IBM XL C/C++ for Linux, V13.1

ランゲージ・リファレンス

バージョン 13.1

SA88-5396-00
(英文原典：SC27-4250-00)

IBM XL C/C++ for Linux, V13.1

ランゲージ・リファレンス

バージョン 13.1

SA88-5396-00
(英文原典：SC27-4250-00)

お願い

本書および本書で紹介する製品をご使用になる前に、 563 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM XL C/C++ for Linux, V13.1 (プログラム 5765-J08; 5725-C73)、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。正しいレベルの製品をご使用になるようお確かめください。

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： SC27-4250-00
IBM XL C/C++ for Linux, V13.1
Language Reference
Version 13.1

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

目次

本書について

本書では、C および C++ プログラミング言語の、構文、セマンティクス、および IBM® XL C/C++ Advanced Edition for Linux におけるインプリメンテーションについて説明します。 XL C および XL C++ コンパイラーは、C および C++ プログラミング言語の ISO 準拠により維持されている仕様に準拠していますが、同時にコア言語に多くの拡張機能を組み込んでいます。これらの拡張機能は、特定のオペレーティング環境における使用可能度を高め、その他のコンパイラーとの互換性をサポートし、新しいハードウェア機能をサポートする目的で、インプリメントされました。 例えば、UNIX プラットフォームでは、2 つの開発環境間で最大限の移植性が実現するよう、GNU Compiler Collection (GCC) との互換性を確保するための言語構造体が多数追加されています。

本書の対象読者

本書は、C または C++ でのアプリケーション・プログラミングの経験を既にお持ちのユーザーのための解説書です。 C または C++ の新規ユーザーも、言語および XL C/C++ に固有の特徴を知るために本書を利用することができます。ただし、この解説書は、プログラミングの概念を教えることも、特定のプログラミング手法を向上させることも目的としていません。

本書の読み方

特に断りがなければ、この解説書の説明文はすべて、C と C++ 両方の言語を対象にしています。 2 つの言語に違いがある場合、違いは、限定テキストと他のグラフィカル・エレメントによって示されます (使用される規則については、下記を参照してください)。

本書には、標準の機能とインプリメンテーション固有の機能の両方が含まれていますが、以下のトピックは含まれていません。

- 標準の C および C++ ライブラリー関数およびヘッダー。 標準の C および C++ ライブラリーについては、ご使用のオペレーティング・システムの資料を参照してください。
- マルチスレッド化プログラムを書くための構造体。例えば、 OpenMP ディレクティブおよび関数、および POSIX Pthread 関数など。OpenMP の構成に関する参照資料については、「*XL C/C++ コンパイラー・リファレンス*」を参照してください。Pthread ライブラリー関数についての資料は、Linux の資料を参照してください。
- コンパイラー・プラグマ、事前定義マクロ、および組み込み関数。 「*XL C/C++ コンパイラー・リファレンス*」に説明があります。

本書の構成

本書は、ISO 規格の言語仕様の構成に概ね従うように編成されており、トピックは、類似した見出しでまとめられています。

- 1 章から 8 章は、C と C++ の両方に共通の言語エレメント、すなわち、字句エレメント、データ型、宣言、宣言子、型変換、式、演算子、ステートメント、関数などについて説明しています。これらの章全体で、標準機能と拡張機能の両方が説明されています。
- 第 9 章から第 16 章では、クラス、多重定義、継承、テンプレート、例外処理を含む標準の C++ 機能のみについて説明します。
- 第 17 章では、プリプロセッサーに対するディレクティブについて説明します。
- 第 18 および 19 章では、各言語によってサポートされるすべての拡張機能の要約リストが記載されています。

規則

活字の規則

以下の表では、IBM XL C/C++ for Linux, V13.1 の資料で使用されている活字の規則について説明します。

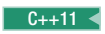
表 1. 活字の規則

書体	意味	例
太字	小文字のコマンド、実行可能ファイル名、コンパイラー・オプション、およびディレクティブ。	コンパイラーには、さまざまな C/C++ 言語レベルおよびコンパイル環境をサポートするために、**xlc** と **xlC** (**xlc++**) という基本呼び出しコマンドとその他のいくつかのコンパイラー呼び出しコマンドが備わっています。
イタリック	パラメーターまたは変数。実際の名前と値はユーザーによって提供されます。イタリックは新規用語の導入にも使用されます。	要求された *size* よりも大きいものを戻す場合には、 *size* パラメーターの更新を確認してください。
下線	コンパイラー・オプションまたはディレクティブのパラメーターのデフォルト設定。	nomaf | maf
モノスペース	プログラミング・キーワードおよびライブラリー関数、コンパイラー・ビルトイン、プログラム・コードの例、コマンド・ストリング、またはユーザー定義の名前。	myprogram.c をコンパイルおよび最適化するには、`xlc myprogram.c -O3` と入力します。

限定を示すエレメント (アイコン)

本書に記述されているフィーチャーの大半は、C と C++ 言語の両方に適用されます。あるフィーチャーが 1 つの言語に限定される場合、あるいは言語間で機能が異なる場合の言語エレメントの説明では、以下のように、アイコンを使用してテキストのセグメントを説明します。

表 2. 限定を示すエレメント

修飾子/アイコン	意味
C のみ、または C のみの始まり   C のみの終わり	このテキストは C 言語のみでサポートされているフィーチャーを記述しています。または、C 言語に特定の振る舞いを記述しています。
C++ のみ、または C++ のみの始まり   C++ のみの終わり	このテキストは C ++ 言語のみでサポートされているフィーチャーを記述しています。または、C++ 言語に特定の振る舞いを記述しています。
IBM の拡張機能、または IBM の拡張機能の始まり   IBM の拡張機能の終わり	テキストは、標準の言語仕様に対する IBM 拡張機能であるフィーチャーを説明します。
C11、または C11 の始まり   C11 の終わり	このテキストは、C11 の一部として標準 C に導入されるフィーチャーを記述しています。
C++11、または C++11 の始まり   C++11 の終わり	このテキストは、C++11 の一部として標準 C++ に導入されるフィーチャーを記述しています。

構文図

本書中では、ダイアグラムは XL C/C++ 構文を図示します。このセクションは、これらのダイアグラムの解釈と使用に役立ちます。

- 構文図は線のパスに沿って、左から右、上から下へと読んでいきます。

  ▶▶── 記号は、コマンド、ディレクティブ、またはステートメントの開始を示します。

  ──▶ 記号は、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。

►── 記号は、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。

──►◄ 記号は、コマンド、ディレクティブ、またはステートメントの終了を示します。

完結したコマンド、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であるフラグメントは、|── 記号で始まり ──| 記号で終わります。

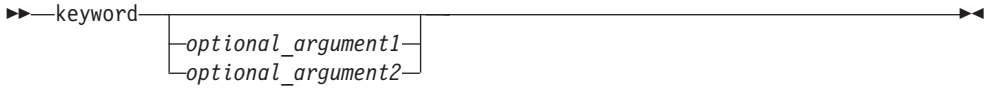
- 必須項目は、次のように横線 (メインパス) 上に表示されます。

- オプショナル項目は、次のようにメインパスの下側に表示されます。

- 2 つ以上の項目から選択できる場合は、縦に重ねて表示されます。

  項目の中から 1 つを*選択しなければならない* 場合は、スタックの 1 つの項目がメインパスに表示されます。

  項目の 1 つを選択することがオプショナルの場合は、スタック全体がメインパスの下に表示されます。



- 主線の上にある左に戻る矢印 (反復矢印) は、スタックされた項目から複数個選択できること、あるいは単一の項目を繰り返すことができることを示します。区切り文字も示されます (それがブランク以外の場合)。

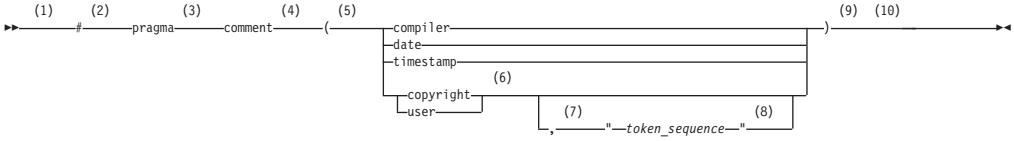
- デフォルトの項目はメインパスの上に表示されます。

- キーワードは、イタリックでない文字で示され、示されているとおりに入力する必要があります。
- 変数は、イタリック体の小文字で示されます。変数は、ユーザー指定の名前や値を表します。

- 句読記号、括弧、算術演算子、または他のそのような記号が表示されている場合は、構文の一部として入力する必要があります。

構文図の例

以下の構文図の例は、**#pragma comment** ディレクティブの構文を示したものです。



注:

1 これが構文図の始まりです。

2 記号 # が最初に示される必要があります。

3 キーワード pragma が # 記号に続いて示される必要があります。

4 プラグマの名前 comment が、キーワード pragma に続いて示される必要があります。

5 左括弧を指定する必要があります。

6 コメント・タイプは、示されているタイプ compiler、date、timestamp、copyright、または user の 1 つとしてのみ入力する必要があります。

7 コメント・タイプ copyright または user とオプショナル文字ストリングとの間にコンマを 1 つ入れる必要があります。

8 コンマの後に文字ストリングが続いている必要があります。文字ストリングは二重引用符で囲む必要があります。

9 右括弧が必要です。

10 これが、構文図の終わりです。

以下の **#pragma comment** ディレクティブの例は、上記の図に従って、構文的に正しいものです。

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

本書の例

本書の例は、特に断りのない限り、単純な形式でコーディングされており、ストレージの節約、エラーのチェック、高速パフォーマンスの実現、特定の成果を達成するために使用可能なすべての方法の提示などの試みはなされていません。

インストール情報の例は、*例* または*基本例* としてラベル付けられています。*基本例* は、基本インストールまたはデフォルト・インストール時に実行する手順の説明用です。例はほとんど変更せずに、または全く変更せずに使用できます。

関連情報

以下のセクションでは、XL C/C++ に関連した情報を説明します。

IBM XL C/C++ 情報

XL C/C++ は、以下の形式で製品資料を提供しています。

- README ファイル

  README ファイルには、製品情報に対する変更と訂正も含め、最新の情報が含まれています。 README ファイルは、デフォルトでは XL C/C++ ディレクトリーと、インストール CD のルート・ディレクトリーにあります。

- インストール可能な man ページ

  man ページは製品に準備されているコンパイラー呼び出しとすべてのコマンド行ユーティリティーに対して提供されています。 man ページのインストールおよびアクセスについての指示は、「*IBM XL C/C++ for Linux, V13.1 インストール・ガイド*」に記載されています。

- インフォメーション・センター

  検索機能が完備された HTML ベースの資料は、次の Web サイトで参照できます。http://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.0/com.ibm.compilers.linux.doc/welcome.html

- PDF 文書

  PDF 文書は、デフォルトでは /opt/ibm/xlC/13.1.0/doc/*LANG*/pdf/ ディレクトリーにあります。ここで *LANG* は en_US、zh_CN、 または ja_JP です。 PDF ファイルは、以下の Web サイト http://www.ibm.com/support/docview.wss?uid=swg27036675でも入手できます。

  以下のファイルは、XL C/C++ 製品資料のフル・セットを構成しています。

表 3. XL C/C++ PDF ファイル

文書タイトル	PDF ファイル名	説明
IBM XL C/C++ for Linux, V13.1 インストール・ガイド, SA88-5404-00	install.pdf	XL C/C++ のインストール方法と基本的なコンパイルおよびプログラム実行のための環境の構成方法に関する情報が含まれています。
IBM XL C/C++ for Linux, V13.1 はじめに, SA88-5392-00	getstart.pdf	XL C/C++ 製品の概要と、環境のセットアップと構成、プログラムのコンパイルとリンク、およびコンパイル・エラーのトラブルシューティングに関する情報が含まれています。
IBM XL C/C++ for Linux, V13.1 コンパイラー・リファレンス, SA88-5388-00	compiler.pdf	さまざまなコンパイラー・オプション、プラグマ、マクロ、環境変数、および組み込み関数 (並列処理に使用されるものを含む) についての情報が含まれます。
IBM XL C/C++ for Linux, V13.1 ランゲージ・リファレンス, SA88-5396-00	langref.pdf	移植性および一般的規格への準拠についての言語拡張機能も含め、 IBM によってサポートされる C および C++ プログラミング言語に関する情報が記載されています。

表 3. XL C/C++ PDF ファイル (続き)

文書タイトル	PDF ファイル名	説明
IBM XL C/C++ for Linux, V13.1 最適化およびプログラミング・ガイド, SA88-5402-00	proguide.pdf	アプリケーションの移植、 Fortran コードによる言語間呼び出し、ライブラリー開発、アプリケーションの最適化および並列処理、および XL C/C++ 高性能ライブラリーなどの高度なプログラミング上のトピックに関する情報が記載されています。

PDF ファイルを読むには、Adobe Reader を使用します。Adobe Reader をお持ちでない場合は、Adobe の Web サイト (http://www.adobe.com) からダウンロードできます (ライセンス条項に従う必要があります)。

IBM Redbooks® 資料、ホワイト・ペーパー、チュートリアル、資料の正誤表、その他の記事など、XL C/C++ に関連する詳細は、次の Web サイトから入手できます。

http://www.ibm.com/support/docview.wss?uid=swg27036675

注: 資料の正誤表は、インフォメーション・センターの英語版にのみ反映されます。

パフォーマンス、生産性、および移植性の向上に関する情報は、 C/C++ café (http://www.ibm.com/software/rational/cafe/community/ccpp) を参照してください。

標準および仕様

XL C/C++ は、以下の標準および仕様をサポートするように設計されています。本情報に含まれているいくつかの機能に関する正確な定義については、これらの標準を参照できます。

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*、別名 *C89*。
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*、別名 *C99*。
- *Information Technology - Programming languages - C, ISO/IEC 9899:2011*、別名 *C11*。(部分サポート)
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*、別名 *C++98*。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*、別名 *標準 C++*。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*、別名 *C++11*。(部分サポート)
- *Information Technology - Programming languages - Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769.* このドラフトの技術レポートは、C 標準委員会によって承認されており、 http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf で入手可能です。

- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. このドラフトの技術レポートは、C 標準委員会に提出されてあり、 http://www.open-std.org/JTC1/SC22/WG21/www/docs/papers/2005/n1836.pdf で入手可能です。
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. ベクトル処理テクノロジーをサポートするための、このベクトル・データ型の仕様はサイト http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf で使用可能です。
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- http://www.openmp.org で使用可能な「*OpenMP Application Program Interface Version 4.0 (部分サポート)*」。

その他の IBM 情報

- *ESSL for AIX® V5.1/ESSL for Linux on POWER V5.1 Guide and Reference* は、Web ページ Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL で入手できます。

その他の情報

- *Using the GNU Compiler Collection* は http://gcc.gnu.org/onlinedocs で入手できます。

テクニカル・サポート

追加の技術サポートを http://www.ibm.com/support/entry/portal/overview/software/rational/xl_c~c++_for_linuxの XL C/C++ のサポート・ページから利用することができます。このページは、選択された大規模な技術情報および他のサポート情報に対する検索機能を備えたポータルを提供します。

必要なものが見つからない場合には、compinfo@ca.ibm.com に E メールを出して問い合わせることができます (英文でのみ対応)。

XL C/C++ に関する最新の情報に関しては、http://www.ibm.com/software/products/us/en/xlcpp-linux/にある製品情報サイトをご覧ください。

第 1 章 スコープとリンケージ

スコープ とは、プログラム・テキストの中の、あるエンティティーを参照するための名前を修飾なしで使用できる可能性がある最大の領域です。すなわち、その名前が有効になりうる最大の領域です。広い意味で言えば、スコープとは、エンティティー名の意味を区分するための一般的なコンテキストです。コンパイラーは、スコープ用の規則とネーム・レゾリューション用の規則を組み合わせることによって、ID の参照がファイル内の任意のポイントで正しいかどうかを判別することができます。

宣言のスコープと ID の可視性は、互いに関連していますが、それぞれ異なる概念です。スコープは、プログラムの中の宣言の可視性を制限できるメカニズムのことです。 ID の *可視性* は、 ID に関連付けられたオブジェクトに正しくアクセスするためのプログラム・テキストの領域です。スコープは可視性を超えることがありますが、可視性がスコープを超えることはありません。重複した ID が内側と外側の宣言領域で使われた結果、外側の宣言領域のオブジェクトが隠蔽されたとき、スコープは可視性を超えます。重複した ID のスコープ (2 番目のオブジェクトの存続期間) が終わるまで、オリジナルの ID を使用して最初のオブジェクトにアクセスすることはできません。

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

リンケージ とは、1 つの名前を複数の変換単位間、または単一の変換単位内で使用すること、または使用できることを指します。*変換単位* とは、ソース・コード・ファイルと、 `#include` ディレクティブでプリプロセスした後に組み込まれるヘッダー・ファイルおよび他のソース・ファイルすべてを合わせたものです。ただし、条件付きプリプロセス・ディレクティブのためにスキップされたソース行は含まれません。リンケージによって、ID の各インスタンスは特定の 1 つのオブジェクトまたは関数に正しく関連付けられます。

スコープとリンケージは、スコープはコンパイラーのためのものであり、リンケージはリンカーのためのものであることで区別できます。ソース・ファイルをオブジェクト・コードに変換するとき、コンパイラーは、外部リンケージを持つ ID を追跡し、最終的には、それをオブジェクト・ファイル内のテーブルに保管します。リンカーは、それによって、どの名前が外部リンケージを持つかを判別することができますが、内部リンケージを持つか、リンケージを持たないかは判別しません。

異なるタイプのスコープの区別については、2 ページの『スコープ』で説明しています。リンケージのタイプの違いについては、8 ページの『プログラム・リンケージ』で説明しています。

関連資料:

54 ページの『ストレージ・クラス指定子』

313 ページの『第 9 章 名前空間 (C++ のみ)』

スコープ

ID のスコープ は、 ID が該当のオブジェクトを参照するために使われる可能性があるプログラム・テキストの最大の領域です。 C++ では、参照されるオブジェクトは固有でなければなりません。ただし、そのオブジェクトにアクセスするための名前、すなわち、ID 自体は再使用できます。ID の意味は、その ID が使用されるコンテキストに依存します。スコープとは、名前の意味を区分するための一般的なコンテキストです。

ID のスコープは不連続である可能性があります。中断の原因の 1 つに、同じ名前を再使用して別のエンティティーを宣言したことにより、包含された宣言領域 (内側) と包含する宣言領域 (外側) が作成された場合があります。このように、どこで宣言するかは、スコープに影響を与える要因になります。不連続スコープが可能であることが、*情報の隠蔽* と呼ばれる手法の基礎です。

C で採用されているスコープの概念は、C++ で拡張され、改良されました。下表に、スコープの種類と用語のわずかな違いを示します。

表 4. 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	グローバル名前空間
	名前空間
	クラス

すべての宣言において、ID は、初期化指定子の前にスコープに入れられます。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・スコープをもっています (グローバル・スコープではなく)。

関連資料:

313 ページの『第 9 章 名前空間 (C++ のみ)』

ブロック/ローカル・スコープ

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

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

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もっている場合、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、その名前が使われているすぐの囲みスコープから始まって、各囲みスコープの外側に進みます。ネーム・レゾリューションでスコープが検索される順序によって、情報の隠蔽という現象が起きます。外側のスコープ内の宣言は、ネストされたスコープ内の同じ ID の宣言によって隠蔽されます。

関連資料:

234 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。

実際のラベルが出現する前に、`goto` ステートメントの中にラベルを使用することができます。

関連資料:

231 ページの『ラベル付きステートメント』

関数プロトタイプ・スコープ

関数宣言 (*関数プロトタイプ* とも呼ばれる) 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。

関連資料:

264 ページの『関数宣言』

ファイル/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、名前にはファイル・スコープ があります。ファイル・スコープと内部リンケージ付きの名前は、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++ *グローバル・スコープ* または*グローバル名前空間スコープ* 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名前空間スコープです。 ID の宣言が、ブロック、名前空間、およびクラスすべての外側に現れる場合、その名前は*グローバル名前空間スコープ* を持ちます。

グローバル名前空間スコープと内部リンケージ付きの名前は、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 (名前空間) 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。

ユーザー定義の名前空間は、名前空間定義を使用してグローバル・スコープ内にネストすることができ、個々のユーザー定義名前空間は、グローバル・スコープとは別の固有のスコープです。 C++

関連資料:

313 ページの『第 9 章 名前空間 (C++ のみ)』

8 ページの『内部リンケージ』

57 ページの『extern ストレージ・クラス指定子』

C におけるスコープの例

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

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

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

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

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

クラス・スコープ (C++ のみ)

メンバー関数内で宣言された名前は、そのスコープがメンバー関数のクラスの終わりまで及んでいるか、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りまで、または終りを超える場合、そのクラスのメンバー定義によって定義された領域は、そのクラスのスコープに含まれます。そのクラスの外で字句的に定義されたメンバーも、このスコープに含まれます。さらに、宣言のスコープは、メンバー定義内の該当の ID に続く宣言子のどの部分も含みます。

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

- そのクラスのメンバー関数内
- そのクラスから派生したクラスのメンバー関数内
- そのクラスのインスタンスに適用された . (ドット) 演算子の後
- そのクラスから派生したクラスのインスタンスに適用された . (ドット) 演算子の後 (派生したクラスが名前を隠さない場合)
- 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さない場合)
- 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

関連資料:

345 ページの『第 11 章 クラス (C++ のみ)』
349 ページの『クラス名のスコープ』
359 ページの『メンバー・スコープ』
376 ページの『フレンド・スコープ』
388 ページの『基底クラス・メンバーのアクセス制御』
173 ページの『スコープ解決演算子 :: (C++ のみ)』

ID の名前空間

名前空間は、ID を使用できるさまざまな構文コンテキストです。同じコンテキストおよび同じスコープ内では、ID は、エンティティーを一意的に識別しなければなりません。ここで使用する用語*名前空間* は、C および C++ に適用され、C++ 名前空間言語機能を指すものではありません。 コンパイラーは、*名前空間* を設定して、種類が異なるエンティティーを参照する ID を区別します。異なる名前空間内の同一の 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 がその名前空間内で固有である限り、同じ ID で異なる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、その名前空間をあいまいさなしに解決します。

次の 4 つの各名前空間内では、ID を固有にする必要があります。

- 以下の型のタグ は、単一スコープ内で固有にする必要があります。
  - 列挙型
  - 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、またはクラス型内で固有にする必要があります。
- *ステートメント・ラベル* には、関数スコープがあり、1 つの関数内で固有にする必要があります。
- 次に示すほかのすべての*通常 ID* は、単一のスコープ内で固有にする必要があります。
  - C 関数名 (C++ 関数名は、多重定義できる)
  - 変数名
  - 関数仮パラメーターの名前
  - 列挙型定数
  - `typedef` 名

ID は、囲まれたプログラム・ブロックを使用して、同じ名前空間内で再定義できます。

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 つの異なる名前空間にあります。次の例の `student` という名前の項目間では、名前の競合は発生しません。

```
int get_item()
{
   struct student        /* structure tag */
   {
```

```
    char student[20];  /* structure member */
    int section;
    int id;
  } student;           /* structure variable */

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

コンパイラーは、student が発生するたびにプログラム内のコンテキストに応じて解釈します。キーワード struct の後で student が現れるときは、構造体タグです。student 型を定義するブロック内で現れるときは、構造体メンバー変数です。構造体定義の最後に現れるときは、構造体変数を宣言します。そして、goto ステートメントの後に現れるときは、ラベルです。

名前の隠蔽 (C++ のみ)

クラス名または列挙名がスコープ内にあって、隠蔽されていなければ、それは*可視*です。クラス名または列挙名は、その同じ名前を (オブジェクト、関数、または列挙子として) ネストされた宣言領域または派生クラスの中で明示宣言を行うことによって、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、名前空間 A のメンバーであると想定します。また、名前空間 A のメンバーは、宣言の使用により、名前空間 B で可視であると想定します。名前空間 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;
}
```

上記の例の出力は、次のとおりです。

```
int
```

名前空間 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。

関連資料:

345 ページの『第 11 章 クラス (C++ のみ)』

357 ページの『メンバー関数』

359 ページの『メンバー・スコープ』

313 ページの『第 9 章 名前空間 (C++ のみ)』

プログラム・リンケージ

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

- 『内部リンケージ』: ID は、変換単位内でのみ見えます。
- 9 ページの『外部リンケージ』: ID は、他の変換単位内で見えます (参照もできます)。
- 10 ページの『リンケージなし』: ID は、それが定義されたスコープ内でのみ見えます。

リンケージはスコープの扱いに影響を与えず、通常の名前検索の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。言語リンケージは C++ コードを、C で書かれたコードにリンクできるようにすることで、C++ と C の間を密接に関係づけることができます。すべての ID は言語リンケージを持ち、デフォルトでは C++ です。言語リンケージは、異なる変換単位間で整合している必要があります。また、 C++ 以外の言語リンケージは、ID は外部リンケージを持つことを意味します。

関連資料:

56 ページの『静的ストレージ・クラス指定子』

57 ページの『extern ストレージ・クラス指定子』

270 ページの『関数のストレージ・クラス指定子』

101 ページの『型修飾子』

無名共用体

内部リンケージ

次の種類の ID は、内部リンケージを持っています。

- `static` と明示的に宣言されたオブジェクト、参照、または関数。
- 指定子 `const` C++11 または `constexpr` C++11 を使用して、名前空間スコープ (または C のグローバル・スコープ) 内で宣言されているが、`extern` と明示的に宣言されておらず、以前に外部結合を持っていると宣言されていない、オブジェクトまたは参照。
- 無名共用体のデータ・メンバー。
- C++ 明示的に `static` と宣言された関数テンプレート
- C++ 名前なし名前空間で宣言された ID

ブロック内部で宣言された関数は、通常外部リンケージをもっています。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、通常外部リンケージをもっています。 static ストレージをもっている変数が、関数の外で定義されている場合、その変数は、内部リンケージをもっていて、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 の直前の宣言が名前空間またはグローバル・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。

外部リンケージ

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

- オブジェクト
- 関数

C 内の ID が extern キーワードを宣言されても、同じ ID を使ったオブジェクトまたは関数の以前の宣言が可視になっている場合は、2 番目の ID は、最初の宣言と同じリンケージを持ちます。例えば、最初にキーワード static を宣言され、後でキーワード extern を宣言された変数または関数は内部リンケージを持ちます。ただし、リンケージを持っていなかった変数または関数が後でリンケージ指定子を宣言されると、その変数または関数は、明示指定されたリンケージを持ちます。

C

C++ 名前空間スコープでは、以下の種類のエンティティーに対する ID は外部リンケージを持ちます。

- 内部リンケージを持たない参照またはオブジェクト
- 内部リンケージを持たない関数
- 名前付きクラスまたは列挙
- typedef 宣言で定義された名前なしクラスまたは列挙
- 外部リンケージを持っている列挙の列挙子
- 内部リンケージを持つ関数テンプレートでないテンプレート
- 名前なし名前空間内で宣言されていない名前空間

クラスの ID が外部リンケージを持っている場合、そのクラスのインプリメンテーションでは、以下のエンティティーに対する ID も外部リンケージを持ちます。

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

C++

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。

- 外部リンケージも内部リンケージも持たない名前
- 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特定のエンティティーの例外あり)
- オブジェクトまたは関数を表さない ID、インクルード・ラベル、列挙子、リンケージを持たないエンティティーを参照する typedef 名、型名、関数仮パラメーター、および C++ テンプレート名 C++

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

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

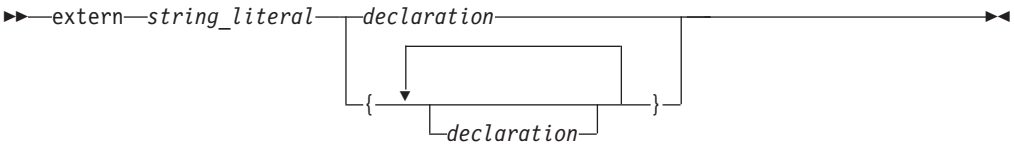
コンパイラーは、外部リンケージをもつ a1 の宣言を許可しません。構造体 A は、リンケージをもっていません。コンパイラーは、外部リンケージをもつ a2 の宣言を許可しません。 A がリンケージをもっていないので、typedef 名 myA は、リンケージをもっていません。

言語リンケージ (C++ のみ)

C++ と C++ 以外のコード・フラグメント間のリンケージは、*言語リンケージ* と呼ばれます。すべての関数の型、関数名、および変数名は言語リンケージを持ちます。これは、デフォルトでは C++ です。

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

リンケージ指定の構文



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

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

"C" C プロシージャーへのリンケージを示します。

C++ の考慮が必要になる前に書かれた共有ライブラリーを呼び出すには、extern "C" {} 宣言内に #include ディレクティブが必要です。

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

以下に、C++ から呼び出される C 印刷関数の例を示します。

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

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

名前マングリング (C++ のみ)

名前マングリングとは、関数名と変数名を固有の名前にエンコードして、リンカーが言語の中の共通の名前を分離できるようにすることです。型名もマングルすることができます。名前マングリングは、多重定義機能および異なるスコープ内での可視性を促進するために広く使用されています。コンパイラーは、モジュールをコンパイルするときに、関数引数の型のエンコード方式で関数名を生成します。変数が名前空間内にある場合、名前空間の名前が変数名にマングルされ、複数の名前空間に同じ変数名が存在できるようになります。 C++ コンパイラーも C 変数名をマングルし、その C 変数が置かれている名前空間を識別できるようにします。

名前をマングルする方法は、ソース・コードのコンパイルに使用したオブジェクト・モデルによって異なります。つまり、あるオブジェクト・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、マングルされた名前は、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、マングルされた名前とは異なります。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 モジュールを、C++ コンパイラーでコンパイルされたライブラリーまたはオブジェクト・ファイルにリンクする場合、名前マングリングは望ましくありません。C++ コンパイラーが関数の名前をマングリングしないようにするには、以下の例に示したように、1 つ以上の宣言に extern "C" リンケージ指定子を適用することができます。

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

この宣言は、関数 f1、f2、および f3 の参照をマングルしないようにコンパイラーに指示しています。

C++ で定義された関数をマングリングしないようにして、その関数を C から呼び出せるようにするためにも extern "C" リンケージ指定子を使用することができます。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 extern 宣言では、最も内側の extern 指定が優先します。

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

この例では、func は C++ リンケージを持ちます。

第 2 章 字句エレメント

字句エレメント とは、ソース・ファイルで使用できる個別の文字、または文字グループのことです。このトピックでは、以下について、基本字句エレメントと C および C++ プログラミング言語の規則を説明します。

トークン

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

- キーワード
- ID
- リテラル
- 区切り子と演算子

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

キーワード

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

C および C++ 言語のキーワード

表 5. C および C++ のキーワード

auto[1]	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern[2]	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

表 5. C および C++ のキーワード (続き)

注:

1. C++11 C++11 では、キーワード auto は、ストレージ・クラス指定子として使用されなくなりました。代わりに、型指定子として使用されます。これは、初期化指定子の式の型から auto 変数の型を推定できます。 C++11
2. C++11 キーワード extern は、以前にはストレージ指定子またはリンケージ指定の一部として使用されていました。 C++11 標準では、3 番目の使用法が追加されており、このキーワードを使用して明示的インスタンス生成宣言を指定できます。 C++11

C

C 言語のキーワードのみ

C99 および C11 レベルの標準 C 言語では、次のキーワードも予約されています。

表 6. C99 および C11 のキーワード

_Bool	_Static_assert[1]
_Complex	inline
_Generic[1]	restrict
_Imaginary[2]	
_Noreturn[1]	

注:

1. C11 このキーワードは、C11 言語レベルで導入されました。 C11
2. キーワード _Imaginary は将来の使用に備えて予約されています。複素数関数には、_Complex を使用してください。詳細については、複素数リテラル を参照してください。

C

C++

C++ 言語のキーワードのみ

C++ 言語では、次のキーワードも予約されています。

表 7. C++ のキーワード

asm	dynamic_cast	new	this
bool	decltype	nullptr	throw
catch	explicit	operator	true
class	export	private	try
	false	protected	typeid
	friend	public	typename
const_cast	inline	reinterpret_cast	using
constexpr	mutable	static_assert	virtual
delete	名前空間	static_cast	wchar_t
		テンプレート	

C++

言語拡張のキーワード (IBM 拡張)

XL C/C++ は、標準の言語キーワードのほかに、言語拡張で使用するために以下のキーワードを予約しています。

表 8. C 言語 および C++ 言語拡張のためのキーワード

__alignof	__extension__	__static_assert[3]
__alignof__	__label__	__volatile
__asm (C のみ)	__imag__	__volatile__
__asm__ (C のみ)	__inline__[1]	__thread
_attribute__	_Noreturn[4]	typeof[2]
__attribute	pixel[6]	
bool (C のみ)[6]	__pixel[6]	__typeof__
__complex__	__real__	vector[6]
__const__	__restrict	__vector[6]
	__restrict__	
__decltype[5]	__signed__	
	__signed	

注:

1. C __inline__ キーワードは、インライン関数用に GNU C のセマンティクスを使用します。詳しくは、 274 ページの『インライン関数のリンケージ』を参照してください。 C
2. C typeof は、**-qkeyword=typeof** が有効になっている場合にのみ認識されます。 C
3. C __static_assert は、C++11 標準との互換性用の C 言語拡張のキーワードです。 C
4. C++ _Noreturn は、C11 標準との互換性を保つための、C++ 言語拡張のキーワードです。 C++
5. C++ __decltype は、C++ 言語拡張のためのキーワードです。これにより、すべての language レベルで C++11 decltype 機能を使用できるようになります。 C++
6. これらのキーワードは、ベクトル・サポートが使用可能となっている場合に、ベクトル宣言コンテキスト内でのみ認識されます。

C++ XL C++ は、C99 との互換性を確保するための言語拡張機能として以下のキーワードを予約しています。

表 9. C99 関連の C++ 言語拡張のためのキーワード

_Complex
_Imaginary[1]
_Pragma
restrict

表 9. C99 関連の C++ 言語拡張のためのキーワード (続き)

注:

1. キーワード `_Imaginary` は将来の使用に備えて予約されています。複素数関数には、`_Complex` を使用してください。詳細については、複素数リテラル を参照してください。

C++

拡張キーワードが有効なコンパイル・コンテキストに関する詳細情報は、各キーワードについて説明しているセクションに記述されています。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照

67 ページの『ベクトル型 (IBM 拡張)』

ID

ID は、次の言語エレメントに名前を付けます。

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

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

ID の文字

ID の先頭文字は、文字または _ (下線) 文字でなければなりません。ただし、下線で始まる ID は、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。

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。 ID の名前の一部に小文字の a が指定された場合、それを大文字の A に置き換えることはできません。小文字を使用しなければなりません。

基本ソース文字セット以外の文字および数字のユニバーサル文字名は、 C++ および C99 言語レベルで許可されています。 C++ C++ では、ユニバーサル文字名サポート用の **-qlanglvl=ucs** オプションを使用してコンパイルする必要があります。

IBM ドル記号は、**-qdollar** 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 の名前に使用することができます。

予約 ID

2 つまたは 1 つの下線から始まり、その後に大文字の英字が続く ID は、コンパイラー用にグローバルに予約されています。

C 単一の下線で始まる ID は、通常の名前空間内およびタグ・名前空間内のファイル・スコープに関する ID として予約されています。

C++ 単一の下線で始まる ID は、グローバル名前空間内で予約されています。

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

__func__ 事前定義 ID

C99 事前定義 ID __func__ は、関数の名前をその関数内で使用する目的で使用可能にします。__func__ は、各関数定義の左中括弧の直後にコンパイラーによって暗黙的に宣言されます。その結果、動作は以下の宣言が行われた場合と同じです。

```
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。関数名はマングルされません。

C++ 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 がクラス X のメンバー関数である場合、foo の事前定義 ID は X::foo です。foo が main という本体内で定義されている場合、foo の事前定義 ID は main::X::foo です。

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

デバッグの目的で、 __func__ ID を明示的に使用して、それが現れる関数の名前を戻させるようにすることができます。次に例を示します。

```
#include <stdio.h>

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

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

このプログラムの出力は、次のようになります。

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

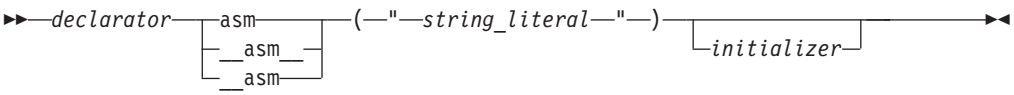
関数定義の中で assert マクロが使用されると、このマクロは、囲む関数の名前を標準エラー・ストリームに追加します。

アセンブリー・ラベル (IBM 拡張)

コンパイラーは、ソース・コードの非静的外部変数名および関数名を、オブジェクト・ファイルおよび出力されるすべてのアセンブリー・コードにコンパイラーが生成する名前にバインドします。コンパイラーは、GCC との互換性を保つために、Standard C および C++ に拡張機能をインプリメントして、グローバル変数または関数プロトタイプの宣言にアセンブリー・ラベル を適用することにより、オブジェクト・ファイルおよびアセンブリー・コードで使用される名前を指定できるようにします。また、通常は下線が関数または変数の名前の前に付加されるシステムでも、下線で始まらない名前を定義することができます。

C++ アセンブリー・ラベルは、メンバー関数、およびグローバル・ネームスペース以外のネームスペースで宣言された関数および変数で使用することができます。

アセンブリー・ラベルの構文



string_literal は、指定されたオブジェクトまたは関数にバインドされる有効なアセンブリー名です。関数宣言に適用されるラベルの場合、この名前はコンパイル単位で定義された既存の関数を指定している必要があります。定義されていない場合、リンク時にエラーが発生します。変数宣言に適用されるラベルの場合は、その他の定義は必要ありません。

次に、アセンブリー・ラベル指定の例を示します。

```
void func3() __asm__("foo3");
int i __asm("abc");
char c asm("abcs") = 'a';
```

C++ 多重定義された関数を区別するために、XL C++ はオブジェクト・ファイル内の関数名をマングルします。したがって、アセンブリー・ラベルを使用して関数名をマップする場合、ターゲットのマングルされた関数名を使用する必要があります。さらに、変数に指定するアセンブリー・ラベル名がその他のマングルされた名前と競合していないことを確認する必要があります。あるいは、C リンケージを持つものとして宣言することにより、ターゲット関数で名前のマングリングが行われないようにすることができます。詳しくは、11 ページの『名前マングリング (C++ のみ)』を参照してください。

以下の制約事項は、アセンブリー・ラベルの使用に適用されます。

- ローカル変数や静的変数にアセンブリー・ラベルを指定することはできません。
- 同じコンパイル単位で、同じアセンブリー・ラベル名を複数の ID (C++ の場合、これはマングリング後の名前です) に適用することはできません。
- ラベル名と ID 名が同じ変数宣言または関数宣言で使用されている場合を除き、同じコンパイル単位で、アセンブリー・ラベル名は、その他のグローバル ID 名 (C++ の場合は、マングリング後の名前) と同じ名前にすることはできません。
- アセンブリー・ラベルを `typedef` 宣言の中で指定することはできません。
- アセンブリー・ラベルは、直前の **#pragma map** ディレクティブが異なる変数や関数に指定した名前と同じ名前にすることはできません。同様に、 **#pragma map** ディレクティブで指定されたマップ名は、直前のアセンブリー・ラベルが異なる変数や関数に指定した名前と同じ名前にすることはできません。
- **#pragma map** ディレクティブが直前の変数宣言や関数宣言で異なる名前にマップした ID に、アセンブリー・ラベルを適用することはできません。同様に、アセンブリー・ラベルが直前に再マップした ID に、**#pragma map** ディレクティブを指定することはできません。
- C 同じ変数または関数の多重宣言で異なるラベルを適用すると、最初の指定は受け入れられますが、後続のアセンブリー・ラベルはすべて無視され、警告が出されます。
- C++ アセンブリー・ラベルは、以下のどれにも適用することはできません。
  - メンバー変数宣言
  - フレンド宣言
  - テンプレート関数およびメンバー宣言、またはテンプレート内に含まれるすべての宣言
  - 仮想メンバー関数
  - コンストラクターおよびデストラクター

関連資料:

169 ページの『ID 式 (C++ のみ)』

41 ページの『ユニコード規格』

13 ページの『キーワード』

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

263 ページの『関数の宣言と定義』

「XL C/C++ コンパイラー・リファレンス」の #pragma map を参照

288 ページの『alias』

指定したレジスターの変数 (IBM 拡張)

252 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』


「XL C/C++ コンパイラー・リファレンス」の『-qreserved_reg』を参照

リテラル

リテラル定数 という語、すなわち、*リテラル* とは、プログラム内で使われる、変更できない値のことです。C 言語は、名詞*リテラル* の代わりに用語*定数* を使用します。形容詞*リテラル* は、定数の概念に、その定数に関しては値だけを扱うという考え方を追加します。リテラル定数はアドレス指定不可です。つまり、その値はメモリー内のどこかに保管されており、そのアドレスにアクセスする手段はないことを意味します。

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

- 『整数リテラル』
- 25 ページの『ブール・リテラル』
- 26 ページの『浮動小数点リテラル』
- IBM 30 ページの『ベクトル・リテラル (IBM 拡張)』
- 33 ページの『文字リテラル』
- 34 ページの『ストリング・リテラル』

整数リテラル

C++11

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

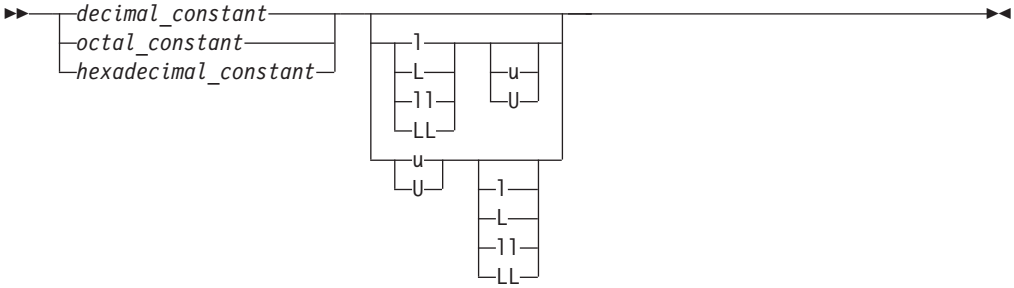
C++11

整数リテラル は、小数点または指数部を持たない数値です。次のように表記できます。

- 10 進整数リテラル
- 16 進整数リテラル
- 8 進整数リテラル

整数リテラルは、基数を指定する接頭部、または型を指定する接尾部を持つことがあります。

整数リテラルの構文



long long 機能

以下の 2 つの long long 機能があります。

- C99 の long long 機能
- C99 以外の long long 機能

IBM これら 2 つの機能の両方に、以下の対応する拡張部分があります。

- 関連する IBM 拡張付き C99 の long long 機能
- C99 以外の IBM long long 拡張

IBM

C99 および C++11 以外の整数リテラルの型

C C99 以外のモードでは、C99 以外の IBM long long 拡張を使用可能にできます。

C++ C99 の long long 機能が効力を持たない場合は、C99 以外の IBM long long 拡張を使用可能にできます。

以下の表に、整数リテラルをリストし、C99 の long long 機能が使用可能になっていない場合に可能なデータ型を示します。

表 10. C99 および C++11[1] 以外の整数リテラルの型

表記	接尾部	プロモーション順序					
		int	unsigned int	long int	unsigned long int	IBM long long int	IBM unsigned long long int
10 進	なし	+		+	+[2]		
8 進、16 進	なし	+	+	+	+		
すべて	u または U		+		+		
10 進	l または L			+	+[2]		

表 10. C99 および C++11[1] 以外の整数リテラルの型 (続き)

表記	接尾部	プロモーション順序					
8 進、16 進	l または L			+	+		
すべて	u または U と l または L の両方				+		
10 進	ll または LL					+	+
8 進、16 進	ll または LL					+	+
すべて	u または U と ll または LL の両方						+

注:

1. long long 機能がいずれも使用可能になっていない場合、整数リテラルの型は、最後の 2 列を除いて、この表にあるすべての型を含みます。
2. IBM unsigned long int 型は、C++98 および C++03 標準では、ここに含まれていません。C++ コンパイラーでは、互換性の目的のみでインプリメンテーションにこの型が含まれています。

C99 および C++11 の整数リテラルの型

C C99 モードでは、C99 の long long 機能が自動的に使用可能になります。

C++ C99 以外の IBM long long 拡張が効力を持たない場合は、C99 の long long 機能を使用可能にできます。

C99 の long long 機能を使用可能にした後、コンパイラーは C99 以外の IBM long long 拡張のすべての機能を持つようになります。範囲外のリテラルは別として、唯一の違いは、u または U を含む接尾部を持たない 10 進整数リテラルに対する特定の型指定の規則です。 C99 以外の IBM long long 拡張では範囲外のリテラルは、関連する IBM 拡張付き C99 の long long 機能では、暗黙の型 long long int または型 unsigned long long int を持つ場合があります。

以下の例は、これらの 2 つの long long モードを使用した場合のコンパイラーの動作の違いを示しています。

```
#include <stdio.h>

int main(){
```

```
    if(0>3999999999-4000000000){
        printf("C99 long long");
    }
    else{
        printf("non-C99 IBM long long extension");
    }
}
```

この例では、値 3999999999 および 4000000000 は大きすぎて 32 ビットの long int 型に収まりませんが、unsigned long 型または long long int 型に収まります。C99 の long long 機能を使用可能にすると、この 2 つの値は long long int 型を持つため、3999999999 と 4000000000 の差はマイナスです。一方、C99 以外の IBM long long 拡張を使用可能にすると、この 2 つの値は unsigned long 型を持つため、その差はプラスです。

C99 および C99 以外の両方の long long 機能が使用不可にされている場合に、以下のいずれかの接尾部を持つ整数リテラルがあると、重大なコンパイル時エラーの原因となります。

- ll または LL
- u または U と ll または LL の両方

IBM 値が long long int 型に収まらない場合、コンパイラーは unsigned long long int 型をそのリテラルに使用する可能性があります。この場合、コンパイラーは値が大きすぎることを示すメッセージを生成します。 IBM

C++11 IBM C++11 標準に厳密に準拠するために、コンパイラーには拡張された整数セーフ動作が導入されているので、プロモーション後に符号付きの値が符号なしの値になることは決してありません。この動作を有効にした後に、接尾部に u または U を含まない 10 進整数リテラルを long long int 型で表現できない場合、コンパイラーはリテラルの値が範囲外であることを示すエラー・メッセージを出します。この拡張された整数セーフ動作が、関連する IBM 拡張付き C99 の long long 機能と C99 の long long 機能の間における唯一の違いです。 IBM C++11

以下の表に、整数リテラルをリストし、C99 の long long 機能が使用可能になっている場合に可能なデータ型を示します。

表 11. C99 および C++11 の整数リテラルの型

表記	接尾部	プロモーション順序					
		int	unsigned int	long int	unsigned long int	long long int	unsigned long long int
10 進	なし	+		+		+	+[1]
8 進、16 進	なし	+	+	+	+	+	+
すべて	u または U		+		+		+
10 進	l または L			+		+	+[1]
8 進、16 進	l または L			+	+	+	+

表 11. C99 および C++11 の整数リテラルの型 (続き)

表記	接尾部	プロモーション順序					
すべて	u または U と l または L の両方				+		+
10 進	ll または LL					+	+[1]
8 進、16 進	ll または LL					+	+
すべて	u または U と ll または LL の両方						+

注:

1. C++11 IBM 拡張された整数セーフ動作が有効な場合、コンパイラーはこの型をサポートしません。

10 進整数リテラル

10 進整数リテラル は 0 から 9 までの数字を使用できます。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 進整数リテラルとしてではなく、8 進整数リテラルとして解釈されます。

10 進整数リテラルの構文

10 進整数リテラルの例は、以下のとおりです。

```
485976
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。次の例を検討してみます。

```
-433132211
+20
```

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、その後に、0 から 9 までの数字と a から f または A から F までの英字の組み合わせが続きます。英字 A (または a) から F (または f) は、それぞれ、値 10 から 15 を表します。

16 進整数リテラルの構文

16 進整数リテラルの例は、以下のとおりです。

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

8 進整数リテラル

8 進整数リテラル は、数字 0 で始まり、その後に 0 から 7 までの数字が続きます。

8 進整数リテラルの構文

8 進整数リテラルの例は、以下のとおりです。

```
0
0125
034673
03245
```

関連資料:

63 ページの『整数型』

154 ページの『整数型変換』

158 ページの『整数および浮動小数点数のプロモーション』

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

555 ページの『C++11 互換性の拡張機能』

ブール・リテラル

C C99 レベルでは、C は、`true` および `false` をヘッダー・ファイル `stdbool.h` の中でマクロとして定義しています。

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

関連資料:

64 ページの『ブール型』

155 ページの『ブール型変換』

浮動小数点リテラル

浮動小数点リテラル は、小数点または指数部を持つ数値です。次のように表記できます。

- 実リテラル
  - 2 進浮動小数点リテラル
  - 16 進浮動小数点リテラル
- 複素数リテラル

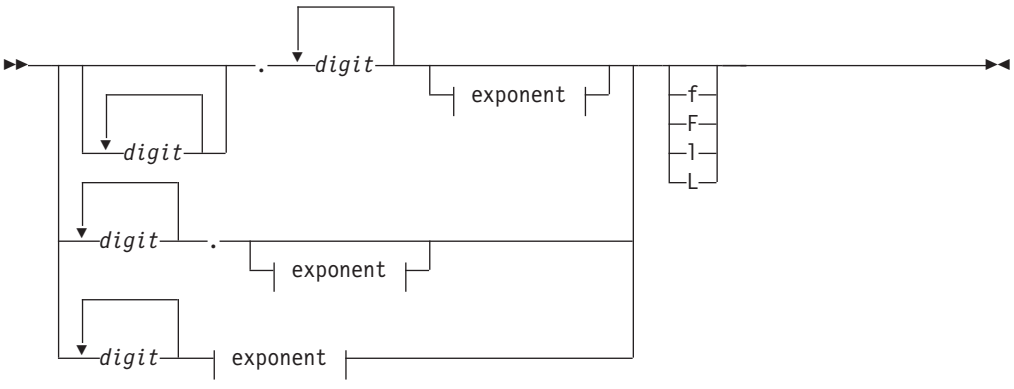
2 進浮動小数点リテラル

実 2 進浮動小数点定数は、次のもので構成されます。

- 整数部分
- 小数点
- 小数部分
- 指数部
- オプションの接尾部

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

2 進浮動小数点リテラルの構文



Exponent:

接尾部 f または F は浮動小数点型を示し、接尾部 l または L は long double 型を示します。接尾部を指定しないと、浮動小数点定数には、double 型が指定されます。

正 (+) または負 (-) 符号を、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。

floating-point constant	値
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

16 進浮動小数点リテラル

実 16 進浮動小数点定数 (C99 の機能) は、次のもので構成されます。

- 16 進数接頭部
- 有効部分
- 2 進指数部
- オプションの接尾部

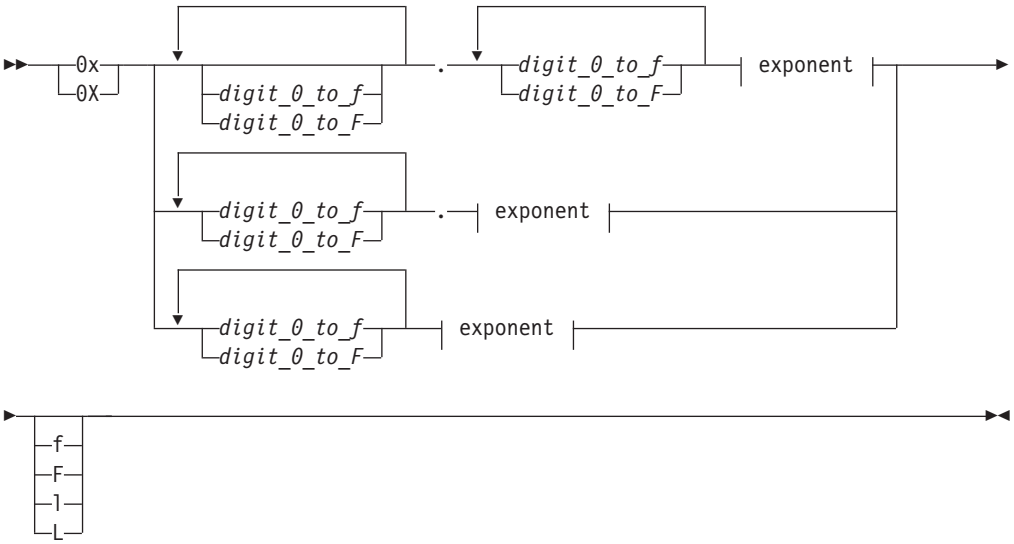
有効部分は有理数を表し、以下のもので構成されます。

- 16 進数字のシーケンス (整数部分)
- オプションの小数部分

オプションの小数部は、ピリオドとその後の 16 進数字のシーケンスです。

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

16 進浮動小数点リテラルの構文



Exponent:

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

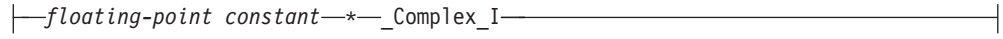
複素数リテラル

C99 標準に導入された複素数リテラルは、実数部と虚数部という 2 つの部分で構成されています。

複素数リテラルの構文

実数部 :

虚数部 :



floating-point constant は、10 進または 16 進浮動小数点定数 (オプションの接尾部を含む) として、ここまでのセクションで述べたいずれの形式ででも指定することができます。

_Complex_I は、complex.h ヘッダー・ファイルの中で定義されているマクロであって、虚数単位 *i*、すなわち、-1 の平方根を表します。

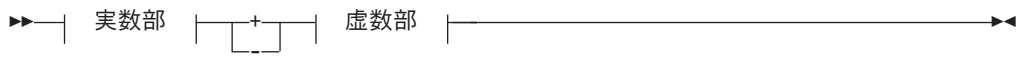
例えば、次の宣言

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

複素数型変数 varComplex を「2.0 + 2.0*i*」の値に初期化します。

IBM また、GNU 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、XL C/C++ を使用すると、複素数の型 (float、double、または long double) を示す標準の接尾部のほかに、接尾部を付けて複素数リテラルの虚数部を示すこともできます。

GNU 接尾部を使用する複素数リテラルの単純化した構文は、次のとおりです。



実数部 :

虚数部 :

floating-point constant は、10 進または 16 進浮動小数点定数 (オプションの接尾部を含む) として、ここまでのセクションで述べたいずれの形式ででも指定することができます。

imaginary-suffix は、接尾部 i、I、j、または J のいずれかであり、虚数単位を表します。

例えば、次の宣言

