互換性は、コンパイラ・オプションと関連プラグマによって制御されます。enum コンパイラ・オプションおよび関連する #pragma の詳しい説明については、「*XL C/C++ コンパイラ・リファレンス*」を参照してください。

C++ C++ では、列挙された型を、お互い別個のものであり、整数型とは異なるものとして取り扱います。整数を列挙型の値として使用するためには、整数を明示的にキャストしなければなりません。

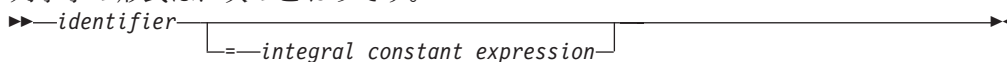
列挙データ型の宣言: 列挙型宣言には、**enum** キーワードと、その後に続くオプションの ID (列挙型タグ) および中括弧で囲まれた列挙子のリストが含まれます。コンマは、列挙子のリスト内で各列挙子を分離します。C99 では、最後の列挙子と右中括弧の間に後続のコンマが使用できます。列挙型の宣言形式は、以下のとおりです。



後に ID が続くキーワード **enum** は、データ型に名前を付けます (**struct** データ型のタグと同様)。列挙子のリストでは、データ型と値のセットが提供されます。

C では、それぞれの列挙子は整数値を表します。C++ では、各列挙子は、整数値に変換可能な値を表します。

列挙子の形式は、次のとおりです。



スペースを節約するために、列挙型を **int** のスペースよりも小さなスペースに保管することができます。

列挙型定数: 列挙データ型を定義するときは、列挙データ型が表す ID のセットを指定します。このセットの各 ID は、列挙型定数 です。

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 (=) と定数式によって定数に明示値が与えられます。ID 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 ID は、前の ID で表される値よりも 1 つ大きい整数値を受け取ります。

C C では、列挙型定数には、**int** 型が指定されます。整数定数のように、列挙型定数の型は、符号なしのサフィックス (**u** または **U**)、long 整数 (**l** または **L**)、および long long 整数 (**ll** または **LL**) によって変更することができます。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 C++ の場合は、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。

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;
}
```

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      */

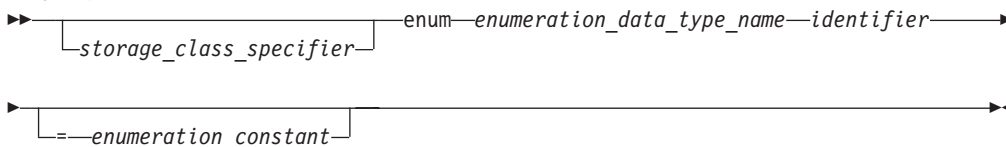
enum grain { oats=1, wheat, barley, corn, rice };
/*          1      2      3      4      5      */

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      */
```

列挙変数の定義: 列挙型変数定義の形式は、次のとおりです。



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。

C++ 列挙変数の初期化指定子には、`=` 記号と、その後にくく `enumeration_constant` が含まれます。C では、初期化指定子は、関連した列挙型と同じ型を持つ必要があります。

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。

型指定子

```
enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;
```

型指定子 `enum grain` は、`g_food` の値が列挙データ型 `grain` のメンバーであることを指示します。

▶ **C++** **enum** キーワードは、列挙型を使用して変数を宣言する場合はオプションです。ただし、列挙そのものを宣言する場合は必須です。例えば、次のステートメントは、両方とも列挙型の変数を宣言しています。

```
enum grain g_food = barley;
    grain cob_food = corn;
```

列挙型および列挙オブジェクトの定義: 型定義の後に、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。次に例を示します。

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

▶ **C++** 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;
```

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。

列挙を使用したプログラムの例: 次のプログラムでは、入力として整数を受け取ります。出力は、整数に関連付けられた曜日にフランス語の名前を付けるセンテンスです。整数が曜日に関連していない場合は、プログラムによって "C'est le mauvais jour." と印刷されます。

```
/**
 ** Example program using enumerations
 **/

#include <stdio.h>

enum days {
    Monday=1, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
} weekday;
```

```

void french(enum days);

int main(void)
{
    int num;

    printf("Enter an integer for the day of the week.  "
           "Mon=1,...,Sun=7\n");
    scanf("%d", &num);
    weekday=num;
    french(weekday);
    return(0);
}

void french(enum days weekday)
{
    switch (weekday)
    {
        case Monday:
            printf("Le jour de la semaine est lundi.\n");
            break;
        case Tuesday:
            printf("Le jour de la semaine est mardi.\n");
            break;
        case Wednesday:
            printf("Le jour de la semaine est mercredi.\n");
            break;
        case Thursday:
            printf("Le jour de la semaine est jeudi.\n");
            break;
        case Friday:
            printf("Le jour de la semaine est vendredi.\n");
            break;
        case Saturday:
            printf("Le jour de la semaine est samedi.\n");
            break;
        case Sunday:
            printf("Le jour de la semaine est dimanche.\n");
            break;
        default:
            printf("C'est le mauvais jour.\n");
    }
}

```

複素数型

複素数型は、実数部および虚数部の 2 つの部分から成り立っています。虚数型は、虚数部だけで成り立っています。

複素数型には、次の 3 つの型指定子があります。

- **float**
- **double**
- **long double**

複素数型のデータ・オブジェクトを宣言するには、次の型指定子のいずれかを使用します。



虚数単位 `I` は、**float Complex** 型の定数です。事前定義マクロ `_Complex_I` は、虚数単位の値を使用して、型 `const float _Complex` の定数式を表します。

複素数型および実浮動小数点型は、総称して *浮動小数点型* と呼ばれます。それぞれの浮動小数点型は、対応する実際の型を持っています。実浮動小数点型の場合、それは同じ型です。複素数型の場合、それは、型名からキーワード **_Complex** を削除することによって得られる型です。

複素数型の表記および位置合わせの要件は、対応する実際の型の 2 つのエレメントを含んでいる配列型と同じです。実数部は最初のエレメントと等しく、虚数部は 2 番目のエレメントと等しくなります。

算術変換は、複素数型の実際の型の変換と同じです。いずれかのオペランドが複素数型の場合、結果は複素数型になり、実数部に小さな方の型を持つオペランドの方が、大きな方の実際の型に対応する複素数型へプロモートされます。例えば、**double _Complex** を **float _Complex** に追加すると、結果の型として **double _Complex** が生まれます。

複素数型を実際の型にキャストすると、虚数部がドロップします。実際の型の値が複素数型に変換されると、複素数結果値の実数部は、対応する実際の型への変換規則によって決定され、複素数結果値の虚数部は正のゼロまたは符号なしのゼロになります。

等号演算子および非等号演算子は、実際の型の場合と同じ振る舞いを持ちます。比較演算子はオペランドのような複素数型を持つことができません。

C99 複素数型は、ヘッダー・ファイル、`/usr/include/complex.h` で定義されます。

次のプログラムをご覧ください。

```
#include <complex.h>
int main() {...}
```

プログラムが、XL C コンパイラーでコンパイルされた場合、C99 `<complex.h>` ヘッダー・ファイルが含まれます。しかし、C++ には `complex.h` ヘッダーも用意されており、これには非 C99 コードが含まれます。プログラムが XL C++ コンパイラーでコンパイルされた場合、C++ `<complex.h>` ヘッダーがインクルードされます。また AIX には、USL スタイルのヘッダー、`<complex.h>` が用意されており、これには非 C99 コードが含まれます。複素数型の C99 の動作を保証するには、マクロ `_C99_COMPLEX_HEADER__` を定義する `-qlanglvl=c99complexheader` を用いてコンパイルする必要があります。

関連参照

- 29 ページの『複素数リテラル』

型修飾子

C は、3 つの型修飾子 **const**、**volatile**、および **restrict** を認識します。C++ は、型修飾子 **const** および **volatile** を *cv-qualifiers* として参照し、型修飾子 **restrict** を言語拡張として認識します。いずれの言語においても、**const** および **volatile** は左辺値の式でのみ意味があります。C++ では、C では許可されない関数に *cv-qualifier* を適用することができます。型修飾子 **restrict** はポインターにのみ適用できます。

const および volatile キーワードの構文

volatile または **const** ポインターの場合は、* 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・オブジェクトを指すポインターの場合は、型指定子、修飾子、およびストレージ・クラス指定子を任意の順序で使用できます。次に例を示します。

```
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 */
```

次の例では、y を指すポインターは定数です。y が指す値は変更できますが、y の値は変更できません。

```
int * const y
```

次の例では、y が指す値は定数整数で、この値は変更できません。ただし、y の値は変更できます。

```
const int * y
```

volatile および **const** 変数の他の型の場合は、定義 (または宣言) 内のキーワードの位置は、あまり重要ではありません。次に例を示します。

```
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;
```

これらの例では、構造変数 group のみが **volatile** 修飾子を受け取ります。同様に、**volatile** キーワードの代わりに **const** キーワードを指定した場合は、構造変数 group のみが **const** 修飾子を受け取ります。**const** および **volatile** 修飾子が構造体、共用体、またはクラスに適用されるときは、構造体、共用体、またはクラスのメンバーにも適用されます。

列挙型、クラス、構造体、および共用体変数は、**volatile** または **const** 修飾子を受け取ることができますが、列挙型、クラス、構造体、および共用体タグは、**volatile** または **const** を持ちません。例えば、以下の例で、**blue** 構造体は、**volatile** 修飾子を持ちません。

```
volatile struct whale {
    int weight;
    char name[8];
} beluga;
struct whale blue;
```


キーワード **volatile** および **const** は、キーワード **enum**、**class**、**struct**、および **union** をそれらのタグから分離できません。

volatile 関数または **const** 関数は、それが非静的メンバー関数である場合にのみ、宣言または定義できます。どの関数も、**volatile** または **const** 関数を指すポインターを戻すように定義または宣言できます。

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。

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。

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。

const として宣言されたオブジェクトは、プログラムの実行全体に渡ってではなく、その存続時間に定数として残ることが保証されます。このため、**const** オブジェクトは定数式では使用できません。以下の例では、**const** オブジェクト **k** は、**foo** 内で宣言され、**foo** の引き数の値へ初期化され、関数が戻るまで定数として残ります。**C** では、**k** は、**foo** が呼び出されるまで未知のため、配列長を指定するためには使用できません。

```
void foo(int j)
{
    const int k = j;
    int ary[k];    /* Violates rule that the length of each
                    array must be known to the compiler */
}
```

C では、ブロックの外側で宣言された **const** オブジェクトは外部結合を持ち、ファイル間で共用することができます。以下の例では、**k** は、おそらく別のファイルで定義されているため、配列長の指定に使用することはできません。

```
extern const int k;
int ary[k];    /* Another violation of the rule that the length of
                each array must be known to the compiler */
```

明示的なストレージ・クラスのない `const` オブジェクトのトップレベル宣言は、C++ では **static** と見なされますが、C では **extern** と見なされます。

```
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++ */
```

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` 宣言が初期化指定子を持たなければなりません。

 このセクションでのここから先の説明は、C++ だけに適用されます。

`const` オブジェクトが整数で、定数に初期化されている場合、これを定数式で表示することができます。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、`const` オブジェクトはデフォルトで内部結合を持つため、`const` オブジェクトをヘッダー・ファイルで定義することができます。

`const` ポインター

ポインター用のキーワード **const** は、型の前か後、もしくは前後ともに表示することができます。使用できる宣言は以下のとおりです。

```
const int * ptr1; /* A pointer to a constant integer:
                  the value pointed to cannot be changed */
int * const ptr2; /* A constant pointer to integer:
                  the integer can be changed, but ptr2
                  cannot point to anything else */
const int * const ptr3; /* A constant pointer to a constant integer:
                        neither the value pointed to
                        nor the pointer itself can be changed */
```

オブジェクトを `const` となるよう宣言することは、`this` ポインターが `const` オブジェクトを指すポインターであることを意味します。`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。

`const` メンバー関数

メンバー関数 **const** を宣言することは、`this` ポインターが `const` オブジェクトを指すポインターであることを意味します。クラスのデータ・メンバーは、関数内で `const` になります。関数は、値をまだ変更することができますが、そのためには `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。

volatile 型修飾子

volatile 修飾子は、データ・オブジェクトに対するメモリー・アクセスの整合性を維持します。揮発性オブジェクトは、それらの値が必要になるたびにメモリーから読み取られ、変更されるたびにメモリーへ書き戻されます。**volatile** 修飾子は、コ

コンパイラーの制御または検出以外の方法 (システム・クロックによって更新された変数など) で、値を変更できるデータ・オブジェクトを宣言します。その結果、コンパイラーは、オブジェクトを参照するコードに対して特定の最適化を適用しないよう通知されます。

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。

restrict 型修飾子

C **restrict** 型修飾子は、ポインターにしか適用できません。この型修飾子を使用するポインター宣言は、ポインターおよびポインターがアクセスするオブジェクトの間で特別な関係を確認し、ポインターおよび そのポインターを基にした式が、オブジェクトの値に直接的または間接的にアクセスする唯一の方法になります。

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でき、プログラムの実行中にそれを変更することができます。 **restrict** 型修飾子は、**restrict** で修飾されたポインターがアドレスした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、制限付きポインターを `foo()` のパラメーターとして示し、2 つの制限付きポインターによっ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との間の区別が付きません。

```
{
    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が含まれるネストされたブロックでは、外部ブロックから内部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たブロックが実行を終了したときです。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。

 C++ は、C99 と互換性を持たせるための非直交の言語拡張として、**restrict** キーワードをサポートします。コンパイラーは、キーワードを消費および無視しますが、誤った使用法に対しては、診断メッセージを発行します。これが非直交である理由は、既存の C++ プログラムが *restrict* を変数名として使用できるためです。__restrict__ キーワードはユーザーのネーム・スペースを侵害しないように提供されています。

cv-qualifiers に適用するすべての Standard C++ および C++98 指定は、**restrict** および __restrict__ キーワードに適用します。これらの指定には、修飾変換および関数の多重定義を制御する規則が含まれています。

例えば、__restrict__ 修飾子は cv-qualifier と同じように型指定に影響を与えます。

```
int * __restrict__ p;      // p is a restricted pointer to int
int * __restrict__ * q;    // q is a pointer to a restricted pointer to int
void foo(float * __restrict__ a, float * __restrict__ b);
                        // a and b point to different (non-overlapping) objects
```

__restrict__ 修飾子はまた、cv-qualifier と同じように多重定義関数のパラメーター型指定にも影響を与えます。これによって、多重定義セットに影響を与えます。

```
void foo(int * __restrict__ * a) { printf("First\n"); } // candidate function 1.
void foo(int * * b) { printf("Second\n"); }           // candidate function 2.

int main() {
    int a;
    int *b=&a;
    int * __restrict__ *c=&b;
    int * *d =&b;
    foo(c);
    foo(d);
}
```

候補関数 1 には、候補関数 2 より多くの cv-qualified 関数仮パラメーターがあります。呼び出し関数が foo(d) の場合、候補関数 2 のほうが好ましい候補関数です。ただし、呼び出し関数が foo(c) の場合、候補関数 1 のみが呼び出されます。

関連参照

- 79 ページの『型修飾子』

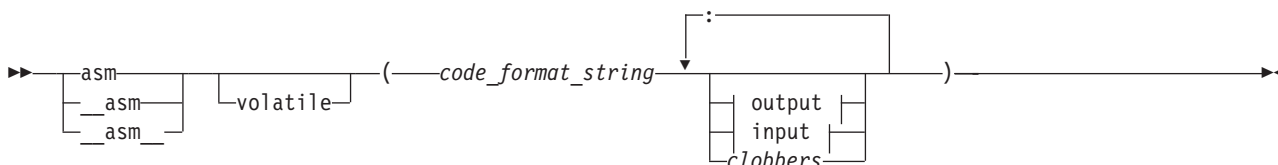
- 250 ページの『多重定義された関数の制約事項』

asm 宣言

キーワード **asm** は、アセンブリー・コードを表します。コンパイラーは、厳しい言語レベル下でコンパイルされる場合、宣言内のキーワード **asm** を認識してそれを無視します。

拡張言語レベルで、コンパイラーは、C ソース・ステートメントと C++ ソース・ステートメントの間の組み込みアセンブリー・コード・フラグメントに対して部分的なサポートを提供します。この拡張機能は、一般的なシステム・プログラミング・コード、カーネル、および最初は GNU C で開発されたデバイス・ドライバで使用するためにインプリメントされています。

構文は以下のとおりです。



input:



output:



ここで、

volatile

アセンブラー命令が、*output*、*input*、または *clobbers* にリストされていないメモリーを更新する可能性があることを、コンパイラーに指示します。

code_format_string

asm 命令のソース・テキストで、*printf* フォーマット指定子に類似したストリング・リテラルです。

input コンマで区切られた、入力オペランドのリスト。

output

コンマで区切られた、出力オペランドのリスト。

clobbers

コンマで区切られた、二重引用符で囲まれた登録名のリスト。これらは、*asm* 命令によって更新できる登録です。

constraint

オペランドの制約を指定するストリング・リテラルで、制約ごとに 1 文字。

C_expression

値が *asm* 命令のオペランドとして使用される C または C++ 式。出力オペランドは、変更可能な左辺値でなければなりません。

次の制約がサポートされます。

- =** 書き込み専用オペランド。
- +** 読み取りおよび書き込みオペランド。
- &** 入力オペランドを使用して命令を終了する前に、オペランドは変更される場合があります。入力として使用されるレジスターは、ここで再利用してはなりません。
- b** ゼロ以外のレジスターを使用。
- f** 浮動小数点レジスターを使用。
- g** 汎用レジスター、メモリー、または即値オペランドを使用。
- i** 即時整数オペランド。
- m** マシンによってサポートされるメモリー・オペランド。
- n** *i* と同じ方法で処理。
- o** *m* と同じ方法で処理。
- r** 汎用レジスターを使用。
- v** ベクトル・レジスターを使用。
- 0、1、2、...8** マッチング制約。対応する入力と同じレジスターを出力で割り振る。
- I、J、K、M、N、O、P、G、S、T** 定数値。オペランド内の式を折り返し、その値を `%` 指定子に置換する。

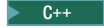
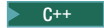
制約事項

asm 文中の命令の数は、最大で 63 に制限されます。


アセンブラー命令は、**asm** ステートメント内で必要なものを完備していなければなりません。**asm** ステートメントは、命令を生成するためにのみ使用できます。残りのプログラムに対するすべての関連付けは、*output* および *input* オペランド・リストによって設定する必要があります。


不完全型

以下は、不完全型です。

- 型 **void**
- 不明サイズの配列
- 不完全型のエレメント配列
- 定義のない構造体、共用体、または列挙型
-  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
-  宣言されているが、定義されていないクラス

void は、完全にできない不完全型です。不完全な構造体または共用体および列挙型タグは、オブジェクトを宣言するために使用される前に、完全にしなければなりません。ただし、不完全な構造体または共用体に対するポインターは、定義できます。

 サイズが指定されていない配列は不完全型です。ただし、定数式の代わりに、配列サイズを可変長配列を示す `[*]` によって指定した場合、サイズは指定されたものと見なされ、配列型は完了型と見なされます。

 関数宣言子がその関数の定義の一部ではない場合、パラメーターは不完全な型を持つこともあります。パラメーターはまた、[*] 表記で示される可変長配列型を持つことがあります。

次は、不完全型の例です。

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

void は、完全にできない不完全型です。不完全な構造体、共用体、または列挙型タグは、オブジェクトの宣言に使用される前に完全にしなければなりません。ただし、不完全な構造体または共用体に対するポインターは定義できます。

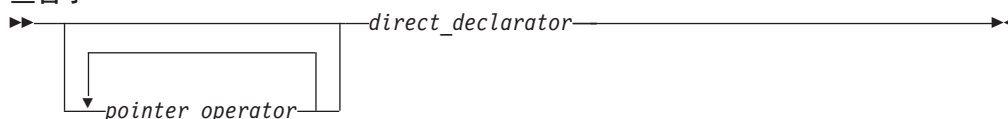
第 4 章 宣言子

宣言子は、データ・オブジェクトまたは関数を指定します。宣言子は、多くのデータ定義と宣言および一部の型定義で使われます。

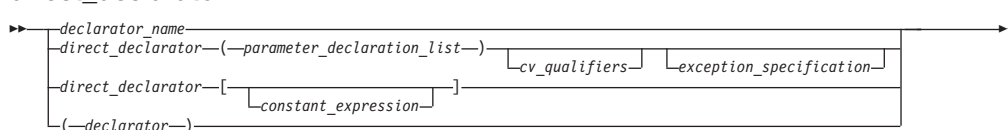
宣言子では、オブジェクトの型として、配列、ポインター、または参照を指定できます。また、宣言子内で初期化を実行することもできます。

宣言子の形式は、次のとおりです。

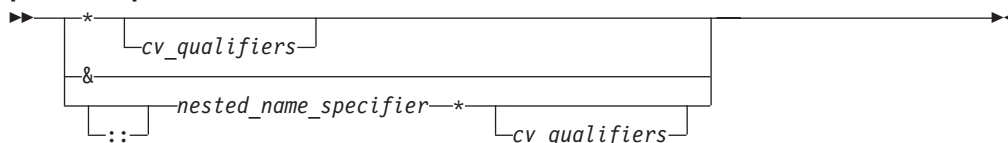
宣言子



direct_declarator



pointer_operator



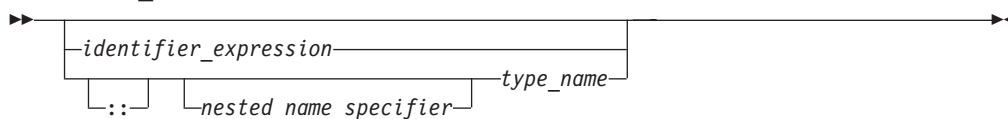
C C における宣言子名の構文

declarator_id



C++ C++ における宣言子名の構文

declarator_id



宣言子構文に関する注意

- *cv_qualifiers* 変数は、**const** および **volatile** のどれか、またはその両方を組み合わせたものを表します。C では、**volatile** または **const** 関数は、宣言または定義できません。ただし、C++ では、非静的メンバー関数を *cv-qualifier* の **const** または **volatile** で修飾することができます。

- **C++** 変数 *exception_specification* と *nested_name_specifier*、およびスコープ解決演算子 `::` は、C++ においてのみ使用可能です。
- **C++** *identifier_expression* は、修飾 ID または非修飾 ID のいずれにもなれます。スコープ解決演算子、テンプレート、およびその他の拡張機能による複雑さは、C には存在しません。
- **C++** *nested_name_specifier* は、修飾された ID 式です。

次の表に、いくつかの宣言子の例を示します。

例	説明
<code>int owner</code>	<code>owner</code> は、 int データ・オブジェクトです。
<code>int *node</code>	<code>node</code> は、 int データ・オブジェクトを指すポインターです。
<code>int names[126]</code>	<code>names</code> は、126 個の int エレメントの配列です。
<code>int *action()</code>	<code>action</code> は、 int を指すポインターを戻す関数です。
<code>volatile int min</code>	<code>min</code> は、 volatile 修飾子がある int です。
<code>int * volatile volume</code>	<code>volume</code> は、 int を指す volatile ポインターです。
<code>volatile int * next</code>	<code>next</code> は、 volatile int を指すポインターです。
<code>volatile int * sequence[5]</code>	<code>sequence</code> は、 volatile int オブジェクトを指す 5 つのポインターの配列です。
<code>extern const volatile int clock</code>	<code>clock</code> 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。

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。

各データ型の初期化特性および特別な要件は、そのデータ型のセクションに記述されています。

初期化指定子は、`=` 記号と、その後続く、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なりません。初期式は、データ・オブジェクトの最初の値に評価されます。

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;
```

共用体、構造体、および集合体クラス (コンストラクター、基底クラス、仮想関数、または `private` か `protected` メンバーがないクラス) の場合は、初期式のセットは、初期化指定子が ストリング・リテラルでない限り、中括弧で囲む必要があります。

中括弧で囲まれた初期化指定子リストを使用して初期化された配列、構造体、または共用体では、初期化されないメンバーまたは添え字を適切な型のゼロに暗黙で初期化します。

例

次の例では、配列 `grid` の最初の 8 つの要素のみが、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 つの要素は、明示的にゼロに初期化されたかのように初期化されます。

```
static short grid[3][4] = {0, 0, 0, 1, 0, 0, 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
<code>grid[0][0]</code>	0	<code>grid[1][2]</code>	1
<code>grid[0][1]</code>	0	<code>grid[1][3]</code>	1
<code>grid[0][2]</code>	0	<code>grid[2][0]</code>	0
<code>grid[0][3]</code>	1	<code>grid[2][1]</code>	0
<code>grid[1][0]</code>	0	<code>grid[2][2]</code>	0
<code>grid[1][1]</code>	0	<code>grid[2][3]</code>	0

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

C++ では、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 では、グローバル・スコープでこれと同じことを行うことはできません。

コードが初期化を含む宣言をジャンプする場合は、コンパイラーはエラーを生成します。例えば、次のコードは無効です。

```
goto skiplabel;    // error - jumped over declaration
int i = 3;         // and initialization of i

skiplabel: i = 4;
```

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、その後続く、中括弧で囲まれ、コンマで区切ら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。


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。ポインターは、一度に 1 つの値しか保持できないという意味で、スカラー型として分類されます。

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。
- 配列の要素、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。

 このセクションでのここから先の説明は、C だけに適用されます。

ポインターを使用して、**register** ストレージ・クラス指定子で宣言されたオブジェクトを参照することはできません。

同じ型修飾子を持つ 2 つのポインター型は、それらが互換タイプのオブジェクトを指す場合には互換性があります。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。

ポインターの宣言

次の例では、`pcoat` を、**long** 型が指定されたオブジェクトを指すポインターとして宣言します。

```
long *pcoat;
```

キーワード **volatile** が * の前に現れる場合は、宣言子は、**volatile** オブジェクトを指すポインターを記述します。キーワード **volatile** が、* と ID の間に現れる場合は、宣言子は **volatile** ポインターを記述します。キーワード **const** は、**volatile** キーワードと同じように機能します。次の例では、`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 short * const pvolt;
```

次の例では、`pnut` を、**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として宣言します。

```
extern int volatile *pnut;
```

次の例では、`psoup` を、**float** 型が指定されたオブジェクトを指す **volatile** ポインターとして定義します。

```
float * volatile psoup;
```

次の例では、`pfowl` を、**bird** 型の列挙オブジェクトを指すポインターとして定義します。

```
enum bird *pfowl;
```

次の例では、`pvish` を、パラメーターを取らずに **char** オブジェクトを戻す関数を指すポインターとして宣言します。

```
char (*pvish)(void);
```

ポインターの割り当て

代入演算でポインターを使用するときは、演算のポインターの型間で互換性を保つ必要があります。

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 */
```

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、添え字がない配列名を、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 に評価される整数定数式 (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。

ポインターの使用

アドレス演算子 (&) と間接演算子 (*) の 2 つの演算子は、一般にポインターを使用した作業で使用できます。& 演算子を使用して、オブジェクトのアドレスを参照できます。例えば、次の関数割り当ては、x のアドレスを変数 p_to_int に割り当てます。これは、変数 p_to_int をポインターとして定義します。

```
void f(int x, int *p_to_int)
{
    p_to_int = &x;
}
```

* (間接) 演算子によって、ポインターが参照するオブジェクトの値にアクセスできます。次の例の割り当てでは、y に、p_to_float が指すオブジェクトの値を割り当てます。

```
void g(float y, float *p_to_float) {
    y = *p_to_float;
}
```

次の割り当ての例では、z の値を、*p_to_char が参照する変数に割り当てます。

```
void h(char z, char *p_to_char) {
    *p_to_char = z;
}
```

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によって、ポインターの値を増やします。例えば、ポインターが配列の 2 番目のエレメントを参照している場合は、++ によって、ポインターに、配列の 3 番目のエレメントを参照させます。

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によって、ポインターの値を減らします。例えば、ポインターが配列の 2 番目のエレメントを参照している場合は、-- によって、ポインターに、配列の 1 番目のエレメントを参照させます。

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 つのポインタの比較は、==、!=、<、>、<=、および >= の各演算子を用いて行うことができます。

ポインタの比較は、ポインタが同じ配列の要素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の比較は、ポインタが異なる配列の要素を指すときにも実行できます。

ポインタには、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の値、または NULL ポインタを割り当てることができます。

ポインタを使用したプログラムの例

次のプログラムには、ポインタ配列が含まれています。

```

/*****
**  Program to search for the first occurrence of a specified
**  character string in an array of character strings.
*****/

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

#define SIZE 20

int main(void)
{
    static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
    char * find_name(char **, char *);
    char new_name[SIZE], *name_pointer;

    printf("Enter name to be searched.\n");
    scanf("%s", new_name);
    name_pointer = find_name(names, new_name);
    printf("name %s%sfound\n", new_name,
        (name_pointer == NULL) ? " not " : " ");
} /* End of main */

/*****
**  Function find_name. This function searches an array of
**  names to see if a given name already exists in the array.
**  It returns a pointer to the name or NULL if the name is
**  not found.
**
**  char **array is a pointer to arrays of pointers (existing names)
**  char *strng is a pointer to character array entered (new name)
*****/

char * find_name(char **array, char *strng)
{
    for (; *array != NULL; array++) /* for each name */
    {
        if (strcmp(*array, strng) == 0) /* if strings match */
            return(*array); /* found it! */
    }
    return(*array); /* return the pointer */
} /* End of find_name */

```

このプログラムとの対話により、次のセッションが作成されます。

出力 Enter name to be searched.

入力	Mark
出力	name Mark found
または	
出力	Enter name to be searched.
入力	Deborah
出力	name Deborah not found


配列

array は、同じデータ型のオブジェクトのコレクションです。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。添え字演算子 (`[]`) 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し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割り当てたり、2 つの配列を同じかどうか比較し、自己認識サイズを戻したりする操作は、いずれの言語でもサポートされていません。

配列型は、特定型のオブジェクトのセット用に隣接して割り振られたメモリを表します。配列型は、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++ では、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。

 同じように修飾された 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、同様に互換性があります。例えば、次のような場合です。

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

には、互換性はありません。互換性のある 2 つの配列型の複合型は、複合エレメント型を持つ配列です。オリジナルの型のサイズはどちらも同等でなければなりません (既知である場合。オリジナルの配列型の 1 つのサイズが既知である場合、複合型はそのサイズを持ちます。例えば:

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

この場合、`ex3` と `ex4` の複合型は `char[42]` となります。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。

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]` ではなく `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。例外は、配列名が配列自

身を渡す場合です。例えば、配列名は、それが **sizeof** 演算子またはアドレス (&) 演算子のオペランドであるときに、配列全体を渡します。

同様に、関数のパラメーター・リスト内の配列型は、対応するポインター型に変換されます。引き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。

C 最適化に有益なこの情報を保存するには、**static** キーワードを使用する引き数配列の指標を宣言します。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・サイズです。

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。

この言語フィーチャーは、C99 言語レベルで使用できます。

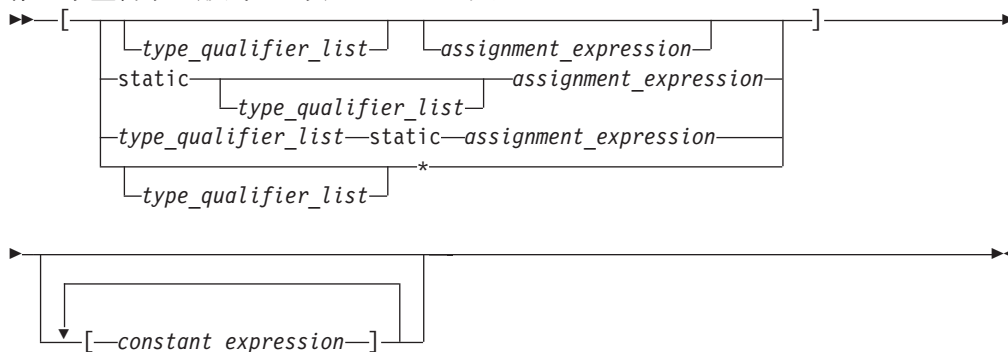
フィーチャーの使用例を次に示します。

```
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 */
```

配列の宣言

配列宣言子には、ID と、その後に続く、オプションの添え字宣言子 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。

添え字宣言子の形式は、次のとおりです。



ここで、*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。

C 宣言をブロックまたは関数スコープに表示する場合は、非定数式を配列添え字宣言子に指定することができ、その配列は可変的に変更される型として見なされます。配列添え字演算子のブラケット内のアスタリスクは、サイズ指定のない可

変長配列を示します。この場合、配列は、定義ではない関数宣言 (つまり、関数プロトタイプ・スコープを使用した宣言) でのみ使用できる、可変的に変更される型と見なされます。

添え字宣言子は、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、または添え字は、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 番目の添え字は、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]
```

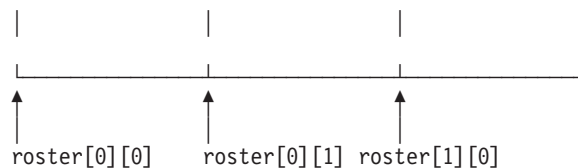
次の例では、**int** 型の 6 つのエレメントを含む 2 次元配列を定義します。

```
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、一番後の添え字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) 添え字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

最初の添え字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、初期化指定子を添え字宣言子と比較し、第 1 次元のエレメントの数を決めます。

可変長配列

可変長配列とは、長さが実行時に決定される自動ストレージ期間を持った配列です。可変長配列型は、正確なストレージの量を割り振る構成を提供します。この量は、アプリケーションが実際に実行されるときに、はじめて決定することができます。

▶ **C++** C++ は、C99 との互換性のために、可変長配列をサポートするように Standard C++ および C++98 を拡張します。この拡張には、可変長配列型への参照のサポートは含まれません。関数仮パラメーターも、可変長配列型への参照にはならない可能性があります。

可変長配列は、次のように書き込むことができます。

▶ `array_identifier` — [`expression` `*`] — `type-qualifier-list`

配列のサイズが、式の代わりに、* で示される場合、可変長配列は指定されていないサイズとして見なされます。そのような配列は完全型として見なされますが、関数プロトタイプ・スコープの宣言でのみ使用することができます。

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りません。可変的に変更される型で宣言された ID は、普通の ID である必要があるため、構造体または共用体のメンバーになることはできません。可変長配列は、初期化できません。

可変長配列は、sizeof 式のオペランドとすることができます。この場合、変数配列の各インスタンスのサイズが存続時間中に変更しなくても、オペランドはランタイムで評価され、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 式で 사용할ことができます。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とすることができます。必要なサイズ式を関数定義に指定しなければなりません。コンパイラーは、関数へのエントリー時に可変に変更されたパラメーターのサイズ式を評価します。パラメーターとして可変長配列で宣言された関数の場合、以下のように、

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

可変長配列引き数のサイズは、関数定義のサイズと一致する必要があります。

関連参照

- 188 ページの『関数呼び出しおよび引き数の受け渡し』

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、コンマで区切られたリストです。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る場合、初期化されないエレメントは、該当する型の 0 の値を受け取ります。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適用されます。(static キーワードで宣言され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には、静的ストレージ期間があります。)

次の定義では、完全に初期化される 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。添え字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 の値に関して、number1[0] および number1[1] は前の定義と同じですが、number1[2] は 0 です。

添え字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、初期化指定子が 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。

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、コンマで区切られた定数のリスト。各定数は、文字に含めることができます。
- スtring定数 (定数を囲む中括弧はオプション)

String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String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++ スtringを指定して文字の配列を初期化する場合、String内の文字数 — 終端の「¥0」を含む — が、配列の中のエレメント数を超えてはなりません。

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、最後の次元の添え字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。

次の定義では、12 のエレメントの配列の中の 6 つのエレメントを明示的に初期化します。

```
static int matrix[3][4] =
{
    { 1, 2 },
    { 3, 4 },
    { 5, 6 }
};
```

`matrix` の初期値は、次の表のとおりです。他のすべてのエレメントはゼロに初期化されます。

エレメント	値	エレメント	値
<code>matrix[0][0]</code>	1	<code>matrix[1][2]</code>	0
<code>matrix[0][1]</code>	2	<code>matrix[1][3]</code>	0
<code>matrix[0][2]</code>	0	<code>matrix[2][0]</code>	5
<code>matrix[0][3]</code>	0	<code>matrix[2][1]</code>	6
<code>matrix[1][0]</code>	3	<code>matrix[2][2]</code>	0
<code>matrix[1][1]</code>	4	<code>matrix[2][3]</code>	0

指定初期化指定子を使用した配列の初期化

C C は、集合体型の指定初期化指定子をサポートしています。*designator* は、初期化する特定の配列エレメントを指し、"*[index]*" の形式を持ちます。ここで、*index* は定数式です。指定機能リスト は、任意の集合体型用の 1 つまたは複数の指定機能の組み合わせです。指定 は、等号が後に付いた指定機能リストで構成されます。

指定がなければ、配列の初期化は、初期化指定子が示す順番で起こります。指定が初期化指定子内で表示されると、指定機能によって指示された配列エレメントが初期化されます。その後の初期化は初期化リスト順に正方向に進み、事前に初期化された配列エレメントをオーバーライドし、明示的に初期化されていない任意の配列エレメントをゼロに初期化します。

指定初期化指定子を持たない宣言構文は、初期化指定子リストを示すために中括弧を使用しますが、大括弧形式 として参照されます。初期化で完全な大括弧形式および最小限の大括弧形式を使用すると、誤解される可能性が少なくなります。以下に、同じものを実行する多次元配列 *matrix* の有効な宣言を示します。

matrix[3][0][0] で始まる行全体など明示的に初期化されていないすべての配列エレメントは、ゼロに初期化されます。

```
/* minimally bracketed form */
int matrix[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};

/* fully bracketed form */
int matrix[4][3][2] = {
    {
        { 1 },
    },
    {
        { 2, 3 },
    },
    {
        { 4, 5 },
        { 6 }
    }
};

/* incompletely but consistently bracketed initialization */
int matrix[4][3][2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};
```

配列の初期化中に事前のサブオブジェクトの初期化をオーバーライドすることは、指定初期化指定子にとって必要な振る舞いです。つまり、配列の両端からスペースを "割り振る" ために単一指定機能が使用されます。指定初期化指定子 `[MAX-5] = 8` は、添え字 `MAX-5` における配列エレメントを値 `8` に初期化する必要があることを意味します。配列添え字大括弧で、定数式を囲む必要があります。

```
int a[MAX] = {
    1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

MAX が 15 の場合、a[5] から a[9] まではゼロに初期化されます。MAX が 7 の場合、a[2] から a[4] まだが、まずそれぞれ値 5、7、および 9 を持ち、その後、値 8、6、および 4 によってオーバーライドされます。つまり、MAX が 7 の場合、初期化は宣言が書き込まれた場合と同じになります。

```
int a[MAX] = {
    1, 3, 8, 6, 4, 2, 0
};
```

配列を使用したプログラムの例

次のプログラムでは、prices と呼ばれる浮動小数点配列を定義します。

最初の for ステートメントは、prices のエレメントの値を印刷します。2 番目の for ステートメントは、prices の各エレメントの値に 5 % を加算し、total にその結果を割り当て、これらを total として印刷します。

```
/**
 ** Example of one-dimensional arrays
 **/

#include <stdio.h>
#define ARR_SIZE 5

int main(void)
{
    static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
    auto float total;
    int i;

    for (i = 0; i < ARR_SIZE; i++)
    {
        printf("price = %.2f¥n", prices[i]);
    }

    printf("¥n");

    for (i = 0; i < ARR_SIZE; i++)
    {
        total = prices[i] * 1.05;
        printf("total = %.2f¥n", total);
    }

    return(0);
}
```

このプログラムの出力は次のようになります。

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86

total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90
```

次のプログラムでは、多次元配列 salary_tbl を定義します。for ループでは、salary_tbl の値が印刷されます。

初期化指定子

```
/**
 ** Example of a multidimensional array
 **/

#include <stdio.h>
#define ROW_SIZE    3
#define COLUMN_SIZE 5

int main(void)
{
    static int
    salary_tbl[ROW_SIZE][COLUMN_SIZE] =
    {
        { 500, 550, 600, 650, 700 },
        { 600, 670, 740, 810, 880 },
        { 740, 840, 940, 1040, 1140 }
    };
    int grade, step;

    for (grade = 0; grade < ROW_SIZE; ++grade)
        for (step = 0; step < COLUMN_SIZE; ++step)
        {
            printf("salary_tbl[%d] [%d] = %d¥n",
                   grade, step, salary_tbl[grade][step]);
        }

    return(0);
}
```

このプログラムの出力は次のようになります。

```
salary_tbl[0] [0] = 500
salary_tbl[0] [1] = 550
salary_tbl[0] [2] = 600
salary_tbl[0] [3] = 650
salary_tbl[0] [4] = 700
salary_tbl[1] [0] = 600
salary_tbl[1] [1] = 670
salary_tbl[1] [2] = 740
salary_tbl[1] [3] = 810
salary_tbl[1] [4] = 880
salary_tbl[2] [0] = 740
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140
```

関数指定子

関数指定子 **inline** は、関数のコードを、呼び出し時点のコードに組み込むようにコンパイラーに提案するために使われます。メモリーに関数命令の単一セットを作成する代わりに、コンパイラーは、**inline** 関数からコードを直接呼び出し関数へコピーします。ただし、標準に準拠したコンパイラーは、最適化を推進するためにこの提案を無視する場合があります。

 このセクションでのここから先の説明は、C++ だけに適用されます。

通常関数とメンバー関数の両方を **inline** として宣言できます。関数がクラス宣言の外部で宣言された場合でも、メンバー関数を、キーワード **inline** を使用して **inline** にすることができます。

キーワード **virtual** および **explicit** は、関数指定子として、C++ の関数宣言でのみ使用されます。

関数指定子 **virtual** は、非静的メンバー関数宣言でのみ使われます。

関数指定子 **explicit** 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。これは、オブジェクトの初期化中に不要な暗黙の型変換を制御するために使用します。明示的なコンストラクターは、直接初期化構文または明示的なキャストが使用されるオブジェクトしか構成できない点において、非明示的なコンストラクターとは異なります。

参照

C++ 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宣言子 **&** 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。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。

関数の引き数は、値によって渡されるため、関数呼び出しは、引き数の実際の値を変更しません。関数が、引き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、引き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。参照による引き数の受け渡しは、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、参照によって引き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う少し簡単です。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、引き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。

参照の初期化

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 **true** 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。

ステートメントの前のリストが **true** 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。


第 5 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。

ISO C と ISO C++ は、実行シーケンスにおいて、直前の評価のすべての副次作用が完了して、以降の評価での副次作用が発生しなくなるポイントを把握します。このような時点を評価順序点と呼びます。スカラー・オブジェクトの変更は、連続する評価順序点の間で一度のみ行うことができます。それ以外の方法で変更を行うと、結果は未定義なものとなります。評価順序点は、次の状況のように、より大きな式の一部ではない式がすべて完了すると生じます。

- 論理 AND `&&`、論理 OR `||`、条件 `?:`、またはコンマ式の第 1 オペランドの評価の後
- 関数呼び出し内の引き数の評価後
- `full` 宣言子の最後
- `full` 式の最後
- ライブラリー関数が戻る前
- 書式付き入出力関数の型変換指定子のアクション後
- 比較関数への呼び出しの前後、および比較関数への呼び出しとその関数呼び出しへの引き数として渡されたオブジェクトの動作の間。

用語 *full* 式は、初期化指定子、式ステートメント、`return` ステートメントの式、および条件、反復、または `switch` 文の制御式のことを意味する場合があります。これには、`for` ステートメントの各式が含まれます。

 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、異なる動作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ます。本章では、多重定義されない演算子の振る舞いについて説明します。

関連参照

- 109 ページの『左辺値と右辺値』
- 251 ページの『演算子の多重定義』

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。

演算子優先順位と結合順序

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。

C++ スコープ・レゾリューション演算子 (::) には、最も高い優先順位が付けられています。コンマ演算子には、最も低い優先順位が付けられます。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。


C および C++ 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	 グローバル・スペース・スコープ・レゾリューション	<code>:: name_or_qualified_name</code>
1		 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	<code>class_or_namespace :: member</code>
2		メンバー選択	<code>object . member</code>
2		メンバー選択	<code>pointer -> member</code>
2		添え字	<code>pointer [expr]</code>
2		関数呼び出し	<code>expr (expr_list)</code>
2		値生成	<code>type (expr_list)</code>
2		後置増分	<code>lvalue ++</code>
2		後置減分	<code>lvalue --</code>
2	はい	 型の識別	<code>typeid (type)</code>
2	はい	 実行時の型識別	<code>typeid (expr)</code>
2	はい	 コンパイル時にチェックされる型変換	<code>static_cast < type > (expr)</code>
2	はい	 実行時にチェックされる型変換	<code>dynamic_cast < type > (expr)</code>
2	はい	 チェックなしの変換	<code>reinterpret_cast < type > (expr)</code>
2	はい	 const の変換	<code>const_cast < type > (expr)</code>
3	はい	オブジェクトのサイズ (バイト)	<code>sizeof expr</code>
3	はい	型のサイズ (バイト)	<code>sizeof (type)</code>
3	はい	接頭部増分	<code>++ lvalue</code>
3	はい	接頭部減分	<code>-- lvalue</code>

C および C++ 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	ビット単位否定	<code>~ expr</code>
3	はい	否定	<code>! expr</code>
3	はい	単項負	<code>- expr</code>
3	はい	単項正	<code>+ expr</code>
3	はい	～のアドレス	<code>& lvalue</code>
3	はい	間接または間接参照	<code>* expr</code>
3	はい	 作成 (メモリーの割り振り)	<code>new type</code>
3	はい	 作成 (メモリーの割り振りと初期化)	<code>new type (expr_list) type</code>
3	はい	 作成 (配置)	<code>new type (expr_list) type (expr_list)</code>
3	はい	 破棄 (メモリーの割り振り解除)	<code>delete pointer</code>
3	はい	 配列の破棄	<code>delete [] pointer</code>
3	はい	型変換 (キャスト)	<code>(type) expr</code>
4		メンバー選択	<code>object .* ptr_to_member</code>
4		メンバー選択	<code>object ->* ptr_to_member</code>
5		乗算	<code>expr * expr</code>
5		除法	<code>expr / expr</code>
5		モジュロ (剰余)	<code>expr % expr</code>
6		2 項加算	<code>expr + expr</code>
6		2 項減算	<code>expr - expr</code>
7		左へのビット単位シフト	<code>expr << expr</code>
7		右へのビット単位シフト	<code>expr >> expr</code>
8		より小さい	<code>expr < expr</code>
8		より小さいまたは等しい	<code>expr <= expr</code>
8		より大きい	<code>expr > expr</code>
8		より大きいまたは等しい	<code>expr >= expr</code>
9		等しい	<code>expr == expr</code>
9		等しくない	<code>expr != expr</code>
10		ビット単位 AND	<code>expr & expr</code>
11		ビット単位排他 OR	<code>expr ^ expr</code>
12		ビット単位包含 OR	<code>expr expr</code>
13		論理 AND	<code>expr && expr</code>
14		論理包含 OR	<code>expr expr</code>
15		条件式	<code>expr ? expr : expr</code>
16	はい	単純代入	<code>lvalue = expr</code>
16	はい	乗算および代入	<code>lvalue *= expr</code>
16	はい	除法および代入	<code>lvalue /= expr</code>
16	はい	モジュロおよび代入	<code>lvalue %= expr</code>
16	はい	加算および代入	<code>lvalue += expr</code>
16	はい	減算および代入	<code>lvalue -= expr</code>
16	はい	左へのシフトおよび代入	<code>lvalue <<= expr</code>
16	はい	右へのシフトおよび代入	<code>lvalue >>= expr</code>
16	はい	ビット単位 AND および代入	<code>lvalue &= expr</code>
16	はい	ビット単位排他 OR および代入	<code>lvalue ^= expr</code>
16	はい	ビット単位包含 OR および代入	<code>lvalue = expr</code>

C および C++ 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
17	はい	 throw 式	throw <i>expr</i>
18		コンマ (順序付け)	<i>expr</i> , <i>expr</i>

関数呼び出しの引き数または 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。次のようなあいまいな式は作成しないようにしてください。

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

上記の例では、すべての C 言語インプリメンテーションで、`++y` と `func1(y)` が同じ順序で評価されるとは限りません。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わかりません。式が評価される前に、2 番目のステートメントで、`i` に値 1 が指定されている場合は、`x[1]` または `x[2]` が `func2()` への 2 番目の引き数として渡されるかどうかはわかりません。

式と優先順位の例

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場合を除き、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分離します。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();
```

左辺値と右辺値

オブジェクトとは、検査が可能で保管場所とすることのできるストレージの領域のことです。左辺値は、このようなオブジェクトを参照する式です。左辺値は、その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。用語 **変更可能な左辺値** は、左辺値によって指定オブジェクトの変更および検査が許可されていることを強調する場合に使用します。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
- オブジェクトが構造体または共用体型であり、そのメンバーの 1 つが **const** 修飾型を持つ場合。

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、メモリ内の同じアドレスに保管されるデータ値を表します。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ともに、右辺値として使用することができます。右辺値を必要とするコンテキストに左辺値が現れる場合、左辺値は暗黙的に右辺値に変換されます。ただし、この逆は **true** ではなく、右辺値を左辺値に変換することはできません。右辺値は常に、完全型または **void** 型を持っています。

C ISO C は、関数指定機能 を、関数型を持つ式として定義します。関数指定機能は、オブジェクト型または左辺値とは異なります。これは、関数の名前、または関数ポインターの間接参照の結果とすることができます。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、すべての式の結果は左辺値、右辺値、または値なしとなります。


左辺値 (lvalue)


C および C++ とともに、演算子のなかには、自身のいくつか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。次の表は、これらの演算子、およびその使用に関する追加の制約事項のリストです。

演算子	要件
& (単項)	オペランドは左辺値である必要があります。
++ --	オペランドは左辺値である必要があります。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
 int& f()	f() への関数呼び出し

 このセクションでの以降の説明は、プラットフォーム固有で、C のみに適用されます。

GNU C 言語拡張を使用可能にしてコンパイルした場合、複合式、条件式、およびキャストは、そのオペランドが左辺値であれば、左辺値とすることができます。この言語拡張は C++ コードには使用しないでください。

シーケンスの最後の式が左辺値である場合、複合式を割り当てることができます。次の式は同じです。

```
(x + 1, y) *= 42;  
x + 1, (y *=42);
```

アドレス演算子は、シーケンスの最後の式が左辺値である場合に、複合式で適用することができます。次の式は同じです。

```
&(x + 1, y);  
x + 1, &y;
```



条件式は、その型がボイドでなく、その true と false の 2 つの分岐がともに有効な左辺値である場合に、有効な左辺値とすることができます。キャストは、オペランドが左辺値である場合に、有効な左辺値となります。主な制約事項は、左辺値キャストのアドレスを取得することができないことです。

関連参照

- 160 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)
- 括弧で囲んだ式
-  **this** ポインタ
-  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、名前の型によって異なり、名前の型はその名前の宣言方法により決まります。次の表は、名前が左辺値式であるかどうかを示します。

1 次式: 名前

名前の宣言	評価対象	左辺値である
算術、ポインタ、列挙、構造体、または共用体型の変数	その型のオブジェクト	左辺値
列挙型定数	関連した整数値	左辺値ではない
配列	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 (sizeof 演算子への引き数として名前が使用される場合を除く)。	 左辺値ではない
関数	その関数。変換の対象となるコンテキストでは、その関数へのポインタ (sizeof 演算子への引き数、または関数呼び出し式内の関数として名前が使用される場合を除く)。	 左辺値ではない  左辺値

名前は、式として、ラベル、typedef 名、構造体コンポーネント名、共用体コンポーネント名、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。式内で名前によって参照される名前は、これらの目的のための名前とは分離したネーム・スペースに常駐します。これらの名前には、特殊な構成を指標することによって式内で参照可能なものもあります。例えば、ドットまたは矢印演算子を使用して構造体および共用体コンポーネント名を参照することができ、typedef 名をキャストで、または sizeof 演算子への引き数として使用することができます。

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。リテラルを式として評価する場合、その値は定数です。字句定数を左辺値とすることはできません。ただし、ストリング・リテラルは左辺値です。

関連参照

- 24 ページの『リテラル』
- 282 ページの『this ポインタ』

ID 式

C++ 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 ID または非修飾 ID とすることができます。また、ドットおよび矢印演算子の後で使用することもできます。

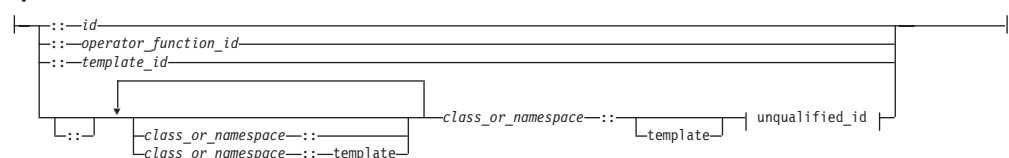
構文 - *id-expression*



unqualified_id:



qualified_id:



関連参照

- 20 ページの『ID』
- 87 ページの『第 4 章 宣言子』

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)

C 可変長配列型に適用された **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。

- 添え字宣言子の中で、バインド済み配列の説明として。
- **switch** 文のキーワード **case** の後。

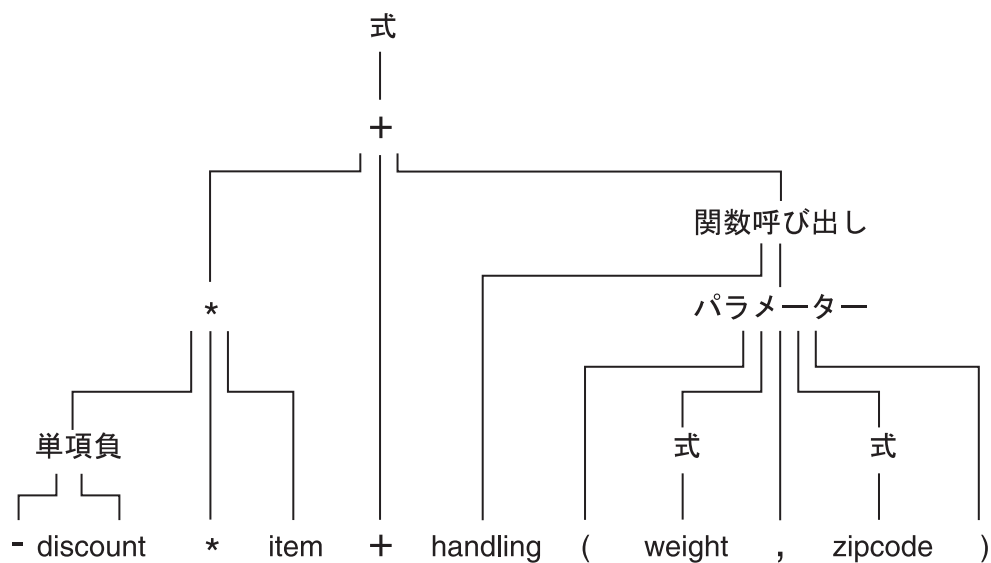
- 列挙子の中で、enum 定数の数値として。
- ビット・フィールド幅の指定子の中で
- プリプロセッサの #if 文の中で。(列挙型定数、アドレス定数、および sizeof は、プリプロセッサの #if 文では指定できません。)

関連参照

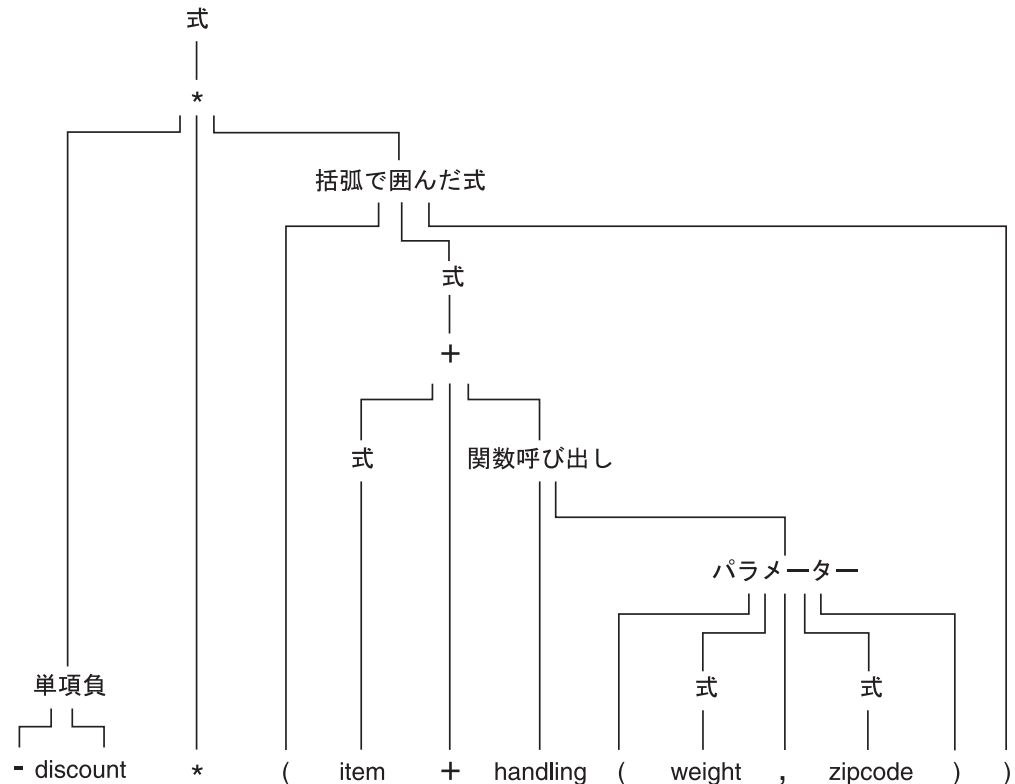
- 131 ページの『sizeof 演算子』

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。次の式では、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。



次の式は前の式と似ていますが、この式には、オペランドと演算子のグループ化の方法を変更する括弧が含まれています。



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);
```

C++ スコープ・レゾリューション演算子 ::

C++ `::` (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、名前・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。次に例を示します。

```
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・名前・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・名前・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
}
```



関連参照

- 269 ページの『クラス名のスコープ』
- 241 ページの『第 10 章 ネーム・スペース』

後置式

後置演算子 は、オペランドの後に表示される演算子です。後置式 は、1 次式表現、または後置演算子を含んでいる 1 次式です。以下に、使用可能な後置演算子を要約します。

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
2		メンバー選択	<i>object . member</i>
2		メンバー選択	<i>pointer -> member</i>
2		添え字	<i>pointer [expr]</i>
2		関数呼び出し	<i>expr (expr_list)</i>
2		値生成	<i>type (expr_list)</i>
2		後置増分	<i>lvalue ++</i>
2		後置減分	<i>lvalue --</i>
2		複合リテラル	<i>(type-name) {initializer-list}</i>
2	はい	 型の識別	typeid (<i>type</i>)
2	はい	 実行時の型識別	typeid (<i>expr</i>)
2	はい	 コンパイル時にチェックされる型変換	static_cast < <i>type</i> > (<i>expr</i>)
2	はい	 実行時にチェックされる型変換	dynamic_cast < <i>type</i> > (<i>expr</i>)
2	はい	 チェックなしの変換	reinterpret_cast < <i>type</i> > (<i>expr</i>)
2	はい	 const の変換	const_cast < <i>type</i> > (<i>expr</i>)

関数呼び出し演算子 ()

関数呼び出しは、単純型の名前と括弧で囲まれた引き数リストを含む式です。引き数リストには、コンマで分離した式をいくつでも入れることができます。引き数リストは、空にすることもできます。

次に例を示します。

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

関数呼び出しには 2 種類あります。すなわち、通常関数呼び出しと C++ メンバー関数呼び出しです。関数 `main` を除き、すべての関数は、それ自身を呼び出すことができます。

関数呼び出しの型

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。関数呼び出しは、関数の型が参照である場合に限り、左辺値です。

引き数およびパラメーター

関数引き数は、関数呼び出しの括弧内で使用する式です。関数仮パラメーターは、関数宣言または定義の括弧内で宣言された、オブジェクトまたは参照です。関数を呼び出すとき、引き数が評価されます。そして各パラメーターが、対応する引き数の値を使用して初期化されます。引き数受け渡しのセマンティクスは、代入のセマンティクスと同じです。

関数は、その非 `const` パラメーターの値を変更できます。ただし、これらの変更は、パラメーターが参照型でない限り、引き数には影響を及ぼしません。

リンケージおよび関数呼び出し

C C では、関数定義に外部結合と **int** 型の戻りの型がある場合は、`extern int func();` という暗黙の宣言が想定されるので、関数が明示的に宣言される前にその関数を呼び出すことができます。これは、C++ の場合には、あてはまりません。

引き数の型変換

配列または関数である引き数は、関数引き数として渡される前に、ポインターに変換されます。

非プロトタイプ C 関数に渡される引き数は型変換されます。型 **short** または **char** パラメーターは、**int** に変換され、**float** パラメーターは、**double** に変換されます。他の型変換の場合は、キャスト式を使用します。

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 `func` が呼び出されると、引き数 `f` は、**double** に、引き数 `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;
}
```

引き数の評価順序

引き数が評価される順序は、規定されていません。次のような呼び出しは、行わないようにしてください。

```
method(sample1, batch.process--, batch.process);
```

この例では、`batch.process--` が最後に評価されることもあり、最後の 2 つの引き数が同じ値で渡されることが生じます。

関数呼び出しの例

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。呼び出し先関数 `func` は、変数自身ではなく、`x` と `y` の値のコピーだけを受け取ります。

```
/**
 ** This example illustrates function calls
 **/

#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
```

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は、添え字 0 を取ります。

定義上、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]` にも等価です。式 `a` と `b` の間で、一方は、型 `T` に対するポインターで

左辺値 (lvalue)


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いければなりません。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>

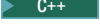
int main(void) {
    int a[3] = { 10, 20, 30 };
    printf("a[0] = %d\n", a[0]);
    printf("a[1] = %d\n", 1[a]);
    printf("a[2] = %d\n", *(2 + a));
    return 0;
}
```

次に、上記の例の出力を示します。

```
a[0] = 10
a[1] = 20
a[2] = 30
```

 C99 では、左辺値ではない配列での配列添え字が可能です。ただし、非左辺値のアドレスを配列添え字として使用することはできません。以下の例は C99 では有効ですが、C89 では無効です。

```
struct trio{int a[3];};
struct trio f();
foo (int index)
{
    return f().a[index];
}
```

 添え字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、添え字演算子とポインター演算の関係も、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、添え字 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、最も頻繁に右端添え字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;
        }
    }
}
```

ドット演算子 .

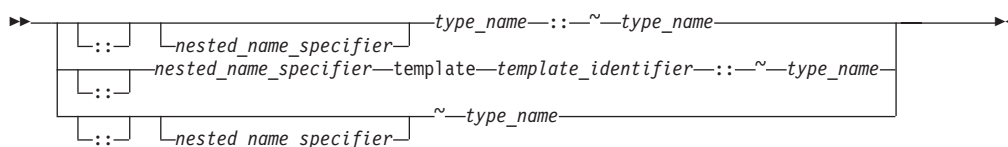
.(ドット) 演算子は、クラス、構造体、または共用体メンバーにアクセスするために使われます。メンバーは、後置式によって指定され、その後に .(ドット) 演算

子、さらにその後に、おそらく修飾された ID または疑似デストラクター名が続きます。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。

疑似デストラクター

▶ **C++** 疑似デストラクター は、以下の構文図における *type_name* という名前の非クラス型のデストラクターです。



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。

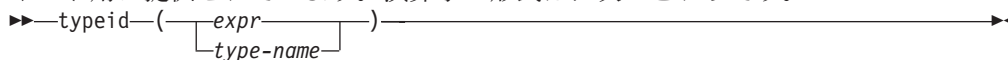
式の値は、選択されたメンバーの値です。名前が左辺値の場合は、式の値も左辺値です。式が修飾された型へのポインターである場合、結果式内の指定メンバーには同じ型修飾子が適用されます。

関連参照

- 118 ページの『ドット演算子 .』

typeid 演算子

▶ **C++** **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。演算子の形式は、次のとおりです。



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** の左辺値を返します。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。

左辺値 (lvalue)

expr が、ポリモフィック・クラスへの参照または間接参照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** オブジェクトを返します。ポリモフィック・クラスでない場合、**typeid** は、参照の型または間接参照ポインターを表す **type_info** オブジェクトを返します。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返します。

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** 修飾子を見捨てます。

static_cast 演算子

 **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。

構文 - reinterpret_cast

►► reinterpret_cast <Type> (—expression—) ◀◀

reinterpret_cast 演算子は、引き数と同じビット・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想定すると、以下の 2 つの変換は同義です。

- reinterpret_cast<T*>(x)
- *reinterpret_cast<T*>(&x)

ISO 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、構文は異なるけれども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、静かに修正します。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。プラットフォームによっては、パフォーマンスの最適化が、ISO 別名割り当て規則へ厳格に準拠して行われます。 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、最適化せずにコンパイルすると、期待通りに実行されます。また、C++ で、これを最適化して正常にコンパイルできますが、ただし、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ため、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。

関連参照

- 160 ページの『標準の型変換』
- 345 ページの『ユーザー定義の型変換』

const_cast 演算子

 `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。

構文 - `const_cast`

►—`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、結果は未定義です。

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。

関連参照

- 79 ページの『型修飾子』

dynamic_cast 演算子

 **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。これにより、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 (run-time 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。次の例が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 : 12FF6C
Address of bobj: 12FF6C
```

dynamic_cast 演算子の主目的は、タイプ・セーフな *downcasts* を実行することです。downcast は、クラス A を指すポインターまたは参照を、クラス B を指すポインターまたは参照に変換します。このクラス A は、B の基底クラスです。downcast には、型 A* のポインターが A から派生した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という問題があります。**dynamic_cast** 演算子は、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、A が指すオブジェクトが、クラス B または B から派生したクラスに属することを保証します。

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() {
```


左辺値 (lvalue)

```
A aobj;
C cobj;
A* ap = &cobj;
A* ap2 = &aobj;
f(ap);
f(ap2);
}
```

次に、上記の例の出力を示します。

```
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フィックです。**dynamic_cast** 演算子は、ポリモフィック・クラスから生成された実行時の型情報を使用します。

関連参照

- 301 ページの『派生』
- 345 ページの『ユーザー定義の型変換』


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。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。このため、単項式は後置式です。





次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。


次の表に、単項式の演算子が要約されています。

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 (<i>expr</i>)
3	はい	型のサイズ (バイト)	sizeof <i>type</i>
3	はい	接頭部増分	++ <i>lvalue</i>
3	はい	接頭部減分	-- <i>lvalue</i>
3	はい	補数	~ <i>expr</i>
3	はい	否定	! <i>expr</i>
3	はい	単項負	- <i>expr</i>
3	はい	単項正	+ <i>expr</i>
3	はい	～のアドレス	& <i>lvalue</i>
3	はい	間接または間接参照	* <i>expr</i>
3	はい	 作成 (メモリーの割り振り)	new <i>type</i>

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	 作成 (メモリーの割り振り と初期化)	<code>new type (expr_list) type</code>
3	はい	 作成 (配置)	<code>new type (expr_list) type (expr_list)</code>
3	はい	 破棄 (メモリーの割り振り 解除)	<code>delete pointer</code>
3	はい	 配列の破棄	<code>delete [] pointer</code>
3	はい	型変換 (キャスト)	<code>(type) expr</code>

 C99 では単項演算子 `_Pragma` が追加され、これにより、プリプロセッサ・マクロは `pragma` ディレクティブを含むことができます。この演算子は、IBM C++ により直交言語拡張としてサポートされています。

XL C/C++ は、C99 および C++ 規格を拡張し、単項演算子 `__real__` および `__imag__` をサポートします。これらの演算子は、複素数型の実数部および虚数部を抽出する機能を提供します。これらの拡張機能は、GNU C および C++ で開発されたアプリケーションの移植を容易にするためにインプリメントされました。

関連参照

- 29 ページの『複素数リテラル』

増分 ++

++ (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 --

-- (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 +

+ (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 -

- (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 !

! (論理否定) 演算子は、オペランドが、0 (false) またはゼロ以外 (true) のいずれかに評価されるかを決めます。

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 (true) 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 (false) になります。

C++ オペランドの評価の結果が false (0) になる場合は、式の値は **true** になり、オペランドの評価の結果が true (ゼロ以外) の値になる場合は、式の値は **false** になります。オペランドは、暗黙的に **bool** に変換されます。そして、結果の型は **bool** です。

次の 2 つの式は、同じです。

```
!right;
right == 0;
```

ビット単位否定 ~

~ (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。

```
00000000000000101
```

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、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。

~0 の 16 ビットの 2 進表示は、次のとおりです。

```
1111111111111111
```

アドレス &

& (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、関数指定機能、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;
```

 このセクションでのここから先の説明は、C++ だけに適用されます。

アンパーサンド記号 `&` は、C++ では、アドレス演算子としてばかりでなく、参照宣言子としても使用されます。これらの意味は関連性がありますが、同じではありません。

```
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
```

これは、参照のアドレスを取得した場合にそのターゲットのアドレスを戻します。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。

ラベルのアドレスは、GNU C アドレス演算子 `&&` を使用して取得することができます。これによりラベルを値として使用することができます。

関連参照

- 89 ページの『ポインター』
- 103 ページの『参照』

間接 *

* (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ることはできません。オペランドがオブジェクトを指す場合、この演算は、そのオブジェクトを参照している左辺値を生成します。オペランドが関数を指す場合、結果は C または C++ での関数指定子、オペランドが指す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。

関連参照

- 89 ページの『ポインター』

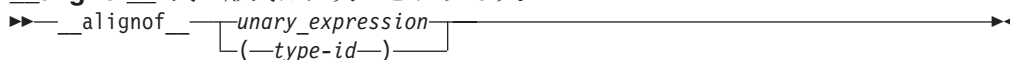
alignof 演算子

__alignof__ 演算子は、オペランドの位置合わせに使用されたバイト数を返します。言語フィーチャーは、C89 および C99 に対して直交です。オペランドは式、または型の括弧付きの ID とすることができます。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 コンパイル時に決定されますが、式自体の評価は行われません。オペランドが型である場合、通常、ターゲット・プラットフォーム上の型に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 式の形式は、次のとおりです。



type-id が参照であるか、または参照されるタイプである場合、結果は参照されるタイプの位置合わせです。*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。

例えば AIX の場合、`__alignof__(wchar_t)` は、32 ビットのターゲットでは 2 を返し、64 ビットのターゲットでは 4 を返します。

関連参照

- 36 ページの『aligned 変数属性』

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** 式の形式は、次のとおりです。



いずれのタイプのオペランドについても結果は左辺値ではなく、定数整数値です。結果の型は、`stddef.h` ヘッダー・ファイルに定義された符号なし整数型 **size_t** です。

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。3 種類の **char** オブジェクト (**unsigned**、**signed**、またはプレーン) は、いずれもサイズがバイト 1 となります。オペランドが可変長配列型の場合は、そのオペランドが評価されます。 **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が、評価は行いません。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は、コンパイラーはこれらの型変換を考慮して型を判別します。

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。

型は、**sizeof** 式では定義できません。

以下の例では、コンパイラーは、コンパイル時にサイズを評価することができます。**sizeof** のオペランド (式) は評価されません。 b の値は、初期化からプログラム実行時の終了まで、整数定数 5 です。

```
#include <stdio.h>



int main(void){
    int b = 5;
    sizeof(b++);
    return 0;
}
```

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、ストレージ割り振り時、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** 演算子を使用します。次に例を示します。

```
sizeof(int);
```

sizeof 式の結果は、それが適用される型によって異なります。

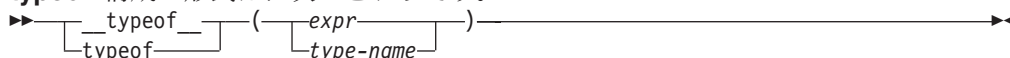
オペランド	結果
配列	結果は、配列内のバイトの合計数になります。例えば、10 のエレメントがある配列では、サイズは、単一のエレメントのサイズの 10 倍になります。コンパイラーは、式を評価する前には、配列をポインターに変換しません。
 クラス	結果は常に非ゼロで、そのクラスのオブジェクトのバイト数になります (配列にクラス・オブジェクトを配置するために必要な埋め込みを含む)。
 参照	結果は、参照されるオブジェクトのサイズになります。

typeof 演算子

typeof 演算子は、その引き数の型を戻します。これは、式または型とすることができます。言語フィーチャーによって、式から型を派生させることができます。代替スペルのキーワード、`__typeof__` が推奨されます。式 `e` が与えられると、`__typeof__ (e)` を、たとえば、宣言やキャストなど、型名が必要となるどの箇所でも使用できます。

typeof 演算子は、GNU C で開発されたプログラムの処理のために提供されている直交言語拡張機能です。

typeof 構成の形式は、次のとおりです。



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、その基本構文を示します。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。以下が与えられていると想定した場合、

```
int T[2];
int i[2];
```

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;
```

コードの振る舞いは、`int a[2];` を宣言した場合と同様になります。

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・プロパティは予約されていないため、`typeof(m) n;` の `n` は `int n` と同じですが、`int n:2` 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则便利です。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; }
```

ラベル値演算子 &&

ラベル値演算子 **&&** は、そのオペランドのアドレスを戻します。これは、現在の関数、または収容側関数内に定義されているラベルである必要があります。値は、型 **void*** の定数で、計算済みの `goto` 文でのみ使用する必要があります。言語フィーチャ

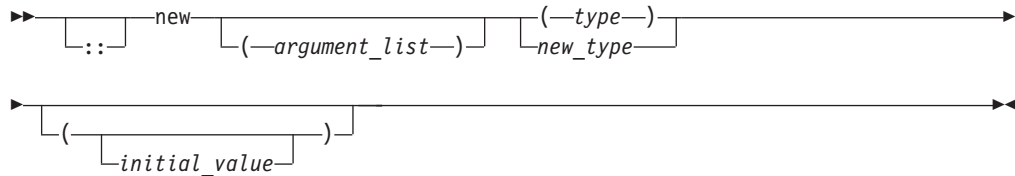
ャーは C および C++ に対する直交拡張で、GNU C で開発されたプログラムの移植を容易にするためにインプリメントされています。

関連参照

- 202 ページの『値としてのラベル』
- 220 ページの『計算済み goto』

C++ の new 演算子

C++ **new** 演算子は、動的ストレージ割り振りを提供します。 **new** 演算子を含む割り振り式の構文は、次のとおりです。



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 演算子を含む割り振り式は、作成されたオブジェクトのフリー・ストアの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返し、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。

set_new_handler() は、**new** が失敗したときに、それが何をするかを指定するためにだけ使用することができます。

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振られます。

```
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。

配置構文

追加引き数は、*argument_list* を使用することによって、**new** に引き数を追加することが できます (配置構文 と呼ばれます)。配置引き数を使う場合は、それらの引き数が指定された **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、`new` 式の配置引き数に指定されたロケーションのオブジェクト (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振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振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振られません。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` に割り振られたメモリーを再初期化します。

`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際に、`delete` を使用するべきではありません。メモリー・プール全体の削除だけが可能です (`delete whole`)。この例では、メモリー・バッファーは保持できますが、明示的にデストラクターを呼び出すことにより、そこに保管されているオブジェクトは破棄されます。

```
#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;
}
```

`new` 配置構文は、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場合にも使用できます。

関連参照

- 339 ページの『フリー・ストア』
- 137 ページの『`set_new_handler()` — `new` 障害のための振る舞いのセット』
- 138 ページの『C++ の `delete` 演算子』
- 114 ページの『C++ スコープ・レゾリューション演算子 `::`』
- 327 ページの『コンストラクターとデストラクターの概要』
- 39 ページの『オブジェクト』