```
varComplex = 3.0f + 4.0fi;
```

複素数型変数 varComplex を「3.0 + 4.0*i*」の値に初期化します。 IBM

関連資料:

65 ページの『浮動小数点型』

155 ページの『浮動小数点変換』

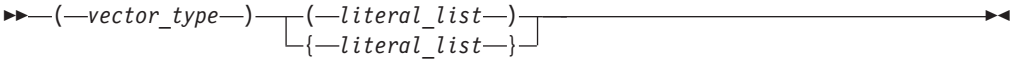
176 ページの『単項式』

複素数浮動小数点型

ベクトル・リテラル (IBM 拡張)

ベクトル・リテラルは、値がベクトル型として解釈される定数式です。ベクトル・リテラルのデータ型は、ベクトル型を括弧で囲んだ形で表され、その値は、ベクトル・エレメントを表す一連の定数式で、括弧または中括弧で囲みます。すべてのベクトル・エレメントの値が同じである場合は、リテラルは、単一の定数式で表すことができます。ベクトル型は、ベクトル・リテラルを使用して初期化できます。

ベクトル・リテラルの構文



literal_list:

vector_type は、サポートされるベクトル型です。 67 ページの『ベクトル型 (IBM 拡張)』に、これらがリストされていますので参照してください。

literal_list は、次の式のどちらかです。

- 単一の式。

  単一の式が小括弧で囲まれている場合は、ベクトルのすべてのエレメントが指定値に初期化されます。単一の式が中括弧で囲まれている場合は、ベクトルの最初のエレメントが指定値に初期化され、ベクトルの残りのエレメントは 0 に初期化されます。
- 式のコンマで区切られたリスト。ベクトルの各エレメントは、それぞれの指定値に初期化されます。

  定数式の数は、ベクトルの型と、中括弧または小括弧のどちらで囲むかによって決まります。

  式のコンマで区切られたリストを中括弧で囲む場合、定数式の数はベクトルのエレメントの数と同じか、それよりも少なくすることができます。定数式の数がベクトルのエレメントの数よりも少ない場合、指定されていないエレメントの値は 0 になります。

  式のコンマで区切られたリストを小括弧で囲む場合、以下のように定数式の数はベクトルのエレメントの数と同じでなければなりません。

  2 `vector long long`、`vector bool long long`、 および `vector double` 型の場合。

  4 `vector int`、`vector long`、および `vector float` 型の場合。

  8 `vector short` および `vector pixel` 型の場合。

  16 `vector char` 型の場合。

次の表は、サポートされているベクトル・リテラルを示すとともに、コンパイラーがこれらのリテラルを解釈してそれぞれの値を決定する方法について説明したものです。

表 12. ベクトル・リテラル

構文	**コンパイラーによる解釈**
(vector unsigned char)(*unsigned int*) (vector unsigned char){*unsigned int*}	それぞれが単一整数値を有する、16 個の符号なし 8 ビット数量値のセット。
(vector unsigned char)(*unsigned int*, ...) (vector unsigned char){*unsigned int*, ...}	各整数 (16 個) によってそれぞれ指定された値を有する、16 個の符号なし 8 ビット数量値のセット。
(vector signed char)(*int*) (vector signed char){*int*}	それぞれが単一の整数値を有する、16 個の符号付き 8 ビット数量値のセット。
(vector signed char)(*int*, ...) (vector signed char){*int*, ...}	各整数 (16 個) によってそれぞれ指定された値を有する、16 個の符号付き 8 ビット数量値のセット。
(vector bool char)(*unsigned int*) (vector bool char){*unsigned int*}	それぞれが単一整数値を有する、16 個の符号なし 8 ビット数量値のセット。
(vector bool char)(*unsigned int*, ...) (vector bool char){*unsigned int*, ...}	各整数 (16 個) によってそれぞれ指定された値を有する、16 個の符号なし 8 ビット数量値のセット。
(vector unsigned short)(*unsigned int*) (vector unsigned short){*unsigned int*}	それぞれが単一整数値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector unsigned short)(*unsigned int*, ...) (vector unsigned short){*unsigned int*, ...}	各整数 (8 つ) によってそれぞれ指定された値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector signed short)(*int*) (vector signed short){*int*}	それぞれが単一整数値を有する、8 つの符号付き 16 ビット数量値のセット。
(vector signed short)(*int*, ...) (vector signed short){*int*, ...}	各整数 (8 つ) によってそれぞれ指定された値を有する、6 つの符号付き 16 ビット数量値のセット。
(vector bool short)(*unsigned int*) (vector bool short){*unsigned int*}	それぞれが単一整数値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector bool short)(*unsigned int*, ...) (vector bool short){*unsigned int*, ...}	各整数 (8 つ) によってそれぞれ指定された値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector unsigned int)(*unsigned int*) (vector unsigned int){*unsigned int*}	それぞれが単一整数値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector unsigned int)(*unsigned int*, ...) (vector unsigned int){*unsigned int*, ...}	各整数 (4 つ) によってそれぞれ指定された値を有する、4 つの符号なし 32 ビット数量値のセット。

表 12. ベクトル・リテラル (続き)

構文	コンパイラーによる解釈
(vector signed int)(*int*) (vector signed int){*int*}	それぞれが単一整数値を有する、4 つの符号付き 32 ビット数量値のセット。
(vector signed int)(*int*, ...) (vector signed int){*int*, ...}	各整数 (4 つ) によってそれぞれ指定された値を有する、4 つの符号付き 32 ビット数量値のセット。
(vector bool int)(*unsigned int*) (vector bool int){*unsigned int*}	それぞれが単一整数値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector bool int)(*unsigned int*, ...) (vector bool int){*unsigned int*, ...}	各整数 (4 つ) によってそれぞれ指定された値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector unsigned long long)(*unsigned long long*) (vector unsigned long long){*unsigned long long*}	両方とも単一の long long 値を有する、2 つの符号なし 64 ビット数量値のセット。
(vector unsigned long long)(*unsigned long long*, ...) (vector unsigned long long){*unsigned long long*, ...}	2 つの unsigned long long のそれぞれによって指定された値を有する、2 つの符号なし 64 ビット数量値のセット。
(vector signed long long)(*signed long long*) (vector signed long long){*signed long long*}	両方とも単一の long long 値を有する、2 つの符号付き 64 ビット数量値のセット。
(vector signed long long)(*signed long long*, ...) (vector signed long long){*signed long long*, ...}	2 つの long long それぞれによって指定された値を有する、2 つの符号付き 64 ビット数量値のセット。
(vector bool long long)(*unsigned long long*) (vector bool long long){*unsigned long long*}	単一の unsigned long long によって指定された値を有する、2 つのブール 64 ビット数量値のセット。
(vector bool long long)(*unsigned long long*, ...) (vector bool long long){*unsigned long long*, ...}	2 つの unsigned long long のそれぞれによって指定された値を有する、2 つのブール 64 ビット数量値のセット。
(vector float)(*float*) (vector float){*float*}	それぞれが単一の浮動小数値を有する、4 つの 32 ビット単精度浮動小数点数量値のセット。
(vector float)(*float*, ...) (vector float){*float*, ...}	各浮動小数点 (4 つ) によってそれぞれ指定された値を有する、4 つの 32 ビット単精度浮動小数点数量値のセット。
(vector double)(*double*) (vector double){*double*}	両方とも single double 値を有する、2 つの 64 ビット倍精度浮動小数点の数量値のセット。
(vector double)(*double*, *double*) (vector double){*double*, *double*}	2 つの double のそれぞれで指定された値を有する、2 つの倍精度浮動小数点数値のセット。
(vector pixel)(*unsigned int*) (vector pixel){*unsigned int*}	それぞれが単一整数値を有する、8 つの符号なし 16 ビット数量値のセット。

表 12. ベクトル・リテラル (続き)

構文	コンパイラーによる解釈
(vector pixel)(*unsigned int*, ...) (vector pixel){*unsigned int*, ...}	各整数 (8 つ) によってそれぞれ指定された値を有する、8 つの符号なし 16 ビット数量値のセット。

注: ベクトル・ブールのエレメントの値は、エレメントの各ビットが 0 に設定される場合は FALSE で、1 に設定される場合は TRUE です。

例えば、符号なし整数ベクトル型の場合、リテラルは次のどちらでも可能です。

```
(vector unsigned int)(10)   /* initializes all four elements to a value of 10  */
(vector unsigned int)(14, 82, 73, 700)    /* initializes the first element
                                             to 14, the second element to 82,
                                             the third element to 73, and the
                                             fourth element to 700   */
```

ベクトル・リテラルは、206 ページの『Cast 演算子 ()』を使用してキャストできます。ベクトル・リテラルを括弧で囲んでキャストすると、コードの可読性を改善できます。例えば、以下のコードを使用して、`vector signed int` リテラルを `vector unsigned char` リテラルにキャストできます。

```
(vector unsigned char)((vector signed int)(-1, -1, 0, 0))
```

関連資料:

67 ページの『ベクトル型 (IBM 拡張)』

132 ページの『ベクトルの初期化 (IBM 拡張)』

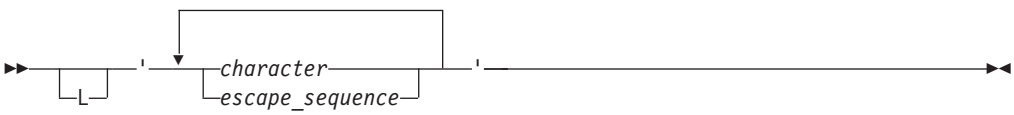
文字リテラル

文字リテラル には、一重引用符記号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'` です。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*ナロー文字リテラル* です。 L 接頭部のある文字リテラルは、*ワイド文字リテラル* です。複数の文字またはエスケープ・シーケンス (単一引用符 (')、円記号 (¥) または改行文字を除く) を含んでいる通常の文字リテラルは、*複数文字リテラル* です。

C ナロー文字リテラルの型は、intです。ワイド文字リテラルの型は `wchar_t` です。複数文字リテラルの型は `int` です。

C++ 1 文字のみを含む文字リテラルの型は、整数型である `char` です。ワイド文字リテラルの型は `wchar_t` です。複数文字リテラルの型は `int` です。

文字リテラルの構文



文字リテラルには、少なくとも 1 つの文字またはエスケープ・シーケンスがなければなりません。また、文字リテラルは、単一の論理ソース行に記述され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。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、単一引用符記号を表すには、円記号の後に単一引用符記号 (¥' エスケープ・シーケンス) を使用する必要があります。(エスケープ文字で表される他の文字のリストについては、40 ページの『エスケープ・シーケンス』を参照してください。)

基本ソース文字セット外では、文字および数字のユニバーサル文字名は、C++ および C99 言語レベルで使用できます。C++ C++ では、ユニバーサル文字名サポート用の **-qlanglvl=ucs** オプションを使用してコンパイルする必要があります。

文字リテラルの例を次に示します。

```
'a'
'¥''
L'0'
'('
```

関連資料:

66 ページの『文字型』

38 ページの『ソース・プログラムの文字セット』

41 ページの『ユニコード規格』

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

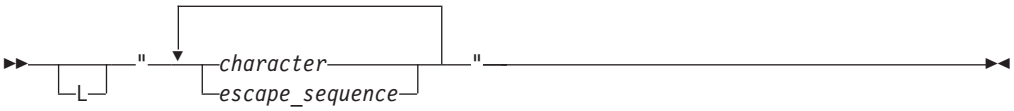
ストリング・リテラル

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

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

C++ ナロー・ストリング・リテラルの型は、const char の配列です。ワイド・ストリング・リテラルの型は、const wchar_t の配列です。どちらの型も静的ストレージ期間を持ちます。

ストリング・リテラルの構文



ストリング・リテラル内に含まれている複数のスペースは保存されます。

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

基本ソース文字セット外では、文字および数字のユニバーサル文字名は、C++ および C99 言語レベルで使用できます。 C++ C++ では、ユニバーサル文字名サポート用の **-qlanglvl=ucs** オプションを使用してコンパイルする必要があります。

ストリング・リテラルの以下の例を参照してください。

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

この例は、ストリング・リテラルの中のエスケープ・シーケンスの例です。

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

int main () {
       char *s ="Hi there! ¥n";

      cout << s;
       char *p = "The backslash character ¥¥.";

       cout << p << endl;
       char *q = "The double quotation mark ¥".¥n";
       cout << q ;
 }
```

このプログラムの出力は次のようになります。

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

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

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

注: プログラム・ソースに同じストリング・リテラルが複数回現れるときは、ストリングが読み取り専用であるか、書き込み可能であるかによって、そのストリングの保管方法は異なります。デフォルトでは、コンパイラーは、ストリングを読み取り専用と見なします。 XL C/C++ は、読み取り専用ストリングに対して 1 つの場所だけを割り振ることがあります。すべてのオカレンスは、その場所だけを参照します。ただし、そのストレージ域は、書き込み保護であると考えられます。ストリングが書き込み可能の場合は、ストリングの各オカレンスに、それぞれ別個の、常に変更可能な保管場所が割り当てられます。 **#pragma strings** ディレクティブまたは **-qro** コンパイラー・オプションを使用して、デフォルトのストリング・リテラル用ストレージを変更します。

ストリング連結

ストリングを継続する別の方法は、複数の連続ストリングを持つことです。隣接するストリング・リテラルを連結して、単一のストリングを生成することができます。次に例を示します。

```
"hello " "there"    //equivalent to "hello there"
"hello" "there"     //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。

ワイド・ストリング・リテラルとナロー・ストリング・リテラルが次の例のように隣接すると、

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

その結果は、ワイド・ストリング・リテラルになります。

注: C C99 では、ナロー・ストリングをワイド・ストリング・リテラルと連結できます。 C++11 C++11 では、C および C++ コンパイラーに対して共通のプリプロセッサー・インターフェースを提供するために、C99 プリプロセッサーのストリング・リテラル連結に対し変更が採用されています。C++11 では、ナロー・ストリングをワイド・ストリング・リテラルと連結できます。詳しくは、 540 ページの『C++11 に採用された C99 プリプロセッサー機能』を参照してください。

いずれの連結の場合もその後で、各ストリングの終わりに、char 型の '¥0' が付加されます。ワイド・ストリング・リテラルに対して、wchar_t 型の '¥0' が付加されます。規則によって、プログラムでは、NULL 文字が検出されると、ストリングの最後と認識されます。 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"
```

関連資料:

66 ページの『文字型』

38 ページの『ソース・プログラムの文字セット』

41 ページの『ユニコード規格』

u リテラルのストリング連結

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qro』 を参照

「XL C/C++ コンパイラー・リファレンス」の #pragma strings を参照

ポインター・リテラル (C++11)

ポインター・リテラルは、型 std::nullptr_t の prvalue である nullptr キーワードのみです。この型の prvalue は、任意のポインター型、pointer-to-member 型、またはブール型に変換できる、NULL ポインター定数です。

関連資料:

120 ページの『NULL ポインター』

160 ページの『ポインター型変換』

区切り子と演算子

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

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

表 13. C および C++ の区切り子

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

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

表 14. C++ の区切り子

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

C++

代替トークン

C および C++ の両方で、一部の演算子および区切り子に対して以下の代替表記が提供されています。代替表記は、*連字 (digraphs)* とも呼ばれます。

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

上記にリストした演算子および区切り子のほかに、C99 言語レベルの C++ および C は以下の代替表記を提供します。C では、これらはヘッダー・ファイル `iso646.h` の中でマクロとして定義されています。

演算子または区切り子	代替表記
&&	and
|	bitor

演算子または区切り子	代替表記
||	or
^	xor
~	compl
&	bitand
&=	and_eq
|=	or_eq
^=	xor_eq
!	not
!=	not_eq

関連資料:

43 ページの『2 文字表記文字』
64 ページの『ブール型』
155 ページの『ブール型変換』
65 ページの『浮動小数点型』
155 ページの『浮動小数点変換』
176 ページの『単項式』
67 ページの『ベクトル型 (IBM 拡張)』
132 ページの『ベクトルの初期化 (IBM 拡張)』
『ソース・プログラムの文字セット』
41 ページの『ユニコード規格』
66 ページの『文字型』
165 ページの『第 6 章 式と演算子』

ソース・プログラムの文字セット

以下は、コンパイル時と実行時の両方で使用可能な基本ソース文字セットのリストです。

- アルファベットの大文字および小文字

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

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

  0 1 2 3 4 5 6 7 8 9
- 以下の図形文字

  ! " # % & ' ( ) * + , - . / : ; < = > ? [¥] _ { } ~
  - ASCII の脱字記号 (^) 文字 (ビット単位の排他 OR 記号)
  - ASCII の分割垂直バー (¦) 文字。
- スペース文字

- 改行、水平タブ、垂直タブ、用紙送り、ストリングの終り (NULL 文字)、アラート、バックスペース、および復帰を表す制御文字

IBM コンパイラー・オプションに応じて、他の特殊な ID (ドル記号 ($) または国別文字セットの中の文字など) を ID の中で使用することができます。

関連資料:

ID の文字

マルチバイト文字

コンパイラーは、ストリング・リテラルと文字定数に十分使用できる追加文字 (拡張文字セット) を認識し、サポートします。外字のサポートには、*マルチバイト文字* セットが含まれます。 *マルチバイト文字* とは、そのビット表現が複数バイトに収まる文字のことです。コンパイラーにマルチバイト文字セットをソース入力として認識させるには、**-qmbcs** オプションを使用してコンパイルしてください。

マルチバイト文字は、以下のいずれのコンテキストにも含めることができます。

- ストリング・リテラルおよび文字定数。マルチバイト・リテラルを宣言するためには、 L の接頭部を付けたワイド文字表現を使用します。次に例を示します。

  ```
  wchar_t *a = L"wide_char_string";
  wchar_t b = L'wide_char';
  ```

  マルチバイト文字を含んだストリングは、マルチバイト文字を含まないストリングの場合と本質的に同様に扱われます。一般に、ワイド文字は、マルチバイト文字が存在するあらゆる場所で許可されています。ただし、ビット・パターンが異なるため、同一ストリング内のマルチバイト文字とは非互換です。許可された場所ではどこでも、同一ストリング内にシングルバイト文字とマルチバイト文字を混在させることができます。
- プリプロセッサー・ディレクティブ。以下のプリプロセッサー・ディレクティブでは、マルチバイト文字定数とストリング・リテラルが許可されています。
  - #define
  - #pragma comment
  - #include

  `#include` ディレクティブの中で指定したファイル名には、マルチバイト文字を含めることができます。 次に例を示します。

  ```
  #include <multibyte_char/mydir/mysource/multibyte_char.h>
  #include "multibyte_char.h"
  ```
- マクロ定義。ストリング・リテラルと文字定数は `#define` ステートメントの一部とすることができるので、マルチバイト文字もオブジェクトや関数のようなマクロ定義内で許可されます。
- `#` および `##` 演算子
- プログラムのコメント

以下は、マルチバイト文字の使用に関する制限です。

- マルチバイト文字を ID の中で使用することはできません。

- マルチバイト文字の 16 進値は、使用しているコード・ページの範囲内に収まるものでなければなりません。
- ワイド文字とマルチバイト文字をマクロ定義の中に混在させることはできません。例えば、ワイド・ストリングとマルチバイト・ストリングを連結するようなマクロ展開を行うことはできません。
- ワイド文字とマルチバイト文字の間での割り当ては禁止されています。
- ワイド文字ストリングとマルチバイト文字ストリングとの連結は禁止されています。

関連資料:

文字リテラル

41 ページの『ユニコード規格』

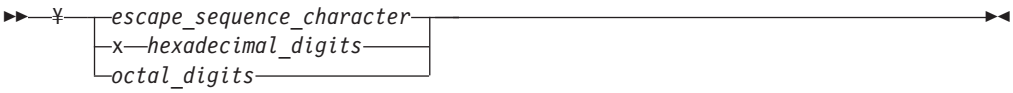
66 ページの『文字型』


「XL C/C++ コンパイラー・リファレンス」の中の『-qmbcs』 を参照

エスケープ・シーケンス

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

エスケープ文字の構文



エスケープ・シーケンスでは、円記号 (¥) の後に、エスケープ・シーケンス文字の 1 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 つ以上の 16 進数字 (0 から 9、A から F、a から 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 から 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、文字ストリングの中で使われ、ソース・コードの現在行が次の行に続くことを表します。

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	用紙送り (改ページ)
¥n	改行
¥r	復帰
¥t	水平タブ

エスケープ・シーケンス	表される文字
¥v	垂直タブ
¥'	一重引用符
¥"	二重引用符
¥?	疑問符
¥¥	円記号

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

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

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

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

このステートメントでは、次のように出力されます。

```
The escape sequence ¥n.
```

ユニコード規格

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

ユニコード規格は、各文字に固有の数値と名前を指定しており、数値のビット表現に 3 つのエンコード方式を定義しています。名前と値のペアで文字の識別を行います。文字を表す 16 進値はコード・ポイント と呼ばれます。仕様には、全体の文字特性、すなわち、各文字の大/小文字、方向性、英字特性、その他のセマンティクス情報も記述されています。 ASCII に基づくと、ユニコード規格は英字、表意文字、およびシンボルを扱い、予約済みコード・ポイント範囲でインプリメンテーション別の文字コードを定義することができます。したがって、ユニコード規格に準拠するエンコード方式は、世界中のすべての歴史的スクリプトへの対応を含め、あらゆる既知の文字エンコード要件を扱える十分な柔軟性があります。

C99 および C++ では、 ISO/IEC 10646 で定義されたユニバーサル文字名の構成は基本ソース文字セット外部の文字を表すことができます。どちらの言語でも、ID、

文字定数、およびストリング・リテラルにユニバーサル文字名を使用することを許可しています。 C++ では、ユニバーサル文字名サポート用の **-qlanglvl=ucs** オプションを使用してコンパイルする必要があります。

以下の表に、一般的なユニバーサル文字名の構成と ISO/IEC 10646 のショート・ネームの対応を示します。

ユニバーサル文字名	ISO/IEC 10646 ショート・ネーム
ここでは、N は 16 進数字です。	
¥UNNNNNNNN	NNNNNNNN
¥uNNNN	0000NNNN

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

以下の文字も禁止されています。

- ショート ID が 00A0 より小さい文字。ただし、0024 ($)、0040 (@)、または 0060 (') は例外です。
- ショート ID が D800 から DFFF まで (両端を含む) のコード・ポイント範囲内にある文字

UTF リテラル (IBM 拡張)

ISO C および ISO C++ 委員会は、ユニコード UTF-16 および UTF-32 文字リテラルをサポートするために、それぞれ *u-literals* および *U-literals* のインプリメンテーションを承認しました。

以下の表に、UTF リテラルの構文を示します。

表 15. UTF リテラル

構文	説明
u'*character*'	UTF-16 文字を示します。
u"*character-sequence*"	UTF-16 文字の配列を示します。
U'*character*'	UTF-32 文字を示します。
U"*character-sequence*"	UTF-32 文字の配列を示します。

u リテラルのストリング連結

u-literals および U-literals に関する連結規則は、ワイド文字リテラルの場合と同じです。通常の文字ストリングがある場合は、ワイド化されます。可能な組み合わせは以下のとおりです。その他の組み合わせはすべて無効です。

組み合わせ	結果
u"a" u"b"	u"ab"
u"a" "b"	u"ab"
"a" u"b"	u"ab"
U"a" U"b"	U"ab"
U"a" "b"	U"ab"

組み合わせ	結果
"a" U"b"	U"ab"

複数の連結を使用することもできますが、上記の規則が再帰的に適用されます。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qutf』 を参照

ストリング連結

2 文字表記文字

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

2 文字表記文字は、次の通りです。

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

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

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

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

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qdigraph』 を参照

3 文字表記シーケンス

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

3 文字表記	単一文字	説明
??=	#	ポンド記号
??(	[	左大括弧
??)	]	右大括弧
??<	{	左中括弧
??>	}	右中括弧
??/	¥	円記号

3 文字表記	単一文字	説明
??'	^	脱字記号
??!	|	垂直バー
??-	~	波形記号 (tilde)

プリプロセッサーが、3 文字表記シーケンスを対応する単一文字表現に置換します。次に例を示します。

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

これは、次のものを表します。

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

コメント

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

2 種類のコメントがあります。

- /* (スラッシュ、アスタリスク) 文字があって、その後に文字の任意のシーケンス (改行を含む) が続き、さらにその後に */ 文字が続きます。この種類のコメントは、通常 *C スタイルのコメント* と呼ばれています。
- // (2 つのスラッシュ) があって、その後に文字の任意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。円記号の文字は 3 文字表記で表すこともできます。

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

マルチバイト文字を含めることもできます。コンパイラーにソース・コードのマルチバイト文字を認識させるには、**-qmbcs** オプション を使用してコンパイルしてください。

注: 文字定数またはストリング・リテラルで検出される /* 文字または */ 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、スペースと等価であるため、このプログラムの出力は次のようになります。

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

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

```
#include <stdio.h>

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

このプログラムの出力は、次のようになります。

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

次の例では、コメントは強調表示されています。

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

    #include <stdio.h>

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

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

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

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

    #define TEST_FUNCTION 0
    #include <stdio.h>

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

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

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

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

```
#include <stdio.h>

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

注: また、**#pragma comment** ディレクティブを使用して、オブジェクト・モジュールにコメントを配置することもできます。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qmbcs』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qcpluscmt』 を参照

39 ページの『マルチバイト文字』

第 3 章 データ・オブジェクトとデータ宣言

このセクションのトピックでは、データ・オブジェクトの宣言を構成する各種エレメントについて説明します。

トピックは、エレメントが宣言の中に現れる順序と概ね同じ順で記載されています。113 ページの『第 4 章 宣言子』でも、引き続いてデータ宣言の追加エレメントについて説明します。

データ・オブジェクトとデータ宣言の概要

以下のセクションで、この解説書全体で使用されるデータ・オブジェクトとデータ宣言に関する基本的な概念を紹介します。

データ・オブジェクトの概要

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

C++ クラス型のインスタンスは、一般にクラス・オブジェクト と呼ばれます。個別クラスのメンバーもまた、オブジェクトと呼ばれます。

データ型は、しばしば、次のように型カテゴリーにグループ化されますが、型カテゴリーは重なり合います。

「基本型」対「派生型」
: 基本 データ型は、言語に対して「基本」または「組み込み」とも呼ばれます。基本データ型としては、整数型、浮動小数点数型、および文字型があります。派生 型は、標準 C++ の「複合」型とも呼ばれ、基本型のセットから作成され、配列、ポインター、構造体、共用体、列挙型、およびベクトルなどが含まれます。C++ のクラスはすべて複合型と見なされます。

「組み込み型」対「ユーザー定義型」
: 組み込み データ型には、基本型のすべてと、基本型のアドレスを参照する型 (配列、ポインターなど) があります。ユーザー定義 型は、ユーザーによって、typedef 定義、構造体定義、共用体定義、列挙型定義の中で基本型セットから作成されます。C++ のクラスはユーザー定義の型と見なされます。

「スカラー型」対「集合体型」
: スカラー 型は単一のデータ値を表し、一方、集合体 型は、同じ型の複数の値、または異なる型の複数の値を表します。スカラーには、算術型とポインターがあります。集合体型には、配列、構造体、およびベクトルがあります。C++ クラスは集合体型と見なされます。

次のマトリックスは、サポートされるデータ型とその種別 (基本型、派生型、スカラー型、および集合体型) をリストしたものです。

表 16. C/C++ データ型

データ・オブジェクト	基礎	複合	組み込み	ユーザー定義済み	スカラー	集合体
整数型	+		+		+	
浮動小数点型 [1]	+		+		+	
文字型			+		+	
ブール	+		+		+	
void 型	+[2]		+		+	
ポインター		+	+		+	
配列		+	+			+
構造体		+		+		+
共用体		+		+		
列挙型		+		+	注 [3] を参照	
C++ クラス		+		+		+
IBM ベクトル型			+			+

注:

1. 複素数浮動小数点型は、内部では 2 つのエレメントの配列として表されますが、位置合わせと算術演算については、実浮動小数点型と同じように振る舞います。そのため、複素数浮動小数点型はスカラー型と見なされます。
2. void 型は実際に不完全な型であって、これについては、『不完全型』で説明しています。それでも、標準 C++ はこれを基本型と定義しています。
3. C C 標準では、列挙型をスカラーとしても、集合体としても分類していません。C++ 標準 C++ は列挙型をスカラーとして分類しています。

不完全型

以下は、不完全型です。

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

配列サイズが、[*] で指定された場合、これは可変長配列であることを表し、サイズは指定されたものと見なされ、そのため、配列型は完全型と見なされます。詳しくは、125 ページの『可変長配列』を参照してください。

以下の例は、不完全型を示しています。

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

互換型および複合型

C

C では、互換型とは、次のように定義されています。

- 変更なしで一緒に使用できる 2 つの型 (割り当て式の中と同じ)
- 変更なしで一方を他方で置き換えられる 2 つの型

複合型 は、2 つの互換型から構成されます。 2 つの互換型から生ずる複合型の決定は、整数型が算術演算子で結合されたときの整数型の通常の 2 項変換の結果と似ています。

明らかに、2 つの同じ型は互換性があります。その複合型は同じ型です。同一でない型、ユーザー定義型、および型修飾された型の型互換性などを統制する規則は、分かりにくくなります。 63 ページの『型指定子』で、C での基本型とユーザー定義型の互換性を説明しています。

C

C++ 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++ の型検査は C より厳密です。C では互換型が必要なところで、 C++ では、同一の型が必要です。

関連資料:

345 ページの『第 11 章 クラス (C++ のみ)』

67 ページの『void 型』

350 ページの『不完全なクラス宣言』

126 ページの『配列の互換性』

120 ページの『ポインターの互換性 (C のみ)』

269 ページの『互換性のある関数 (C のみ)』

データ宣言とデータ定義の概要

宣言 では、プログラムで使われるデータ・オブジェクトの名前および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り、そのオブジェクトに ID を関連付けます。*型* を宣言または定義するときは、ストレージは割り振られません。

次の表は、宣言と定義の例を示しています。最初の列に宣言されている ID は、ストレージを割り振りません。これらの ID は、対応する定義を参照します。2 番目の列に宣言されている ID は、ストレージを割り振ります。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
`extern double pi;`	`double pi = 3.14159265;`

宣言	宣言と定義
struct payroll;	struct payroll { char *name; float salary; } employee;

注: C C99 標準では、すべての宣言が関数の先頭の最初のステートメントの前に置かれていなければならないという制限がなくなりました。C++ C++ と同様に、コード内で宣言を他のステートメントと混在させることができます。

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、ID がそのオブジェクトにアクセスするために使われるプログラム・テキストの領域を定義します。
- 可視性。これは、ID のオブジェクトに正しくアクセスできるプログラム・テキストの領域を定義します。
- 期間。これは、ID の物理オブジェクトが実際にメモリー内に割り振られている期間を定義します。
- リンケージ。ある 1 つの ID と 1 つの特定のオブジェクトの正しい関連付けを定義します。
- 型。これにより、オブジェクトに割り振るメモリーの大きさと、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はどのように解釈するかが決まります。

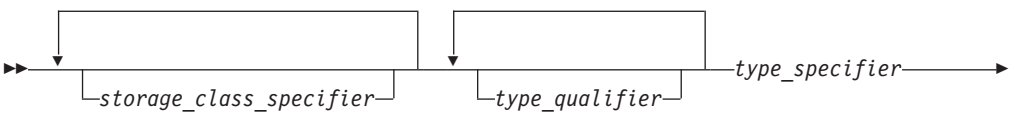
データ・オブジェクトを宣言する際のエレメントは、次のとおりです。

- 54 ページの『ストレージ・クラス指定子』。これは、ストレージ期間とリンケージを指定します。
- 63 ページの『型指定子』。これは、データ型を指定します。
- 101 ページの『型修飾子』。これは、データ値の変更の可/不可を指定します。
- 宣言子。これは、ID を紹介して組み込みます。
- 127 ページの『初期化指定子』。これは、ストレージを初期値で初期化します。

IBM さらに XL C/C++ では、GCC との互換性のために、ユーザーが*属性* を使用してデータ・オブジェクトのプロパティーを変更できるようになりました。*型属性* (これを使用してユーザー定義型の定義を変更できます) については、107 ページの『型属性 (IBM 拡張)』で説明しています。変数属性 (これを使用して変数の宣言を変更できます) については、145 ページの『変数属性 (IBM 拡張)』で説明しています。 IBM

すべての宣言の形式は、次のとおりです。

データ宣言の構文



仮定義

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

C 以下のステートメントは、通常の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。

関連資料:

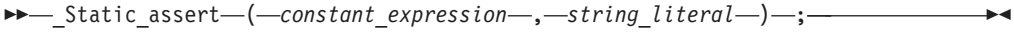
263 ページの『関数の宣言と定義』

_Static_assert 宣言 (C11)

注: IBM は、C11 の選択された機能 (C1X と呼ばれる) をその承認の前にサポートします。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C11 標準ライブラリーのサポートを含め、すべての C11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による C11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは特に行いません。

静的アサーションを宣言して、コンパイル時に一般的な使用エラーを検出および診断することができます。_Static_assert 宣言の形式は、以下のとおりです。

_Static_assert 宣言の構文



constant_expression は整数定数式でなければなりません。整数定数式の評価の結果が 0 になった場合、コンパイラーは、_Static_assert 宣言のソース・ロケーションと一緒にストリング・リテラル を含む重大エラーを出します。それ以外の場合は、_Static_assert 宣言は影響を与えません。

静的アサーションの宣言は、新しい型もオブジェクトも宣言せず、実行時にサイズも時間コストも暗黙指定しません。

static_assert は、C の assert.h に定義されているマクロです。

静的アサーションを C 言語に追加すると、以下のような利点があります。

- ライブラリーがコンパイル時に一般的な使用エラーを検出できます。
- C 標準ライブラリーのインプリメンテーションにより、一般的な使用エラーを検出および診断できるため、ユーザビリティーが向上します。

静的アサーションを宣言すると、重要なプログラム・インバリアントをコンパイル時に検査できます。

例: _Static_assert 宣言

例 1: 次の例は、構造体の中での _Static_assert 宣言の使用法を示しています。

```
#include <stddef.h>
struct __attribute__((packed)) B{
   char a;
   int i;
};

struct A{
   struct B b;
   _Static_assert(offsetof(struct B,i)==1,"S not packed");
};
```

例 2: 次の例では、static_assert を使用して宣言された静的アサーションがあるため、assert.h ヘッダー・ファイルをインクルードする必要があります。

```
/* static_assert requires <assert.h> */
#include <assert.h>
static_assert(sizeof(long) >= 8, "64-bit not enabled.");
```

例 3: 次の例は、無効な定数式を指定した _Static_assert 宣言の使用について示しています。

```
_Static_assert(1 / 0, "never shows up!");
```

このプログラムをコンパイルすると、コンパイラーは _Static_assert 宣言のストリング・リテラル を表示しません。その代わり、コンパイラーは、除数をゼロにできないことを示すエラー・メッセージを出します。

関連資料:

547 ページの『C11 との互換性のための拡張機能』

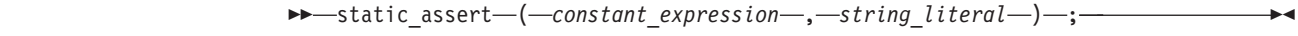
static_assert 宣言 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11

機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

静的アサーションを宣言して、コンパイル時に一般的な使用エラーを検出および診断することができます。static_assert 宣言の形式は、以下のとおりです。

static_assert 宣言の構文



constant_expression は、コンテキスト上で bool に変換できる定数式であることが必要です。このように変換された式の値が false の場合、コンパイラーは、static_assert 宣言のソース・ロケーションと一緒にストリング・リテラル を含む重大エラーを出します。それ以外の場合は、static_assert 宣言は影響を与えません。

静的アサーションは、名前空間スコープ、ブロック・スコープ、クラス・メンバー宣言リストなど、using 宣言を使用する任意の場所に宣言できます。

静的アサーションの宣言は、新しい型もオブジェクトも宣言せず、実行時にサイズも時間コストも暗黙指定しません。

C++ プログラミング言語では、C プログラミング言語との互換性を高めるために、すべての言語レベルで _Static_assert キーワードもサポートされます。

静的アサーションを C++ 言語に追加すると、以下のような利点があります。

- ライブラリーがコンパイル時に一般的な使用エラーを検出できます。
- C++ 標準ライブラリーのインプリメンテーションにより、一般的な使用エラーを検出および診断できるため、ユーザビリティーが向上します。

静的アサーションを宣言すると、重要なプログラム・インバリアントをコンパイル時に検査できます。

例: static_assert 宣言

次の例は、名前空間スコープでの static_assert 宣言の使用法を示しています。

```
static_assert(sizeof(long) >= 8, "64-bit code generation not
enabled/supported.");
```

次の例は、テンプレートに対するクラス・スコープでの static_assert 宣言の使用法を示しています。

```
#include <type_traits>
#include <string>

template<typename T>
struct X {
   static_assert(std::tr1::is_pod<T>::value, "POD required to
instantiate class template X.");
   // ...
};
```

```
int main() {
   X<std::string> x;
}
```

次の例は、テンプレートに対するブロック・スコープでの static_assert 宣言の使用法を示しています。

```
template <typename T, int N>
void f() {
   static_assert (N >=0, "length of array a is negative.");
   T a[N];
   // ...
}

int main() {
   f<int, -1>();
}
```

次の例は、無効な定数式を指定した static_assert 宣言の使用について示しています。

```
static_assert(1 / 0, "never shows up!");
```

このプログラムをコンパイルすると、コンパイラーは static_assert 宣言のストリング・リテラル を表示しません。その代わり、コンパイラーは、除数をゼロにできないことを示すエラー・メッセージを出します。

関連資料:

555 ページの『C++11 互換性の拡張機能』

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改良するために使用します。ストレージ・クラスは次のことを決定します。

- オブジェクトに、内部リンケージまたは外部リンケージがあるか、またはリンケージがないか
- オブジェクトは、メモリーまたはレジスター (使用可能な場合) のどちらに格納されるか
- オブジェクトは、デフォルトの初期値 0 を受け取るか、または中間デフォルトの初期値を受け取るか
- オブジェクトが、プログラム全体で参照されるか、または変数を定義した関数、ブロック、ソース・ファイル内でのみ参照されるか
- オブジェクトのストレージ期間が維持されるのは、プログラムのランタイム時全体か、またはそのオブジェクトが定義されているブロックの実行中のみか

変数の場合は、そのデフォルトのストレージ期間、スコープ、およびリンケージは、それがどこで宣言されたかによって異なります。つまり、ブロック・ステートメントまたは関数本体の内側なのか、外側なのかによります。これらのデフォルトが十分でない場合、ストレージ・クラス指定子を使用してストレージ・クラスを明示的に指定することができます。C および C++ のストレージ・クラス指定子は、次のとおりです。

- auto
- static

- extern
- C++ mutable
- register
- IBM __thread

C++11

C++11 では、キーワード auto は、ストレージ・クラス指定子として使用されなくなりました。代わりに、型指定子として使用されます。コンパイラーは、初期化指定子の式の型から auto 変数の型を推定します。詳しくは、91 ページの『auto 型指定子 (C++11)』を参照してください。

キーワード extern は、以前にはストレージ指定子またはリンケージ指定の一部として使用されていました。C++11 標準では、3 番目の使用法が追加されており、このキーワードを使用して明示的インスタンス生成宣言を指定できます。詳しくは、463 ページの『明示的インスタンス生成』を参照してください。

C++11

関連資料:

270 ページの『関数のストレージ・クラス指定子』

127 ページの『初期化指定子』

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、ブロック内で宣言される変数の場合はデフォルトです。自動ストレージをもつ変数 x は、x が宣言されたブロックが終了するときに削除されます。

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

自動変数のストレージ期間

auto ストレージ・クラス指定子が指定されたオブジェクトには、自動ストレージ期間が指定されます。ブロックに入るたびに、そのブロックに定義された auto オブジェクトに対するストレージが使用可能になります。ブロックが終了すると、そのオブジェクトは使用できなくなります。リンケージ指定がなく、static ストレージ・クラス指定子なし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、ブロックの呼び出しごとに、新規オブジェクトが割り振られます。

自動変数のリンケージ

auto 変数はブロック・スコープを持ち、リンケージはありません。

注: C++11 C++11 では、キーワード auto は、ストレージ・クラス指定子として使用されなくなりました。代わりに、型指定子として使用されます。コンパイラーは、初期化指定子の式の型から auto 変数の型を推定します。詳しくは、 91 ページの『auto 型指定子 (C++11)』を参照してください。

関連資料:

128 ページの『初期化とストレージ・クラス』

234 ページの『ブロック・ステートメント』

250 ページの『goto ステートメント』

静的ストレージ・クラス指定子

static ストレージ・クラス指定子で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、このようなオブジェクトのためのメモリーは、プログラムが実行を開始するときに割り振られ、プログラムが終了するときに解放されます。変数の静的ストレージ期間は、ファイル・スコープまたはグローバル・スコープとは異なります。変数は静的期間を持ちますが、ローカル・スコープです。

C キーワード static は、C において情報の隠蔽を行うための主要なメカニズムです。

C++ C++ では、情報の隠蔽を名前空間言語機能とクラスのアクセス制御によって強制的に行います。外部変数のスコープを制限するためのキーワード static の使用は、名前空間スコープ内のオブジェクトの宣言には推奨できません。

static ストレージ・クラス指定子は、以下の宣言に適用できます。

- データ・オブジェクト
- C++ クラス・メンバー
- 無名共用体

以下で static ストレージ・クラス指定子を使用することはできません。

- 型宣言
- 関数仮パラメーター

C C99 言語レベルでは、static キーワードは、関数への配列パラメーターの宣言で使用できます。 static キーワードは、関数に渡される引数は、少なくとも指定されたサイズの配列を指すポインターであることを示します。このようにして、コンパイラーは、ポインター引数が NULL でないことを知らされます。詳しくは、 282 ページの『関数仮パラメーター宣言の中の静的配列指標 (C のみ)』を参照してください。

静的変数のリンケージ

オブジェクトの宣言に、static ストレージ・クラス指定子が含まれ、ファイル・スコープがある場合、その ID は内部リンケージを持ちます。従って、そのような特定の ID の各インスタンスは 1 つのファイル内でのみ同じオブジェクトを表します。オブジェクトの宣言に static ストレージ・クラス指定子が含まれ、関数スコープがある場合、オブジェクトは静的に割り振られ、すべての関数呼び出しは同じオ

ブジェクトを使用します。例えば、静的変数 x が関数 f で宣言されている場合、プログラムが f のスコープから出るとき、x は破棄されません。

```
#include <stdio.h>

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

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

上記の例の出力は、以下のとおりです。

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

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

関連資料:

271 ページの『静的ストレージ・クラス指定子』

365 ページの『静的メンバー』

128 ページの『初期化とストレージ・クラス』

8 ページの『内部リンケージ』

313 ページの『第 9 章 名前空間 (C++ のみ)』

extern ストレージ・クラス指定子

extern ストレージ・クラス指定子により、複数のソース・ファイルから使用できるオブジェクトを宣言できます。 extern 宣言は、記述された変数を現行ソース・ファイルの以降の部分で使用できるようにします。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リンケージを持つオブジェクトを記述する場合は、キーワード extern はオプションです。

ある ID の宣言がファイル・スコープに既にある場合は、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 は外部リンケージを持ちます。

C++ C++ は、extern ストレージ・クラス指定子の使用をオブジェクトまたは関数の名前だけに制限しています。型宣言と一緒に extern 指定子を使用することは正しくありません。 extern 宣言を、クラス・スコープ内で発生させることはできません。

外部変数のストレージ期間

すべての extern オブジェクトには、静的ストレージ期間が指定されています。メモリーは、main 関数の実行が開始される前に 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、それがプログラム・テキスト内のどこで宣言されたかによって異なります。宣言がブロック内で行われた場合、その変数はブロック・スコープを持ちます。そうでない場合、ファイル・スコープです。

外部変数のリンケージ

C スコープと同様に、extern と宣言された変数のリンケージは、プログラム・テキスト内の宣言の場所によって異なります。変数宣言が関数定義の外側で行われ、ファイルの前方で static と宣言されている場合、その変数は内部リンケージを持ちます。そうでない場合、ほとんどの場合に、外部リンケージです。関数の外側で発生するオブジェクトや、ストレージ・クラス指定子を含まないオブジェクトの宣言では、すべて外部リンケージを持つ ID を宣言します。

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

注: C++11 キーワード extern は、以前にはストレージ指定子またはリンケージ指定の一部として使用されていました。 C++11 標準では、3 番目の使用法が追加されており、このキーワードを使用して明示的インスタンス生成宣言を指定できます。詳しくは、 463 ページの『明示的インスタンス生成』を参照してください。

関連資料:

9 ページの『外部リンケージ』

128 ページの『初期化とストレージ・クラス』

271 ページの『extern ストレージ・クラス指定子』

313 ページの『第 9 章 名前空間 (C++ のみ)』

5 ページの『クラス・スコープ (C++ のみ)』

mutable ストレージ・クラス指定子 (C++ のみ)

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

次の例では、

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

int main()
```

```
{
  const A var2;
  var2.x = 345;
  // var2.y = 2345;
}
```

コンパイラーは、var2 が const として宣言されているため、割り当て var2.y = 2345 を許可しません。 A::x は、mutable として宣言されているため、コンパイラーは割り当て var2.x = 345 を許可します。

関連資料:

101 ページの『型修飾子』

126 ページの『参照 (C++ のみ)』

register ストレージ・クラス指定子

register ストレージ・クラス指定子は、コンパイラーに、そのオブジェクトの値はマシン・レジスターに入れることを示します。register ストレージ・クラス指定子は、一般的には、アクセス時間を最小にすることによりパフォーマンスを上げたいときに、使用頻度の高い変数 (ループ制御変数など) に対して指定します。ただし、コンパイラーはこの要求を受け入れる必要はありません。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 が設定されているものとして処理されます。

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

次の制約事項は、register ストレージ・クラス指定子に適用されます。

- C register ストレージ・クラス指定子を持つオブジェクトを参照するためにポインターを使用することはできません。
- C グローバル・スコープ内でオブジェクトを宣言するとき、register ストレージ・クラス指定子を使用することはできません。
- C レジスターにはアドレスはありません。したがって、register 変数にアドレス演算子 (&) を適用することはできません。
- C++ 名前空間スコープ内でオブジェクトを宣言するとき、register ストレージ・クラス指定子を使用することはできません。

C++ 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
```

レジスター変数のストレージ期間

register ストレージ・クラス指定子が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 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内で register オブジェクトを定義すると、ブロックの呼び出しごとに、新規オブジェクトが割り振られます。

レジスター変数のリンケージ

register オブジェクトは、auto ストレージ・クラスのオブジェクトと等価のものとして扱われるので、リンケージはありません。

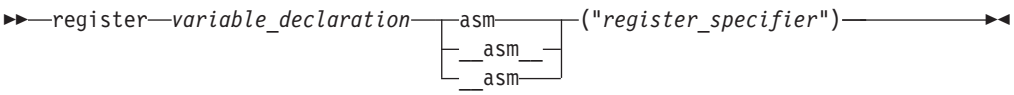
指定したレジスターの変数 (IBM 拡張)

asm レジスター変数 宣言を使用すれば、特定のハードウェア・レジスターを変数専用にすることができます。この言語拡張機能により、GNU C との互換性ができます。

グローバル・レジスター変数は、プログラム実行中はずっとレジスターを予約します。予約済みレジスターに保管した内容が削除されることはありません。

ローカル・レジスター変数は、実際にはレジスターを予約しません。ただし、変数がインライン・アセンブリー・ステートメントで入力または出力オペランドとして使用される場合は例外です。この場合、変数を asm オペランドとして使用すると、特定のレジスターをそのオペランド専用に確実に使用できますので、使用されるレジスターのコントロールが容易にできます。

レジスター変数宣言の構文規則



レジスター_ 指定子 は、ハードウェア・レジスターを表すストリングです。レジスター名は、CPU に固有のものです。以下に、有効なレジスター名を示します。

r0 から r31
汎用レジスター

f0 から f31
浮動小数点レジスター

v0 から v31
ベクトル・レジスター (選択されたプロセッサー上の)

注: レジスター r30 または r14 は、XL C++ ランタイム環境で使用するために予約されているため、これらをレジスター変数宣言に使用することはできません。

以下に、レジスター変数の使用に関する規則を示します。

- 次の型の変数に対してレジスターを予約することはできません。
  - long long 型
  - 集合体型
  - C++ クラス型
  - C++ 参照型
  - void 型
  - _Complex 型

– 128 ビット long double 型

- 汎用レジスターは、整数型またはポインター型の変数のためにだけ予約できます。
- 浮動小数点レジスターは、float、double、または 64 ビットの long double 型の変数に対してのみ予約できます。
- ベクトル・レジスターは、ベクトル型の変数のためだけに予約できます。
- グローバル・レジスター変数は初期化できません。
- グローバル・レジスター変数専用のレジスターは揮発性レジスターであってはなりません。そうでないと、グローバル変数に保管された値が関数呼び出しで保持されない可能性があります。
- 複数のレジスター変数で同じレジスターを予約することも可能です。しかし 2 つの変数が互いに別名になり、警告を伴う診断が下されます。
- 同じグローバル・レジスター変数が、複数のレジスターを予約することはできません。
- レジスター変数は、OpenMP 節あるいは OpenMP 並列領域またはワーク・シェアリング領域で使用することはできません。
- グローバル・レジスター宣言で指定されたレジスターは、レジスター宣言が指定されているコンパイル単位でのみ、宣言された変数のために予約されます。共通ヘッダー・ファイルにグローバル・レジスター宣言を指定するか、**-qreserved_reg** コンパイラー・オプションを使用する場合を除き、レジスターは他のコンパイル単位で予約されません。

関連資料:

128 ページの『初期化とストレージ・クラス』

2 ページの『ブロック/ローカル・スコープ』

126 ページの『参照 (C++ のみ)』

アセンブリー・ラベル (IBM 拡張)

252 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』


「XL C/C++ コンパイラー・リファレンス」の『-qreserved_reg』を参照

__thread ストレージ・クラス指定子 (IBM 拡張)

__thread ストレージ・クラスは、*スレッド・ローカル*のストレージ期間を持つことで、静的変数を表します。つまり、マルチスレッド・アプリケーションでは、変数の固有のインスタンスがそれを使用する各スレッドごとに作成され、スレッドが終了すると破棄されます。 __thread ストレージ・クラス指定子は、スレッド・セーフティーを保証する便利な方法を提供します。スレッドごとにオブジェクトを宣言すると、マルチスレッドが競合状態の心配をすることなく、オブジェクトにアクセスすることができます。一方、スレッドの同期化を行う低水準のプログラミングや、大規模なプログラム再構築の必要はありません。

tls_model 属性を使用すると、特定の変数に対して使用されるスレッド・ローカル・ストレージ・モデルをソース・レベルで制御することができます。tls_model 属性には、local-exec、initial-exec、local-dynamic、または global-dynamic アクセス方式のいずれか 1 つを指定する必要があります。これは、その変数の **-qtls** オプションをオーバーライドします。次に例を示します。

```
__thread int i __attribute__((tls_model("local-exec")));
```

tls_model 属性を使用すると、リンカーが、アプリケーションまたは共用ライブラリーの作成に正しいスレッド・モデルが使用されたか否かを検査できるようになります。リンカー/ローダーの動作は次のとおりです。

表 17. スレッド・アクセス・モデルのリンク時/実行時の動作

アクセス方式	リンク時診断	実行時診断
local-exec	参照されたシンボルがインポートされていると失敗します。	モジュールがメインプログラムでない場合は失敗します。参照されたシンボルがインポートされていると失敗します (リンカーはすでにエラーを検出しているはずです)。
initial-exec	なし	参照されたシンボルが、実行時にロードされたモジュール内にない場合、dlopen() は失敗します。
local-dynamic	参照されたシンボルがインポートされていると失敗します。	参照されたシンボルがインポートされていると失敗します (リンカーはすでにエラーを検出しているはずです)。
global-dynamic	なし	なし

この指定子は、以下のものに適用できます。

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

関数スコープまたはブロック・スコープの自動変数または非静的データ・メンバーには適用できません。

スレッド指定子は、static または extern 指定子の前後いずれにおくこともできます。

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

C++ スレッド変数は、定数式を使用して初期化する必要があります。

アドレス演算子 (&) をスレッド・ローカル変数へ適用すると、その変数の、現行スレッドのインスタンスのランタイム・アドレスが戻ります。そのスレッドは、このアドレスを他のスレッドに渡すことができますが、最初のスレッドが終了すると、スレッド・ローカル変数を指すポインターは無効になります。

関連資料:


「XL C/C++ コンパイラー・リファレンス」の中の『-qtls』 を参照

型指定子

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

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

C++ 型は、以下のいずれかの条件を満たす場合はリテラル型です。

- スカラー型である。
- 参照型である。
- リテラル型の配列である。
- C++11 以下のすべてのプロパティーを持つクラス型である。
  - クラスに単純デストラクターがある。
  - 非静的データ・メンバーの初期化指定子がある場合に、その中の各コンストラクター呼び出しと完全式が定数式である。
  - クラスが集合体型である。または、少なくとも 1 つの `constexpr` コンストラクターがあるか、コピー・コンストラクターや移動コンストラクターではないコンストラクター・テンプレートがある。
  - クラスのすべての非静的データ・メンバーおよび基底クラスが、リテラル型である。 C++11

C++

C++11 C++11 標準では、以下の型指定子が導入されています。

- auto 型指定子
- decltype(*expression*) 型指定子
- C++11

関連資料:

278 ページの『関数の戻りの型指定子』

整数型

整数型は、次のカテゴリーに分かれます。

- 符号付き整数型:
  - signed char
  - short int

  - int
  - long int
  - long long int
- 符号なし整数型:
  - unsigned char
  - unsigned short int
  - unsigned int
  - unsigned long int
  - unsigned long long int

unsigned 接頭部は、オブジェクトが負でない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 型は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。

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

関連資料:

整数リテラル

154 ページの『整数型変換』

154 ページの『算術変換およびプロモーション』

325 ページの『第 10 章 多重定義 (C++ のみ)』

ブール型

ブール変数を使用すると、整数値 0 または 1C++ 、あるいはリテラル true または falseC++ を保持することができます。リテラルの true および false は、算術値が必要なときにはいつでも、それぞれ整数 1 および 0 に暗黙的にプロモートされます。ブール型は符号なしであって、標準の符号なし整数型カテゴリーの中では最も低いランキングです。ブール型は、指定子 signed、unsigned、short、または long で修飾することはできません。単純な割り当てでは、左方オペランドがブール型の場合、右方オペランドは算術型またはポインターでなければなりません。

C ブール型は C99 の機能です。ブール変数を宣言するには、 _Bool 型指定子を使用してください。

IBM トークン bool は、ベクトル宣言コンテキストで使用される場合、およびベクトルのサポートが使用可能になっている場合にのみ、C のキーワードとして認識されます。 IBM

C++ でブール変数を宣言するには、bool 型指定子を使用してください。等価演算子、関係演算子、および論理演算子の結果は型ブールです。すなわち、ブール定数の true または false です。

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

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

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

関連資料:

ブール・リテラル

155 ページの『ブール型変換』

67 ページの『ベクトル型 (IBM 拡張)』

浮動小数点型

浮動小数点型指定子は、次のカテゴリーに分かれます。

- 『実浮動小数点型』
- 『複素数浮動小数点型』

実浮動小数点型

一般型、バイナリー型、浮動小数点型は次の通りです。

- float
- double
- long double

実浮動小数点型の大きさの範囲は次の表の通りです。

表 18. 実浮動小数点型の大きさの範囲

型	範囲
float	約 1.2^{-38} から 3.4^{38}
double、long double	約 2.2^{-308} から 1.8^{308}

浮動小数点定数が長すぎたり、短すぎたりする場合は、言語によって結果は未定義です。

単純な浮動小数点宣言の宣言子は、ID です。浮動定数あるいは整数か浮動小数点数に評価される変数または式を使用して、単純な浮動小数点変数を初期化します。

複素数浮動小数点型

複素数型指定子は、次のとおりです。

- float _Complex
- double _Complex
- long double _Complex

複素数型の表記要件および位置合わせ要件は、対応する実数型の 2 つのエレメントを含む配列型と同じです。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一致しています。

等価演算子および非等価演算子は、実数型の場合と同じように動作します。関係演算子には、オペランドとして複素数型を指定できません。

IBM C99 および標準 C++ に対する拡張機能として、複素数も単項演算子 ++ (増分)、-- (減分)、および ~ (ビット単位否定) のオペランドとして指定できます。
IBM

関連資料:

浮動小数点リテラル

155 ページの『浮動小数点変換』

154 ページの『算術変換およびプロモーション』

複素数リテラル (C のみ)

187 ページの『__real__ および __imag__ 演算子』

文字型

文字型は、次のカテゴリーに分かれます。

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

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

C char は signed char および unsigned char とは別個の型であって、この 3 つの型に互換性はありません。

C++ 多重定義関数を区別する目的で、C++ の char は、signed char および unsigned char とは別の型となっています。

char データ・オブジェクトが signed または unsigned のどちらであるかが重要でない場合は、そのオブジェクトをデータ型 char として宣言することができます。そうでない場合、signed char または unsigned char を明示的に宣言して、単一バイトを占有する数値変数を宣言します。char (signed または unsigned) が、int に広げられるときは、その値は保存されます。

デフォルトでは、char は、unsigned char のように動作します。このデフォルトを変更するには、**-qchars** オプションまたは **#pragma chars** ディレクティブを使用できます。詳しくは、in the 「*XL C/C++ コンパイラー・リファレンス*」の『』『**-qchars**』を参照してください。

関連資料:

文字リテラル

ストリング・リテラル

154 ページの『算術変換およびプロモーション』

void 型

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

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

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

関連資料:

116 ページの『ポインター』

201 ページの『コンマ演算子 ,』

203 ページの『条件式』

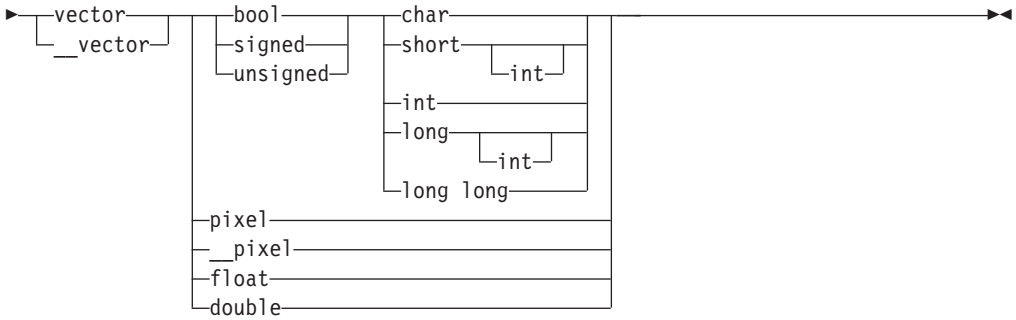
263 ページの『関数の宣言と定義』

ベクトル型 (IBM 拡張)

XL C/C++ は、言語拡張機能によって、ベクトルの処理技術をサポートします。XL C/C++ は、AltiVec プログラミング・インターフェース仕様をインプリメントおよび拡張します。拡張構文では、型修飾子およびストレージ・クラス指定子を宣言内でキーワード vector (または代替スペル __vector) に先行させることができます。

以下のダイアグラムには、構文の有効な形式のほとんどが取り込まれています。ただし、複雑になるのを避けるために、このダイアグラムでは一部のバリエーションは省略されています。例えば、const などの型修飾子や static などのストレージ・クラス指定子は、宣言の中でどのような順序で指定しても構いませんが、キーワード vector (または __vector) の直後に置くことはできません。

ベクトル宣言の構文



注:

1. キーワード `vector` は、型指定子として使用される場合、およびベクトルのサポートが使用可能になっている場合にのみ、宣言コンテキストで認識されます。キーワード `vector` または `__vector` がその前に指定されている場合にのみ、キーワード `pixel`、`__pixel`、および `bool` は、有効な型指定子として認識されます。。
2. `long` 型指定子は、ベクトル構文では推奨されず、`int` 型として処理されます。
3. ベクトル宣言コンテキストでは、重複型指定子は無視されます。

以下の表に、サポートされるベクトル・データ型と、それぞれの型のサイズおよび可能な値をリストします。

表 19. ベクトル・データ型

型	内容の解釈	値の範囲
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0, 255
vector unsigned short	8 unsigned short	0..65535
vector unsigned short int		
vector signed short	8 signed short	-32768..32767
vector signed short int		
vector bool short	8 unsigned short	0, 65535
vector bool short int		
vector unsigned int	4 unsigned int	$0..2^{32}-1$
vector unsigned long		
vector unsigned long int		
vector signed int	4 signed int	$-2^{31}..2^{31}-1$
vector signed long		
vector signed long int		
vector bool int	4 unsigned int	0, $2^{32}-1$
vector bool long		
vector bool long int		

表 19. ベクトル・データ型 (続き)

型	内容の解釈	値の範囲
vector unsigned long long	2 unsigned long long	$0..2^{64}-1$
vector bool long long		$0, 2^{64}-1$
vector signed long long	2 signed long long	$-2^{63}..2^{63}-1$
vector float	4 float	IEEE-754 単精度 (32 ビット) 浮動小数点値
vector double	2 double	IEEE-754 倍精度 (64 ビット) 浮動小数点値
vector pixel	8 unsigned short	1/5/5/5 pixel

注: vector unsigned long long、**vector bool long long**、**vector signed long long**、および **vector double** 型では、POWER7® などの VSX 命令セット拡張機能をサポートするアーキテクチャーが必要です。これらの型を使用する場合は、**-qarch=pwr7** コンパイラー・オプションを指定する必要があります。

すべてのベクトル型は 16 バイト境界上で位置合わせされます。 1 つ以上のベクトル型を含む集合体は 16 バイト境界上で位置合わせされ、ベクトル型の各メンバーも 16 バイトで位置合わせされるように、必要に応じて埋め込みが行われます。

ベクトル・データ型演算子

ベクトル・データ型には、プリミティブ・データ型で使用される単項演算子、2 項演算子、および関係演算子の一部を使用できます。特に断わりがない限り、すべての演算子には、オペランドとして互換タイプが必要であることに注意してください。これらの演算子は、グローバル・スコープまたは静的期間を持つオブジェクトに対してはサポートされず、定数の折り畳みは行われません。

単項演算子の場合、ベクトルに含まれる各エレメントに演算が適用されます。

表 20. 単項演算子

演算子	整数ベクトル型	Vector double	ブール・ベクトル型
++	可	可	否
––	可	可	否
+	可	可	否
–	可 (符号なしベクトルを除く)	可	否
~	可	否	可

2 項演算子の場合、ベクトルに含まれる各エレメントに、第 2 オペランドの同じ位置にあるエレメントとの演算が適用されます。2 項演算子には割り当て演算子も含まれます。

表 21. 2 項演算子

演算子	整数ベクトル型	Vector double	ブール・ベクトル型
+	可	可	否
–	可	可	否

表 21. 2 項演算子 (続き)

演算子	整数ベクトル型	Vector double	ブール・ベクトル型
*	可	可	否
/	可	可	否
%	可	否	否
&	可	否	可
|	可	否	可
^	可	否	可
<<	可	否	可
>>	可	否	可
[]	可	可	可

注: [] 演算子は、指定された位置のベクトル・エレメントを返します。指定された位置が有効範囲外である場合の動作は未定義です。

関係演算子の場合、ベクトルに含まれる各エレメントに第 2 オペランドの同じ位置にあるエレメントとの演算が適用され、その結果に AND 演算子が適用されて、最終結果として単一の値が得られます。

表 22. 関係演算子

演算子	整数ベクトル型	Vector double	ブール・ベクトル型
==	可	可	可
!=	可	可	可
<	可	可	否
>	可	可	否
<=	可	可	否
>=	可	可	否

次のコードの場合

```
vector unsigned int a = {1,2,3,4};
vector unsigned int b = {2,4,6,8};
vector unsigned int c = a + b;
int e = b > a;
int f = a[2];
vector unsigned int d = ++a;
```

`c` の値は `(3,6,9,12)`、`d` の値は `(2,3,4,5)`、`e` の値はゼロ以外、また `f` の値は `3` になります。

関連資料:

ベクトル・リテラル

132 ページの『ベクトルの初期化 (IBM 拡張)』

102 ページの『__align 型修飾子 (IBM 拡張)』

146 ページの『aligned 変数属性』

ユーザー定義の型

ユーザー定義の型は、以下のとおりです。

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

C++ のクラスを 345 ページの『第 11 章 クラス (C++ のみ)』で説明しています。詳述型指定子については、 349 ページの『クラス名のスコープ』で解説されています。

関連資料:

107 ページの『型属性 (IBM 拡張)』

構造体および共用体

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

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

C++ C++ では、構造体と共用体は、そのメンバーと継承がデフォルトで共通である点を除き、クラスと同じです。 C++

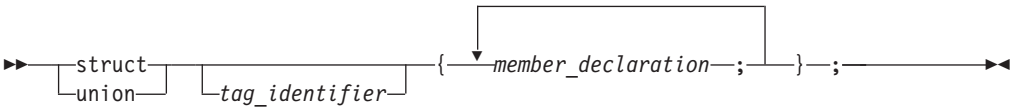
構造体または共用体の型は、『構造体および共用体の型定義』および 78 ページの『構造体および共用体の変数宣言』で説明しているように、その型の変数の定義とは別に宣言することができます。あるいは、構造体または共用体のデータ型とその型のすべての変数を、 78 ページの『1 つのステートメントでの構造体および共用体の型定義と変数定義』で説明しているように、1 つのステートメントで定義することができます。

構造体および共用体は位置合わせの考慮事項の対象です。位置合わせについて詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『位置合わせデータ』を参照してください。

構造体および共用体の型定義

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

構造体または共用体の型定義の構文



tag_identifier は、型に名前を付けます。タグ名を指定しない場合は、 78 ページの『1 つのステートメントでの構造体および共用体の型定義と変数定義』で説明しているように、その型を参照するすべての変数定義を型の宣言内に入れる必要があります。同様に、構造体または共用体の定義で型修飾子を使用することはできません。`struct` または `union` キーワードの前に置かれた型修飾子は、型定義内で宣言された変数にのみ適用されます。

メンバーの宣言

メンバーのリストには、構造体または共用体のデータ型と、その構造体または共用体に保管できる値の記述を指定します。メンバーの定義は、標準の変数宣言の形になっています。メンバー変数の名前は 1 つの構造体または共用体内では固有でなければなりませんが、同じスコープ内で定義される別の構造体型または共用体型の中では同じメンバー名を使用することができます。また、変数名、関数名、または型名と同じであってもかまいません。

構造体または共用体のメンバーは、以下を除いて、どの型であってもかまいません。

- すべての可変変更型
- `void` 型
- C 関数
- すべての不完全型

不完全型はメンバーとして許可されていないので、構造体型または共用体型は、それ自身のインスタンスをメンバーとして持つ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持つことができます。特殊なケースとして、複数のメンバーを持つ構造体の最後のメンバーは、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の説明にあるように*柔軟な配列メンバー* と呼びます。

IBM GNU C との互換性を確保するための標準 C および C++ に対する拡張機能として、XL C/C++ では、ゼロ・エクステント配列メンバー (IBM 拡張) の説明にあるように、構造体および共用体のメンバーとしてゼロ・エクステント配列を使用することもできます。 IBM

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

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

構造体メンバーはメモリー・アドレスに昇順に割り当てられます。最初のコンポーネントは、構造体名そのものの先頭アドレスから始まります。コンポーネントの適切な位置合わせを許容するために、構造体レイアウト中の隣り合わせのメンバー間に埋め込みバイトが存在することがあります。

共用体に割り振られるストレージは、共用体の最も大きいメンバーで必要なストレージです (これに、最も厳格な要件を持つメンバーの自然な境界で共用体が終了するのに必要な埋め込みが加わります)。共用体のすべてのコンポーネントはメモリー内でオーバーレイされます。共用体の各メンバーは共用体の先頭から始まるストレージを割り振られ、一時点では、1 つのメンバーしかそのストレージを使用できません。

柔軟な配列メンバー

柔軟な配列メンバー は、構造体内部に存在する無制限の配列です。 これは C99 の機能の一つであり、可変長オブジェクトのアクセスにも使用できます。 柔軟な配列メンバーは、構造体が複数の名前付きメンバーで構成されている場合は構造体の最後のメンバーとして認められます。次のように、空の索引を宣言することができます。

array_identifier[ ];

例えば、b は構造体 f の柔軟な配列メンバーです。

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

柔軟な配列メンバーは不完全型なので、柔軟な配列に sizeof 演算子を適用することはできません。この例で、ステートメント sizeof(f) は sizeof(f.a) と同じ結果 (整数のサイズ) を戻します。b は不完全型の柔軟な配列メンバーであるため、ステートメント sizeof(f.b) を使用することはできません。

柔軟な配列メンバーが含まれる構造体は、いずれも別の構造体または配列のエレメントのメンバーにすることができません。

```
struct f{
  int a;
  int b[];
};
struct f fa[10]; // Error.
```

IBM GNU C との互換性を確保するために、XL C/C++ は、標準 C および C++ を拡張して柔軟な配列メンバーに関する制限を緩和し、以下の状態を可能にしています。

- 柔軟な配列メンバーを、構造体の最後のメンバーとしてだけではなく、構造体のどの部分ででも宣言できます。柔軟な配列メンバーの後に続くメンバーの型では、その柔軟な配列メンバーの型との互換性は必要ありません。しかし、柔軟な配列メンバーの後に互換性のない型のメンバーが続く場合は、警告メッセージが出されます。次の例は、このことを示しています。

  ```
  struct s {
    int a;
    int b[];
    char c; // XL C and XL C/C++ compilers both issue a warning message.
  } f;
  ```

- 柔軟な配列メンバーを含む構造体を他の構造体のメンバーにすることができます。
- 次の 2 つの条件のどちらかが真の場合に限り、柔軟な配列メンバーを静的に初期化できます。
  - 柔軟な配列メンバーが構造体の最後のメンバーである場合。以下に例を示します。

```
struct f {
  int a;
  int b[];
} f1 = {1,{1,2,3}}; // Fine.

struct a {
  int b;
  int c[];
  int d[];
} e = { 1,{1,2},3}; // Error, c is not the last member
                    // of structure a.
```

  - 柔軟な配列メンバーが、ネストされた構造体の最外部の構造体に含まれる場合。内側の構造体のメンバーは静的に初期化できません。例:

```
struct b {
  int c;
  int d[];
};

struct c {
  struct b f;
  int g[];
} h ={{1,{1,2}},{1,2}}; // Error, member d of structure b is
                        // in the inner nested structure.
```

IBM

ゼロ・エクステント配列メンバー (IBM 拡張)

ゼロ・エクステント配列は、GNU C/C++ との互換性を確保するために用意されており、可変長オブジェクトへのアクセスに使用できます。

ゼロ・エクステント配列とは、次元として明示的にゼロが指定された配列のことです。

array_identifier [0]

例えば、b は構造体 f のゼロ・エクステント配列メンバーです。

```
struct f{
  int a;
  int b[0];
};
```

sizeof 演算子はゼロ・エクステント配列に適用でき、戻される値は 0 です。この例で、ステートメント sizeof(f) は sizeof(f.a) と同じ結果 (整数のサイズ) を戻します。ステートメント sizeof(f.b) は 0 を戻します。

ゼロ・エクステント配列を含む構造体は、配列のエレメントになることができます。次に例を示します。

```
struct f{
  int a;
  int b[0];
};
struct f fa[10]; // Fine.
```

ゼロ・エクステント配列は、空集合 {} で静的に初期化できるだけです。そうでない場合は、動的に割り振られる配列として初期化する必要があります。次に例を示します。

```
struct f{
  int a;
  int b[0];
};
struct f f1 = {100, {}}; //Fine.
struct f f2 = {100, {1, 2}};     //Error
```

ゼロ・エクステント配列はメンバーを持たないように定義されているため、ゼロ・エクステント配列が初期化されない場合、静的なゼロ埋め込みは発生しません。次の例は、このことを示しています。

```
#include <stdio.h>

struct s {
  int a;
  int b[0];
};

struct t1 {
  struct s f;
  int c[3];
} g1 = {{1},{1,2}};

struct t2 {
  struct s f;
  int c[3];
} g2 = {{1,{}},{1,2}};

int main() {
  printf("%d %d %d %d¥n", g1.f.a, g1.f.b[0], g1.f.b[1], g1.f.b[2]);
  printf("%d %d %d %d¥n", g2.f.a, g2.f.b[0], g2.f.b[1], g2.f.b[2]);
  return 0;
}
```

この例で、2 つの printf ステートメントは、以下に示す同じ出力を生成します。

```
1 1 2 0
```

ゼロ・エクステント配列の宣言は、構造体の最後のメンバーとしてだけでなく、どの部分でも行うことができます。ゼロ・エクステント配列の後に続くメンバーの型では、そのゼロ・エクステント配列の型との互換性は必要ありません。しかし、ゼロ・エクステント配列の後に互換性のない型のメンバーが続く場合は、警告が出されます。 次に例を示します。

```
struct s {
  int a;
  int b[0];
  char c;    // Issues a warning message
} f;
```

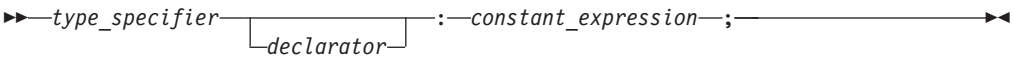
ゼロ・エクステント配列は、集合体型のメンバーとしてのみ宣言できます。次に例を示します。

```
int func(){
  int a[0];        // error
  struct S{
    int x;
    char b[0];     // fine
  };
}
```

ビット・フィールド・メンバー

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

ビット・フィールド・メンバー宣言の構文



constant_expression は、フィールド幅をビット数で示す定数整数式です。ビット・フィールドの宣言では、型修飾子 const または volatile は使用できません。

C C99 では、ビット・フィールドの許容データ型として、_Bool、int、signed int、および unsigned int があります。ビット・フィールドのデフォルトの整数型は、signed です。 C

IBM コンパイラーは、char 型、short 型、long 型または long long 型のどの型にでも成りうる拡張ビット・フィールドをサポートします。 IBM

C++ ビット・フィールドは、整数型でも列挙型のどちらでもかまいません。
C++

最大のビット・フィールド長は 64 ビットです。移植性のために、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。

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

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

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

次の制約事項は、ビット・フィールドに適用されます。次のことはできません。

- ビット・フィールドの配列の定義
- ビット・フィールドのアドレスの取得
- ビット・フィールドを指すポインターの保持

- ビット・フィールドへの参照の保持

ビット・フィールドはビット・パックされます。これは、ワード境界およびバイト境界をまたがることができます。2 つの (ゼロ以外の長さの) ビット・フィールド・メンバー間に埋め込みは挿入されません。ビット埋め込みは、次のメンバーが長さゼロのビット・フィールドまたは非ビット・フィールドである場合に、ビット・フィールド・メンバーの後に発生することがあります。非ビット・フィールド・メンバーは、宣言された型に基づいて調整されます。例えば、以下の構造体は、ビット・フィールド・メンバー間に埋め込みがない場合、および非ビット・フィールド・メンバーに先行するビット・フィールド・メンバーの後に埋め込みが挿入される場合を示しています。

```
struct {
  int larry : 25; // Bit Field: offset 0 bytes and 0 bits.
  int curly : 25; // Bit Field: offset 3 bytes and 1 bit (25 bits).
  int moe;        // non-Bit Field: offset 8 bytes and 0 bits (64 bits).
} stooges;
```

larry と **curly** の間には埋め込みはありません。 **curly** のビット・オフセットは 25 ビットです。メンバー **moe** は、次の 4 バイト境界で調整され、**curly** と **moe** の間では 14 ビットの埋め込みが発生します。

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

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

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

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

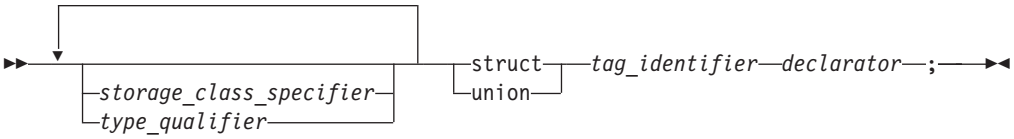
メンバー名	占有されるストレージ
`light`	1 ビット
`toaster`	1 ビット
(埋め込み — 30 ビット)	次の `int` 境界まで
`count`	`int` のサイズ (4 バイト)
`ac`	4 ビット
(名前なしフィールド)	4 ビット
`clock`	1 ビット
(埋め込み — 23 ビット)	次の `int` 境界まで (名前なしフィールド)
`flag`	1 ビット

メンバー名	占有されるストレージ
(埋め込み — 31 ビット)	次の int 境界まで

構造体および共用体の変数宣言

構造体または共用体の宣言 は、宣言が中括弧で囲まれたメンバーのリストをもっていないということを除けば、定義と同じ形式です。構造体または共用体のデータ型を宣言してからでないと、その型を持つ変数を定義することはできません。

構造体または共用体の変数宣言の構文



tag_identifier は、構造体または共用体の、データ型を示します。

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

任意のストレージ・クラスを持つ構造体または共用体を宣言できます。変数のストレージ・クラス指定子および型修飾子は、ステートメントの先頭に書き込む必要があります。register ストレージ・クラス指定子を宣言された構造体または共用体は自動変数として扱われます。

次の例は、構造体型 address を定義しています。

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

次の例は、型 address の 2 つの構造体変数を宣言しています。

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

1 つのステートメントでの構造体および共用体の型定義と変数定義

構造体または共用体の型と構造体または共用体の変数を 1 つのステートメントで定義することができます。それには、型定義の後に宣言子とオプションの初期化指定子を書き込みます。次の例では、共用体データ型 (名前なし) と共用体変数 (名前 length 付き) を定義します。

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

この例では、データ型に名前を付けていないので、length が、このデータ型になることができる唯一の変数です。struct または union キーワードの後に ID を書き込むと、データ型の名前として使用され、このデータ型の追加の変数を後のプログラムで宣言することができます。

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

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

この場合、 perm_address および temp_address は静的ストレージを割り当てられます。

型定義で宣言された変数 (1 つ以上) に型修飾子を適用することができます。次の例は両方とも有効です。

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

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

両方とも、構造体 file1 および file2 は volatile と修飾されています。

構造体メンバーおよび共用体メンバーへのアクセス

構造体または共用体変数が宣言されると、そのメンバーは、ドット演算子 (.) を付けた変数名を指定するか、または矢印演算子 (->) を付けたポインターとメンバー名を指定することにより、参照できます。例えば、次の例は、

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

ストリング "Ontario" を、構造体 perm_address の中にあるポインター prov に割り当てます。

構造体または共用体のメンバーの参照は、ビット・フィールドを含めてすべて、完全修飾される必要があります。前述の例では、4 番目のフィールドは prov のみで参照することはできず、perm_address.prov で参照できるだけです。

C11

無名構造体

注: IBM は、C11 の選択された機能 (C1X と呼ばれる) をその承認の前にサポートします。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C11 標準ライブラリ

ーのサポートを含め、すべての C11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による C11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは特に行いません。

無名構造体 とは、タグも名前も持たず、別の構造体や共用体のメンバーになっている構造体のことです。無名構造体のメンバーはすべて、親である構造体のメンバーであるかのように振る舞います。無名構造体は、以下の条件を満たしている必要があります。

- 構造体が、別の構造体または共用体の内側にネストされている。
- 構造体にはタグがありません。
- 構造体には名前がありません。

例として、無名構造体が満たす必要がある条件を以下のコードに示します。

```
struct v {
  union {
     // This is an anonymous structure, because it has no tag, no name,
     // and is a member of another structure or union.
     struct { int i, j; };

     // This is not an anonymous structure, because it has a name.
     struct { long k, l; } w;

    // This is not an anonymous structure, because
    // the structure has a tag "phone".
    struct phone {int number, areanumber;};
  };

  int m;
} v1;
```

C11

IBM

無名共用体

無名共用体 とは、タグも名前も持たず、別の共用体や構造体のメンバーになっている共用体のことです。その後に、宣言子を続けることはできません。無名共用体は型ではありません。つまり、名前なしオブジェクトを定義します。

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

例えば、次のコードでは、データ・メンバー `i` および `cptr` に直接アクセスできます。これは、これらのデータ・メンバーが、無名共用体を含むスコープにあるからです。`i` と `cptr` は、共用体メンバーで、同じアドレスが指定されているので、一度にいずれか一方のみしか使用できません。メンバー `cptr` への割り当ては、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++ 無名共用体は保護メンバーまたは専用メンバーを持つことはできません。また、メンバー関数を持つことはできません。グローバルまたは名前空間無名共用体は、キーワード static を使用して宣言される必要があります。 C++

IBM

C11

無名共用体は、以下の条件を満たす必要があります。

- 共用体が、別の共用体または構造体の内側にネストされている。
- 共用体にはタグがありません。
- 共用体には名前がありません。

C11

関連資料:

348 ページの『クラスと構造体』
108 ページの『aligned 型属性』
109 ページの『packed 型属性』
125 ページの『可変長配列』
「XL C/C++ 最適化およびプログラミング・ガイド」のビット・フィールドの位置合わせを参照
146 ページの『aligned 変数属性』
102 ページの『__align 型修飾子 (IBM 拡張)』
149 ページの『packed 変数属性』
133 ページの『構造体および共用体の初期化』
88 ページの『構造体、共用体、および列挙型の互換性 (C のみ)』
176 ページの『ドット演算子 .』
176 ページの『矢印演算子 ->』
54 ページの『ストレージ・クラス指定子』
101 ページの『型修飾子』
56 ページの『静的ストレージ・クラス指定子』
357 ページの『メンバー関数』

列挙型

列挙型 とは、整数定数 (*列挙型定数* と呼ばれる) を表す一連の名前付き値で構成されるデータ型です。列挙型定数のそれぞれに名前を作成するときに、値のそれぞれをリスト (列挙) しなければならないので、*列挙型* と呼ばれます。整数定数のセッ

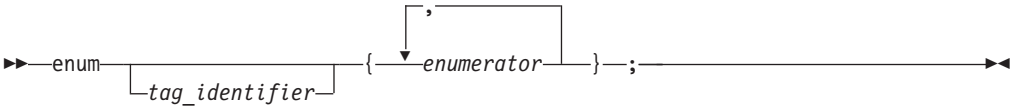
トを定義してグループ化する方法を提供することに加えて、列挙型は、起こりうる値の数が少ない変数に対しても有用です。

列挙型は、『列挙型の定義』 および 87 ページの『列挙型変数の宣言』で説明しているように、その型の変数の定義とは別に宣言することができます。あるいは、列挙型データ型とその型のすべての変数を、 87 ページの『1 つのステートメントでの列挙型と変数の定義』で説明しているように、1 つのステートメントで定義することができます。

列挙型の定義

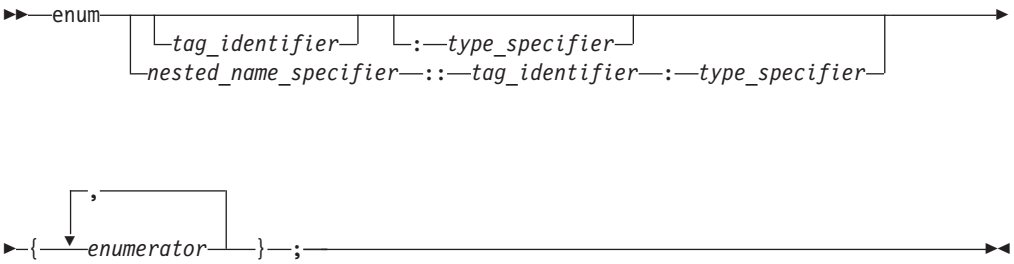
列挙型の定義には、enum、 C++11 enum class、または enum struct C++11 キーワードと、それに続くオプション識別子 (列挙型タグ)、 C++11 基礎となる型指定子 (オプション) C++11 、および中括弧で囲まれた列挙子リストが含まれます。 C++11 スコープ付き列挙型の場合、識別子は必須です。 C++11 コンマは、列挙子のリスト内で各列挙子を分離します。 C99 では、最後の列挙子と右中括弧の間に後書きコンマを使用することができます。XL C++ も、C99 との互換性のためにこの機能をサポートします。

列挙型定義の構文

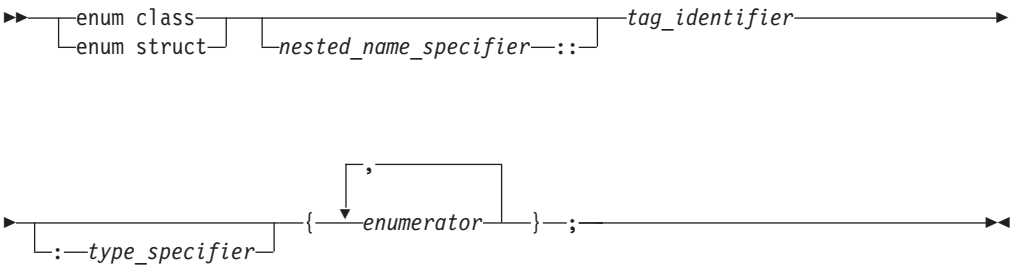


C++11

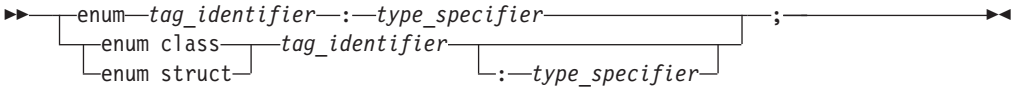
スコープのない列挙型定義の構文



スコープ付き列挙型定義の構文



列挙型フォワード宣言の構文



キーワード enum は、スコープのない列挙型を宣言するものであり、その列挙子はスコープのない列挙子です。キーワード enum class または enum struct は、スコープ付き列挙型を宣言するものであり、その列挙子はスコープ付き列挙子です。キーワード enum class と enum struct は、セマンティクス上は同じです。

スコープのない列挙型の場合、各列挙子は列挙型指定子のスコープ内と、列挙型指定子が直接含まれるスコープ内の両方で宣言されます。

スコープ付き列挙型の場合、各列挙子は列挙型指定子のスコープ内で宣言されます。また、スコープ付き列挙型では、列挙子 (または列挙型のオブジェクト) の値を整数拡張によって整数に変換することはできませんが、int、char、float などのスカラー型への明示的キャストは許可されます。次に例を示します。

```
enum class Colour { orange, green, purple };
enum class Fruit { apple, pear, grape, orange };

void functionx(int) {}
void functionx(Colour) {}
void functionx(Fruit) {}

int main()
{
  functionx(green);            // error, green is not introduced into
                               // the global scope
  functionx(Colour::green);    // calls functionx(Colour)
  functionx(Fruit::apple);     // calls functionx(Fruit)
  int i = Colour::orange;      // error, no conversion from Colour to int
  int j = (int)Colour::green;  // valid, explicit cast to integer is allowed
}
```

C++11

tag_identifier は、列挙型に名前を付けます。タグ名を指定しない場合は、87 ページの『1 つのステートメントでの列挙型と変数の定義』で説明しているように、その列挙型を参照するすべての変数定義をその型の宣言内に入れる必要があります。同様に、列挙型の定義で型修飾子を使用することはできません。enum キーワードの前に置かれた型修飾子は、型定義内で宣言された変数に適用できるだけです。

C++11 *type_specifier* は、列挙型の基礎となる型を明示的に指定します。各列挙型には、基礎となる型があります。 *type_specifier* には、任意の整数型を指定できます。列挙型は整数型ではないことに注意してください。列挙型の基礎となる型を指定することにより、プログラムの正確性を保証し、構造体の中で列挙型を安全に使用することができます。*type_specifier* を指定しない場合、列挙型の基礎となる型は次のように決定されます。

- 列挙型がスコープ付きの場合、基礎となる型は、固定の型 int になります。
- 列挙型にスコープがない場合、基礎となる型は、列挙型に定義されたすべての列挙子の値を表すことができる整数型です。

次に例を示します。

```
// the underlying type is "unsigned int"
enum Colour : unsigned int { orange = 0, green = 1, purple = 0x02U };

// the underlying type is "unsigned long"
enum class E : unsigned long { E1 = 0, E2 = 1, Emax = 0xFFFFFFF0U };

// if the type_specifier is not specified, the underlying type
// is "int" for the scoped enumeration
enum class A { A1, A2, A3 = 100, A4 };

// the underlying type is an integer type enough to represent the given
// enumerator values, eg. [signed/unsigned] long long
enum A { A1, A2, A3 = 10000000000000000000, A4 };
```

nested_name_specifier は、列挙型が以前に宣言されたクラスまたは名前空間を参照します。次に例を示します。

```
class C{

//Declaration of a scoped enumeration A.
enum class A;

};

// C is the nested_name_specifier that refers to the class where
// the enumeration A was declared.
enum class C::A { a, b };
```

列挙子のリストを指定せずに列挙型をフォワード宣言することができます。列挙型をフォワード宣言すると、列挙型の実装仕様とその使用を物理的に切り離すことができるため、コンパイル時間と依存関係を削減できます。

列挙型のフォワード宣言は、現在のスコープ内での列挙型の再宣言、または新規列挙型の宣言です。同じ列挙型の再宣言は、前の宣言と一致する必要があります。列挙型の再宣言についての規則は、以下のとおりです。

- スコープ付き列挙型を、後で、スコープのない列挙型や、基礎となる型が異なる列挙型として再宣言することはできません。
- スコープのない列挙型を、後で、スコープ付き列挙型として再宣言することはできず、それぞれの再宣言には、基礎となる型として同じ型を指定する *type_specifier* が含まれている必要があります。

次に例を示します。

```
enum E1 : unsigned long;        // valid, forward declaration of an unscoped
                                // enumeration with the underlying type
                                // "unsigned long"

enum class E2 : char;           // valid, forward declaration of a scoped
                                // enumeration with the underlying type
                                // "char"

enum class E3;                  // valid, forward declaration of a scoped
                                // enumeration with the implied underlying
                                // type "int"

enum E4;                        // error, you must specify the underlying type
                                // when you forward declare an unscoped
                                // enumeration

enum E1 : unsigned long;        // valid, the redeclaration of E1 matches the
                                // previous declaration of E1
```

```
enum class E2 : short;          // error, the previously declared enumeration
                                // E2 had the underlying type "char", and it
                                // cannot be redeclared with a different
                                // underlying type

enum E3 : int;                  // error, the previously declared enumeration
                                // E3 was a scoped enumeration, and it cannot
                                // be redeclared as an unscoped enumeration

enum class E3 : int { a, b, c };// valid, definition of E3
```

C++11

詳述型指定子

詳述型指定子の構文

詳述型指定子は、以前に宣言された列挙型を参照します。*x* は、型 *tag_identifier* を持つ変数です。

enum キーワードを使用して、変数の宣言または定義の中でスコープ付き列挙型またはスコープのない列挙型を参照できます。次に例を示します。

```
// a scoped enumeration
enum class color { red, white, black, yellow };

// an unscoped enumeration
enum letter {A, B, C, D};

// valid, regular type name usage
color pic1 = color :: white;

// valid, elaborated type usage
enum color pic2 = color :: red;
```

詳述型指定子では enum class も enum struct も使用できません。次に例を示します。

```
enum class color pic3 = color :: black;    // invalid
```

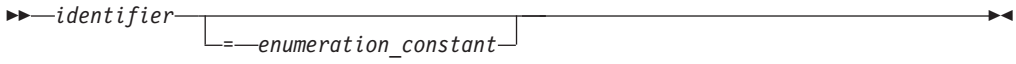
スコープのない列挙型に対する詳述型指定子は、スコープ付き列挙型の場合と同じです。次に例を示します。

```
enum letter let1 = letter :: A;            // valid
```

列挙メンバー

列挙型メンバーのリスト、すなわち、*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。

列挙型メンバー宣言の構文



C C では、*列挙型定数* は int 型です。初期化指定子として定数式が使われる場合、その定数式の値は int の範囲 (すなわち、ヘッダー limits.h で定義された INT_MIN から INT_MAX まで) を超えてはなりません。

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

列挙型定数の値は、次の方法で判別されます。

1. 列挙型定数の後の等号 (=) と定数式によって定数に明示値が与えられます。列挙型定数は、定数式の値を表します。
2. 最初の列挙子に明示的な値が割り当てられていない場合、その列挙子は値 0 (ゼロ) を取ります。
3. 明示的に割り当てられた値がない列挙型定数は、前の列挙型定数で表された値より 1 つ大きい整数値を受け取ります。

C++11

以下の列挙型の基礎となる型は固定です。前の規則に従って取得された列挙子の値は、基礎となる型で表すことができる必要があります。そうでない場合、列挙型は不適格になります。

- スコープ付き列挙型
- スコープのない列挙型で、基礎となる型が明示的に指定されているもの

スコープのない列挙型で、基礎となる型が明示的に指定されていない場合、基礎となる型は固定ではないため、列挙子の値によって次のように決定されます。

- 基礎となる型は、サイズが int 以下で、列挙型のすべての値を表すことができる、実装によって定義された整数型です。
- 列挙型の値を int または unsigned int で表すことができない場合、基礎となる型は、すべての列挙子の値を表すのに十分なサイズの、実装によって定義された整数型です。
- すべての列挙子の値を表すことができる型がない場合、その列挙型は不適格となります。

C++11

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       */
```

スコープのない各列挙型定数は、列挙型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。

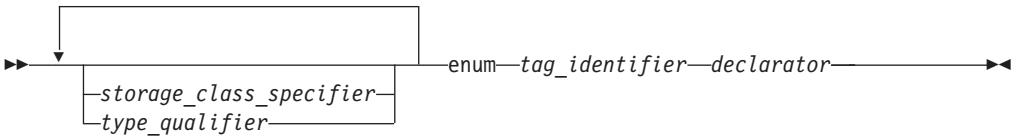
```
func()
    {
       enum score { poor, average, good };
       enum rating { below, average, above };
       int poor;
    }
```

C++11 スコープ付き列挙型を使用すると、列挙子は列挙型が含まれるスコープ内でなく、列挙型のスコープ内でのみ宣言されるため、このようなエラーを回避できます。 C++11

列挙型変数の宣言

列挙型データ型を宣言してから、列挙型データ型の変数を定義できます。

列挙型変数宣言の構文



tag_identifier は、以前に定義された列挙型のデータを示します。

C++ キーワード enum は列挙型変数宣言ではオプションです。

1 つのステートメントでの列挙型と変数の定義

変数定義の後に、宣言子とオプションの初期化指定子を使用することによって、型と変数を 1 つのステートメント内で定義できます。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;
```

これらの例はいずれも次の 2 つの宣言と等価です。

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

両方の例では、列挙型データ型 score と変数 rating を定義します。 rating で、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 を定義します。

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

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

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

関連資料:

154 ページの『算術変換およびプロモーション』

63 ページの『整数型』

135 ページの『列挙型の初期化』

『構造体、共用体、および列挙型の互換性 (C のみ)』

構造体、共用体、および列挙型の互換性 (C のみ)

1 つのソース・ファイル内で、個々の構造体または共用体定義で、他の構造体や共用体とは同じでもなく、また互換性もない新しい型を作成します。ただし、以前定義された構造体や共用体への参照となる型指定子は同じ型です。タグで参照と定義を関連付けるので、結果的には型名として機能します。これを図示すると、この例では構造体の型名 j と k だけが互換性をもっています。

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

互換性のある構造体は、互いに割り当てることができます。

同一メンバーであっても別のタグが付けられている構造体と共用体は互換性はなく、互いに割り当てることができません。同一メンバーであっても、異なる位置合わせを使用する構造体と共用体には互換性がなく、互いに割り当てることはできません。

コンパイラーは、列挙型変数と列挙型定数を整数型として扱うので、異なる列挙型の値を、型互換性に関係なく、混在させることができます。列挙型と、それを表す整数型の間に互換性があるということは、コンパイラー・オプションと関連プラグマで制御できることを意味しています。 **-qenum** コンパイラー・オプションおよび関連するプラグマについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の『』『**-qenum**』および『』『**#pragma enum**』を参照してください。

別々のソース・ファイル間の互換性

2 つの構造体、共用体、または列挙型の定義が別々のソース・ファイルで定義されている場合、各ファイルは、同じ名前で同じ型のオブジェクトに対して、理論上は別の定義を有することになります。しかし、この 2 つの宣言には互換性がなければなりません。そうでなければ、プログラムの実行時の動作は未定義になります。そのため、この互換性規則は、同じソース・ファイル内の互換性に関する互換性規則よりも制限がより厳しく、より具体的になります。別々にコンパイルされたファイルで定義された構造型、共用体型、および列挙型については、現行ソース・ファイル内の型は複合型です。

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

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

列挙型の場合、対応するメンバーも同じ値を持つこと。

構造体と共用体については、型互換性に関して以下の追加要件を満たす必要があります。

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

関連資料:

154 ページの『算術変換およびプロモーション』

345 ページの『第 11 章 クラス (C++ のみ)』

構造体または共用体の型定義

不完全型

typedef 定義

typedef 宣言で、ユーザー独自の ID を定義でき、int、float、または double の型指定子の代わりにも使用できます。 typedef 宣言は、ストレージの予約はしません。 typedef を使用して定義する名前は、新しいデータ型ではなく、データ型の同義語またはその名前で代表するデータ型の組み合わせになります。

typedef 名の名前空間は、他の ID と同じです。オブジェクトが typedef ID を使用して定義する場合は、定義したオブジェクトの属性は、ID で関連付けられ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を定義した場合とまったく同じです。

IBM

typedef 定義は、ベクトルのサポートが使用可能になっている場合に、ベクトル型を処理するために拡張されています。ベクトル型は typedef 定義の中で使用できます。この新しい型名は、他のベクトル型を宣言した場合以外は、通常の方法で使用できます。ベクトル宣言のコンテキストの中では、型指定子として typedef 名を使用することは禁止されています。以下の例は、ベクトル型を指定する typedef の一般的な使用方法を示しています。

```
typedef vector unsigned short vint16;
vint16 v1;
```

IBM

C11

、typedef 再宣言を使用して、同じスコープ内で以前の typedef 名である名前を再定義し、同じ型を参照することができます。次に例を示します。

```
typedef char AChar;
typedef char AChar;
```

typedef 再宣言機能は、任意の拡張言語レベルで使用可能にすることができます。

IBM

いずれかの拡張言語レベルが有効である場合、typedef 再宣言は、可変変更型を含む、すべてのタイプをサポートします。

IBM

可変変更型について詳しくは、 125 ページの『可変長配列』を参照してください。

C11

typedef 定義の例

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

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

上記の宣言は、次の宣言と同じです。

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

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

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

そうすると、構造体 WEIGHT は、以下の宣言で使用できます。

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

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(void);
extern SCROLL *yds;
```

次の typedef 定義では、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 と ptr1 が指すオブジェクトの型と互換性があります。ex1 型と、char、ex2、または struct b とは互換性はありません。

C++

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には、その typedef がクラス名の同義語である場合に限ります。

名前を付けずに typedef 定義で定義された C++ のクラスには、ダミーの名前が付けられます。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してはいけません。次の例を検討してみます。

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

この例では、名前なしクラスが typedef 定義に定義されています。Trees 名前なしクラスの別名で、クラス型名ではありません。そのため、この名前なしクラスにはデストラクター ~Trees() を定義できません。定義すると、コンパイラーからエラーが出されます。

C++

C++11

フレンドとしての typedef 名の宣言

C++11 標準では、拡張フレンド宣言機能が導入されており、typedef 名をフレンドとして宣言できます。詳しくは、374 ページの『拡張フレンド宣言』を参照してください。

C++11

関連資料:

115 ページの『型名』

63 ページの『型指定子』

71 ページの『構造体および共用体』

345 ページの『第 11 章 クラス (C++ のみ)』

372 ページの『フレンド』

auto 型指定子 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 は、キーワード auto を新しい型指定子として導入しています。auto は、変数の初期化指定子の式から推定される型のプレースホルダーとして機能します。auto 型推定を使用可能にすると、変数を宣言するときの型の指定が不要になります。代わりに、コンパイラーは初期化指定子の式の型から auto 変数の型を推定します。

以下の例は、auto 型推定の使用法を示したものです。

```
auto x = 1;      //x : int

float* p;
auto x = p;      //x : float*
auto* y = p;     //y : float*

double f();
auto x = f();              //x : double
const auto& y = f();       //y : const double&

class R;
R* h();
auto* x = h();    //x : R*
auto y = h();     //y : R*

int& g();
auto x = g();           //x : int
const auto& y = g();    //y : const int&
auto* z = g();          //error, g() does not return a pointer type
```

型推定のタスクをコンパイラーに委任できるため、auto 型推定を使用すると、プログラミングの利便性が向上し、プログラマーによる型指定のエラーを除去できる場合もあります。さらに、auto 型推定を使用すると、プログラムのサイズが削減されるため、プログラムの可読性も向上します。

以下の 2 つの例は、auto 型推定を使用可能にする利点を示したものです。最初の例は、auto 型推定を使用可能にしていない場合です。

```
vector<int> vec;
for (vector<int>::iterator i = vec.begin(); i < vec.end(); i++)
{
   int* a = new int(1);
   //...
}
```

auto 型推定を使用可能にすると、最初の例は以下のように簡略化できます。

```
vector<int> vec;
for (auto i = vec.begin(); i < vec.end(); i++)
{
   auto a = new auto(1);
   //...
}
```

以下の規則および制約が、auto 型推定で型指定子として auto を使用する場合に適用されます。

- auto 型推定では、配列型を推定できません。

  ```
  int x[5];
  auto y[5] = x;    //error, x decays to a pointer,
                    //which does not match the array type
  ```

- auto 型推定では、初期化指定子から *cv-qualifier* および参照型を推定できません。

  ```
  int f();
  auto& x = f();    //error, cannot bind a non-const reference
                    //to a temporary variable
  int& g();
  auto y = g();     //y is of type int
  auto& z = g();    //z is of type int&
  ```

- auto 型推定では、複数変数の auto 宣言をサポートしています。宣言子のリストに複数の宣言子が含まれる場合、各宣言子の型は、独立して推定することができます。それぞれの推定において、推定された型が同一ではない場合、プログラムは不適格です。

```
auto x=3, y=1.2, *z=new auto(1);     //error y: deduced as double,
                                     //but was previously deduced as int
```

- 宣言されるオブジェクトの名前は、その初期化指定子の式内では使用できません。

```
auto x = x++;     //error
```

- auto は、関数仮パラメーター内では使用できません。

```
int func(auto x = 3)     //error
{
    //...
}
```

注: C++11 では、キーワード auto は、ストレージ・クラス指定子として使用されなくなりました。

関連資料:

54 ページの『ストレージ・クラス指定子』

55 ページの『auto ストレージ・クラス指定子』

101 ページの『型修飾子』

555 ページの『C++11 互換性の拡張機能』

decltype(*expression*) 型指定子 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

decltype(expression) 指定子は、C++11 に導入される型指定子です。この型指定子を使用すると、型に依存する可能性のある式の結果の型に基づいて型を取得できます。

decltype(*expression*) は、オペランドとして *expression* を取ります。
decltype(*expression*) を使用して変数を定義する場合、これはコンパイラーによって *expression* の型または派生型に置換されると見なすことができます。次の例を検討してみます。

```
int i;
static const decltype(i) j = 4;
```

この例では、decltype(i) は型名 int と等価です。

decltype の一般的な使用規則

decltype(*expression*) を使用して型を取得する場合、以下の規則が適用されます。

1. *expression* が、括弧で囲まない *ID* 式 またはクラス・メンバーである場合、decltype(*expression*) は、*expression* によって命名されるエンティティーの型です。そのようなエンティティーが存在しないか、または *expression* が多重定義関数のセットを命名する場合、プログラムは不適格になります。
2. 上記以外の場合に、*expression* が xvalue である場合、decltype(*expression*) は、T&& です。ここで、T は *expression* の型です。
3. 上記以外の場合に、*expression* が左辺値である場合、decltype(*expression*) は、T& です。ここで、T は *expression* の型です。
4. 上記以外の場合、decltype(*expression*) は *expression* の型です。

以下の例は、これらの規則の使用法を示しています。

```
const int* g(){
    return new int[0];
}

int&& fun(){
   int&& var = 1;
   return 1;
}

struct A{
    double x;
};

template <class T> T tf(const T& t){
    return t;
}

bool f(){
    return false;
}

struct str1{
    template <typename T, typename U>
    static decltype((*(T*)0) * (*(U*)0)) mult(const U& arg1, const T& arg2){
       return arg1 * arg2;
    }
};

template <typename T, typename U>  struct str2{
    typedef decltype((*(T*)0) + (*(U*)0)) btype;
    static btype g(T t, U u);
};

int main(){
    int i = 4;
    const int j = 6;
    const int& k = i;
    int&& m = 1;
    int a[5];
    int *p;

    decltype(i) var1;            // int
    decltype(1) var2;            // int
    decltype(2+3) var3;          // int(+ operator returns an rvalue)
    decltype(i=1) var4 = i;      // int&, because assignment to int
                                 // returns an lvalue
    decltype((i)) var5 = i;      // int&
```

```
    decltype(j) var6 = 1;        // const int
    decltype(k) var7 = j;        // const int&
    decltype("decltype") var8 = "decltype";     // const char(&)[9]
    decltype(a) var9;            // int[5]
    decltype(a[3]) var10 = i;    // int&([] returns an lvalue)
    decltype(*p)  var11 = i;     // int&(*operator returns an lvalue)
    decltype(fun()) var12 = 1;   // int&&
    decltype(tf(A())) var13;     // A
    decltype(f()) var14;         // bool
    decltype((f())) var15;       // bool, parentheses around f() are ignored
    decltype(f)  var16;          // bool()
    decltype(&f) var17;          // bool(*)()
    decltype(&A::x) var18;       // double A::*
    decltype(str1::mult(3.0, 4u))  var19;           // double
    decltype(str2<float, short>::g(1,3)) var20;     // float
    decltype(m) var21 = 1;       // int&&
    decltype((m)) var22 = m;     // int&
    return 0;
}
```

この例では、各 decltype ステートメントの後のコメントで、定義される変数の型を説明しています。

以下の例は、decltype(*expression*) の不正な使用法を示しています。

```
int func(){
    return 0;
}
int func(int a){
    return 0;
}

int main(){
    int i = 4;

    // Incorrect usage. func names an overload function
    decltype(func) var1;

    // Correct usage. The overload operation is not ambiguous
    decltype(func(i)) var2;

    return 0;
}
```

この例では、コンパイラーは、一致する func 関数を認識できないため、エラー・メッセージを出します。

decltype を構造体メンバー変数と共に使用する場合の規則

decltype(*expression*) を使用して型を取得するときに、*expression* がオブジェクト式 (. 演算子を使用) またはポインター式 (-> 演算子を使用) の括弧で囲まないメンバー変数である場合、以下の規則が適用されます。

- オブジェクト式またはポインター式が、定数または volatile 修飾子を使用して指定される場合、型修飾子は decltype(*expression*) の結果を導きません。
- オブジェクト式またはポインター式の左辺値または右辺値は、decltype(*expression*) が参照型かどうかということに影響を与えません。

例:

```
struct Foo{
    int x;
};
```

```
int main(){
    struct Foo f;
    const struct Foo g = {0};
    volatile struct Foo* h = &f;
    struct Foo func();

    decltype(g.x) var1;         // int
    decltype(h->x) var2;        // int
    decltype(func().x) var3;    // int
    return 0;
}
```

この例では、オブジェクト式 `g` の定数修飾子は、`decltype(g.x)` の結果では望まれません。同様に、ポインター式 `h` の volatile 修飾子は、`decltype(h->x)` の結果では望まれません。オブジェクト式 `g` およびポインター式 `h` は左辺値であり、オブジェクト式 `func()` は右辺値ですが、これらは括弧で囲まないメンバー変数の decltype の結果が参照型かどうかということには影響を与えません。

decltype(*expression*) で宣言される *expression* が、括弧で囲まれた静的でない非参照クラス・メンバー変数である場合、*expression* の親オブジェクト式または親ポインター式の定数または volatile 型修飾子は、decltype(*expression*) の結果に反映されます。同様に、オブジェクト式またはポインター式の左辺値または右辺値は、decltype(*expression*) の結果に影響を与えます。

例:

```
struct Foo{
    int x;
};

int main(){
    int i = 1;
    struct Foo f;
    const struct Foo g = {0};
    volatile struct Foo* h = &f;
    struct Foo func();

    decltype((g.x)) var1 = i;         // const int&
    decltype((h->x)) var2 = i;        // volatile int&
    decltype((func().x)) var3 = 1;    // int
    return 0;
}
```

この例では、`decltype((g.x))` の結果は、オブジェクト式 `g` の定数修飾子を継承します。同様に、`decltype((h->x))` の結果は、ポインター式 `h` の volatile 修飾子を継承します。オブジェクト式 `g` およびポインター式 `h` は左辺値であるため、`decltype((g.x))` および `decltype((h->x))` は参照型です。オブジェクト式 `func()` は右辺値であるため、`decltype((func().x))` は非参照型です。

組み込み演算子 `.*` または `->*` を decltype(*expression*) 内で使用する場合、*expression* の親オブジェクト式または親ポインター式の定数または volatile 型修飾子は、*expression* が括弧で囲まれているか、または括弧で囲まない構造体メンバー変数であるかに関係なく、decltype(*expression*) の結果を導きます。同様に、オブジェクト式またはポインター式の左辺値または右辺値は、decltype(*expression*) の結果に影響を与えます。

例:

```
class Foo{
    int x;
};
int main(){
   int i = 0;
   Foo f;
   const Foo & g = f;
   volatile Foo* h = &f;
   const Foo func();

   decltype(f.*&Foo::x)  var1 = i;        // int&, f is an lvalue
   decltype(g.*&Foo::x) var2 = i;         // const int&, g is an lvalue
   decltype(h->*&Foo::x) var3 = i;        // volatile int&, h is an lvalue
   decltype((h->*&Foo::x)) var4 = i;      // volatile int&, h is an lvalue
   decltype(func().*&Foo::x) var5 = 1;    // const int, func() is an rvalue
   decltype((func().*&Foo::x)) var6 = 1;  // const int, func() is an rvalue
   return 0;
}
```

副次作用および decltype

decltype(*expression*) を使用して型を取得する場合、decltype の括弧付きのコンテキストで追加命令を実行できますが、decltype コンテキストの外側には副次作用はありません。次の例を検討してみます。

```
int i = 5;
static const decltype(i++) j = 4;    // i is still 5
```

変数 *i* は、decltype コンテキストの外側では 1 増加しません。

この規則には例外があります。以下の例では、decltype に有効な式を指定する必要があるため、コンパイラーはテンプレートのインスタンス化を実行する必要があります。

```
template <int N>
struct Foo{
    static const int n=N;
};
int i;

decltype(Foo<101>::n,i) var = i;     // int&
```

この例では、var が変数 i の型によってのみ決定される場合でも、Foo テンプレートのインスタンス生成が行われます。

decltype での冗長修飾子および指定子

decltype(*expression*) は構文上は型指定子と見なされるため、以下の冗長修飾子または指定子は無視されます。

- 定数修飾子
- volatile 修飾子
- & 指定子

以下に、これらの使用例を示します。

```
int main(){
    int i = 5;
    int& j = i;
    const int k = 1;
    volatile int m = 1;
```

```
    // int&, the redundant & specifier is ignored
    decltype(j)&  var1 = i;

    // const int, the redundant const qualifier is ignored
    const decltype(k) var2 = 1;

    // volatile int, the redundant volatile qualifer is ignored
    volatile decltype(m) var3;
    return 0;
}
```

注: decltype(*expression*) の & 冗長指定子を無視する機能は、現行 C++11 標準ではサポートされませんが、このコンパイラー・リリースではインプリメントされています。