`new` 演算子を使用して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();
```

クラスがデフォルトのコンストラクターを持っていない場合、そのクラスの任意のオブジェクトが割り振られるときには、**new** 初期化指定子が提供される必要があります。**new** 初期化指定子の引き数は、コンストラクターの引き数と一致している必要があります。


配列には、初期化指定子を指定できません。クラスにデフォルトのコンストラクターがある場合だけ、クラス・オブジェクトの配列を初期化できます。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。

関連参照

- 339 ページの『フリー・ストア』
- 327 ページの『コンストラクターとデストラクターの概要』

set_new_handler() — new 障害のための振る舞いのセット

 **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振りできないときには、**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。

- メモリ割り振りのためにより多くのストレージを入手し、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 **throw** します
- **abort()** または **exit()** のいずれかを呼び出します

set_new_handler() 関数には、以下のプロトタイプがあります。

```
typedef void(*PNH)();
PNH set_new_handler(PNH);
```

set_new_handler() は、引き数として関数 (**new** ハンドラー) を指すポインターを取ります。この関数には引き数がなく、**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振りできない場合に、**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振り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.
```

関連参照

- 134 ページの『C++ の new 演算子』
- 339 ページの『フリー・ストア』

C++ の delete 演算子

C++ **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振り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。この演算子の構文は、次のとおりです。

```
→ [::] 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振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振られたストレージを解放します。

この演算子の構文は、次のとおりです。

```
→ [::] delete [—] array →
```

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()` によって割り振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 配列に割り振られたストレージを解放することができます。

関連参照

- 339 ページの『フリー・ストア』
- 327 ページの『コンストラクターとデストラクターの概要』
- 58 ページの『void 型』

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。この演算子は、次のような形式です。ここで、*T* は型、*expr* は式です。

`(T) expr`

これは *expr* の値を、型 *T* に変換します。C の場合は、この操作の結果は左辺値ではありません。C++ では、*T* が参照である場合、この演算の結果は左辺値です。それ以外の場合は、結果は右辺値です。

 このセクションでのここから先の説明は、C++ だけに適用されます。

キャストは、そのオペランドが左辺値であれば、有効な左辺値です。次の単純な代入式では、まず最初に右側が指定の型に変換され、次に左側内部の式の型へと変換されて、結果が 保管されます。値は、指定された型に変換され、代入の値となります。次の例では、*i* の型は `char *` 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た複合代入演算の場合、複合代入の算術演算子は、キャストの結果の型を使用して実行され、単純な代入の場合と同様に進められます。次の式は同じです。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))
```

左辺値キャストのアドレスの取得は機能しません。その理由は、アドレス演算子をビット・フィールドに適用できないためです。

次の関数スタイルの表記を使用して、*expr* の値を型 *T* に変換することも可能です。

expr(*T*)

引き数を取らない関数スタイルのキャスト (*X*() など) は、宣言 *X t*() に等価です。ここで、*t* は、一時オブジェクトです。同様に、複数の引き数 (*X*(*a*, *b*) など) を持つ関数スタイルのキャストは、宣言 *X t*(*a*, *b*) に等価です。

C++ の場合は、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。

明示的型変換は、C++ 型変換演算子 **static_cast** を使用することによっても表現できます。

例

以下は、キャスト演算子の使用法を示したものです。この例では、サイズ 10 の整数配列を動的に作成します。


```
#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、その引き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、配列の初期の要素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。

共用体型へのキャスト

 共用体型へのキャストは、共用体メンバーを、それが所属する共用体と同じ型にキャストする機能です。そのようなキャストは、他のキャストとは異なり、左辺値を生成しません。このフィーチャーは C99 に対する直交拡張としてサポートされており、GNU C で開発されたプログラムの移植を容易にするためにインプリメントされています。

共用体型のメンバーとして明示的に存在する型だけが、その共用体型にキャストすることができます。キャストは、共用体型のタグ、または **typedef** 式で宣言された共用体型名のいずれかを使用することができます。指定される型は、完全な共用体

型でなければなりません。無名共用体型は、タグまたは型名を持つ場合、共用体型へのキャストで 사용할 ことができます。共用体に、同じ型のビット・フィールド・メンバーが含まれるが、必ずしも同じ長さではない場合、ビット・フィールドを共用体型にキャストすることができます。

ネストされた共用体へのキャストも認められています。以下の例では、**double** 型 `dd` は、ネストされた共用体 `u2_t` にキャストすることができます。

```
int main() {
    union u_t {
        char a;
        short b;
        int c;
        union u2_t {
            double d;
        }u2;
    };
    union u_t U;
    double dd = 1.234;
    U.u2 = (union u2_t) dd;    // Valid.
    printf("U.u2 is %f\n", U.u2);
}
```

この例の出力は次のようになります。

```
U.u2 is 1.234
```

共用体キャストは、関数引き数、初期化の定数式の一部として、また複合リテラル・ステートメントでも有効です。

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。

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。

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。

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。

次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。

次の表に、2 項式の演算子が要約されています。

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
5		乗算	<i>expr</i> * <i>expr</i>
5		除法	<i>expr</i> / <i>expr</i>
5		モジュロ (剰余)	<i>expr</i> % <i>expr</i>
6		2 項加算	<i>expr</i> + <i>expr</i>
6		2 項減算	<i>expr</i> - <i>expr</i>
7		左へのビット単位シフト	<i>expr</i> << <i>expr</i>
7		右へのビット単位シフト	<i>expr</i> >> <i>expr</i>
8		より小さい	<i>expr</i> < <i>expr</i>

2 項式

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
8		より小さいまたは等しい	<i>expr</i> <= <i>expr</i>
8		より大きい	<i>expr</i> > <i>expr</i>
8		より大きいまたは等しい	<i>expr</i> >= <i>expr</i>
9		等しい	<i>expr</i> == <i>expr</i>
9		等しくない	<i>expr</i> != <i>expr</i>
10		ビット単位 AND	<i>expr</i> & <i>expr</i>
11		ビット単位排他 OR	<i>expr</i> ^ <i>expr</i>
12		ビット単位包含 OR	<i>expr</i> <i>expr</i>
13		論理 AND	<i>expr</i> && <i>expr</i>
14		論理包含 OR	<i>expr</i> <i>expr</i>
16	はい	単純代入	<i>lvalue</i> = <i>expr</i>
16	はい	乗算および代入	<i>lvalue</i> *= <i>expr</i>
16	はい	除法および代入	<i>lvalue</i> /= <i>expr</i>
16	はい	モジュロおよび代入	<i>lvalue</i> %= <i>expr</i>
16	はい	加算および代入	<i>lvalue</i> += <i>expr</i>
16	はい	減算および代入	<i>lvalue</i> -= <i>expr</i>
16	はい	左へのシフトおよび代入	<i>lvalue</i> <<= <i>expr</i>
16	はい	右へのシフトおよび代入	<i>lvalue</i> >>= <i>expr</i>
16	はい	ビット単位 AND および代入	<i>lvalue</i> &= <i>expr</i>
16	はい	ビット単位排他 OR および代入	<i>lvalue</i> ^= <i>expr</i>
16	はい	ビット単位包含 OR および代入	<i>lvalue</i> = <i>expr</i>
18		コンマ (順序付け)	<i>expr</i> , <i>expr</i>

関連参照

- 105 ページの『演算子優先順位と結合順序』
- 166 ページの『算術変換』

乗算 *

* (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 /

/ (除法) 演算子は、そのオペランドの商を生成します。どちらのオペランドも整数である場合、小数部 (剰余) は廃棄されます。小数部の破棄は、ゼロへの切り捨てと呼ばれることがあります。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 になる場合、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。

  両方のオペランドが負の場合、剰余の符号も負となります。


剰余 %

% (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 を生成します。結果は左辺値にはなりません。

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、結果は未定義です。いずれかのオペランドに負の値がある場合で、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。

 両方のオペランドが負の場合、剰余の符号も負となります。それ以外の場合、剰余の符号は商の符号と同じです。

加法 +

+ (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、オリジナルの要素から、添え字として扱われる整数値の量だけ相対位置変更された、配列の中の別の要素を参照します。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、結果は未定義です。配列の終了を過ぎた 1 つの要素に対するポインターを使用して、そのアドレスのメモリー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、以下のように、加算の後で ptr は 配列の 3 番目の要素を指します。

```
int array[5];
int *ptr;
ptr = array + 2;
```

減法 -

- (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の要素を指すポインターである場合は、その結果は、2 つのアドレスを分離するオブジェクトの数です。数は、型 `ptrdiff_t` で、ヘッダー・ファイル `stddef.h` で定義されます。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、振る舞いは未定義です。

ビット単位左シフトと右シフト << >>

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。
>>	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 < > <= >=

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。

C 結果の型は **int** で、指定された関係が **true** であれば値 1 を持ち、 **false** であれば 0 を持ちます。

C++ 結果の型は **bool** で、値 **true** または **false** を持ちます。

結果は、左辺値ではありません。

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 結果は、**int** 型を持ちます。

C++ 結果は、**bool** 型を持ちます。

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。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、より大きい添え字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

2 項式

```
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 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 == !=

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。

C 結果の型は **int** で、指定された関係が **true** であれば値 1 を持ち、**false** であれば 0 を持ちます。

C++ 結果の型は **bool** で、値 **true** または **false** を持ちます。

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。結果は、C では型 **int**、C++ では **bool** です。

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 **true** 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、**true** 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。

例えば、次のような場合です。

`if (x == 3)` `x` が 3 の場合 **true** (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、**true** になります。また、この式では、3 が `x` に代入されます。

関連参照

- 153 ページの『単純代入 =』

ビット単位 AND &

& (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。

<code>a</code> のビット・パターン	0000000001011100
<code>b</code> のビット・パターン	0000000000101110
<code>a & b</code> 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 **true** になります。

関連参照

- 149 ページの『論理 AND &&』

ビット単位排他 OR ^

ビット単位排他 OR 演算子 (EBCDIC では、^ 記号は ~ 記号で表します) 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

関連参照

- 18 ページの『3 文字表記』

ビット単位包含 OR |

| (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 true になります。

関連参照

- 18 ページの『3 文字表記』

- 『論理 OR ||』

論理 AND &&

&& (論理 AND) 演算子は、両方のオペランドが **true** であるかどうかを示します。

C 両方のオペランド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++ 両方のオペランドの値が **true** である場合は、その結果の値は、**true** になります。そうでない場合は、結果の値は **false** 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 **false**) 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	false または 0
1 && 4	true または 1
0 && 0	false 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 **false**) 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。次に例を示します。

1 && 4 は 1 (または **true**) になります。
一方、
1 & 4 は 0 になります。

関連参照

- 147 ページの『ビット単位 AND &』

論理 OR ||

|| (論理 OR) 演算子は、いずれかのオペランドが **true** 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 **true** である場合は、その結果の値は、**true** になります。そうでない場合は、結果の値は **false** 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 **true**) 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	true または 1
1 4	true または 1
0 0	false 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 **true**) 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。次に例を示します。

1 || 4 は 1 (または **true**) になります。
一方、
1 | 4 は 5 になります。

関連参照

- 148 ページの『ビット単位包含 OR |』

メンバーを指す C++ ポインター演算子 (.* ->*)

▶ **C++** 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、->* 演算子も使用します。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。

関連参照

- 109 ページの『左辺値と右辺値』
- 281 ページの『メンバーへのポインター』

条件式

条件式 は、C++ で暗黙的に型 `bool` に変換される条件 ($operand_1$)、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た 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 `true` の場合は、第 2 オペランドが評価されます。
- 値が `false` 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。


条件式には、その第 1 および第 3 オペランドについて、右から左へ結合順序が付けられます。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。次の式は同じです。

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

C の条件式の型

C では、条件式は左辺値またはその結果ではありません。

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

 GNU C の場合、条件式は、その型が **void** でなく、分岐がともに有効な左辺値であれば、有効な左辺値となります。次の条件式 `(a ? b : c)` は、GNU C では有効となります。

```
(a ? b : c) = 5
/* Under GNU C, equivalent to (a ? b = 5 : (c = 5)) */
```

この式は、いずれかの拡張言語レベルにおけるコンパイルで使用可能です。

C++ の条件式の型

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。

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、結果はあいまいになります。
throw 式	その他 (型、ポインター、参照)	throw 式でない式の型

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 `false` 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);
```

代入式

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、単純代入と複合代入の 2 種類があります。

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。

代入式の結果は、C では左辺値ではありませんが、C++ では左辺値です。

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。

単純代入 `=`

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、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、コンパイラーは、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。

関連参照

- 109 ページの『左辺値と右辺値』
- 89 ページの『ポインター』
- 79 ページの『型修飾子』

複合代入

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 (変更可能な左辺値である必要がある) に保管します。

次の表では、複合代入式のオペランドの型を示します。

演算子	左方オペランド	右方オペランド
<code>+=</code> または <code>-=</code>	算術	算術
<code>+=</code> または <code>-=</code>	ポインター	整数型
<code>*=</code> 、 <code>/=</code> 、および <code>%=</code>	算術	算術
<code><<=</code> 、 <code>>>=</code> 、 <code>&=</code> 、 <code>^=</code> 、および <code> =</code>	整数型	整数型

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。

演算子	例	等価な式
<code>+=</code>	<code>index += 2</code>	<code>index = index + 2</code>
<code>-=</code>	<code>*(pointer++) -= 1</code>	<code>*pointer = *(pointer++) - 1</code>
<code>*=</code>	<code>bonus *= increase</code>	<code>bonus = bonus * increase</code>
<code>/=</code>	<code>time /= hours</code>	<code>time = time / hours</code>
<code>%=</code>	<code>allowance %= 1000</code>	<code>allowance = allowance % 1000</code>

演算子	例	等価な式
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
&=	mask &= 2	mask = mask & 2
^=	test ^= pre_test	test = test ^ pre_test
=	flag = ON	flag = flag ON

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)。ただし、左方オペランドがクラス型の場合、そのクラスは完成し、クラスのオブジェクトへの割り当ては、コピー代入演算と同様に行われます。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。

C GNU C C 言語フィーチャーが使用可能である場合、複合式および条件式は、そのオペランドが左辺値であれば、左辺値とすることができます。次の複合式 (a, b) の複合代入は、GNU C では有効となります (式 b、またはより一般的にシーケンスの最後の式が左辺値である場合)。

```
(a,b) += 5 /* Under GNU C, this is equivalent to
a, (b += 5) */
```

コンマ式

コンマ式 には、1 つのコンマによって分離される任意の型の 2 つのオペランドが含まれ、左から右への結合順序があります。左方オペランドは完全に評価されますが、副次作用が生じる可能性があり、値があれば、この値は廃棄されます。その上で、右方オペランドが評価されます。コンマ式の結果の型と値は、通常の単項型変換が行われた後のその右方オペランドの型と値になります。C の場合、コンマ式の結果は左辺値ではありません。C++ では、右方オペランドが左辺値の場合、結果は左辺値です。 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。

同様に、複合式のアドレスは、右方オペランドが左辺値の場合に取得が可能です。

```
&(a, b)
a, &b
```

コンマ演算子には結合順位があるため、コンマで分離された任意の数の式によって、単一式が形成されます。コンマ演算子を使用することによって、副次式が左から右に確実に評価され、最後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。

例えば、次の式

```
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

コンマ文字が使用されるコンテキストでは、あいまいさを避けるために括弧が必要になります。例えば、次の関数

```
f(a, (t = 3, t + 2), c);
```

の引き数は、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、構造体および共用体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。

先の例では、コンマを使用して関数呼び出しにおける引き数の式を分離しています。このコンテキストでは、これにより関数引き数の評価順序 (左から右) が保証されるわけではありません。

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
<pre>for (i=0; i<2; ++i, f());</pre>	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。
<pre>if (f(), ++i, i>1) { /* ... */ }</pre>	if 文では、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、その結果が if 文を処理するかどうかを判別します。
<pre>func(++a, f(a));</pre>	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、引き数が 1 つだけ渡されます。これは、関数引き数のリスト内で、コンマ式が括弧で囲まれているからです。

C++ の throw 式

▶ **C++** `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。

関連参照

- 387 ページの『第 17 章 例外処理』
- 58 ページの『void 型』

第 6 章 暗黙の型変換

与えられた型の式 `e` は、以下の状況のいずれかの場合で使用されると、暗黙的に変換されます。

- 式 `e` 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 `e` が、**if** 文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 `e` は、**bool** (C では **int**) に変換されます。
- 式 `e` が、**switch** 文の中で使用される。式 `e` は、整数型に変換されます。
- 式 `e` が、初期化で使用される。これには、次の場合が含まれます。
 - 代入が `e` とは異なる型を持つ左辺値に行われる。
 - 関数に、パラメーターとは異なる型を持つ `e` の引き数値が提供されている。
 - 式 `e` が、関数の **return** ステートメントに指定されていて、`e` が、関数の定義済み戻り型とは異なる型を持っている。

コンパイラーが以下のステートメントを許す場合にかぎって、コンパイラーは、式 `e` から型 `T` への暗黙的な変換を許します。


```
T var = e;
```

例えば、異なるデータ型の値を足す時、まずは両方の値を同じ型に変換します。**short int** の値と **int** の値を加算する場合、**short int** の値を **int** 型に変換します。

キャスト演算子、関数スタイル・キャスト、または C スタイル・キャストのうちのいずれかを使用して、明示的な型変換を行うことができます。

整数および浮動小数点拡張

整数拡張 とは、1 つの整数型を別の整数型へ変換することです。この場合、2 番目の型は、最初の型のすべての可能な値を保持することができます。整数が使用できるところではどこでも、ある種の基本型を使用することができます。整数拡張によって変換できる基本型は、以下のとおりです。

- **char**
-  **bool**
- **wchar_t**
- **short int**
- 列挙子
- 列挙型のオブジェクト
- 整数ビット・フィールド (符号付きおよび符号なしの両方)

wchar_t を除き、**int** で表せない値は、**unsigned int** に変換されます。**wchar_t** では、元の型のすべての値を **int** で表せる場合、その値は、元の型のすべての値を最も適切に表すことができる型に変換されます。例えば、**long** がすべての値を表すことができる場合、その値は、**long** に変換されます。

浮動小数点拡張

型 **float** の右辺値を、型 **double** の右辺値に変換できます。式の値は、変更されません。この変換は、浮動小数点拡張 です。

標準の型変換

多くの C および C++ 演算子によって、式の型を変更する暗黙の型変換 が行われます。異なるデータ型の値を足す時、まず、両方の値を共通の形式に変換します。例えば、**short int** の値と **int** の値を加算する場合、**short int** の値を **int** 型に変換します。オリジナルのオブジェクトの値が、より短い型によって表すことができる範囲を超えている場合、データの損失が生じます。

暗黙の型変換は、以下の場合に行われます。

- オペランドが算術演算または論理演算用に用意される。
- 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き数値が与えられる。
- 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。

C スタイル・キャスト、C++ 関数スタイル・キャスト、または C++ キャスト演算子の 1 つを使用して、明示的な型変換を行うことができます。

```
#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;
}
```

次に、上記の例の出力を示します。

```
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。

関連参照

- 345 ページの『ユーザー定義の型変換』

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。

型 T の左辺値 e は、T が関数型または配列型でなければ、右辺値に変換できます。変換後、e の型は T になります。これに対する例外を、次の表にリストしま

す。

変換前の状況	結果の振る舞い
T は不完全型である	コンパイル時エラー
e は、未初期化オブジェクトを参照している	未定義な振る舞い
e は、型 T でないオブジェクトを参照している	未定義な振る舞い
<div>C++</div> e は、型 T でないオブジェクト、または T から派生した型でないオブジェクトを参照している	未定義な振る舞い
<div>C++</div> T は非クラス型である	変換後の e の型は T で、 const または volatile によって修飾されていない

関連参照

- 109 ページの『左辺値と右辺値』

ブール変換

C

 任意のスカラー値を型 **_Bool** に変換すると、その結果のスカラー値が 0 に等しい場合は 0 になります。スカラー値が 0 以外の場合は 1 になります。

C++

 整数、浮動小数点、算術、列挙、ポインター、およびメンバー右辺値型を指すポインターを、型 **bool** の右辺値に変換できます。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、**false** に変換されます。他のすべての値は、**true** に変換されます。

次にブール変換ステートメントを示します。

```
void f(int* a, int b)
{
    bool d = a; // false if a == NULL
    bool e = b; // false if b == 0
}
```

a がヌル・ポインターに等しい場合、変数 d は **false** になります。そうでない場合には、d は **true** になります。b がゼロに等しい場合、変数 e は **false** になります。そうでない場合には、e は **true** になります。

関連参照

- 55 ページの『ブール変数』

整数変換

以下の変換を行うことができます。

- 整数型 (符号付きおよび符号なしの整数型を含む) の右辺値から、別の整数型の右辺値へ
- 列挙型の右辺値から、整数型の右辺値へ

整数 a を符号なし型へ変換する場合、結果値 x は、a および x が、モジュロ 2^n で合同であるような、最少の符号なし整数です。ここで n は、符号なし型を表

すのに使用されるビット数です。2 つの数 a および x が、モジュロ 2^n で一致する場合、次の式は `true` です。ここで、関数 `pow(m, n)` は、 m の n 乗の値を返します。

```
a % pow(2, n) == x % pow(2, n)
```

整数 a を符号付き型に変換する場合、コンパイラーは、新規の型が a を保持するのに十分に大きい場合、結果の値を変更しません。新規の型が十分には大きくない場合、その振る舞いは、コンパイラーでは、未定義です。

C++ `bool` を整数に変換する場合、`false` の値は `0` に変換され、`true` の値は `1` に変換されます。

整数拡張は、変換の別のカテゴリーに属します。整数変換ではありません。

関連参照

- 57 ページの『整数変換』

浮動小数点の型変換

浮動小数点型の右辺値を、別の浮動小数点型の右辺値に変換できます。

浮動小数点拡張 (`float` から `double` への変換) は、変換の別のカテゴリーに属します。浮動小数点変換ではありません。

関連参照

- 56 ページの『浮動小数点変数』
- 159 ページの『整数および浮動小数点拡張』

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 ポインターを含む型変換では、明示的な型キャストを使用する必要があります。ただし、`C` ポインターで許可されている割り当ての型変換の場合は、この規則の例外です。次の表の `const` で修飾された左辺値は、割り当ての左オペランドとしては使用できません。

表 1. `C` ポインターの正しい割り当て型変換

左オペランドの型	許可されている右オペランドの型
(オブジェクト) <code>T</code> へのポインター	定数 <code>0</code> <code>T</code> と互換性のある型へのポインター <code>void</code> へのポインター (<code>void*</code>)
(関数) <code>F</code> へのポインター	定数 <code>0</code> <code>F</code> と互換性のある関数へのポインター

左オペランドの型の修飾子は、右オペランドの修飾子と同じである必要があります。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 ポインターは型 **int** とサイズが同じである必要はありません。関数で想定している正しい型が渡されるようにするには、関数に与えられたポインターの引き数を明示的にキャストする必要があります。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。

void* への変換

オプションとして型が修飾された、型 **T** の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 左オペランドとして許可されている割り当ての型変換 (**void*** を含む) を次の表に示します。

表 2. C における **void*** の正しい割り当て型変換

左オペランドの型	許可されている右オペランドの型
(void*)	定数 0 (オブジェクト) T へのポインター (void*)

オブジェクト **T** は型が不完全な場合があります。

C++ 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。

派生から基底への変換

C++ 変換があいまいでない限り、**A** が **B** のアクセス可能な基底クラスであれば、型 **B*** の右辺値ポインターを、クラス **A*** の右辺値ポインターに変換できます。アクセス可能な基底クラスに対する式が、複数の別個のクラスを参照できる場合には、変換はあいまいなものになります。結果として得られる値は、派生クラス・オブジェクトの基底クラス・サブオブジェクトを指します。型 **B*** のポインターがヌルの場合、そのポインターは型 **A*** のヌル・ポインターに変換されます。基底クラスが、派生クラスの仮想基底クラスである場合、クラスを指すポインターを、その基底クラスを指すポインターに変換できないことに注意してください。

NULL ポインター定数

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。

配列からポインターへの変換

型「**N** の配列」(**N** は、配列の単一エレメントの型です) での左辺値または右辺値を、**N*** に変換できます。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の変換

型 `T` の関数である左辺値を、型 `T` 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は除きます。

参照変換

C++ 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 (引き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。

メンバーを指すポインターの型変換

C++ メンバーを指すポインターの型変換は、メンバーを指すポインターの型変換が初期化、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。

評価がゼロになる定数式は、メンバーへのヌル・ポインターに変換されます。

以下の条件が `true` 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り、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ます。その場合、その基底クラスは、アクセス可能 であるといえます。
- 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。

修飾変換

C++ 型 `cv1 T*` (ここで、`cv1` は、ゼロ以上の `const` または `volatile` の修飾の任意の組み合わせ) の右辺値を、型 `cv2 T*` の右辺値に変換できます。ただし、`cv2 T*` が `cv1 T*` よりも、多くの `const` または `volatile` で修飾されている場合です。

`cv1 T` のクラス `X` のメンバーを指す型ポインターの右辺値を、`cv2 T` のクラス `X` 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。ただし、`cv2 T` が、`cv1 T` よりも、多くの `const` または `volatile` で修飾されている場合に換換できます。

関連参照

- 79 ページの『型修飾子』

関数引き数変換

C 関数宣言が存在していて、宣言された引き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。関数を呼び出したとき、またはプロトタイプ引き

数リストの可変部の引き数として式が使用されているときに、関数宣言を参照できないと、コンパイラはデフォルトの引き数拡張を実行したり、式の値を変換してから、引き数を関数に渡します。自動型変換の構成は次のとおりです。

- 整数拡張
- **double** 型に変換される **float** 型の引き数

C GNU C セマンティクスを許可するコンパイラ・オプションを使用してコンパイルすると、関数プロトタイプによって以降の K&R の非プロトタイプ定義が上書きされることがあります。この振る舞いは ISO C では正しくありません。ISO C では、自動型変換のあと、関数の引き数型は関数のプロトタイプの型と一致している必要があります。

```
int func(char);      /* Legal in GCC, illegal in ISO C          */

int func(ch)         /* ch is automatically promoted to int,    */
char ch;             /* which does not match the prototype argument type char */
{ return ch == 0; }

int func(float);     /* Legal in GCC, illegal in ISO C          */

int func(ch)         /* ch is automatically promoted to double,    */
float ch;            /* which does not match the prototype argument type float */
{ return ch == 0; }
```

C++ C++ での関数宣言は、常にパラメーター型を指定する必要があります。また、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もあります。

関連参照

- 159 ページの『整数および浮動小数点拡張』
- 170 ページの『関数宣言』

その他の変換

void 型

定義上、**void** 型には、値がありません。したがって、他の型に変換できないし、代入によって、他の値を **void** に変換することもできません。ただし、明示的に値を **void** にキャストすることはできます。

構造体または共用体型

構造体型間または共用体型間の変換は、次の場合以外は実行できません。C では、右オペランドの型が左オペランドの型と互換性がある場合は、互換性のある構造体または共用体型間で割り当て型変換を実行できます。

表 3. C における構造体または共用体型の正しい割り当て型変換

左オペランドの型	許可されている右オペランドの型
C 構造体または共用体型	互換性のある構造体または共用体型

クラス型

▶ **C++** クラス型同士の間には標準の型変換はありませんが、クラス型用の独自の型変換オペレーターを記述することはできます。

列挙型

▶ **C** C では、**enum** 型指定子を使用して値を定義すると、その値は **int** として扱われます。**enum** 値への変換およびその値からの変換は、**int** 型に対する場合と同様に進められます。

enum から任意の整数型に変換することはできますが、整数型から **enum** に変換することはできません。

関連参照

- 58 ページの『void 型』
- 345 ページの『ユーザー定義の型変換』
- 73 ページの『列挙』

算術変換

変換は、個々の演算子およびオペランドの型によってそれぞれ異なります。ただし、整数型および浮動小数点型のオペランドについては、多くの演算子が同様の変換を行います。これらの標準型変換は、本来は算術で使用される値の型に適用されるため、**算術変換** と呼ばれます。

算術変換は、算術演算子のオペランドをマッチングするために使用されます。

算術変換は、以下の順序で進められます。


オペランド型	型変換
1 つのオペランドは long double 型です。	他のオペランドは long double に変換されます。
1 つのオペランドは double 型です。	他のオペランドは double に変換されます。
1 つのオペランドは float 型です。	他のオペランドは float に変換されます。
1 つのオペランドは unsigned long long int 型です。	他のオペランドは unsigned long long int に変換されます。
1 つのオペランドは long long 型です。	他のオペランドは long long に変換されます。
1 つのオペランドは unsigned long int 型です。	他のオペランドは unsigned long int に変換されます。
1 つのオペランドは unsigned int 型で他のオペランドは long int 型です。 unsigned int の値は long int で表すことができます。	unsigned int 型のオペランドは long int に変換されます。
1 つのオペランドは unsigned int 型で他のオペランドは long int 型です。 unsigned int の値は long int で表すことができません。	両方のオペランドとも unsigned long int に変換されます。
1 つのオペランドは long int 型です。	他のオペランドは long int に変換されます。

オペランド型	型変換
1 つのオペランドは unsigned int 型です。	他のオペランドは unsigned int に変換されます。
両方のオペランドとも int 型です。	結果は int 型になります。

関連参照

- 105 ページの『第 5 章 式と演算子』
- 57 ページの『整変数』
- 56 ページの『浮動小数点変数』

explicit キーワード

 引き数を 1 つだけ使用し、**explicit** キーワードを使用せずに宣言されたコンストラクターは、変換コンストラクターです。代入演算子を使用して、変換コンストラクターでオブジェクトを構成できます。 **explicit** キーワードを使用してこの型のコンストラクターを宣言すると、この振る舞いを防ぎます。明示的キーワードは、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。

explicit 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);
```


関連参照

- 347 ページの『コンストラクターによる変換』

第 7 章 関数

プログラム言語のコンテキストでは、関数 という用語は、出力を計算するために使用されるステートメントの集合を意味します。この言葉は、数学における場合ほど厳密に使用されているわけではありません。数学における関数は、入力変数を出力変数に一意的に 関連付けている集合を意味します。C または C++ のプログラムにおける関数の場合は、すべての入力に対して整合した出力が得られなかったり、出力がまったくなかったり、副次効果が生じたりすることがあります。関数は、パラメーター・リストのパラメーターがある場合、これがオペランドになる、ユーザー定義の操作と理解することができます。

関数は、ユーザーが作成した関数と、C 言語のインプリメンテーションで提供されている関数の 2 つのカテゴリに分類されます。後者はライブラリー関数 と呼ばれます。ライブラリー関数はコンパイラーが備えている関数のライブラリーに属しているからです。

関数の結果は、その関数の戻り値 と呼ばれます。戻り値のデータ型は、戻りの型 と呼ばれます。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。関数本体 という用語は、関数が実行する処理を表すステートメントのことをいいます。関数の本体は、中括弧で囲まれ、関数ブロック を構成します。関数の戻りの型の後には、関数の名前、パラメーター・リストが続き、本体とともに関数定義 を構成します。

関数名の後に関数呼び出し演算子 () が続くと、その関数が評価されます。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。これらの値は、そのパラメーターに対する引き数 で、今説明したプロセスのことを、関数に引き数を渡す といいます。

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。

関連参照

- 249 ページの『第 11 章 多重定義』

C 関数に対する C++ の拡張

C++ C++ 言語では、C 関数に対して多くの拡張を行っています。その内容は次のとおりです。

- 参照引き数
- デフォルト引き数
- 参照の戻りの型
- メンバー関数
- 多重定義関数
- 演算子関数

C 関数に対する C++ の拡張

- コンストラクター関数およびデストラクター関数
- 型変換関数
- 仮想関数
- 関数テンプレート
- 例外の指定
- コンストラクター初期化指定子

関連参照

- 190 ページの『参照による引き数の受け渡し』
- 192 ページの『C++ 関数におけるデフォルト引き数』
- 195 ページの『戻りの型としての参照の使用』
- 277 ページの『メンバー関数』
- 249 ページの『関数の多重定義』
- 251 ページの『演算子の多重定義』
- 327 ページの『コンストラクターとデストラクターの概要』
- 348 ページの『型変換関数』
- 318 ページの『仮想関数』

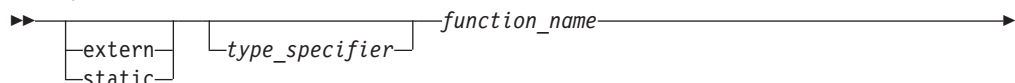
関数宣言

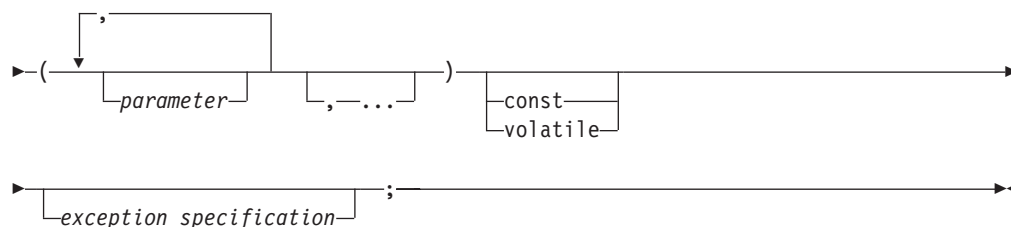
関数宣言 は、関数の名前およびそのパラメーターの数と型を明確にします。関数宣言は、戻りの型、名前、およびパラメーター・リストで構成されます。さらに、関数宣言はオプションで、その関数のリンケージを指定することができます。C++ では、宣言で、例外指定、const 修飾、または volatile 修飾を指定することもできます。

宣言は、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、関数の形式と存在を通知します。すべての宣言が一致する場合は、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。関数の暗黙宣言は許可されていません。すべての関数は、呼び出される前に明示的に宣言しておく必要があります。C89 では、明示的なプロトタイプなしで関数が呼び出される場合、コンパイラーは暗黙宣言を提供します。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ミスマッチを検査します。コンパイラーは、引き数の型の検査および引き数の変換のためにも関数宣言を使用します。

関数定義 には、関数宣言と関数本体が含まれます。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、関数定義はソース・ファイルに入れます。





関数引き数 は、関数呼び出しの括弧内で使用する式です。関数仮パラメーターは、関数宣言または定義の括弧内で宣言された、オブジェクトまたは参照です。関数を呼び出すとき、引き数が評価されます。そして各パラメーターが、対応する引き数の値を使用して初期化されます。引き数受け渡しのセマンティクスは、代入のセマンティクスと同じです。

宣言の中には、パラメーター・リスト内にパラメーターの名前がないものもあります。つまり、下記の例のように、宣言は単にパラメーターおよび戻り値の型を指定するだけのものがあります。これはプロトタイピング と呼ばれます。関数プロトタイプは、関数の戻りの型、関数の名前、およびパラメーター・リストで構成されます。次の例は、このことを示しています。

```
int func(int, long);
```

関数プロトタイプは、C と C++ との間の互換性を保持するのに必要とされます。空のパラメーター・リストを持つ関数の非プロトタイプ形式は、C ではその関数のパラメーター数が不明であることを意味しますが、C++ ではその関数はパラメーターを取らないことを意味します。

関数の戻りの型

関数は、配列の型および関数の型以外は、任意の型の値を戻すように定義できます。配列および関数呼び出しの結果を処理するには、その配列または関数へのポインターを戻す必要があります。関数は、関数を指すポインター、または配列の最初のエレメントを指すポインターを戻すことができます。しかし、配列または関数の型を持っている値を戻すことはできません。関数が値を戻さないように指示するには、戻りの型を **void** として関数を宣言します。

戻りのステートメントに式が含まれない場合、関数の戻りの型は、**void** にする必要があります。ただし、戻りのステートメントに式が含まれる場合、関数の戻りの型を **void** にすることはできません。これは、コンパイラーが戻り式を関数の戻りの型に代入するかのように変換するためです。

C++ 関数の戻りの型が **void** である場合や、関数がコンストラクターまたはデストラクターである場合、関数の戻りのステートメントには式は不要です。このような場合、関数は、値を戻しません。関数が値を戻す場合、戻りのステートメントには、式が含まれている必要があります。この式は、この式を含む関数の戻りの型に暗黙的に変換された後、関数の呼び出し元に戻されます。

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。

C++ での関数の宣言時の制限

すべての関数宣言では、戻りの型を指定する必要があります。

メンバー関数だけが、括弧で囲まれたパラメーター・リストの後に、**const** または **volatile** 指定子を持つことができます。

exception_specification は、関数が指定されたリストの例外だけを **throw** するよう制限します。

関数の宣言時のその他の制限

省略符号 (...) を C++ における唯一の引き数にすることもできます。この場合は、コンマは必要ありません。C では、唯一の引き数として省略符号を持つことはできません。

戻りまたは引き数の型の中で、型は定義できません。例えば、C++ コンパイラーでは、**print()** の以下のような宣言ができます。

```
struct X { int i; };  
void print(X x);
```

C コンパイラーでは、以下の宣言ができます。

```
struct X { int i; };  
void print(struct X x);
```

C コンパイラーおよび C++ コンパイラーのどちらも、同じ関数を次のように宣言することはできません。

```
void print(struct X { int i; } x); //error
```

この例は、クラス **X** のオブジェクト **x** を引き数として採用する関数 **print()** を宣言しようとしています。しかし、引き数リスト内では、クラス定義はできません。

別の例では、C++ コンパイラーでは、**counter()** の以下のような宣言ができます。

```
enum count {one, two, three};  
count counter();
```

同様に、C コンパイラーでは、以下の宣言ができます。

```
enum count {one, two, three};  
enum count counter();
```

C および C++ のどちらのコンパイラーでも、同じ関数を次のように宣言することはできません。

```
enum count{one, two, three} counter(); //error
```

counter() の宣言の例では、関数宣言の戻りの型に列挙型定義を入れることはできません。

関連参照

- 79 ページの『型修飾子』
- 399 ページの『例外の指定』

C++ 関数の宣言

▶ **C++** C++ では、メンバー関数宣言で修飾子の **volatile** および **const** を指定することができます。また、関数宣言で例外指定を指定することもできます。すべての C++ 関数は、呼び出される前に、宣言されている必要があります。

関連参照

- 79 ページの『型修飾子』
- 278 ページの『**const** および **volatile** メンバー関数』
- 399 ページの『例外の指定』

複数の関数宣言

▶ **C++** 1 つの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の型は、関数型の一部ですが、デフォルトの引き数と例外指定は、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。