テンプレート従属名および decltype

decltype 機能を使用せずに、関数同士でパラメーターを受け渡す場合、返される結果の正確な型を確認できない場合があります。decltype 機能は、戻りの型を簡単に一般化するための機構を提供します。以下のプログラムは、複数のオペランドで乗算演算を実行する汎用関数を示します。

```
struct Math{
    template <typename T>
    static T mult(const T& arg1, const T& arg2){
        return arg1 * arg2;
    }
};
```

arg1 および arg2 が同じ型でない場合、コンパイラーは引数から戻りの型を推定できません。この問題を解決するために、以下の例に示すように、decltype 機能を使用できます。

```
struct Foo{
   template<typename T, typename U>
   static decltype((*(T*)0)*(*(U*)0)) mult(const T& arg1, const U& arg2)
   {
       return arg1 * arg2;
   }
};
```

この例では、関数の戻りの型は、テンプレートに依存する 2 つの関数仮パラメーターの乗算結果の型です。

typeof 演算子および decltype

IBM decltype 機能は、既存の *typeof* 機能に類似しています。これらの 2 つの機能間の 1 つの違いは、decltype はオペランドとして式のみを受け入れるのに対して、typeof は型名も受け入れることができる点です。次の例を検討してみます。

```
__typeof__(int)  var1;     // okay
decltype(int) var2;        // error
```

この例では、int は型名であるため、decltype のオペランドとしては無効です。

注: __typeof__ は typeof の代替スペルです。

関連資料:

13 ページの『キーワード』

489 ページの『名前のバインディングおよび従属名』

555 ページの『C++11 互換性の拡張機能』
165 ページの『左辺値と右辺値』
126 ページの『参照 (C++ のみ)』

constexpr 指定子 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 標準は、宣言指定子として新規キーワード constexpr を導入しています。constexpr 指定子は、以下のコンテキストに対してのみ適用できます。

- 変数の定義
- 関数または関数テンプレートの宣言
- 静的データ・メンバーの宣言

次に例を示します。

```
constexpr int i = 1;       // OK, definition
constexpr extern int j;    // Error, not a definition
constexpr int f1();        // OK, function declaration, but must be defined before use
```

コンストラクターではない関数を constexpr 指定子を指定して宣言すると、その関数は constexpr 関数になります。同様に、コンストラクターを constexpr 指定子を指定して宣言すると、そのコンストラクターは constexpr コンストラクターになります。 constexpr 関数と constexpr コンストラクターのどちらも、暗黙的にインラインになります。次に例を示します。

```
struct S {
  constexpr S(int i) : mem(i) { }   // OK, declaration of a constexpr constructor
  private:
    int mem;
};
constexpr S s(55);      // OK, invocation of a constexpr constructor
```

関数または関数テンプレートのいずれかの宣言が constexpr を使用して指定されている場合、そのすべての宣言には constexpr 指定子を含める必要があります。次に例を示します。

```
constexpr int f1();         // OK, function declaration
int f1() {                  // Error, the constexpr specifier is missing
  return 55;
}
```

関数仮パラメーターは、constexpr 指定子を指定して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。

```
constexpt int f4(constexpr int);      //Error
```

オブジェクト宣言内で使用される constexpr 指定子は、オブジェクトを const として宣言します。このようなオブジェクトは、リテラル型で、初期化されていなければなりません。これがコンストラクター呼び出しで初期化される場合、その呼び出しは定数式でなければなりません。そうでない場合、constexpr 指定子が参照宣言内で使用されるのであれば、その初期化指定子に指定されるすべての完全式は、定数式でなければなりません。初期化に用いる初期化指定子式と各コンストラクター呼び出しの変換に使用されるそれぞれの暗黙的な変換は、定数式内で有効でなければなりません。次に例を示します。

```
constexpr int var;         // Error, var is not initialized
constexpr int var1 = 1;  // OK

void func() {
  var1 = 5;   //Error, var1 is const
}

struct L {
  constexpr L() : mem(55) { }
  constexpr L(double d) : mem((int)d) { }
  L(int i) : mem(i) { }
  operator int() { return mem; }
private:
  int mem;
};

// Error, initializer involves a non-constexpr constructor.
constexpr L var2(55);

double var3 = 55;

// Error, initializer involves a constexpr constructor with non-constant argument
constexpr L var4(var3);
// Error, involves conversion that uses a non-constexpr conversion function
constexpr int var5 = L();
```

コンストラクターではない非静的メンバー関数の constexpr 指定子は、そのメンバー関数を const と宣言します。その constexpr メンバー関数のクラスは、リテラル型でなければなりません。次の例では、クラス NL は、ユーザー提供のデストラクターを持っているため、非リテラル型です。

```
struct NL {
  constexpr int f(){       //error, enclosing class is not a literal type
     return 55;
  }
  ~NL() { }
};
```

constexpr 関数への呼び出しは、同等の非 constexpr 関数への呼び出しと同じ結果を生みます。ただし、constexpr 関数への呼び出しは定数式内に指定できるという点が異なります。

main 関数は、constexpr 指定子を指定して宣言することはできません。

関連資料:

20 ページの『リテラル』

309 ページの『constexpr 関数 (C++11)』

414 ページの『constexpr コンストラクター (C++11)』

174 ページの『一般化された定数式 (C++11)』

算術型の互換性 (C のみ)

2 つの算術型は、それらが同じ型の場合のみ、互換性があります。

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

型修飾子があると、型は変更されます。すなわち、const int は int と同じ型ではなく、そのため、この 2 つの型は互換性はありません。

型修飾子

型修飾子に以下を指定して、変数、関数、およびパラメーターの宣言をより詳細化するために使用します。

- オブジェクトの値は変更可能かどうか
- オブジェクトの値は、常にレジスターからではなく、メモリーから読み取る必要があるかどうか
- 複数のポインターは変更可能なメモリー・アドレスにアクセスできるかどうか

XL C/C++ は、次の型修飾子を認識します。

- IBM __align
- const
- restrict
- volatile

標準 C++ は、型修飾子 const および volatile を *cv 修飾子* として参照します。どちらの言語においても、cv 修飾子は、左辺値である式でのみ意味があります。

const および volatile キーワードをポインターで使用する場合、修飾子の配置は非常に重要です。それが修飾されるポインター自身か、あるいはポインターが指すオブジェクトかを決定するからです。volatile または const として修飾したいポインターの場合、* 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で、型修飾子が * 演算子の後に続いていない場合、型指定子および修飾子を任意の順序で使用できます。例えば、volatile データ・オブジェクトを指すポインターの場合、以下のようになります。

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

または

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

const データ・オブジェクトを指すポインターの場合、以下のようになります。

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

または

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

次の例で、これら宣言の意味を対比します。

宣言	説明
`const int * ptr1;`	定数整数を指すポインターを定義します。指される値は変更できません。
`int * const ptr2;`	整数を指す定数ポインターを定義します。この整数は変更できますが、ptr2 はそれ以外のものを指すことはできません。
`const int * const ptr3;`	定数整数を指す定数ポインターを定義します。指される値も、ポインター自身も変更できません。

宣言に複数の修飾子を入れることができます。コンパイラーは、重複した型修飾子を無視します。

型修飾子はユーザー定義型には適用できませんが、ユーザー定義型から作成されたオブジェクトにのみ適用できます。したがって、次の宣言は無効です。

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

ただし、変数 (1 つ以上) が型の定義内で宣言される場合、型修飾子をステートメントの先頭に置くか、または変数宣言子の前に置くと、型修飾子はその変数に適用できます。したがって、

```
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;
```

どちらの例でも、volatile 修飾子は、構造体変数 group に適用されます。

型修飾子が構造体、C++ クラス C++ 、または共用体変数に適用される場合、それらは構造体、クラス、または共用体のメンバーにも適用されます。

関連資料:

116 ページの『ポインター』

358 ページの『定数および volatile メンバー関数』

__align 型修飾子 (IBM 拡張)

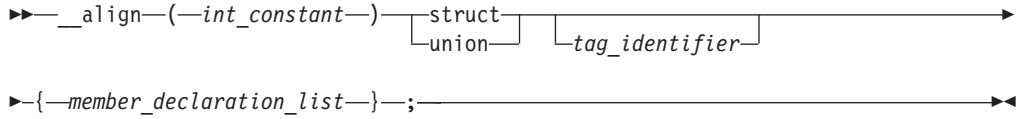
__align 修飾子は、集合体または静的 (またはグローバル) 変数に明示的位置合わせを指定できるようにした言語拡張機能です。ここで指定するバイト境界は、集合体

のメンバーではなく集合体全体の位置合わせに影響を与えます。 __align 修飾子は、他の集合体定義の中にネストされている集合体定義には適用されますが、集合体の個々のエレメントには適用されません。パラメーターおよび自動変数については、位置合わせ指定は無視されます。

宣言は、次のいずれかの形式をとります。

単純変数のための __align 修飾子の構文

構造体または共用体のための __align 修飾子の構文



ここで、*int_constant* は、バイト位置合わせ境界を示す正の整数値です。有効な値は、32768 までの 2 の累乗です。

以下の制限が適用されます。

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

__align 修飾子の使用例

静的変数またはグローバル変数への **__align** の適用:

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

int main()
{...}

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

// Error
int __align(128) functionB( );

// Error
typedef int __align(128) T;

// Error
__align enum C {a, b, c};
```

集合体メンバーに影響を与えない、集合体タグの位置合わせと埋め込みのための **__align** の適用：

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


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

構造体または共用体への **__align** の適用 (この構造体または共用体を使用する集合体のサイズと位置合わせが影響を受ける場合):

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

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

// Error: alignment of variable is smaller than alignment of type
struct S __align(64) svar;

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

配列への **__align** の適用：

次の例では、arrayA のみが 64 バイト境界上に位置合わせされ、その配列内のエレメントは AnyType の位置合わせに従って位置合わせされます。埋め込みは、配列の開始前に適用されるため、配列メンバー自体のサイズには影響しません。

```
AnyType __align(64) arrayA[10];
```

変数位置合わせのサイズが型位置合わせのサイズと異なる場合の **__align** の適用：

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

// Error: alignment of variable is smaller than alignment of type.
struct S __align(32) s1;

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

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

// Error
int __align(1) s4;

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

関連資料:

146 ページの『aligned 変数属性』

183 ページの『__alignof__ 演算子 (IBM 拡張)』


「XL C/C++ 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

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、割り当てステートメントの左側では使用できません。

C const オブジェクトを定数式で使用することはできません。明示ストレージ・クラスがないグローバル const オブジェクトは、デフォルトで extern と見なされます。

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

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

C++ では、明示ストレージ・クラスがないグローバル const オブジェクトは、デフォルトで static と見なされ、内部リンケージ付きです。

```
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++ */
```

そのリンケージは内部リンケージと想定されるので、const オブジェクトはヘッダー・ファイルの中で、C よりも、C++ でのほうが容易に定義できます。 C++

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

関連資料:

518 ページの『#define ディレクティブ』

362 ページの『this ポインター』

restrict 型修飾子

ポインターは、メモリー内のロケーションのアドレスです。複数のポインターがメモリーの同じチャンクにアクセスし、プログラムの途中でそのチャンクを変更することができます。 restrict (あるいは __restrict または __restrict__)[1] 型修飾子は、ポインター型に適用して、restrict で修飾されたポインターを形成できます。restrict で修飾されたポインターを指定する方法を示す、オブジェクトの宣言に関連付けられたブロックの実行中に、restrict で修飾されたポインターを通じてアドレス指定されたメモリーを変更することはできません。また、ポインターが const で修飾された型を指していない場合、メモリーには、このポインターを通じてのみアク

セスできます。 コンパイラーは、restrict で修飾されたポインターを含むコードを、誤った振る舞いを引き起こす可能性がないようにするための唯一の方法で最適化することを選択できます。 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るようにするのは、プログラマーの責任です。そうでないと、動作は未定義になる可能性があります。

メモリーの特定のチャンクが変更されない場合、そのチャンクには、複数の制限付きポインターで別名を割り当てることができます。以下の例は、制限付きポインターを foo() のパラメーターとして示し、無変更のオブジェクトに 2 つの制限付きポインターによって別名をどのように割り当てるかを示しています。

```
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、ネストされた等価のブロックの間に区別はありません。

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

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

注:

1. restrict 修飾子は、次のキーワード (すべて、同じ意味です) によって表されます。
   - C restrict キーワードは、**xlc** または **c99** を使用するか、あるいは**-qlanglvl=stdc99** オプションまたは **-qlanglvl=extc99** オプション、**-qkeyword=restrict** オプションを使用したコンパイルで認識されます。__restrict キーワードおよび __restrict__ キーワードは、すべての言語レベルで認識されます。
   - C++ IBM restrict、__restrict および __restrict__ キーワードは、デフォルトで認識されます。
2. C **-qrestrict** オプションを使用することと、指定された関数内のポインター・パラメーターに restrict キーワードを追加することは等価です。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

 「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照

 「XL C/C++ コンパイラー・リファレンス」の中の『-qrestrict』 を参照

volatile 型修飾子

volatile 修飾子は、データ・オブジェクトへのメモリー・アクセスの整合性を保持します。Volatile オブジェクトは、その値が必要になるたびに、メモリーから読み取られ、値が変更されるたびにメモリーに書き戻されます。volatile 修飾子は、その値がコンパイラーが制御できない、または検出できない様々な方法で変更されるデータ・オブジェクト (システム・クロックや別のプログラムによって更新される変数など) を宣言します。これにより、コンパイラーは、オブジェクトの値をメモリーから読み取るのではなく、レジスターに保管し、レジスターから読み取り直すので、オブジェクトを参照するコードを最適化できなくなります。レジスターの中では、オブジェクトの値は変更される可能性があります。

volatile で修飾された左辺値式にアクセスすると、副次作用が発生します。副次作用とは、実行環境の状態が変わることを意味します。

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。型「volatile T を指すポインター」の値を型「T を指すポインター」のオブジェクトに割り当てるには、明示キャストを使用しなければなりません。以下は、volatile オブジェクトの有効な例を示しています。

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

C シグナル処理関数は、sig_atomic_t 型変数に volatile が宣言されている場合、その変数に値を保管できます。これは、シグナル処理関数は、静的ストレージ期間の変数にアクセスできないという規則の例外です。

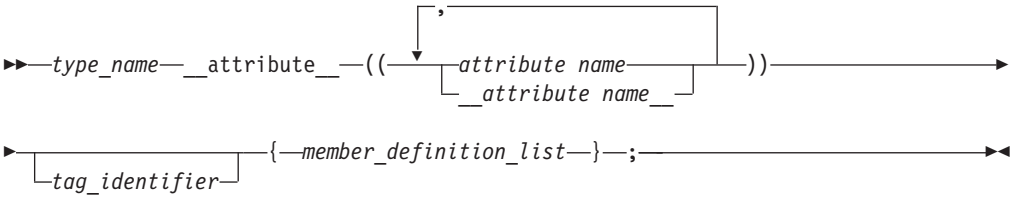
1 つの項目が、const および volatile の両方になり得ます。この場合、その項目はそれ自体のプログラムによって正当に変更することはできないが、ある種の非同期処理によって変更することはできます。

型属性 (IBM 拡張)

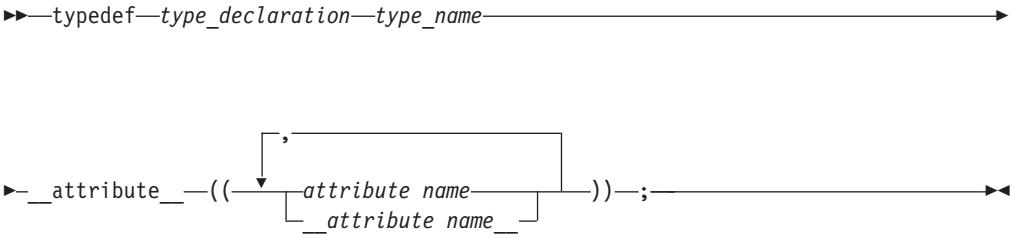
型属性は、GNU C/C++ コンパイラーで開発されたプログラムのコンパイルを容易にするために提供された言語拡張機能です。この言語機能により、名前付き属性を使用して、データ・オブジェクトの特殊なプロパティーを指定することができます。その型を持つと宣言された変数は、それらに適用される属性を持ちます。

型属性は、キーワード __attribute__、その後に続く属性名、さらに、その属性名が必要とする追加引数によって指定されます。変形はありますが、型属性の一般的な構文形式は次のとおりです。

型属性の構文



型属性の構文 - typedef 宣言



attribute name は、前後に 2 つの下線文字を付けても付けなくても指定できます。ただし、2 つの下線文字を使用すると、同じ名前のマクロと名前が競合する可能性が低くなります。サポートされない属性名については、XL C/C++ コンパイラーは診断メッセージを出して、その属性の指定を無視します。同じ属性指定に複数の属性名を指定することができます。

次の型属性がサポートされています。

- 『aligned 型属性』
- 109 ページの『packed 型属性』
- 110 ページの『may_alias 型属性』
- 111 ページの『transparent_union 型属性 (C のみ) 』

関連資料:

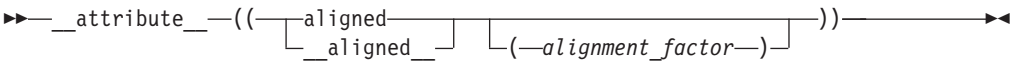
145 ページの『変数属性 (IBM 拡張)』

286 ページの『関数属性 (IBM 拡張)』

aligned 型属性

aligned 型属性を使用すると、typedef 宣言で作成された構造体、C++ クラス C++ 、共用体、列挙型、または他のユーザー定義型に対し、デフォルトの位置合わせモードをオーバーライドして、最小の位置合わせ値をバイト数で指定することができます。 aligned 属性は、一般的には、その属性が適用される型を宣言された変数の位置合わせを大きくするために使用します。

aligned 型属性の構文



alignment_factor はバイト数ですが、定数式として指定され、その式が評価されて 2 の正の累乗になります。最大 1048576 バイトまでの値を指定できます。位置合わせ係数 (およびそれを囲む括弧) を省略すると、コンパイラーは自動的に 16 バイトを使用します。最大より大きい位置合わせ係数を指定すると、属性指定は無視され、コンパイラーは、有効になっているデフォルトの位置合わせを使用します。

ここで指定された位置合わせ値は、その型の全インスタンスに適用されます。また、この位置合わせ値は、全体としての変数に適用されます。変数が集合体の場合、位置合わせ値は全体としての集合体に適用されます。集合体の個々のメンバーに適用されるのではありません。

例

次のすべての例で、aligned 属性は構造体型 A に適用されます。a は型 A の変数として宣言されているため、型 A として宣言されているその他のインスタンスと同様に、a もこの位置合わせ指定を受け取ります。

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

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

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

関連資料:

102 ページの『__align 型修飾子 (IBM 拡張)』

146 ページの『aligned 変数属性』

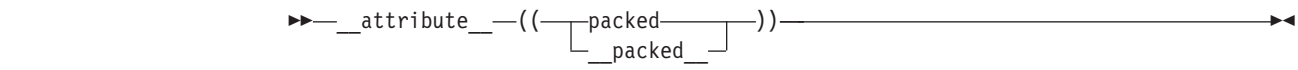
183 ページの『__alignof__ 演算子 (IBM 拡張)』

「XL C/C++ 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

packed 型属性

packed 型属性は、構造体、クラス、共用体、または列挙型のメンバーに対して最小の位置合わせを使用することを指定します。 構造体型、クラス型、または共用体型の場合、位置合わせは、メンバーに対しては 1 バイト、ビット・フィールド・メンバーに対しては 1 ビットです。列挙型の場合、位置合わせは、列挙型の値の範囲を収納する最小のサイズです。 その型の全インスタンスの全メンバーが最小の位置合わせを使用します。

packed 型属性の構文



packed 型属性は、aligned 型属性と違って typedef 宣言では使用できません。

関連資料:

102 ページの『__align 型修飾子 (IBM 拡張)』

149 ページの『packed 変数属性』

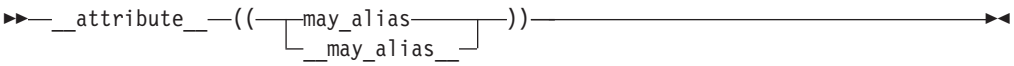
183 ページの『__alignof__ 演算子 (IBM 拡張)』

 「XL C/C++ 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

may_alias 型属性

型に対して may_alias 型属性を指定できます。これによりその型の間接参照演算子を持つ左辺値は、char 型と同様に、任意の型のオブジェクトに別名を割り当てることができます。 may_alias 属性を持つ型は、型ベースの別名割り当て規則には従いません。

may_alias 型属性の構文



may_alias 型属性は、以下の方法で指定できます。

```
struct __attribute__((__may_alias__)) my_struct {} *ps;
typedef long __attribute__((__may_alias__)) t_long;
typedef struct __attribute__((__may_alias__)) my_struct {} t_my_struct;
```

-qalias=noansiを指定する代わりに、型に may_alias 型属性を指定すれば、その型の左辺値を含む式をコンパイルするときに、ANSI 別名割り当て規則に違反することが可能になります。次に例を示します。

```
#define __attribute__(x)     // Invalidates all __attribute__ declarations
typedef long __attribute__((__may_alias__)) t_long;

int main (void){
  int i = 42;
  t_long *pa = (t_long *) &i;
  *pa = 0;
  if (i == 42)
    return 1;
  return 0;
}
```

このコードを、**-O4** などの高最適化レベルで、**-qalias=ansi** オプションを指定してコンパイルする場合、実行可能プログラムは 1 を返します。 ANSI 別名割り当て規則に従えば、左辺値 *pa は long 型であるので、左辺値 *pa への代入により、int 型である i の値を変更することはできません。

#define __attribute__(x) ステートメントを削除し、以前と同じオプションを指定してコードをコンパイルすると、実行可能プログラムは 0 を返します。*pa の型は long __attribute__((__may_alias__)) であるので、*pa は任意の型の他の任意のオブジェクトに別名を割り当てることができ、左辺値 *pa への代入により i の値を 0 に変更することができます。

may_alias 型属性を使用すると、あまり従来の規則に縛られていない別名割り当て関係となるため、コンパイラー・オプション **-qalias=ansi** を使用する場合と比較して、最適化できる機会が多くなります。

C この属性は、extc89、extc99、extended、 および extc1x 言語レベルでサポートされます。 C

C++ この属性は、extended および extended0x 言語レベルでサポートされます。 C++

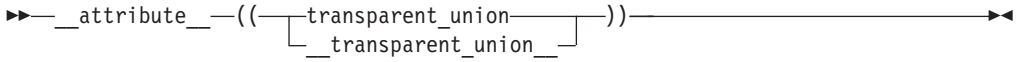
関連資料:

118 ページの『型ベースの別名割り当て』

transparent_union 型属性 (C のみ)

transparent_union 属性を共用体定義または共用体 typedef に適用した場合は、その共用体を*透過共用体* として使用できることを示します。ある関数仮パラメーターの型が透過共用体であるときに、その関数が呼び出された場合は、その透過共用体は、その共用体のいずれかのメンバーの型に一致するすべての型の引数を、明示キャストなしで受け入れることができます。この関数仮パラメーターへの引数は、該当の共用体型の最初のメンバーの呼び出し規則に従って、透過共用体に渡されます。したがって、この共用体のすべてのメンバーについて、マシン表現が同じで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の中で使用すると便利です。

transparent_union 型属性の構文



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

transparent_union 型属性が、ネストされた共用体の外部共用体に適用される場合は、その内部共用体のサイズ (つまりその最大メンバー) 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っているかどうかが判別されます。次に例を示します。

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

この例では、属性は無視されます。なぜなら、共用体 u_t の最初のメンバー (このメンバー自体も 1 つの共用体です) のマシン表現が 2 バイトであるのに対し、共用体 u_t のその他のメンバーの型は int であり、したがってマシン表現は 4 バイトだからです。

共用体の中の構造体であるメンバーにも、同じ原理が適用されます。型属性 transparent_union が適用された共用体メンバーが struct である場合は、メンバーではなくその struct 全体のマシン表現が考慮に入れられます。

共用体のすべてのメンバーは、共用体の最初のメンバーとマシン表現が同じでなければなりません。つまり、すべてのメンバーが、共用体の最初のメンバーと同じ量のメモリーで表現できることが必要です。最初のメンバーのマシン表現は、共用体の残りのメンバーすべてについて、個々のメンバー最大メモリー・サイズを表しま

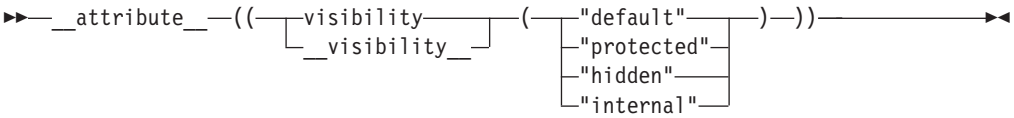
す。例えば、transparent_union 属性が適用された共用体の最初のメンバーの型が int であるとすれば、以降のメンバーはすべて最大 4 バイトで表現できることが必要です。この透過共用体では、1、2、または 4 バイトで表現できるメンバーは有効と見なされます。

浮動小数点型 (float、double、float_Complex 、または double _Complex) またはベクトル型は、透過共用体のメンバーでかまいませんが、最初のメンバーであってはなりません。この場合も、透過共用体のすべてのメンバーのマシン表現が、最初のメンバーと同じでなければならないという制約条件は変わりません。

visibility 型属性 (C++ のみ)

visibility 型属性を使用することで、あるモジュール内で定義されている構造体/共用体/クラスまたは列挙を、他のモジュール内で参照または使用できるかどうかおよびその方法を制御できます。 visibility 属性は、外部リンケージを持つ型のみに影響を与えます。この機能を使用することで、共有ライブラリーを小さくして、シンボル競合の可能性を減らすことができます。詳しくは、「XL C/C++ 最適化およびプログラミング・ガイド」の『visibility 属性の使用』を参照してください。

visibility 型属性の構文



例

以下の例で、クラス A の visibility 属性は保護され、列挙 E の visibility 属性は隠されます。

```
class __attribute__((visibility("protected"))) A {};
enum __attribute__((visibility("hidden"))) E {e1,e2} e;
```

関連資料:

9 ページの『外部リンケージ』

151 ページの『visibility 変数属性』

visibility

323 ページの『visibility 名前空間属性 (IBM 拡張)』

「XL C/C++ 最適化およびプログラミング・ガイド」の中の『visibility 属性の使用』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qvisibility』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qmkshrobj』 を参照

「XL C/C++ コンパイラー・リファレンス」の『#pragma GCC visibility push、#pragma GCC visibility pop (IBM 拡張)』を参照

第 4 章 宣言子

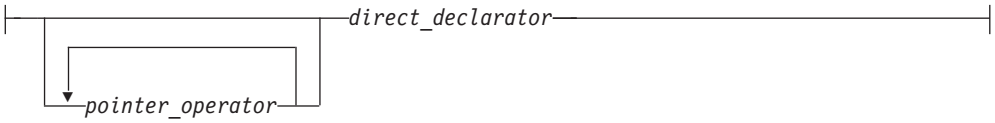
このセクションでは、引き続きデータ宣言について説明します。ここには、型名、ポインター、配列、C++ 参照 C++ 、初期化指定子、および変数属性についての情報が記載されています。

宣言子の概要

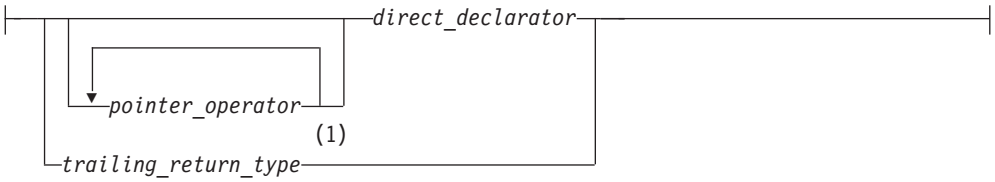
宣言子は、オブジェクト、関数、または C++ 参照 C++ を宣言の一部として宣言します。

宣言子の形式は、以下のとおりです。

宣言子の構文 (C のみ) :



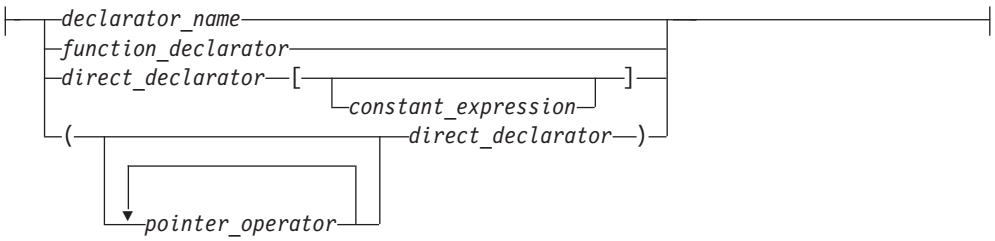
宣言子の構文 (C++ のみ) :



注:

1 C++11

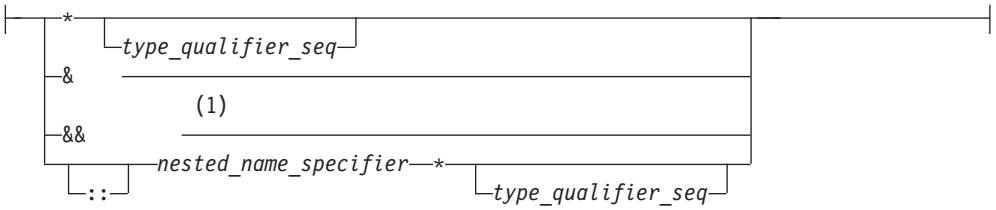
direct declarator:



ポインター演算子 (C のみ) :

宣言子名　(C のみ)　:

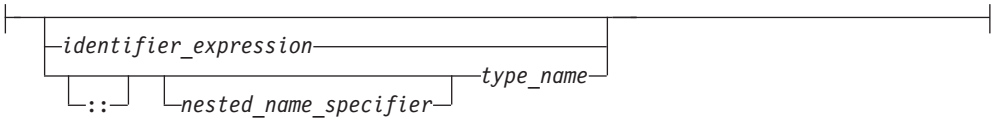
ポインター演算子 (C++ のみ) :



注:

1　C++11

宣言子名 (C++ のみ) :



注:

- *type_qualifier_seq* は、1 つの型修飾子、または複数の型修飾子の組み合わせを表します。型修飾子について詳しくは、 101 ページの『型修飾子』を参照してください。
- C++ *nested_name_specifier* は、修飾された ID 式です。*identifier_expression* は、修飾された ID であっても、修飾されていない ID であってもかまいません。 C++

関数宣言子について詳しくは、 280 ページの『関数宣言子』を参照してください。 C++11 後置戻り型について詳しくは、 283 ページの『後置戻り型 (C++11)』を参照して下さい。 C++11

以下の型は、*派生宣言子 型*と呼ばれています。このセクションでは、それらの型について説明します。

- 116 ページの『ポインター』
- 123 ページの『配列』
- 126 ページの『参照 (C++ のみ)』

IBM さらに、GNU C および C++ との互換性を保つために、XL C/C++ では、*変数属性* を使用してデータ・オブジェクトのプロパティーを変更することができます。これらは通常、宣言で宣言子の一部として指定されるため、これらについては、　145 ページの『変数属性 (IBM 拡張)』で説明します。 IBM

関連資料:

127 ページの『初期化指定子』

101 ページの『型修飾子』

宣言子の例

次の表に、宣言で使われる宣言子を示します。

宣言	宣言子	説明
`int owner;`	`owner`	`owner` は、整数データ・オブジェクトです。
`int *node;`	`*node`	`node` は、整数データ・オブジェクトを指すポインターです。
`int names[126];`	`names[126]`	`names` は、126 個の整数エレメントの配列です。
`volatile int min;`	`min`	`min` は、揮発性整数です。
`int * volatile volume;`	`* volatile volume`	`volume` は、整数を指す揮発性ポインターです。
`volatile int * next;`	`*next`	`next` は、揮発性整数を指すポインターです。
`volatile int * sequence[5];`	`*sequence[5]`	`sequence` は、揮発性整数データ・オブジェクトを指す 5 つのポインターの配列です。
`extern const volatile int clock;`	`clock`	`clock` は、静的ストレージ期間および外部リンケージが指定された定数および volatile 整数です。

関連資料:

101 ページの『型修飾子』

199 ページの『配列添え字演算子 [ ]』

173 ページの『スコープ解決演算子 :: (C++ のみ)』

280 ページの『関数宣言子』

型名

型名 は、オブジェクトを宣言せずに何かを指定する必要がある場合のコンテキストに必要です。例えば、明示的にキャスト式を書く場合、または型に `sizeof` 演算子を適用する場合などです。構文的には、データ型の名前は、その型の関数またはオブジェクトの宣言と同じですが、ID はありません。

型名を正しく読んだり、書いたりするには、構文内に「仮想の」ID を入れ、型名をより単純なコンポーネントに分割してください。例えば、`int` は型指定子ですが、これは、宣言内で常に ID の左に現れます。この単純な場合には、仮想 ID は不要です。ただし、`int *[5]` (`int` への 5 つのポインターの配列) も型の名前です。型指定子 `int *` は、常に ID の左に現れ、配列添え字演算子は常に右に現れます。この場合には、仮想 ID があると、型指定子を見分けやすくなります。

一般的な規則として、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に現れ、型指定子、型修飾子、または間接演算子の右側に現れます。型名宣言では、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使用できま

す。これらは、通常の演算子優先順位に従ってバインドされます。つまり、間接演算子は添え字演算子または関数呼び出し演算子より優先順位は低く、添え字演算子と関数呼び出し演算子の優先順位のランキングは同じです。括弧を使用して間接演算子のバインドを制御することができます。

型名の中に型名を持つことができます。例えば、関数型において、パラメーター型の構文は関数型名内にネストします。同じ経験法則が再帰的に適用されます。

以下の構造体は、型命名規則の適用を示しています。

表 23. 型名

構文	説明
`int *[5]`	`int` を指す 5 つのポインターの配列
`int (*)[5]`	5 つの整数の配列を指すポインター
`int (*)[*]`	整数の数が指定されていない可変長配列を指すポインター
`int *()`	`int` を指すポインターを戻す、パラメーター指定がない関数
`int (*)(void)`	`int` を戻す、パラメーターがない関数
`int (*const [])(unsigned int, ...)`	`int` を戻す関数を指す定数ポインターの数が指定されていない配列。各関数は、型 `unsigned int` を指定する 1 つのパラメーターと、数が指定されていない他のパラメーターを取ります。

コンパイラーは関数指定子を関数を指すポインターに変えます。この動作は関数呼び出しの構文を単純化します。

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

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

関連資料:

224 ページの『演算子優先順位と結合順序』

226 ページの『式と優先順位の例』

491 ページの『typename キーワード』

170 ページの『括弧で囲んだ式 ( )』

ポインター

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

ポインターに共通な用法を次に示します。

- リンクされたリスト、ツリー、キューなどの動的データ構造にアクセスする。

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

型修飾子 volatile および const の配置は、ポインター宣言のセマンティクスに影響することに注意してください。 * の前にどちらの修飾子が現れても、その宣言子は、型で修飾されたオブジェクトを指すポインターを記述します。* と ID の間にどちらの修飾子が現れても、その宣言子は、型で修飾されたポインターを記述します。

次の表に、いくつかのポインター宣言の例を示します。

表 24. ポインター宣言

宣言	説明
`long *pcoat;`	pcoat は、型 long を持つオブジェクトを指すポインターです。
`extern short * const pvolt;`	pvolt は、short 型が指定されたオブジェクトを指す定数ポインターです。
`extern int volatile *pnut;`	pnut は、volatile 修飾子が指定された int オブジェクトを指すポインターです。
`float * volatile psoup;`	psoup は、float 型が指定されたオブジェクトを指す volatile ポインターです。
`enum bird *pfowl;`	pfowl は、型 bird の列挙型オブジェクトを指すポインターです。
`char (*pvish)(void);`	pvish は、パラメーターを取らずに char を戻す関数を指すポインターです。
C++11 `nullptr_t pnull;`	pnull は、有効なオブジェクトまたは関数を指していない、NULL ポインターです。

関連資料:

101 ページの『型修飾子』
136 ページの『ポインターの初期化』
120 ページの『ポインターの互換性 (C のみ)』
160 ページの『ポインター型変換』
180 ページの『アドレス演算子 &』
181 ページの『間接演算子 *』
307 ページの『関数を指すポインター』

ポインター演算

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

- 増分と減分
- 加算および減算

- 比較
- 割り当て

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

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

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

IBM ポインター演算は、ベクトル型に対するポインターに定義します。次の場合、

```
vector unsigned int *v;
```

式 v + 1 は、v の次のベクトルを指すポインターを表します。 IBM

関連資料:

177 ページの『増分演算子 ++』

123 ページの『配列』

178 ページの『減分演算子 --』

165 ページの『第 6 章 式と演算子』

型ベースの別名割り当て

-qalias=ansi オプションが有効な場合 (デフォルトで有効になります)、コンパイラーは、C および C++ 標準による、型ベースの別名割り当て規則に従います。この規則では、ANSI 別名割り当て規則としても知られているように、ポインターは、同

じ型のオブジェクトまたは互換型に対してのみ間接参照が可能であると定めています。[1] 互換性のない型にポインターをキャストしてから、それを間接参照するという、一般的なコーディング方法はこのルールに違反しています。 (ただし、char ポインターは、この規則に対し例外であることに注意してください。)

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

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

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

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

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

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

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

---

1. C 標準では、オブジェクトの保管値は、次のいずれかの型を持つ左辺値からのみアクセスできると明記されています。
   - 宣言されたオブジェクト型、
   - 宣言されたオブジェクト型の修飾バージョン、
   - 宣言されたオブジェクト型と対応する、符号付きまたは符号なしの型、
   - 宣言されたオブジェクト型の修飾バージョンと対応する、符号付きまたは符号なしの型、
   - メンバー (さらに、副集合体、または包含されている共用体のメンバーを含む) の中に前述の型のいずれかが含まれる構造体または共用体の型または
   - 文字型

   C++ 標準では、プログラムが以下のいずれか以外の型の左辺値によってオブジェクトの保管値にアクセスしようとした場合、その動作は未定義です。
   - 動的型のオブジェクト、
   - 動的型のオブジェクトの cv 修飾バージョン、
   - 動的型のオブジェクトと対応する、符号付きまたは符号なしの型、
   - 動的型のオブジェクトの cv 修飾バージョンと対応する、符号付きまたは符号なしの型、
   - メンバー (さらに、副集合体、または包含されている共用体のメンバーを含む) の中に前述の型のいずれかが含まれる構造体または共用体の型
   - 動的型のオブジェクトの基底クラス型 (おそらく cv 修飾された) である型、
   - char または符号なし char 型

関連資料:

110 ページの『may_alias 型属性』

211 ページの『reinterpret_cast 演算子 (C++ のみ)』

「XL C/C++ コンパイラー・リファレンス」の中の『-qalias』を参照

ポインターの互換性 (C のみ)

同じ型修飾子を指定された 2 つのポインター型は、それらが互換型のオブジェクトを指している場合、互換性があります。 2 つの互換ポインター型に関する複合型は、複合型への、同じように修飾されたポインターです。

次の例では、割り当て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、割り当て演算用の互換性がない宣言を示します。

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

NULL ポインター

NULL ポインターは、そのポインターが有効なオブジェクトや関数を指していないことを示すために、NULL ポインター定数と呼ばれる予約済みの値を保有しています。NULL ポインターは、次のような場合に使用します。

- ポインターを初期化する。
- 長さが不明のリストの終わりなどの条件を示す。
- ポインターを関数から戻す際のエラーを示す。

NULL ポインター定数は、ゼロに評価される整数定数式です。例えば、NULL ポインター定数には、0、0L、または型 (void *)0 にキャストできる式などがあります。 C++11 C++11では、新しい NULL ポインター定数 nullptr を定義しています。この定数は、任意のポインター型、pointer-to-member 型、またはブール型にのみ変換できます。 C++11

NULL ポインター定数には、以下のいずれかの値を指定できます。

- 0
- NULL
- C++11 nullptr C++11

注: NULL はマクロです。使用する前に定義する必要があります。

NULL ポインター定数

0 値 0 を使用した整数定数式、または (void *)0 にキャストされる式を NULL ポインター定数として使用できます。

NULL マクロ NULL および値 0 は、NULL ポインター定数として等価ですが、NULL の方が、ポインターの定数として使用する目的がより明確に示されます。

C++11

nullptr nullptr は、明示的なNULL ポインター定数です。 C++ では、0 または NULL を使用して NULL ポインターを初期化する場合、以下の問題があります。

- 多重定義された関数の場合、NULL ポインターと整数の 0 を区別できない。例えば、f(int) および f(char*) の 2 つの多重定義関数がある場合、0 はポインター型でなく整数型に変換されるため、呼び出し f(0) は f(int) に解決されます。例 1 を参照してください。
- NULL ポインター定数には、タイプ・セーフな名前がない。多重定義関数とエラー検出で、マクロの NULL と 0 定数を区別できません。

NULL ポインター定数の問題を解決するために、C++11 では、新しいキーワード nullptr が導入されています。nullptr 定数は、多重定義関数で、整数 0 と区別できます。例 2 を参照してください。

nullptr 値を持つ NULL ポインター定数には、以下の特性があります。

- 任意のポインター型または pointer-to-member 型に変換できる。
- 他の任意の型へ暗黙的に変換することはできない (ブール型を除く)。
- 算術式では使用できない。
- 整数 0 と比較できる。
- 関係式で使用して、std::nullptr_t 型のポインターまたはデータと比較できる。

すべてのポインター型および比較式で初期化指定子として nullptr を使用する方法については、『例 3』を参照してください。

注: 引き続き、値 0 または NULL をポインターに割り当てることもできます。

nullptr キーワードは、型 decltype(nullptr) の定数右辺値を指定します。typedef 式は typedef decltype(nullptr) nullptr_t です。ここで nullptr_t は、<cstddef> に定義された decltype(nullptr) の typedef です。非型テンプレート・パラメーターと引数で、型 std::nullptr_t を使用できます。非型テンプレート・パラメーターの型が std::nullptr_t である場合、対応する引数の型は std::nullptr_t でなければなりません。 例 4 を参照してください。非型テンプレート・パラメーターの型が以下のいずれかである場合、対応する非型テンプレート引数の型に std::nullptr_t を使用できます。以下の最後の3 つの型の場合に、NULL ポインター変換が行われます。

- std::nullptr_t
- ポインター
- pointer-to-member
- bool

例外処理で nullptr を使用する場合、throw 引数と catch 引数に注意してください。ハンドラーの型が cv T または const T& である場合、ハンドラーは、型 E の例外オブジェクトを突き合わせます。T は、ポインター型または pointer-to-member 型であり、E の型は std::nullptr_t です。 例 5 を参照してください。

例

例 1

この例は、多重定義関数における NULL 定数の適切でない使用について説明しています。

```
#include <stdio.h>

void func(int* i){
   printf("func(int*)¥n");
}

void func(int i){
   printf("func(int)¥n");
}

int main(){
   func(NULL);
}
```

main 関数によって func(int* i) を呼び出す場合を想定します。この場合、定数 NULL が整数 0 と等しいため、main 関数は func(int* i) ではなく func(int i) を呼び出します。定数 0 は、func(int i) が存在しない場合にのみ、(void*)0 に暗黙的に変換されます。

例 2

この例では、多重定義関数でどのように nullptr が使用されるかを示しています。

```
void f( char* );
void f( int );
f( nullptr );     // calls f( char* )
f( 0 );           // calls f( int )
```

例 3

以下の式は、nullptr 定数の正しい使用法と誤った使用法を説明しています。

```
char* p = nullptr;     // p has the null pointer value
char* p1 = 0;          // p has the null pointer value
int a = nullptr;       // error
int a2 = 0;            // a2 is zero of integral type
if( p == 0 );          // evaluates to true
if( p == nullptr );    // evaluates to true
if( p );               // evaluates to false
if( a2 == 0 );         // evaluates to true
if( a2 == nullptr );   // error, no conversion
if( nullptr );         // OK, conversion to the bool type
if( nullptr == 0 );    // OK, comparison with 0
nullptr = 0;           // error, nullptr is not an lvalue
nullptr + 2;           // error
```

例 4

次の例では、非型テンプレート・パラメーターまたは引数で、型 std::nullptr_t を使用できることを示しています。

```
typedef decltype(nullptr) nullptr_t;
template <nullptr_t> void fun1(); // non-type template parameter

fun1<nullptr>();      // non-type template arguments
fun1<0>();            // error. The corresponding argument must be of type
                      // std::nullptr_t if the parameter is of type std::nullptr_t.

template <int* p> void fun2();
fun2<nullptr>();      //Correct

template<typename T> void h( T t );
h( 0 );                   // deduces T = int
h( nullptr );             // deduces T = nullptr_t
h( (float*) nullptr );    // deduces T = float*
```

例 5

次の例では、例外処理で nullptr を使用する方法を示しています。

```
int main() {
try {
   throw nullptr;
} catch(int* p) {  // match the pointer.
   return p == 0;
}
}
```

C++11 ◀

配列

配列 とは、メモリー内で連続して割り振られた、同じデータ型のオブジェクトの集合です。配列内の個々のオブジェクトは *エレメント* と呼ばれ、それらは、配列内のそれぞれの位置によってアクセスされます。サブスクリプト演算子 ([]) は、配列エレメントへの指標を作成する方法を提供します。このアクセス方式は、*指標方式* または *添え字方式* と呼ばれます。配列では、各エレメントに実行されるステートメント群をループに入れて、そのループを配列内の各エレメントに繰り返すようにできるので、反復タスクのコーディングが容易になります。

C 言語および C++ 言語は、配列型に対して、限定された組み込みサポートを提供します。すなわち、個々のエレメントの読み取りおよび書き込みのサポートです。ある配列を別の配列に割り当てたり、2 つの配列を等価のものかどうか比較したり、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。

配列型はエレメントの型から派生し、このことは、*配列型派生* と呼ばれます。配列オブジェクトが不完全型の場合、配列型も不完全と見なされます。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、関数を指すポインター配列は許可されます。

C++ 配列エレメントは、参照型にも抽象クラス型にもすることはできません。

配列宣言子には、ID と、その後に続く、オプションの *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。

配列添え字宣言子の構文

constant_expression は定数整数式であって、配列のサイズを示し、正でなければなりません。

宣言がブロック・スコープまたは関数スコープで行われる場合、配列添え字宣言子に対して定数式以外の式を指定することができます。その配列は、125 ページの『可変長配列』で説明しているように、*variable-length array* 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]
```

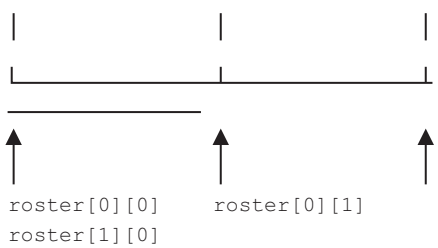
次の例では、int 型の 6 つのエレメントを含む 2 次配列を定義します。

```
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。

関連資料:

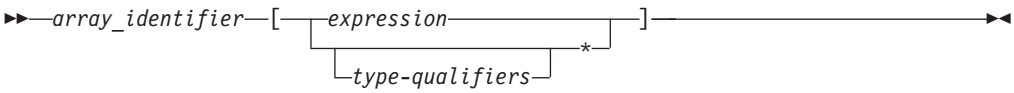
199 ページの『配列添え字演算子 [ ]』

137 ページの『配列の初期化』

可変長配列

可変長配列 (C99 の機能) は自動ストレージ期間の配列であって、その長さは実行時に決定されます。

可変長配列宣言子の構文



可変長配列のサイズが式ではなく * で示されている場合は、その配列のサイズは指定されていないものと見なされます。このような配列は完全型と見なされますが、関数プロトタイプ・スコープの宣言の中でしか使用できません。

可変長配列および可変長配列を指すポインターは、*可変変更型* と見なされます。可変変更型の宣言は、ブロック・スコープまたは関数プロトタイプ・スコープになければなりません。 extern ストレージ・クラス指定子を宣言される配列オブジェクトは、可変長配列型であってはなりません。 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りますが、実際の可変長配列であってはなりません。可変長配列は初期化することはできません。

注: C++ C++ アプリケーションでは、可変長配列の使用のために割り振られたストレージは、その中の関数が実行を終了するまでリリースされません。

可変長配列は、sizeof 式のオペランドとして使用できます。この場合は、オペランドは実行時に評価されます。したがって、可変長配列の各インスタンスのサイズがその存続期間中に変化しなかったとしても、そのサイズは整数定数でも定数式でもありません。

可変長配列は typedef ステートメントの中で使用できます。typedef 名は、ブロック・スコープだけを持ちます。配列の長さは、typedef 名が定義されたときに固定されるのであって、それが使用されるたびに固定されるのではありません。

関数仮パラメーターは可変長配列となることができます。必要なサイズ式は、関数定義の中で指定する必要があります。コンパイラーは、関数に入るときに、変更さ

れたパラメーターのサイズ式を評価します。以下のように、関数が可変長配列をパラメーターとして宣言されているとします。

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

この場合、可変長配列引数のサイズは、関数定義のサイズと一致していなければなりません。

IBM C++ C++ 拡張機能には、可変長配列型への参照に対するサポートは含まれていません。また、関数仮パラメーターによる可変長配列型への参照もサポートされない場合があります。

関連資料:

柔軟な配列メンバー

配列の互換性

2 つの互換配列型は、互換性のあるエレメント型を持たなければなりません。さらに、整数定数式であるサイズ指定子が各配列にある場合、両方のサイズ指定子は同じ定数値を持つ必要があります。例えば、以下の 2 つの配列の型には互換性がありません。

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

2 つの互換配列型から成る複合型は、複合エレメント型を持つ配列です。 2 つの互換配列の複合型は、以下の規則により決定されます。

1. 元の型の 1 つが既知の定数サイズの配列である場合、複合型はそのサイズの配列です。次に例を示します。

   ```
   // The composite type is char [42].
   char ex3[];
   char ex4[42];
   ```

2. そうでない場合は、元の型の 1 つが可変長配列の場合、複合型はその型になります。

関連資料:

9 ページの『外部リンケージ』

参照 (C++ のみ)

参照 は、オブジェクトまたは関数の別名または代替名です。オブジェクト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または関数のアドレスです。

左辺値参照型は、型指定子の後に参照修飾子 `&` または `bitand` を挿入することによって定義されます。C++11 右辺値参照型は、型指定子の後に参照修飾子 `&&` または `and` を挿入することによって定義されます。右辺値参照について詳しくは、**右辺値参照の使用 (C++11)**を参照してください。C++11 参照型には、左辺値参照型 C++11 および右辺値参照型C++11 があります。

関数の引数は、値によって渡されるため、関数呼び出しは、引数の実際の値を変更しません。関数が、引数の実際の値を変更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、引数は*参照によって渡す* (*値によって渡す* と対比) 必要があ

ります。参照による引数の受け渡しは、参照またはポインターのいずれかを使用して行うことができます。 C と異なり、C++ は、参照によって引数を渡したい場合には、ポインターの使用を強制しません。参照を使用する構文は、ポインターを使用する構文より、単純です。オブジェクトを参照によって渡すと、関数は、参照されているオブジェクトを変更するときに、関数のスコープ内でそのオブジェクトのコピーを作成する必要がありません。元の実際のオブジェクトのアドレスだけがスタックに書き込まれるのであって、オブジェクト全体が書き込まれるのではありません。

次に例を示します。

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

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

以下のタイプの参照が無効です。

- `NULL` への参照
- `void` への参照
- 無効なオブジェクトまたは関数への参照
- ビット・フィールドへの参照
- 参照に対する参照 C++11 (ただし、参照が縮約される場合を除く)。詳しくは、227 ページの『参照の縮約 (reference collapsing) (C++11)』を参照してください。
  C++11

また、参照の配列、参照を指すポインター、および参照に対する cv 修飾子を宣言することはできません。cv 修飾子が `typedef` またはテンプレート引数の推定を通じて導入される場合、cv 修飾子は無視されます。

関数の参照について詳しくは、307 ページの『関数を指すポインター』を参照してください。

関連資料:

140 ページの『参照の初期化 (C++ のみ)』

116 ページの『ポインター』

180 ページの『アドレス演算子 &』

302 ページの『参照による受け渡し (C++ のみ)』

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。特定の宣言で有効な初期化指定子は、初期化されるオブジェクトの型とストレージ・クラスに依存します。

初期化指定子は、`=` シンボルと、その後に続く、初期式 (*expression*)、またはコンマで区切ら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

マで区切られる必要があります。式のグループは中括弧で囲み、コンマで区切ることができます。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、中括弧 ({ }) はオプションです。初期化指定子の数は、初期化されるエレメントの数よりも多くてはいけません。初期式は、データ・オブジェクトの最初の値に評価されます。

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

```
int group = 3;
```

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

C++ *(C++ のみ)* 名前空間スコープで定数式以外の式を指定して、変数を初期化することができます。 C 定数式以外の式を使用して、グローバル・スコープの変数を初期化することはできません。

『初期化とストレージ・クラス』で、変数のストレージ・クラスに従った初期化の規則を説明しています。

130 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』 で、指定された初期化指定子を説明しています。これは、配列、構造体、および共用体を初期化するために使用できる C99 の機能です。

以下のセクションで、派生型に対する初期化を説明しています。

- 132 ページの『ベクトルの初期化 (IBM 拡張)』
- 133 ページの『構造体および共用体の初期化』
- 136 ページの『ポインターの初期化』
- 137 ページの『配列の初期化』
- 140 ページの『参照の初期化 (C++ のみ)』

関連資料:

346 ページの『クラス・オブジェクトの使用』

初期化とストレージ・クラス

このトピックには、以下の事項の説明が含まれています。

- 自動変数の初期化
- 静的変数の初期化
- 外部変数の初期化
- レジスター変数の初期化

自動変数の初期化

auto 変数 (関数仮パラメーターを除く) は、初期化できます。自動オブジェクトを明示的に初期化しない場合は、その値は確定できません。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。

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

注: C++11 C++11 では、キーワード auto は、ストレージ・クラス指定子として使用されなくなりました。代わりに、型指定子として使用されます。コンパイラーは、初期化指定子の式の型から auto 変数の型を推定します。詳しくは、 91 ページの『auto 型指定子 (C++11)』を参照してください。

静的変数の初期化

静的オブジェクトの初期化は、定数式、または既に 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 (多くの場合、定数式によって変更されます) を使用して実行できます。 静的 (または外部) 変数を明示的に初期化しない場合、それがポインターでなければ、その初期値は、該当する型の値ゼロになります。その変数がポインターの場合は、NULL に初期化されます。

C ブロック内の static 変数は、プログラム実行の前に 1 回初期化されるだけですが、初期化指定子を持つ auto 変数は、それが存在するようになるたびに初期化されます。

C++ ブロック内の静的変数は、制御の流れがブロック内の定義を介して初めて受け渡されるときに、動的に初期化できます。静的変数の動的初期化は、非定数式を使用して実行できます。 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。

外部変数の初期化

extern ストレージ・クラス指定子を持つオブジェクトは、C のグローバル・スコープまたは C++ の名前空間で初期化できます。extern オブジェクトの初期化指定子は次のどちらかでなければなりません。

- C 定義の一部として現れ、定数式によって初期値が記述されていなければならない。
- C++ 定義の一部として指定する必要がある。
- 静的ストレージ期間を持つ、既に宣言済みのオブジェクトのアドレスに変換されなければならない。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。

レジスター変数の初期化

関数仮パラメーターを除く register オブジェクトを初期化できます。自動オブジェクトを初期化しない場合は、その値は確定できません。初期値を提供する場合

は、初期値を表す式を C または C++ の有効な式にすることができます。オブジェクトの定義を含むプログラム・ブロックに入るたびに、オブジェクトはその初期値にセットされます。

関連資料:

55 ページの『auto ストレージ・クラス指定子』

56 ページの『静的ストレージ・クラス指定子』

57 ページの『extern ストレージ・クラス指定子』

59 ページの『register ストレージ・クラス指定子』

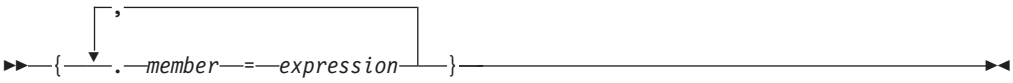
集合体型に対する、指定された初期化指定子 (C のみ)

集合体型 (配列、構造体、共用体など) 用に、*指定された初期化指定子* (C99 の機能) がサポートされています。指定された初期化指定子、すなわち、*指定子* は、初期化する特定のエレメントを示します。*指定子リスト* は、1 つ以上の指定子をコンマで区切ったリストです。直後に等号が記入された指定子リストによって、*指定* が構成されます。

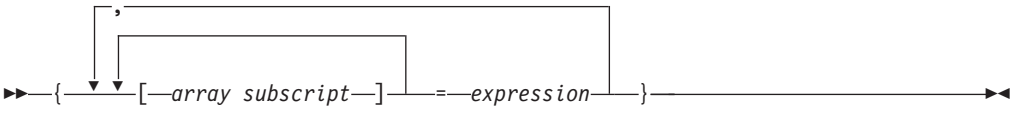
指定された初期化指定子は次の柔軟性を考慮しています。

- 集合体内のエレメントは、任意の順に初期化できます。
- 初期化指定子リストは、集合体の終わりに宣言されるエレメントのみでなく、集合体の任意の場所に宣言されるエレメントも省略できます。省略されたエレメントは、それが静的オブジェクトであるかのように初期化されます。すなわち、算術型は 0 に初期化され、ポインターは NULL に初期化されます。
- 多次元配列またはネストされた集合体に対する初期化指定子の大括弧の使用法が矛盾しているか、または不完全であるかを理解することが難しそうな場所では、指定子は、初期化するエレメントまたはメンバーをより明確に識別することができます。

構造体または共用体に対する指定子リストの構文



配列に対する指定子リストの構文



以下の例において、指定子は `.any_member` であり、指定された初期化指定子は `.any_member = 13` です。

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

次の例は、構造体変数 `klm` の 2 番目と 3 番目のメンバー `b` および `c` は、指定された初期化指定子で初期化されることを示しています。

```
struct xyz {
       int a;
       int b;
       int c;
       } klm = { .a = 99, .c = 100 };
```

次の例では、1 次元の配列 aa の 3 番目と 2 番目のエレメントは、それぞれ、 3 および 6 に初期化されます。

```
int aa[4] = { [2] = 3, [1] = 6 };
```

次の例は、最初の 4 つのエレメントと最後の 4 つのエレメントを初期化し、中間の 4 つのエレメントを省略しています。

```
      static short grid[3] [4] = { [0][0]=8, [0][1]=6,
                                   [0][2]=4, [0][3]=1,
                                   [2][0]=9, [2][1]=3,
                                   [2][2]=1, [2][3]=1 };
```

grid の、省略された 4 つ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
grid[0] [0]	8	grid[1] [2]	0
grid[0] [1]	6	grid[1] [3]	0
grid[0] [2]	4	grid[2] [0]	9
grid[0] [3]	1	grid[2] [1]	3
grid[1] [0]	0	grid[2] [2]	1
grid[1] [1]	0	grid[2] [3]	1

次の例のように、指定された初期化指定子は通常の初期化指定子と結合することができます。

```
int a[10] = {2, 4, [8]=9, 10}
```

この例では、a[0] は 2 に初期化され、a[1] は 4 に初期化されます。a[2] から a[7] は 0 に初期化され、 a[9] は 10 に初期化されます。

この例では、単一の指定子を使用して、配列の両端からスペースを「割り振って」います。

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

指定された初期化指定子 [MAX-5] = 8 は、添え字 MAX-5 の配列エレメントを値 8 に初期化することを意味します。MAX が 15 の場合、a[5] から a[9] は、ゼロに初期化されます。MAX が 7 の場合、a[2] から a[4] は、最初は、それぞれ、値 5、7、および 9 になり、それらは、値 8、6、および 4 でオーバーライドされます。つまり、MAX が 7 の場合、初期化は、宣言が次のように書かれるのと同じです。

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

指定子を使用して、ネストされた構造体のメンバーを表すこともできます。次に例を示します。

```
struct a {
         struct b {
              int c;
              int d;
          } e;
         float f;
} g = {.e.c = 3 };
```

この例は、構造体変数 g のメンバーである構造体変数 e のメンバー c を値 3 に初期化します。

関連資料:

133 ページの『構造体および共用体の初期化』

137 ページの『配列の初期化』

ベクトルの初期化 (IBM 拡張)

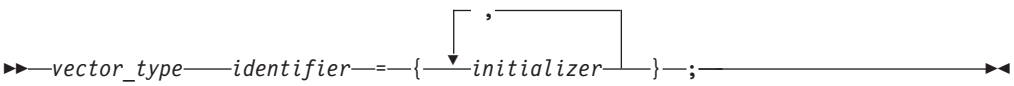
ベクトル型は、ベクトル・リテラル、または同じベクトル型を持つ任意の式により初期化されます。次に例を示します。

```
vector unsigned int v1;
vector unsigned int v2 = (vector unsigned int)(10);
v1 = v2;
```

C

AltiVec 仕様は、初期化指定子リストによってベクトル型を初期化できるようにします。 この機能は、GNU C との互換性のための拡張機能です。

ベクトル初期化指定子リストの構文



中括弧に入れた初期化指定子リスト内の値の数は、該当のベクトル型のエレメントの数以下でなければなりません。未初期化エレメントはゼロに初期化されます。

以下のコードは、初期化指定子リストを使用したベクトル初期化の例です。

```
vector unsigned int v1 = {1};  // initialize the first 4 bytes of v1 with 1
                               //  and the remaining 12 bytes with zeros

vector unsigned int v2 = {1,2};  // initialize the first 4 bytes of v2 with 1
                                 // and the next 4 bytes with 2

vector unsigned int v3 = {1,2,3,4};  // equivalent to the vector literal
                                     // (vector unsigned int) (1,2,3,4)
```

初期化指定子リスト内の値は、ベクトル・リテラルとは異なり、初期化されるベクトル変数が静的期間を持っているのでなければ、定数式である必要はありません。したがって、以下は正しいコードです。

```
int i=1;
int function() { return 2; }
int main()
{
   vector unsigned int v1 = {i, function()};
   return 0;
}
```

C

構造体および共用体の初期化

構造体の初期化指定子は、中括弧で囲まれた、コンマで区切られた値のリストです。共用体の場合、中括弧に入れられた単一の値です。初期化指定子の前には等号 (=) を付けます。

C99 および C++ では、共用体型または構造体型の自動メンバー変数の初期化指定子は定数式または非定数式にすることができます。

C++ 共用体型または構造体型の静的メンバー変数の初期化指定子は定数式またはストリング・リテラルにすることができます。詳しくは、 366 ページの『静的データ・メンバー』を参照してください。

構造体および共用体の初期化指定子は、次の 2 とおりの方法で指定できます。

- C89 スタイルの初期化指定子については、構造体メンバーは、宣言された順に初期化されなければならず、共用体の場合、最初のメンバーのみ初期化できます。
- C *指定された* 初期化指定子、つまり初期化されるメンバーに*名前* を付けることを可能にする C99 の機能を使用すると、構造体メンバーを任意の順序で初期化することができ、かつ共用体の任意の (単一) メンバーを初期化することができます。指定された初期化指定子については、 130 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』 で詳しく説明しています。

次の例では、C89 スタイルの初期化を使用して、共用体変数 `people` の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ています。

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = {"23/07/57"};
```

C 指定された初期化指定子を同じ例に使用すると、以下は 2 番目の共用体メンバー `age` を初期化します。

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = { .age = 14 };
```

次の定義では、完全に初期化される構造体を示します。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                 };
static struct address perm_address =
                { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

`perm_address` の値は、次のとおりです。

メンバー	値
perm_address.street_no	3
perm_address.street_name	ストリング "Savona Dr." のアドレス
perm_address.city	ストリング "Dundas" のアドレス
perm_address.prov	ストリング "Ontario" のアドレス
perm_address.postal_code	ストリング "L4B 2A1" のアドレス

名前なし構造体メンバーまたは共用体メンバーは初期化には関係せず、初期化後に中間値を持ちます。従って、次の例では、ビット・フィールドは初期化されず、初期化指定子 3 は、メンバー b に適用されます。

```
struct {
     int a;
     int :10;
     int b;
     } w = { 2, 3 };
```

構造体または共用体の全メンバーを初期化する必要はありません。初期化されていない構造体メンバーの初期値は、その構造体変数または共用体変数に関連付けられたストレージ・クラスによって異なります。静的と宣言された構造体では、初期化されないメンバーはすべて、該当の型のゼロに暗黙的に初期化されます。自動ストレージが指定された構造体のメンバーは、デフォルトの初期化は行われません。静的ストレージが指定された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す。自動ストレージが指定された共用体は、デフォルトの初期化は行われません。

次の定義では、一部のみが初期化される構造体を示します。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                };
struct address temp_address =
                { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

temp_address の値は、次のとおりです。

メンバー	値
temp_address.street_no	44
temp_address.street_name	ストリング "Knyvet Ave." のアドレス
temp_address.city	ストリング "Hamilton" のアドレス
temp_address.prov	ストリング "Ontario" のアドレス
temp_address.postal_code	temp_address 変数のストレージ・クラスによって異なります。それが静的の場合、値は NULL になります。

C temp_address 変数の 3 番目と 4 番目のメンバーのみを初期化するには、指定された初期化指定子リストを次のように使用します。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                };
struct address temp_address =
               { .city = "Hamilton", .prov = "Ontario" };
```

関連資料:

構造体および共用体の変数宣言

416 ページの『コンストラクターでの明示的初期化』

189 ページの『割り当て演算子』

列挙型の初期化

列挙変数の初期化指定子には、= シンボルと、その後に続く式 *enumeration_constant* が含まれます。

C++ C++ では、初期化指定子は、関連した列挙型と同じ型を持つ必要があります。 C++

以下のステートメントは、C++11 スコープのない C++11 列挙型 grain を宣言します。

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

以下のステートメントは変数 g_food を定義し、g_food を barley の値に初期化します。 barley に関連付けられた整数値は 2 です。

```
enum grain g_food = barley;
```

C++11

以下の規則は、スコープ付き列挙型とスコープのない列挙型の両方に適用されます。

- 明示的キャストを使用しない場合、異なる列挙型から整数または列挙型定数を使用して列挙型を初期化することはできません。
- 初期化されていない列挙型変数の値は未定義です。

以下のステートメントは、スコープのない列挙型 color を宣言します。

```
enum color { white, yellow, green, red, brown };
```

以下のステートメントは、スコープ付きの列挙型 letter を宣言し、列挙のスコープの内側でスコープ付き列挙子を直接参照しています。A、B、C、および D の初期値はそれぞれ 0、1、1、および 2 です。

```
enum class letter { A, B, C = B, D = C + 1 };
```

以下のステートメントは変数 let1 を定義し、let1 を A の値に初期化します。 A に関連付けられた整数値は 0 です。

```
letter let1 = letter :: A;
```

スコープのある列挙子をその列挙型のスコープの外側で参照するには、列挙型の名前を使用して列挙子を修飾する必要があります。例えば、以下のステートメントは無効です。

```
letter let2 = A;     //invalid
```

以下のステートメントにおけるキーワード enum は任意指定であり、省略できます。

```
enum letter let3 = letter :: B;
```

color はスコープのない列挙型であるため、white 列挙子は以下のステートメントで可視です。

```
color color1 = white;     // valid
```

スコープのない列挙型は、その列挙スコープで修飾することもできます。例えば、

```
color color2 = color :: yellow;    // valid
```

異なる列挙型の列挙型定数や、明示的なキャストを行っていない整数で列挙型を初期化することはできません。例えば、以下の 2 つのステートメントは無効です。

```
letter let4 = color :: white;   // invalid

letter let5 = 1;                // invalid
```

明示的なキャストを使用すれば、異なる列挙型の列挙型定数または整数で列挙型を初期化することができます。例えば、以下の 2 つのステートメントは有効です。

```
letter let6 = (letter) color :: white;     // valid

letter let7 = (letter) 2;                  // valid
```

C++11

関連資料:

列挙型変数の宣言

ポインターの初期化

初期化指定子は、= (等号) と、その後に続く、ポインターに入れられるアドレスを表す式によって表されます。次の例では、変数 time と speed には double 型、amount には double を指す型ポインターが指定されるように定義します。この例では、ポインター amount が total を指すように初期化が行われています。

```
 double time, 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、以下と等価です。

```
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、ストリング定数の先頭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、C++11 または nullptr キーワード C++11 を使用して、NULL に初期化することもできます。そのようなポインターは、NULL ポインターであり、いずれのオブジェクトをも指していません。 NULL ポインターについて詳しくは、『 120 ページの『NULL ポインター』』を参照してください。

次の例は、NULL ポインター値を使用して、ポインターを定義しています。

```
char *a = 0;
char *b = NULL;
```

C++11

```
char *ch = nullptr;
```

C++11

関連資料:

116 ページの『ポインター』

配列の初期化

配列の初期化指定子は、中括弧 ({ }) で囲まれた定数式の、コンマで区切られたリストです。初期化指定子の前には等号 (=) を付けます。配列内のすべてのエレメントを初期化する必要はありません。配列が部分的に初期化されている場合は、初期化されていないエレメントの値は、該当の型の値 0 となります。静的ストレージ期間を持つ配列のエレメントについても、同じことが言えます。 (静的ストレージ期間を持つのは、static キーワードを指定して宣言されているすべてのファイル・スコープ変数および関数スコープ変数です。)

配列の初期化指定子は、次の 2 とおりの方法で指定できます。

- C89 スタイルの初期化指定子の場合、配列エレメントは添え字順に初期化しなければなりません。
- C *指定された* 初期化指定子を使用すると、初期化する添え字エレメントの値を指定することができ、配列エレメントを任意の順序で初期化することができます。指定された初期化指定子については、130 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』 で詳しく説明しています。

次の定義では、C89 スタイルの初期化指定子を使用して、完全に初期化される 1 次元の配列を示しています。

```
static int number[3] = { 5, 7, 2 };
```

配列 number には、次の値が含まれます。すなわち、number[0] は 5、number[1] は 7、number[2] は 2 です。エレメントの数 (この例では、3) を定義する添え字宣言子の中で式を使用する場合、配列のエレメントの数より多い初期化指定子を使用することはできません。

次の定義は、一部のみが初期化される 1 次元配列を示しています。

```
static int number1[3] = { 5, 7 };
```

number1[0] および number1[1] の値は前の定義のときと同じですが、number1[2] は 0 です。

C

次の定義では、明示的に初期化しない配列のエレメントをスキップオーバーするための、指定された初期化指定子の使用法を示しています。

```
static int number[3] = { [0] = 5, [2] = 7 };
```

配列 number には、次のような値が入れられます。すなわち、 number[0] は 5、number[1] は暗黙的に 0 に初期化され、number[2] は 7 です。

C

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おらず、5 つの初期化指定子があるので、コンパイラーは item に、初期化された 5 つのエレメントを与えます。

文字配列の初期化

次のものを指定して、1 次元の文字配列を初期化できます。

- 中括弧で囲まれ、コンマで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。
- ストリング定数 (定数を囲む中括弧はオプション)

ストリング定数を初期化すると、空いている場所がある場合または配列の次元が指定されていない場合は、 NULL 文字 (¥0) をストリングの終わりに入れます。

以下の定義は、文字配列の初期化を示します。

```
static char name1[ ] = { 'J', 'a', 'n' };
static char name2[ ] = { "Jan" };
static char name3[4] = "Jan";
```

以下の定義では、次のエレメントを作成します。

エレメント	値	エレメント	値	エレメント	値
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	¥0	name3[3]	¥0

次の定義では、NULL 文字はなくなります。

```
static char name3[3]="Jan";
```

C++ 文字の配列をストリングで初期化する場合、ストリング内の文字数 (終端の '¥0' を含む) は、配列の中のエレメント数を超えてはなりません。

多次元配列の初期化

次の方法のいずれかにより、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する。各エレメントかエレメントのネスト・レベルを中括弧で囲むことができます。次の定義には、第 1 次元の 2 つのエレメントが含まれます (これらのエレメントは、行と見なすことができます)。初期化には、これらの 2 つの各エレメントを囲む中括弧が含まれます。

  ```
  static int month_days[2][12] =
  {
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
   { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
  };
  ```

- ネストされた中括弧を使用して、次元および次元のエレメントを選択的に初期化する。次の例では、配列 grid の最初の 8 つのエレメントのみが、明示的に初期化されます。明示的に初期化されない残りの 4 つのエレメントは、自動的にゼロに初期化されます。

  ```
  static short grid[3] [4] = {8, 6, 4, 1, 9, 3, 1, 1};
  ```

  grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	8	grid[1] [2]	1
grid[0] [1]	6	grid[1] [3]	1
grid[0] [2]	4	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	9	grid[2] [2]	0

エレメント	値	エレメント	値
grid[1] [1]	3	grid[2] [3]	0

- C *指定された* 初期化指定子の使用。以下の例は、指定された初期化指定子を使用して、配列の最後の 4 つのエレメントだけを明示的に初期化します。明示的に初期化されない最初の 8 個のエレメントは、自動的にゼロに初期化されます。

```
static short grid[3] [4] = { [2][0] = 8, [2][1] = 6,
                             [2][2] = 4, [2][3] = 1 };
```

grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	0	grid[1] [2]	0
grid[0] [1]	0	grid[1] [3]	0
grid[0] [2]	0	grid[2] [0]	8
grid[0] [3]	0	grid[2] [1]	6
grid[1] [0]	0	grid[2] [2]	4
grid[1] [1]	0	grid[2] [3]	1

関連資料:

123 ページの『配列』

130 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』

参照の初期化 (C++ のみ)

参照を初期化するときには、その参照をオブジェクトにバインドします。このときのオブジェクトは、必ずしも初期化指定子の式によって指定されるオブジェクトではありません。

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、参照への割り当て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に参照をバインドすることによって、実際の参照で動作します。割り当ては、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

参照バインディング

T および U が 2 つの型であるとします。トップレベルの cv 修飾子が無視された結果、T が U と同じ型になるか U の基底クラスになる場合は、T と U に*参照関連*があります。

例 1

```
typedef int t1;
typedef const int t2;
```

この例では、t1 と t2 に参照関連があります。

T および U に参照関連があり、T が少なくとも U と同等の cv 修飾を持つ場合、T には U との *参照の互換性* があります。例 1 において、t1 には t2 との参照の互換性がありませんが、t2 には t1 との参照の互換性があります。

型 T に対する左辺値参照 r が型 U の式 e によって初期化され、T に U との参照の互換性がある場合は、参照 r を e または e の基底クラス・サブオブジェクトに直接バインドできます (ただし、T がアクセス可能であり、U のあいまいな基底クラスでもないとします)。

例 2

```
int a = 1;
const int& ra = a;

struct A {};
struct B: A {} b;

A& rb = b;
```

この例において、const int 型は int 型との参照の互換性があるため、ra は直接 a にバインドできます。構造体 A には構造体 B との参照関連があるため、rb は直接 b にバインドできます。

型 T に対する左辺値参照 r を型 U の式 e で初期化する場合、以下の条件が満たされれば、r を e または e の基底クラスの変換の左辺値結果にバインドできます。この場合、変換関数は多重定義の解決によって選択されます。

- U はクラス型です。
- T に U との参照関連がない。
- e を型 S の左辺値に変換でき、かつ T に S との参照の互換性がある。

例 3

```
struct A {
   operator int&();
};

const int& x= A();
```

この例において、構造体 A はクラス型であり、const int 型には構造体 A との参照関連がありません。しかし、A は型 int の左辺値に変換でき、const int には int との参照の互換性があります。したがって、型 const int の参照 x は A() の変換結果にバインドできます。

デフォルトでは、コンパイラーは非定数または揮発性の左辺値参照を右辺値にバインドできません。

例 4

```
int& a = 2; // error
const int& b = 1; // ok
```

この例において、変数 a は const でない左辺値参照です。コンパイラーは、右辺値の式 2 で初期化された一時変数に a をバインドすることができず、エラー・メッセージを出します。変数 b は volatile でない const の左辺値参照であり、右辺値の式 1 で初期化される一時変数で初期化できます。

IBM

-qlanglvl=compatrvaluebinding オプションを指定すると、コンパイラーは非定数または揮発性の左辺値参照を初期化指定子が必須ではないユーザー定義型の右辺値にバインドできます。このオプションのデフォルト値は、**-qlanglvl=nocompatrvaluebinding** です。このコンパイラーの動作は、非定数参照または揮発性の左辺値参照の右辺値へのバインドを許可していない、右辺値参照機能と競合します。両方の機能を有効にした場合、コンパイラーはエラー・メッセージを発行します。右辺値参照機能について詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『右辺値参照の使用(C++11)』を参照してください。

注:

- 非定数または揮発性の左辺値参照を組み込み型の右辺値にバインドすることはできません。
- クラス・メンバーである非定数または揮発性の左辺値参照を右辺値にバインドすることはできません。

IBM

C++11

型 U の式 e が以下のいずれかの値カテゴリーに属しているとします。

- xvalue
- クラス prvalue
- 配列 prvalue
- 関数左辺値

型 T に対する右辺値参照または volatile でない const の左辺値参照 r を式 e で初期化する場合に、T に U との参照の互換性がある場合は、参照 r を式 e で初期化

できるほか、直接 e または e の基底クラス・サブオブジェクトにバインドすることができます (ただし、T がアクセス可能であり、U のあいまいな基底クラスでもないとします)。

例 5

```
int& func1();
int& (&&rf1)()=func1;

int&& func2();
int&& rf2 = func2();

struct A{
   int arr[5];
};
int(&&ar_ref)[5] = A().arr;
A&& a_ref = A();
```

この例において、rf1、rf2、ar_ref、および a_ref はいずれも右辺値参照です。rf1 は関数左辺値 func1 にバインドされ、rf2 は呼び出し func2() の xvalue 結果にバインドされ、ar_ref は配列 prvalue A().arr にバインドされ、a_ref はクラス prvalue A() にバインドされています。

r が、型 T に対する右辺値参照または volatile でない const の左辺値参照であり、その r を型 U の式 e で初期化するとします。以下の条件が満たされる場合は、r を e または e の基底クラスの変換結果にバインドできます。この場合、変換関数は多重定義の解決によって選択されます。

- U はクラス型です。
- T に U との参照関連がない。
- e を S の xvalue、クラス prvalue、または関数左辺値型に変換でき、かつ T に S との参照の互換性がある。

例 6

```
int i;
struct A {
   operator int&&() {
     return static_cast<int&&>(i);
   }
 };

const int& x = A();

int main() {
   assert(&x == &i);
}
```

この例において、構造体 A はクラス型であり、const int 型には構造体 A との参照関連がありません。しかし、A を型 int の xvalue に変換することができ、const int には int との参照の互換性があるため、const int の参照 x を A() で初期化し、変数 i にバインドすることができます。

以下のコンテキストでは、右辺値参照を左辺値で初期化できます。

- 関数左辺値
- 左辺値から変換された一時変数
- クラス型である左辺値オブジェクトを対象とする変換関数の右辺値結果

例 7

```
int i = 1;
int&& a = 2; // ok
int&& b = i; // error
double&& c = i; // ok
```

この例において、右辺値参照 a は右辺値式 2 で初期化される一時変数にバインドできますが、右辺値参照 b は左辺値式 i にバインドできません。右辺値参照 c は、変数 i から変換された一時値 1.0 にバインドできます。

C++11

関連資料:

126 ページの『参照 (C++ のみ)』

302 ページの『参照による受け渡し (C++ のみ)』

165 ページの『左辺値と右辺値』

複素数型の初期化 (C11)

C11 の複素数初期化機能が使用可能になっている場合は、C99 の複素数型を、x + yi の形式の値を持つ複素数型で初期化できます。ここで、x および y は任意の浮動小数点値にすることができます (Inf や NaN も可能です)。

C C11 複素数初期化機能は **-qlanglvl=extc1x** グループ・オプションで使用可能にすることができます。 C

C++ C11 複素数初期化機能は、**-qlanglvl=extended** または **-qlanglvl=extended0x** グループ・オプションで使用可能にすることができます。**-qlanglvl=complexinit** サブオプションを使用してこの機能を使用可能にすることもできます。**-qlanglvl=nocomplexinit** オプションを指定した場合は、C11 形式の複素数初期化のみが使用不可になります。 C++

C これらの複素数型の初期化を有効にするため、マクロ **CMPLX**、**CMPLXF**、および **CMPLXL** が C11 コンパイルの標準ヘッダー・ファイル complex.h 内に定義されています。これらは、以下の関数が使用されているのと同様に動作します。

```
float complex CMPLXF( float x, float y );
double complex CMPLX( double x, double y );
long double complex CMPLXL( long double x, long double y );
```

注: これらのマクロは、ユーザーの名前空間を侵害する可能性があります。これらのマクロ名を他の目的で使用しないようにしてください。

C

C++ 複素数初期化マクロは、以下のように __I_ImaginaryOnly キーワードで定義できます。

```
#define CMPLX(x, y) ((double)(x) + __I_ImaginaryOnly * (double)(y))
#define CMPLXF(x, y) ((float)(x) + __I_ImaginaryOnly * (float)(y))
#define CMPLXL(x, y) ((long double)(x) + __I_ImaginaryOnly * (long double)(y))
```

注: これらのマクロ定義は、ご使用のシステムの対応するマクロと互換性がない場合があります。

C++

これらのマクロは、C 言語ヘッダー・ファイル complex.h がインクルードされている場合にのみ使用でき、引数が静的初期化に適している場合は、結果として静的初期化に適した値が得られます。C++ C99 言語ヘッダー・ファイル complex.h を C++ プログラムで使用するには、**-qlanglvl=c99complex** オプションを指定し、#include ステートメントでの指定またはインクルード・パスの変更によってこの C99 ヘッダー・ファイルの絶対パスを指定する必要があります。そうしないと、コンパイラーによって警告メッセージが出されます。C++

関連資料:

547 ページの『C11 との互換性のための拡張機能』

26 ページの『浮動小数点リテラル』

65 ページの『浮動小数点型』

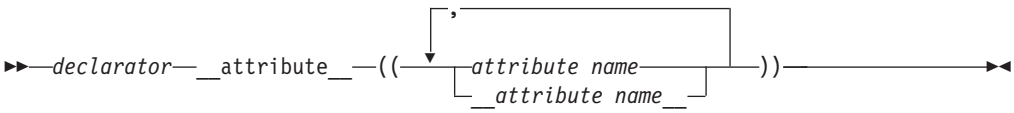
「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

変数属性 (IBM 拡張)

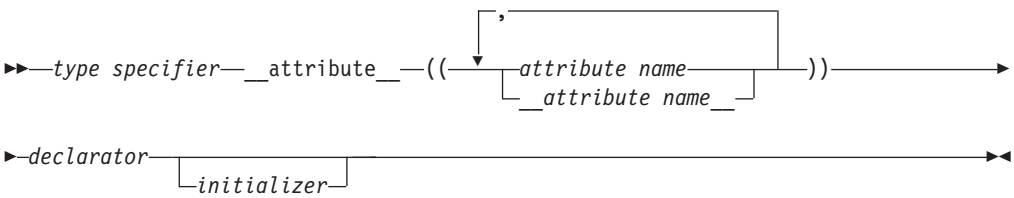
変数属性は、GNU C/C++ コンパイラーで開発されたプログラムのコンパイルを容易にするために提供された言語拡張機能です。この言語機能により、名前付き属性を使用して、データ・オブジェクトの特殊なプロパティーを指定することができます。変数 属性は、単純変数、集合体、および集合体のメンバー変数の宣言に適用されます。

変数属性は、キーワード __attribute__、その後に続く属性名、さらに、その属性名が必要とする追加引数によって指定されます。変数の __attribute__ 指定は、変数の宣言の中に組み込み、宣言子の前または後に置くことができます。変形はありますが、構文形式は一般的に次のどちらかです。

変数属性の構文: 宣言子の後



変数属性の構文: 宣言子の前



attribute name は、前後に 2 つの下線文字を付けても付けなくても指定できます。ただし、2 つの下線文字を使用すると、同じ名前のマクロと名前が競合する可能性が低くなります。サポートされない属性名については、XL C/C++ コンパイラーは診断メッセージを出して、その属性の指定を無視します。同じ属性指定に複数の属性名を指定することができます。

単一の宣言行の、コンマで区切った宣言子のリストにおいて、変数属性がすべての宣言子の前にある場合、その変数属性は、宣言内のすべての宣言子に適用されます。属性が宣言子の後にある場合、その属性は、直前の宣言子にのみ適用されます。次に例を示します。

```
struct A  {

 int b __attribute__((aligned));          /* typical placement of variable */
                                                      /* attribute */

 int __attribute__((aligned))__ c = 10;   /* variable attribute can also be */
                                            /* placed here */

 int d, e, f __attribute__((aligned));    /* attribute applies to f only */

 int g __attribute__((aligned)), h, i;    /* attribute applies to g only */

 int __attribute__((aligned)) j, k, l;    /* attribute applies to j, k, and l */

 };
```

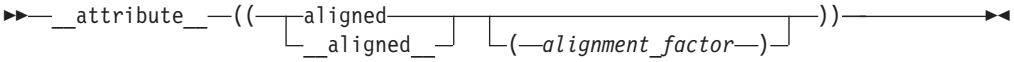
aligned 変数属性

aligned 変数属性を使用すると、以下の型の変数に対し、デフォルトのメモリー位置合わせモードをオーバーライドして、最小のメモリー位置合わせ値をバイト数で指定することができます。

- 非集合体変数
- 集合体変数 (構造体、クラス、共用体など)
- 選択されたメンバー変数

この属性は、一般的には、特定の変数の位置合わせを大きくするために使用します。

aligned 変数属性の構文



alignment_factor はバイト数ですが、定数式として指定し、その式が評価されて 2 の正の累乗になります。最大 1 GB までの値を指定できます。位置合わせ係数、およびそれを囲む小括弧を省略すると、コンパイラーは自動的に 16 バイトを使用します。最大値を超える位置合わせ係数を指定すると、コンパイラーは、有効になっているデフォルトの位置合わせを使用し、ユーザーの指定は無視します。

aligned 属性を ビット・フィールド構造体のメンバー変数に適用すると、属性指定は、ビット・フィールドのコンテナーに適用されます。コンテナーのデフォルトの位置合わせが位置合わせ係数より大きい場合、デフォルトの位置合わせが使用されます。

例

次の例では、構造体 first_address および second_address は 16 バイトの位置合わせに設定されています。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                } first_address __attribute__((__aligned__(16))) ;

struct address second_address __attribute__((__aligned__(16))) ;
```

次の例では、メンバー first_address.prov および first_address.postal_code のみが 16 バイトの位置合わせに設定されています。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov __attribute__((__aligned__(16))) ;
                  char *postal_code __attribute__((__aligned__(16))) ;
                } first_address  ;
```

関連資料:

102 ページの『__align 型修飾子 (IBM 拡張)』

「XL C/C++ 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

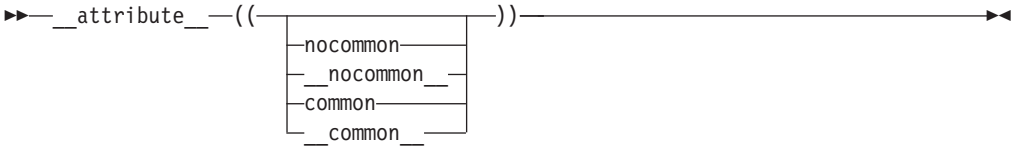
183 ページの『__alignof__ 演算子 (IBM 拡張)』

108 ページの『aligned 型属性』

common および nocommon 変数属性

変数属性 common を使用すると、未初期化グローバル変数または明示的に 0 に初期化されたグローバル変数を、オブジェクト・ファイルの common セクションに割り振るように指定することができます。変数属性 nocommon は、未初期化グローバル変数を、オブジェクト・ファイルの data セクションに割り振るように指定します。変数は、自動的にゼロに初期化されます。

nocommon および common 変数属性の構文



次に例を示します。

```
int i __attribute__((nocommon));      /* allocate i at .data  */
int k __attribute__((common));        /* allocate k at .comm  */
```

変数属性は、グローバル・スカラーまたは集合体変数にのみ適用することができます。属性を静的または自動的変数または構造あるいは共用体メンバーのいずれかに割り当てようと試みると、その属性は無視され、警告が出されます。

nocommon を使用して未初期化グローバル変数を data セクションで割り振ると、生成されたオブジェクトのサイズが劇的に増大することがあることに注意してください。また、同時にいくつかの異なるオブジェクト・ファイルで定義されているグローバル変数で nocommon を指定すると、リンク時にエラーが発生することになります。そのような変数は、1 つのファイルで定義し、他のファイルでは extern 宣言を使用して参照する必要があります。

属性は、-qcommon|nocommon コンパイラー・オプションより優先順位が高くなります。

同じ属性ステートメント内に属性の複数の指定が現れる場合には、最後に指定されたものが有効になります。次に例を示します。

```
int i __attribute__((common, nocommon));           /* allocate i at .data  */
int k __attribute__((common, nocommon, common));   /* allocate k at .comm  */
```

common または nocommon 属性と section 属性の両方が同じ変数に適用された場合には、section 属性が優先されます。

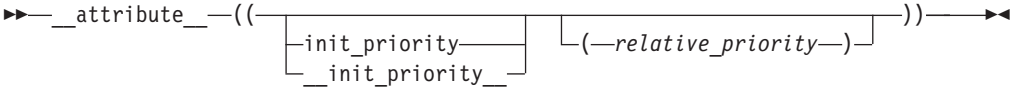
関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qcommon』を参照

init_priority 変数属性 (C++ のみ)

変数属性 init_priority は、複数のコンパイル単位にわたりネームスペース・スコープで定義された静的オブジェクトの初期化の順序を制御できる、C++ の拡張機能です。

init_priority 変数属性の構文

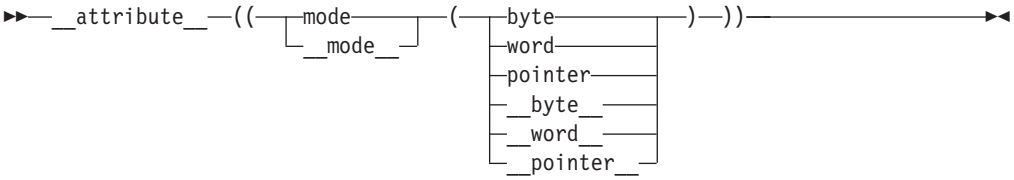


relative_priority は、包括的な 101 から 65535 までの定数整数式です。数字が低いほど、優先順位が高いことが示されます。

mode 変数属性

変数属性 mode は変数宣言内の型指定子をオーバーライドして、特定の整数型のサイズを指定することができます。

mode 変数属性の構文



モードの有効な引数は、特定の幅を示す以下の型指定子のいずれかです。

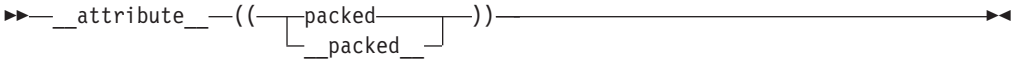
- byte は 1 バイトの整数型を意味します。
- word は 4 バイトの整数型を意味します。

- pointer は、32 ビット・モードでは 4 バイトの整数型を表し、 64 ビット・モードでは 8 バイトの整数型を表します。

packed 変数属性

変数属性 packed はデフォルトの位置合わせをオーバーライドして、集合体の全メンバーまたは集合体の選択されたメンバーの位置合わせを可能な最小の位置合わせに縮小することができます。すなわち、メンバーの場合は 1 バイトに、ビット・フィールド・メンバーの場合は 1 ビットに縮小します。

packed 変数属性の構文



関連資料:

102 ページの『__align 型修飾子 (IBM 拡張)』

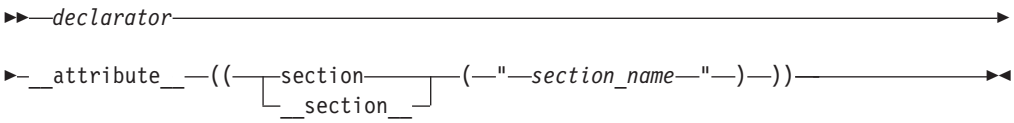

「XL C/C++ 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

183 ページの『__alignof__ 演算子 (IBM 拡張)』

section 変数属性

section 変数属性は、コンパイラーがその生成済みコードを配置する、オブジェクト・ファイル内のセクションを指定します。この言語機能を使用すると、変数が現れるセクションを制御できます。

section 変数属性の構文



section_name は、ストリング・リテラルとして、スペースをカウントせずに最大長 16 文字で名前付きセクションを指定します。ストリング内のスペースは無視されます。

section 変数属性は、以下の型の変数の宣言または定義に適用できます。

- 初期化済み変数または静的グローバル変数またはネームスペース変数
- 静的ローカル変数
- C++ 未初期化グローバルまたは名前空間変数
- C++ 静的な構造体またはクラス・メンバー変数

自動ストレージ期間を持つローカル変数はスタックに保管されているため、そのような変数に適用された section 属性は無視され、警告が出されます。

C 構造体メンバーに適用された section 属性は、無視され、警告が出されます。初期化されていないグローバル変数に適用される section 属性は無視され、警告は出されません。初期化されていないグローバル変数のシンボルは常に、共通セクションに配置されます。

複数の section 属性が変数宣言に適用された場合、最後の指定が用いられます。変数定義は上書きできないため、用いられる変数宣言で示されるセクションは、変数定義のセクションと一致している必要があります。定義済みの各変数は、1 つのセクションにのみ存在できます。

section 属性は、**-qcommon|nocommon** コンパイラー・オプションおよび common|nocommon 属性をオーバーライドします。すなわち、同じ変数に対して両方の属性が指定されると、section 属性の優先順位が高くなります。

1 つの名前付きセクションを複数の変数用に使用できますが、同じコンパイル単位内の変数と関数の両方に使用することはできません。

関連資料:

295 ページの『section 関数属性』

147 ページの『common および nocommon 変数属性』

「XL C/C++ コンパイラー・リファレンス」の中の『-qcommon』を参照

tls_model 属性

tls_model 属性を使用すると、特定の変数に対して使用されるスレッド・ローカル・ストレージ・モデルをソース・レベルで制御することができます。tls_model 属性には、local-exec、initial-exec、local-dynamic、または global-dynamic アクセス方式のいずれか 1 つを指定する必要があります。これは、その変数の **-qtls** オプションをオーバーライドします。次に例を示します。

```
__thread int i __attribute__((tls_model("local-exec")));
```

tls_model 属性を使用すると、リンカーが、アプリケーションまたは共用ライブラリーの作成に正しいスレッド・モデルが使用されたか否かを検査できるようになります。リンカー/ローダーの動作は次のとおりです。

表 25. スレッド・アクセス・モデルのリンク時/実行時の動作

アクセス方式	リンク時診断	実行時診断
local-exec	参照されたシンボルがインポートされていると失敗します。	モジュールがメインプログラムでない場合は失敗します。参照されたシンボルがインポートされていると失敗します (リンカーはすでにエラーを検出しているはずです)。
initial-exec	なし	参照されたシンボルが、実行時にロードされたモジュール内にない場合、dlopen() は失敗します。

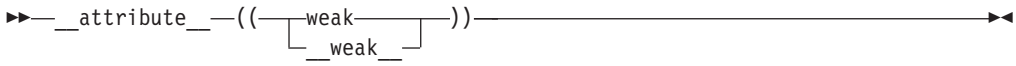
表 25. スレッド・アクセス・モデルのリンク時/実行時の動作 (続き)

アクセス方式	リンク時診断	実行時診断
local-dynamic	参照されたシンボルがインポートされていると失敗します。	参照されたシンボルがインポートされていると失敗します (リンカーはすでにエラーを検出しているはずです)。
global-dynamic	なし	なし

weak 変数属性

weak 変数属性があると、変数宣言の結果によるシンボルは、オブジェクト・ファイルの中にグローバル・シンボルとしてではなく、弱いシンボルとして現れます。この言語機能は、ライブラリー関数を書くプログラマーが、ユーザー・コード内の変数定義でライブラリー宣言をオーバーライドした場合に、重複した名前エラーを回避するための機能です。

weak 変数属性の構文



関連資料:

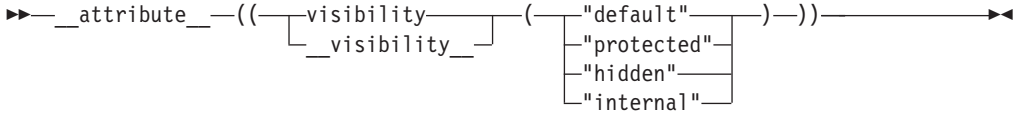
「XL C/C++ コンパイラー・リファレンス」の #pragma weak を参照

296 ページの『weak』

visibility 変数属性

visibility 変数属性は、あるモジュール内で定義されている変数が、他のモジュール内で参照または使用できるかどうか、およびその方法を記述します。 visibility 属性は、外部リンケージを持つ変数のみに影響を与えます。この機能を使用することで、共有ライブラリーを小さくして、シンボル競合の可能性を減らすことができます。詳しくは、「XL C/C++ 最適化およびプログラミング・ガイド」の『visibility 属性の使用』を参照してください。

visibility 変数属性の構文



例

以下の例で、変数 a の visibility 属性は保護され、変数 b の visibility 属性は隠されます。

```
struct str{
   int var;
};
int a __attribute__((visibility("protected")));
struct str __attribute__((visibility("hidden"))) b;
```

関連資料:

9 ページの『外部リンケージ』

visibility

112 ページの『visibility 型属性 (C++ のみ)』

323 ページの『visibility 名前空間属性 (IBM 拡張)』


「XL C/C++ 最適化およびプログラミング・ガイド」の中の『visibility 属性の使用』を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qvisibility』を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qmkshrobj』 を参照


「XL C/C++ コンパイラー・リファレンス」の『#pragma GCC visibility push、#pragma GCC visibility pop (IBM 拡張)』を参照

第 5 章 型変換

所定の型の式が、以下の状況で使用される場合は、*暗黙的に変換* されます。

- 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場合。
- `if` ステートメントまたは繰り返しステートメント (`for` ループなど) の中で条件として使用される場合。この式はブール (C89 では、整数) に変換されます。
- `switch` ステートメントで使用される場合。この式は整数型に変換されます。
- 割り当ての右オペランドである場合、または初期化指定子である場合。
- 初期設定として使用される場合。これには、以下の型が含まれます。
  - 関数に、パラメーターとは異なる型を持つ引数値が与えられる。
  - 関数の `return` ステートメントに指定された値が、その関数用に定義されている戻りの型と異なった型になっている。

C 暗黙の型変換の結果は右辺値です。 C

C++ 暗黙の型変換の結果は、変換後の式の型に応じて、以下のいずれかの値のカテゴリーに属します。

- 型が左辺値参照型である場合 C++11 または関数型に対する右辺値参照である場合 C++11 は左辺値
- C++11 型がオブジェクト型に対する右辺値参照型である場合は xvalue C++11
- それ以外の場合は C++11 (prvalue) C++11 右辺値

C++

206 ページの『キャスト式』で説明しているように、*cast* 式を使用して*明示的* 型変換を行うことができます。

ベクトル型キャスト (IBM 拡張)

ベクトル型は他のベクトル型にキャストすることができます。キャストでは変換は行われません。つまり、128 ビット・パターンはそのまま維持されますが、値は必ずしも維持されません。ベクトル型とスカラー型との間でキャストすることはできません。

ベクトル・ポインターと非ベクトル型を指すポインターの間では、相互にキャストすることができます。非ベクトル型を指すポインターをベクトル・ポインターにキャストするときは、アドレスは 16 バイトで位置合わせすることが必要です。非ベクトル型を指すポインターが参照するオブジェクトを 16 バイト境界で位置合わせするには、`__align` 指定子または `__attribute__((aligned(16)))` を使用できます。

関連資料:

427 ページの『ユーザー定義の変換』

428 ページの『変換コンストラクター』

430 ページの『変換関数』

238 ページの『switch ステートメント』

236 ページの『if ステートメント』

249 ページの『return ステートメント』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

算術変換およびプロモーション

以下のセクションでは、算術型に関する標準の変換の規則を説明します。

- 『整数型変換』
- 155 ページの『浮動小数点変換』
- 155 ページの『ブール型変換』

式の中に 2 つの異なる型のオペランドが含まれている場合、 156 ページの『通常の算術変換』で説明されているように、*通常の算術変換* の規則に従います。

整数型変換

符号なし整数から符号なし整数へ、または符号付き整数から符号付き整数へ

2 つの型が同一の場合、変更はありません。2 つの型のサイズが異なる場合、その値が新規の型で表現できるならば、値は変更されません。値が新規の型で表現できない場合、切り捨てが行われるか、または符号シフトが行われます。

符号付き整数から符号なし整数へ

結果の値は、元の整数に適合する、最小の符号なし整数型です。その値が新規の型で表現できない場合、切り捨てが行われるか、または符号シフトが行われます。

符号なし整数から符号付き整数へ

符号付き型が元の値を入れられる十分な大きさである場合、変更はありません。値が新規の型で表現できる場合、値は変更されません。値が新規の型で表現できない場合、切り捨てが行われるか、または符号シフトが行われます。

符号付きおよび符号なし文字型から整数へ

文字型は `int` 型にプロモートされます。

ワイド文字型 wchar_t から整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、それを収容できる最小の型、すなわち、`unsigned int`、`long`、または `unsigned long` にプロモートされます。

符号付きおよび符号なし整数ビット・フィールドから整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、`unsigned int` にプロモートされます。

列挙型から整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、それを収容できる最小の型、すなわち、`unsigned`

int、long、または unsigned long にプロモートされます。ただし、列挙型は整数型に変換できますが、整数型は列挙型に変換できないことに注意してください。

ブール型変換

スコープのない列挙型、ポインター、またはメンバー型へのポインターを、ブール型に変換できます。

C スカラー値が 0 に等しい場合、ブール値は 0 です。それ以外の場合、ブール値は 1 です。

C++ ゼロ、NULL ポインター、または NULL メンバー・ポインター値は、false に変換されます。他の値はすべて true に変換されます。

C++11 nullptr 値を持つ NULL ポインターは、false に変換されます。

浮動小数点変換

実浮動小数点型 間の変換の標準規則は、次のとおりです。

変換される値が新しい型で厳密に表現できる場合、その型は変更されません。変換される値が表現可能な値の範囲内にあるが、厳密に表現できない場合、コンパイル時またはランタイム時の、現在有効な丸めの方式に従がって結果が丸められます。変換する値が表現可能な値の範囲外の場合、結果は丸めの方式に左右されます。

整数から浮動小数点へ

変換される値が新しい型で厳密に表現できる場合、その型は変更されません。変換される値が表現可能な値の範囲内にあるが、厳密に表現できない場合、結果は正確に丸められます。変換する値が表現可能な値の範囲外の場合、結果は静止 NaN です。

浮動小数点から整数へ

小数部分は破棄されます (つまり、値はゼロに切り捨てられます)。整数部分の値が整数型で表現できない場合、結果は次のいずれかです。

- 整数型が符号なしで、かつ浮動小数点数が正の場合、結果は表示可能な最大数であり、それ以外の場合は 0 です。
- 整数型がサイン付きの場合、浮動小数点数の符号に応じて、結果は負の最小値または正の最大値です。

複素数の変換

複素数から複素数へ

2 つの型が同一の場合、変更はありません。2 つの型のサイズが異なり、かつその値が新規の型で表現できる場合は、値は変更されません。値が新規の型で表現できない場合、実数部と虚数部は上記の標準変換規則に従って変換されます。

複素数から実数 (2 進数) へ

複素数値の虚数部は破棄されます。必要に応じて、実数部の値は上記の標準変換規則に従って変換されます。

実数 (2 進数) から複素数へ
ソース値が複素数値の実数部として使用され、必要に応じて上記の標準変換規則に従って、変換されます。虚数部の値はゼロです。

関連資料:

65 ページの『浮動小数点型』

通常の算術変換

所定の型の式の中でオペランドとして異なる算術型が使用されている場合、*通常の算術変換* と呼ばれる標準の変換が適用されます。

例えば、2 つの異なる整数型の値を加算する場合、最初に両方の値が同じ型に変換されます。すなわち、`short int` 値と `int` 値を加算する場合、`short int` 値が `int` 型に変換されます。165 ページの『第 6 章 式と演算子』 に、通常の算術変換に関係のある演算子と式のリストを掲載しています。

算術型の変換ランク

以下の表内のランクは、最高位のものから順にリストしています。

表 26. 浮動小数点型の変換ランク

オペランドの型
`long double` または `long double _Complex`
`double` または `double _Complex`
`float` または `float _Complex`

表 27. 整数型の変換ランク

オペランドの型
`long long int`、`unsigned long long int`
`long int`、`unsigned long int`
`int`、`unsigned int`
`short int`、`unsigned short int`
`char`、`signed char`、`unsigned char`
bool
注: • `long long int` 型および `unsigned long long int` 型は、C89、C++98、および C++03 標準に含まれていません。 • C wchar_t 型は特殊タイプではなく、整数型の typedef です。wchar_t 型のランクは、その基礎となる型のランクと一致しています。 • C 列挙型のランクは、その基礎となる型のランクと一致しています。

その他の浮動小数点オペランドの規則

演算に 2 つのオペランドが含まれるコンテキストでは、オペランドのいずれかが浮動小数点型である場合、コンパイラーは、通常の算術変換を実行して、これらの 2 つのオペランドを共通の型にします。浮動小数点数のプロモーションが両方のオペ

ランドに適用されます。10 進浮動小数点オペランドの規則が適用されない場合、プロモートされたオペランドに以下の規則が適用されます。

1. 両方のオペランドが同じ型を持つ場合、変換は不要です。
2. 上記以外の場合に、両方のオペランドが複素数型である場合、下位の整数変換ランクの型が上位のランクの型に変換されます。詳しくは、 155 ページの『浮動小数点変換』を参照してください。
3. 上記以外の場合に、一方のオペランドが複素数型であれば、変換後の両方のオペランドの型は、以下の型のうち上位のランクのものになります。
   - 一般の浮動小数点オペランドの型に対応する複素数型
   - 複素数オペランドの型

   詳しくは、 155 ページの『浮動小数点変換』を参照してください。
4. 上記以外の場合、両方のオペランドは一般の浮動小数点型です。以下の規則が適用されます。
   a. 一方のオペランドが `long double` 型であれば、もう一方のオペランドは `long double` に変換されます。
   b. 上記以外の場合に、一方のオペランドが `double` 型であれば、もう一方のオペランドは `double` に変換されます。
   c. 上記以外の場合に、一方のオペランドが `float` 型であれば、もう一方のオペランドは `float` に変換されます。

整数オペランドの規則

演算に 2 つのオペランドが含まれるコンテキストでは、オペランドの両方が整数型である場合、コンパイラーは、通常の算術変換を実行して、これらの 2 つのオペランドを共通の型にします。整数拡張が両方のオペランドに適用され、プロモートされたオペランドに以下の規則が適用されます。

1. 両方のオペランドが同じ型を持つ場合、変換は不要です。
2. 上記以外の場合に、両オペランドが符号付き整数型を持つか、または両オペランドが符号なし整数型を持つ場合、下位の整数変換ランクの型が上位のランクの型に変換されます。
3. 上記以外の場合に、一方のオペランドが符号なし整数型を持ち、もう一方のオペランドが符号付き整数型を持つ場合、以下の規則が適用されます。
   a. 符号なし整数型のランクが、符号付き整数型のランクより上位または同じである場合、符号付き整数型が符号なし整数型に変換されます。
   b. 上記以外の場合に、符号付き整数型が符号なし整数型のすべての値を表現できる場合、符号なし整数型が符号付き整数型に変換されます。
   c. 上記以外の場合、両方の型は符号付き整数型に対応する符号なし整数型に変換されます。

関連資料:

63 ページの『整数型』
64 ページの『ブール型』
65 ページの『浮動小数点型』
66 ページの『文字型』
81 ページの『列挙型』

189 ページの『2 項式』

整数および浮動小数点数のプロモーション

整数および浮動小数点数のプロモーション は、通常の算術変換およびデフォルトの引数のプロモーションの一部として、自動的に使用されます。整数および浮動小数点数のプロモーションによって、値の大きさと符号は変更されません。通常の算術変換について詳しくは、 156 ページの『通常の算術変換』を参照してください。

C++11

wchar_t の整数拡張規則

値が wchar_t 型である場合、wchar_t の基礎となる型のすべての値を表現できる以下の最初の型に値の型を変換できます。

- `int`
- `unsigned int`
- `long int`
- `unsigned long int`
- `long long int`
- `unsigned long long int`

wchar_t の基礎となる型のすべての値を表現できる型がリストにない場合、wchar_t 型を wchar_t の基礎となる型に変換します。

C++11

ビット・フィールドの整数拡張規則

C この規則は、以下の条件に対して適用されます。

- **-qupconv** オプションが有効なとき。
- 整数ビット・フィールドの型が `unsigned` である。
- 整数ビット・フィールドの型が `int` 型よりも小さい。

これらの条件がすべて満たされる場合、整数ビット・フィールドのプロモーションには、以下のいずれかの規則が適用されます。

- `unsigned int` 型が整数ビット・フィールドのすべての値を表現できる場合、そのビット・フィールドを `unsigned int` に変換します。
- 上記以外の場合、ビット・フィールドには整数拡張は適用されません。

これらの条件のいずれかが満たされない場合、整数ビット・フィールドのプロモーションには、以下のいずれかの規則が適用されます。

- `int` 型が整数ビット・フィールドのすべての値を表現できる場合、そのビット・フィールドを `int` に変換します。
- 一方、`unsigned int` 型がすべての値を表現できる場合は、ビット・フィールドが `unsigned int` に変換されます。
- 上記以外の場合、ビット・フィールドには整数拡張は適用されません。

C

C++ 整数ビット・フィールドのプロモーションには、以下のいずれかの規則が適用されます。

- int 型が整数ビット・フィールドのすべての値を表現できる場合、そのビット・フィールドを int に変換します。
- 一方、unsigned int 型がすべての値を表現できる場合は、ビット・フィールドが unsigned int に変換されます。
- 上記以外の場合、ビット・フィールドには整数拡張は適用されません。

C++

ブールの整数拡張規則

C **-qupconv** オプションが有効な場合は、ブール値を unsigned int 型に変換し、値は変更せずそのままにします。一方、**-qnoupconv** オプションが有効な場合は、ブール値を int 型に変換し、値は変更せずそのままにします。 C

C++ ブール値が false の場合は、それを int に変換し、値は 0 にします。ブール値が true の場合は、int に変換し、値は 1 にします。

その他の型の整数拡張規則

C この規則は、以下の条件に対して適用されます。

- **-qupconv** オプションが有効なとき。
- ビット・フィールドとブール以外の整数型の型が unsigned である。
- 整数型の型が int 型よりも小さい。

これらの条件がすべて満たされる場合、整数型を unsigned int 型に変換します。

これらの条件のいずれかが満たされない場合、整数型のプロモーションには、以下のいずれかの規則が適用されます。

- 整数型を int型で表すことができ、そのランクが int のランクより低い場合、その整数型を int 型に変換します。
- それ以外の場合、整数型は unsigned int 型に変換されます。

C

C++ wchar_t、ビット・フィールド、およびブール以外の整数型のプロモーションには、以下のいずれかの規則が適用されます。

- 整数型を int型で表すことができ、そのランクが int のランクより低い場合、その整数型を int 型に変換します。
- それ以外の場合、整数型は unsigned int 型に変換されます。

C++

浮動小数点のプロモーション規則

float 型は double 型に変換できます。プロモーション後、float 値は変更されません。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qupconv (C のみ)』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qbitfields』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qchars』 を参照

左辺値から右辺値への変換

左辺値 C++11 または xvalue C++11 が、コンパイラーが C++11 (prvalue) C++11 右辺値を予期する状態で使用された場合、コンパイラーは左辺値 C++11 または xvalue C++11 を C++11 (prvalue) C++11 右辺値に変換します。ただし、C++11 (prvalue) C++11 右辺値は、暗黙的に左辺値 C++11 または xvalue C++11 には変換できません C++ (ユーザー定義変換を除く) C++ 。次の表に、この規則の例外をリストします。

変換前の状況	結果の動作
左辺値は関数型です。	関数を指すポインター
左辺値は配列です。	配列の最初のエレメントを指すポインター
左辺値 C++11 または xvalue C++11 は不完全型です。	コンパイル時エラー
左辺値 C++11 または xvalue C++11 は、未初期化オブジェクトを参照しています。	未定義の動作
左辺値 C++11 または xvalue C++11 は、C++11 (prvalue) C++11 右辺値の型でなく、C++11 (prvalue) C++11 右辺値の型から派生した型でもないオブジェクトを参照しています。	未定義の動作
C++ 左辺値 C++11 または xvalue C++11 は、`const` または `volatile` のいずれかによって修飾された非クラス型です。 C++	変換後の型は、`const` にも、`volatile` にも修飾されません。

関連資料:

165 ページの『左辺値と右辺値』

ポインター型変換

ポインター型変換は、ポインターが使用されるときに実行されます。この型変換には、ポインターの割り当て、初期化、および比較が含まれます。

C ポインターを含む変換は、明示的型キャストを使用しなければなりません。この規則の例外は、C ポインターに関する許容割り当て変換です。次の表の中で、`const` 修飾された左辺値は、割り当ての左方オペランドとして使用できません。

表 28. C ポインターに関する有効な割り当て変換

左方オペランドの型	許可される右方オペランドの型
(オブジェクト) T を指すポインター	• 定数 0 • T と互換型を指すポインター • void (void*) を指すポインター
(関数) F を指すポインター	• 定数 0 • F と互換性のある関数を指すポインター

左方オペランドの参照される型には、右方オペランドの参照される型と同等以上の cv 修飾子が必要です。

C

ゼロ定数から NULL ポインターへ

評価がゼロになる整数定数式は、*NULL 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、NULL ポインター (ゼロ値を持つポインター) となり、どのオブジェクトをも指さないようになります。

C++ 評価がゼロになる定数式は、メンバーを指す NULL ポインターに変換できます。

配列からポインターへ

型「*N* の配列」(*N* は、配列の単一エレメントの型) の左辺値または右辺値は *N* * へ。結果は、配列の初期エレメントを指すポインターです。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ている場合 C++ あるいは配列が配列型の参照にバインドされている場合 C++ 、この変換は実行されません。

関数からポインターへ

関数である左辺値は同じ型の関数へのポインターである C++11 (prvalue) C++11 右辺値に変換できます。ただし、式が & (アドレス) 演算子、() (関数呼び出し) 演算子、または sizeof 演算子のオペランドとして使用される場合を除きます。

関連資料:

116 ページの『ポインター』

169 ページの『整数定数式』

123 ページの『配列』

307 ページの『関数を指すポインター』

361 ページの『メンバーを指すポインター』

あいまいな基底クラス (C++ のみ)

165 ページの『左辺値と右辺値』

void* への変換

C ポインターは、必ずしも、型 int と同じサイズではありません。関数に渡されるポインター引数は、その関数によって期待される正しい型が必ず渡されるように、

明示的にキャストでなければなりません。 C で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ありません。

オブジェクトを指すポインター (型で修飾されることがあります) は、同じ const または volatile 修飾を保持しながら、void* に変換することができます。

C 以下の表に、左方オペランドとして void* を持つ許容割り当て変換を示します。

表 29. C での void に関する有効な割り当て変換*

左方オペランドの型	許可される右方オペランドの型
(void*)	• 定数 0 • オブジェクトを指すポインター。このオブジェクトは、不完全型でもかまいません。 • (void*)

C++ 関数を指すポインターは、標準の変換では型 void* に変換できません。void* に、それを保持できるだけの十分なビットがあれば、明示的に行うことができます。

関連資料:

67 ページの『void 型』

参照変換 (C++ のみ)

参照変換は、引数の受け渡しおよび関数からの戻り値で実行される参照初期化を含め、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。変換があいまいでなければ、クラスへの参照を、そのクラスのアクセス可能な基底クラスへの参照に変換することができます。変換の結果は、派生クラス・オブジェクトの基底クラス・サブオブジェクトへの参照になります。

参照変換は、対応するポインター型変換が許される場合に許されます。

関連資料:

126 ページの『参照 (C++ のみ)』

140 ページの『参照の初期化 (C++ のみ)』

299 ページの『関数呼び出し』

279 ページの『関数からの戻り値』

関数引数の変換

関数が呼び出されたとき、関数宣言が存在していて、宣言された引数の型が含まれている場合、コンパイラーは型検査を行います。コンパイラーは、呼び出し側の関数によって提供されるデータ型を、呼び出し先の関数が予想するデータ型と比較し、必要な型変換を行います。例えば、関数 funct が呼び出されると、引数 f は double に変換され、引数 c は int に変換されます。