引き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** 名または未指定の引き数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** の型指定子も関数型の一部ですが、宣言の一部、あるいは非静的メンバー関数の定義の一部でしかありません。

2 つの関数宣言が戻りの型とパラメーター・リストの両方で一致する場合、2 番目の宣言が最初の宣言の再宣言として処理されます。次の例は、同じ関数を宣言します。

```
int foo(const string &bar);
int foo(const string &);
```

戻りの型が違うだけの 2 つの関数を宣言することは、無効な関数の多重定義となり、コンパイル時のエラーのフラグが付けられます。次に例を示します。

```
void f();
int f();      // error, two definitions differ only in
              // return type
int g()
{
    return f();
}
```

関連参照

- 249 ページの『関数の多重定義』

関数宣言内のパラメーター名

▶ **C++** ユーザーが関数宣言でパラメーターの名前を付けることは可能ですが、次の 2 つの状態にある場合を除いて、コンパイラーはその名前を無視します。

1. 1 つの宣言内に同じ名前のパラメーターが 2 つある場合。これはエラーになります。
2. パラメーターの名前が、関数外の何かの名前と同じである場合。この場合、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こと、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);
```

関数属性

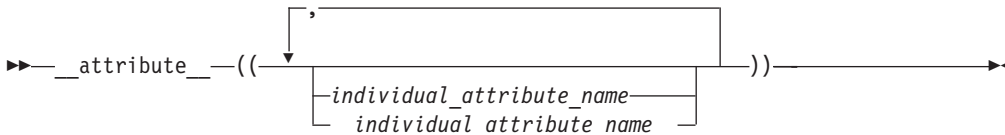
関数属性は直交拡張で、GNU C で開発されたプログラムの移植性を拡張するためにインプリメントされています。関数に対して指定可能な属性を使うと、コンパイラーによる関数呼び出しの最適化を補助し、コンパイラーに対してコードを多面的に検査するよう明示的に指示することができます。これ以外のものは、その他の機能性を提供します。

IBM の C および C++ は、GNU C の関数属性のサブセットをインプリメントしています。特定の関数属性がインプリメントされていない場合は、それを指定していても受け入れられますが、そのセマンティクスは無視されます。これらの言語フィーチャーは、どの拡張言語レベルでコンパイルする場合でも集合的に使用できます。

関数属性に対する IBM 言語拡張機能は、GNU C の構文を保持しています。

`__attribute__` という形式 (すなわち、前後に 2 つの下線文字が付いた関数属性キーワード) を使用した関数属性を指定すると、同じ名前のマクロと名前との競合の可能性が低くなります。

キーワード `__attribute__` は、属性指定子を導入します。関数属性の一部は、変数に適用することもできます。構文の一般的な形式は次のとおりです。



関数属性は宣言子に付加されます。

▶ **C** 関数プロトタイプ宣言に指定される属性の場合、宣言子にその属性を付すると、パラメーター・リストの右小括弧の後に配置するのと同じ効果があります。


```
/* 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 古いスタイルのパラメーター宣言の解析にはあいまいさがあるので、関数定義での属性指定は宣言子の前で行わなければなりません。例えば以下の `foo` の定義では、正しい配置が示されます。

```
int __attribute__((individual_attribute_name)) foo(int i) { }
int __attribute__((individual_attribute_name)) foo(i,j)
    int i; int j;
{ }
```

C++ C++ の場合、関数属性は、宣言または定義のどちらかの宣言子の後に置かれます。一般的な関数の場合、これは右小括弧の後でもあります。ただし、関数に例外指定が存在する場合には、関数属性をその指定の後に置く必要があります。

関連参照

- 36 ページの『変数属性』
- 50 ページの『型属性』

alias 関数属性

AIX **Linux** **z/OS** **alias** 関数属性を指定すると、関数宣言が別のシンボルの別名として、オブジェクト・ファイルに現れます。この言語フィーチャーを使用して、重複する名前、または複雑な名前に対処することができます。

alias 関数属性は関数属性の一般的な構文に従います。以下のダイアグラムで、サポートされる形式が示されます。

```

┌── __attribute__((alias(──"original_function_name"──)))──┐
│   └── alias ───┘
└── __alias__ ───┘
```

C この関数属性を持つ別名の指定の後に、別名を付けた関数を定義することができます。また、同じコンパイル単位に別名の割り当てられた関数の定義がない場合、C は別名の指定を許可します。

C++ `original_function_name` はマングルされた名前ではありません。

次の宣言は、`bar` を `__foo` の別名と宣言しています。

C

```
void __foo(){ /* function body */ }
void bar() __attribute__((alias("__foo")));
```

C++

```
extern "C" __foo(){ /* function body */ }
void bar() __attribute__((alias("__foo")));
```

このコンパイラーは `bar` の宣言と `__foo` の定義の間の整合性をチェックしません。そのような整合性を保持するのはプログラマーの責任です。

関連参照

- 179 ページの『weak 関数属性』

always_inline 関数属性

関数属性 `always_inline` は、コンパイル時に最適化が指定されたかどうかに関わらず、**inline** 関数をインラインするようにコンパイラーに指示します。ただし、プログラムが非最適化レベルでコンパイルされた場合、この属性に効果はありません。また、**inline** を指定せずに関数にこの属性を指定しても、有効ではありません。この属性は、インライン・コンパイラー・オプションよりも優先されます。この言語フィーチャーは、C89、C99、および Standard C++ および C++98 に対する直交拡張で、GNU C および C++ で開発されたプログラムの移植を容易にするためにインプリメントされています。

構文を以下の図で示します。

```

▶—__attribute__((always_inline)))————▶
                |
                |__always_inline__

```

const 関数属性

const 関数属性を指定すると、コンパイラーに対して、ソース・コードで示されているより少ない回数で関数を安全に呼び出せることを指示することができます。この言語フィーチャーは、関数とその引き数以外の値を検査しないこと、およびその戻り値以外には何の影響も及ぼさないことを示すことにより、コンパイラーによるコードの最適化を明示的に補助する手段をプログラマーに提供するものです。

const 関数属性は関数属性の一般的な構文に従います。

```

▶—__attribute__((const)))————▶
                |
                |__const__

```

次の種類の関数には、**const** を宣言しないでください。

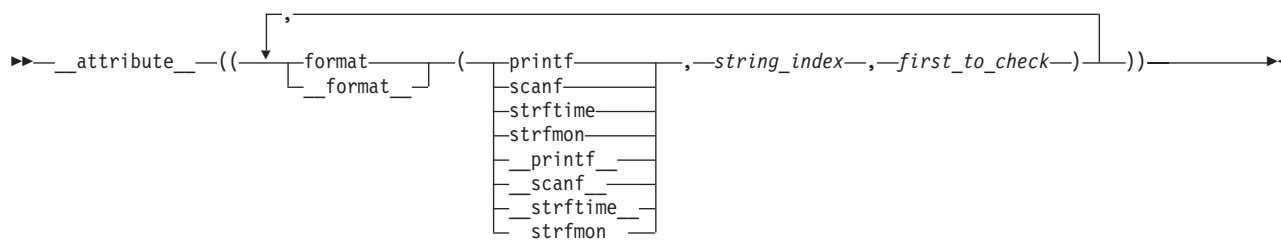
- ポインター引き数を持つ、ポイントされているデータを調べる関数。
- **const** 関数以外の関数を呼び出す関数。

「XL C/C++ コンパイラー・リファレンス」で、`#pragma isolated_call` についても参照してください。

format 関数属性

関数属性 `format` は、引き数などの書式制御ストリングをとるユーザー定義の関数を識別する手段を提供するものです。これにより、これらの関数の呼び出しにおいて、関数 `printf`、`scanf`、`strftime`、および `strfmon` に対する呼び出しでエラーについてコンパイラーが検査するのと同様の方法で、書式制御ストリングに関する型検査を行えるようにします。このフィーチャーは、C89、C99、および Standard C++ および C++98 に対する直交拡張で、GNU C および C++ で開発されたアプリケーションの移植を容易にするためにインプリメントされています。

構文を以下の図で示します。最初の引き数は、書式制御ストリングを解釈する方法のプロトタイプを示します。



ここで、

string_index

ユーザー関数の宣言内のどの引き数が書式制御ストリングの引き数であるかを指定する、定数の整数式です。C++ では、最初の引き数が暗黙の **this** 引き数であるため、非静的メンバー関数に対する *string_index* の最小値は、2 です。この振る舞いは、GNU C++ の振る舞いと整合します。

first_to_check

書式制御ストリングに対して検査する最初の引き数を指定する、定数の整数式です。書式制御ストリングに対して検査する引き数がない（つまり、書式ストリング構文およびセマンティクスで診断のみを実行する必要がある）場合、*first_to_check* には、値 0 を指定する必要があります。strftime スタイルの書式の場合は、*first_to_check* は 0 でなければなりません。

同じ関数で複数の *format* 属性を指定することが可能であり、その場合、すべてが適用されます。

```
void my_fn(const char* a, const char* b, ...)
    __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

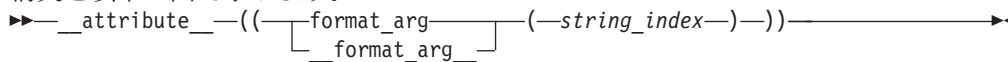
また、同じストリングで異なる書式スタイルについて診断することも可能です。すべてのスタイルが診断されます。

```
void my_fn(const char* a, const char* b, ...)
    __attribute__((__format__(__printf__,2,3),
                  __format__(__strftime__,2,0),
                  __format__(__scanf__,2,3)));
```

format_arg 関数属性

関数属性 *format_arg* は、書式制御ストリングを変更するユーザー定義の関数を識別する手段を提供します。関数が識別されると、そのオペランドがユーザー定義関数への呼び出しであるような関数 *printf*、*scanf*、*strftime*、または *strfmon* への呼び出しで、エラーを検査することができます。この言語フィーチャーは、C89、C99、および Standard C++ に対する直交拡張で、GNU C および C++ で開発されたプログラムの移植を容易にするためにインプリメントされています。

構文を以下の図で示します。



ここで、*string_index* は、どの引き数が書式制御ストリングであるかを指定する、1 から始まる定数の整数式です。C++ の非静的メンバー関数の場合、*string_index* は、最初のパラメーターが暗黙の **this** パラメーターであるため、2 から始まります。

同じ関数で複数の `format_arg` 属性を指定することが可能であり、その場合、すべてが適用されます。

```
extern char* my_dgettext(const char* my_format, const char* my_format2)
    __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%", "%"));
//printf-style format diagnostics are performed on both "%" strings
```

noinline 関数属性

関数属性 `noinline` は、適用される関数が、インラインまたは非インラインのいずれで宣言されているかにかかわらず、インラインされないようにします。この属性は、インライン・コンパイラー・オプション、**`inline`** キーワード、および `always_inline` 関数属性よりも優先されます。この言語フィーチャーは、C89、C99、および Standard C++ および C++98 に対する直交拡張で、GNU C および C++ で開発されたプログラムの移植を容易にするためにインプリメントされています。

構文を以下の図で示します。

```

▶▶ __attribute__((noinline)))
                    [noinline]

```

この属性は、インラインされないようにすること以外、インライン関数のセマンティクスは除去しません。

noreturn 関数属性

noreturn 関数属性を指定すると、関数が戻らないことをコンパイラーに示すことができます。プログラマーはこの言語フィーチャーを使用しても、明示的に、コンパイラーによるコードの最適化を補助し、未初期化の変数に対する誤った警告を削減することができます。

この関数の戻りの型は、**void** でなければなりません。

noreturn 関数属性は、関数属性の一般的な構文に従います。

►► `__attribute__((noretturn))` —

呼び出し側の関数によって保管されたレジスタは、非戻り関数を呼び出す前に必ずしも復元されるとは限りません。



「[XL C/C++ コンパイラー・リファレンス](#)」で、`#pragma leaves` についても参照してください。

pure 関数属性

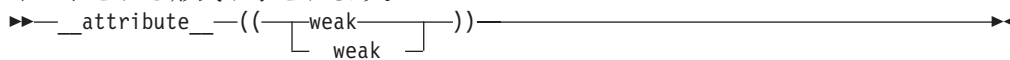
関数属性 **pure** を指定すると、ソース・コードで指定されているより少ない回数で呼び出すことのできる関数を宣言できます。属性 **pure** で関数を宣言すると、その関数は、パラメーター、グローバル変数、またはその両方にだけ依存する戻り値以外は何の影響も与えないことが示されます。この構文は **const** の場合と同じです。

「*XL C/C++ コンパイラー・リファレンス*」で、`#pragma isolated_call` についても参照してください。

weak 関数属性

  **weak** 関数属性を指定すると、関数宣言の結果得られるシンボルは、グローバル・シンボルではなく弱いシンボルとしてオブジェクト・ファイルに作成されます。**weak** 属性は、変数に適用することもできます。プログラマーはこの言語フィーチャーを使用すると、ユーザーが自作のコード内の関数定義をオーバーライドした場合の名前の重複エラーを回避するようにライブラリー関数を作成できます。

weak 関数属性は関数属性の一般的な構文に従います。以下のダイアグラムで、サポートされる形式が示されます。



通常、いくつかの再配置可能オブジェクト・ファイルを処理するときは、リンカーは同じ名前を持つグローバル・シンボルの多重定義を許可しません。ただし、同じ名前を持つグローバル・シンボルが存在している場合は、リンカーは弱い定義を許可します。この弱い定義は無視されます。グローバル・シンボルと弱いシンボルの間のもう 1 つの違いは、リンカーがアーカイブ・ライブラリーを検索するかどうかにあります。未定義のグローバル・シンボルを解決するために、リンカーはアーカイブ・ライブラリーを検索して、定義を含むメンバーを抽出します。未定義の弱いシンボルの解決にはこれを行いません。

弱いシンボルには、次の制約事項および制限が適用されます。

- 弱いシンボルは `static` ストレージ期間を持ってません。
- 弱いシンボルに対する多重定義を、同一の変換単位内で行うことはできません。多重定義が存在している場合は、リンカーは最初に検出された弱い定義を使用します。

関数宣言の例

次のコード・フラグメントは、複数の関数宣言を示しています。最初の部分は、2 つの整数の引き数を採用し、戻りの型が `void` である関数 `f` を宣言しています。

```
void f(int, int);
```

次のコードは、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 つの整数の引き数を取り、整数の引き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);
```

 このセクションでのここから先の説明は、C++ だけに適用されます。

関数宣言

次の宣言は、最初の引き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き数を持つことができ、型 **int** を戻します。

```
int extern f2(const int ...);
```

ただし、C では、省略符号の前にコンマが必要です。

```
int extern f2(const int, ...);
```

関数 `f3` は、戻りの型 **int** を持っています。そして、関数 `f2` から戻された値であるデフォルト値を持つ、**int** 引き数を取ります。

```
const int j = 5;
int f3( int x = f2(j) );
```

関数 `f6` は、クラス `X` の **const** クラス・メンバー関数で、引き数は取りません。**int** の戻りの型を持っています。

```
class X
{
public:
    int f6() const;
};
```

関数 `f4` は、引き数を取らず、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);
```

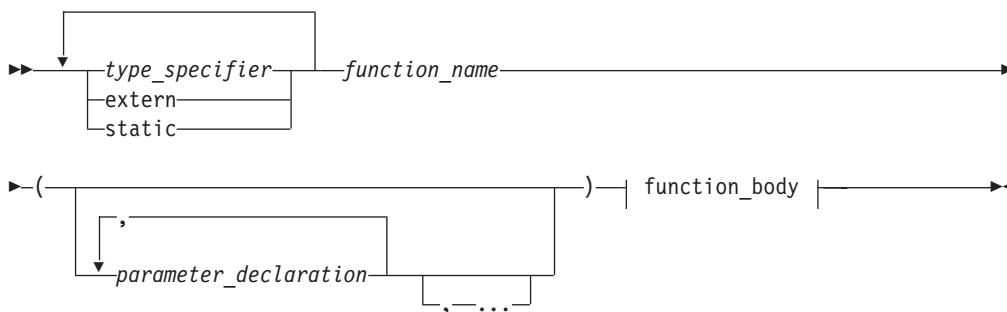
関数 `f5` は、引き数を取らないで、戻りの型 **void** を持っています。この関数は、任意の型の例外をスローする場合、`unexpected()` を呼び出します。

```
void f5() throw();
```

関数定義

関数定義 には、関数宣言と関数本体が含まれます。

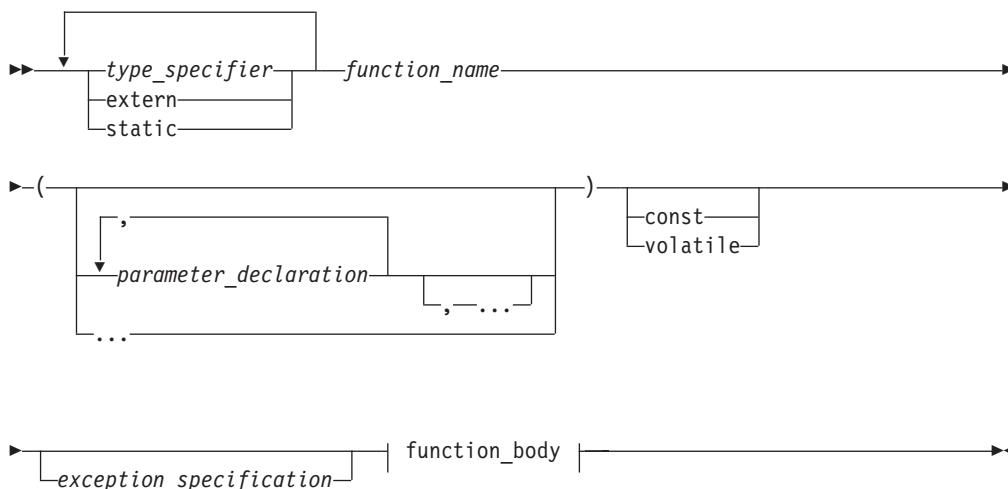
 C 関数定義の構文は、以下のとおりです。



function_body:

```
|—block_statement—|
```

▶ **C++** C++ 関数定義の構文は以下のとおりです。

**function_body:**

```
|—body—|
|—try—| body |—catch_handlers—|
```

body:

```
|—:—constructor_initializer_list—| block_statement—|
```

catch_handlers:

```
|—catch—(—parameter_declaration—)—block_statement—|
|—...—|
```

どちらの言語でも、関数定義には以下が含まれます。

- 少なくとも 1 つの型指定子。関数が返す値の型を決定します。例えば **unsigned long int** を返す関数の構文は、3 つの型指定子を使用します。
- オプションのストレージ・クラス指定子 **extern** または **static**。これは、その関数のスコープを決定します。ストレージ・クラス指定子が指定されない場合は、その関数には外部結合があります。
- 関数宣言子 は関数名であり、括弧で囲んだパラメーターの型と名前のリストが続きます。各パラメーターは関数が必要とするものです。以下の関数定義では、`f(int a, int b)` が関数宣言子です。


```
int f(int a, int b) {
    return a + b;
}
```
- ブロック・ステートメント。これには、データ定義とコードが含まれます。

▶ **C++** C++ 関数定義には、以下のものを任意に含めることができます。

- メンバー関数の関数宣言子の後の **const** または **volatile** 指定子。
- 例外指定。これは、関数が、指定されたリストの例外だけを throw するよう制限します。
- ブロック・ステートメント の代わりに 1 つ以上のキャッチ・ハンドラー を持つ *try* ブロック。
- 関数定義がコンストラクターを対象としているとき、ブロック・ステートメント の前のコンストラクター初期化指定子リスト。クラス A、*x(0)*、*y('c')* の以下の定義は、コンストラクター初期化指定子リストです。

```
class A {  
    int x;  
    char y;  
public:  
    A() : x(0), y('c') { }  
};
```

関数は、それ自体によって、または他の関数によって呼び出すことができます。デフォルトにより、関数定義は、外部結合を持っていて、他のファイルで定義された関数によっても呼び出すことができます。 **static** のストレージ・クラス指定子は、関数名はグローバル・スコープだけを持っていて、同じ変換単位内からのみ直接起動できるということを意味します。

▶ **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。

▶ **C** C では、関数定義に外部結合と **int** 型の戻りの型がある場合は、**extern int func();** という暗黙の宣言が想定されるので、関数が可視になる前にその関数を呼び出すことができます。C++ との互換性を保つためには、プロトタイプを使ってすべての関数を宣言する必要があります。

▶ **C** 関数が値を戻さない場合は、型指定子としてキーワード **void** を使用してください。関数が引き数をまったく取らない場合は、空のパラメーター・リストではなくキーワード **void** を使用して、関数に引き数が渡されないことを示してください。C では、空のパラメーター・リストを持つ関数は、受け取るパラメーター数が不明の関数を意味しますが、C++ では、その関数はパラメーターを取らないことを意味します。

▶ **C** C では、関数を構造体または共用体のメンバーとして宣言することはできません。

関数宣言の互換性

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。

関数型の互換性

C 型の互換性という考え方は、C のみに関係があります。2 つの関数型に互換性を持たせるには、戻りの型に互換性を持たせる必要があります。両方の関数型がプロトタイプなしで指定されている場合は、このことが唯一の要件になります。

プロトタイプ付きで宣言されている 2 つの関数の場合は、複合型は次の追加要件を満たす必要があります。

- 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。
- どちらの型もパラメーター・リストを持つ関数型の場合は、複合されるパラメーター・リスト内の各パラメーターは、対応するパラメーターの複合型です。

そして、宣言子指定子のシーケンスで `[*]` 表記を使用して、可変長配列型を指定することができます。

関数宣言子が関数定義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、その宣言子指定子のシーケンスで `[*]` 表記を使用して、可変長配列型を指定することもできます。互換性のある関数プロトタイプ宣言子の例を次に示します。

```
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返し、`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` の合計を返す 1 個のステートメントが入っています。

以下の例において、`ary` は、2 つの関数ポインターの配列です。互換性を保つためには、この例では `ary` に割り当てられた値に対して、型キャストが行われます。

```
#include <stdio.h>

typedef void (*ARYTYPE)();

int func1(void);
void func2(double a);

int main(void)
{
    double num = 333.3333;
    int retnum;
    ARYTYPE ary[2];
    ary[0]=(ARYTYPE)func1;
    ary[1]=(ARYTYPE)func2;

    retnum=((int (*)(void))ary[0])(); /* calls func1 */
    printf("number returned = %i\n", retnum);
    ((void (*)(double))ary[1])(num); /* calls func2 */
}
```

```
    return(0);
}

int func1(void)
{
    int number=3;
    return number;
}

void func2(double a)
{
    printf("result of func2 = %f\n", a);
}
```

次に、上記の例の出力を示します。

```
number
returned = 3
result of func2 = 333.333300
```

関連参照

- 42 ページの『`extern` ストレージ・クラス指定子』
- 45 ページの『`static` ストレージ・クラス指定子』
- 79 ページの『型修飾子』

省略符号および `void`

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されます。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。省略符号の前には、少なくとも 1 つのパラメーター宣言がなければなりません。

```
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 省略符号の前のコンマ、同じく省略符号の前のパラメーター宣言は、C では両方とも必須です。

パラメーター拡張は必要に応じて行われます。ただし、型検査は、可変引き数に対しては行われません。

引き数のない関数を、次の 2 つの方法で宣言することができます。

```
int f(void);
int f();
```

C++ 空の引き数宣言や `(void)` の引き数宣言リストは、引き数を使用しない関数を示します。

C 空の引き数宣言リストは、関数がパラメーターとして任意の数または型をとることができるということを意味します。

void から派生した型 (**void** へのポインターなど) は使用できますが、型 **void** を引き数型として使用することはできません。

次の例で、関数 `f()` は 1 つの整数引き数を使用して値は戻しませんが、`g()` は引き数がないことを予期しており、整数を戻します。

```
void f(int);
int g(void);
```

関数定義の例

以下の例には、**int** へのポインターとして宣言された `table` と、**int** 型として宣言された `length` が指定されている関数宣言子、`i_sort` が入っています。パラメーターとしての配列を、暗黙的にエレメント型へのポインターに変換することに注意してください。

```
/**
 ** This example illustrates function definitions.
 ** Note that arrays as parameters are implicitly
 ** converted to a pointer to the type.
 **/

#include <stdio.h>

void i_sort(int table[ ], int length);

int main(void)
{
    int table[ ]={1,5,8,4};
    int length=4;
    printf("length is %d\n",length);
    i_sort(table,length);
}

void i_sort(int table[ ], int length)
{
    int i, j, temp;

    for (i = 0; i < length -1; i++)
        for (j = i + 1; j < length; j++)
            if (table[i] > table[j])
            {
                temp = table[i];
                table[i] = table[j];
                table[j] = temp;
            }
}
```

次の例は、関数宣言 (また、関数プロトタイプ と呼ばれます) の例です。

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

次の例は、関数宣言子で **typedef** ID を使用する方法を示しています。

```
typedef struct tm_fmt { int minutes;
                        int hours;
                        char am_pm;
                    } struct_t;
long time_seconds(struct_t arrival)
```

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;
}
```

C C99 では、宣言内のパラメーターごとに少なくとも 1 つの型指定子が必要です。これにより、暗黙の **int** が宣言されているかのようにコンパイラーが振る舞う状況の数が少なくなります。C99 より前は、foo の宣言内の b または c の型があいまいであり、コンパイラーは両方に対して暗黙の **int** を想定しました。

```
int foo( char a, b, c )
{
    /* statements */
}
```

下位互換性のために、C99 の規則に違反しているように見える一部の構成も許可されます。例えば、次の foo の定義は、各パラメーターの型を明示的に宣言しています。

```
int foo( int a, char b, int c )
{
    /* statements */
}
```

ただし、次の定義は古い構文を使用していますが、b および c が関数本体で参照されていない場合に限り、等価なものとして受け入れられます。

```
int foo( int a, b, c )
{
    int a;
    /* okay if neither b nor c is used within the function */
}
```

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。どのプログラムにも、main という名前の関数が 1 つなければなりません。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 関数の形式は、次の 2 つのうちいずれかです。

C `int main (void) block_statement`

C++ `int main ()block_statement`

`int main (int argc, char ** argv)block_statement`

引き数 argc は、プログラムに渡されたコマンド行引き数の数です。引き数 argv は、ストリングの配列を指すポインターです。ここで、argv[0] は、コマンド行からプログラムを実行するために使用した名前です。argv[1] はプログラムに渡された最初の引き数、argv[2] は 2 番目の引き数、等々です。

デフォルトでは、main は、ストレージ・クラス **extern** を持ちます。

▶ **C++** `main` を、**inline** または **static** として宣言することはできません。プログラムの中から `main` を呼び出したり、`main` のアドレスを使用したりすることはできません。この関数は、多重定義することはできません。

main への引き数

関数 `main` は、パラメーターの指定ありでも、なしでも宣言することができます。

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

パラメーターにはどのような名前でも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。

最初のパラメーターの `argc` (引き数カウント) は、**int** 型で、コマンド行に入力する引き数の数を示します。

2 番目のパラメーターの `argv` (引き数ベクトル) の型は、**char** 配列オブジェクトへのポインターの配列型です。**char** 配列オブジェクトは、ヌル終了ストリングです。

`argc` の値は、配列 `argv` 内のポインターの数を示します。プログラム名が使用可能な場合、`argv` 内の最初の要素は、文字配列を指し、この配列には、プログラム名または実行しているプログラムの起動名が含まれています。名前が判別不可能な場合、`argv` 内の最初の要素はヌル文字を指します。

この名前は、関数 `main` への引き数の 1 つとしてカウントされます。例えば、プログラム名だけをコマンド行に入力すると、`argc` の値は 1 で、`argv[0]` は、そのプログラム名を指します。

コマンド行に入力された引き数の数に関係なく、`argv[argc]` には、常に `NULL` が入っています。

main への引き数の例

以下のプログラム `backward` は、コマンド行に入力された引き数を、最後の引き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]);
}
```

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
```

引き数 `argc` と `argv` には、以下の値が入ります。

オブジェクト	値
<code>argc</code>	3

オブジェクト	値
argv[0]	ストリング "backward" へのポインター
argv[1]	ストリング "string1" へのポインター
argv[2]	ストリング "string2" へのポインター
argv[3]	NULL

注: 環境によって大/小文字の区別をするものとししないものがあるため、コマンド行に大/小文字混合文字を入力する際は注意してください。また、argv[0] が指すストリングの正確な形式もシステムによって異なります。

関数呼び出しおよび引き数の受け渡し

関数呼び出しの引き数は、関数定義のパラメーターを初期化するために使用されます。引き数としての配列式および C の関数指定子は、呼び出しの前にポインターに変換されます。

最初に、整数拡張および浮動小数点拡張が、関数が呼び出される前に、引き数の値に対して行われます。

引き数の型は、関数宣言の対応するパラメーターの型に照らし合わせて検査されます。可変的に変更されるパラメーターのサイズ式は、関数に入る際に評価されます。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。各引き数式の値は、代入による場合のように、対応するパラメーターの型に変換されます。

次に例を示します。

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

/* Declaration */
extern double root(double, double);

/* Definition */
double root(double value, double base) {
    double temp = exp(log(value)/base);
    return temp;
}

int main(void) {
    int value = 144;
    int base = 2;
    printf("The root is: %f\n", root(value, base));
    return 0;
}
```

出力は、The root is: 12.000000 となります。

上記の例では、関数 root は **double** 型の引き数を予期していますので、value と base の 2 つの **int** 引き数は、この関数を呼び出すと、暗黙的に **double** 型に変換されます。

引き数が評価されて関数に渡される順序は、インプリメンテーションでの定義に依存します。例えば、以下の一連のステートメントで、関数 tester を呼び出します。


```
int x;
x = 1;
tester(x++, x);
```

この例のような `tester` の呼び出しは、別のコンパイラーでは、異なる結果を生む場合があります。インプリメンテーションによって、`x++` が最初に評価されるか、または `x` が最初に評価されるかが決まります。このようなあいまいさを避けて、`x++` が最初に評価されるようにするには、前述の一連のステートメントを次のように置き換えてください。

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

 このセクションでのここから先の説明は、C++ だけに適用されます。

非静的クラス・メンバー関数を引き数として渡すと、この引き数はメンバーへのポインターに変換されます。

関数仮パラメーターが可変長配列である場合、左端の次元以外の配列次元は、その関数の多重定義セットを区別します。

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。

関数引き数がクラス・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。

値による引き数の受け渡し

非参照パラメーターに対応する引き数を使用して関数を呼び出す場合、値によってその引き数を渡したことになります。パラメーターは、引き数の値を使用して初期化されます。関数のスコープ内のパラメーターの値を変更することができます (そのパラメーターが **const** と宣言されていないければ)。ただし、これらの変更は、呼び出方の関数の引き数の値には、影響しません。

以下は、値による引き数の受け渡しの例です。

次のステートメントは、関数 **printf** を呼び出します。この関数は、文字ストリングと、(a および b の値を受け取る) 関数 `sum` からの戻り値を受け取ります。

```
printf("sum = %d\n", sum(a,b));
```

以下のプログラムは、`count` の値を関数 `increment` に渡します。関数は、パラメーター `x` の値を 1 ずつ増やします。

```
/**
 ** An example of passing an argument to a function
 **/

#include <stdio.h>
```

関数呼び出しおよび引き数の受け渡し

```
void increment(int);

int main(void)
{
    int count = 5;

    /* value of count is passed to the function */
    increment(count);
    printf("count = %d\n", count);

    return(0);
}

void increment(int x)
{
    ++x;
    printf("x = %d\n", x);
}
```

出力は、main の count の値が未変更のままであることを示しています。

```
x = 6
count = 5
```

関連参照

- 116 ページの『関数呼び出し演算子 ()』

参照による引き数の受け渡し

参照による受け渡し は、呼び出す方の関数の引き数の値が、呼び出される関数の中で変更できる、引き数受け渡しのメソッドを指しています。

C 参照によって引き数を渡すには、対応するパラメーターをポインター型を使用して宣言します。

C++ C++ の参照により引き数を受け渡すためには、対応するパラメーターは、ポインター型だけではなく、任意の参照型を使用して構いません。

次の例は、参照によって引き数がどのように渡されるかを示しています。この関数を呼び出すと、参照パラメーターが実引き数によって初期化されることに注意してください。

```
#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;
}
```

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

実引き数の値を関数 `swapnum()` で変更させるには、`swapnum()` のパラメーターを参照として定義する必要があります。

 **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。

ポインター・パラメーターによって、非定数オブジェクトの値を変更することができます。次の例は、このことを示しています。

```
#include <stdio.h>

int main(void)
{
    void increment(int *x);
    int count = 5;

    /* address of count is passed to the function */
    increment(&count);
    printf("count = %d¥n", count);

    return(0);
}

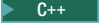
void increment(int *x)
{
    ++*x;
    printf("*x = %d¥n", *x);
}
```

次に、上記のコード出力を示します。

```
*x = 6
count = 6
```

例では、`count` のアドレスを `increment()` に渡します。関数 `increment()` は、ポインター・パラメーター `x` によって、`count` を増分します。

C++ 関数におけるデフォルト引き数

 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 です。

デフォルト引き数は、暗黙的に、パラメーター型に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、エラーになります。関数の型は、デフォルト値を持つ引き数によって影響を受けません。

以下の例は、デフォルト引き数が、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。デフォルト引き数を使用すると、すべての引き数を指定しなくても関数を呼び出すことができます。すべての引き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、明示的引き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、引き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.
```

関連参照

- 197 ページの『関数へのポインター』

デフォルト引き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き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。

デフォルト引き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き数を指定すると、その引き数を、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き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き数値でも与えることができます。デフォルト引き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き数値を持っていなければなりません。

デフォルト引き数の式では、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を生成します。

```
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
}
```

関連参照

- 116 ページの『関数呼び出し演算子 ()』
- 134 ページの『C++ の new 演算子』
- 192 ページの『C++ 関数におけるデフォルト引き数』

デフォルト引き数の評価

デフォルト引き数を使用して定義された関数が、末尾の引き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
```

デフォルト引き数は、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。デフォルトの引き数の評価の順序は、定義されていません。デフォルト引き数式は、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。

C++ 関数におけるデフォルト引き数

```
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、引き数 r を引き数 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き数 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、引き数 D により隠されます。したがって、D が引き数の名前であり、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 のオブジェクトが構築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ため、メンバー b をデフォルト引き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;
};
```

関連参照

- 192 ページの『C++ 関数におけるデフォルト引き数』

関数からの戻り値

関数の戻りの型が **void** でない限り、関数は値を戻す必要があります。

戻り値は **return** ステートメントで指定します。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、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。

関連参照

- 217 ページの『`return` ステートメント』
- 218 ページの『戻り式の値および関数値』
- 344 ページの『一時オブジェクト』

戻りの型としての参照の使用

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。これにより、関数呼び出しを代入ステートメントの左側に配置することができます。

▶ **C++** 参照された戻り値は、代入演算子および添え字演算子が多重定義され、多重定義された演算子の結果を実際の値として使用できるようになるときに、使用されます。

注: 自動変数に参照を戻すと、予測できない結果となります。

割り振り関数および割り振り解除関数

▶ **C++** 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。
- ユーザーの割り振り関数は、テンプレート関数にすることもできます。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できません。
- ユーザーが空の例外指定 `throw()` を使用してユーザーの割り振り関数を宣言する場合、ユーザーの割り振り関数は、ユーザーの関数が失敗したときに、ヌル・ポインターを戻すようにする必要があります。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。

関数からの戻り値

- 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。
- ユーザーの割り振り解除関数は、テンプレート関数であってもかまいません。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できません。

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 40 bytes
operator new with 33 bytes
operator new with 4 bytes
Constructor of S
Destructor of S
delete object
Array of size 2:
operator new with 16 bytes
Constructor of S
Constructor of S
Destructor of S
Destructor of S
delete object
```

関連参照

- 339 ページの『フリー・ストア』

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、関数を引き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` を引き数として取り、`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` 引き数を受け取り、`int` を戻す関数へのポインターと解釈します。

関連参照

- 162 ページの『ポインター型変換』

インライン関数


インライン関数とは、コンパイラーがメモリー内に独立した命令セットを作成するのではなく、関数定義のコードを呼び出し側の関数のコードに直接コピーする関数のことです。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。このようにして、関数呼び出しのパフォーマンス上のオーバーヘッドが回避されます。

inline 関数指定子を使用して、あるいはクラスまたは構造体定義の中でメンバー関数を定義して、関数をインラインとして宣言します。**inline** 指定子は、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。

以下のコードの断片では、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、関数をインライン展開すると、実引き数の評価の順序が保たれなくなる場合があります。また、インライン展開によって関数のリンケージが変更されることもありません。リンケージはデフォルトでは外部になっています。

 **C++** では、メンバー関数と非メンバー関数は、両方ともインラインにすることができます。クラス宣言の本体内部にインプリメントされたメンバー関数は、暗黙で **inline** を宣言されます。コンパイラーによって作成されるコンストラクター、コピー・コンストラクター、代入演算子、およびデストラクターも、暗黙で

`inline` を宣言されます。コンパイラーがインラインにしない `inline` 関数は、通常の関数と同じように処理されます。つまり、その関数が定義されている変換単位の数に関係なく、関数のコピーは 1 つしか存在しません。

C C では、内部結合を持つ関数はどの関数でもインラインにできますが、外部結合を持つ関数には制限が課せられています。この制限は以下のとおりです。

- **inline** キーワードが関数宣言で使用されている場合は、その関数はそれと同じ変換単位内で定義する必要があります。
- 関数のインライン定義 とは、同じ変換単位内のファイルのファイル・スコープ宣言がすべて、**extern** が指定されていない **inline** 指定子を含んでいる定義をいいます。
- インライン定義は、関数に外部定義を与えません。外部定義は別の変換単位内で行います。インライン定義は、同じ変換単位内から呼び出された場合は、外部定義の代替として機能します。C99 規格では、インライン定義が使用されるか外部定義が使用されるかは規定されていません。

C では、インライン定義は、対応する外部定義、および別の変換単位内の別の対応するインライン定義とは異なっています。

言語拡張できるようにソース・コードをコンパイルした場合、インライン関数の振る舞いは GNU C のセマンティクスに従います。関数定義に **extern inline** が明示的に指定されている場合は、コンパイラーはインライン化のためだけに **extern inline** 定義を使用します。その場合の振る舞いはマクロ展開に似ています。ある関数の **extern inline** 定義がヘッダー・ファイルにある場合、**extern** または **inline** が指定されていないその関数の外部定義を別のファイルから得るようにしなければなりません。この定義は、ヘッダー・ファイルをインクルードしていないファイルからその関数を呼び出す場合に使用されます。

次の例は、**extern inline** のセマンティクスを示したものです。GNU セマンティクスでコンパイルされると、`two()` に対するインラインではない関数本体は生成されません。

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

extern inline void two(void){ // GNU C uses this definition only for inlining
    printf("From inline.h\n");
}

main.c:
#include "inline.h"

int main(void){
    void (*pTwo)() = two;
    two();
    (*pTwo)();
}

two.c:
#include<stdio.h>

void two(){
    printf("In two.c\n");
}
```

以下の出力は、two への最初の関数呼び出しが実際にインラインにされた場合の結果を示したものです。


```
Using the gcc semantics for the inline keyword:
  From inline.h
  In two.c
```

コンパイラーは **inline** 関数指定子があっても、**extern inline** 関数 two をインラインにしない場合もあります。

関連参照

- 277 ページの『メンバー関数』
- 42 ページの『extern ストレージ・クラス指定子』

ネストされた関数

 ネストされた関数とは、別の関数の定義内で定義された関数のことです。これは変数宣言が許可される場所ならばどこでも定義することができるため、ネストされた関数をネストされた関数内に入れることができます。収容側関数内で、ネストされた関数は、**auto** キーワードを使用して定義する前に宣言できます。それ以外の場合、ネストされた関数は内部結合を持ちます。この言語フィーチャーは C89 および C99 に対する直交拡張で、GNU C で開発されたプログラムの移植を容易にするためにインプリメントされています。

ネストされた関数は、その定義の前にある収容側関数のすべての ID にアクセスすることができます。

制約および制限事項

ネストされた関数は、収容側関数が終了した後に呼び出してはなりません。


ネストされた関数は、**goto** 文を使用して、収容側関数のラベル、または収容側関数から継承された `__label__` キーワードで宣言されたローカル・ラベルにジャンプできません。

関連参照

- 202 ページの『ローカルに宣言されたラベル』

第 8 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
 - ID ラベル
 - **case** ラベル
 - **default** ラベル
- 式ステートメント
- ブロック・ステートメントまたは複合ステートメント
- 選択ステートメント
 - **if** 文
 - **switch** 文
- 繰り返しステートメント
 - **while** ステートメント
 - **do** ステートメント
 - **for** ステートメント
- 分岐ステートメント
 - **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** 文
- 宣言ステートメント
-  **try** ブロック


ラベル


ラベルには *identifier*、**case**、および **default** の 3 つの種類があります。

identifier ラベル・ステートメントの形式は、次のとおりです。

▶ *identifier* : *statement* ▶▶

ラベルは、*identifier* およびコロン (:) 文字から構成されます。

 ラベル名は、そのラベルが使用されている関数内で一意である必要があります。

 C++ では、*identifier* ラベルは **goto** 文のターゲットとしてのみ使用することができます。 **goto** 文では、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。

case および **default** ラベル・ステートメントは、**switch** 文でのみ使用されます。これらのラベルは、最も近い **switch** 文内でのみアクセス可能です。

case ステートメントの形式は、次のとおりです。

▶▶ `case constant_expression : statement` ▶▶

default ステートメントの形式は、次のとおりです。

▶▶ `default : statement` ▶▶

ラベルの例

```
comment_complete : ; /* null statement label */
test_for_null : if (NULL == pointer)
```

関連参照

- 219 ページの『goto 文』
- 207 ページの『switch 文』

ローカルに宣言されたラベル

ローカルに宣言されたラベル、すなわちローカル・ラベル は、ステートメント式の先頭で宣言され、スコープがそのラベルが宣言され定義されているステートメント式である identifier ラベルです。この言語フィーチャーは C および C++ に対する直交拡張で、GNU C で開発されたプログラムの処理を容易にします。

ローカル・ラベルは、**goto** 文のターゲットとして使用することができ、そのラベルが宣言されている ブロックと同じブロック内からそのターゲットにジャンプすることができます。この言語拡張は、ネストされたループを含むマクロを作成する場合に特に役立ち、そのステートメント・スコープと通常のラベルの関数スコープとの違いを利用します。

構文は以下のとおりです。

▶▶ `__label__ identifier ;` ▶▶



ステートメント式において、ローカル・ラベルの宣言は、左括弧および左中括弧の直後で、通常の宣言およびステートメントよりも前でなければなりません。ラベルは、ステートメント式のステートメント内で、名前とコロンの使用した一般的な方法で定義されます。

関連参照

- 201 ページの『ラベル』

値としてのラベル

現在の関数内、または収容側関数に定義されているラベルのアドレスは、**void*** 型の定数が有効な場合はいつでも、値として取得し使用することができます。このアドレスは、ラベルが単項演算子 **&&** のオペランドの場合は戻り値になります。ラベルのアドレスを値として使用する機能は、C99 および C++ に対する直交拡張機能で、GNU C で開発されたプログラムの移植を容易にするためにインプリメントされています。

以下の例で、計算済みの goto 文は label1 および label2 の値を使用して、関数内のこれらのスポットにジャンプします。


```

int main()
{
    void * ptr1, *ptr2;
    ...
    label1: ...
    ...
    label2: ...
    ...
    ptr1 = &label1;
    ptr2 = &label2;
    if (...) {
        goto *ptr1;
    } else {
        goto *ptr2;
    }
    ...
}

```

関連参照

- 133 ページの『ラベル値演算子 &&』
- 220 ページの『計算済み goto』

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形式は、次のとおりです。

```

┌──────────┐
│ 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。

式の例

```

printf("Account Number: ¥n");          /* call to the printf */
marks = dollars * exch_rate;             /* assignment to marks */
(difference < 0) ? ++losses : ++gain;     /* conditional increment */

```

関連参照

- 105 ページの『第 5 章 式と演算子』

C++ でのあいまいなステートメントの解決

▶ **C++** 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後に、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
```

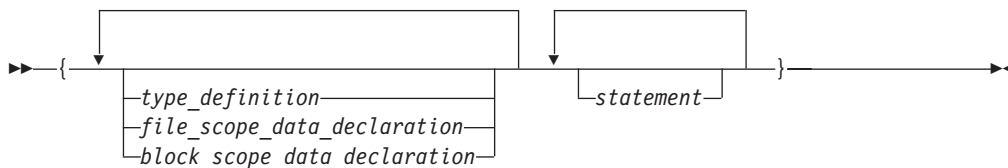
関連参照

- 35 ページの『第 3 章 宣言』
- 105 ページの『第 5 章 式と演算子』
- 116 ページの『関数呼び出し演算子 ()』

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まとめられます。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・ステートメントの形式は、次のとおりです。



C C89 言語レベルでは、定義および宣言はステートメントの前に配置しなければなりません。

C99 言語レベルの C および Standard C++ および C++98 では、宣言および定義を任意の位置に配置し、他のコードと混在させることができます。

関連参照

- 202 ページの『値としてのラベル』
- 133 ページの『typeof 演算子』

if 文

if 文は、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** 文を使用すると、指定されたテスト式 (暗黙的に **bool** に変換される) が **true** 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** 文を使用すると、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** 文で **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** 節がある場合、**else** 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** 節は無視されます。

if 文の形式は、次のとおりです。

```

▶▶ if (—expression—) —statement—
    |
    | else —statement—
    |
    ▶▶
    
```

if 文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** 文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** 文の後、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** 文内でのみ可視です。同じ規則が **if** 文の **else** 部分にも適用されます。

if 文の例

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` 文を示しています。

```
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` 文を示しています。 `else` 節は、常に、最も近い `if` 文に関連付けられるため、特定の `else` 節を強制的に正しい `if` 文に関連付けるには、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` 文に関連付けられることになります。

```
if (kegs > 0) {
    if (furlongs > kegs)
        fpk = furlongs/kegs;
}
else
    fpk = 0;
```

以下の例は、`else` 節の中にネストされた `if` 文を示しています。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` 文全体が終了します。

```
if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;
```

switch 文

`switch` 文は、`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` 文の本体には、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

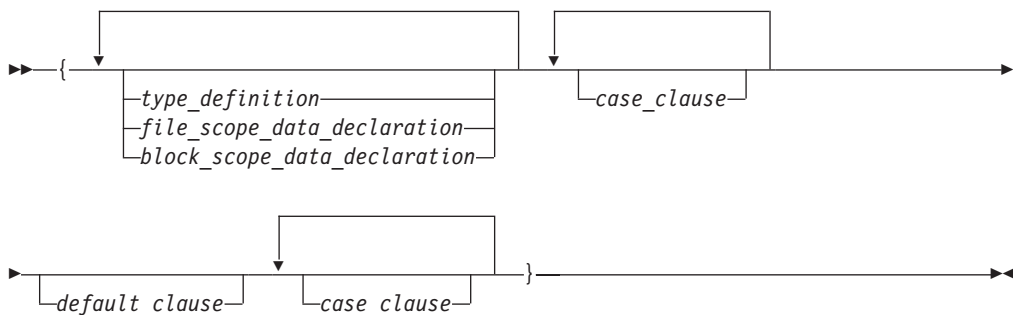
`switch` 式の値が `case` 式の 1 つの値と同じ場合、その `case` 式に続くステートメントが処理されます。そうでない場合、`default` ラベル・ステートメント (あれば) が処理されます。

switch 文

switch 文の形式は、次のとおりです。

▶▶ **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。

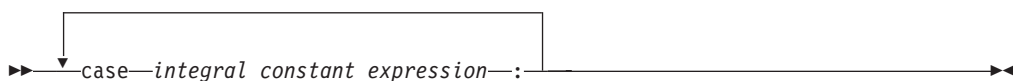


注: *type_definition*、*file_scope_data_declaration* または *block_scope_data_declaration* 内の初期化指定子は、無視されます。

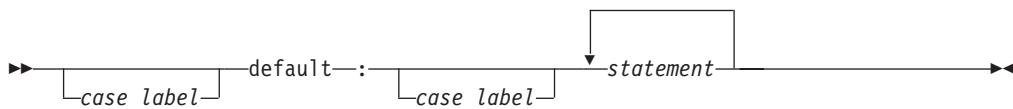
case 文節 には、任意の数のステートメントが続く *case label* が含まれます。 *case* 文節の形式は、次のとおりです。



case 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の入っています。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。



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** 文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。



switch 文は、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。

制約および制限事項

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** 文の本体の中に、宣言を入れることができます。

switch 文を使用して、初期化を飛び越えることはできません。

C 可變的に変更される型を持つ ID のスコープに、**switch** 文の **case** またはデフォルトのラベルが含まれている場合は、**switch** 文全体がその ID のスコープ内にあるものと見なされます。つまり、その ID の宣言は **switch** 文よりも前にある必要があります。

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** 文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 文の例

以下の **switch** 文には、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** 文は関数 `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 '-':
```


switch 文

```
        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** 文は、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。

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 ステートメントの形式は、次のとおりです。

▶ `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。

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。

C++ `expression` は、**bool** に変換可能なものでなければなりません。

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** 文があると、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。

do ステートメントの例

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 ステートメントの形式は、次のとおりです。

```
▶▶ for ( expression1 ; expression2 ; expression3 )
      ▶▶ statement ▶▶
```

expression1 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。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 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 これは、算術型またはポインター型に評価され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 ステートメント が反復された後で、毎回この式を評価します。この

for ステートメント

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** 文があると、2 番目の式の評価が **false** でない場合でも、**for** ステートメントを終了できます。 *expression2* を省略する場合、**break**、**return**、または **goto** 文を使用して、**for** ステートメントを終了する必要があります。

expression1 を使用すると、変数を初期化するだけでなく、その変数を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。

 **for** ステートメントのスコープ内で宣言された変数が **for** ステートメントに対してローカルでないスコープを持つようにするためのコンパイラ・オプションを設定することができます。

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。

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** 文を終了して、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 ステートメントの形式は、次のとおりです。

```
▶▶—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 文においては、**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** のような反復ステートメントの本体の中にしか含めることができません。

continue ステートメントは、反復ステートメントのアクション部分の処理を終了し、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 (pointer = strings[i]; *pointer != '\0'; /* for each character */
            ++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 ステートメントの形式は、次の 2 つのうちいずれかです。

```

>>—return—┐—————;
             └—expression—┘

```

値を戻す関数には、**return** ステートメントに式が含まれている必要があります。戻りの型が `void` である関数は、その **return** ステートメントに式を含めることはできません。

戻りの型が `void` の関数には、**return** ステートメントは不要です。**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、暗黙の戻りが最後のステートメントの完了直後に発生し、制御が自動的に呼び出し側の関数に戻ります。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;
}

```

return ステートメント

```
    for (int i = 0; i < c; ++i;) /* do the copying */
        b[i] = a[i];
    /* 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。

戻り式の値および関数値

return ステートメントに式がある場合、その式の値を呼び出し元に戻します。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、関数の戻りの型と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。

戻りの型が `void` の関数の **return** ステートメントの値は、この関数が値を戻さないことを意味しています。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、コンパイラによってエラー・メッセージが出力されます。

void 型を戻すものとして関数が宣言されている場合、式を持つ **return** ステートメントを使用することはできません。

return ステートメントの例

```
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` の値を返します。一致するものが存在しない場合、関数 `match` は `-1` (マイナス 1) という値を返します。

```
int match(int number, int array[ ], int n)
{
    int i;

    for (i = 0; i < n; i++)
        if (number == array[i])
            return (i);
    return(-1);
}
```

goto 文

`goto` 文を使用すると、プログラムは、**goto** 文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 文の形式は、次のとおりです。

```
▶▶ goto label_identifier; ▶▶
```

goto 文は、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** 文を使用しなくても済みます。

goto 文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 文を使用して、初期化を飛び越えることはできません。

 **goto** 文は、可変長配列の範囲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。

goto 文の例

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** 文を示しています。この関数は、**goto** 文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");
}
```

計算済み goto

計算済みの `goto` は、ターゲットが、同じ関数からのラベルである `goto` 文です。ラベルのアドレスは **void*** 型の定数であり、単項ラベル値演算子 **&&** をラベルに適用することによってラベルのアドレスを取得します。計算済み `goto` のターゲットは、ランタイムで既知で、同じ関数からの計算済み `goto` 文はすべて、同じターゲットを持ちます。この言語フィーチャーは C99 および C++ に対する直交拡張で、GNU C で開発されたプログラムの移植を容易にするためにインプリメントされています。

計算済み `goto` の形式は、次のとおりです。

```
▶▶ goto *expression; ▶▶
```

ここで、**expression* は、**void*** 型の式です。

関連参照

- 202 ページの『値としてのラベル』
- 133 ページの『ラベル値演算子 &&』

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。

ヌル・ステートメントの例

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 9 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あり、このファイルがコンパイラーの入力になるので、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・ディレクティブの構文は、それ以外の言語の構文からは独立してはいますが類似しています。また、プリプロセッサーの字句規則はコンパイラーの字句規則とは異なっています。プリプロセッサーは標準の C および C++ のトークンを認識するだけでなく、ファイル名、空白の有無、および行末マーク文字の位置をプリプロセッサーが認識できるようにするその他の文字を認識します。

本節では、プリプロセッサー・ディレクティブ、およびそれに関連するマクロ展開について説明します。プリプロセッサー・ディレクティブの概要に続いて、テキスト・マクロ、ファイルのインクルード、ISO 規格および事前定義されたマクロ名、条件付きコンパイル・ディレクティブ、およびプラグマを含むトピックについて説明します。

プリプロセッサーの概要

プリプロセス は、C および C++ ファイルがコンパイラーに渡される前に、それらのファイルに対して行われる予備操作です。これによって、以下のことを実行できます。

- 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。
- 現在のファイル内にファイルを組み込む。
- 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。
- 診断メッセージを生成する。
- ソースの次の行の行番号を変更し、現在のファイルのファイル名を変更する。
- マシン特有の規則を、コードの指定されたセクションに適用する。

トークン は、空白で区切られた一連の文字です。プリプロセッサー・ディレクティブで認められている空白は、スペース、水平タブ、垂直タブ、改ページ、およびコメントだけです。改行文字も、プリプロセッサー・トークンを分離することができます。

プリプロセスされるソース・プログラム・ファイルは、有効な C または C++ プログラムでなければなりません。

プリプロセッサーは、以下のディレクティブによって制御されます。

#define マクロを定義します。
#undef プリプロセッサー・マクロ定義を除去します。

プリプロセッサの概要

<code>#error</code>	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。
<code>#include</code>	別のソース・ファイルからテキストを挿入します。
<code>#if</code>	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抑止します。
<code>#ifdef</code>	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。
<code>#ifndef</code>	マクロ名が定義されない場合に、ソース・テキストを条件によりインクルードします。
<code>#else</code>	直前の <code>#if</code> 、 <code>#ifdef</code> 、 <code>#ifndef</code> 、または <code>#elif</code> 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。
<code>#elif</code>	直前の <code>#if</code> 、 <code>#ifdef</code> 、 <code>#ifndef</code> 、または <code>#elif</code> テストが失敗した場合に、定数式の値を基にして、条件によりソース・テキストをインクルードします。
<code>#endif</code>	条件テキストを終了します。
<code>#line</code>	コンパイラ・メッセージの行番号を提供します。
<code>#pragma</code>	コンパイラに対してインプリメンテーション定義の命令を指定します。

プリプロセッサ・ディレクティブの形式

プリプロセッサ・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・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。

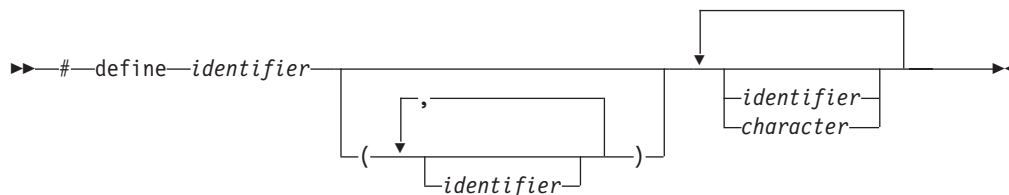
行の最後の文字が `¥` (円記号) 文字でない限り、プリプロセッサ・ディレクティブは改行文字で終了します。`¥` 文字がプリプロセッサ行の最後の文字として現れると、プリプロセッサは `¥` と改行文字を継続マーク文字として解釈します。プリプロセッサは、`¥` (およびそれに続く改行文字) を削除して、物理ソース行を継続する論理行に継ぎます。円記号と行の終わりの文字または物理的レコード終わりの間に空白を置くことができます。ただし、この空白は通常、編集中には表示されません。

一部の `#pragma` ディレクティブを除いて、プリプロセッサ・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。

マクロの定義および展開 (`#define`)

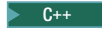
プリプロセッサ定義ディレクティブは、これ以降のマクロの出現を、指定された置換トークンに置き換えるようプリプロセッサに指示します。

プリプロセッサの `#define` ディレクティブの形式は、次のとおりです。



#define ディレクティブには、オブジェクト類似の定義または関数類似の定義を含めることができます。

#define と const の比較

- **#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** オブジェクトとは異なり、マクロの値はインライン展開されるので、コンパイラによって使用される中間ソース・コード内には現れません。インライン展開のため、マクロ値はデバッガーからは使用できません。
- マクロは配列結合などの定数式で使用するのに対し、**const** オブジェクトは定数式では使用できません。
-  コンパイラは、マクロ引き数を含め、マクロの型検査を行いません。

関連参照

- 『オブジェクト類似マクロ』
- 224 ページの『関数類似マクロ』
- 80 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 array[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、プリプロセッサは、このステートメントをプリプロセッサの出力で、以下のステートメントのように変更します。

```
int array[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
```


#define

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

オブジェクト類似マクロよりも複雑な関数類似マクロ定義では、括弧内の仮パラメーターの名前を、コンマで区切って宣言します。仮パラメーター・リストは空であってもかまいません。そのようなマクロは引き数を取らない関数をシミュレートするために使用できます。C では、数が可変の引き数を持つ関数類似のマクロのサポートが追加されています。

C++ C++ では、C との互換性のための言語拡張として、数が可変の引き数を持つ関数類似マクロがサポートされています。

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分離することが必要です。

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。このパラメーター・リストの最後には、省略符号 (...) を使用できます。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。

関数類似マクロの起動

小括弧に入れられ、コンマで区切られた引き数のリストが後に続く ID。引き数の数は、定義内のパラメーター・リストが省略符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。後者の場合、起動における引き数の数は、マクロ定義内のパラメーター数よりも多くなければなりません。この過剰分の引き数は後続引き数 と呼ばれます。プリプロセッサは、関数類似マクロの起動を確認すると、引き数の置換を行います。置換コード内のパラメーターは、対応する引き数に置き換えられます。後続の引き数がマクロ定義によって許可されている場合は、それらの引き数は、その間にコンマを挟んでマージされ、それら後続引き数があっても 1 つの引き数であるかのように、`__VA_ARGS__` で置き換えられます。引き数自体に含まれるマクロの起動はすべて、引き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。

マクロ引き数は空 (ゼロ個のプリプロセス・トークンで構成) であってもかまいません。例えば、次のような場合です。

```
#define SUM(a,b,c) a + b + c
SUM(1,,3) /* No error message.
           1 is substituted for a, 3 is substituted for c. */
```

この言語フィーチャーは、C++ の直交拡張です。

ID リストが省略符号で終了していない場合は、マクロ起動における引き数の数は、対応するマクロ定義内のパラメーターの数と同じでなければなりません。指定された引き数がすべて置換された後に残る引き数はすべて (分離するコンマを含む)、パ

ラメーターの置換時に可変引き数と呼ばれる 1 つの引き数に結合されます。可変引き数は、パラメーター・リスト内に現れるすべての ID `__VA_ARGS__` と置き換わります。次の例は、このことを示したものです。

```
#define debug(...)    fprintf(stderr, __VA_ARGS__)

debug("flag");      /*    Becomes fprintf(stderr, "flag");    */
```

マクロ起動引き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tring・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、式を正しく評価するため、定義内の各パラメーター `c` のまわりに小括弧を必要とします。

```
y = SQR(a + b);
```

プリプロセッサ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、プリプロセッサは評価の正しい順序を保てず、プリプロセッサの出力は次のようになります。

```
y = (a + b * a + b);
```

および ## 演算子の引き数は、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 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。

#define

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 **#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 **#define** ディレクティブの前に、プリプロセッサ **#undef** ディレクティブがある場合に限りです。 **#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、プリプロセッサはマクロ起動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。

#define ディレクティブの例

以下のプログラムには、2 つのマクロ定義と、その定義されている両方のマクロを参照するマクロ起動が含まれています。

```
/**
 ** 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
```

Variadic マクロ拡張

Variadic マクロ拡張は、引き数を可変数個持つマクロに関連する C99 に対する 2 つの拡張のことです。1 つ目の拡張は、可変引き数 ID を `__VA_ARGS__` からユーザー定義の ID にリネームするためのメカニズムです。この拡張は、C99 に対して直交です。もう 1 つの拡張は、変数引き数が指定されていない場合に、variadic マクロ内のぶら下がりコンマを取り外す方法を提供します。この拡張は非直交です。どちらの拡張も、GNU C および C++ で開発されたプログラムの移植を容易にするためにインプリメントされています。

 この C++ のインプリメンテーションは、C との互換性のために、variadic マクロ拡張をサポートするように Standard C++ および C++98 を拡張します。

`__VA_ARGS__` の代わりの ID

以下の例は、`__VA_ARGS__` の代わりの ID の使用法を示したものです。マクロ `debug` の最初の定義は、`__VA_ARGS__` の通常の使用の実例となります。2 番目の定義は、`__VA_ARGS__` の代わりの ID `args` の使用を示します。

```
#define debug1(format, ...) printf(format, __VA_ARGS__)
#define debug2(format, args ...) printf(format, args)
```

呼び出し

```
debug1("Hello %s\n", "World");
debug2("Hello %s\n", "World");
```

マクロ展開の結果

```
printf("Hello %s\n", "World");
printf("Hello %s\n", "World");
```

末尾のコンマの取り外し

プリプロセッサは、関数マクロに対する可変引き数が省略されているか空であり、`##` に先立つコンマが、関数マクロ定義の可変引き数 ID より前にある場合、その末尾のコンマを取り外します。

マクロ名のスコープ (#undef)

プリプロセッサの `undef` ディレクティブにより、プリプロセッサはプリプロセッサ定義のスコープを終わらせます。

プリプロセッサの **#undef** ディレクティブの形式は、次のとおりです。

▶—#—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。

#undef ディレクティブの例

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` の起動はすべて、`ABC` に渡された引き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
<code>ABC(1)</code>	<code>"1"</code>
<code>ABC>Hello there)</code>	<code>"Hello there"</code>

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き数を含む文字ストリング・リテラルに変換されます。
- プリプロセッサは、マクロに渡された引き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き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き数にストリング・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 2 番目の ¥ 文字が挿入されます。
- マクロに渡された引き数に " (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。
- 引き数のストリング・リテラルへの変換は、その引き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、その振る舞いは定義されません。

演算子の例

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¥"

演算子とのマクロ連結

(二重番号記号) 演算子は、マクロ定義に含まれるマクロの起動 (テキストまたは引き数、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 に対する引き数の最後のトークンは、y に対する引き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、引き数の中のマクロのど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。

演算子の例

演算子

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

プリプロセッサの Error ディレクティブ (#error)

プリプロセッサの *error* ディレクティブを使用すると、プリプロセッサはエラー・メッセージを生成して、コンパイルを失敗させます。

#error ディレクティブの形式は、次のとおりです。



#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セクションに到達したら、コードが生成されないようにすることができます。

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.

プリプロセッサの warning ディレクティブ (#warning)

プリプロセッサの *warning* ディレクティブを使用すると、プリプロセッサ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** に対する引き数は、マクロ展開されません。

#warning ディレクティブの形式は、次のとおりです。

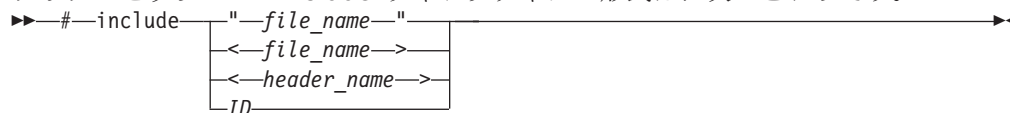


プリプロセッサの **#warning** ディレクティブは、GNU C で開発されたプログラムを扱いやすくするために提供されている直交言語拡張です。IBM のインプリメンテーションでは複数の空白文字を保持します。

ファイルのインクルード (#include)

プリプロセッサの `include` ディレクティブを使用すると、プリプロセッサは、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。

プリプロセッサの **#include** ディレクティブの形式は、次のとおりです。



C および C++ のすべてのインプリメンテーションにおいて、プリプロセッサは **#include** ディレクティブに含まれている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
```

例えば次のように、ファイル名が二重引用符で囲まれている場合、

```
#include "payroll.h"
```

プリプロセッサは、これをユーザー定義のファイルとして扱い、プリプロセッサが定義した方法でファイルを検索します。

例えば次のように、ファイル名が不等号括弧で囲まれている場合、

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

プリプロセッサは、これをシステム定義のファイルとして扱い、プリプロセッサが定義した方法でファイルを検索します。

改行および `>` の文字は、`<` と `>` で区切られたファイル名の中に入れることができません。改行および `"` (二重引用符) の文字は、`"` と `"` で区切られたファイル名の中に入れることはできません。 `>` は入れることができます。

いくつかのファイルによって使用される宣言を 1 つのファイルの中に入れ、**#include** を用いて、それらを使用する各ファイルに含めることができます。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
```

#include

```
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"
```

以下の例では、**#define** は、いくつかのプリプロセッサ・マクロを結合して C 標準入出力ヘッダー・ファイルを表すマクロを定義しています。**#include** により、ヘッダー・ファイルをプログラムで使用できるようになります。

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 * #include <stdio.h>
 */

#include C_IO_HEADER
```

特殊化されたファイルのインクルード (#include_next)

プリプロセッサ・ディレクティブ **#include_next** は、プリプロセッサに対して、指定されたファイル名の検索を続行し、現行ディレクトリー以降に検出したインスタンスをインクルードするよう指示します。このディレクティブの構文は、**#include** の構文と類似しています。

この言語フィーチャーは、C および C++ に対する直交拡張です。この言語フィーチャーは、アプリケーションおよび共用ライブラリー間でファイル名が重複している問題の処理に使用できる手法を拡張したものです。

ISO 規格事前定義マクロの名前

C および C++ は両方とも、ISO C 言語規格で指定されている次のマクロ名を事前定義しています。__FILE__ および __LINE__ の場合以外は、事前定義マクロの値が変換単位全体を通して変化することがありません。

マクロ名	説明
__DATE__	ソース・ファイルがコンパイルされた日付が入っている文字ストリング・リテラル。 ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__DATE__ の値が変わります。日付は次の形式です。 "Mmm dd yyyy" ここで、 Mmm 月を省略形式 (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, または Dec) で表します。 dd 日を表します。日が 10 より小さい場合、最初の d はブランク文字になります。 yyyy 年を表します。
__FILE__	ソース・ファイルの名前が入った文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、`__FILE__` の値が変わります。これは、**#line** ディレクティブを使用して設定できます。

`__LINE__` 現行のソース行番号を表す整数

コンパイラーがソース・プログラムの後続の行を処理すると、コンパイル中に `__LINE__` の値が変わります。これは、**#line** ディレクティブを使用して設定できます。

`__STDC__` C の場合、整数 1 であると、C コンパイラーが ISO 規格をサポートすることを示します。言語レベルを ANSI 以外に設定すると、このマクロは定義されません。(マクロが未定義の場合、`#if` 文で使われると、マクロが整数値 0 を持っているかのように動作します。)

C++ の場合、このマクロは、値 0 (ゼロ) を持つように事前定義されます。これは、C++ 言語が C の正しいスーパーセットではなく、コンパイラーは、ISO C に準拠しないことを示します。

`__STDC_HOSTED__` この C99 マクロの値は 1 です。これは、C コンパイラーがホストされたインプリメンテーションであることを示しています。

`__STDC_VERSION__` **long int** 型の整数定数: C89 言語レベルでは 199409L、C99 では 199901L。

`__TIME__` ソース・ファイルがコンパイルされた時刻が入っている文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、`__TIME__` の値が変わります。時刻は次の形式です。

`"hh:mm:ss"`

ここで、

hh 時間を表します。

mm 分を表します。

ss 秒を表します。

`__cplusplus` C++ プログラムの場合、このマクロは、長整数リテラル 199711L に展開し、コンパイラーが C++ コンパイラーであることを示します。C プログラムの場合、このマクロは定義されていません。このマクロ名には、末尾に下線がないことに注意してください。

言語標準で必要とされる事前定義マクロのほか、事前定義マクロ `__IBMC__` は、C コンパイラーのレベルを示しています。事前定義マクロ `__IBMCPP__` は、C++ コンパイラーのレベルを示しています。

値は、VRM の形式の整数です。ここで、

V バージョン番号を表します。

R リリース番号を表します。

M 変更番号を表します。

関連参照

- 238 ページの『行制御 (#line)』
- 223 ページの『オブジェクト類似マクロ』

条件付きコンパイル・ディレクティブ

プリプロセッサ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、プリプロセッサがコンパイラ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**
- **#ifdef**
- **#else**
- **#ifndef**
- **#elif**
- **#endif**

プリプロセッサ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がコンパイラ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がコンパイラ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がコンパイラ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。

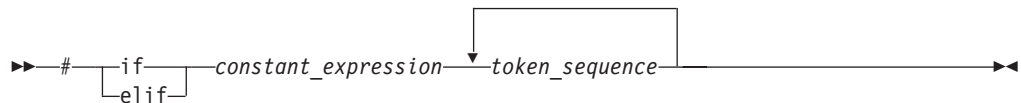
各ディレクティブは、検出された順序で処理されます。式の評価がゼロの場合、ディレクティブの後に続くブロックは無視されます。

プリプロセッサ・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・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、#elif

#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。



定数式の評価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 **#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 **#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・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

#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に渡されます。

プリプロセッサの **#ifdef** ディレクティブの形式は、次のとおりです。



以下の例は、プリプロセッサに対して EXTENDED が定義されている場合に、MAX_LEN を 75 であるとして定義します。定義されていない場合には、MAX_LEN を 50 であるとして定義します。

```
#ifdef EXTENDED
#    define MAX_LEN 75
#else
#    define MAX_LEN 50
#endif
```

#ifndef

#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に渡されます。

プリプロセッサの **#ifndef** ディレクティブの形式は、次のとおりです。



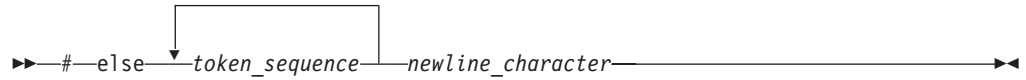
ID は、**#ifndef** キーワードの後に続いていなければなりません。以下の例は、プリプロセッサに対して EXTENDED が定義されていない場合に、MAX_LEN を 50 であるとして定義します。定義されていない場合、MAX_LEN を 75 であるとして定義します。

```
#ifndef EXTENDED
#    define MAX_LEN 50
#else
#    define MAX_LEN 75
#endif
```

#else

#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 **#else** ディレクティブを含んでいる場合、プリプロセッサ **#else** ディレクティブおよびプリプロセッサ **#endif** ディレクティブとの間にあるコードの行が、プリプロセッサによって選択され、コンパイラに渡されます。

プリプロセッサの **#else** ディレクティブの形式は、次のとおりです。



#endif

プリプロセッサ・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。

形式は次のとおりです。



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;
    }
}

```



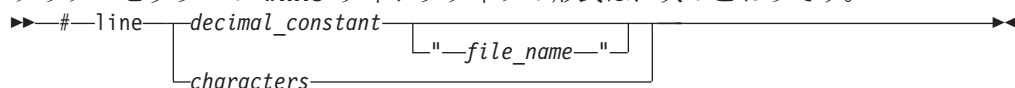
```
#if TEST >= 1
    printf("i = %d\n", i);
    printf("array[i] = %d\n",
        array[i]);
#endif

    }
    return(0);
}
```

行制御 (#line)

プリプロセッサ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。

プリプロセッサの **#line** ディレクティブの形式は、次のとおりです。



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。

AIX C99 言語レベルでは、`#line` プリプロセッサ・ディレクティブの最大値は 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 ディレクティブの例

#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。以下のプログラムは、**#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
```

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・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。

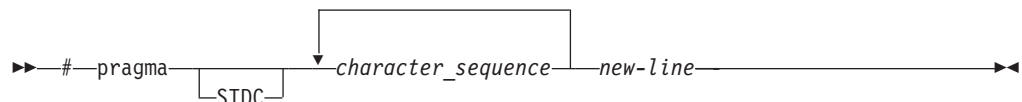
```
#ifdef MINVAL
#
#else
#define MINVAL 1
#endif
```

関連参照

- 228 ページの『# 演算子』

プラグマ・ディレクティブ (#pragma)

pragma は、コンパイラに対してインプリメンテーションを定義した命令です。この一般形式は次のとおりです。



ここで、*character_sequence* は、特定のコンパイラ命令および引き数 (あれば) を指定する一連の文字です。トークン `STDC` は、標準プラグマを示しています。したがって、このディレクティブに対してはマクロ置換は行われません。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。例えば、次のような場合です。

#pragma

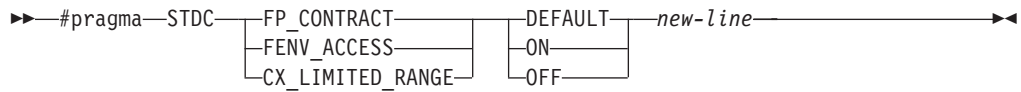
```
#define
XX_ISO_DATA
isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

1 つの **#pragma** 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。

使用可能なプラグマについては、「*XL C/C++ コンパイラー・リファレンス*」で解説されています。

標準プラグマ

C 標準プラグマ とは、C 規格で定義されている構文およびセマンティクスに準拠し、かつマクロ置き換えが行われないうプラグマ・プリプロセッサ・ディレクティブのことです。標準プラグマは、次のいずれかでなければなりません。



#pragma STDC CX_LIMITED_RANGE のデフォルトは OFF です。

C 標準プラグマについては、「*XL C/C++ コンパイラー・リファレンス*」で解説されています。

_Pragma 演算子

C 単項演算子 **_Pragma** を使用すると、プリプロセッサ・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** 式の形式は、次のとおりです。

```
▶▶—_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 ( "align(power)" )
```

これは、以下と同等です。

```
#pragma align(power)
```

第 10 章 ネーム・スペース

C++ 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。

関連参照

- 114 ページの『C++ スコープ・レゾリューション演算子 ::』

ネーム・スペースの定義

C++ 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。

構文 - ネーム・スペース

→ namespace *identifier* { namespace_body } →

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。

関連参照

- 244 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

C++ 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・スペースの元素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。

関連参照

- 4 ページの『グローバル・スコープ』

ネーム・スペース別名の作成

C++ 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。

```
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の名前として宣言されると、コンパイラ・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の名前として宣言されると、コンパイラ・エラーの結果を生じます。

関連参照

- 4 ページの『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

C++ 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。

ネーム・スペースの拡張

C++ 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・スペースおよび多重定義

 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。

関連参照

- 249 ページの『第 11 章 多重定義』

名前なしネーム・スペース

C++ 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ことを示しています。

```
#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。

関連参照

- 7 ページの『プログラム・リンケージ』
- 7 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定義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・スペースおよびフレンド

C++ 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。

関連参照

- 292 ページの『フレンド』

using ディレクティブ

C++ `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。

構文 - Using ディレクティブ

▶▶—`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、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。

関連参照

- 307 ページの『`using` 宣言およびクラス・メンバー』

using 宣言およびネーム・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。

構文 - Using 宣言

▶▶ `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
```