```
char * funct (double d, int i);

int main(void){
```

```
   float f;
   char c;
   funct(f, c) /* f is converted to a double, c is converted to an int */
   return 0;
}
```

関数が呼び出されたときに可視の関数宣言がない場合、またはプロトタイプ引数リストの可変部分に式が引数として現れると、コンパイラーは、関数に引数を渡す前にデフォルトの引数プロモーションを行うか、または式の値を変換します。自動変換では、次のことが行われます。

- 整数および浮動小数点数のプロモーションが実行されます。
- 配列または関数はポインターに変換されます。

関連資料:

158 ページの『整数および浮動小数点数のプロモーション』

111 ページの『transparent_union 型属性 (C のみ) 』

175 ページの『関数呼び出し式』

299 ページの『関数呼び出し』

第 6 章 式と演算子

式は、計算を指定する演算子、オペランド、および区切り子のシーケンスです。式に含まれている演算子と、それらの演算子が使われるコンテキストに基づいて式の評価を行います。式は結果として値になりますが、*副次作用* が発生します。副次作用とは、実行環境の状態の変化のことです。

224 ページの『演算子優先順位と結合順序』に、先にリストされた各セクションで説明されるすべての演算子の優先順位をリストした表を掲載してあります。

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、これについては327 ページの『演算子の多重定義』で説明しています。

左辺値と右辺値

式は、以下のいずれかの値のカテゴリーに分類できます。

左辺値 式が `const` で修飾されていない場合は、式を割り当て式の左辺に記述できます。

Xvalue

有効期限付きの右辺値参照。

(Prvalue) 右辺値

割り当て式の右辺のみに指定可能な 非 xvalue 式。

右辺値には、xvalue と prvalue の両方が含まれます。左辺値および xvalue を glvalue と呼びます。

注:

- クラス (prvalue) 右辺値は cv 修飾できますが、非クラス (prvalue) 右辺値は cv 修飾できません。
- 左辺値 および xvalue は不完全型にすることができますが、(prvalue) 右辺値は完全型または `void` 型でなければなりません。

オブジェクトは、検査して、保管に使用できるストレージの領域です。左辺値 または xvalue は、そのようなオブジェクトを参照する式です。左辺値は、それが指定するオブジェクトの変更を必ずしも許可するわけではありません。例えば、`const` オブジェクトは、変更できない左辺値です。*変更可能な左辺値* という用語は、その左辺値は、指定されたオブジェクトを検査するだけでなく、変更できることを強調するために使用されます。以下のオブジェクト型の左辺値は、変更可能な左辺値ではありません。

- 配列型
- 不完全型
- `const` 修飾された型

- メンバーの 1 つが const 型として修飾されている構造体または共用体の型

これらの左辺値は変更できないため、割り当てステートメントの左辺に指定することはできません C++ (ただし、適切な割り当て演算子が存在する場合を除きます) C++ 。

C C では、*関数指定子* が関数型を持つ式として定義されます。関数指定子は、オブジェクト型または左辺値とは区別されます。関数指定子は関数の名前、または関数ポインターを間接参照した結果を使用できます。 C 言語は、関数ポインターの処理とオブジェクト・ポインターの処理を変えています。 C

C++ 左辺値参照を戻す関数呼び出しは左辺値です。E式は左辺値、 C++11 xvalue C++11 、 C++11 (prvalue) C++11 右辺値を生成するか、値を生成しないかのいずれかです。 C++

C++ 組み込み C++ 演算子の中には、それら演算子の一部のオペランドに左辺値を必要とするものがあります。 以下の表は、そのような演算子と、その使い方に対する追加制約を掲載しています。

演算子	要件
& (単項)	オペランドは、左辺値でなければなりません。
++ --	オペランドは、変更可能な左辺値にしてください。これは、接頭部と接尾部の両方の形式に適用されます。
= += -= *= %= <<= >>= &= ^= |=	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。

例えば、すべての割り当て演算子は、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割り当てます。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。

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。以下の例に、式と、その対応する左辺値を示します。

式	左辺値
x = 42	x
*ptr = newvalue	*ptr
a++	a
C++ f()	f() への関数呼び出し

C++11

以下の式は xvalue です。

- 戻りの型が右辺値参照型である関数を呼び出した結果
- 右辺値参照へのキャスト
- xvalue 式を通じてアクセスする非参照型の非静的データ・メンバー

- 第 1 オペランドが xvalue 式であり、第 2 オペランドがメンバー型を指すポインターであるメンバー・アクセス式を指すポインター

以下の例を参照してください。

```
int a;
int&& b= static_cast<int&&>(a);

struct str{
   int c;
};

int&& f(){
   int&& var =1;
   return var;
}

str&& g();
int&& rc = g().c;
```

この例において、右辺値参照 b の初期化指定子は、右辺値参照へのキャストの結果であるため、xvalue です。関数 f() を呼び出すと、この関数の戻りの型が int&& 型であるため、xvalue が生成されます。右辺値参照 rc の初期化指定子は、xvalue 式を通じて静的でない非参照データ・メンバー c にアクセスする式であるため、xvalue です。

C++11

C IBM GNU C 言語拡張機能を使用可能にしてコンパイルした場合は、複合式、条件式、およびキャストは、それぞれのオペランドが左辺値であれば、左辺値として使用できます。

シーケンス内の最後の式が左辺値である場合は、複合式を割り当てることができます。以下の式は等価です。

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

シーケンス内の最後の式が左辺値であれば、複合式にアドレス演算子を適用することができます。以下の式は等価です。

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

条件式は、その型が void でなく、真と偽に関する、その分岐が両方とも有効な左辺値である場合、有効な左辺値になりえます。キャストは、オペランドが左辺値である場合、有効な左辺値です。基本的な制限は、左辺値キャストのアドレスを使用できないことです。 IBM C

関連資料:

123 ページの『配列』

160 ページの『左辺値から右辺値への変換』

126 ページの『参照 (C++ のみ)』

1 次式

1 次式 は、次のカテゴリーに分類できます。

- 名前 (ID)
- リテラル (定数)
- 整数定数式
- ID 式
- 括弧で囲んだ式 ( )
- C11 汎用選択
- C++ this ポインター (362 ページの『this ポインター』で説明しています。)
- C++ スコープ解決演算子 (::) によって修飾された名前

名前

名前の値は、その型によって異なります。型は、その名前がどのように宣言されるかによって決まります。次の表に、名前が左辺値式であるかどうかを示します。

表 30. 1 次式: 名前

宣言された名前は	評価された結果は	左辺値か
算術型、ポインター型、列挙型、構造体型、または共用体型の変数	その型のオブジェクト	はい
列挙型定数	関連付けられた整数値	いいえ
配列	その配列。変換が必要なコンテキストでは、名前が sizeof 演算子の引数として使用される場合を除いて、配列内の最初のオブジェクトを指すポインター。	C いいえ C++ はい
関数	その関数。変換が必要なコンテキストでは、名前が sizeof 演算子の引数として使用される場合を除いて、または関数呼び出し式の中で関数として使用される場合を除いて、その関数を指すポインター。	C いいえ C++ はい

式としては、名前は、ラベル、typedef 名、構造体メンバー、共用体メンバー、構造体タグ、共用体タグ、または列挙型タグを参照できません。このような目的に使用される名前は、式で使用される名前の名前空間とは別の名前空間に置かれています。ただし、このような名前のいくつかは、特殊な構成によって式の中で参照することができます。例えば、ドット演算子または矢印演算子は、構造体または共用体のメンバーを参照するために使用できます。typedef 名は、キャストの中で、または sizeof 演算子への引数として使用できます。

リテラル

リテラルは、数値定数またはストリング・リテラルです。リテラルが式として評価されると、その値は定数です。字句定数は左辺値にはなりません。ただし、ストリング・リテラルは左辺値です。

関連資料:

20 ページの『リテラル』

362 ページの『this ポインター』

整数定数式

整数定数 は、コンパイル時に決定される値であり、実行時に変更することはできません。*整数定数式* は、定数で構成されていて、コンパイル時に定数に評価される式です。

整数定数式は、以下のエレメントだけで構成される式です。

- リテラル
- 列挙子
- コンパイル時定数式または C++11 constexpr 式 C++11 で初期化された const 変数
- コンパイル時定数式または C++11 constexpr 式 C++11 で初期化された static const データ・メンバー
- 整数型のキャスト
- sizeof 式。ただし、オペランドは可変長配列であってはならない。

可変長配列型に適用される sizeof 演算子は、実行時に評価され、そのため、定数式ではありません。

以下のような状況においては、整数定数式を使用する必要があります。

- 添え字宣言子の中 (バインドされた配列の記述として)。
- switch ステートメントのキーワード case の後。
- 列挙子の中で、列挙型定数の数値として。
- ビット・フィールド幅の指定子の中。
- プリプロセッサーの #if ステートメントの中。(プリプロセッサーの #if ステートメントに列挙型定数、アドレス定数、および sizeof は指定できません。)

注: C++11 C++11 標準は、定数式の概念を一般化します。詳しくは、 174 ページの『一般化された定数式 (C++11)』を参照してください。 C++11

関連資料:

184 ページの『sizeof 演算子』

ID 式 (C++ のみ)

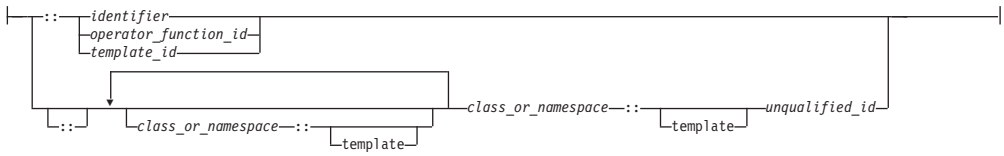
ID 式、すなわち、*id-expression* は、制限された形式の 1 次式です。構文的には、*id-expression* は、C++ の言語エレメントのすべてに名前を与えるための単純な ID より高水準の複雑さが必要です。

id-expression は、修飾された ID であっても、修飾されない ID であってもかまいません。また、ドット演算子および矢印演算子の後にあってもかまいません。

ID 式の構文

unqualified_id:

qualified_id:



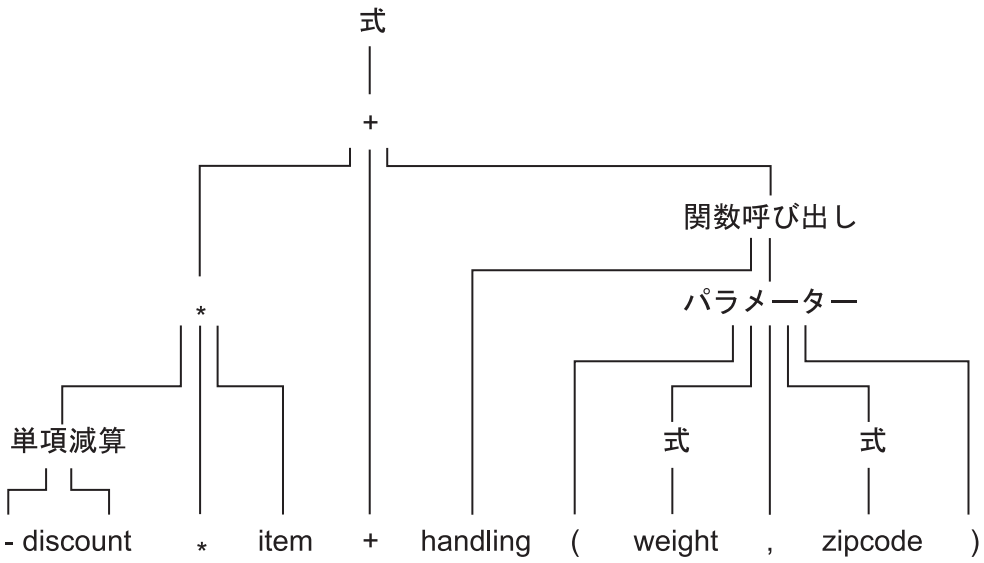
関連資料:

16 ページの『ID』

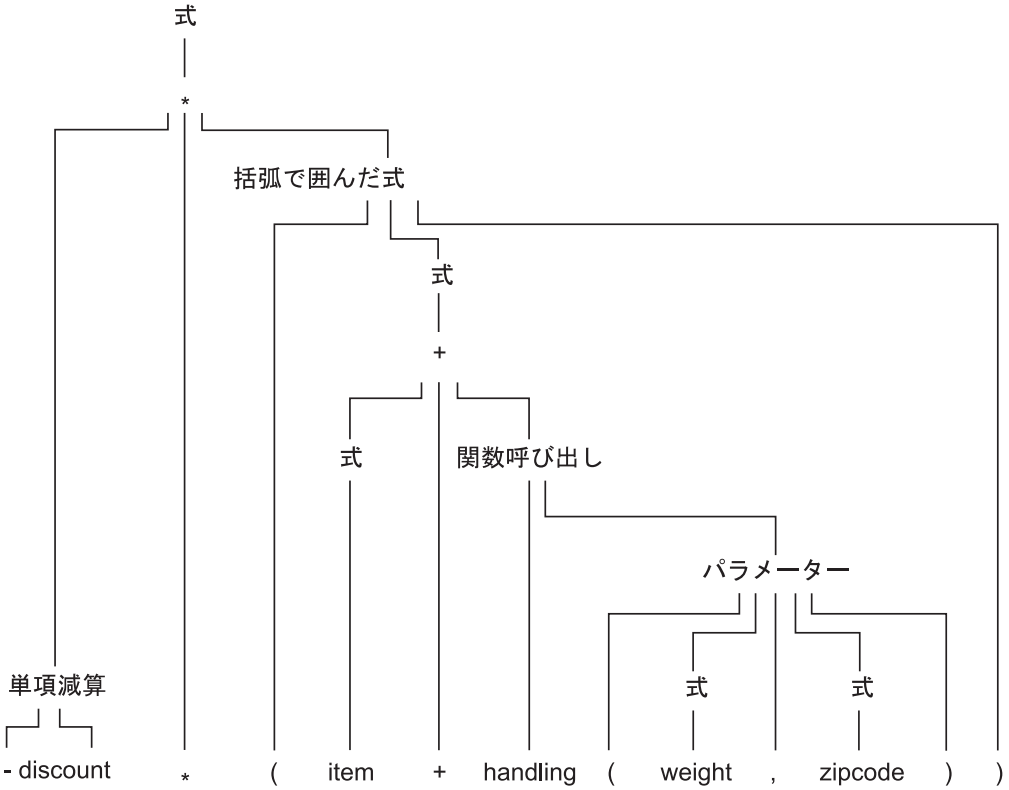
113 ページの『第 4 章 宣言子』

括弧で囲んだ式 ( )

小括弧を使用して、式の評価順序を明示的に強制します。次の式は、オペランドと演算子をグループ化するための小括弧は使用していません。 `weight , zipcode` を囲む括弧は、関数呼び出しを形成するのに使用されます。コンパイラーが式の中のオペランドと演算子を、演算子の優先順位や結合順序の規則に従って、グループ化する方法に注意してください。



次の式は、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);
```

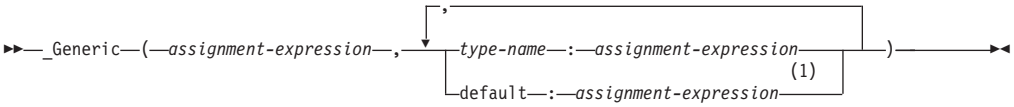
関連資料:

224 ページの『演算子優先順位と結合順序』

汎用選択 (C11)

汎用選択は 1 次式です。その型と値は、選択した汎用アソシエーションに依存しています。

以下の図は、汎用選択の構文を示しています。



注:

1 汎用選択は、最大で 1 つのデフォルトの汎用アソシエーションを持つことができます。

ここで、

type-name
汎用アソシエーションのタイプを指定します。汎用アソシエーションに指定する型名は、可変変更型以外の完全オブジェクト型でなければなりません。

assignment-expression
割り当て式です。最初の割り当て式は、制御式と呼ばれます。

汎用アソシエーション・リストは、汎用アソシエーションのグループです。汎用アソシエーションには、以下の 2 つの形式があります。

- `type-name`: assignment-expression
- `default`: assignment-expression

1 つの汎用選択は、互換タイプを指定する複数の汎用アソシエーションを持つことはできません。 1 つの汎用選択で、制御式は、汎用アソシエーション・リスト内に最大で 1 つの互換型名を持つことができます。汎用選択にデフォルトの汎用アソシエーションがない場合、その制御式は、その汎用アソシエーション・リスト内に厳密に 1 つの互換型名を持つ必要があります。

汎用選択内の制御式と互換性がある型名との汎用アソシエーションがある場合、汎用選択内の式は結果式になります。そうでない場合、汎用選択の結果式は、デフォルトの汎用アソシエーションの式になります。汎用選択の制御式は評価されません。汎用選択の他のどの汎用アソシエーションからの式も評価されません。

汎用選択の型および値は、その結果式の型および値と同じです。例えば、汎用選択の結果式が左辺値、関数指定子、またはボイド式であれば、汎用選択は左辺値、関数指定子、またはボイド式です。

例

以下の例では、`myprogram.c` は型総称マクロを定義します。

```
#define myfunction(X) _Generic((X), ¥
long double:myfunction_longdouble, ¥
default:myfunction_double, ¥
float:myfunction_float ¥
)(X)
void myfunction_longdouble(long double x){printf("calling %s¥n",__func__);}
void myfunction_double(double x){printf("calling %s¥n",__func__);}
void myfunction_float(float x){printf("calling %s¥n",__func__);}

int main()
{
  long double ld;
  double d;
  float f;
  myfunction(ld);
  myfunction(d);
  myfunction(f);
}
```

以下のプログラムを実行するとします。

```
xlc myprogram.c -qldbl128 -qlanglvl=extc1x
./a.out
```

結果は以下のようになります。

```
calling myfunction_longdouble
calling myfunction_double
calling myfunction_float
```

スコープ解決演算子 :: (C++ のみ)

:: (スコープ・レゾリューション) 演算子は、隠された名前を修飾して、それらの名前を引き続き使用できるようにするために使われます。ブロックまたはクラス内の同じ名前の明示宣言によって、名前空間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、グローバル名前空間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、グローバル名前空間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 とスコープ解決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
}
```

関連資料:

349 ページの『クラス名のスコープ』

313 ページの『第 9 章 名前空間 (C++ のみ)』

一般化された定数式 (C++11)

C++11 標準は、定数式の概念を一般化し、宣言指定子として新規キーワード constexpr を導入しています。定数式は、コンパイル時にコンパイラーによって評価できる式です。この機能の主要な利点は、以下のとおりです。

- コンパイル時の評価を必要とする、コードの型の安全性と移植性を向上させる。
- システム・プログラミング、ライブラリー作成、および汎用プログラミングのサポートを向上させる。
- 標準ライブラリー・コンポーネントのユーザビリティーを向上させる。コンパイル時に評価できるライブラリー関数は、定数式を必要とするコンテキストで使用できます。

constexpr 指定子を持つオブジェクト宣言は、オブジェクトを定数として宣言します。 constexpr 指定子は、以下のコンテキストに対してのみ適用できます。

- オブジェクトの定義
- 関数または関数テンプレートの宣言
- リテラル型の静的データ・メンバーの宣言

コンストラクターではない関数を constexpr 指定子を指定して宣言すると、その関数は constexpr 関数になります。同様に、コンストラクターを constexpr 指定子を指定して宣言すると、そのコンストラクターは constexpr コンストラクターになります。

この機能を使用して、定数式は、テンプレートおよび非テンプレートの constexpr 関数、クラス・リテラル型の constexpr オブジェクト、および定数式で初期化された const オブジェクトに対する参照境界を組み込むことができます。

コンパイル時の浮動小数点演算の評価は、**-qfloat** オプションのデフォルトのセマンティクスを使用します。

関連資料:

99 ページの『constexpr 指定子 (C++11)』

309 ページの『constexpr 関数 (C++11)』

414 ページの『constexpr コンストラクター (C++11)』

555 ページの『C++11 互換性の拡張機能』

関数呼び出し式

関数呼び出し は、関数名と、その後に続く関数呼び出し演算子 () とで構成される式です。関数がパラメーターを受け取るように定義されている場合、関数に送られる値は、関数呼び出し演算子の括弧内にリストされています。引数リストには、コンマで区切った式をいくつでも入れることができます。引数リストは、空にすることもできます。

関数呼び出し式の型は、その関数の戻りの型です。この型は、完全型、参照型、または `void` 型のいずれかです。

C 関数呼び出しは常に右辺値です。 C

C++ 関数呼び出しは、関数の結果の型に応じて、以下のいずれかの値のカテゴリーに属します。

- 結果の型が左辺値参照型である場合 C++11 または関数型に対する右辺値参照である場合 C++11 は左辺値
- C++11 結果の型がオブジェクト型に対する右辺値参照型である場合は xvalue C++11
- それ以外の場合は C++11 (prvalue) C++11 右辺値

C++

次は、関数呼び出し演算子の例です。

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

関数呼び出しの引数の評価順序は指定されていません。次の例では、

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

引数 `batch.process--` が最後に評価される可能性があり、その場合、最後の 2 つの引数は同じ値で渡されることになります。

関連資料:

162 ページの『関数引数の変換』

299 ページの『関数呼び出し』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

メンバー式

メンバー式は、クラス、構造体、または共用体のメンバーを示します。メンバー演算子は次のとおりです。

- ドット演算子 .
- 矢印演算子 ->

ドット演算子 .

. (ドット) 演算子は、クラス、構造体、または共用体メンバーにアクセスするために使用されます。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、場合によっては、修飾された ID または疑似デストラクター名が続きます。(*疑似デストラクター* は、非クラス型のデストラクターです。)後置式は、class、struct、または union 型のオブジェクトでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。後置式と名前が左辺値の場合は、その式の値も左辺値です。後置式が型修飾されている場合、結果の式の中の指定されたメンバーには同じ型修飾子が適用されます。

関連資料:

構造体メンバーおよび共用体メンバーへのアクセス

426 ページの『疑似デストラクター』

矢印演算子 ->

-> (矢印) 演算子は、ポインターを使用するクラス、構造体または共用体メンバーにアクセスするために使われます。後置式、その後に続く -> (矢印) 演算子、さらにその後に、できるだけ修飾された ID または疑似デストラクター名が続き、ポインターが指すオブジェクトのメンバーを指定します。(*疑似デストラクター* は、非クラス型のデストラクターです。) 後置式は、class、struct、または union 型のオブジェクトを指すポインターでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。名前が左辺値の場合は、式の値も左辺値です。式が修飾型を指すポインターである場合、同じ型修飾子が結果の式の中の指定されたメンバーに適用されます。

関連資料:

116 ページの『ポインター』

構造体メンバーおよび共用体メンバーへのアクセス

71 ページの『構造体および共用体』

355 ページの『第 12 章 クラスのメンバーとフレンド (C++ のみ)』

426 ページの『疑似デストラクター』

単項式

単項式 には、1 つのオペランドと単項演算子が含まれています。

サポートされる単項演算子は次のとおりです。

- 177 ページの『増分演算子 ++』
- 178 ページの『減分演算子 --』
- 178 ページの『単項正演算子 +』
- 178 ページの『単項減算演算子 -』
- 179 ページの『論理否定演算子 !』
- 179 ページの『ビット単位否定演算子 ~』

- 180 ページの『アドレス演算子 &』
- 181 ページの『間接演算子 *』
- C++ typeid
- IBM alignof
- sizeof
- IBM typeof
- IBM __real__ および __imag__
- IBM vec_step

すべての単項演算子には、 225 ページの表 35 に示されているように、同じ優先順位が付けられ、右から左の結合順序が適用されます。

演算子の説明で示されているように、ほとんどの単項式のオペランドで、通常の算術変換が行われます。

関連資料:

117 ページの『ポインター演算』

165 ページの『左辺値と右辺値』

154 ページの『算術変換およびプロモーション』

増分演算子 ++

++ (増分) 演算子は、スカラー・オペランドの値に 1 を加えます。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ増分します。オペランドは、増分演算の結果を受け取ります。オペランドは、算術型またはポインター型の変更可能な左辺値でなければなりません。

++ は、オペランドの前にも後にも置くことができます。それがオペランドの前にくると、オペランドは増分されます。増分された値が、式の中で使用されます。オペランドの後に ++ を置くと、オペランドを増分する*前*に、そのオペランドの値が使用されます。前置インクリメント式は、左辺値です。後置インクリメント式は、右辺値です。次に例を示します。

```
play = ++play1 + play2++;
```

これは、以下の式に似ています。play1 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、整数拡張後にオペランドと同じ型になります。

オペランドには、通常の算術変換が実行されます。

IBM 増分演算子は、複素数型を扱うように拡張されています。この演算子の働きは、オペランドの実数部のみが増分され、虚数部は変更されないことを除いて、実数型に対する場合と同じです。

減分演算子 --

-- (減分) 演算子は、スカラー・オペランドの値から 1 を減算します。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ、減じます。オペランドは、減分演算の結果を受け取ります。オペランドは、変更可能な左辺値でなければなりません。

減分演算子は、オペランドの前後に -- を入れることができます。この演算子がオペランドの前にあると、オペランドを減分し、減らした値が式で使われます。 -- がオペランドの後にある場合は、オペランドの現行値が式で使われ、その後でオペランドが減らされます。前置デクリメント式は、左辺値です。後置デクリメント式は、右辺値です。

次に例を示します。

```
play = --play1 + play2--;
```

これは、以下の式に似ています。play1 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、左辺値ではなく、整数拡張後にオペランドと同じ型になります。

オペランドには、通常の算術変換が実行されます。

IBM 減分演算子は、GNU C との互換性のために、複素数型を扱うように拡張されています。この演算子の働きは、オペランドの実数部のみが減分され、虚数部は変更されないことを除いて、実数型に対する場合と同じです。

単項正演算子 +

+ (単項正) 演算子は、オペランドの値を保持します。オペランドには、任意の算術型またはポインター型を指定できます。結果は、左辺値ではありません。

結果は、整数拡張後にオペランドと同じ型になります。

注: 定数の前の正符号は、定数の一部ではありません。

単項減算演算子 -

- (単項減算) 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、整数拡張後にオペランドと同じ型になります。

注: 定数の前の負符号 (-) は、定数の一部ではありません。

論理否定演算子 !

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

C オペランドの評価の結果が 0 になる場合は、式の値は 1 (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) になる場合は、式の値は false になります。オペランドは、暗黙的にブールに変換されます。そして、結果の型はブールです。

次の 2 つの式は、同じです。

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

関連資料:

64 ページの『ブール型』

ビット単位否定演算子 ~

~ (ビット単位否定) 演算子は、オペランドのビット単位の補数を生成します。結果の 2 進表現では、すべてのビットは、オペランドの 2 進表現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 進表現は次のとおりです。

```
0000000000000101
```

式 ~x の結果は、次のようになります (ここでは、16 ビットの 2 進数で表されます)。

```
1111111111111010
```

~ 文字は、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 進表現は、次のとおりです。

```
1111111111111111
```

IBM ビット単位否定演算子は、複素数型を扱うように拡張されています。複素数型では、この演算子は、虚数部の符号を反転することにより、オペランドの複素数共役を計算します。

関連資料:

43 ページの『3 文字表記シーケンス』

アドレス演算子 &

& (アドレス) 演算子は、そのオペランドを指すポインターを生成します。オペランドは、左辺値、関数指定機能、または修飾名でなければなりません。ビット・フィールドを指定することはできません。

C ストレージ・クラス register を持つことはできません。

オペランドが左辺値または関数である場合は、結果の型は式の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 に割り当てます。

```
p_to_y = &y;
```

IBM アドレス演算子は、ベクトルのサポートが使用可能になっている場合に、ベクトル型を処理するために拡張されました。ベクトル型にアドレス演算子を適用した結果は、互換性のあるベクトル型を指すポインターに保管することができます。ベクトル型のアドレスを使用して、ベクトル型を指すポインターを初期化することができます。ただし、初期化の両辺が互換型であることが必要です。 void を指すポインターも、ベクトル型のアドレスで初期化することができます。

C++ C++ におけるアンパーサンド記号 & は、アドレス演算子であるだけでなく、左辺値参照の宣言子を記述する目的にも使用されます。意味は関連していますが、同一ではありません。

```
int target;
int &rTarg = target;  // rTarg is an lvalue reference to an integer.
                      // The reference is initialized to refer to target.
void f(int*& p);      // p is an lvalue reference to a pointer
```

参照のアドレスを取ると、そのターゲットのアドレスが戻されます。上記の宣言を使用すると、&rTarg は、&target と同じメモリー・ターゲットです。

レジスター変数のアドレスを取ることができます。

多重定義関数の使われるバージョンを、左側が固有に判別する初期化または割り当てでのみ、多重定義関数で & 演算子を使用できます。 C++

IBM ラベルのアドレスは、GNU C アドレス演算子 && を使用して取ることができます。したがって、アドレスは値として使用できます。

関連資料:

181 ページの『間接演算子 *』

116 ページの『ポインター』

126 ページの『参照 (C++ のみ)』

232 ページの『値としてのラベル (IBM 拡張)』

間接演算子 *

* (間接) 演算子は、ポインター型オペランドによって参照される値を決めます。オペランドは、cv void ではない不完全型を指すポインターにすることができます。したがって、取得された左辺値は、C++11 prvalue C++11 右辺値に変換することはできません。オペランドがオブジェクトを指し示している場合は、演算子の結果、そのオブジェクトを参照する左辺値が導き出されます。オペランドが関数を指す場合、結果は C 関数指定子 C C++ オペランドが指すオブジェクトを参照する左辺値 C++ 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。

IBM

間接演算子 * は、ベクトルのサポートが使用可能になっている場合に、ベクトル型を指すポインターを処理するために拡張されました。ベクトル・ポインターは、16 バイトに位置合わせされたメモリー位置を指す必要があります。ただし、コンパイラーは、この制約を強制適用しません。ベクトル・ポインターから逆参照したときに、ベクトル型とその 16 バイト位置合わせが維持されます。プログラムが、16 バイトで位置合わせされたアドレスを含まないベクトル・ポインターから逆参照した場合は、どのような振る舞いが生じるかは未定義です。

IBM

関連資料:

123 ページの『配列』

116 ページの『ポインター』

typeid 演算子 (C++ のみ)

typeid 演算子は、プログラムが、ポインターまたは参照によって参照されるオブジェクトの実際の派生型を検索できるようにします。この演算子は、dynamic_cast 演算子とともに、C++ のランタイム型 ID (RTTI) のサポート用に提供されています。

typeid 演算子の構文

typeid 演算子は、ランタイム型情報 (RTTI) が生成される必要があります。その情報は、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しなければなりません。

typeid 演算子は、式 *expr* の型を表す型 const std::type_info の左辺値を戻します。 typeid 演算子を使用するためには、標準テンプレート・ライブラリー・ヘッダー <typeinfo> を組み込んでおく必要があります。

expr が、ポリモアフィック・クラスへの参照または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 オブジェクトを戻します。ポリモアフィック・クラスでない場合、typeid は、参照の型または間接参照ポインターを表す type_info オブジェクトを戻します。次の例は、このことを示しています。

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

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };

int main() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;

  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}
```

上記の例の出力は、以下のとおりです。

```
ap: B
ar: B
cp: C
cr: C
```

クラス A と B はポリモアフィックで、クラス C と D はそうではありません。 cp と cr は、型 D のオブジェクトを参照していますが、typeid(*cp) と typeid(cr) は、クラス C を表すオブジェクトを戻します。

左辺値から右辺値へ、配列からポインターへ、および関数からポインターへの変換は、*expr* へは適用されません。例えば、次の例の出力は int [10] であって、int * ではありません。

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

int main() {
  int myArray[10];
  cout << typeid(myArray).name() << endl;
}
```

expr がクラス型であれば、そのクラスは、完全に定義される必要があります。

typeid 演算子は、トップレベルの const または volatile 修飾子を無視します。

関連資料:

115 ページの『型名』

186 ページの『typeof 演算子 (IBM 拡張)』

__alignof__ 演算子 (IBM 拡張)

__alignof__ 演算子は、その演算子のオペランドの位置合わせ先の位置を戻す、C99 および標準 C++ の言語拡張機能です。オペランドは式であってもよいし、括弧で囲んだ型 ID であってもかまいません。オペランドが左辺値を表す式である場合、__alignof__ によって戻される数値は、その左辺値が持つことが判明している位置合わせを表します。式の型はコンパイル時に判別されますが、式そのものは評価されません。オペランドが型である場合、その数値は、ターゲット・プラットフォーム上でその型に対して通常必要な位置合わせを表します。

__alignof__ 演算子は、次の状況に適用することはできません。

- ビット・フィールドを表す左辺値
- 関数型
- 未定義の構造体またはクラス
- 不完全型 (void など)

__alignof__ 演算子の構文

type-id が参照、または参照された型である場合、結果は、参照された型の位置合わせです。 *type-id* が配列である場合、結果は、配列エレメント型の位置合わせです。*type-id* が基本型である場合、結果は、インプリメンテーションで定義されたとおりです。

例えば、Linux では、__alignof__(long) は、32 ビット・モードでは 4 を戻し、64 ビット・モードでは 8 を戻します。

ベクトルのサポートが使用可能になっている場合には、__alignof__ のオペランドをベクトル型にすることができます。次に例を示します。

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int *pv1 = &v1;
__alignof__(v1); // vector type alignment: 16.
__alignof__(&v1); // address of vector alignment: 4.
__alignof__(*pv1); // dereferenced pointer to vector alignment: 16.
__alignof__(pv1); // pointer to vector alignment: 4.
__alignof__(vector signed char); // vector type alignment: 16.
```

__attribute__((aligned)) を使用して、ベクトル型の変数の位置合わせ値を増加させた場合は、__alignof__ 演算子から戻される値は、__attribute__((aligned)) に指定されている位置合わせ計数です。

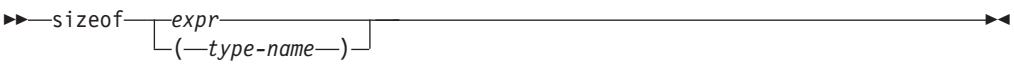
関連資料:

146 ページの『aligned 変数属性』
551 ページの『ベクトル処理の拡張機能サポート』

sizeof 演算子

sizeof 演算子は、オペランドのサイズをバイト数で生成します。オペランドは、式であってもよいし、括弧で囲んだ型の名前であってもかまいません。

sizeof 演算子の構文



どちらの種類のオペランドであっても、結果は左辺値ではなく、定数整数値です。結果の型は、ヘッダー・ファイル stddef.h で定義された符号なし整数型 size_t です。

プリプロセッサー・ディレクティブの中以外では、整数定数が必要なときは、sizeof 式を使用できます。sizeof 演算子がよく使われる 1 つの例は、ストレージ割り振り時、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 の使用法は、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。sizeof 演算子を使用して、データ型が表すサイズを判別することができます。次に例を示します。

```
sizeof(int);
```

型名に適用された sizeof 演算子は、その型のオブジェクトに使用されるメモリー量を、内部埋め込みまたは末尾埋め込みを含めて計算します。

IBM sizeof 演算子のオペランドは、ベクトルのサポートが使用可能になっている場合には、ベクトル変数、ベクトル型、またはベクトル型へのポインターの間接参照の結果とすることができます。このような場合、sizeof の戻り値は常に 16 です。

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1); // vector type: 16.
sizeof(&v1); // address of vector: 4.
sizeof(*pv1); // dereferenced pointer to vector: 16.
sizeof(pv1); // pointer to vector: 4.
sizeof(vector bool int); // vector type: 16.
```

IBM

複合型の場合、結果は次のとおりです。

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。

オペランド	結果
C++ クラス	結果は常にゼロ以外です。これは、そのクラスのオブジェクトのバイト数と等しくなり、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も含まれます。
C++ 参照	結果は、参照されるオブジェクトのサイズになります。

sizeof 演算子は、次のもの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、その式の型の名前だけに適用された場合と同じ結果になります。コンパイル時、コンパイラーは式を分析してその型を判別します。式の型分析で行われる通常の型変換はいずれも、直接的には sizeof 演算子に起因しません。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、型は判別するときに、これらの型変換を考慮します。例えば、次の例の 2 行目では、通常の算術変換が行われます。 short が 2 バイトのストレージを使用し、int が 4 バイトを使用するものとすると、次のようになります。

```
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 型であり、sizeof(int) と等価です。x が char、short、int 型、またはデフォルトの enum サイズの任意の列挙型の場合にも、値は 4 です。

可変長配列は、sizeof 式のオペランドとして使用できます。この場合は、オペランドは実行時に評価されます。したがって、可変長配列の各インスタンスのサイズがその存続期間中に変化しなかったとしても、そのサイズは整数定数でも定数式でもありません。

C++11 sizeof... は、可変数引数テンプレート機能によって導入される単項式演算子です。この演算子は、そのオペランドとしてパラメーター・パックを命名する式を受け入れます。次に、パラメーター・パックを展開し、そのパラメーター・パックに用意された引数の数を返します。次の例を検討してみます。

```
template<typename...T> void foo(T...args){
    int v = sizeof...(args);
}
```

この例では、変数 v が、パラメーター・パック args に用意される引数の数に割り当てられます。

注:

- sizeof... 演算子のオペランドは、パラメーター・パックを命名する式にする必要があります。
- sizeof 演算子のオペランドは、パラメーター・パックまたはパック拡張を命名する式にすることはできません。

詳しくは、 476 ページの『可変数引数テンプレート (C++11)』を参照してください。

C++11

関連資料:

115 ページの『型名』

169 ページの『整数定数式』

123 ページの『配列』

126 ページの『参照 (C++ のみ)』

typeof 演算子 (IBM 拡張)

typeof 演算子は、その引数の型を戻します。引数は、式または型です。この言語機能は、式から型を引き出す手段を提供していることになります。式 e を例にとると、__typeof__(e) は、型名が必要などの場所ででも、例えば、宣言の中ででも、キャストの中ででも使用できます。キーワードの代替スペル、__typeof__ が推奨されています。

typeof 演算子は、ベクトルのサポートが使用可能になっている場合に、ベクトル型をオペランドとして受け入れるように拡張されました。

typeof 演算子の構文

typeof 構成そのものは式ではなく、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が、構文は sizeof の構文に似ています。

以下の例は、その基本的な構文を示しています。式 e の場合:

```
int e;
__typeof__(e + 1) j;   /* the same as declaring int j;     */
e = (__typeof__(e)) f; /* the same as casting e = (int) f; */
```

typeof 構成を使用することは、typedef 名を宣言することと同等です。次の場合、

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

次のように書くことができます。

```
__typeof__(i) a;         /* all three constructs have the same meaning */
__typeof__(int[2]) a;
__typeof__(T) a;
```

このコードの動作は、int a[2]; を宣言した場合と同じです。

ビット・フィールドの場合、typeof は、そのビット・フィールドの基礎となる型を表します。例えば、int m:2;、typeof(m) は int です。ビット・フィールドのプロパティーは予約されないので、typeof(m) n; 内の n は int n と同じですが、int n:2 と同じではありません。

typeof 演算子は、sizeof の中およびそれ自身の中にネストすることができます。int を指すポインターの配列としての arr の以下の 2 つの宣言は等価です。

```
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の中で有用です。次に例を示します。

```
#define SWAP(a,b) { __typeof__(a) temp; temp = a; a = b; b = temp; }
```

注:

1. typeof キーワードおよび __typeof__ キーワードは、次のようにサポートされています。
   - C __typeof__ キーワードは、**xlc** 呼び出しコマンドまたは **-qlanglvl=extc89**、**-qlanglvl=extc99**、または **-qlanglvl=extended** オプションを使用したコンパイルで認識されます。 typeof キーワードは、**-qkeyword=typeof** でコンパイルしたときのみ認識されます。
   - C++ typeof キーワードおよび __typeof__ キーワードは、デフォルトで認識されるようになっています。
2. C++11 decltype は、__typeof__ と同じ目的を果たします。

関連資料:

93 ページの『decltype(*expression*) 型指定子 (C++11)』

115 ページの『型名』

89 ページの『typedef 定義』

67 ページの『ベクトル型 (IBM 拡張)』

__real__ および __imag__ 演算子

XL C/C++ では、C99 および C++ 標準は、単項演算子 __real__ および __imag__ をサポートするように拡張されています。これらの演算子を使用することにより、複素数型の実数部と虚数部を抽出することができます。これらの拡張機能は、GNU C を使用して開発されたアプリケーションの移植を容易にすることを目的としてインプリメントされています。

__real__ および __imag__ 演算子の構文

var_identifier は、以前に宣言された複素数変数の名前です。__real__ 演算子は複素数変数の実数部を戻し、__imag__ 演算子はその変数の虚数部を戻します。これらの演算子のオペランドが左辺値の場合、結果の式は、左辺値が許可されているどのコンテキストででも使用できます。 これらの演算子は、複素数変数の初期化のときと、複素数型を表すフォーマット指定子を持たない、printf および scanf などのライブラリー関数呼び出しへの引数として特に有用です。次に例を示します。

```
float _Complex myvar;
__imag__(myvar) = 2.0f;
__real__(myvar) = 3.0f;
```

上記の例は、複素数変数 myvar の虚数部を 2.0*i* に、実数部を 3.0 に初期化し、さらに次の関数は、

```
printf("myvar = %f + %f * i¥n", __real__(myvar), __imag__(myvar));
```

以下を印刷します。

```
myvar = 3.000000 + 2.000000 * i
```

関連資料:

複素数リテラル (C のみ)

複素数浮動小数点型 (C のみ)

vec_step 演算子

vec_step 演算子はベクトル型オペランドを使用し、ベクトル・エレメントへのポインターを 16 バイト分 （ベクトルのサイズ) 移動させるために必要な、ポインターの増分量を示す整数値を返します。以下の表は、データ型別の値の要約を示しています。

表 31. データ型別の vec_step の増分値

vec_step 式	値
vec_step(vector unsigned char) vec_step(vector signed char) vec_step(vector bool char)	16
vec_step(vector unsigned short) vec_step(vector signed short) vec_step(vector bool short)	8
vec_step(vector unsigned int) vec_step(vector signed int) vec_step(vector bool int)	4
vec_step(vector unsigned long long) vec_step(vector signed long long) vec_step(vector bool long long)	2
vec_step(vector pixel)	8
vec_step(vector float)	4
vec_step(vector double)	2

vec_step 演算子に関する詳しい説明については、「*AltiVec Technology Programming Interface Manual*」(http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手可能) を参照してください。

2 項式

2 項式 には、1 つの演算子によって分離される 2 つのオペランドが含まれます。サポートされる 2 項演算子は次のとおりです。

- 『割り当て演算子』
- 191 ページの『乗算演算子 *』
- 192 ページの『除算演算子 /』
- 192 ページの『剰余演算子 %』
- 192 ページの『加算演算子 +』
- 193 ページの『減算演算子 -』
- 193 ページの『ビット単位の左および右シフト演算子 << >>』
- 194 ページの『関係演算子 < > <= >=』
- 195 ページの『等価および非等価演算子 == !=』
- 196 ページの『ビット単位 AND 演算子 &』
- 197 ページの『ビット単位排他 OR 演算子 ^』
- 197 ページの『ビット単位包含 OR 演算子 |』
- 198 ページの『論理 AND 演算子 &&』
- 199 ページの『論理 OR 演算子 ||』
- 199 ページの『配列添え字演算子 [ ]』
- 201 ページの『コンマ演算子 ,』
- 203 ページの『メンバーを指すポインター演算子 .* ->* (C++ のみ)』

2 項演算子は、すべてに左から右への結合順序が指定されますが、すべてに同じ優先順位が付けられるわけではありません。2 項演算子のランキングおよび優先順位規則を 225 ページの表 36 で要約しています。

ほとんどの 2 項演算子のオペランドが評価される順序は、指定されていません。正しい結果を得るためには、コンパイラーがオペランドを評価する順序に依存する 2 項式を作成しないようにします。

演算子の説明で示されているように、ほとんどの 2 項式のオペランドで、通常の算術変換が行われます。

関連資料:

165 ページの『左辺値と右辺値』

154 ページの『算術変換およびプロモーション』

割り当て演算子

割り当て式 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。割り当て演算子には、次の 2 つの型があります。

- 190 ページの『単純割り当て演算子 =』
- 190 ページの『複合割り当て演算子』

すべての割り当て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、割り当てが完了した後の左方オペランドの値です。

C 割り当て式の結果は左辺値ではありません。C++ 割り当て式の結果は左辺値です。

すべての割り当て演算子には、同じ優先順位が付けられ、右から左の結合順序が適用されます。

単純割り当て演算子 =

単純割り当て演算子の形式は、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またはベクトル型ではない場合は、右方オペランドは暗黙的に左方オペランドの型に変換されます。この変換された型は、`const` または `volatile` によって修飾されることはありません。

左方オペランドがクラス型である場合、その型は完全でなければなりません。左方オペランドのコピー割り当て演算子が呼び出されます。

左方オペランドが参照型のオブジェクトの場合は、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割り当てます。

IBM 割り当て演算子が拡張され、ベクトル型のオペランドが使用できるようになりました。割り当て式の両辺は同じベクトル型でなければなりません。

複合割り当て演算子

複合割り当て演算子は、2 項演算子と単純割り当て演算子で構成されます。複合割り当て演算子は、両方のオペランドに 2 項演算子の演算を実行し、その演算の結果を左方オペランドに保管します。左方オペランドは変更可能な左辺値でなければなりません。

次の表では、複合割り当て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
+= または -=	算術	算術
+= または -=	ポインター	整数型
*=、/=、および %=	算術	算術
<<=、>>=、&=、^=、および |=	整数型	整数型

次の式

```
a *= b + c
```

は、以下と等価です。

```
a = a * (b + c)
```

そして、次の式とは等価でない ことに注意してください。

```
a = a * b + c
```

次の表では、複合割り当て演算子をリストし、各演算子を使用した式を示します。

演算子	例	等価な式
+=	index += 2	index = index + 2
-=	*pointer -= 1	*pointer = *pointer - 1
*=	bonus *= increase	bonus = bonus * increase
/=	time /= hours	time = time / hours
%=	allowance %= 1000	allowance = allowance % 1000
<<=	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++ オペランドの型の表に加えて、左方オペランドがクラス型でない場合、式は左方オペランドの cv 非修飾の型に暗黙的に変換されます。しかし、左方オペランドがクラス型の場合、そのクラスは完全になり、そのクラスのオブジェクトへの割り当ては、コピー割り当て操作として行われます。複合式および条件式は C++ では左辺値であり、そのため、これらは、複合割り当て式の中で左方オペランドになることができます。

IBM C GNU C 言語機能が使用可能になっているとき、複合式および条件式は、それらのオペランドが左辺値である場合に左辺値として認められます。複合式 (a, b) の以下の複合割り当ては、式 b が左辺値であれば、すなわち、より一般的に言えば、シーケンス中の最後の式が左辺値であれば、GNU C では有効です。

```
(a,b) += 5  /* Under GNU C, this is equivalent to
a, (b += 5)    */
```

関連資料:

165 ページの『左辺値と右辺値』

116 ページの『ポインター』

101 ページの『型修飾子』

乗算演算子 *

* (乗算) 演算子は、そのオペランドの積を生成します。オペランドは、算術型または列挙型でなければなりません。結果は、左辺値ではありません。オペランドには、通常の算術変換が実行されます。

乗算演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数の乗算演算子を含む式の中でオペランドの再配置を行うことができます。例えば、次の式

```
sites * number * cost
```

は、次のいずれかの方法に解釈できます。

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

除算演算子 /

/ (除算) 演算子はそのオペランドの商を生成します。オペランドが両方とも整数である場合は、小数部分 (剰余) は破棄されます。小数部分の破棄は、しばしば、*ゼロに向かった切り捨て* と呼ばれます。オペランドは、算術型または列挙型でなければなりません。右方オペランドはゼロにできません。右方オペランドが 0 に評価される場合、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。

C++ 両方のオペランドが負の場合、剰余の符号も負です。そうでない場合、剰余の符号は商の符号と同じです。

剰余演算子 %

% (剰余) 演算子は、左方オペランドを右方オペランドで割り算した剰余を生成します。例えば、式 5 % 3 は 2 を生成します。結果は、左辺値ではありません。

オペランドは両方とも、整数型または列挙型でなければなりません。右方オペランドが評価の結果 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は、b が 0 でなく、a/b が表示可能な場合は、結果は次の式のように、常に値 a になります。