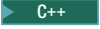
この例では、`using` 宣言の後にネーム・スペース `A` の名前である `A` がきます。次にその後にスコープ演算子 (`::`) および `k` がきます。この形式は、`using` 宣言によって、ネーム・スペース `A` の外で `k` にアクセスすることを可能にします。`using` 宣言を出した後、その特定のネーム・スペースに対して行われたすべての拡張は、`using` 宣言が行われた時点では、認識されません。

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。

関連参照

- 307 ページの『`using` 宣言およびクラス・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。

構文 - 明示的アクセス修飾

▶ `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。

関連参照

- 114 ページの『C++ スコープ・レゾリューション演算子 `::`』

第 11 章 多重定義

C++ 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き数型を、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。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

C++ 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、型または引き数リスト中の引き数の数、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、関数呼び出しの引き数リストを、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、正しい関数を選択されます。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
```

多重定義された関数の制約事項

▶ **C++** 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、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る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));
```

- パラメーターで **const**、**volatile**、および **restrict** の各 `cv` 修飾子のいずれかが使用されるという点においてのみ異なる関数宣言。この制限は、これらの修飾子

のどれかが、パラメーター型指定の最外部レベルに現れ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、**volatile**、および **restrict** の各修飾子をパラメーターの型指定内で 適用する場合には、これらの修飾子を使い分けてパラメーターを区別できるという点に注目してください。例えば、以下の宣言は、**const** および **volatile** が * ではなく **int** を修飾しており、パラメーターの型指定の最外部レベルにはないため、等価ではありません。

```
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き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)。

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、演算子で使用す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。次の例では、**complex** と呼ばれるクラスが、複素数のモデルに定義されます。そして + (プラス) 演算子は、2 つの複素数を加算するようにこのクラスで再定義されます。

```
// This example illustrates overloading the plus (+) operator.
```

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

```
class complex
{
```



```

        double real,
            imag;
public:
    complx( double real = 0., double imag = 0.); // constructor
    complx operator+(const complx&) const;      // operator+()
};

// define constructor
complx::complx( double r, double i )
{
    real = r; imag = i;
}

// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const
{
    complx result;
    result.real = (this->real + c.real);
    result.imag = (this->imag + c.imag);
    return result;
}

int main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}

```

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、引き数リストの中にデフォルト引き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。

関連参照

- 339 ページの『フリー・ストア』

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、どんな型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。演算子:

```
#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。)

関連参照

- 126 ページの『単項式』

増分と減分の多重定義

 1 個のクラス型の引き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、あるいは引き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 (`++`) を多重定義します。

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 つの関数実引き数 (1 番目はクラス型、2 番目は型 `int` を持っている) 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 個の引き数を持つ、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` 引き数を使用して、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。

関連参照

- 127 ページの『増分 ++』
- 128 ページの『減分 --』

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 進単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、どんな型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。

関連参照

- 141 ページの『2 項式』

代入の多重定義

C++ 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どれも、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** と宣言されているので、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。

関連参照

- 351 ページの『コピー代入演算子』
- 153 ページの『代入式』

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き数リスト内に、デフォルト引き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、その引き数リストが、クラス `A` に定義された関数呼び出しパラメーター・リストのど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 つの整数引き数を渡す場合、関数呼び出し演算子は、渡した引き数の値を、それぞれ `Point::x` および `Point::y` に加えます。

関連参照

- 116 ページの『関数呼び出し演算子 ()』

添え字の多重定義

▶ **C++** 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、例外を `throw` 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。

関連参照

- 117 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間接参照演算子と組にして)。これらのポインターは、普通のポインターのように動作する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時、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時)、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。

関連参照

- 119 ページの『矢印演算子 `->`』

多重定義解決

C++ 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き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き数を、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。コンパイラーは、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、1 つの実行可能関数を検出しようと試みます。コンパイラーが検出する実行可能関数が、最良の実行可能関数です。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。

可変長配列が関数仮パラメーターであるとき、左端の配列次元では、候補関数間で関数を区別しません。以下で `f` の 2 番目の定義が許されないのは、`void f(int [])` がすでに定義済みであるからです。

```
void f(int a[]) {}  
void f(int a[5]) {} // illegal
```

ただし、可変長配列内の左端のもの以外の配列次元は、可変長配列が関数仮パラメーターであるとき、候補関数の区別を行います。例えば、関数 `f` の多重定義セットは、以下の関数から成る可能性があります。

```
void f(int a[][5]) {}  
void f(int a[][4]) {}  
void f(int a[][g]) {} // assume g is a global int
```

しかし、以下をインクルードすることはできません。


```
void f(int a[][g2]) {} // illegal, assuming g2 is a global int
```

2 次レベル配列の次元を伴う候補関数があるため、`g` および `g2` は、どの関数 `f` が呼び出されるべきかという点にあいまいさを作成します。`g` および `g2` のいずれもコンパイル時に不明です。

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*)  
}
```

暗黙の変換シーケンス

 暗黙の変換シーケンスは、関数呼び出しの中の引き数を、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、各引き数に対する暗黙の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、各暗黙の変換シーケンスを 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、引き数に対する暗黙の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
 - メンバーを指すポインターの型変換
 - 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。

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型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、前者の方がい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き数を、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。

関連参照

- 160 ページの『左辺値から右辺値への変換』
- 162 ページの『ポインター型変換』
- 164 ページの『修飾変換』
- 161 ページの『整数変換』
- 162 ページの『浮動小数点の型変換』
- 161 ページの『ブール変換』

多重定義された関数のアドレスの解決

 引き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。引き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` を使用した場所に依じて、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、宣言を型「関数へのポインター」または「関数への参照」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「メンバーを指すポインター関数」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、テンプレート引き数推論を実行して、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、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。

関連参照

- 197 ページの『関数へのポインター』
- 281 ページの『メンバーへのポインター』
- 364 ページの『関数テンプレート』
- 376 ページの『明示的特殊化』

第 12 章 クラス

C++ 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、クラスの型は 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。

- キーワード **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、デフォルトにより、**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、デフォルトにより **public** になります。構造体は、デフォルトにより、**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++ では、ポリモアフィック・クラスを持つことができます。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。

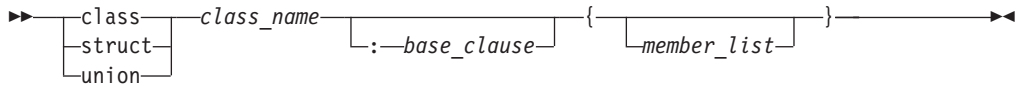
クラス・タイプの宣言

C++ クラス宣言は、固有の型のクラス名を作成します。

クラス・オブジェクトの宣言

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れ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。クラス指定子の形式は、次のとおりです。

構文 - クラス指定子



class_name は、スコープ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は、非ゼロのサイズを持ちます。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。

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++ クラスの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
}
```

コピー制限のないクラス型のオブジェクトは、関数への引き数として、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。

クラスと構造体

C++ 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;  
}
```

次に、上記の例の出力を示します。

Here are a and b: 0 1

関連参照

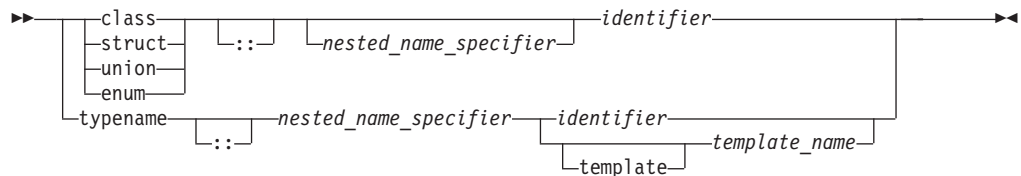
- 60 ページの『構造体』

クラス名のスコープ

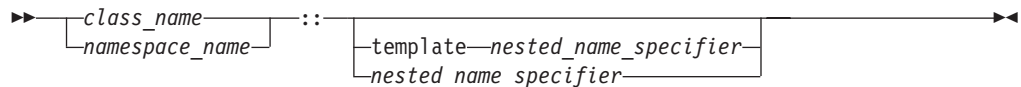
C++ クラス宣言は、クラスが宣言されたスコープ内にクラス名を導入します。囲みスコープ内にある、その名前のクラス、オブジェクト、関数、または他の宣言はすべて隠蔽されます。

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。

構文 - 詳述型指定子



構文 - ネストされた名前指定子



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。

不完全なクラス宣言

C++ **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。

クラス名のスコープ

```
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
};
```

関連参照

- 275 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・クラスでの宣言で利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 z = 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・フラグメントでは、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。


クラス名のスコープ

```
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。

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。

関連参照

- 277 ページの『メンバー関数』
- 197 ページの『インライン関数』

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を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` の引き数を取ります。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。

関連参照

- 2 ページの『スコープ』

クラス名のスコープ

- 47 ページの『typedef』

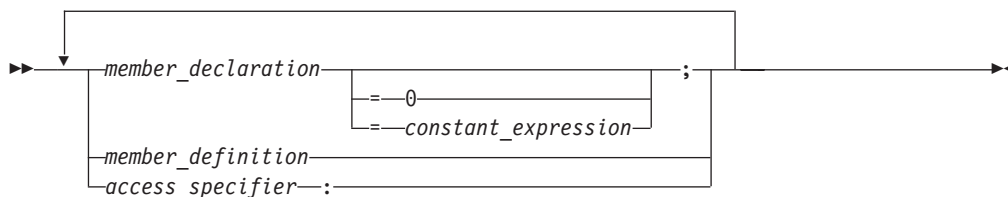
第 13 章 クラス・メンバーとフレンド

▶ **C++** 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。

クラス・メンバー・リスト

▶ **C++** 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。

構文 - クラス・メンバー・リスト



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必要があります。
- 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。
- 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** として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われます。

純粹指定子 (`= 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よって決まります。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
- ・ すべてのタイプ・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

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ができます。

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 の引き数を取り、X 型を戻します。例えば、次のようになります。

```
class X
{
    X();
    X *xptr;
```

```

X &xref;
static X xcount;
X xfunc(X);
};

```

メンバー関数

▶ **C++** 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。

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 のアドレスとなります。

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えるようにします。以下の例では、関数 add() が呼び出される時、データ変数の a、b、および c を add() の本体で使うことができます。

```

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様な方法は、**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* Z::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

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、暗黙的にインライン関数になります。これらのインライン・メンバー関数はリンケージを持ちません。

関連参照

- 292 ページの『フレンド』
- 288 ページの『静的メンバー関数』
- 169 ページの『第 7 章 関数』
- 197 ページの『インライン関数』
- 272 ページの『ローカル・クラス』


const 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。

関連参照

- 79 ページの『型修飾子』
- 80 ページの『const 型修飾子』

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。

関連参照

- 318 ページの『仮想関数』
- 324 ページの『抽象クラス』

特殊なメンバー関数

C++ 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
- 型変換関数
- コピー・コンストラクター

関連参照

- 327 ページの『第 15 章 特殊なメンバー関数』

メンバー・スコープ

C++ 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;
```

メンバー・スコープ

```
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、使用できるクラス・メンバーは、そのクラスとネスト・クラス内のメンバー関数のクラス・メンバーのみです。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。

関連参照

- 4 ページの『クラス・スコープ』

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、引き数の型、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() {
```

メンバーへのポインター


```
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。

関連参照

- 150 ページの『メンバーを指す C++ ポインター演算子 (`.*` `->*`)』
- 39 ページの『オブジェクト』

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` のアドレスです。 **this** ポインターを宣言したり、またはそれへの割り当てを行うことはできません。

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。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、次の表記を許可しません。

```
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き数として渡され、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。

this ポインターを使用しないコード	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。
<pre> #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(); } </pre>	<pre> #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); } </pre>

両方の例は、以下の出力を作成します。

```

abcdefgh
abcdefghijk1

```

関連参照

- 256 ページの『代入の多重定義』

- 349 ページの『コピー・コンストラクター』

静的メンバー

▶ **C++** 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有します。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();
}
```

関連参照

- 45 ページの『static ストレージ・クラス指定子』
- 275 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

▶ **C++** 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。

そのクラスと同じスコープ内、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。

1 つのクラス的全オブジェクトによって共用される静的メンバーは 1 つだけなので、クラス・オブジェクトとのいかなる関連付けとも関係なく、静的メンバーを参照することができます。静的メンバーは、たとえクラスのオブジェクトが宣言されていないなくても存在します。

関連参照

- 118 ページの『ドット演算子 `.`』
- 119 ページの『矢印演算子 `->`』

静的データ・メンバー

 クラスの静的データ・メンバーのコピーが 1 つだけ存在し、そのクラスのすべてのオブジェクトで共用されます。

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、外部結合があります。静的データ・メンバーは、通常のクラス・アクセス規則に従いますが、ファイル・スコープで初期化できる点が異なります。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

アクセスすることができます。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。

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。静的データ・メンバーの定義は、外部変数の定義と同じです。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・メンバーをいったん定義したあとは、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。

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。それでも、静的メンバーは囲みネーム・スペース内に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。

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。名前のないクラスや、名前のないクラスに含まれるクラスは、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・メンバーを **mutable** として宣言できません。

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。

静的メンバー関数

 同じ名前、および引き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();
}

```

次に、上記の例の出力を示します。

```

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();
}

```

次に、上記の例の出力を示します。

静的メンバー

```
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。

関連参照

- 282 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` のメンバーだとすると、クラス・メンバー `x` を宣言して、次のアクセス可能性のレベルの 1 つを持つことができます。

- **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	<code>A::a</code>	<code>A::b</code>	<code>A::c</code>
関数 <code>B::f()</code>	アクセス不可。メンバー <code>A::a</code> は、 <code>private</code> です。	アクセス可能。メンバー <code>A::b</code> は、 <code>public</code> です。	アクセス可能。クラス <code>B</code> は、 <code>A</code> から継承します。
関数 <code>C::f()</code>	アクセス可能。クラス <code>C</code> は、 <code>A</code> のフレンドです。	アクセス可能。メンバー <code>A::b</code> は、 <code>public</code> です。	アクセス可能。クラス <code>C</code> は、 <code>A</code> のフレンドです。
<code>main()</code> のオブジェクト <code>y</code>	アクセス不可。メンバー <code>y.a</code> は、 <code>private</code> です。	アクセス可能。メンバー <code>y.a</code> は、 <code>public</code> です。	アクセス不可。メンバー <code>y.c</code> は、 <code>protected</code> です。
<code>main()</code> のオブジェクト <code>z</code>	アクセス不可。メンバー <code>z.a</code> は、 <code>private</code> です。	アクセス可能。メンバー <code>z.a</code> は、 <code>public</code> です。	アクセス不可。メンバー <code>z.c</code> は、 <code>protected</code> です。

アクセス指定子は、次のアクセス指定子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度を指定します。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・メンバーは、それがそのクラスの内側、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、同じアクセス制御を持っています。

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。

メンバー・アクセス


```
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。

アクセス可能度と可視性は、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ません。

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);
}
```

次に、上記の例の出力を示します。

```
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 のフレンドであるとします。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 にアクセスすることができます。

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);
}
```

次に、上記の例の出力を示します。

```
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、次の表記を許可しません。

```
class F;
class X {
    friend class F { };
};
```

しかし、フレンド宣言で関数を定義することができます。クラスは非ローカル・クラス関数になっている必要があり、関数名は非修飾で、関数はネーム・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。

関連参照

- 290 ページの『メンバー・アクセス』
- 304 ページの『継承されたメンバー・アクセス』

フレンドのスコープ

C++ 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) のスコープ内ではなく、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。

フレンド宣言で最初に宣言されるクラスは、**extern** 宣言と同等です。次に例を示します。

```
class B {};
class A
{
    friend class B; // global class B is a friend of A
};
```

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 (::) を使用する必要があります。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;
    }
};
```

フレンド

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()` が、ネーム・スペース・スコープに宣言されたとは認めません。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。

関連参照

- 272 ページの『ローカル・クラス』

フレンドのアクセス

 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アクセスするには、クラスのメンバー関数または、そのクラスから派生したクラスを介してのみです。

フレンド宣言は、アクセス指定子には影響されません。

関連参照

- 290 ページの『メンバー・アクセス』

フレンド

第 14 章 継承

C++ 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はほぼ同じです。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 { };

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。

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、いくつかのさまざま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。

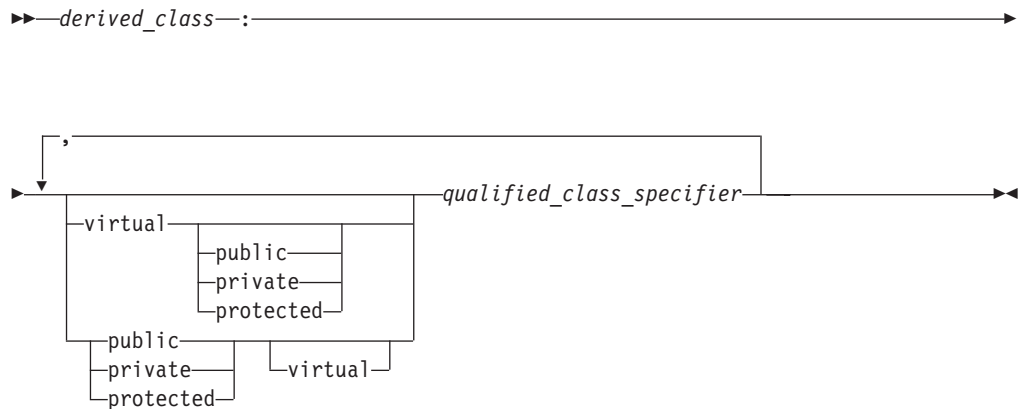
関連参照

- 313 ページの『マルチアクセス』
- 311 ページの『多重継承』
- 318 ページの『仮想関数』

派生

C++ 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。

構文 - 派生クラスの派生



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();
}

```

次に、上記の例の出力を示します。

```

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();
}
```

次に、上記の例の出力を示します。

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。


関連参照

- 312 ページの『仮想基底クラス』
- 269 ページの『不完全なクラス宣言』
- 114 ページの『C++ スコープ・レゾリューション演算子 ::』

継承されたメンバー・アクセス

本節では、protected 非静的基底クラス・メンバーに影響するアクセス規則、およびアクセス指定子を使用する派生クラスの宣言方法について説明します。

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、基底クラス的全 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なので、`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。

関連参照

- 103 ページの『参照』
- 39 ページの『オブジェクト』

基底クラス・メンバーのアクセス制御

C++ 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)

関連参照

- 290 ページの『メンバー・アクセス』
- 279 ページの『メンバー・スコープ』

using 宣言およびクラス・メンバー

C++ 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。

構文 - using 宣言

```

▶ using typename :: nested_name_specifier unqualified_id;
      :: 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、次の表記を許可しません。

```

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、ど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。

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

C++ クラス A で `f` と名付けられたメンバー関数は、戻りの型や引き数に関係なく、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();
}

```

次に、上記の例の出力を示します。

```

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);
}

```


次に、上記の例の出力を示します。

```

void B::f(int)

```

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て、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。

継承されたメンバー・アクセス

- **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。

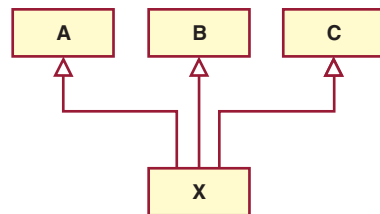
多重継承

C++ 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。

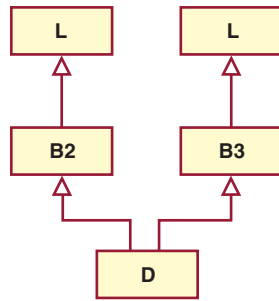


派生の順序は、コンストラクターによるデフォルト初期化およびデストラクターによる終結処理の順序の決定にのみ関係します。

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.

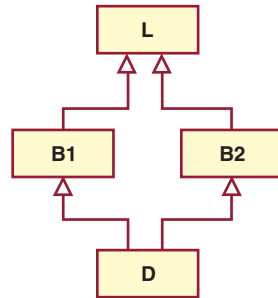
また基底クラスの宣言に基底指定子 **virtual** を使用しても、このようなあいまいさを避けることができます。

仮想基底クラス

C++ 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。

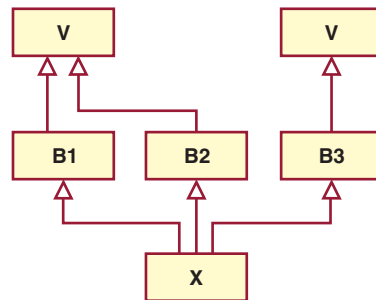


```

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。

マルチアクセス

C++ 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。

あいまいな基底クラス

▶ **C++** 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 `D::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。

あいまいなクラス・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。次に例を示します。

```
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
}
```

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。

関連参照

- 114 ページの『C++ スコープ・レゾリューション演算子 `::`』
- 312 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);
}
```

次に、上記の例の出力を示します。

Class A

関数 `g()` が呼び出されると、引き数は、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()` の引き数が、`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。引き数が、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);
}
```

次に、上記の例の出力を示します。

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();
}
```

次に、上記の例の出力を示します。

```
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返し、`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、または参照を返す。
- `B::f` が返すポインター、あるいは参照における `const` または `volatile` 修飾は、`A::f` が返すポインター、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();
};

```

次に、上記の例の出力を示します。

```

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。

- 定義済み
- 宣言された純粹

- 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。

あいまいな仮想関数呼び出し

C++ 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ています。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、1 つは B から継承されたもので、もう 1 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。ただし、コンパイラは D::B::f または D::C::f のいずれ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。

仮想関数のアクセス

 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は警告を出しますが、間接呼び出しには警告を出しません。

第 15 章 特殊なメンバー関数

▶ C++ 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。

コンストラクターとデストラクターの概要

▶ C++ 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、オブジェクトのメモリー割り振り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、オブジェクトのメモリーの終結処理や割り振り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、デフォルトの引き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、コンストラクターは、明示的に **new** 演算子を呼び出します。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振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。

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` という名前で型 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。

関連参照

- 134 ページの『C++ の new 演算子』

- 138 ページの『C++ の delete 演算子』
- 339 ページの『フリー・ストア』

コンストラクター

▶ **C++** 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、戻り値を持ってません。

関連参照

- 339 ページの『フリー・ストア』

デフォルト・コンストラクター

▶ **C++** 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、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 true 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 false 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、デフォルト引き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、末尾引き数は、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き数がある場合、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き数でコピー・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き数である限り、追加パラメーターを持つコピー・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:

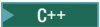
    // default constructor with one
    // default argument
    Y( int = 0);

    // default argument
    // copy constructor
    Y(const Y&, int = 0);
};
```

関連参照

- 349 ページの『コピー・コンストラクター』

コンストラクターでの明示的初期化

 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い必要があります。コン

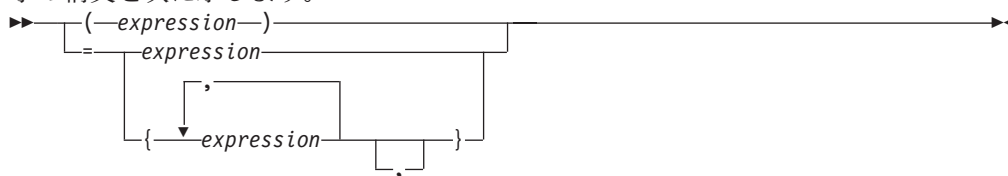
ストラクチャーを使用する明示的初期化は、集合体初期化の場合を除き、非静的定数および参照クラス・メンバーを初期化する唯一の方法です。

コンストラクター、仮想関数、`private` メンバーまたは `protected` 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き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。単一引き数の型は、コンストラクターに対する最初の引き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き数がある場合、これらの引き数はデフォルト値を持っている必要があります。

コンストラクターを用いてクラス・オブジェクトを明示的に初期化する初期化指定子の構文を次に示します。



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);
}
```

```

    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

```

関連参照

- 88 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

C++ 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き数を使用して、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。

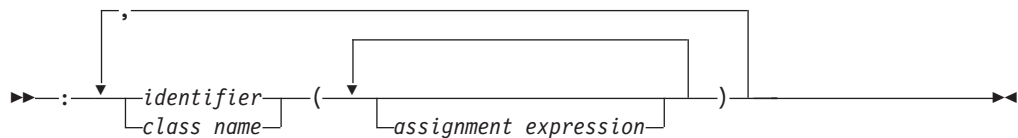
```
complx(double r, double i = 0.0) { re = r; im = i; }
```

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、関数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、引き数値は適切なクラスのデータ・メンバーに代入されます。

コンストラクター初期化指定子リストの構文を次に示します。



コンストラクター宣言の一部ではなく、関数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);
}

```

次に、上記の例の出力を示します。

```

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。次の宣言は、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)` への引き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)` への引き数の値を指定した `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、未定義です。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ています。

次の例は、B::B() の初期化指定子が異なる引き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;
}
```

以下に、上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。

関連参照

- 329 ページの『デフォルト・コンストラクター』
- 119 ページの『typeid 演算子』
- 124 ページの『dynamic_cast 演算子』

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;
}
```

次に、上記の例の出力を示します。

```
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。

関連参照

- 312 ページの『仮想基底クラス』

デストラクター

C++ デストラクター は、オブジェクトを破棄するときに、メモリーの割り振り解除、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、引き数を使用せず、戻りの型也没有。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 **true** 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 **false** 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. 非静的メンバー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・ストアにではなく、`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。

非クラス型は、疑似デストラクターを持っています。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。

関連参照

- 344 ページの『一時オブジェクト』

フリー・ストア

 フリー・ストア は、プログラムの実行中に、オブジェクトのストレージを割り振り (および割り振り解除) できるメモリーのプールです。 **new** 演算子および **delete** 演算子は、それぞれ、フリー・ストアの割り振りと割り振り解除に使用されます。

ユーザー専用のクラス用の **new** と **delete** を多重定義して、独自バージョンを定義することができます。 **new** 演算子と **delete** 演算子に追加パラメーターを宣言することができます。 **new** 演算子と **delete** 演算子がクラス・オブジェクトに使用されると、クラス・メンバー演算子関数の **new** と **delete** が、宣言してあれば、呼び出されます。

クラス・オブジェクトを **new** 演算子で作成する場合、オブジェクト作成のために、演算子関数の **operator new()** または **operator new[]()** (宣言済みの場合) のいずれかが呼び出されます。クラスの **operator new()** や **operator new[]()** は、キーワード **static** なしで宣言されている場合でも、常に静的クラス・メンバーです。この戻りの型は **void*** です。その最初のパラメーターは、オブジェクト型のサイズで、**std::size_t** 型でなければなりません。これを **virtual** にすることは、できません。

型 **std::size_t** は、インプリメンテーション依存の符号なし整数型で、標準ライブラリー・ヘッダー `<cstddef>` に定義されています。 **new** 演算子を多重定義するときは、戻り型が **void*** で、最初のパラメーターが **std::size_t** のクラス・メンバーとして宣言する必要があります。 **operator new()** または **operator new[]()** の宣言で、追加パラメーターを宣言できます。割り振り式の中で、これらのパラメーターの値を指定する配置構文を使用します。

次の例は、2 つの **operator new** 関数を多重定義します。

- **X::operator new(size_t sz)**: これは、**malloc()** が失敗した場合に、C 関数 **malloc()** でメモリーを割り振り、ストリングをスロー (**std::bad_alloc** の代わりに) することによって、デフォルトの **new** 演算子を多重定義します。
- **X::operator new(size_t sz, int location)**: この関数は、追加の整数パラメーター、**location** を取ります。この関数は、X オブジェクト 3 つまでに対してストレージを管理する、「メモリー・マネージャー」を最も単純化したものをインプリメントします。

静的配列 **X::buffer** は、3 つの **Node** オブジェクトを保持します。各 **Node** オブジェクトは、**data** という名前の X オブジェクトを指すポインター、および **filled** という名前のブール変数を含んでいます。各 X オブジェクトは、**number** と呼ぶ整数を保管します。

この **new** 演算子を使用する場合、引き数 **location** を渡して、新規の X オブジェクトを「作成」したい **buffer** の配列ロケーションを示します。配列ロケーションが「filled」(**filled** のデータ・メンバーは、配列ロケーションにおいては **false** と同じ) でなければ、**new** 演算子は、**buffer[location]** に配置された X オブジェクトを指すポインターを戻します。

```
#include <new>
#include <iostream>

using namespace std;

class X;

struct Node {
    X* data;
    bool filled;
    Node() : filled(false) { }
};
```

```

class X {
    static Node buffer[];

public:

    int number;

    enum { size = 3};

    void* operator new(size_t sz) throw (const char*) {
        void* p = malloc(sz);
        if (sz == 0) throw "Error: malloc() failed";
        cout << "X::operator new(size_t)" << endl;
        return p;
    }

    void *operator new(size_t sz, int location) throw (const char*) {
        cout << "X::operator new(size_t, " << location << ")" << endl;
        void* p = 0;
        if (location < 0 || location >= size || buffer[location].filled == true) {
            throw "Error: buffer location occupied";
        }
        else {
            p = malloc(sizeof(X));
            if (p == 0) throw "Error: Creating X object failed";
            buffer[location].filled = true;
            buffer[location].data = (X*) p;
        }
        return p;
    }

    static void printbuffer() {
        for (int i = 0; i < size; i++) {
            cout << buffer[i].data->number << endl;
        }
    }

};

Node X::buffer[size];

int main() {
    try {
        X* ptr1 = new X;
        X* ptr2 = new(0) X;
        X* ptr3 = new(1) X;
        X* ptr4 = new(2) X;
        ptr2->number = 10000;
        ptr3->number = 10001;
        ptr4->number = 10002;
        X::printbuffer();
        X* ptr5 = new(0) X;
    }
    catch (const char* message) {
        cout << message << endl;
    }
}

```

次に、上記の例の出力を示します。

```

X::operator new(size_t)
X::operator new(size_t, 0)
X::operator new(size_t, 1)
X::operator new(size_t, 2)
10000

```

```

10001
10002
X::operator new(size_t, 0)
Error: buffer location occupied

```

ステートメント `X* ptr1 = new X` は、`X::operator new(sizeof(X))` を呼び出します。ステートメント `X* ptr2 = new(0) X` は、`X::operator new(sizeof(X),0)` を呼び出します。

delete 演算子は、**new** 演算子によって作成されたオブジェクトを破棄します。**delete** のオペランドは **new** によって戻されたポインタでなければなりません。デストラクターを持つオブジェクトに対して **delete** を呼び出すと、オブジェクトの割り振り解除を行う前に、デストラクターが呼び出されます。

delete 演算子によってクラス・オブジェクトを破棄する場合、演算子関数の **operator delete()** または **operator delete[]()** (これが宣言されている場合) が呼び出され、これによってオブジェクトが破棄されます。クラスの **operator delete()** や **operator delete[]()** は、キーワード **static** なしで宣言されている場合でも、常に静的メンバーです。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。 **operator delete()** および **operator delete[]()** の戻りの型は **void** なので、これらは値を戻すことはできません。

次の例では、演算子関数 **operator new()** および **operator delete()** の宣言と使用方法について説明します。

```

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

class X {
public:
    void* operator new(size_t sz) throw (const char*) {
        void* p = malloc(sz);
        if (p == 0) throw "malloc() failed";
        return p;
    }

    // single argument
    void operator delete(void* p) {
        cout << "X::operator delete(void*)" << endl;
        free(p);
    }
};

class Y {
    int filler[100];
public:
    // two arguments
    void operator delete(void* p, size_t sz) throw (const char*) {
        cout << "Freeing " << sz << " byte(s)" << endl;
        free(p);
    }
};

int main() {
    X* ptr = new X;

    // call X::operator delete(void*)

```

```

delete ptr;

Y* yptr = new Y;

// call Y::operator delete(void*, size_t)
// with size of Y as second argument
delete yptr;
}

```

上記の例は、次の式と同じ出力を生成します。

```

X::operator delete(void*)
Freeing 400 byte(s)

```

ステートメント `delete ptr` は、`X::operator delete(void*)` を呼び出します。ステートメント `delete yptr` は、`Y::operator delete(void*, size_t)` を呼び出します。

削除されたオブジェクトにアクセスしようとした場合の結果は、削除後にオブジェクトの値が変化する可能性があるので、未定義です。

演算子関数 **new** および **delete** を宣言していないクラス・オブジェクト、または非クラス・オブジェクトに対して、**new** および **delete** を呼び出す場合、グローバル演算子 **new** および **delete** が使用されます。グローバル演算子 **new** および **delete** は、C++ ライブラリーに用意されています。

クラス・オブジェクトの配列の割り振りおよび割り振り解除用の C++ 演算子は、**operator new[]()** および **operator delete[]()** です。

delete 演算子を仮想として宣言できません。ただし、基底クラスのデストラクターを仮想として宣言することで、**delete** 演算子に、ポリモフィックの振る舞いを追加できます。次の例は、このことを示しています。

```

#include <iostream>
using namespace std;

struct A {
    virtual ~A() { cout << "~A()" << endl; };
    void operator delete(void* p) {
        cout << "A::operator delete" << endl;
        free(p);
    }
};

struct B : A {
    void operator delete(void* p) {
        cout << "B::operator delete" << endl;
        free(p);
    }
};

int main() {
    A* ap = new B;
    delete ap;
}

```

以下に、上の例の出力を示します。

```

~A()
B::operator delete

```

ステートメント `delete ap` は、`A` のデストラクターが仮想として宣言されているので、クラス `A` の代わりに、クラス `B` から **`delete`** 演算子を使用します。

`delete` 演算子からポリモフィックの振る舞いを獲得できますが、静的に可視である **`delete`** 演算子は、別の **`delete`** 演算子が呼び出されることがあっても、アクセス可能にしておく必要があります。例えば、上記の例で、代わりに `B::operator delete(void*)` が呼び出されても、関数 `A::operator delete(void*)` をアクセス可能にしておく必要があります。

仮想デストラクターは、配列 (**`operator delete[]()`**) に対するどの割り振り解除演算子にも影響を与えません。次の例は、このことを示しています。

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

struct A {
    virtual ~A() { cout << "~A()" << endl; }
    void operator delete[](void* p, size_t) {
        cout << "A::operator delete[]" << endl;
        ::delete [] p;
    }
};

struct B : A {
    void operator delete[](void* p, size_t) {
        cout << "B::operator delete[]" << endl;
        ::delete [] p;
    }
};

int main() {
    A* bp = new B[3];
    delete[] bp;
};
```


ステートメント `delete[] bp` の振る舞いは、未定義です。

`delete` 演算子を多重定義するときは、前述のように、戻り型が **`void`** で、最初のパラメーターの型が **`void*`** のクラス・メンバーとして宣言する必要があります。宣言に型 **`size_t`** の 2 番目のパラメーターを追加することができます。単一クラスには、**`operator delete()`** または **`operator delete[]()`** を 1 つしか持つことができません。

関連参照

- 134 ページの『C++ の `new` 演算子』
- 138 ページの『C++ の `delete` 演算子』
- 135 ページの配置構文
- 195 ページの『割り振り関数および割り振り解除関数』

一時オブジェクト

 コンパイラーによる一時オブジェクトの作成が必要になる場合があります。コンパイラーがこれらのオブジェクトを使用するのは、参照を初期化するときや、標準型変換、引き数の受け渡し、関数からの戻り、および **`throw`** 式の評価が含まれている式を評価するときです。

コンパイラーが参照変数の初期化のために一時オブジェクトを作成する場合、一時オブジェクトの名前は、参照変数の名前と同じスコープを持ちます。一時オブジェクトが、完全式 (別の式の副次式ではない式) の評価中に作成される場合、一時オブジェクトは、それが作成されたポイントを字句として含んでいる評価の最終ステップとして破棄されます。

完全式の破棄には、例外が 2 つあります。

- 式は、オブジェクトを定義する宣言のための初期化指定子として現れます。一時オブジェクトは、初期化が完了すると破棄されます。
- 参照は、一時オブジェクトに拘束されます。一時オブジェクトは、参照が存続期間を終了する時に破棄されます。

コンパイラーは、コンストラクターを用いてクラスの一時的オブジェクトを作成する場合、適切な (一致する) コンストラクターを呼び出して、一時オブジェクトを作成します。


一時オブジェクトが破棄され、デストラクターが存在すると、コンパイラーは、そのデストラクターを呼び出して、一時オブジェクトを破棄します。ユーザーが、一時オブジェクトが作成されたスコープから出るときに、その一時オブジェクトが破棄されます。参照が一時オブジェクトにバインドされている場合、一時オブジェクトが破棄されるのは、完了前に制御の流れが中断されたために一時オブジェクトが破棄された場合以外では、参照がスコープ外に出た場合です。例えば、参照メンバー用にコンストラクター初期化指定子によって作成された一時オブジェクトは、コンストラクターを離れるときに破棄されます。

そのような一時オブジェクトが冗長な場合は、コンパイラーは、最適化されたより効率的なコードを作成するために、それらの一時オブジェクトを構成しません。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、この振る舞いを考慮に入れてください。

関連参照

- 393 ページの『catch ブロックの引き数』
- 103 ページの『参照の初期化』
- 139 ページの『キャスト式』
- 194 ページの『関数からの戻り値』

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型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、関数引き数、関数からの戻り値、式オペランド、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。

- コンストラクターによる変換
- 型変換関数

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。

引き数を持つコンストラクターを呼び出し、その引き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、その引き数を、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き数型に変換します。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き数を変換するために、他のコンストラクターや型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。

関連参照

- 160 ページの『標準の型変換』

コンストラクターによる変換

 変換コンストラクター は、関数指定子 **explicit** の指定なしでは宣言されない、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、文字列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。

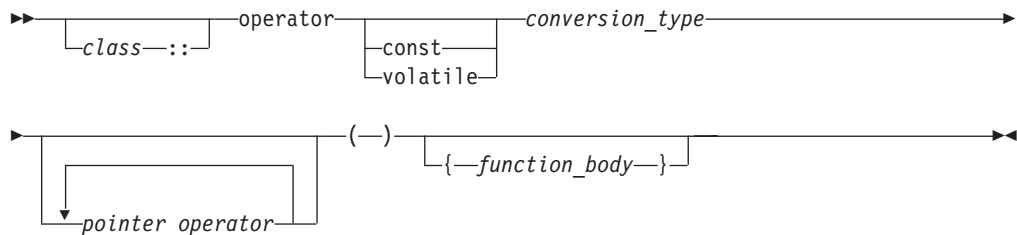
コピー・コンストラクターは、変換コンストラクターです。

関連参照

- 167 ページの『explicit キーワード』
- 134 ページの『C++ の new 演算子』
- 120 ページの『static_cast 演算子』

型変換関数

C++ 型変換関数 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。



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()` という型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。

型変換関数には引き数がなく、戻りの型が暗黙的に変換の型になります。型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。

関連参照

- 160 ページの『標準の型変換』
- 345 ページの『ユーザー定義の型変換』
- 347 ページの『コンストラクターによる変換』
- 159 ページの『第 6 章 暗黙の型変換』

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);
}

```

次に、上記の例の出力を示します。

```

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 `true` 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 `true` 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、次のなかで 1 つまたは複数 `true` 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。

関連参照

- 327 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

C++ 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;
}

```

次に、上記の例の出力を示します。

```

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 `true` 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 `true` 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、演算子の引き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 `true` 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。

関連参照

- 153 ページの『代入式』

第 16 章 テンプレート

C++ 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き数を提供すると、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、特殊化 と呼ばれます。

構文 - テンプレート宣言

→ `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、次に示す種類のテンプレート・パラメーターをコンマで区切ったリストです。

- 非型
- 型
- テンプレート

`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

`type` の `identifier` は、テンプレート宣言のスコープ内の `type_name` になるように定義します。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 L> class Key
{
    L k;
    L* kptr;
    int length;
public:
    Key(L);
    // ...
};
```

その後、次の宣言が現れるとします。

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

コンパイラーは、3 つのオブジェクトを作成します。次の表は、3 つのオブジェクトが、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。

<code>class Key<int> i;</code>	<code>class Key<char*> c;</code>	<code>class Key<mytype> m;</code>
<pre>class Key { int k; int * kptr; int length; public: Key(int); // ... };</pre>	<pre>class Key { char* k; char** kptr; int length; public: Key(char*); // ... };</pre>	<pre>class Key { mytype k; mytype* kptr; int length; public: Key(mytype); // ... };</pre>

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。不等号括弧の中に含まれている引き数は、単にクラス名に対する引き数ではなく、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

▶ **C++** 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。

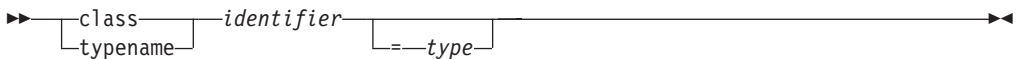
- type
- 非型
- template

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

▶ **C++** 以下は、「型」テンプレート・パラメーター宣言の構文です。

構文 - 「型」テンプレート・パラメーター宣言



identifier は、型の名前です。

「非型」テンプレート・パラメーター

▶ **C++** 「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 の型は、**int *** で、 &g の型は、**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。

「テンプレート」テンプレート・パラメーター

C++ 以下は、「テンプレート」テンプレート・パラメーター宣言の構文です。

構文 - 「テンプレート」テンプレート・パラメーター宣言

```
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;
```

テンプレート・パラメーターのデフォルトの引き数

C++ テンプレート・パラメーターは、デフォルトの引き数を持つことができます。デフォルトのテンプレート引き数のセットは、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。

デフォルトの引き数を、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、次の表記を許可しません。

```
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、デフォルトの引き数を持つ場合、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き数を持つはずです。例えば、コンパイラーは、次の表記を許可しません。

```
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、デフォルトの引き数が必要です。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き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 { };
```

テンプレート引き数

C++ 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き数があります。

- type
- 非型
- template

テンプレート引き数は、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き数を省略します。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たとしても、<> 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。

テンプレート型引き数

C++ 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き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

テンプレート引き数が、型なのか、式なのかあいまいな場合は、テンプレート引き数は、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` をテンプレート引き数として、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。

テンプレート非型引き数

C++ テンプレート引き数リストに指定されている「非型」テンプレート引き数は、コンパイル時に値が決められる式です。このような引き数は、定数式、関数のアドレス、外部結合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」テンプレート引き数を使用して、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。

非型整数引き数の場合、インスタンス引き数に、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。

非型アドレス引き数の場合、インスタンス引き数の型は、`identifier` また `&identifier` の形式でなければなりません。また、インスタンス引き数の型は、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。

テンプレート引き数リスト内に「非型」テンプレート引き数がある場合、結果として得られる値は、そのテンプレート・クラスの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、それらの引き数の値が同じ場合、それらは同じクラスであるといえます。

次の例では、クラス・テンプレートが、型引き数だけでなく、「非型」テンプレート `int` 引き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、テンプレート引き数 `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、クラスの型引き数 `T` と「非型」テンプレート引き数 `size` の値の両方を指定して作成されます。

オブジェクト `x` および引き数 `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き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、テンプレート引き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 > 記号を含む引き数は、実際に関係演算子として使用される場合は、どちらの記号もテンプレート引き数リスト区切り文字として解析されないように、それを括弧で囲む必要があります。例えば、次の定義の中の引き数は有効です。

```
myfilebuf<double, (75>25)> x;          // valid
```

ただし次の定義では、より大の演算子 (>) がテンプレート引き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
```

テンプレート引き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、テンプレート引き数の型が **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、非型引き数を持つ場合は、型引き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、これらの非型引き数の値が異なるので、別個のクラスを参照しています。

「テンプレート」テンプレート引き数

▶ C++ 「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き数は、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」テンプレート引き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。(主テンプレートとは、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 だけを考慮に入れます。

「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」テンプレート引き数に基づく部分的な特殊化を考慮に入れます。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;
}
```

次に、上記の例の出力を示します。

```
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。

クラス・テンプレート

C++ 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、コンマで区切られたリストです。

- type
- 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き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: テンプレート引き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、出力演算子 `>>` と 2 つのテンプレート・リスト区切り文字 `>` の間があいまいになります。

```
template<class L,class T> class key { /* ... */};
template<class L> class vector { /* ... */ };

int main ()
```



```
{
    class key <int, vector<int> > my_key_vector;
    // implicitly instantiates template
}
```

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)。

constructor	<pre>vehicle<char> bicycle; // constructor called automatically, // object bicycle created</pre>
オブジェクト <code>bicycle</code>	<pre>strcpy (bicycle.kind, "10 speed"); bicycle.kind[0] = '2';</pre>
関数 <code>drive()</code>	<pre>char* n = bicycle.drive();</pre>
関数 <code>roadmap()</code>	<pre>vehicle<char>::roadmap();</pre>

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・クラスの宣言を行う必要があります。クラス・テンプレートの定義は、どの変換単位でも 1 回しか現れません。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・クラス名の外観を持つクラス名は、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。

先の例では、詳述型指定子 **class** を使用して、クラス・テンプレート `key` およびポインター `keyiptr` を宣言します。`keyiptr` の宣言は、詳述型指定子なしで行うこともできます。

```

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
{ /* ... */ };

```

静的データ・メンバーとテンプレート

C++ 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、テンプレート引き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・メンバーに値を割り当てます。

クラス・テンプレートのメンバー関数

C++ 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き数を使用します。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。

フレンドとテンプレート

C++ 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化へのフレンドにすることができます。
- **多対 1**: テンプレート関数のすべてのインスタンス化は、通常、非テンプレート・クラスへのフレンドにすることができます。
- **1 対 1**: テンプレート引き数の 1 セットを使用したテンプレート関数のインスタンス化は、同じテンプレート引き数のセットを使用してインスタンス化された 1 つのテンプレート・クラスのフレンドにすることができます。これは、通常、非テンプレート・クラスと通常、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化は、テンプレート・クラスのすべてのインスタンス化へのフレンドにすることができます。

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化のフレンドです。
- 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。ここより前で宣言する必要があります。さもないと、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` のインスタンス化ごとに、`g()` にマッチングするインスタンス化が 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化にとって、`h()` のインスタンス化は、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。

関数テンプレート

▶ **C++** 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き数と 3 つの関数引き数を取ります。


- 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、テンプレート引き数を推定できる場合、テンプレート引き数を省略できます。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。

テンプレート引き数の推定

 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き数は、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き数の型を比較することにより、テンプレート引き数を推定しようとします。テンプレート引き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き数）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、テンプレート型引き数を表す
- 10 は、整数定数を示す
- i は、テンプレート非型引き数を示す
- [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」テンプレート引き数を示す
- (T)、(U)、および (V) は、少なくとも 1 つのテンプレート型引き数を持つ引き数リストを示す
- () は、テンプレート引き数を持っていない引き数リストを示す
- <T> は、少なくとも 1 つのテンプレート型引き数を持つテンプレート引き数リストを示す
- <i> は、少なくとも 1 つのテンプレート非型引き数を持つテンプレート引き数リストを示す
- <C> は、テンプレート・パラメーターに從属するテンプレート引き数を持っていない、テンプレート引き数リストを示す

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、引き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、宣言のために (テンプレート引き数を使用せずに) 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」テンプレート引き数の推定

 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、テンプレート引き数を推定できます。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き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き数を推定できません。型が、この理由により推定できな

いテンプレート引き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き数は、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>` のテンプレート引き数 `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き数を指すポインターから、関数テンプレート引き数を推定できます。しかし、多重定義関数が、どれも関数テンプレートではないこともあれば、1 つ以上の多重定義関数が、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、デフォルト引き数の型からテンプレート引き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、関数呼び出しの引き数値からテンプレート引き数 (**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()` のデフォルトの引き数からテンプレート引き数を推定できないので、呼び出し `f()` を許可しません。

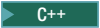
コンパイラーは、「非型」テンプレート引き数の型からテンプレート型引き数を推定できません。例えば、コンパイラーは、次の表記を許可しません。

```
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き数の推定

 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、テンプレート引き数 `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」テンプレート引き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>` の引き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、テンプレート引き数 `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」テンプレート引き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、次の表記を許可しません。


```
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。

関数テンプレートの多重定義

 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き数の推定を試み、明示的に宣言されたテンプレート引き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');
}
```

次に、上記の例の出力を示します。

```
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き数型と一致します。非テンプレート関数は、多重定義解決で優先されるため、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き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、引き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、関数引き数 **'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。

関数テンプレートの部分選択

C++ 関数テンプレートの特異化は、テンプレート引き数の推定で、特異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、最も特異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 からの特異化と一致する引き数リストは、どれも Y からの特異化と一致するが、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 よりもさらに特異化されています。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*)` よりもさらに特異化されています。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、特異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異化の呼び出し
- 関数テンプレート特異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化、または明示的特異化が、関数テンプレートの特異化を参照するとき
- 任意の配置演算子 `new` の関数テンプレートでもある適切な割り振り解除関数の決定

テンプレートのインスタンス化

C++ 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き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、**特異化** と呼ばれます。

テンプレート・インスタンス化のフォワード宣言は、**extern** キーワードに続いて明示的なテンプレート・インスタンス化の形式を持ちます。

▶—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++ および C++98 に対する直交拡張です。

暗黙のインスタンス化

▶ **C++** 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化を必要とします。

- 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化を必要としません。

- ポインター `p` が宣言されたときにクラス `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;
}
```

代入 `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、どれも静的メンバーの自分用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;

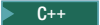
X<char*> a;
X<float> b;
X<float> c;
```

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。

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。

構文 - 明示的インスタンス化宣言

▶▶—`template—template_declaration`—▶▶

下記は、明示的インスタンス化の例です。

```
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化のインスタンス化ポイントのスコープ内になければなりません。テンプレート特殊化の明示的インスタンス化は、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化には適用されません。明示的インスタンス化の宣言に含まれるテンプレート引き数および名前、`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、明示的インスタンス生成である `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、デフォルトの引き数を使用しません。上記の例では、コンパイラーは、デフォルトの引き数が型 `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。

extern 修飾テンプレート宣言は、クラスまたは関数をインスタンス化しません。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 **extern** テンプレートの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。クラスの場合、メンバー(静的と非静的の両方)はインスタンス化されません。クラス自体は、そのクラスをマップする必要がある場合はインスタンス化されます。関数の場合、プロトタイプはインスタンス化されますが、テンプレート関数の本体はインスタンス化されません。

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>

```


テンプレート特殊化

▶ **C++** 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き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。

明示的特殊化

▶ **C++** テンプレート引き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、コンパイラは、それらのテンプレート引き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、コンパイラがテンプレート引き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

構文 - 明示的特殊化宣言

▶ `template` ↔ `declaration_name` └── `template_argument_list` ──┘ `declaration_body` ↔

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ことを示しています。 `declaration_name` は、以前に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;
}

```

次に、上記の例の出力を示します。

```

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。

明示的特殊化の定義と宣言

C++ 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、X<char> の明示的特殊化が定義されていないので、X<char> j の宣言を許可しません。

明示的特殊化とスコープ

C++ 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、次の表記を許可しません。

```

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

C++ 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

C++ 関数テンプレート特殊化では、コンパイラーが関数引き数の型からテンプレート引き数を推定できるのであれば、テンプレート引き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き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) { };
```

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、テンプレート引き数 **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、テンプレート引き数 **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。

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き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き数が、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、テンプレート引き数リストだけを持っています。部分的特殊化は、テンプレート引き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、テンプレート引き数リストがテンプレートのインスタンス化のテンプレート引き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化の一致しない残りのテンプレート引き数を使用して、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。

構文 - 部分的特殊化

►—`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、部分的特殊化を前持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();
}
```

次に、上記の例の出力を示します。

```
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・パラメーターと引き数リスト

C++ 主テンプレートは、テンプレート引き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。部分的特殊化の引き数リストの順序は、主テンプレートの暗黙の引き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き数リストでは、式が ID のみとなる場合を除いて、非型引き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」テンプレート引き数の型は、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き数リストは、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。

クラス・テンプレートの部分的特殊化のマッチング

C++ 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き数と、主テンプレートおよび部分的特殊化のテンプレート引き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き数リストは、どれも 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および従属名

C++ 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化のときに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()` を呼び出すと、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、テンプレート引き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に入れます。したがって、呼び出し `f(123)` は、`f(double)` を呼び出します。 `f(int)` のほうがうまく当てはまりますが、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化のポイントは、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化のポイントは、`i()` の直前に配置されます。関数呼び出し `h(a)` はテンプレート引き数に依存するので、コンパイラーは、関数テンプレート `g(T)` のインスタンス化の前に宣言された名前を検討します。したがって、`h(a)` の呼び出しは、`h(double)` を呼び出します。`h(int)` は `g<int>(234)` のインスタンス化のポイントのあるスコープ内になかったため、コンパイラーは、この関数を検討しません。

インスタンス化バインディングのポイントは、次の意味を持ちます。

- 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。

typename キーワード

C++ 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、非ローカル引き数 `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、書式が不正です。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。

修飾子としてのキーワード・テンプレート

C++ 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、書式が不正です。コンパイラーは、`<` が「より小演算子」と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、`.`、`->`、または `::` 演算子の後で現れ、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、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);
}
```

次に、上記の例の出力を示します。

```
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、書式が不正です。

```
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 17 章 例外処理

▶ **C++** 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。 例外的な状況进行处理するには、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。

例外処理のメカニズムは、次のエレメントで構成されます。

- **try** ブロック: 特別なプロセスで処理したい例外をスローするコードのブロック
- **catch** ブロックまたはハンドラー: **try** ブロックが例外に遭遇するときに実行されるコードのブロック
- **throw** 式: プログラムが例外に遭遇するときを示す
- 例外指定: 関数が、どの例外 (例外があれば) をスローするのかを指定する
- **unexpected()** 関数: 例外指定に指定されていない例外を、関数がスローした場合に呼び出される
- **terminate()** 関数: キャッチされない例外に対して呼び出される

関連参照

- 『try キーワード』
- 389 ページの 『catch ブロック』
- 396 ページの 『throw 式』
- 399 ページの 『例外の指定』
- 402 ページの 『unexpected()』
- 403 ページの 『terminate()』

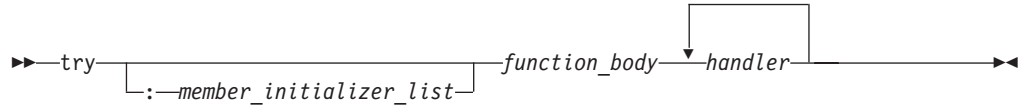
try キーワード

▶ **C++** *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、関数の本体すべてで例外を検出することを指示します。

構文 - try ブロック



構文 — 関数 try ブロック



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(...) { }
}
```

次に、上記の例の出力を示します。


```
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。

関連参照

- 332 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、引き数が `throw` 式の引き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に、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、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。これは、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。

関連参照

- 403 ページの『`terminate()`』
- 402 ページの『`unexpected()`』
- 402 ページの『特殊な例外処理関数』

`catch` ブロック

 以下は、例外ハンドラー、または `catch` ブロックの構文です。

►► `catch` — (— `exception_declaration` —) — { — `statements` — } ►►

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、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** 引き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する場合は、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ります。

関連参照

- 79 ページの『型修飾子』
- 290 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;
}
```

次に、上記の例の出力を示します。

```
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;
}
```

次に、上記の例の出力を示します。

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・ハンドラーが入ると、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(...) { }
}
```

次に、上記の例の出力を示します。

```
~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・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 a; }
    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;
}
```


次に、上記の例の出力を示します。

```
Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1
```

関連参照

- 186 ページの『main() 関数』
- 45 ページの『static ストレージ・クラス指定子』
- 241 ページの『第 10 章 ネーム・スペース』
- 337 ページの『デストラクター』

catch ブロックの引き数

 catch ブロックの引き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

引き数を初期化します。その引き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、ハンドラーがあるときはそれを破棄します。

ISO C++ 仕様では、冗長と思われる場合にコンパイラーが一時オブジェクトを作成する必要はありません。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作成します。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、このことを考慮に入れてください。

関連参照

- 344 ページの『一時オブジェクト』

スローされた例外とキャッチされた例外とのマッチング

C++ ハンドラーの `catch` 引き数の中の引き数は、次のいずれかの条件が満たされる場合、`throw` 式 (`throw` 引き数) の `assignment_expression` の引き数と一致します。

- `catch` 引き数の型が、スローされたオブジェクトの型と一致する。
- `catch` 引き数が、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` 引き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` 引き数も **`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` 引き数が、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` 引き数型は、同じ型のオブジェクトへの参照と一致します。

関連参照

- 162 ページの『ポインター型変換』
- 79 ページの『型修飾子』
- 103 ページの『参照』
- 402 ページの『特殊な例外処理関数』

キャッチの順序

C++ 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

基底クラスから派生したクラスのオブジェクトの `catch` ブロックの前に、その基底クラスのオブジェクトの `catch` ブロックがある場合、コンパイラーは、警告を発行し、派生クラス・ハンドラーで到達不能コードであるにもかかわらず、プログラムのコンパイルを継続します。

`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;
}
```

次に、上記の例の出力を示します。

```
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内で、一致するハンドラーを検出できない場合は、terminate() 関数を呼び出します。

関連参照

- 387 ページの『try キーワード』
- 403 ページの『terminate()』

throw 式

▶ **C++** `throw` 式を使用して、プログラムがいつ例外に遭遇したのかを示すことができます。

構文 - `throw` 式

▶ `throw` `assignment_expression` ▶▶

`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き数、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。

関連参照

- 85 ページの『不完全型』

例外の `rethrow`

▶ **C++** `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;
    }
}

```

次に、上記の例の出力を示します。

```

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。

関連参照

- 396 ページの『throw 式』

スタック・アンwind

 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wind と呼ばれます。自動オブジェクトは、その構築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構築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構築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構築された場合にのみ呼び出されます。

スタック・アンwind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は、`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;
    }
}
```


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できます。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、型 C が関数の例外指定に指定されておらず、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。この関数は、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することができます。

仮想関数をオーバーライドする関数は、仮想関数が指定する例外だけをスローできます。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別なメンバー関数は (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)、例外指定を持っています。暗黙的に宣言された特別なメンバー関数は、自身の例外指定に、その特別な関数が起動する関数の例外指定に宣言された型を取り込みます。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。

関連参照

- 85 ページの『不完全型』
- 170 ページの『関数宣言』
- 197 ページの『関数へのポインター』
- 327 ページの『第 15 章 特殊なメンバー関数』
- 『unexpected()』

特殊な例外処理関数

C++ 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。これらの関数は、`unexpected()` および `terminate()` です。

unexpected()

C++ 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` によって指示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()` の例外指定に対する違反が生じた場合を想定してみましょう。 `unexpected()` が `f()` の例外指定で許可された例外をスローすると、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;
    }
}

```

次に、上記の例の出力を示します。

```

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、関数 main() 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）しなかった場合は、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 が含まれている場合、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 が含まれていないと、関数 terminate() が呼び出されます。

terminate()

 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() 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wind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");
}
```

次に、上記の例の出力を示します。

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

`atexit()` を使用して関数を登録するには、`atexit()` へのパラメーターに、登録したい関数を指すポインターを渡します。プログラムが正常に終了したときには、`atexit()` は、引き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スロー式が例外を再びスローしようとしたが、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` によって指示された関数を呼び出します。デフォルトにより、`terminate_handler` は、プログラムを終了するための関数 `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` を使用しても、例外をスローすることによっても、呼び出し元に戻ることはできません。

set_unexpected() と set_terminate()


▶ **C++** 関数 `unexpected()` は、起動されたときに、`set_unexpected()` に、最後に引き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、起動されたときに、`set_terminate()` に、最後に引き数として渡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できます。関数 `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き数の型として、戻りの型が **void** で引き数なしの関数を指すポインターを持っています。引き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()` への引き数が `unexpected()` によって呼び出される関数になり、`set_terminate()` への引き数が `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、それぞれに特有の関数 (`unexpected()` および `terminate()`) から以前に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、後で元の特有の関数を復元して、`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、ユーザー独自の関数を登録する場合、その関数は、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。

 `terminate()` によって呼び出された関数からリターンしようとする、代わりに `abort()` が呼び出されて、プログラムは終了します。

例外処理関数の使用例

 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;
    }
}

```

次に、上記の例の出力を示します。

```


In first try block
Call to my_unexpected()
Caught bad_exception

```

```

In second try block
Call to my_unexpected()
Call to my_terminate

```

 また、`abort()` 関数の呼び出しにより、出力には現行ディレクトリーのコア・ファイル内のメモリー・ダンプが含まれます。

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、`set_terminate()` が以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 `old_term` に割り当てられます。

2. `set_unexpected()` を呼び出すと、`set_unexpected()` が以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を `old_unex` に割り当てます。
3. 最初の `try` ブロックの中で、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、標準出力にメッセージを印刷します。関数 `my_unexpected()` は、型 `A` の例外を再度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、例外を処理することができます。
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、`abort()` を呼び出し、これがプログラムを終了します。

 また、`abort()` 関数は、**SIGIOT** シグナルを送信します。これにより、プロセスが終了し、現行ディレクトリーのコア・ファイル内にメモリー・ダンプが生成されます。

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。

付録 A. IBM C 言語拡張機能

この付録では、IBM C 拡張機能をカテゴリーに分けて表します。大きなカテゴリーの分類基準は、拡張機能がベース言語に対して直交であるか、または非直交であるかです。直交の拡張機能は、ベース言語を妨害しません。直交の拡張機能は、`extended`、`extc89`、または `extc99` のいずれか 1 つの拡張モードでコンパイルすると、まとめて使用可能になります。 `extended` モードは、C89 に基づいています。

これに対し、非直交の拡張機能は、ベース言語フィーチャーの構文またはセマンティクスを変更する可能性があります。したがって、ベース言語に対して非直交であるか、または GNU C インプリメンテーションと競合するような IBM C 拡張機能は、それぞれ明示的に指定する必要があります。

肯定および否定の `langlvl` サブオプションの構文は、以下のとおりです。