```
( a / b ) * b + a %b;
```

オペランドには、通常の算術変換が実行されます。

両方のオペランドが負の場合、剰余の符号も負です。そうでない場合、剰余の符号は商の符号と同じです。

加算演算子 +

+ (加算) 演算子は、そのオペランドの合計を生成します。オペランドは両方とも算術型を保持するか、一方のオペランドがオブジェクト型を指すポインターで、もう一方のオペランドが整数型または列挙型を保持する必要があります。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

配列内のオブジェクトを指すポインターは、整数型を持つ値に加算できます。結果は、ポインター・オペランドと同じ型のポインターになります。結果は、オリジナ

ルのエレメントから、添え字として扱われる整数値の量だけオフセットされた、配列の中の別のエレメントを参照します。結果のポインターが、配列の外側のストレージ (配列の外側の最初のロケーション以外) を指す場合、結果は未定義です。配列の終わりより 1 つ後のエレメントを指すポインターを使用して、そのアドレスにあるメモリーの内容にアクセスすることはできません。コンパイラーは、ポインターの境界検査は行いません。例えば、以下の例で、加算の後で、ptr は配列の 3 番目のエレメントを指します。

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

関連資料:

117 ページの『ポインター演算』

160 ページの『ポインター型変換』

減算演算子 -

- (減算) 演算子は、そのオペランドの差を生成します。オペランドは両方とも算術型または列挙型を保持するか、左方オペランドがポインター型で、右方オペランドがポインターの型と同じ型か整数型または列挙型を保持する必要があります。整数値からポインターを減算することはできません。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

左方オペランドがポインターで、右方オペランドが整数型を保持するときは、コンパイラーは、右方の値を相対位置のアドレスに変換します。結果は、ポインター・オペランドと同じ型のポインターになります。

オペランドが両方とも同じ配列内のエレメントを指すポインターである場合は、結果は、2 つのアドレスを分離するオブジェクトの数です。この数値の型は ptrdiff_t で、これはヘッダー・ファイル stddef.h で定義されています。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、動作は未定義です。

関連資料:

117 ページの『ポインター演算』

160 ページの『ポインター型変換』

ビット単位の左および右シフト演算子 << >>

ビット単位シフト演算子は、2 進数オブジェクトのビット値を移動します。左方オペランドには、シフトされる値を指定します。右方オペランドには、値のビットがシフトされる桁数を指定します。結果は、左辺値ではありません。両方のオペランドに同じ優先順位が付けられ、左から右の結合順序が指定されます。

演算子	使用法
<<	ビットが左方にシフトされることを指示します。
>>	ビットが右方にシフトされることを指示します。

各オペランドは、整数型または列挙型でなければなりません。コンパイラーは、オペランドの整数拡張を実行し、その後、右方オペランドが int 型に変換されます。結果には、左方オペランドと同じ型が指定されます (算術変換の後)。

右方オペランドが負の値、またはシフトされる式のビット幅より大きいかまたは等しい値を保持しないようにする必要があります。このような値でビット単位シフトを行うと、結果は予測不能な値になります。

右方オペランドに 0 がある場合は、結果は左方オペランドの値になります (通常の算術変換が実行された後)。

<< 演算子は、空になったビットをゼロで埋めます。例えば、left_op が値 4019 を持っている場合、left_op のビット・パターン (16 ビットの形式) は次のようになります。

```
0000111110110011
```

式 left_op << 3 は、次のビット・パターンを生成します。

```
0111110110011000
```

式 left_op >> 3 は、次のビット・パターンを生成します。

```
0000000111110110
```

関係演算子 < > <= >=

関係演算子は、2 つのオペランドを比較して、リレーションシップの妥当性を判別します。次の表では、4 つの関係演算子を説明します。

演算子	使用法
<	左方オペランドの値が、右方オペランドの値より小さいかどうかを示します。
>	左方オペランドの値が、右方オペランドの値より大きいかどうかを示します。
<=	左方オペランドの値が、右方オペランドの値より小さいまたは等しいかどうかを示します。
>=	左方オペランドの値が、右方オペランドの値より大きいまたは等しいかどうかを示します。

オペランドは両方とも、算術型または列挙型を保持するか、同じ型を指すポインターでなければなりません。

C 結果の型は int で、指定された関係が真であれば値 1 を持ち、偽であれば 0 を持ちます。C++ 結果の型は bool で、値 true または値 false を持ちます。

結果は、左辺値ではありません。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。ポインターが同じ配列のオブジェクトを参照しない場合は、結果は未定義になります。

ポインターは、0 に評価される定数式と比較することができます。また、ポインターを `void*` 型のポインターと比較することもできます。ポインターを `void*` 型のポインターに変換します。

2 つのポインターが同じオブジェクトを参照する場合は、この 2 つのポインターは等しいと見なされます。 2 つのポインターが同一オブジェクトの非静的メンバーを参照する場合、これらのポインターがアクセス指定子によって分離されていなければ、後で宣言されるオブジェクトを指すポインターの方がより大きいです。分離されていれば、比較は未定義です。 2 つのポインターが同じ共用体のデータ・メンバーを参照する場合は、この 2 つのポインターは同じアドレス値を保持します。

2 つのポインターが同じ配列のエレメント、または配列の最後のエレメントを超えて最初のエレメントを参照する場合、より大きい添え字値をもっているエレメントを指すポインターの方が、より大きいです。

関係演算子では、同じオブジェクトのメンバーだけを比較できます。

関係演算子には、左から右の結合順序が指定されます。例えば、次の式

```
a < b <= c
```

は、次のように解釈されます。

```
(a < b) <= c
```

`a` の値が `b` の値よりも小さい場合は、最初の関係演算は、C では値 1 を、C++ では `true` を生成します。その後で、コンパイラーは、値 `true` (または 1) を `c` の値と比較します (必要であれば、整数拡張が実行されます)。

等価および非等価演算子 == !=

等価演算子は、関係演算子と同様に、リレーションシップの妥当性について 2 つのオペランドを比較します。ただし、等価演算子には、関係演算子よりも低い優先順位が付けられます。次の表で、2 つの等価演算子を説明します。

演算子	使用法
==	左方オペランドの値が、右方オペランドの値と等価かどうかを示します。
!=	左方オペランドの値が、右方オペランドの値と等価でないかどうかを示します。

オペランドは両方とも算術型または列挙型を保持するか、同じ型を指すポインターである必要があります。あるいは、一方のオペランドがポインター型を保持し、もう一方のオペランドが void を指すポインターまたは NULL ポインターである必要があります。

C 結果の型は int で、指定された関係が真であれば値 1 を持ち、偽であれば 0 を持ちます。C++ 結果の型は bool で、値 true または値 false を持ちます。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。

一方のオペランドがポインターで、もう一方のオペランドが値 0 の整数の場合、== 式は、ポインターのオペランドが NULL に評価される場合にだけ真になります。!= 演算子は、ポインター・オペランドが NULL に評価されない 場合に、真になります。

等価演算子を使用して、型が同じでも、同じオブジェクトに属さないメンバーを指すポインターを比較することもできます。次の式には、等価演算子と関係演算子の例が含まれています。

```
time < max_time == status < complete
letter != EOF
```

注: 等価演算子 (==) を、割り当て (=) 演算子と混同しないでください。

次に例を示します。

if (x == 3)

x が 3 に等しい場合、評価の結果真 (または 1) になります。このような等価テストでは、意図しない割り当てを防ぐために、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) が評価の結果ゼロ以外の値 (3) になるので、真になります。また、この式では、3 が x に割り当てられます。

関連資料:

単純割り当て演算子 =

ビット単位 AND 演算子 &

& (ビット単位 AND) 演算子は、第 1 オペランドの各ビットと第 2 オペランドの対応するビットを比較します。ビットが両方とも 1 であれば、対応する結果のビットを 1 にセットします。 1 でなければ、対応する結果のビットを 0 にセットします。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。

ビット単位 AND 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位 AND 演算子を含む式の中でオペランドの再配置を行うことができます。

次の例では、16 ビットの 2 進数で表された、a、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	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、

```
    1 & 4 は評価の結果 0 になります。
一方
    1 && 4 は評価の結果 true になります。
```

ビット単位排他 OR 演算子 ^

ビット単位排他 OR 演算子 (EBCDIC では ^ シンボルは ¬ シンボルで表す) 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 ??' で表すことができ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、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

関連資料:

43 ページの『3 文字表記シーケンス』

ビット単位包含 OR 演算子 |

| (ビット単位包含 OR) 演算子は、各オペランドの値 (2 進形式) を比較し、いずれかのオペランドのどのビットの値が 1 であるかを示すビット・パターンの値を生成します。両方のビットが 0 であると、その結果のビットは 0 になり、それ以外の結果は 1 になります。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。結果は、左辺値ではありません。

ビット単位包含 OR 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位包含 OR 演算子を含む式の中でオペランドの再配置を行うことができます。 | 文字は、3 文字表記文字の ??! によって表されることに注意してください。

次の例では、16 ビットの 2 進数で表された、a、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、

```
1 | 4 は評価の結果 5 になります。
```
一方
```
1 || 4 は評価の結果 true になります。
```

関連資料:

43 ページの『3 文字表記シーケンス』

論理 AND 演算子 &&

&& (論理 AND) 演算子は、両方のオペランドが真であるかどうかを示します。

C 両方のオペランドにゼロ以外の値がある場合、結果の値は 1 になります。そうでない場合、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、その結果の値は true になります。そうでない場合は、結果の値は偽になります。両オペランドは、暗黙的に bool へ変換されます。結果型は bool です。

& (ビット単位 AND) 演算子と異なり、&& 演算子では、オペランドは必ず左から右に評価されます。左方オペランドが 0 (または偽) になる場合、右方オペランドは評価されません。

次の例では、論理 AND 演算子を含む式が評価される方法を示します。

式	結果
1 && 0	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 偽) に評価されるときは、式 x / y は評価されません。

注: 論理 AND 演算子 (&&) を、ビット単位 AND 演算子 (&) と混同しないでください。次に例を示します。

1 && 4 は評価の結果 1 (または true) になります。
一方
1 & 4 は評価の結果 0 になります。

論理 OR 演算子 ||

|| (論理 OR) 演算子は、いずれかのオペランドが真であるかどうかを示します。

C オペランドのいずれかにゼロ以外の値がある場合は、結果の値は 1 になります。そうでない場合、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++ いずれか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、結果の値は偽になります。両オペランドは、暗黙的に bool へ変換されます。結果型は bool です。

| (ビット単位包含 OR) 演算子と異なり、|| 演算子では、オペランドは必ず左から右に評価されます。左方オペランドがゼロ以外の値 (または真) である場合は、右方オペランドは評価されません。

次の例では、論理 OR 演算子を含む式が評価される方法を示します。

式	結果
1 || 0	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 (または真) の値になる場合、式 ++y は、評価されません。

注: 論理 OR 演算子 (||) を、ビット単位 OR 演算子 (|) と混同しないでください。次に例を示します。

1 || 4 は評価の結果 1 (または真) になります。
一方
1 | 4 は評価の結果 5 になります。

配列添え字演算子 [ ]

後置式とその後に続く [ ] (大括弧) 内の式が、配列のエレメントを指定します。大括弧内の式は、*添え字* と呼ばれます。配列の最初のエレメントの添え字はゼロです。

定義により、式 a[b] は、式 *((a) + (b)) と等価です (また、加算は結合であるので、b[a] とも等価です)。式 a と b の間で、一方は、型 T に対するポインターでなければなりません。そして他方は、整数型または列挙型をもっていなければなりません。配列添え字の結果は、左辺値になります。次の例は、このことを示しています。

```
#include <stdio.h>

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

上記の例の出力は、次のとおりです。

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

C++ クラスの operator[] を多重定義した場合、添え字演算子とポインター演算の間の関係だけでなく、添え字演算子が必要とする式の型に関する上述の制限も適用されません。 C++

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;
         }
      }
   }
```

C C99 は、左辺値でない配列に配列添え字を許可します。次の例は、C99 で有効です。

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

C

関連資料:

116 ページの『ポインター』

63 ページの『整数型』
165 ページの『左辺値と右辺値』
123 ページの『配列』
334 ページの『添え字の多重定義』
117 ページの『ポインター演算』

ベクトル添え字演算子 [ ] (IBM 拡張)

配列エレメントへのアクセス方法と同様に、ベクトル・データ型の個々のエレメントへは、大括弧を使用することによってアクセスできます。ベクトル・データ型の後には、エレメントの位置が入っている 1 組の大括弧が続きます。最初のエレメントの位置は 0 です。結果の型は、ベクトル型に含まれているエレメントの型です。

例:

```
vector unsigned int v1 = {1,2,3,4};
unsigned int u1, u2, u3, u4;
u1 = v1[0];       // u1=1
u2 = v1[1];       // u2=2
u3 = v1[2];       // u3=3
u4 = v1[3];       // u4=4
```

注: 以下の組み込み関数を使用して、ベクトルの個々のエレメントにアクセスし、操作することもできます。

- **vec_extract**
- **vec_insert**
- **vec_promote**
- **vec_splats**

コンマ演算子 ,

コンマ式 には、任意の型の 2 つのオペランドがコンマで区切られて含まれており、左から右の結合順序が適用されます。左方オペランドは評価されますが、副次作用が生じる可能性があり、値がある場合、その値は破棄されます。次に、右方オペランドが評価されます。コンマ式の結果の型および値は、通常の単項変換後の右方オペランドの型および値です。

C コンマ式の結果は左辺値ではありません。

C++ C++ では、結果は、右方オペランドが左辺値であれば、左辺値です。以下のステートメントは等価です。

```
r = (a,b,...,c);
a; b; r = c;
```

違いは、ループ制御式などの式コンテキストには、コンマ演算子が適切なことがあるということです。

右方オペランドが左辺値である場合、複合式のアドレスを取ることができます。

```
&(a, b)
a, &b
```

C++

コンマ演算子は結合するので、コンマで区切られた任意の数の式は単一の式を形成します。コンマ演算子を使用すると、副次式は左から右の順に評価されることが保証され、最後の副次式の値が式全体の値になります。次の例では、omega が 11 の場合は、式は delta を増分し、値 3 を alpha に割り当てます。

```
alpha = (delta++, omega % 4);
```

最初のオペランドの評価後に評価順序点が生じます。 delta の値は破棄されます。同様に、以下の例では、次の式

```
intensity++, shade * increment, rotate(direction);
```

の値は、次の式の値になります。

```
rotate(direction)
```

コンマ文字が使われているコンテキストによっては、あいまいさを避けるために括弧が必要な場合があります。例えば、次の関数

```
f(a, (t = 3, t + 2), c);
```

の引数は値 a、値 5、および値 c の 3 個だけです。括弧が必要な他のコンテキストは、構造体宣言子リストおよび共用体宣言子リスト内のフィールド長の式の中、列挙型宣言子リスト内の列挙型の値の式の中、および宣言と初期化指定子内の初期化の式の中です。

上の例では、関数呼び出しの中の引数の式を区切るためにコンマが使用されています。このコンテキストでは、コンマを使用しても、関数引数の評価順序 (左から右) は保証されません。

コンマ演算子の基本的な使用目的は、次のような状況で、副次作用をもたらすことです。

- 関数の呼び出し
- 反復ループへの入力または繰り返し
- 条件のテスト
- 副次作用は必要であるが、式の結果は今すぐに必要ではないその他の状況

次の表では、いくつかのコンマ演算子の使用例を示します。

ステートメント	効果
`for (i=0; i<2; ++i, f() );`	for ステートメントでは、i が増分され、反復のたびに f() が呼び出されます。
`if ( f(), ++i, i>1 ) { /* ... */ }`	if ステートメントでは、関数 f() が呼び出され、変数 i が増分され、変数 i が値に対してテストされます。このコンマ式内の最初の 2 つの式は、式 i>1 の前に評価されます。最初の 2 つの式の結果に関係なく、3 番目の式が評価され、その結果が if ステートメントを処理するかどうかを判別します。

ステートメント	効果
`func( ( ++a, f(a) ) );`	`func()` への関数呼び出しでは、 `a` が増分され、結果の値が関数 `f()` に渡され、 `f()` の戻り値が `f()` に渡されます。関数 `func()` には、引数が 1 つだけ渡されます。これは、関数引数のリスト内で、コンマ式が括弧で囲まれているからです。

メンバーを指すポインター演算子 .* ->* (C++ のみ)

メンバーを指すポインター演算子には、`.*` と `->*` の 2 つがあります。

クラス・メンバーを指すポインターを間接参照するには、`.*` 演算子を使用します。第 1 オペランドは、クラス型でなければなりません。第 1 オペランドの型がクラス型 `T`、またはクラス型 `T` から派生したクラスの場合は、第 2 オペランドはクラス型 `T` のメンバーを指すポインターでなければなりません。

クラス・メンバーを指すポインターを間接参照するには、`->*` 演算子を使用します。第 1 オペランドは、クラス型を指すポインターでなければなりません。第 1 オペランドの型がクラス型 `T` を指すポインター、またはクラス型 `T` から派生したクラスを指すポインターの場合は、第 2 オペランドはクラス型 `T` のメンバーを指すポインターでなければなりません。

`.*` および `->*` 演算子は、第 2 オペランドを第 1 オペランドにバインドします。結果は、第 2 オペランドで指定された型のオブジェクトまたは関数になります。

`.*` または `->*` の結果が関数の場合は、結果を `( )` (関数呼び出し) 演算子のオペランドとしてだけ使用できます。第 2 オペランドが左辺値の場合は、`.*` または `->*` の結果は左辺値になります。

関連資料:

355 ページの『クラス・メンバー・リスト』

361 ページの『メンバーを指すポインター』

条件式

条件式 は、C++ では暗黙的にタイプ `bool` へ変換される条件 ($operand_1$)、条件が true に評価される場合に評価される式 ($operand_2$)、および条件が値 false を持っている場合に評価される式 ($operand_3$) を含む複合式です。

条件式には、2 つの部分で構成される 1 つの演算子があります。 `?` 記号は、条件の後に続き、 `:` 記号は 2 つのアクション式の間に使用されます。 `?` と `:` の間の式は、すべて 1 つの式として扱われます。

第 1 オペランドは、スカラー型を持つ必要があります。第 2 オペランドと第 3 オペランドの型は、次のいずれかでなければなりません。

- 算術型
- 互換ポインター、構造体、または共用体型
- void

第 2 オペランドと第 3 オペランドは、ポインターまたは NULL 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 指すポインターの場合は、これらのポインター・オブジェクトには互換性があります。

第 1 オペランドが評価され、その値によって第 2 オペランドまたは第 3 オペランドを評価するかどうかが判別されます。

- 値が真の場合は、第 2 オペランドが評価されます。
- 値が偽の場合は、第 3 オペランドが評価されます。

結果は、第 2 オペランドまたは第 3 オペランドの値になります。

2 番目または 3 番目の式が評価の結果、算術型になる場合、値に通常の算術変換を実行します。次の表は、第 2 オペランドと第 3 オペランドの型により結果の型がどのように決まるかを示します。

条件式は、第 1 オペランドと第 3 オペランドについては右から左の結合順序が適用されます。左端のオペランドが最初に評価され、次に、残りの 2 つのオペランドのいずれか 1 つだけが評価されます。以下の式は等価です。

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

C の条件式での型 (C のみ)

C では、条件式は左辺値ではなく、その結果も左辺値ではありません。

表 32. C の条件式のオペランドと結果の型

一方のオペランドの型	もう一方のオペランドの型	結果の型
算術	算術	通常の算術変換後の算術型
構造体または共用体型	互換構造体または共用体型	両方のオペランドにすべての修飾子が付く構造体または共用体型
void	void	void
互換型を指すポインター	互換型を指すポインター	型に指定されたすべての修飾子が付く型を指すポインター
型を指すポインター	NULL ポインター (定数 0)	型を指すポインター
オブジェクトまたは不完全型を指すポインター	void を指すポインター	型に指定されたすべての修飾子が付く void を指すポインター

IBM GNU C では、条件式は、その型が void でなく、その分岐の両方が有効な左辺値であれば、有効な左辺値です。次の条件式 (a ? b : c) は、GNU C では有効です。

```
(a ? b : c) = 5
/*  Under GNU C, equivalent to (a ? b = 5 : (c = 5))  */
```

この拡張機能は、拡張言語レベルの 1 つでコンパイルされるとき、使用可能です。

IBM

C++ の条件式での型 (C++ のみ)

表 33. C++ の条件式のオペランドと結果の型

一方のオペランドの型	もう一方のオペランドの型	結果の型
型への参照	型への参照	通常の参照変換後の参照
クラス T	クラス T	クラス T
クラス T	クラス X	型変換が存在する場合のクラス型。可能な型変換が複数ある場合は、結果は不確定になります。
throw 式	その他 (型、ポインター、参照)	throw 式でない式の型

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 に割り当てます。

```
x = (y > z) ? y : z;
```

以下のステートメントは、前述の式と同等です。

```
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割り当て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 ?: 演算子より低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 が割り当て式 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 に割り当てるのは、i == 7 が偽のときで、最後のオペランドを小括弧で囲みます。

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

キャスト式

キャスト演算子は*明示的型変換* に使用されます。この演算子は式の値を、指定された型に変換します。

次のキャスト演算子がサポートされています。

- 『Cast 演算子 ()』
- 209 ページの『static_cast 演算子 (C++ のみ)』
- 211 ページの『reinterpret_cast 演算子 (C++ のみ)』
- 212 ページの『const_cast 演算子 (C++ のみ)』
- 214 ページの『dynamic_cast 演算子 (C++ のみ)』

Cast 演算子 ()

Cast 式の構文

C C では、キャスト演算の結果は左辺値ではありません。 C

C++ C++ では、キャスト結果は、以下のいずれかの値のカテゴリーに属します。

- type が左辺値参照型であるか C++11 関数型に対する右辺値参照である場合 C++11 、キャストの結果は左辺値です。
- C++11 type がオブジェクト型に対する右辺値参照である場合、キャストの結果は xvalue です。 C++11
- それ以外の場合はすべて、キャストの結果は C++11 (prvalue) C++11 右辺値になります。

C++

以下は、サイズ 10 の整数配列を動的に作成するためのキャスト演算子の使用法を示したものです。

```
#include <stdlib.h>

int main(void) {
   int* myArray = (int*) malloc(10 * sizeof(int));
   free(myArray);
   return 0;
}
```

malloc ライブラリー関数は、その引数のサイズのオブジェクトを保持するメモリーを指す void ポインターを戻します。ステートメント int* myArray = (int*) malloc(10 * sizeof(int)) では以下のステップが実行されます。

- 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へ変換します。
- その整数ポインターを myArray に割り当てます。

C++ C++ では、キャスト式で以下のオブジェクトも使用することができます。

- 関数スタイルのキャスト
- C++ 変換演算子。static_cast など。

関数スタイルの表記は、*expression* の値を型 *type* に変換します。

```
type(expression)
```

次の例は、C スタイルのキャスト演算子、C++ 関数スタイルのキャスト演算子、および C++ キャスト演算子を使った同じ値のキャストを示しています。

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

int main() {
  float num = 98.76;
  int x1 = (int) num;
  int x2 = int(num);
  int x3 = static_cast<int>(num);

  cout << "x1 = " << x1 << endl;
  cout << "x2 = " << x2 << endl;
  cout << "x3 = " << x3 << endl;
}
```

上記の例の出力は、次のとおりです。

```
x1 = 98
x2 = 98
x3 = 98
```

整数 x1 には、C スタイル・キャストを使用して num が明示的に int へ変換された値が割り当てられます。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割り当てられます。整数 x3 には、static_cast 演算子を使用して変換された値が割り当てられます。

C++ の場合は、キャスト式のオペランドには、クラス型を指定できます。オペランドにクラス型が指定された場合は、クラスにユーザー定義の型変換関数がある任意の型にキャストすることができます。これらのキャストは、ターゲット型がクラスであれば、コンストラクターを呼び出すことができますし、ソース型がクラスであれば、型変換関数を呼び出すことができます。これらのキャストは、両方の条件が該当する場合は、未確定になることがあります。 C++

共用体型へのキャスト (C のみ) (IBM 拡張)

共用体型へのキャストにより、共用体のメンバーを、そのメンバーが属している共用体と同じ型にキャストすることができます。このようなキャストでは、左辺値は生成されません。この機能は、GNU 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たものであり、C99 の拡張機能としてサポートされています。

共用体型にキャストできるのは、その共用型のメンバーとして明示的に存在している型のみです。キャストには、共用体型のタグか、または typedef 式で宣言されている共用体型名を使用できます。指定する型は完全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っている場合、共用体型へのキャストで使用できます。ビット・フィールドを共用体型にキャストできるのは、共用体に同じ型のビット・フィールド・メンバーが含まれている場合であり、その長さは同じでなくても構いません。以下のコードは、共用体への単純なキャストの例を示しています。

```
#include <stdio.h>

union f {
   char t;
   short u;
   int v;
   long w;
   long long x;
   float y;
   double z;
};

int main() {
  union f u;
  char a = 1;
  u = (union f)a;
  printf("u = %i¥n", u.t);
}
```

この例の出力は次のとおりです。

```
u = 1
```

ネストされた共用体へのキャストも可能です。以下の例では、double 型 dd を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とおりです。

```
U.u2 is 1.234
```

共用体キャストは、関数引数として、さらに静的または非静的なデータ・オブジェクト の初期化のための定数式の一部として有効であり、かつ複合リテラル・ステートメントの中でも有効です。次の例は、静的データ・オブジェクトの初期化のための式の一部として使用される共用体へのキャストです。

```
struct S{
  int a;
}s;
```

```
union U{
  struct S *s;
};

struct T{
  union U u;
};

static struct T t[] = { {(union U)&s} };
```

関連資料:

71 ページの『構造体および共用体』

111 ページの『transparent_union 型属性 (C のみ) 』

115 ページの『型名』

430 ページの『変換関数』

428 ページの『変換コンストラクター』

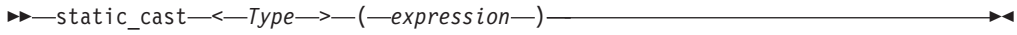
165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

static_cast 演算子 (C++ のみ)

static_cast 演算子は、与えられた式を指定された型に変換します。

static_cast 演算子の構文



C++11 右不等号括弧の機能により、2 つの連続する > トークンの代わりに >> トークンを使用して、template_id を static_cast 演算子の Type として指定することができます。詳しくは、447 ページの『クラス・テンプレート』を参照してください。 C++11

static_cast<Type>(expression) の結果は、以下のいずれかの値のカテゴリーに属します。

- Type が左辺値参照型である場合 C++11 または関数型に対する右辺値参照である場合 C++11 、static_cast<Type>(expression) は左辺値です。
- C++11 Type がオブジェクト型に対する右辺値参照である場合、static_cast<Type>(expression) は xvalue です。 C++11
- それ以外すべての場合は、static_cast<Type>(expression) は、 C++11 (prvalue) C++11 右辺値です。

static_cast 演算子の例を以下に示します。

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

int main() {
  int j = 41;
  int v = 4;
  float m = j/v;
  float d = static_cast<float>(j)/v;
  cout << "m = " << m << endl;
  cout << "d = " << d << endl;
}
```

この例の出力は次のとおりです。

```
m = 10
d = 10.25
```

この例では、m = j/v; は、型 int の答えを作成します。なぜなら、j と v の両方とも整数であるからです。逆に、d = static_cast<float>(j)/v; は、型 float の答えを作成します。static_cast 演算子は、変数 j を、型 float に変換します。これによって、コンパイラーは、型 float の答えをもつ割り算を生成できます。すべての static_cast 演算子は、コンパイル時に解決します。そして、どの const または volatile 修飾子も除去しません。

static_cast 演算子を NULL ポインターに適用すると、ターゲット型の NULL ポインター値に変換されます。

本コンパイラーは、以下の種類のキャスト演算をサポートしています。

- 型 A の左辺値から型 B&。キャスト結果は、型 B の左辺値です
- C++11 型 A の左辺値または xvalue から型 B&&。キャスト結果は、型 B の xvalue になります。 C++11
- C++11 (prvalue) C++11 A へのポインターの右辺値から B へのポインター
- C++11 型 A の左辺値から型 B&& (型 A の xvalue を型 B&& の参照に直接バインドできる場合) C++11
- 式 e から型 T (直接初期化 T t(e) が有効な場合)。

最初の 3 つのキャスト演算がサポートされるには、以下の条件が満たされている必要があります。

- A は、B の基底クラスである。
- 型 B を指すポインターから型 A を指すポインターへの標準の変換が存在する。
- 型 B が型 A と同等以上に cv 修飾されている。
- A が仮想基底クラスでなく、B の仮想基底クラスの基底クラスでもない。

以下の条件が満たされている場合は、型が cv1 T である A のメンバーへのポインターの C++11 (prvalue) C++11 右辺値を、型が cv2 T である B のメンバーへのポインターの C++11 (prvalue) C++11 右辺値にキャストできます。

- B は、A の基底クラスである。
- 型が T である B のメンバーを指すポインターから、型が T である A のメンバーを指すポインターへの標準の変換が存在する。
- cv2 が cv1 と同等以上の cv 修飾である。

cv2 が cv1 と同等以上の cv 修飾である場合は、cv1 void を指すポインターを、cv2 void を指すポインターに明示的に変換できます。

関連資料:

427 ページの『ユーザー定義の変換』

118 ページの『型ベースの別名割り当て』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

reinterpret_cast 演算子 (C++ のみ)

reinterpret_cast 演算子は、無関係の型の間の変換を扱います。

reinterpret_cast 演算子の構文

C++11 右不等号括弧の機能により、2 つの連続する > トークンの代わりに >> トークンを使用して、template_id を reinterpret_cast 演算子の Type として指定することができます。詳しくは、447 ページの『クラス・テンプレート』を参照してください。 C++11

reinterpret_cast<Type>(expression) の結果は、以下のいずれかの値のカテゴリーに属します。

- Type が左辺値参照型である場合 C++11 または関数型に対する右辺値参照である場合 C++11 、reinterpret_cast<Type>(expression) は左辺値です。
- C++11 Type がオブジェクト型に対する右辺値参照である場合、reinterpret_cast<Type>(expression) は xvalue です。 C++11
- それ以外すべての場合は、reinterpret_cast<Type>(expression) は、 C++11 (prvalue) C++11 右辺値です。

reinterpret_cast 演算子は、引数と同じビット・パターンをもっている、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、両方とも関数型か両方ともオブジェクト型である場合。

NULL ポインター値は、宛先型の NULL ポインター値へ変換されます。

型 T と左辺値式 x がある場合に、左辺値参照に関する以下の 2 つの式は構文が異なりますが、意味は同じです。

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

C++11 型 T と左辺値式 x がある場合に、右辺値参照に関する以下の 2 つの式は構文が異なりますが、意味は同じです。

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

C++11

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再解釈は、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・キャストよりもはっきりとわかりやすく、明示的キャストが可能な強い型宣言を持つ言語の中で矛盾した部分が強調されます。

C++ コンパイラーは、全部ではないがほとんどの違反を検出し、修正します。プログラムがコンパイルされても、そのソース・コードが、完全には正しくない場合があるということを覚えておくことは重要です。プラットフォームによっては、パフォーマンスの最適化は、標準の別名割り当て規則に厳密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ますが、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、期待どおりに実行されます。また、C++ でも最適化して正常にコンパイルできますが、必ずしも期待どおりには実行されません。問題の 7 行目は、x の古いまたは未初期化の値を印刷してしまいます。

```
1  extern int y = 7.;
2
3  int main() {
4       float x;
5       int i;
6       x = y;
7       i = *(int *) &x;
8       printf("i=%d. x=%f.¥n", i, x);
9  }
```

次のコードの例は、キャストが 2 つの異なるファイルにまたがっているので、コンパイラーが検出すらもできない、誤ったキャストを含んでいます。

```
1  /* separately compiled file 1 */
2      extern float f;
3      extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */
4
5  /* separately compiled file 2 */
6      extern float f;
7      extern int * int_pointer_to_f;
8      f = 1.0;
9      int i = *int_pointer_to_f;          /* no suspicious cast but wrong */
```

8 行目において、int i = *int_pointer_to_f がロード元としている同じオブジェクトを、 f = 1.0 が保管先としていることを、コンパイラーが知る方法はありません。

関連資料:

427 ページの『ユーザー定義の変換』

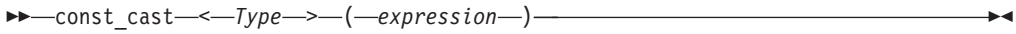
165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

const_cast 演算子 (C++ のみ)

const_cast 演算子 は、型に対する const 修飾子または volatile 修飾子の追加や除去を行います。

const_cast 演算子の構文



C++11 右不等号括弧の機能により、2 つの連続する > トークンの代わりに >> トークンを使用して、template_id を const_cast 演算子の Type として指定することができます。詳しくは、447 ページの『クラス・テンプレート』を参照してください。 C++11

const_cast<Type>(expression) の結果は、以下のいずれかの値のカテゴリーに属します。

- Type がオブジェクト型に対する左辺値参照である場合、const_cast<Type>(expression) は左辺値です。
- C++11 Type がオブジェクト型に対する右辺値参照である場合、const_cast<Type>(expression) は xvalue です。 C++11
- それ以外のすべての場合、const_cast<Type>(expression) は、C++11 (prvalue) C++11 右辺値です。

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<T2> を使用して T1 を指すポインターを T2 を指すポインターに変換できる場合 (T1 および T2 はオブジェクト型とします)、以下の種類の変換も可能です。

- 型 T1 の左辺値から型 T2 の左辺値。const_cast<T2&> を使用します
- C++11 型 T1 の左辺値または xvalue から型 T2 の xvalue。const_cast<T2&&> を使用します。 C++11
- C++11 クラス型 T1 の prvalue から型 T2 の xvalue。const_cast<T2&&> を使用します。 C++11

T1 へのポインター型の C++11 (prvalue) C++11 右辺値を T2 へのポインター型に変換すると定数性がキャストされる場合、以下の種類の変換でも定数性がキャストされます。

- 型 T1 の左辺値から型 T2 の左辺値
- C++11 型 T1 の式から型 T2 の xvalue C++11
- C++11 (prvalue) C++11 型 T1 のデータ・メンバー X へのポインター型の右辺値から型 T2 のデータ・メンバー Y のポインター型

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。その結果、コンパイラーは、関数呼び出し `f(c)` を許可します。

コンパイラーは、割り当て `*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動作は未定義です。`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。

関連資料:

101 ページの『型修飾子』

118 ページの『型ベースの別名割り当て』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

dynamic_cast 演算子 (C++ のみ)

dynamic_cast 演算子は、以下の種類の変換を実行時に検査します。

- 基底クラスを指すポインターから、派生クラスを指すポインター

- 基底クラスを参照する左辺値から、派生クラスを参照する左辺値
- C++11 基底クラスを参照する xvalue から、派生クラスを参照する右辺値 C++11

それによって、プログラムはクラス階層を安全に使用することができます。この演算子と typeid 演算子は、C++ でのランタイム型情報 (RTTI) サポートを提供します。

dynamic_cast 演算子の構文

C++11 右不等号括弧の機能により、2 つの連続する > トークンの代わりに >> トークンを使用して、template_id を dynamic_cast 演算子の T として指定することができます。詳しくは、447 ページの『クラス・テンプレート』を参照してください。 C++11

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。

dynamic_cast<T>(v) という式には、以下の規則が適用されます。

- T がポインター型である場合は、v は、C++11 (prvalue) C++11 右辺値でなければならず、dynamic_cast<T>(v) は、C++11 (prvalue) C++11 型 T の右辺値になります。
- T が左辺値参照型である場合は、v が左辺値でなければならず、dynamic_cast<T>(v) は T によって参照される型の左辺値になります。
- C++11 T が右辺値参照型である場合、dynamic_cast<T>(v) は T によって参照される型の xvalue になります。 C++11

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 NULL 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、ランタイム型情報 (RTTI) が生成される必要があります。その情報は、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しなければなりません。

T が void ポインターの場合、dynamic_cast は、v が指すオブジェクトの開始アドレスを戻します。次の例は、このことを示しています。

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

struct A {
  virtual ~A() { };
};

struct B : A { };

int main() {
  B bobj;
```

```
  A* ap = &bobj;
  void * vp = dynamic_cast<void *>(ap);
  cout << "Address of vp  : " << vp << endl;
  cout << "Address of bobj: " << &bobj << endl;
}
```

この例の場合、出力結果は次のようになります。 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、型が異なっても支障のないダウンキャスト を実行することです。ダウンキャストとは、クラス A が B の基底クラスである場合に、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することです。ダウンキャストの問題は、型 A* のポインターが、型 B のオブジェクトに属する型 A の基底クラス・サブオブジェクトでも、B から派生したクラスでもないオブジェクトを指している可能性があることです。dynamic_cast 演算子を使用すると、クラス A を指すポインターをクラス B を指すポインターに変換する場合に、前者が指す型 A のオブジェクトが、型 B のオブジェクトまたは基底クラス・サブオブジェクトとして B から派生したクラスのオブジェクトに属することが保証されます。

以下の例は、dynamic_cast 演算子の使用法を示したものです。

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

struct A {
  virtual void f() { cout << "Class A" << endl; }
};

struct B : A {
  virtual void f() { cout << "Class B" << endl; }
};

struct C : A {
  virtual void f() { cout << "Class C" << endl; }
};

void f(A* arg) {
  B* bp = dynamic_cast<B*>(arg);
  C* cp = dynamic_cast<C*>(arg);

  if (bp)
    bp->f();
  else if (cp)
    cp->f();
  else
    arg->f();
};

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

上記の例の出力は、次のとおりです。

```
Class C
Class A
```

関数 f() は、ポインター arg が、型 A、B、または C のオブジェクトを指すかどうかを判別します。関数は、dynamic_cast 演算子を使用して、arg を、型 B のポインターへ、次に型 C のポインターへ変換しようと試みることによって、この判別を行います。 dynamic_cast 演算子が正常に行われると、arg によって表されるオブジェクトを指すポインターを戻します。 dynamic_cast が失敗すると、0 が戻されます。

downcast は、ポリモアフィック・クラスにおいてのみ、dynamic_cast 演算子を使用して、実行することができます。上記の例では、クラス A は、仮想関数をもっているので、すべてのクラスはポリモアフィックです。 dynamic_cast 演算子は、ポリモアフィック・クラスから生成されたランタイム時の型情報を使用します。

関連資料:

383 ページの『派生』

427 ページの『ユーザー定義の変換』

118 ページの『型ベースの別名割り当て』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

複合リテラル式

複合リテラル とは、値が初期化指定子リストによって与えられる名前なしオブジェクトを提供する後置式です。 C99 言語機能では、一時変数を必要とせずに、パラメーターを関数に受け渡すことができます。これは、集合体型 (配列、構造体、および共用体) のインスタンスが 1 つだけ必要な場合に、その型の定数を指定するのに便利です。

複合リテラルの構文は cast 式の構文に似ています。ただし、複合リテラルは左辺値ですが、cast 式の結果はそうではありません。さらに、キャストはスカラー型または void にしか変換できませんが、複合リテラルは指定された型のオブジェクトになります。

複合リテラルの構文

type_name は、ベクトル型およびユーザー定義の型など任意のデータ型にすることができます。サイズが不明な配列にすることができますが、可変長配列にすることはできません。型が不明サイズの配列の場合、サイズは、初期化指定子リストによって決まります。

次の例では、2 つの整数メンバーを含む point 型の定数構造体変数を、関数 drawline に受け渡します。

```
drawline((struct point){6,7});
```

複合リテラルが関数の本体外で発生する場合、初期化指定子リストは定数式で構成される必要があり、名前なしのオブジェクトには静的ストレージ期間が指定されます。複合リテラルが関数の本体内で発生する場合、初期化指定子リストは定数式で構成される必要はなく、名前なしのオブジェクトには自動ストレージ期間が指定されます。

IBM GNU C との互換性を確保するために、初期化指定子リストのすべての初期化指定子が定数式であれば、同じ型の複合リテラルを使用して静的変数を初期化することができます。

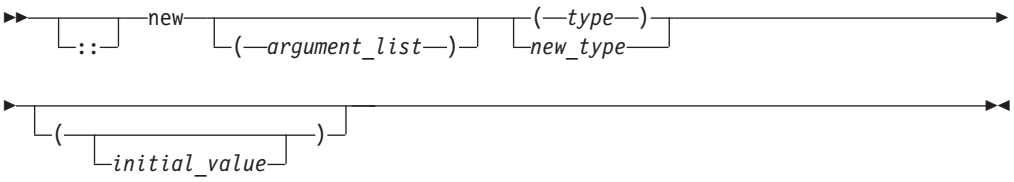
関連資料:

ストリング・リテラル

new 式 (C++ のみ)

new 演算子は、動的ストレージ割り振りを提供します。

new 演算子の構文



スコープ解決演算子 (::) に new を接頭部として付けると、グローバルな 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戻し、これを使用してオブジェクトを初期化することができます。オブジェクトが配列の場合は、最初のエレメントを指すポインターが戻されます。

関数型、void、または不完全クラス型はオブジェクトの型ではないので、new 演算子を使用してこれらの型を割り振ることはできません。ただし、new 演算子を使用して、関数を指すポインターを割り振ることはできます。new 演算子を使用して、参照を作成することはできません。

作成されるオブジェクトが配列の場合は、最初の次元だけが汎用式になります。以降のすべての次元は、正の値に評価される、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を指す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 オブジェクトまたは volatile オブジェクトを指すポインターは含めることができます。

例えば、const char* は使用できますが、char* const は使用できません。

関連資料:

一般化された定数式 (C++11)

配置構文

追加引数は、*argument_list* を使用することにより、 new に追加することができます (*配置構文* とも呼ばれます)。配置引数を使う場合は、それらの引数が指定された operator new() または operator new[]() が宣言されていなければなりません。次に例を示します。

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

class X
{
public:
      void* operator new(size_t,int, int){ /* ... */ }
};

// ...
```

```
int main ()
{
      X* ptr = new(1,2) X;
}
```

配置構文は、グローバル配置 new 関数を呼び出すために広く使われています。グローバル配置 new 関数は、配置 new 式の中で配置引数によって指定されたロケーションにあるオブジェクト (1 つ以上) を初期化します。グローバル配置 new 関数はそれ自体ではメモリーを割り振らないので、このロケーションは、以前に他の方法で割り振られたストレージを指定していなければなりません。以下の例では、呼び出し new(whole) X(8);、new(seg2) X(9);、または new(seg3) X(10); によって新しいメモリーは割り振られません。そうではなく、コンストラクター X(8)、X(9)、および X(10) が呼び出されて、バッファー whole に割り振られたメモリーが再初期化されます。

配置 new はメモリーを割り振らないので、配置構文で作成されたオブジェクトを割り振り解除するために delete を使用しないでください。削除できるのは、メモリー・プール全体だけです (delete whole)。次の例では、メモリー・バッファーを残しておいて、デストラクターを明示的に呼び出すことにより、そこに保管されていたオブジェクトを破棄することができます。

```
#include <new>
class X
{
   public:
      X(int n): id(n){ }
      ~X(){ }
   private:
      int id;
      // ...
};

int main()
{
   char* whole = new char[ 3 * sizeof(X) ];  // a 3-part buffer
   X * p1 = new(whole) X(8);                 // fill the front
   char* seg2 = &whole[ sizeof(X) ];         // mark second segment
   X * p2 = new(seg2) X(9);                  // fill second segment
   char* seg3 = &whole[ 2 * sizeof(X) ];     // mark third segment
   X * p3 = new(seg3) X(10);                 // fill third segment

   p2->~X();   // clear only middle segment, but keep the buffer
   // ...
   return 0;
}
```

配置 new 構文は、コンストラクターではなく、割り振りルーチンにパラメーターを渡すためにも使用できます。

関連資料:

222 ページの『delete 式 (C++ のみ)』

173 ページの『スコープ解決演算子 :: (C++ のみ)』

409 ページの『コンストラクターとデストラクターの概要』

new 演算子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を、複数の方法で初期化することができます。非クラス・オブジェクト、またはコンストラクターが指定されていない

クラス・オブジェクトの場合、*new 初期化指定子* の式は、new 式に (式) または () を使用して指定することができます。次に例を示します。

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

クラスにデフォルトのコンストラクターが指定されていない場合は、そのクラスのオブジェクトが割り振られる際に new 初期化指定子を指定する必要があります。new 初期化指定子の引数は、コンストラクターの引数と一致していなければなりません。

配列に初期化指定子を指定することはできません。クラスにデフォルトのコンストラクターが指定されている場合のみ、クラス・オブジェクトの配列を初期化することができます。コンストラクターは、各配列エレメント (クラス・オブジェクト) を初期化するために呼び出されます。

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。

関連資料:

409 ページの『コンストラクターとデストラクターの概要』

新規割り振り失敗の処理

new 演算子が新しいオブジェクトを作成する際は、必要なストレージを得るために operator new() 関数または operator new[]() 関数を呼び出します。

new が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ることができない場合は、set_new_handler() への呼び出しによって、インストール済みであれば *new handler* 関数を呼び出します。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 handler、またはデフォルトの new handler を呼び出します。

new handler は、次のいずれかを行います。

- メモリー割り振りのためにさらにストレージを取得し、それから戻ります。
- 型 std::bad_alloc の例外、または std::bad_alloc から派生したクラスをスローします。
- abort() または exit() のどちらかを呼び出します。

set_new_handler() 関数はプロトタイプを持ちます。

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

set_new_handler() は、引数として関数 (new handler) を指すポインターを取りますが、この関数は引数がなく void を戻します。そして、前の new handler 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、new は型 std::bad_alloc の例外をスローします。

次のプログラム・フラグメントでは、new 演算子がストレージを割り振れない場合に、どのように set_new_handler() を使用してメッセージを戻すことができるかを示しています。

```
#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ることができないためにプログラムが失敗した場合は、次のメッセージを表示してプログラムは終了します。

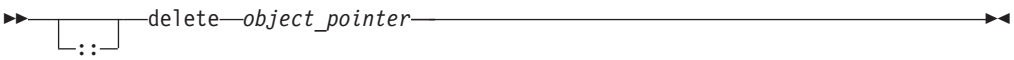
```
 Operator new failed:
no storage is available.
```

delete 式 (C++ のみ)

delete 演算子は、オブジェクトに関連付けられたメモリーを割り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 戻りの型があります。

delete 演算子の構文



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。NULL 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。

delete[] 演算子の構文

delete によって配列オブジェクトを削除した結果は、未定義です。 delete[] によって個々のオブジェクトを削除する場合も同じです。配列の次元は、delete[] で指定する必要はありません。

削除されたオブジェクトまたは配列へアクセスしようとする試みの結果は、未定義です。

デストラクターがクラスに定義されている場合は、delete によってそのデストラクターが呼び出されます。デストラクターがあるかどうかに関係なく、delete は、クラスの関数 operator delete() がある場合は、この関数呼び出しによって指し示されたストレージを解放します。

次の場合には、グローバル ::operator delete() が使用されます。

- クラスに operator delete() がない場合
- オブジェクトがクラス以外の型の場合
- ::delete 式によってオブジェクトが削除される場合

次の場合には、グローバル ::operator delete[]() が使用されます。

- クラスに operator delete[]() がない場合
- オブジェクトがクラス以外の型の場合
- ::delete[] 式によってオブジェクトが削除される場合

デフォルトのグローバル operator delete() だけが、デフォルトのグローバル operator new() によって割り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。

関連資料:

67 ページの『void 型』

409 ページの『コンストラクターとデストラクターの概要』

throw 式 (C++ のみ)

throw 式は、C++ の例外ハンドラーに例外をスロー (throw) するために使われます。 throw 式は、void 型です。

関連資料:

493 ページの『第 16 章 例外処理 (C++ のみ)』

67 ページの『void 型』

ラベル値演算子 (IBM 拡張)

ラベル値演算子 && は、オペランドのアドレスを戻します。オペランドは、現行関数またはそれを包含する関数で定義されているラベルでなければなりません。値は void* 型の定数であり、computed goto (計算型 goto) ステートメントの中でのみ使用すべきものです。この言語機能は C および C++ の拡張機能の 1 つであり、GNU 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ことを目的としてインプリメントされています。

関連資料:

232 ページの『値としてのラベル (IBM 拡張)』

計算型 goto ステートメント

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もつ式が、最初に評価されます。小括弧を使用することによって、強制的にオペランドをグループ化することができます。

例えば、次のステートメントでは、値 5 は、= 演算子の右から左への結合順序を使用して、a と b の両方に割り当てられます。最初に値 c が b に割り当てられ、次に値 b が a に割り当てられます。

```
b = 9;
c = 5;
a = b = c;
```

副次式の評価順序が指定されていないので、小括弧を使用して、演算子付きのオペランドのグループ化を明示的に強制することができます。

次の式では、

```
a + b * c / d
```

優先順位により、+ 演算の前に、* および / 演算が実行されます。結合順序により、b は d によって除算される前に、c と乗算されます。

次の表に、C および C++ 言語の演算子を優先順位の順序でリストし、各演算子の結合順序の方向を示します。同位にある演算子には、同じ優先順位が付けられています。

表 34. 後置演算子の優先順位と結合順序

ランク	右結合か	演算子関数	使用法
1	はい	C++ グローバル・スコープ解決	:: *name_or_qualified* *name*
1		C++ クラスまたは名前空間スコープ解決	*class_or_namespace*::*member*
2		メンバー選択	*object* . *member*
2		メンバー選択	*pointer* -> *member*
2		添え字	*pointer* [*expr*]
2		関数呼び出し	*expr* (*expr_list*)
2		値生成	*type* (*expr_list*)
2		後置増分	*lvalue* ++
2		後置減分	*lvalue* --
2	はい	C++ 型識別	typeid (*type*)
2	はい	C++ 実行時の型識別	typeid (*expr*)
2	はい	C++ コンパイル時の変換チェック	static_cast < *type* > (*expr*)

表 34. 後置演算子の優先順位と結合順序 (続き)

ランク	右結合か	演算子関数	使用法
2	はい	C++ 実行時の変換チェック	dynamic_cast < *type* > (*expr*)
2	はい	C++ チェックされない変換	reinterpret_cast < *type* > (*expr*)
2	はい	C++ const 変換	const_cast < *type* > (*expr*)

表 35. 単項演算子の優先順位と結合順序

ランク	右結合か	演算子関数	使用法
3	はい	オブジェクトのサイズ (バイト)	sizeof *expr*
3	はい	型のサイズ (バイト)	sizeof (*type*)
3	はい	前置増分	++ *lvalue*
3	はい	前置減分	-- *lvalue*
3	はい	ビット単位否定	~ *expr*
3	はい	not	! *expr*
3	はい	単項減算	- *expr*
3	はい	単項正	+ *expr*
3	はい	のアドレス	& *lvalue*
3	はい	間接指示または間接参照	* *expr*
3	はい	C++ 作成 (メモリーの割り振り)	new *type*
3	はい	C++ 作成 (メモリーの割り振りと初期化)	new *type* (*expr_list*) *type*
3	はい	C++ 作成 (配置)	new *type* (*expr_list*) *type* (*expr_list*)
3	はい	C++ 破棄 (メモリーの割り振り解除)	delete *pointer*
3	はい	C++ 配列の破棄	delete [ ] *pointer*
3	はい	型変換 (キャスト)	(*type*) *expr*

表 36. 2 項演算子の優先順位と結合順序

ランク	右結合か	演算子関数	使用法
4		C++ メンバー選択	*object*.**ptr_to_member*
4		C++ メンバー選択	*object* ->* *ptr_to_member*
5		乗算	*expr* * *expr*
5		除算	*expr* / *expr*
5		モジュロ (剰余)	*expr* % *expr*
6		2 項加算	*expr* + *expr*

表 36. 2 項演算子の優先順位と結合順序 (続き)

ランク	右結合か	演算子関数	使用法
6		2 項減算	*expr - expr*
7		左へのビット単位シフト	*expr << expr*
7		右へのビット単位シフト	*expr >> expr*
8		より小さい	*expr < expr*
8		より小さいまたは等しい	*expr <= expr*
8		より大きい	*expr > expr*
8		より大きいまたは等しい	*expr >= expr*
9		等しい	*expr == expr*
9		等しくない	*expr != expr*
10		ビット単位 AND	*expr & expr*
11		ビット単位排他 OR	*expr ^ expr*
12		ビット単位包含 OR	*expr | expr*
13		論理 AND	*expr && expr*
14		論理包含 OR	*expr || expr*
15		条件式	*expr ? expr : expr*
16	はい	単純割り当て	*lvalue=expr*
16	はい	乗算および割り当て	*lvalue *= expr*
16	はい	除算および割り当て	*lvalue /= expr*
16	はい	モジュロおよび割り当て	*lvalue %= expr*
16	はい	加算および割り当て	*lvalue += expr*
16	はい	減算および割り当て	*lvalue-=expr*
16	はい	左へのシフトおよび割り当て	*lvalue <<=expr*
16	はい	右へのシフトおよび割り当て	*lvalue >>=expr*
16	はい	ビット単位 AND および割り当て	*lvalue &=expr*
16	はい	ビット単位排他 OR および割り当て	*lvalue ^= expr*
16	はい	ビット単位包含 OR および割り当て	*lvalue|=expr*
17	はい	C++ 例外のスロー	throw *expr*
18		コンマ (順序付け)	*expr , expr*

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、小括弧で示されている次のような方法が可能です。

```
total = (price + (prov_tax + city_tax));
total = ((price + prov_tax) + city_tax);
total = ((price + city_tax) + prov_tax);
```

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。

中間値が丸められるため、浮動小数点演算子の異なるグループ化は、異なる結果になる場合があります。

ある式では、オペランドや演算子のグループ化によっては、結果に影響を与えます。例えば、次の式では、各関数呼び出しは同じグローバル変数を変更します。

```
a = b() + c() + d();
```

この式は、関数が呼び出される順序によって、異なる結果になります。

式に結合属性と可換属性の両方がある演算子が含まれており、オペランドを演算子にグループ化する順序が式の結果に影響を与える場合は、式をいくつかの式に区切ります。例えば、呼び出し先関数が変数 `a` に影響を与えるような副次作用を作成しない場合は、次の式によって、前の例の式を置換できます。

```
a = b();
a += c();
a += d();
```

関数呼び出しの引数または 2 項演算子のオペランドの評価順序は、指定されていません。したがって、次の式はあいまいです。

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

最初のステートメントの前に、`y` に値 1 が指定されている場合は、値 1 または値 2 が `func1()` に渡されるかどうかは不明です。2 番目のステートメントでは、式が評価される前に `i` の値が 1 である場合は、`x[1]` または `x[2]` が 2 番目の引数として `func2()` に渡されるかどうかは不明です。

参照の縮約 (reference collapsing) (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11

機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 以前の C++ 言語では、参照の参照は不適格です。 C++11 では、以下のいずれかのコンテキストを通じて参照の参照を使用する場合は、参照の縮約 (reference collapsing) の規則が適用されます。

- decltype 指定子
- typedef 名
- テンプレート型パラメーター

変数の宣言型 TR が型 T への参照であり、かつ T も参照型である変数 var を定義できます。次に例を示します。

```
// T denotes the int& type
typedef int& T;

// TR is an lvalue reference to T
typedef T& TR;

// The declared type of var is TR
TR var;
```

var の実際の型を次の表にケースごとに示します。ここで、TR と T はどちらも cv 修飾子で修飾されません。

表 37. cv 修飾子がない場合の参照の縮約 (reference collapsing)

T	**TR**	**var の型**
A	T	A[1]
A	T&	A&[1]
A	T&&	A&&[1]
A&	T	A&[1]
A&	T&	A&
A&	T&&	A&
A&&	T	A&&[1]
A&&	T&	A&
A&&	T&&	A&&

注:

1. T と TR の両方が参照型というわけではないため、この場合、参照の縮約 (reference collapsing) は適用されません。

この表の原則として、T と TR は両方とも参照型であるが、両方が右辺値参照型というわけではない場合、var は左辺値参照型になります。

例 1

```
typedef int& T;

// a has the type int&
T&& a;
```

この例で、T は int& 型であり、a の宣言型は T&& です。参照の縮約 (reference collapsing) 後、a の型は int& になります。

例 2

```
template <typename T> void func(T&& a);
auto fp = func<int&&>;
```

この例で、T の実パラメーターは int&& 型であり、a の宣言型は T&& です。右辺値参照の右辺値参照が形成されます。参照の縮約 (reference collapsing) 後、a の型は int&& になります。

例 3