```
-qlanglvl=lang_suboption  
-qlanglvl=nolang_suboption
```

オプションおよびサブオプションでは、大文字と小文字は区別されません。

直交拡張機能

直交の IBM C 拡張機能は、以前のリリースの個々のオプション・コントロールが備わった言語フィーチャー、C99 フィーチャーとしての拡張機能、および GNU C に関連する拡張機能という、3 つのサブグループに分類されます。

個々のオプション・コントロールが備わった既存の IBM C 拡張機能

C89 に対して直交である既存の言語フィーチャーの中には、肯定および否定の各のオプション・コントロールが備わっているものもあります。下位互換性を保つために、これらのコンパイラー・オプションおよびサブオプションのサポートは保持する必要があります。フィーチャーを重複して使用可能にしても、その使用可能状態は変更されません。

個々のオプション・コントロールが備わった IBM C 言語拡張機能

言語拡張機能	コンパイラー・オプション	注釈
ID 中のドル記号	<code>-qdollar</code>	すべてのレベルで受け入れられます。
UCS	<code>-qlanglvl=ucs</code>	否定の設定である <code>-qlanglvl=noucs</code> は STDC99 により無視され、通知メッセージが出されます。
二重字表記文字	<code>-qdigraph</code>	否定の設定である <code>-qnodigraph</code> は STDC99 により無視され、通知メッセージが出されます。

IBM C 拡張機能: C89 の拡張機能としての C99 フィーチャー

C99 に関連する言語フィーチャーのほとんどは、C89 に対して直交です。ただし、**restrict** キーワードは例外であり、ユーザーの可変ネーム・スペースを侵害します。`-qkeyword=restrict` オプションを使用することにより、明示的にサポートを要求することができます。

C89 の拡張機能としての C99 フィーチャー

言語フィーチャー	注釈
restrict 型修飾子	制限付きポインターを定義します。
可変長配列	<code>-qlanglvl=c99vla</code>
フレキシブルな配列メンバー	C99 では、フレキシブルな配列メンバーが構造体の末尾にのみ許可されています。GNU C では、これは構造体のいずれの場所にも許可されます。
複素数データ型のサポート	
<code>long long int</code> 型	
16 進浮動小数点定数のサポート	
暗黙の <code>int</code> の除去	
整数除法の詳細な定義	ゼロに切り捨て。
汎用文字名	
拡張 ID	内部名および外部名に対する制限が除去されました。
複合リテラル	
指定の初期化指定子	
C++ 形式のコメント	
暗黙の関数宣言の除去	
<code>intmax_t/uintmax_t</code> で実行されるプリプロセッサ演算	
混合宣言とコード	
選択および繰り返しステートメントに対する新規のブロック・スコープ	
整数定数型ルール	<code>long long int</code> 型に対応するため。
整数プロモーション・ルール	<code>long long int</code> 型に対応するため。
<code>vararg</code> マクロ	可変引き数を持つ、関数に似たマクロ。
<code>enum</code> 宣言内での末尾のコンマの許可	
<code>_Bool</code> 型の定義	
べき等の型修飾子	「重複する型修飾子」とも呼ばれます。
空のマクロ引き数	
追加の事前定義マクロ名	
<code>_Pragma</code> プリプロセス演算子	
標準プラグマ	<code>#pragma STDC FP_CONTRACT#pragma STDC FENV_ACCESS#pragma STDC CX_LIMITED_RANGE</code>
<code>__func__</code> 事前定義 ID	
UTF-16、UTF-32 リテラル	

GNU C に関連する IBM C 拡張機能

IBM C コンパイラーは、GNU C 言語拡張機能の以下のサブセットを認識します。以下のテーブルに使用されている説明ラベルは、GNU C のドキュメンテーションで使用されているラベルと類似しています。

GNU C に関連する IBM C 拡張機能

言語フィーチャー	注釈
式の中のステートメントおよび宣言	
ローカルに宣言されたラベル	
値としてのラベル	計算済み goto 文を含みます
ネストされた関数	
typeof を持つ型の参照	代替スペルの __typeof__ が推奨されます。
一般化された左辺値	
ダブルワードの整数	
GNU C 複素数型	
GNU C 16 進浮動小数点定数	
長さゼロの配列	
可変長の配列	
引き数の変数番号を含むマクロ	__VA_ARGS__ の代わりに ID を使用します。
左辺値でない配列が添え字を持つことができる	
定数でない初期化指定子	
複合リテラル	
共用体型へのキャスト	C のみ
関数の属性の宣言	
非プロトタイプ定義をオーバーライドする関数プロトタイプ	
位置合わせについて問い合わせるための __alignof__	
変数の属性の指定	
型の属性の指定	
C 式のオペランドを含むアセンブラー命令	
指定レジスター内の変数	コンパイラーは GNU 構文は受け入れますが、セマンティクスは無視します。
代替キーワード	
#warning	
#include_next	

非直交拡張機能

非直交の IBM C 拡張機能は、以前のリリースの言語フィーチャー、C99 フィーチャーとしての拡張機能、および GNU C に関連する拡張機能という、3 つのサブグループに分類されます。

個々のオプション・コントロールが備わった既存の IBM C 拡張機能

厳密には、IBM C 言語フィーチャーの `upconv` は非直交として分類されます。ただし、これは `extended` 言語レベルの一部として自動的に使用可能になります。これが、`extended` と `extc89` の大きな違いです。

非直交の IBM C 言語拡張機能

言語拡張機能	コンパイラ・オプション	注釈
<code>long long</code> リテラル	<code>-qlonglit</code>	<code>stdc99</code> により無視され、警告が出されます。
<code>upconv</code>	<code>-qupconv</code>	デフォルトでは <code>-qlanglvl=extended</code> 言語レベルで使用可能です。

IBM C 拡張機能: C89 の拡張機能としての C99 フィーチャー

非直交の IBM C 言語拡張機能

言語拡張機能	注釈
<code>inline</code> キーワード	C89 および GNU C に対して非直交です。
フレキシブルな配列メンバー	C99 では、フレキシブルな配列メンバーが構造体の末尾にのみ許可されています。GNU C では、これは構造体のいずれの場所にも許可されます。

GNU C に関連する IBM C 拡張機能

非直交の GNU C 拡張機能

言語拡張機能	コンパイラ・サブオプションおよび注釈
引き数の変数番号を含むマクロ	変数引き数が指定されていない場合、末尾のコンマを除去します。

付録 B. IBM C++ 言語拡張機能

IBM C++ は C のスーパーセットとしての互換性を保つため、C99 言語レベルでの互換性および GNU C 言語拡張機能との互換性を実現するフィーチャーをインプリメントしています。また、IBM C++ は、C++ の GNU 拡張機能のサブセットをサポートしています。IBM C 言語拡張機能と同様に、C++ 拡張機能も、直交と非直交の両方の言語フィーチャーを持っています。直交拡張機能 とは、既存の言語フィーチャーの振る舞いを変更することなく、ベース部分の上に追加されるフィーチャーのことです。非直交拡張機能 とは、既存構成のセマンティクスを変更したり、ベースと競合する構文を導入したりすることができる拡張機能のことです。

この付録では、IBM C++ 言語拡張機能をカテゴリーに分けて表します。個々の言語フィーチャーを使用可能および使用不可化にするためのコンパイラー・オプション構文は、IBM C の構文と同じです。extended モードでコンパイルすると、Standard C++ または C++98 に対して直交の言語拡張機能はすべて使用可能になります。特定のフィーチャーを重複して使用可能化しても、その使用可能状態は変更されません。

コンパイラー・オプションについての詳細は、「XL C/C++ コンパイラー・リファレンス」に記載されています。

直交拡張機能

直交の IBM C++ 拡張機能は、C との互換性を維持するための C99 フィーチャーに関連する拡張機能、GNU C に関連する拡張機能、および GNU C++ に関連する拡張機能という、3 つのサブグループに分類されます。

C99 との互換性のための IBM C++ 拡張機能

IBM C++ では、以下の C99 言語フィーチャーが追加されています。

C99 言語レベルでの IBM C との互換性のための IBM C++ 拡張機能

言語フィーチャー	注釈
restrict 型修飾子	制限付きポインターまたは参照を定義します。
可変長配列	
フレキシブルな配列メンバー	C99 では、フレキシブルな配列メンバーが構造体の末尾にのみ許可されています。GNU C では、これは構造体のいずれの場所にも許可されます。
複素数データ型のサポート	
16 進浮動小数点定数のサポート	
汎用文字名	C++ のコード・ポイント範囲が拡張され、C99 のコード・ポイント範囲と一致するようになりました。
複合リテラル	

C99 言語レベルでの IBM C との互換性のための IBM C++ 拡張機能

言語フィーチャー	注釈
vararg マクロ	可変引き数を持つ、関数に似たマクロ。
enum 宣言内での末尾のコンマの許可	
空のマクロ引き数	
追加の事前定義マクロ名	
_Pragma プリプロセス演算子	
__func__ 事前定義 ID	
UTF-16、UTF-32 リテラル	

GNU C に関連する IBM C++ 拡張機能

GNU C に関連する直交の IBM C++ 言語拡張機能

言語拡張機能	コンパイラー・オプションおよび注釈
式の中のステートメントおよび宣言	-qlanglvl=extended
ローカルに宣言されたラベル	-qlanglvl=gnu_locallabel
値としてのラベル	-qlanglvl=gnu_labelvalue
計算済み goto	-qlanglvl=gnu_computedgoto
typeof を持つ型の参照	-qkeyword=__typeof__
GNU C 複素数型	-qlanglvl=c99complex、-qlanglvl=gnu_complex
GNU C 16 進浮動小数点定数	-qlanglvl=c99hexfloat
長さゼロの配列	-qlanglvl=compatzea
可変長の配列	-qlanglvl=c99vla
引き数の変数番号を含むマクロ	-qlanglvl=varargsmacros、-qlanglvl=gnu_varargmacros __VA_ARGS__ の代わりに ID を使用
複合リテラル	-qlanglvl=c99compoundliteral
関数、変数、および型の属性	-qkeyword=__attribute__ すべてがキーワード __attribute__ を使用するため、ひとまとめになります。
位置合わせについて問い合わせるための __alignof__	-qkeyword=__alignof__
C 式のオペランドを含むアセンブラー命令	-qasm=gcc、-qkeyword=asm
指定レジスター内の変数	-qlanglvl=gnu_explicitregvar サブオプションを指定した場合、コンパイラーは GNU 構文は受け入れますが、セマンティクスは無視します。

GNU C に関連する直交の IBM C++ 言語拡張機能

言語拡張機能	コンパイラー・オプションおよび注釈	
代替キーワード	-qkeyword=inline -qkeyword=const -qkeyword=volatile -qkeyword=signed	-qkeyword=__alignof__ -qkeyword=__asm__ -qkeyword=__inline__ -qkeyword=__const__ -qkeyword=__extension__ -qkeyword=__restrict__ -qkeyword=__signed__ -qkeyword=__typeof__ -qkeyword=__volatile__
	これらのオプションは、問題のあるキーワードのために作成されたものです。	
#assert、#unassert、#cpu、 #machine、#system	-qlanglvl=gnu_assert すべてアサーションで機能するため、ひとまとめになります。	
#warning	-qlanglvl=gnu_warning	
#include_next	-qlanglvl=gnu_include_next	

GNU C++ に関連する IBM C++ 拡張機能

GNU C++ に関連する直交の IBM C++ 言語拡張機能

言語拡張機能	コンパイラー・オプションおよび注釈
ローカルに宣言されたラベル	ブロック内で宣言される場合にのみサポートされます。

非直交拡張機能

非直交の IBM C++ 拡張機能は、C99 フィーチャーに関連する拡張機能および GNU C に関連する拡張機能という、2 つのサブグループに分類されます。

C99 との互換性のための IBM C++ 拡張機能

C99 言語レベルでの IBM C との互換性のための非直交の IBM C++ 拡張機能

言語拡張機能	注釈
inline キーワード	Non-orthogonal to Standard C++, C++98、C89、および GNU C に対して非直交です。
フレキシブルな配列メンバー	C99 では、フレキシブルな配列メンバーが構造体の末尾にのみ許可されています。GNU C では、これは構造体のいずれの場所にも許可されます。C++ では、ゼロ・エクステント配列を許可することで、部分的なサポートを提供しています。

GNU C に関連する IBM C++ 拡張機能

GNU C に関連する非直交の IBM C++ 言語拡張機能

言語拡張機能	コンパイラー・オプションおよび注釈
typeof	-qkeyword=typeof
	typeof はユーザーのネーム・スペース下にあるため、非直交です。
引き数の変数番号を含むマクロ	-qlanglvl=varargsmacros
	変数マクロ引き数が指定されていない場合、末尾のコンマを除去します。

付録 C. 言語機能に関連した定義済みマクロ

XL C/C++ 用の定義済みマクロは、言語機能関連マクロと AIX プラットフォーム関連マクロという 2 つの一般カテゴリーに分類されます。本書では、言語機能関連マクロをについて説明します。プラットフォーム関連マクロについては、*XL C/C++ コンパイラー・リファレンス* に記載されています。

以下のマクロを使用して、C99 の機能、GNU C/C++ 関連の機能、および他の IBM 言語拡張機能を検査したり、使用可能にしたりすることができます。マクロは、リストされた機能が、指定された `qlanglvl` サブオプションについて、サポートされる場合、1 の値に対して定義されます。リストされた機能がサポートされない場合、マクロは定義されません。定義済みのマクロはすべてプロテクトされています。

C99 の機能、GNU C/C++ 関連の機能、および他の IBM 言語拡張機能について定義済みのマクロ

機能	事前定義マクロ名	-qlanglvl サブオプションでサポートされるもの
複素数	<code>_COMPLEX_I</code>	コンパイラー・オプションの適切なサブオプション、 <code>-qlanglvl</code> が必要
ネスト関数	<code>_IBM_NESTED_FUNCTION</code>	 <code>extc99</code> および拡張
柔軟な配列メンバー	<code>__C99_FLEXIBLE_ARRAY_MEMBER</code>	 <code>stdc99</code> および <code>extc99</code>
重複型修飾子	<code>__C99_DUP_TYPE_QUALIFIER</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張
#line に対する新たな制限	<code>__C99_MAX_LINE_NUMBER</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張
_Bool 型	<code>__C99_BOOL</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張
long long 型	<code>__C99_LLONG</code>	 <code>stdc99</code> および <code>extc99</code>
インライン関数指定子	<code>__C99_INLINE</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張
制限修飾子	<code>__C99_RESTRICT</code>	 <code>stdc99</code> および <code>extc99</code> <code>-qkeyword=restrict</code>  <code>-qkeyword=restrict</code>
配列宣言の静的キーワード	<code>__C99_STATIC_ARRAY_SIZE</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張
汎用文字名	<code>__C99_UCN</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張  拡張
可変長配列	<code>__C99_VAR_LEN_ARRAY</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張
__func__ キーワード	<code>__C99_FUNC__</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張  拡張
16 進浮動小数点定数	<code>__C99_HEX_FLOAT_CONST</code>	 <code>stdc99</code> 、 <code>extc99</code> 、 <code>extc89</code> および拡張  拡張
C++ 形式のコメント	<code>__C99_CPLUSCMT</code>	 <code>stdc99</code> および <code>extc99</code>

C99 の機能、GNU C/C++ 関連の機能、および他の IBM 言語拡張機能について定義済みのマクロ

機能	事前定義マクロ名	-qclanglvi サブオプションでサポートされるもの
複合リテラル	__C99_COMPOUND_LITERAL	➤ C stdc99、extc99、extc89 および拡張
指定初期化	__C99_DESIGNATED_INITIALIZER	➤ C stdc99、extc99、extc89 および拡張
混合宣言とコード	__C99_MIXED_DECL_AND_CODE	➤ C stdc99、extc99、extc89 および拡張
可変引き数を持つ、関数に似たマクロ	__C99_MACRO_WITH_VA_ARGS	➤ C stdc99、extc99、extc89 および拡張 ➤ C++ 拡張
空のマクロ引き数	__C99_EMPTY_MACRO_ARGUMENTS	➤ C stdc99、extc99、extc89 および拡張 ➤ C++ 拡張
標準プラグマ	__C99_STD_PRAGMAS	➤ C stdc99、extc99、extc89 および拡張
_Pragma 演算子	__C99_PRAGMA_OPERATOR	➤ C stdc99、extc99、extc89 および拡張 ➤ C++ 拡張
複素数型	__C99_COMPLEX	➤ C stdc99、extc99、extc89 および拡張
C99 スタイルの複素数ヘッダー	__C99_COMPLEX_HEADERS__	➤ C++ 拡張
汎用マクロの入力 <tgmath.h>	__C99_TGMATH	➤ C stdc99、extc99、extc89 および拡張
サポートされていない関数の暗黙的な宣言	__C99_REQUIRE_FUNC_DECL	➤ C stdc99
広範のストリングと広範でないストリングの連結	__C99_MIXED_STRING_CONCAT	➤ C stdc99、extc99、extc89 および拡張
非 lvalue 配列の添え字	__C99_NON_LVALUE_ARRAY_SUB	➤ C stdc99、extc99、extc89 および拡張
非一定の配列の初期化指定子	__C99_NON_CONST_AGGREG_INITIALIZER	➤ C stdc99、extc99、extc89 および拡張
GNU C インライン asm	__IBM_GCC_ASM	➤ C extc89、extc99 および拡張
ローカル・ラベル	__IBM_LOCAL_LABEL	➤ C extc99、extc89 および拡張 ➤ C++ 拡張
__alignof__	__IBM_ALIGNOF__	➤ C extc99、extc89 および拡張 ➤ C++ 拡張
__typeof__ キーワード	__IBM_TYPEOF__	➤ C extc99、extc89、拡張および -qkeyword=typeof ➤ C++ 拡張および -qkeyword=typeof
typeof キーワード	__IBM_TYPEOF__	➤ C -qkeyword=typeof ➤ C++ -qkeyword=typeof
関数属性	__IBM_ATTRIBUTES	➤ C extc99、extc89 および拡張 ➤ C++ 拡張

C99 の機能、GNU C/C++ 関連の機能、および他の IBM 言語拡張機能について定義済みのマクロ

機能	事前定義マクロ名	-qlanglvl サブオプションでサポートされるもの
型属性	<code>__IBM_ATTRIBUTES</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div> <div> <div>C++</div> <div>拡張</div> </div>
変数属性	<code>__IBM_ATTRIBUTES</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div> <div> <div>C++</div> <div>拡張</div> </div>
ID のドル記号	<code>__IBM_DOLLAR_IN_ID</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
汎用 lvalues	<code>__IBM_GENERALIZED_LVALUE</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
gnu 89 <code>__inline__</code> サポート	<code>__IBM_GCC_INLINE__</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
明示的登録変数	<code>__IBM_REGISTER_VARS</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
代替キーワード	<code>__IBM_ALTERNATE_KEYWORDS</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
<code>__extension__</code>	<code>__IBM_EXTENSION_KEYWORD</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div> <div> <div>C++</div> <div>拡張</div> </div>
<code>#assert</code> 、 <code>#unassert</code> 、 <code>#cpu</code> 、 <code>#machine</code> 、 <code>#system</code>	<code>__IBM_PP_PREDICATE</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
<code>#warning</code>	<code>__IBM_PP_WARNING</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div>
<code>#include_next</code>	<code>__IBM_INCLUDE_NEXT</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div> <div> <div>C++</div> <div>拡張</div> </div>
UTF-16 および UTF-32 ストリング・リテラル	<code>__IBM_UTF_LITERALS</code>	<div> <div>C</div> <div>extc99、extc89 および拡張</div> </div> <div> <div>C++</div> <div>拡張</div> </div>
ネーム・スペース <code>std::tr1</code> の不規則結合コンテナ (TR1 ライブラリ拡張機能)	<code>__IBMCPP_TR1__</code>	<div> <div>C++</div> <div>拡張</div> </div>

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、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。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

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を提供し、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。

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

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せんが、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、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をお願いします。

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。お客様は、IBM 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。

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, 2004 年. 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

以下は、IBM Corporation の商標です。

AIX	OS/390	pSeries
@server	POWER	S/390
IBM	PowerPC	VisualAge
		z/OS

UNIX は、The Open Group の米国およびその他の国における登録商標です。

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 C (ANSI/ISO-IEC 9899-1990 [1992]) に準拠しています。この規格は、American National Standard for Information Systems-Programming Language C (X3.159-1989) を正式に置き換えたもので、ANSI C 規格と技術的に同等です。コンパイラは、ISO/IEC 9899:1990/Amendment 1:1994 によって C 標準に採用された変更をサポートしています。
- 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 Systems-Programming Language C++ (ISO/IEC 14882:1998) に準拠しています。
- C++ 言語はまた、International Standard for Information Systems-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 203, 316

仮想関数呼び出し 322

基底および派生メンバー名 314

基底クラス 312

アクセシビリティ 290, 313

アクセス規則

仮想関数 323

基底クラス 306

クラス型 265, 290

フレンド 297

マルチアクセス 313

メンバー 290

protected メンバー 304

アクセス指定子 276, 290, 301, 309

クラス派生における 306

値による受け渡し 189

アドレス演算子 (&) 92, 129

左辺値キャスト 139

GNU C 拡張 202

暗黙の int 186

暗黙のインスタンス化

テンプレート 373

暗黙の型変換 159

型 159

基底クラスへのポインター 304

左辺値 (lvalue) 109

派生クラスへのポインター 303, 306

ブール 161

位置合わせ 36, 38

共用体 65

構造体 36, 65

構造体および共用体 38

構造体メンバー 62

ビット・フィールド 67

一時オブジェクト 344, 394

インプリメンテーションへの依存性

整数型の割り振り 58

浮動小数点型の割り振り 56

インライン

関数 197, 277

関数指定子 198

キーワード 42

打ち切り機能 404

右辺値 109

エスケープ文字 ¥ 15

エスケープ・シーケンス 15

アラーム ¥a 15

一重引用符 ¥' 15

円記号 ¥¥ 15

改行 ¥n 15

改ページ ¥f 15

疑問符 ¥? 15

垂直タブ ¥v 15

水平タブ ¥t 15

二重引用符 ¥" 15

バックスペース ¥b 15

復帰 ¥r 15

演算子 13

演算子 115

関係 144

結合順序 105

式 115

スコープ・レゾリューション 302,

314, 321

代替表記 14, 24

代入 153

コピー代入 351

多重定義 251, 277

単項 253

2 項 255

単項 126

単項正演算子 (+) 128

定義済み 235

等価 146

ビット単位否定演算子 (~) 129

複合代入 154

プリプロセッサ

pragma 240

228

229

メンバーへのポインター 150, 282

優先順位 105

型名 49

例 108

2 項 141

const_cast 123

delete 138, 342

dynamic_cast 124

new 134, 340

reinterpret_cast 121

sizeof 131

static_cast 120

typeid 119

演算子 (続き)

typeof 133

! (論理否定) 128

!= (非等価) 146

& (アドレス) 129

& (ビット単位 AND) 147

&& (論理 AND) 149

() (関数呼び出し) 116, 169

* (間接) 130

* (乗算) 142

+ (加法) 143

++ (増分) 127

, (コンマ) 155

- (減法) 143

- (単項負) 128

-- (減分) 128

-> (矢印) 119

->* (メンバーを指すポインター) 150

. (ドット) 118

.* (メンバーを指すポインター) 150

/ (除法) 142

:: (スコープ・レゾリューション) 114

= (単純代入) 153

== (等価) 146

? : (条件) 151

> (より大きい) 144

>= (より大きいまたは等しい) 144

>> (右シフト) 144

< (より小さい) 144

<= (より小さいまたは等しい) 144

<< (左シフト) 144

| (ビット単位包含 OR (包含論理和)) 148

|| (論理 OR) 149

% (剰余) 143

[] (配列添え字) 117

^ (ビット単位排他 OR) 147

演算子関数 251

演算子の結合順序 105

演算子の優先順位 105

エントリー・ポイント

プログラム 186

オーバーライド、仮想関数の 322

共変仮想関数 320

オブジェクト 109

クラス

宣言 266

静的

デストラクターでスローされる例外
390

説明 39

オブジェクト (続き)
 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 390
 restrict で修飾されたポインター 82
オブジェクト類似マクロ 223

[力行]

隠れた名前 267, 269
囲みクラス 277, 294
可視性 1, 6
 クラス・メンバー 292
 ブロック 3
下線文字 21, 23
仮想関数 278, 318
 あいまいな呼び出し 322
 アクセス 323
 オーバーライド 322
 純粹指定子 324

型

型変換 139
 可変的に変更される 96
 互換 53
 集合体 49
 スカラー 49
 複合 53, 59
 浮動 78
 列挙された 73
 class 265
型指定子 49
 オーバーライド 37
 関数定義で 181
 虚数 77
 クラス型 265
 詳述 269
 単純 54
 複素数 77
 列挙 73
 char 55
 float 56
 int 57
 long 57
 long long 57
 short 57
 unsigned 57
 wchar_t 55, 57
 (long) double 56

型修飾子

関数仮パラメーターで 83, 250
 構造体メンバー定義内 62
 制限 79, 82
 const 79, 80
 const および volatile 87
 volatile 79

型属性 50
型変換
 関数 348
 関数からポインターへ 164
 関数引き数 164
 キャスト 139
 コンストラクター 347
 左辺値から右辺値へ 109, 160
 左辺値から右辺値への 261
 算術 166
 参照 164
 修飾 164
 整数 161
 配列からポインターへ 163
 派生から基底へ 163
 派生クラスへのポインター 317
 引き数式 188
 標準の 160
 プール 161
 ポインター 162
 メンバーへのポインター 164
 ユーザー定義の 345
 explicit キーワード 167
 void ポインター 163

型名

型名キーワード 384
 修飾された 114, 271
 ローカル 273
 typeof 演算子 133

型名キーワード

括弧で囲んだ式 49, 113

仮定義

可変長配列 85, 97, 219
 型名 50
 関数仮パラメーター 97, 188, 189,
 261
 共用体メンバーとして 69
 構造体メンバーとして 61
 sizeof 112, 131

可変的に変更される型 61, 69, 96, 97,
209

サイズ評価 188

加法演算子 (+)

関数

インライン 197, 198, 277
 仮想 318
 型変換関数 348
 型名 50
 関数からポインターへの変換 164
 関数テンプレート 364
 関数呼び出し演算子 169
 クラス・テンプレート 362
 シグニチャー 169
 事前定義済み ID 22
 指定可能な属性 174
 指定子 102, 197

関数 (続き)

宣言 169, 170
 多重 173
 パラメーター名 174
 例 179
 例外指定 172
 C++ 173
 多重定義 249
 定義 169, 170, 180
 型指定子 181
 コンストラクター初期化指定子リス
 ト 182
 スコープ 181
 ストレージ・クラス指定子 181
 宣言子 181
 戻りの型 181
 例 185
 例外指定 182
 try ブロック 182
デフォルト引き数 192
 制約事項 193
 評価 193
テンプレート関数
 テンプレート引き数の推定 365
名前 170, 181
 診断 22
パラメーター 86, 116, 171, 188
引き数 116, 169, 171
 型変換 164
フレンド 292
ブロック 169
プロトタイプ 169, 171
 へのポインター 197
ポリモアフィック 301
本体 169
戻り値 169, 194
戻りの型 169, 182, 194, 195
呼び出し 116, 188
 左辺値として 109
ライブラリー関数 169
例外指定 399
例外処理 402
割り振り 195
割り振り解除 195
C++ の拡張 169
inline 42
main 186
return ステートメント 217
virtual 278, 322
関数 try ブロック 182, 387
 ハンドラー 390
関数指定機能 109
関数指定子
 明示的 347
 explicit 167
関数属性 174

関数テンプレート

明示的特殊化 378

関数類似マクロ 224

間接演算子 (*) 92, 130

間接基底クラス 300, 312

間接参照演算子 130

キーワード 22

下線文字 23

言語拡張 23

テンプレート 384

例外処理 387

template 353

疑似デストラクター 119

基底クラス

あいまいさ 312, 314

アクセス規則 306

間接 300, 312

初期化 332

抽象 324

直接 311

へのポインター 303

マルチアクセス 313

virtual 312, 317

基底リスト 311

基本型 54

キャスト式 139

共用体型 140

複素数から実際へ 78

狭幅の文字リテラル 29

共変仮想関数 320

共用体 69

位置合わせ 65

型属性 50

共用体型へのキャスト 140

組み込みプラグマ・ディレクティブ

66

クラスの型として 265, 266

互換性 53, 66, 70

指定子 70

指定初期化指定子 63

初期化 70

名前なしメンバー 64

ネスト 66

虚数タイプ 77

虚数単位 78

切り捨て

整数除法 142

空白 13, 18, 221, 222, 228

区切り子 13

代替表記 14, 24

クラス 268

アクセス規則 290

概要 265

キーワード 265

基底 301

基底リスト 301

クラス (続き)

クラス指定子 265

クラス・オブジェクト 39, 40

クラス・テンプレート 360

継承 299

集合体 266

静的メンバー 285

宣言 265

不完全な 269, 276

抽象 324

名前のスコープ 269

ネストされた 270, 295

派生 301

フレンド 292

ポリモフィック 265

メンバー関数 277

メンバー・スコープ 279

メンバー・リスト 275

ローカル 272

this ポインター 282

using 宣言 307

virtual 312, 318

クラス・テンプレート

静的データ・メンバー 362

宣言と定義 361

明示的特殊化 379

メンバー関数 362

テンプレート・クラス との区別 360

クラス・メンバー

アクセス演算子 118

アクセス規則 290

クラス・メンバー・リスト 275

初期化 332

宣言 276

割り振りの順序 276

グローバル変数 4, 8

未初期化 43

計算済み goto 133, 202, 220

継承

概要 299

多重 300, 311

継続文字 31, 222

言語拡張 23

言語拡張機能 viii

C

直交 409

非直交 411

C99 409

C++

直交 413

非直交 415

GNU C ix, 409

減分演算子 (--) 128

減法演算子 (-) 143

構造体 60, 268

位置合わせ 38, 65

構造体 (続き)

型属性 50

基底クラス 306

組み込みプラグマ・ディレクティブ 66

クラスの型として 265, 266

互換性 53, 61, 66

柔軟な配列メンバー 60, 62

初期化 63

名前なしメンバー 64

ネーム・スペース 6

ネスト 66

パック 62

メンバー 61

位置合わせ 62

埋め込み 62

ゼロ・エクステント配列 60

パック 66

不完全型 62

メモリー内のレイアウト 63

ID (タグ) 61

後置

演算子 115

式 115

++ および -- 127, 128

候補関数 249, 260

互換型

算術型 53

条件式 151

ソース・ファイル間の 53

配列 94

コピー代入演算子 351

コピー・コンストラクター 349

コメント 18

コンストラクター 329

概要 327

コピー 349

コンストラクターでスローされる例外 390

初期化

明示的 330

初期化指定子リスト 182

単純 329, 337

非単純 329, 337

変換 167, 347

例外処理 398

コンマ 155

列挙子リスト内 74

[サ行]

最良の実行可能関数 260

サフィックス

整数のリテラル定数 25

浮動小数点リテラル 26

16 進浮動小数点定数 28

サブスクリプトされていない配列

説明 94, 96

左辺値 109, 111

型変換 109

キャスト 139

算術型 54

型互換性 53

算術変換 78, 166

参照

型変換 164

初期化 103

説明 103

宣言子 130

バインディング 104

戻りの型として 195

参照による受け渡し 103, 190

式

あいまいなステートメントの解決 203

括弧で囲んだ 113

キャスト 139

コンマ 155

条件付き 151

ステートメント 203

整数定数 112

説明 105

代入 153

単項 126

メンバーへのポインター 150

割り振り 134

割り振り解除 138

1 次 111

2 項 141

full 105

new 初期化指定子 136

throw 157, 396

字下げ、コードの 222

指数 27

事前定義済み ID 22

事前定義マクロ

CPLUSPLUS 233

DATE 232

FILE 232

LINE 233

STDC 233

STDC_HOSTED 233

STDC_VERSION 233

TIME 233

__IBMCPP__ 233

__IBMC__ 233

指定機能 63, 100

共用体 70

指定 63, 100

指定機能リスト 63, 100

指定子

アクセス制御 306

インライン 197

指定子 (続き)

純粹 279

ストレージ・クラス 40

指定初期化指定子

共用体 70

集合体型 63, 100

シフト演算子 << および >> 144

集合体型 49, 331

初期化 63, 331

修飾子

制限 82

パラメーター型指定内 83, 250

const 79, 182

volatile 79, 81, 182

修飾変換 164

修飾名 114, 271

従属名 383

柔軟な配列メンバー 62

純粹仮想関数 324

純粹指定子 276, 279, 321, 324

条件式 (? :) 151, 155

条件付きコンパイル・ディレクティブ
234

例 237

elif プリプロセッサ・ディレクティブ 235

else プリプロセッサ・ディレクティブ 237

endif プリプロセッサ・ディレクティブ 237

if プリプロセッサ・ディレクティブ 235

ifdef プリプロセッサ・ディレクティブ 236

ifndef プリプロセッサ・ディレクティブ 236

乗算演算子 (*) 142

詳述型指定子 269

情報隠蔽 2, 3, 275, 304

剰余演算子 (%) 143

省略符号

関数宣言で 172

関数定義で 184

変換シーケンス 262

マクロ引き数リスト内の 224

初期化

基底クラス 332

共用体メンバー 70

クラス・メンバー 332

参照 164

自動オブジェクト 41

集合体型 63

順序 37

静的オブジェクト 46

静的データ・メンバー 288

extern オブジェクト 43

初期化 (続き)

register オブジェクト 44

初期化指定子 88, 182, 332

共用体 70

集合体型 63, 100

列挙型 75

除法演算子 (/) 142

スカラー型 49, 89

スコープ 1

囲みおよびネスト 3

関数 3

関数プロトタイプ 4

クラス名 269

グローバル 4

グローバル・ネーム・スペース 4

説明 2

ネスト・クラス 270

フレンド 294

マクロ名 228

メンバー 279

ローカル (ブロック) 3

ローカル・クラス 272

class 4

ID 5

スコープ・レゾリュション演算子

あいまいな基底クラス 314

仮想関数 321

継承 302

説明 114

スタック・アンwind 398

ステートメント 201

あいまいさの解決 203

式 203

選択 206, 207

ヌル 220

複合 205

ブロック 204

ラベル 201

break 215

continue 215

do 212

for 213

return 194, 217

while 211

ステートメント式 205

ストリング

ターミネーター 31

リテラル 31

ストレージ期間 1

静的 45

デストラクターでスローされる例外
390

auto ストレージ・クラス指定子 42

extern ストレージ・クラス指定子 43

register ストレージ・クラス指定子 44

ストレージ・クラス指定子 40, 181

ストレージ・クラス指定子 (続き)

auto 41
extern 42
mutable 44
register 44
static 45

スペース文字 222

制限 82

パラメーター型指定内 83, 250

整数

暗黙の int 182
拡張 159
型変換 161
データ型 57
定数式 73, 112
リテラル 25

静的

データ・メンバー 286
データ・メンバーの初期化 288
配列宣言で 46
メンバー 271, 285
メンバー関数 288

接続プリプロセッサ・ディレクティブ

228

接続プリプロセッサ・ディレクティブ

229

接頭部

16 進整数リテラル 26
16 進浮動小数点定数 28
8 進整数リテラル 26
++ および -- 127, 128

ゼロ・エクステンント配列 62

宣言

あいまいなステートメントの解決 203
クラス 265, 269
構文 35, 49
サブスクリプトされていない配列 96
説明 35
フレンド 297
メンバーへのポインター 281
メンバー・リストのフレンド指定子 292

宣言子

参照 103
説明 87

宣言領域 2

増分演算子 (++) 127

添え字演算子 94, 117

型名内 50

添え字宣言子

配列の 95

[タ行]

大括弧

大括弧形式 100

代入演算子 (=)

単純 153
複合 154
ポインター 90

タグ

共用体 70
構造体 61
列挙 73, 74

多次元配列 96

多重

アクセス 313
継承 300, 311

多重定義

演算子 251, 265

関数呼び出し 257

クラス・メンバー・アクセス 259

減分 254

増分 254

添え字 258

代入 256

単項 253

2 項 255

関数 249, 308

制約事項 250

関数テンプレート 371

説明 249

delete 演算子 342

new 演算子 340

多重定義解決 260, 317

多重定義された関数のアドレスの解決 262

単項演算子 126

正 (+) 128

負 (-) 128

ラベル値 133

単項式 126

単純型指定子 54

char 55

wchar_t 55

抽象クラス 322, 324

直接基底クラス 311

直交の言語拡張機能

C 409

C++ 413

データ・メンバー

スコープ 279

静的 286

説明 276

定義

仮の 39

説明 35

マクロ 222

メンバー関数 277

定数式 73, 112

定数初期化指定子 276

デストラクター 337

デストラクター (続き)

概要 327

疑似 119, 339

デストラクターでスローされる例外 390

例外処理 398

デフォルト・コンストラクター 329

テンプレート

インスタンス化 353, 372, 376

暗黙の 373

フォワード宣言 373, 375

明示的 374

インスタンス化のポイント 384

関数

多重定義 371

引き数の推定 370

部分選択 372

関数テンプレート 364

「型」テンプレート引き数の推定 368

クラス

静的データ・メンバー 362

宣言と定義 361

明示的特殊化 379

メンバー関数 362

テンプレート・クラス との区別 360

クラスとそのフレンド間の関係 363

従属名 383

スコープ 377

宣言 353

定義のポイント 383

特殊化 353, 372, 376

名前のバインディング 383

パラメーター 354

デフォルト引き数 355

非型 354

template 355

type 354

引き数

型 356

非型 357

部分的特殊化 380

パラメーターと引き数リスト 382

マッチング 382

明示的特殊化 376, 377

関数テンプレート 378

クラス・メンバー 378

宣言 376

定義と宣言 377

テンプレート引き数 356

型 356

推定 365

推定、型 368

推定、非型 370

非型 357

テンプレート引き数 (続き)

template 359

テンプレート・キーワード 384

トークン 13, 221

演算子および区切り子の代替表記 14

等価演算子 (==) 146

動的バインディング 318

特殊なメンバー関数 279

特殊文字 15

ドット演算子 118

ドル記号 15, 21

[ナ行]

名前

隠れた 114, 267, 269

レゾリューション 307, 316

ローカル型 273

名前なしネーム・スペース 244

名前の隠蔽 6, 114

あいまいさ 315

アクセス可能な基底クラス 317

名前のバインディング 383

二重字表記文字 24

ヌル

ステートメント 220

プリプロセッサ・ディレクティブ
239

ポインター定数 163

文字 ¥0 31

pointer 91

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 241

拡張 242

クラス名 269

コンテキスト 6

宣言 241

多重定義 243

定義 241

名前なし 244

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 390

フレンド 246

別名 242

明示的アクセス 248

メンバー定義 245

ユーザー定義の 2

ID 5

using 宣言 247

using ディレクティブ 246

ネスト・クラス

スコープ 270

フレンドのスコープ 295

[ハ行]

排他 OR 演算子、ビット単位 (^) 147

配置構文 135, 341

配列

型互換性 94

可変長 50, 97

関数仮パラメーター 46, 94, 95

柔軟な配列メンバー 60, 62

初期化 98, 100

説明 94

ゼロ・エクステンント 60, 62

宣言 46, 95

添え字演算子 117

多次元 96

配列からポインターへの変換 163

declaration 276

バインディング 104

仮想関数 318

静的 318

直接 104

動的 318

派生 301

配列型 94

public、protected、private 306

派生クラス

構築順序 336

へのポインター 303

catch ブロック 394

パック

構造体メンバー 66

番号記号 (#)

プリプロセッサ演算子 228

プリプロセッサ・ディレクティブの
文字 222

ハンドラー 389

汎用文字名 16, 21, 30, 31

引き数

値による受け渡し 189

受け渡し 188

後続 224, 227

参照による受け渡し 190

デフォルト 192

評価 193

main 関数 187

左シフト演算子 (<<) 144

非直交の言語拡張機能

C 411

C++ 415

ビット単位否定演算子 (~) 129

ビット・フィールド 66

型名 133

ビット・フィールド (続き)

構造体メンバーとして 61

非等価演算子 (!=) 146

評価順序点 105, 156

標準の型変換 159, 160

ブール

型変換 161

変数 55

リテラル 24

ファイルのインクルード 231, 232

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 96

不完全型 85, 94

クラス宣言 269

構造体メンバーとして 60, 61, 62

複合

型 59

式 155

ステートメント 204

代入 154

リテラル 24, 32, 115

複合型 53

ソース・ファイル間の 53

副次作用 82, 105

複数文字リテラル 29

複素数型 77, 127

浮動小数点

拡張 159

定数 28

変換 162

リテラル 26

浮動小数点型 55, 78

プラグマ

標準の 240

プリプロセッサ・ディレクティブ
239

_Pragma 240

プラグマ演算子 240

フリー・ストア 339

delete 演算子 138

new 演算子 134

プリプロセッサ演算子

228

229

_Pragma 240

プリプロセッサ・ディレクティブ 221

条件付きコンパイル 234

特殊文字 222

プリプロセスの概要 221

warning 230

プリプロセッサ・ディレクティブの定義
222

フレンド

アクセス規則 297

指定子 292

スコープ 294

フレンド (続き)

テンプレートを必要とする場合のクラスとの関係 363
ネスト・クラス 295
ポインターの暗黙的型変換 306
メンバー関数 277

ブロックの可視性 3

ブロック・ステートメント 204

プロトタイプ 171

プロモーション

関数引き数の値 188
整数および浮動小数点 159

文

goto 219
if 206
switch 207

ベクトル型 133

別名 103

変換

暗黙の変換シーケンス 261
浮動小数点 162

変換シーケンス

暗黙の 261
省略符号 262
標準の 261
ユーザー定義の 262

変換単位 1

変更可能な左辺値 109, 153

変数属性 36

ポインター

型修飾 90
型変換 162, 317
関数への 197
互換 90
説明 89
ヌル 91
汎用 163
ポインター演算 92
メンバーへの 150, 281
const 81
cv 修飾 90
restrict で修飾された 82
this 282
void* 162

包含 OR 演算子、ビット単位 (|) 148

ポリモアフィズム

ポリモアフィック関数 301
ポリモアフィック・クラス 265, 319

ポンド記号 (#)

ブリプロセッサ演算子 228
ブリプロセッサ・ディレクティブの文字 222

[マ行]

マクロ

オブジェクト類似 223
可変引き数 224, 227
関数類似 224
定義 222

typeof 演算子 133

呼び出し 224

マルチバイト文字 18

連結 32

右シフト演算子 (>>) 144

明示的

インスタンス化、テンプレート 374
型変換 139
キーワード 347
特殊化、テンプレート 376, 377

メンバー

アクセス 290
アクセス制御 309
仮想関数 278
クラス・メンバー・アクセス演算子 118
スコープ 279
静的 271, 285
データ 276
へのポインター 150, 281
protected 304

メンバー関数

静的 288
定義 277
特殊な 279
フレンド 277
const および volatile 278
this ポインター 282, 323

メンバーへのポインター

演算子 150, 282
型変換 164
宣言 281

メンバー・リスト 266, 275

文字

マルチバイト 18, 32
リテラル 29

文字セット

ソース 14
extended 18

モジュール演算子 (%) 143

戻りの型

として参照 195
size_t 131

[ヤ行]

ユーザー定義の型変換 345

ユニコード 16

より大きい演算子 (>) 144

より大きいまたは等しい演算子 (>=) 144

より小さい演算子 (<) 144

より小さいまたは等しい演算子 (<=) 144

弱いシンボル 38

[ラ行]

ラベル

値として 202
暗黙宣言 3
ステートメント 201
ローカルに宣言 202
switch 文内 208

リテラル 24

ストリング 31

整数 25

データ型 25

10 進数 25

16 進 26

8 進 26

ブール 24

複合 24, 32, 115

浮動小数点 26

文字 29

ユニコード 17

リンケージ 1, 7

インライン関数 198

インライン・メンバー関数 278

外部 8, 116

関数定義で 181, 182

関数ポインターで 197

言語 9

指定 9

内部 7, 45

なし 8

複数の関数宣言 173

弱いシンボル 38

auto ストレージ・クラス指定子 42

const cv-qualifier 80

extern ストレージ・クラス指定子 10, 43

register ストレージ・クラス指定子 45

static ストレージ・クラス指定子 47

例外

関数 try ブロック・ハンドラー 390

指定 182, 399

declaration 390

例外処理 387

関数 try ブロック 387

キャッチの順序 394

コンストラクター 398

試行例外 390

スタック・アンワインド 398

デストラクター 398

特殊な関数 402

ハンドラー 387, 389

例外処理 (続き)

引き数のマッチング 394

例、C++ 405

例外オブジェクト 387

例外の rethrow 396

catch ブロック 389

arguments 393

set_terminate 404

set_unexpected 404

terminate 関数 403

throw 式 389, 396

try ブロック 387

unexpected 関数 402

列挙型 73

後続のコンマ 74

互換性 53, 73

初期化 75

宣言 74

列挙子 74

連結

マクロ 229

マルチバイト文字 32

u-literals、U-literals 17

ローカル

型名 273

クラス 272

論理演算子

! (論理否定) 128

&& (論理 AND) 149

|| (論理 OR) 149

[ワ行]

ワイド文字

リテラル 29

ワイド・ストリング・リテラル 32

割り振り

関数 195

式 134

割り振り解除

関数 195

式 138

[数字]

1 次式 111

1 の補数演算子 (~) 129

10 進整数リテラル 25

16 進

浮動小数点定数 28

16 進整数リテラル 26

2 進式および演算子 141

3 文字表記 18

8 進整数リテラル 26

A

alignof 演算子 23, 131

AND 演算子、ビット単位 (&) 147

AND 演算子、論理 (&&) 149

argc (引き数カウント) 187

例 187

arguments

受け渡し 169

マクロ 224

catch ブロックの 393

argv (引き数ベクトル) 187

例 187

ASCII 文字コード 16

asm 22, 84

atexit 関数 403

auto ストレージ・クラス指定子 41

B

break ステートメント 215

C

case ラベル 208

catch ブロック 182, 387, 389

キャッチの順序 394

引き数のマッチング 394

char 型指定子 55

character

データ型 55

Classic C vii

const 80

オブジェクト 109

型名内に配置 50

関数属性 176

修飾子 79

メンバー関数 278

const 型のキャスト 191

#define との比較 223

const_cast 123, 191

continue ステートメント 215

CPLUSPLUS マクロ 233

cv-qualifier 62, 79, 87

関数定義で 182

構文 79

パラメーター型指定内 83, 250

C++ 以外のプログラムへのリンク 9

D

DATE マクロ 232

default

文節 208

ラベル 208

defined 単項演算子 235

delete 演算子 138

do ステートメント 212

double 型指定子 56

downcast 125

dynamic_cast 124

E

EBCDIC 文字コード 16

elif プリプロセッサ・ディレクティブ 235

else

ステートメント 206

プリプロセッサ・ディレクティブ 237

endif プリプロセッサ・ディレクティブ 237

enum

キーワード 74, 76

error プリプロセッサ・ディレクティブ 230

explicit

関数指定子 102

キーワード 167

extern inline

キーワード 42, 198

extern ストレージ・クラス指定子 8, 10, 42, 198

暗黙宣言 116

可変長配列の使用 97

関数ポインターで 197

テンプレート宣言で 373, 375

F

FILE マクロ 232

float 型指定子 56

for ステートメント 213

friend

仮想関数 321

G

goto 文 219

計算済み goto 220

制約事項 219

I

ID 20, 111

事前定義済み 22

大/小文字の区別 21

特殊文字 15, 21

ネーム・スペース 5

ID (続き)

予約済み 21, 22, 23

リンケージ 8

ID 式 112

id-expression 88

identifier

ラベル 201

if

プリプロセッサ・ディレクティブ
235

文 206

ifdef プリプロセッサ・ディレクティブ
236

ifndef プリプロセッサ・ディレクティブ
236

include プリプロセッサ・ディレクティブ
231

include_next プリプロセッサ・ディレクティブ
232

inline

関数指定子 102

キーワード 23

K

K&R C vii

L

line プリプロセッサ・ディレクティブ
238

LINE マクロ 233

long double 型指定子 56

long long 型指定子 54, 58

long 型指定子 57

lvalues 79

型変換 160, 261

M

main 関数 186

引き数 187

例 187

mutable ストレージ・クラス指定子 44

N

new 演算子

初期化指定子の式 136

説明 134

デフォルト引き数 193

配置構文 135, 340, 341

set_new_handler 関数 137

O

OR 演算子、論理 (II) 149

P

packed

変数属性 38

pragma 演算子 127

R

register ストレージ・クラス指定子 44

reinterpret_cast 121

return ステートメント 194, 217
値 218

RTTI サポート 119, 124

S

set_new_handler 関数 137

set_terminate 関数 404

set_unexpected 関数 402, 404

short 型指定子 57

signed 型指定子

char 55

int 57

long 57

long long 57

sizeof 演算子 131

可変長配列の使用 97

size_t 131

Standard C vii

Standard C++ vii

static

可変長配列の使用 97

ストレージ・クラス指定子 45

リンケージ 47

配列宣言で 95

バインディング 318

static ストレージ・クラス指定子 8

static_cast 120

STDC マクロ 233

STDC_HOSTED マクロ 233

STDC_VERSION マクロ 233

struct 型指定子 61

switch 文 207

T

terminate 関数 387, 389, 394, 398, 402,
403

set_terminate 404

this ポインター 81, 282, 323

throw 式 157, 387, 396

throw 式 (続き)

ネストされた try ブロック内 389

引き数のマッチング 394

例外の rethrow 396

TIME マクロ 233

try キーワード 387

try ブロック 387

関数定義で 182

ネストされた 389

typedef 指定子 47

および型互換性 53

可変長配列の使用 97

クラス宣言 273

修飾された型名 271

メンバーへのポインター 281

ローカル型名 273

typeid 演算子 119

typeof 演算子 23, 133

U

undef プリプロセッサ・ディレクティブ
228

unexpected 関数 387, 402, 403

set_unexpected 404

unsigned 型指定子

char 55

int 57

long 57

long long 57

short 57

using 宣言 247, 307, 316

メンバー関数の多重定義 308

メンバー・アクセスの変更 309

using ディレクティブ 246

UTF-16、UTF-32 17

u-literal、U-literal 17

V

virtual

関数指定子 102

基底クラス 301, 312, 317

void 58

関数定義で 182, 184

引き数型 184

ポインター 162, 163

volatile

修飾子 79, 81

メンバー関数 278

W

warning プリプロセッサ・ディレクティブ
230

wchar_t 型指定子 30, 55, 57
while ステートメント 211

__VA_ARGS__ 224, 227
~ (ビット単位否定演算子) 129
^ (ビット単位排他 OR 演算子) 147
^= (複合代入演算子) 154
¥ エスケープ文字 15
¥ 継続文字 31, 222

【特殊文字】

! (論理否定演算子) 128
!= (非等価演算子) 146
プリプロセッサ演算子 228
プリプロセッサ・ディレクティブの文
字 222
(マクロ連結) 229
\$ 15, 21
& (アドレス演算子) 129
& (参照宣言子) 103
& (ビット単位 AND 演算子) 147
&& (ラベル値演算子) 133, 202
&& (論理 AND 演算子) 149
&= (複合代入演算子) 154
* (間接演算子) 130
* (乗算演算子) 142
*= (複合代入演算子) 154
+ (加法演算子) 143
+ (単項正演算子) 128
++ (増分演算子) 127
+= (複合代入演算子) 154
, (コンマ演算子) 155
- (減法演算子) 143
- (単項負演算子) 128
-- (減分演算子) 128
-> (矢印演算子) 119
. (ドット演算子) 118
/ (除法演算子) 142
/= (複合代入演算子) 154
:: (スコープ・レゾリューション演算
子) 114
= (単純代入演算子) 153
== (等価演算子) 146
?: (条件演算子) 151
[] (配列添え字演算子) 117
> (より大きい演算子) 144
>= (より大きいまたは等しい演算子) 144
>> (右シフト演算子) 144
>>= (複合代入演算子) 154
< (より小さい演算子) 144
<= (より小さいまたは等しい演算子) 144
<< (左シフト演算子) 144
<<= (複合代入演算子) 154
| (ビット単位包含 OR 演算子) 148
|| (論理 OR 演算子) 149
% (剰余) 143
_Complex_I 417
_Pragma 240
__align 23, 38
__cdecl 197
__func__ 22
__inline__ 23



プログラム番号: 5724-I11

Printed in Japan

SC88-9958-00



日本アイ・ビー・エム株式会社
〒106-8711 東京都港区六本木3-2-12