```
auto func(int& a) -> const decltype(a)&;
```

この例で、decltype(a) は、後置戻り型であり、パラメーター a を参照します。このパラメーター a の型は int& です。参照の縮約 (reference collapsing) 後、func の戻りの型は int& になります。

変数の宣言型 TR が型 T への参照であり、かつ T も参照型である変数 var を定義できます。TR または T のいずれかが cv 修飾子で修飾されている場合について、var の実際の型を次の表にケースごとに示します。

表 38. cv 修飾子がある場合の参照の縮約 (reference collapsing)

T	TR	var の型
A	const T	const A[1]
const A	volatile T&	const volatile A&[1]
A	const T&&	const A&&[1]
A&	const T	A&[1]
const A&	volatile T&	const A&
const A&	T&&	const A&
A&&	const T	A&&[1]
const A&&	volatile T&	const A&
const A&&	T&&	const A&&

注:

1. T と TR の両方が参照型というわけではないため、この場合、参照の縮約 (reference collapsing) は適用されません。

この表での原則として、T が参照型の場合、var の型は T から cv 修飾子のみを継承します。

関連資料:

「XL C/C++ 最適化およびプログラミング・ガイド」の『右辺値参照の使用 (C++11)』を参照

93 ページの『decltype(*expression*) 型指定子 (C++11)』

89 ページの『typedef 定義』

440 ページの『「型」テンプレート・パラメーター』

第 7 章 ステートメント

ステートメントは最小の独立した計算単位であって、実行するアクションを指定します。ほとんどの場合、ステートメントは順々に実行されます。以下のリストに、C および C++ で使用可能なステートメントの要約を示します。

- ラベル付きステートメント
- 式ステートメント
- ブロック・ステートメント
- 選択ステートメント
- 繰り返しステートメント
- 分岐ステートメント
- 宣言ステートメント
- C++ try ブロック
- NULL ステートメント
- IBM インライン・アセンブリー・ステートメント (C のみ) (IBM 拡張)

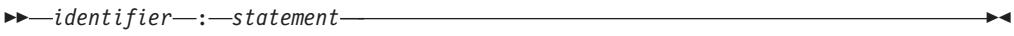
関連資料:

47 ページの『第 3 章 データ・オブジェクトとデータ宣言』
264 ページの『関数宣言』
493 ページの『try ブロック』

ラベル付きステートメント

ラベルには identifier、case、および default の 3 つの種類があります。

ラベル付きステートメントの構文



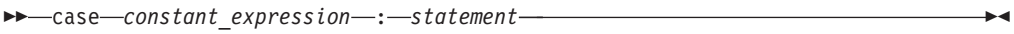
ラベルは、*identifier* およびコロン (:) 文字から構成されます。

C ラベル名は、それが現れる関数内で固有でなければなりません。

C++ C++ では、identifier ラベルは、goto ステートメントのターゲットとしてのみ使用できます。goto ステートメントは、ラベルの定義の前にラベルを使用できます。 identifier ラベルは、それ自体の名前空間を持っています。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、1 つの関数内で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 ステートメントでのみ使用されます。これらのラベルは、最も近くの、括弧で囲まれた switch ステートメント内でのみアクセス可能です。

case ステートメントの構文



default ステートメントの構文

ラベルの例を次に示します。

```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```

関連資料:

250 ページの『goto ステートメント』

238 ページの『switch ステートメント』

ローカルに宣言されたラベル (IBM 拡張)

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る identifier ラベルであって、そのためのスコープは、そのラベルが宣言されて定義されたステートメント式です。この言語機能は C および C++ の拡張機能であって、GNU C で開発されたプログラムの処理を容易にするためのものです。

ローカル・ラベルは、goto ステートメントのターゲットとして使用することができ、ラベルが宣言されたブロック内から、そのラベルにジャンプすることができます。この言語拡張機能は、ネストされたループを含むマクロを書く場合に特に有用で、ローカル・ラベルのステートメント・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することができます。

ローカルに宣言されるラベルの構文

ローカル・ラベルの宣言は、あらゆる通常の宣言およびステートメントよりも前になければなりません。ラベル宣言はラベル名のみを定義するので、ラベルそのものは、名前とコロンを使用し、ステートメント式のステートメント内で、通常の方法で定義する必要があります。

関連資料:

235 ページの『ステートメント式 (IBM 拡張)』

値としてのラベル (IBM 拡張)

現行関数またはそれを包含する関数の中で定義されているラベルのアドレスを取得して、void* 型の定数が有効な場所であればどこでも、そのアドレスを値として使用することができます。ラベルが単項演算子 && のオペランドのときは、「アドレス」がその戻り値です。ラベルのアドレスを値として使用できる機能は、C99 およ

び C++ の拡張機能の 1 つで、GNU 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。

C++11

注: 単項演算子 && は右辺値参照の修飾子としても使用できます。右辺値参照について詳しくは、 126 ページの『参照 (C++ のみ)』を参照してください。

C++11

以下の例では、computed goto (計算型 goto) ステートメントにより、label1 および label2 の値が、関数内のこれらのラベル位置にジャンプするために使用されています。

```
int main()
{
   void * ptr1, *ptr2;
   ...
   label1: ...
   ...
   label2: ...
   ...
   ptr1 = &&label1;
   ptr2 = &&label2;
   if (...) {
      goto *ptr1;
   } else {
      goto *ptr2;
   }
   ...
}
```

関連資料:

計算型 goto ステートメント

126 ページの『参照 (C++ のみ)』

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、NULL でもかまいません。

式ステートメントの構文

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、NULL ステートメントです。

ステートメントの例は、以下のとおりです。

```
printf("Account Number: ¥n");          /* call to the printf    */
marks = dollars * exch_rate;                /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

関連資料:

165 ページの『第 6 章 式と演算子』

あいまいなステートメントの解決 (C++ のみ)

C++ 構文は、式ステートメントと宣言ステートメントの間のあいまいさを明確化していません。式ステートメントの左端の副次式に関数スタイル・キャストがあると、あいまいさが生じます。 (C では、関数スタイルのキャストをサポートしないため、C プログラムではこのようなあいまいさは起きません。) ステートメントを宣言または式のいずれにも解釈できる場合、ステートメントは宣言ステートメントとして解釈されます。

注: あいまいさは、構文レベルでのみ解決されます。明確化に際して、名前の意味が型名であるかどうかを評価する場合を除いて、名前の意味を使用しません。

以下の式は、あいまいな副次式のあとに、割り当て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
```

関連資料:

47 ページの『第 3 章 データ・オブジェクトとデータ宣言』

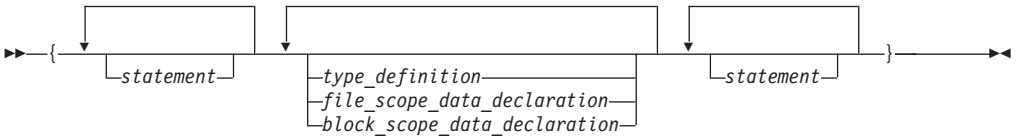
165 ページの『第 6 章 式と演算子』

175 ページの『関数呼び出し式』

ブロック・ステートメント

ブロック・ステートメント または *複合ステートメント* を使用すると、任意の数のデータ定義、宣言、およびステートメントを 1 つのステートメントにグループ化します。 1 組の中括弧に囲まれた定義、宣言、およびステートメントはすべて、単一のステートメントとして扱います。単一のステートメントが使用可能な場所ならばどこでもブロックを使用することができます。

ブロック・ステートメントの構文



ブロックは、ローカル・スコープを定義します。データ・オブジェクトがブロック内で使用可能であり、その ID が再定義されていない場合には、ネストされたすべてのブロックが、そのデータ・オブジェクトを使用できます。

ブロックの例

以下のプログラムは、ネストされたブロック内でデータ・オブジェクトの値がどのように変更されるかを示しています。

```
/**
 ** This example shows how data objects change in nested blocks.
 **/
  #include <stdio.h>

  int main(void)
  {
     int x = 1;                    /* Initialize x to 1  */
     int y = 3;

     if (y > 0)
     {
        int x = 2;                 /* Initialize x to 2  */
        printf("second x = %4d¥n", x);
     }
     printf("first  x = %4d¥n", x);

     return(0);
  }
```

プログラムは、以下の出力を作成します。

```
second x =    2
first  x =    1
```

x という名前の 2 つの変数が main の中で定義されています。 x の最初の定義は、main が実行されている間、ストレージを保存します。しかし、x の 2 番目の定義 (再定義) がネストされたブロック内で発生するため、 printf("second x = %4d¥n", x); は、x を前の行で定義された変数であると認識します。 printf("first x = %4d¥n", x); は、ネストされたブロックの一部ではないので、x は、x の最初の定義として認識されます。

ステートメント式 (IBM 拡張)

複合ステートメントは、中括弧で囲まれた複数のステートメントのシーケンスです。 GNU C では、小括弧内の複合ステートメントは、いわゆる、ステートメント式 の中に式として使用することができます。

ステートメント式の構文

ステートメント式の値は、構成全体に現れる最後の単純式の値です。最後のステートメントが式でない場合、その構成は void 型であって、値はありません。

このステートメント式を typeof 演算子と組み合わせると、各オペランドが一度だけ評価される関数に似た複雑なマクロを作成することができます。次に例を示します。

```
#define SWAP(a,b) ( {__typeof__(a) temp; temp=a; a=b; b=temp;} )
```

選択ステートメント

選択ステートメントは、以下のステートメントで構成されます。

- if ステートメント
- switch ステートメント

if ステートメント

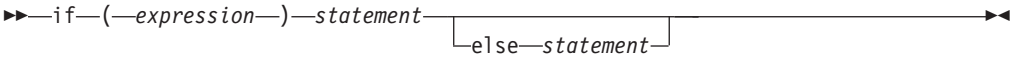
if ステートメントは、複数の制御のフローを可能にする選択ステートメントです。

C++ if ステートメントを使用すると、指定されたテスト式 (暗黙的に bool に変換されている) の評価が true 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。bool への暗黙的な変換が失敗した場合、プログラムは不適格です。

C C では、if ステートメントを使用すると、指定されたテスト式の評価が非ゼロ値になった場合に、ステートメントを条件付きで処理することができます。テスト式は、算術型またはポインター型でなければなりません。

オプションで、if ステートメントで else 文節を指定することができます。テスト式の評価が false (または C では、ゼロ値) になり、else 文節がある場合、else 文節に関連付けられているステートメントが実行されます。テスト式の評価が true になった場合、その式に続くステートメントが実行され、else 文節は無視されます。

if ステートメントの構文



if ステートメントがネストされていて、else 文節が存在する場合、指定された else は、同じブロック内の直前の if ステートメントに関連付けられます。

任意の選択ステートメント (if、switch) に続く単一のステートメントは、オリジナルのステートメントを含んでいる複合ステートメントとして扱われます。結果とし

て、そのステートメントで宣言されたすべての変数は、if ステートメントの後、スコープの外にあります。次に例を示します。

```
if (x)
int i;
```

は、以下と等価です。

```
if (x)
{  int i; }
```

変数 i は、if ステートメント内でのみ可視です。同じ規則が if ステートメントの else 部分にも適用されます。

if ステートメントの例

以下の例では、score の値が 90 以上である場合に、grade が A という値を受け取るようにします。

```
if (score >= 90)
   grade = 'A';
```

以下の例では、number の値が 0 またはそれ以上である場合に Number is positive と表示します。 number の値が 0 より小さい場合には、 Number is negative と表示します。

```
if (number >= 0)
   printf("Number is positive¥n");
else
   printf("Number is negative¥n");
```

以下の例は、ネストされた if ステートメントを示しています。

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
cout << "salary is " << salary << endl;
```

以下の例は、else 文節を持たない、ネストされた if ステートメントを示しています。 else 文節は、常に、最も近い if ステートメントに関連付けられるため、特定の else 文節を強制的に正しい if ステートメントに関連付けるには、中括弧が必要なことがあります。この例では、中括弧を省略すると、else 文節は、ネストされた if ステートメントに関連付けられることになります。

```
if (kegs > 0) {
   if (furlongs > kegs)
      fxph = furlongs/kegs;
}
else
   fxph = 0;
```

以下の例は、else 文節の中にネストされた if ステートメントを示しています。この例では、複数の条件がテストされます。テストは、それらの条件が書かれている順序で行われます。 1 つのテストがゼロ以外の値に評価されると、ステートメントが実行され、if ステートメント全体が終了します。

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

関連資料:

64 ページの『ブール型』

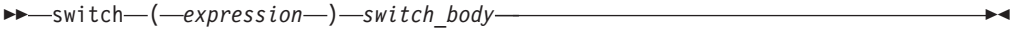
switch ステートメント

switch ステートメントは、 *switch* 式の値に基づいて、*switch* 本体の中の別のステートメントに制御を移すことができる選択ステートメントです。 switch 式の評価は、整数値または列挙値にならなければなりません。 switch ステートメントの本体には、以下のもので構成される *case* 文節 が含まれています。

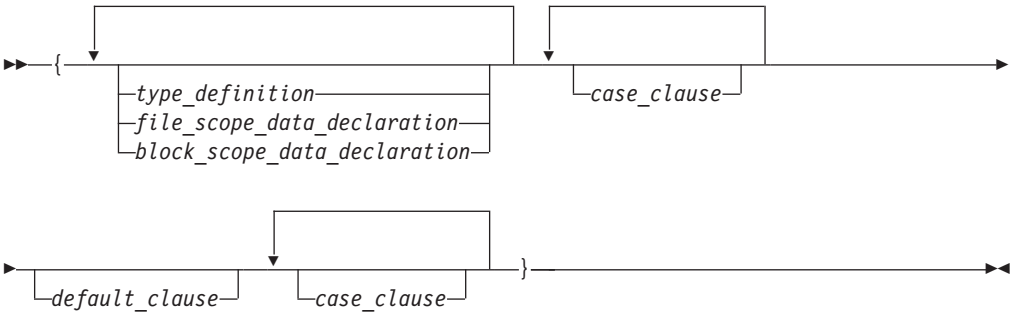
- case ラベル
- オプションの default ラベル
- case 式
- ステートメントのリスト

switch 式の値が case 式の 1 つの値と同じ場合、その case 式に続くステートメントが処理されます。そうでない場合、default ラベル・ステートメント (あれば) が処理されます。

switch ステートメントの構文



switch 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。



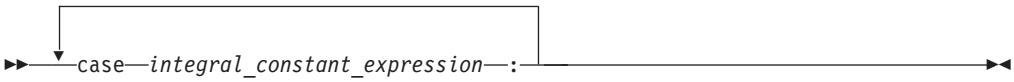
注: *type_definition*、*file_scope_data_declaration* または *block_scope_data_declaration* 内の初期化指定子は、無視されます。

case 文節 には、任意の数のステートメントが続く *case* ラベル が含まれます。 CASE 節の形式は、次のとおりです。

Case 文節の構文

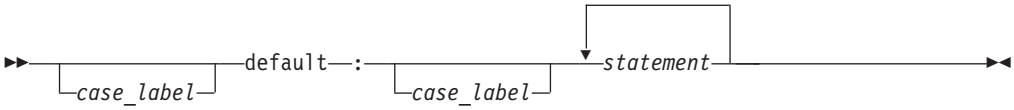
case ラベル には、case というワードと、それに続く整数定数式とコロンが入っています。各整数定数式の値は、別々の値を表していることが必要です。重複した case ラベルを持つことはできません。 1 つの case ラベルを置けるところであればどこでも、複数の case ラベルを置くことができます。 case ラベルの形式は、次のとおりです。

case ラベルの構文



default 文節 には、default ラベルと、それに続く 1 つ以上のステートメントが入っています。 case ラベルは、default ラベルのどちらの側にも置くことができます。 switch ステートメントに入れることができる default ラベルは、1 つだけです。 *default_clause* の形式は、次のとおりです。

default 文節のステートメント



switch ステートメントは、いずれか 1 つのラベルに続くステートメント、または switch 本体に続くステートメントに制御を渡します。switch 本体の前にある式の値によって、制御を受け取るステートメントが決まります。この式は *switch 式* と呼ばれます。

switch 式の値は、各 case ラベルの式の値と比較されます。一致している値が検出されれば、一致したその値が入っている case ラベルに続くステートメントに、制御が渡されます。一致する値はないが、switch 本体の中に default ラベルがある場合には、default ラベルの付いたステートメントに制御が渡されます。一致する値が見つからず、switch 本体の中のどこにも default ラベルがない場合には、switch 本体のどの部分も処理されません。

switch 本体の中のステートメントに制御が渡されると、 break ステートメントが検出されたとき、または switch 本体の中の最後のステートメントが処理されたときにのみ、制御は switch 本体を離れます。

必要であれば、整数拡張が、制御式上で実行されます。また、case ステートメント内のすべての式が、制御式と同じ型に変換されます。 switch 式は、整数型または列挙型への単一変換があれば、クラス型の式にもなります。

オプション **-qinfo=gen** を使用してコンパイルすると、失敗すべきでないときに失敗する case ラベルが検出されます。

switch ステートメントの制約事項

データ定義を switch 本体の先頭に置くことができますが、コンパイラーは、switch 本体の先頭にある auto 変数および register 変数を初期化しません。switch ステートメントの本体の中に、宣言を入れることができます。

switch ステートメントを使用して、初期化を飛び越えることはできません。

可変変更型の ID のスコープに switch ステートメントの case ラベルまたは default ラベルが含まれるときは、switch ステートメント全体がその ID のスコープ内にあると見なされます。すなわち、この ID の宣言は switch ステートメントの前になければなりません。

C++ C++ では、暗黙的または明示的な初期化指定子を含んでいる宣言を越えて、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迂回される内部ブロックに入っている場合は、制御権を移動することができます。初期化指定子が入っている switch ステートメント本体内の宣言はすべて、内部ブロックに入れる必要があります。

switch ステートメントの例

以下の switch ステートメントには、複数の case 文節と 1 つの default 文節が入っています。各文節には、関数呼び出しと break ステートメントが入っています。break ステートメントは、switch 本体内の各ステートメントに制御が移されていくのを防止します。

switch 式の評価が '/' となった場合、その switch ステートメントは関数 divide を呼び出すことになります。その後、switch 本体に続くステートメントに制御が渡されます。

```
char key;

printf("Enter an arithmetic operator¥n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;
```

```
   default:
      printf("invalid key¥n");
      break;
}
```

switch 式と case 式が一致した場合には、break ステートメントが検出されるまで、または switch 本体の終わりに達するまで、case 式に続くステートメントが処理されます。以下の例には、break ステートメントはありません。 text[i] の値が 'A' と等しい場合、 3 つのカウンターすべてが増分されます。 text[i] の値が 'a' と等しい場合には、lettera と total が増分されます。 text[i] が 'A' または 'a' と等しくない場合には、total のみが増分されます。

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

次の switch ステートメントは、複数の case ラベルに対して同じステートメントを実行します。

```
/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
 **/

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
     case 2:
        printf("month %d is a winter month¥n", month);
        break;

     case 3:
     case 4:
     case 5:
        printf("month %d is a spring month¥n", month);
        break;

     case 6:
     case 7:
```

```
    case 8:
       printf("month %d is a summer month¥n", month);
       break;

    case 9:
    case 10:
    case 11:
       printf("month %d is a fall month¥n", month);
       break;

    case 66:
    case 99:
    default:
       printf("month %d is not a valid month¥n", month);
  }

  return(0);
}
```

式 month の値が 3 の場合には、次のステートメントに制御が渡されます。

```
printf("month %d is a spring month¥n", month);
```

break ステートメントは、switch 本体に続くステートメントに制御を渡します。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qinfo=gen』を参照

case ラベルと default ラベル

247 ページの『break ステートメント』

174 ページの『一般化された定数式 (C++11)』

繰り返しステートメント

繰り返しステートメントは、以下のステートメントで構成されます。

- while ステートメント
- do ステートメント
- for ステートメント

関連資料:

64 ページの『ブール型』

while ステートメント

while ステートメント は、制御式の評価が false (または C では 0) になるまで、ループの本体を繰り返し実行します。

while ステートメントの構文

C *expression* は、算術型またはポインター型でなければなりません。

C++ 式は、bool に変換可能でなければなりません。

式は評価されて、ループの本体を処理するかどうかを判別します。式の評価が false の場合、ループの本体は実行されません。式が false に評価されないと、ループ本体は処理されます。本体が実行された後、制御は式に戻されます。それ以降の処理は、条件の値によって決まります。

break、return、または goto ステートメントがあると、条件の評価が false でない場合でも、while ステートメントを終了できます。

C++ throw 式がある場合も、条件が評価される前に while ステートメントが終了することがあります。

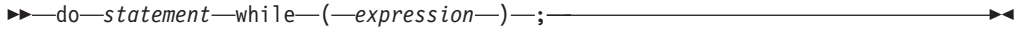
次の例では、式 ++index の値が MAX_INDEX より小さい間は、 item[index] は 3 倍され、印刷されます。 ++index の評価が MAX_INDEX になると、while ステートメントは終了します。

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d¥n", index, item[index]);
      ++index;
   }

   return(0);
}
```

do ステートメント

do ステートメント は、テスト式の評価が false (または C では 0) になるまで、ステートメントを繰り返し実行します。処理の順序のために、そのステートメントは少なくとも 1 回は実行されます。

do ステートメントの構文



C *expression* は、算術型またはポインター型でなければなりません。
C++ 制御する *expression* は、型 bool へ変換可能でなければなりません。

制御する while 文節が評価される前に、ループの本体が実行されます。それ以降の do ステートメントの処理は、while 文節の値によって決まります。 while 文節が false に評価されない場合には、再度ステートメントが実行されます。while 文節の評価が false になると、ステートメントは終了します。

break、return、または goto ステートメントは、while 文節が false に評価されなくても、do ステートメントの処理を終了させることができます。

C++ throw 式がある場合も、条件が評価される前に while ステートメントが終了することがあります。

次の例では、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;
}
```

上記の例の出力は、以下のとおりです。

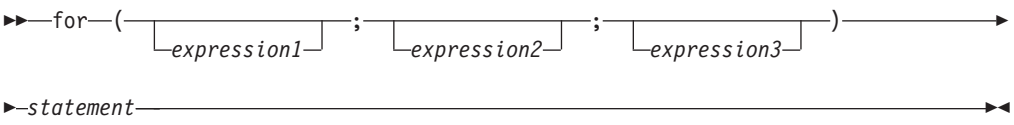
```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

for ステートメント

for statement を使用すると、以下の利点があります。

- ステートメントの最初の反復の前に式を評価する。(*初期化*)
- 式を指定して、ステートメントを処理するかどうかを判別する (*条件*)
- ステートメントが反復されるたびに、その後で式を評価する (反復のための増分によく使用される)
- 制御部分の評価が false (C では、0) にならない場合に、ステートメントを繰り返し処理する。

for ステートメントの構文



expression1 は*初期化式* です。これは、ステートメント が始めて処理される前においてのみ評価されます。この式を使用すると、変数を初期化することができます。この式を使用して変数を宣言することもできますが、その変数は、静的として宣言されていてはなりません (その変数は自動でなければならず、また、register としても宣言されていてかまいません)。この式で、またはステートメント の中のどこででも、変数を宣言すると、その変数は、for ループの終わりでスコープの外に出ます。ステートメントの最初の反復の前に式の評価を行いたくない場合には、この式を省略することができます。

expression2 は条件式 です。ステートメント の各反復の前に評価されます。

C *expression2* は算術型またはポインター型でなければなりません。

C C++ *expression2* は bool 型に変換可能でなければなりません。

C++

expression2 が C 0 C または C++ false C++ に評価される場合、ステートメントは処理されず、制御が for ステートメントに続く次のステートメントに移ります。 *expression2* の評価が false でない場合、ステートメントは処理されます。*expression2* を省略すると、この式が true によって、置き換えられたのと同様になり、この条件の不備により for ステートメントが終了しないことになります。

expression3 は、*statement* の毎回の反復後に評価されます。この式は、変数に対する増分、減分、または割り当てのために頻繁に使用されます。この式はオプションです。

break、return、または goto ステートメントがあると、 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* を省略する場合は、break、return、または goto ステートメントを使用して for ステートメントを終了する必要があります。

C++ 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、コンパイラー・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。

関連資料:


「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl=noansifor』 を参照

分岐ステートメント

分岐ステートメントは、以下のステートメントで構成されます。

- break ステートメント
- continue ステートメント
- return ステートメント
- goto ステートメント
- IBM 計算後の goto ステートメント (IBM 拡張)

break ステートメント

break ステートメント を使用すると、反復 (do、for、while) ステートメントまたは switch ステートメントを終了して、論理終了以外の任意のポイントで、ステートメントから出ることができます。 break は、これらのステートメントのいずれかだけに使用できます。

break ステートメントの構文

反復するステートメントにおいては、break ステートメントはループを終了して、ループの外側にある次のステートメントに制御を移します。ネストされたステートメントの中では、break ステートメントは、囲んでいる最小の do、for、switch、または while ステートメントのみを終了します。

switch ステートメントにおいては、break は、制御を switch 本体から switch 本体の外側にある、次のステートメントに渡します。

continue ステートメント

continue ステートメント を使用すると、進行中のループの反復を終了することができます。プログラム制御は、continue ステートメントからループ本体の終わりに渡されます。

continue 文の形式は、次のとおりです。

continue ステートメントは、do、for、while などの繰り返しステートメントの本体の中にしか存在できません。

continue ステートメントは、反復するステートメントのアクション部分の処理を終了します。そして、制御を、ステートメントのループ連結部分へ移します。例えば、反復するステートメントが for ステートメントである場合、制御は、ステートメントの条件部分の 3 番目の式に移動します。次に、ステートメントの条件部分の 2 番目の式 (テスト) に移動します。

ネストされたステートメントの中では、continue ステートメントは、直接に continue ステートメントを囲んでいる do、for、または while ステートメントの現行の反復だけを終了します。

continue ステートメントの例

以下の例は、for ステートメントにおける continue ステートメントを示しています。continue ステートメントを使用すると、値が 1 以下の配列 rates のエレメントに対する処理がスキップ・オーバーされます。

```
/**
 ** This example shows a continue statement in a for statement.
 **/

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

```
int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00¥n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f¥n", rates[i]);
   }

   return(0);
}
```

プログラムは、以下の出力を作成します。

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

以下の例は、ネストされたループにおける continue ステートメントを示しています。内部ループが配列 strings の中である数に遭遇すると、そのループの反復は終了します。処理は、内部ループの 3 番目の式から続けられます。内部ループは、'¥0' エスケープ・シーケンスを検出した時点で終了します。

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '¥0';
      ++pointer)
      {                                   /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d¥n", letter_count);

   return(0);
}
```

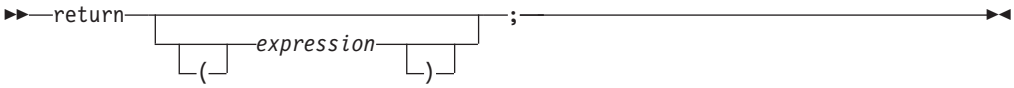
プログラムは、以下の出力を作成します。

```
letter count = 5
```

return ステートメント

return ステートメント は、現行の関数の処理を終了して、関数の呼び出し元に制御を戻します。

return ステートメントの構文



値を戻す関数は、return ステートメントが組み込まれていなければならず、その中には、*expression* が含まれていなければなりません。

C 非 void の戻りの型が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、コンパイラーは警告メッセージを出します。

C++ 非 void の戻りの型が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、コンパイラーはエラー・メッセージを出します。

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、関数の戻りの型が同じであるオブジェクトに割り当てられたかのように、戻り値の変換が行われます。

戻りの型 void の関数の場合、return ステートメントは必ず必要というわけではありません。 return ステートメントを検出することなくそのような関数の終わりに達すると、制御は、式のない return ステートメントが検出された場合と同じように呼び出し元へ渡されます。つまり、最終ステートメントの完了時に暗黙的な戻りが実行され、制御は自動的に呼び出し元関数に戻ります。

C++ return ステートメントが使用される場合、そこに式があってはなりません。

return ステートメントの例

return ステートメントの例を次に示します。

```
return;            /* Returns no value            */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1         */
return (x * x);    /* Returns the value of x * x  */
```

以下の関数は、整数の配列を検索して、変数 number と一致するものが存在するかどうか判別します。一致するものが存在すると、関数 match は i の値を戻します。一致するものが存在しない場合、関数 match は -1 (マイナス 1) という値を戻します。

```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
      if (number == array[i])
         return (i);
   return(-1);
}
```

1 つの関数に、複数の return ステートメントを含めることができます。次に例を示します。

```
void copy( int *a, int *b, int c)
{
   /* Copy array a into b, assuming both arrays are the same size */

   if (!a || !b)       /* if either pointer is 0, return */
      return;

   if (a == b)         /* if both parameters refer */
      return;          /*    to same array, return */

   if (c == 0)         /* nothing to copy */
      return;

   for (int i = 0; i < c; ++i;) /* do the copying */
      b[i] = a[1];
                       /* implicit return */
}
```

この例では、return ステートメントを使用して関数の早期終了が行われます。break ステートメントに似ています。

return ステートメントにある式は、そのステートメントが属している関数の戻りの型に変換されます。暗黙的な変換が不可能な場合、return ステートメントは無効です。

関連資料:

278 ページの『関数の戻りの型指定子』

279 ページの『関数からの戻り値』

goto ステートメント

goto ステートメント を使用すると、プログラムは、goto ステートメントで指定されたラベルに関連付けられたステートメントに無条件に制御を移します。

goto ステートメントの構文

goto ステートメントは、通常の処理シーケンスを妨げる可能性があるため、これを使用すると、プログラムの読み取りおよび保守が難しくなります。多くの場合、break ステートメント、continue ステートメント、または関数呼び出しを使用すれば、goto ステートメントを使用しなくてもすみます。

goto ステートメントを使用してアクティブ・ブロックを終了する場合、そのブロックから制御が移動するときに、すべてのローカル変数は破棄されます。

goto ステートメントを使用して、初期化を飛び越えることはできません。

goto ステートメントは、可変長配列のスコープ内でジャンプすることができますが、可変的に変更される型を持つオブジェクトの宣言を飛び越えてジャンプすることはできません。

以下の例は、ネストされたループからジャンプするために使用される goto ステートメントを示しています。この関数は、goto ステートメントを使用しなくても作成することができます。

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
   int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d¥n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6¥n");
}
```

計算後の goto ステートメント (IBM 拡張)

computed goto (計算型 goto) は、ターゲットが同一関数からのラベルである goto ステートメントです。ラベルのアドレスは void* 型の定数であり、これは、単項ラベル値演算子 && をラベルに適用することにより取得されます。computed goto のターゲットは実行時に判明し、同じ関数からの computed goto ステートメントのターゲットはすべて同じです。この言語機能は C99 および C++ の拡張機能であり、GNU 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ことを目的としてインプリメントされています。

計算型 goto ステートメントの構文

expression は、型 void の式です。

関連資料:

231 ページの『ラベル付きステートメント』

232 ページの『値としてのラベル (IBM 拡張)』

223 ページの『ラベル値演算子 (IBM 拡張)』

NULL ステートメント

NULL 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。

NULL ステートメントは、ラベル付きステートメントのラベルを保持したり、あるいは反復するステートメントの構文を完了したりすることができます。

以下の例は、配列 price のエレメントを初期化します。初期化は for 式の中で起きるため、for 構文を終了するのに必要なものはステートメントのみで、オペレーションは必要ありません。

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

ブロック・ステートメントの終わりの前にラベルを必要とするときに、 NULL 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 */
}
```

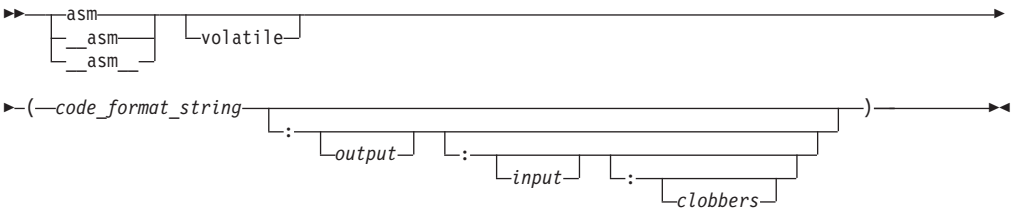
インライン・アセンブリー・ステートメント (IBM 拡張)

拡張言語レベルの下では、コンパイラーは、C および C++ ソース・ステートメント間での組み込みアセンブリー・コード・フラグメントを完全にサポートします。この拡張機能は、元々は GNU C を使用して開発された汎用システム・プログラミング・コード、およびオペレーティング・システム・カーネルおよびデバイス・ドライバーで使用するために、インプリメントされています。

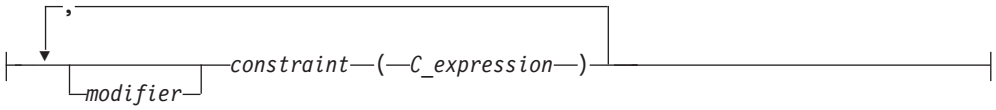
キーワード asm は、アセンブリー・コードを表します。コンパイルで厳格な言語レベルが使用されると、C コンパイラーは宣言内のキーワード asm を認識して、それを無視します。C++ コンパイラーは、常にこのキーワードを認識します。

構文は次のとおりです。

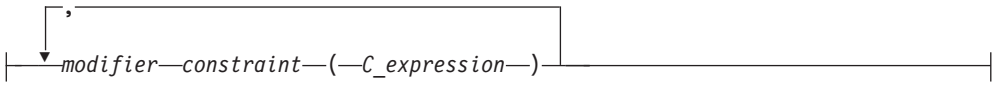
asm ステートメントの構文 - ローカル・スコープのステートメント



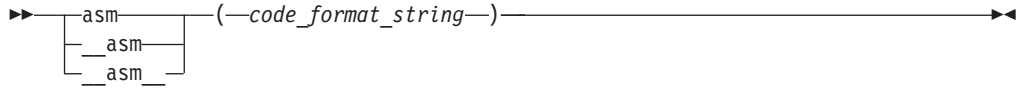
input:



output:



asm ステートメントの構文 - グローバル・スコープのステートメント



volatile

修飾子 volatile は、アセンブリー・ブロックで最小限の最適化のみを実行するようコンパイラーに指示します。コンパイラーは、アセンブリー・ブロックを囲む暗黙的なフェンスを超えて命令を移動させることはできません。詳しくは、例 1 を参照してください。

code_format_string

code_format_string は、asm 命令のソース・テキストであり、 printf フォーマット指定子に似たストリング・リテラルです。

オペランドは *%integer* の形式で参照されます。ここで、*integer* は入力または出力オペランドのシーケンス番号を示します。詳しくは、例 1 を参照してください。

読みやすくするために、各オペランドに大括弧で囲まれたシンボル名を割り当てることができます。アセンブラー・コード・セクションでは、各オペランドを *%[symbolic_name]* 形式で参照できます。これにより、オペランド・リスト内の *symbolic_name* が参照されます。シンボリック・オペランドには、既存の C または C++ シンボルをはじめとする任意の名前を使用できます。これは、シンボリック・オペランド名が C または C++ の ID と何の関係もないためです。ただし、同一のアセンブリー・ステートメントの中では、2 つのオペランドが同一のシンボル名は使用できません。詳しくは、例 2 を参照してください。

output *output* は、コンマで区切られた、ゼロまたは 1 つ以上のオペランドから成り立ちます。各オペランドは *constraint*(*C_expression*) の対から成り立ちます。出力オペランドは、= または + 修飾子 (下で説明) 、および追加の % または & 修飾子 (オプション) により制約する必要があります。

input *input* は、コンマで区切られた、ゼロまたは 1 つ以上のオペランドから成り立ちます。各オペランドは *constraint*(*C_expression*) の対から成り立ちます。

clobbers

clobbers は、二重引用符で囲み、コンマで区切ったレジスター名のリストです。 asm ステートメントの *input* または *output* にリストされていないレジスターを asm 命令で更新する場合、そのレジスターは破壊可能レジスターとしてリストする必要があります。 以下に、有効なレジスター名を示します。

r0 から r31
汎用レジスター

f0 から f31
浮動小数点レジスター

lr リンク・レジスター

ctr ループのカウント、減分およびブランチ用レジスター

fpscr 浮動小数点の状況および制御レジスター

xer 固定小数点例外レジスター

cr0 から cr7
条件レジスター。例 3 に、*clobbers* での条件レジスターの典型的な使用法を示します。

v0 から v31
ベクトル・レジスター (選択されたプロセッサー上のみ)

レジスター名のほかに、cc および memory を破壊可能レジスターのリストで使用することもできます。cc および memory の使用法に関する情報を以下に示します。

cc アセンブラー命令が条件コード・レジスターを変更する可能性がある場合は、cc を破壊可能レジスターのリストに追加します。

memory
アセンブラー命令がメモリー位置を予測不能な方法で変更する可能性がある場合は、memory を破壊可能レジスターのリストに追加します。memory という破壊可能レジスターを使用すると、アセンブリー・ステートメントの完了後に使用されるデータが有効であり同期化されることが保証されます。

ただし、memory という破壊可能レジスターを指定すると不要な再ロードを何度も実行する可能性があるため、ハードウェア・プリフェッチの効果が下がってしまう可能性があります。このため、memory という破壊可能レジスターはパフォーマンスを劣化させる可能性があり、注意して使用する必要があります。使用法について詳しくは、例 4 および 例 1 を参照してください。

modifier

修飾子 は、次の演算子のいずれかです。

= この命令では、オペランドが書き込み専用であることを示します。前の値は破棄され、出力データで置き換えられます。詳しくは、例 5 を参照してください。

+ オペランドが、命令によって読み込みと書き込みの両方に使用されることを示します。詳しくは、例 6 を参照してください。

& 入力オペランドを使用して、命令が終了する前にオペランドが変更される場合があることを示します。入力として使用されるレジスターは、ここでは再使用できません。

% 命令が、このオペランドと次のオペランドに対して可換性があることを宣言します。 このことは、このオペランドと次のオペランドの順序は、命令の生成時に交換できることを意味します。この修飾子は、入力または出力オペランドで使用できますが、最後のオペランドでは使用できません。詳しくは、例 7 を参照してください。

constraint

constraint は、許可されるオペランドの種類を記述するストリング・リテラルで、1 つの制約につき 1 文字です。指定できる制約は以下のとおりです。

b ゼロ以外の汎用レジスターを使用する。一部の命令は、レジスター 0 の指定を特別な方法で処理するため、コンパイラーが r0 を選択すると、予期するとおりに動作しません。これらの命令では、r0 の指定は、r0 が使用されることを意味しません。リテラル値 0 が指定されたことを意味します。詳しくは、例 8 を参照してください。

c CTR レジスターを使用する。

d 浮動小数点レジスターを使用する。

f 浮動小数点レジスターを使用する。詳しくは、例 7 を参照してください。

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する。POWER® では、レジスター、メモリー指定子、または即値オペランドを同義で使用できる命令はありません。ただし、可能な場合は、この制約が許容されます。

h CTR または LINK レジスターを使用する。

i オペランドに即値整数またはストリング・リテラルを使用する。

l CTR レジスターを使用する。

m マシンがサポートしているメモリー・オペランドを使用する。この制約は D(R) 形式のオペランドに対して使用できます。ここで、D は変位であり、R はレジスターです。詳しくは、例 9 を参照してください。

n 即値整数を使用する。

o オフセット可能なメモリー・オペランドを使用する。つまり、メモリー・オペランドは、基底アドレスに整数を加算することによってアドレス指定できます。POWER では、メモリー・オペランドは常にオフセット可能であるため、制約 o と m を同義で使用できます。

r 汎用レジスターを使用する。詳しくは、例 5 を参照してください。

s オペランドにストリング・リテラルを使用する。

v ベクトル・レジスターを使用。

0, 1, 2, ...
マッチング制約。出力では、対応する入力と同じレジスターを割り当てる。

I, J, K, L, M, N, O, P
定数値。オペランド内の式を折りたたんで、値を % 指定子に置き換える。これらの制約では、オペランドに対する最大値を、次のように指定します。

- I — 符号付き 16 ビット
- J — 符号なし 16 ビット、左シフト 16 ビット
- K — 符号なし 16 ビット定数
- L — 符号付き 16 ビット、左シフト 16 ビット
- M — 符号なし 31 より大きい定数
- N — 符号なし 2 の累乗の定数
- O — ゼロ
- P — 符号付で、その否定形が符号付き 16 ビット定数

C_expression

C_expression は C または C++ 式です。この式の値は、`asm` 命令のオペランドとして使用されます。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。 *C_expression* は、それに指定された制約と整合している必要があります。例えば、`i` を指定した場合には、オペランドは整数定数であることが必要です。

注: ポインター式が *input* または *output* で使用される場合、アセンブリー命令は ANSI 別名割り当て規則を順守する必要があります (詳しくは、118 ページの『型ベースの別名割り当て』を参照)。つまり、ポインター式のオペランドの値を使用する間接アドレッシングは、ポインター型と整合している必要があります。そうでない場合、コンパイル時に **-qalias=ansi** オプションを使用不可にする必要があります。

関連資料:


「XL C/C++ コンパイラー・リファレンス」の中の『-qalias=ansi』を参照

サポートされる構文とサポートされない構文

サポートされる構文

インライン・アセンブリー・ステートメントでは、以下の構文がサポートされます。

- 「Assembler Language Reference」にリストされているすべての命令ステートメント
- すべての拡張命令ニーモニック
- ラベル定義
- ラベルへのブランチ

サポートされない構文

インライン・アセンブリー・ステートメントでは、以下の構文はサポートされません。

- 疑似演算ステートメント。これらは、.function のように、ドット (.) で始まるアセンブリー・ステートメントです。
- 異なる asm ブロック間のブランチ

また、GNU コンパイラーから来ている一部の制約はサポートされませんが、可能な場合は、これらの制約が許容されます。例えば、制約 S および T は即値として処理されますが、コンパイラーは、それらがサポートされないことを示す警告メッセージを発行します。

インライン・アセンブリー・ステートメントの制約事項

以下の制限は、インライン・アセンブリー・ステートメントの使用に関するものです。

- アセンブラー命令は、asm ステートメント内で自己完結していることが必要です。 asm ステートメントは、命令を生成するために使用できるだけです。プログラムのその他の部分への関連付けは、すべて、出力および入力オペランド・リストを使用して設定する必要があります。
- オペランド・リストを経過せずに、外部シンボルを直接参照することはサポートされていません。
- 対のレジスターを必要とするアセンブラー命令は、制約によって指定できないので、サポートされません。例えば、long double オペランドでは %f 制約を使用できません。
- POWER7 上の浮動小数点スカラーとベクトル・レジスター間の共有レジスター・ファイルは、インライン・アセンブリー・ステートメントで共有されるためモデル化されません。clobbers リストでレジスター f0-f31 および v0-v31 を指定する必要があります。 結合された x0-x63 は存在しません。
- オペランド置換 (%0、%1 など) では、番号またはシンボル名の前でオプションの x を使用して、vsx レジスター参照を使用しなければならないことを指示することができます。例えば、レジスター v0 に割り振られたベクトル・オペランド %1 は、0 に置き換えられます (VMX 命令で使用するため)。同じオペランドがアセンブリー・テキストで %x1 として使用されている場合、それは 32 に置き換えられます (VSX 命令で使用するため)。この制限は、POWER7 など、VSX アーキテクチャー拡張をサポートするアーキテクチャーのみに適用されることに注意してください。

関連資料:

アセンブリー・ラベル (IBM 拡張)

指定したレジスターの変数 (IBM 拡張)


「XL C/C++ コンパイラー・リファレンス」の中の『-qasm』を参照

インライン・アセンブリー・ステートメントの例

例 1: 次の例は、volatile キーワードの使用法を示しています。

```
#include <stdio.h>

inline bool acquireLock(int *lock){
  bool returnvalue = false;
  int lockval;
  asm volatile(
              /*--------a fence here-----*/

               "  0: lwarx %0,0,%2   ¥n" // Loads the word and reserves
                                         // a memorylocation for the subsequent
                                         // stwcx. instruction.

               "     cmpwi %0,0      ¥n" // Compares the lock value to 0.
               "     bne- 1f         ¥n" // If it is 0, you can acquire the
                                         // lock. Otherwise, you fail get the
                                         // lock and must try again later.

               "     ori %0,%0,1     ¥n" // Sets the lock to 1.
               "     stwcx. %0,0,%2  ¥n" // Tries to conditionally store 1
                                         // into the lock word to acquire
                                         // the lock.

               "     bne- 0b         ¥n" // Reservation was lost. Try again.

               "     isync           ¥n" // Lock acquired. The isync instruction
                                         // implements an import barrier to
                                         // ensure that the instructions that
                                         // access the shared region guarded by
                                         // this lock are executed only after
                                         // they acquire the lock.

               "     ori  %1,%1,1    ¥n" // Sets the return value for the
                                         // function acquireLock to true.

               "  1:                 ¥n" // Did not get the lock. // Will return false.

               /*------a fence here------*/

               :     "+r"   (lockval),
                     "+r"   (returnvalue)
               :     "r"    (lock)       // Lock is the address of the lock in
                                         // memory.

               :     "cr0"               // cr0 is clobbered by cmpwi and stwcx.
               );

  return returnvalue;
}
int main()
{
  int myLock;
  if(acquireLock(&myLock)){
       printf("got it!¥n");
  }else{
       printf("someone else got it¥n");
  }
  return 0;
}
```

この例で、%0 は第 1 オペランド "+r"(lockval) を参照し、%1 は第 2 オペランド "+r"(returnvalue) を参照し、%2 は第 3 オペランド "r"(lock) を参照します。

アセンブリー・ステートメントは、ロックを使用して共有ストレージへのアクセスを制御します。命令が共有ストレージにアクセスするには、ロックを獲得する必要があります。

volatile キーワードでは、アセンブリー命令グループの周囲にフェンスを暗黙指定するため、アセンブリー命令をアセンブリー・ブロックの外や周囲に移動させることはできません。

volatile キーワードを指定しない場合、コンパイラーは最適化のために命令を周囲に移動させることができます。これにより、一部の命令が、ロックを獲得せずに共有ストレージにアクセスしてしまう可能性があります。

このアセンブリー・ステートメント内では、命令が予期しない方法でメモリーを変更することはないため、memory という破壊可能レジスターを使用する必要はありません。memory という破壊可能レジスターを使用しても、プログラムは適切に機能します。ただし、memory という破壊可能レジスターを使用すると不要な再ロードを何度も実行するため、パフォーマンスが劣化します。

例 2: 次の例は、入力および出力オペランドのシンボル名の使用法を示しています。

```
int a ;
int b = 1, c = 2, d = 3 ;
__asm("  addc %[result], %[first], %[second]"
      : [result]      "=r"       (a)
      : [first]       "r"        (b),
        [second]      "r"        (d)
      );
```

この例で、%[result] は出力オペランド変数 a を参照し、%[first] は入力オペランド変数 b を参照し、%[second] は入力オペランド変数 d を参照します。

例 3: 次の例は、*clobbers* での条件レジスターの典型的な使用法を示しています。

```
  asm ("   add. %0,%1,%2  ¥n"
       :   "=r"    (c)
       :   "r"     (a),
           "r"     (b)
       :   "cr0"
       );
```

この例で add. 命令は、アセンブリー・ステートメントの *input* および *output* にリストされたレジスターだけでなく、条件レジスター・フィールド 0 にも影響を及ぼします。そのため、cr0 を *clobbers* に追加することで、コンパイラーにそのことを知らせる必要があります。

例 4: 次の例は、memory という破壊可能レジスターの使用法を示しています。

```
 asm volatile ("   dcbz 0, %0      ¥n"
                :   "=r"(b)
                :
                :   "memory"
                );
```

この例で、命令 dcbz はキャッシュ・ブロックをクリアし、それによってそのメモリー位置の変数が変更された可能性があります。コンパイラーには、どの変数が変更されたかを知る方法がありません。そのため、コンパイラーは、その命令によって変更されたメモリーにより、すべてのデータに別名が付けられている可能性があるとみなします。

結果として、アセンブリー・ステートメントの完了後に、必要なものすべてをメモリーから再ロードする必要があります。memory という破壊可能レジスターを使用すると、プログラムの正確性は保証されますが、プログラムのパフォーマンスが低下

します。これは、コンパイラーがアセンブリー・ステートメントと関係のないデータを再ロードする可能性があるためです。

例 5: 次の例は、= 修飾子および r 制約の使用法を示しています。

```
int a ;
int b = 100 ;
int c = 200 ;
asm("   add %0, %1, %2"
    :   "=r"     (a)
    :   "r"      (b),
        "r"      (c)
    );
```

add 命令は、2 つの汎用レジスターの内容を加算します。%0、%1、および %2 オペランドは、出力/入力オペランド欄の C 式によって置換されます。

出力オペランドは、= 修飾子を使用して、変更可能オペランドが必須であることを示し、r 制約を使用して、汎用レジスターが必須であることを示します。同様に、入力オペランドの r 制約は、汎用レジスターが必要であることを示します。これらの制約事項の範囲内で、コンパイラーは %0、%1、および %2 を置換するレジスターを自由に選択できます。

注: コンパイラーが第 2 オペランドとして r0 を選択すると、add 命令はリテラル値 0 を使用して、予期しない結果をもたらします。このため、コンパイラーが第 2 オペランドとして r0 を選択しないようにするために、b 制約を使用して第 2 オペランドを表すことができます。

例 6: 次の例は、+ 修飾子および K 制約の使用法を示しています。

```
 asm ("   addi %0,%0,%2"
      :   "+r"    (a)
      :   "r"     (a),
          "K"     (15)
      );
```

このアセンブリー・ステートメントはオペランド %0 とオペランド %2 を加算し、その結果をオペランド %0 に書き込みます。出力オペランドに + 修飾子を使用することで、命令がオペランド %0 を読み書きできることを指定します。K 制約は、オペランド %2 にロードされた値が符号なし 16 ビット定数値である必要があることを指定します。

例 7: 次の例は、% 修飾子および f 制約の使用法を示しています。

```
asm("   fadd %0, %1, %2"
    :   "=f"    (c)
    :   "%f"    (a),
        "f"     (b)
    );
```

このアセンブリー・ステートメントは、オペランド a と b を加算し、その結果をオペランド c に書き込みます。% 修飾子は、オペランド a と b の交換によってコンパイラーがより良いコードを生成できる場合は、それらを交換できることを指定します。各オペランドには f 制約があり、これは浮動小数点レジスターが必須であることを指定します。

例 8: 次の例は、b 制約の使用法を示しています。

```
char res[8]={'a','b','c','d','e','f','g','h'};
char a='y';
int  index=7;

asm ("   stbx %0,%1,%2       ¥n"     ¥
     :                               ¥
     :    "r"    (a),
          "b"    (index),
          "r"    (res)
     );
```

この例で、b 制約は、入力オペランド %1 として r0 以外の汎用レジスターを選択するようにコンパイラーに指示します。このプログラムの結果ストリングは abcdefgy です。しかし、r 制約を使用して、コンパイラーが %1 として r0 を選択すると、この命令は間違った結果ストリング ybcdefgh を生成します。このため、r0 の指定を特別な方法で処理する命令では、b 制約を使用して入力オペランドを表すことが重要です。

例 9: 次の例は、m 制約の使用法を示しています。

```
 asm ("   stb %1,%0       ¥n"     ¥
      :   "=m"    (res)           ¥
      :   "r"     (a)
      );
```

この例で、命令 stb の構文は stb RS,D(RA) です。ここで、D は変位であり、R はレジスターです。D+RA は D(RA) から計算されて、有効アドレスを形成します。レジスターと変位を別々に指定して有効アドレスを手動で構築する必要はありません。

正しいオフセットが何になるか、またそれがスタックからのオフセットか TOC (目次) からのオフセットかにかかわらず、単一制約 m または o を使用して、命令で 2 つのオペランドを参照できます。これによりコンパイラーは正しいレジスター (例えば、自動変数の場合は r1) を選択し、正しい変位を自動的に適用することができます。

第 8 章 関数

プログラミング言語のコンテキストでは、*関数* という語は、出力値を計算するために使用されるステートメントの集まりを意味します。この語は、数学で使われるときほど厳密に使われていません。数学では、関数は、入力変数を出力変数に一意的に 関連付ける集合を意味します。 C または C++ プログラムにおける関数は、すべての入力に対して一貫性のある出力を生成するとは限らず、出力をまったく生成しないこともあり、副次作用を持つこともあります。関数は、パラメーター・リストのパラメーター (存在する場合) をオペランドとする、ユーザー定義の演算と考えることができます。

関数については、以下の情報があります。

- 関数の宣言と定義
- 関数のストレージ・クラス指定子
- 関数指定子
- 関数の戻りの型指定子
- 関数宣言子
- 関数属性 (IBM 拡張)
- main() 関数
- 関数呼び出し
- C++ 関数内のデフォルトの引数
- 関数を指すポインター
- ネストされた関数 (IBM 拡張)

関数の宣言と定義

関数宣言 と*関数定義* の違いは、データ宣言とデータ定義の違いに似ています。宣言では関数の名前と特性を設定しますが、関数にストレージを割り振りません。一方、*定義* では、関数の本体を指定し、関数に ID を関連付け、関数にストレージを割り振ります。したがって、次の例で宣言された ID は、

```
float square(float x);
```

ストレージを割り振りません。

関数定義 には、関数宣言と関数本体が含まれます。本体とは、関数の処理を実行するステートメントのブロックです。この例で宣言された ID はストレージを割り振ります。これらは、宣言と定義の両方になります。

```
float square(float x)
{ return x*x; }
```

1 つの関数をプログラム内で複数回宣言することができますが、特定の関数のすべての宣言は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

ーの型が同じでなければなりません。ただし、1 つの関数には 1 つの定義しか認められません。通常、宣言はヘッダー・ファイルに入れますが、定義はソース・ファイルに入れます。

関数宣言

前に戻りの型が付き、後にパラメーター・リストが続く関数 ID は、*関数宣言* または*関数プロトタイプ* と呼ばれます。プロトタイプは、コンパイラーに、その関数を使用する前に関数の形式と存在を知らせます。コンパイラーは、関数呼び出しのパラメーターと関数宣言におけるパラメーターとの不一致を検査することができます。コンパイラーは、引数の型の検査および引数の変換のためにも、この宣言を使用します。

C++ 関数の暗黙宣言は許可されていません。どの関数も、呼び出す前に、明示的に宣言しておく必要があります。

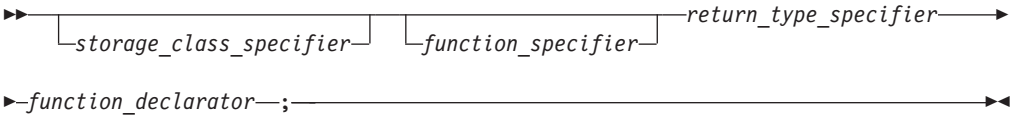
C 関数の呼び出しが行われた時点で関数宣言が可視でない場合、コンパイラーは、`extern int func();` の暗黙宣言を想定します。ただし、C99 に準拠するためには、ユーザーは、関数を呼び出す前に、その関数を明示的にプロトタイプ化する必要があります。

関数を宣言する際のエレメントは、次のとおりです。

- 270 ページの『関数のストレージ・クラス指定子』。これは、リンケージを指定します。
- 278 ページの『関数の戻りの型指定子』。これは、戻される値のデータ型を指定します。
- 273 ページの『関数指定子』。これは、関数の追加プロパティーを指定します。
- 280 ページの『関数宣言子』。これは、パラメーターのリストだけでなく、関数の ID を含みます。

関数宣言の形式はすべて、次のとおりです。

関数宣言の構文



C++11

注: *function_declarator* が後置戻り型を組み込む場合、*return_type_specifer* は `auto` でなければなりません。後置戻り型について詳しくは、283 ページの『後置戻り型 (C++11)』を参照してください。

C++11

IBM さらに、GNU C および C++ との互換性を保つために、XL C/C++ では、*属性* を使用して関数のプロパティーを変更することができます。これについては、 286 ページの『関数属性 (IBM 拡張)』で説明しています。

関数定義

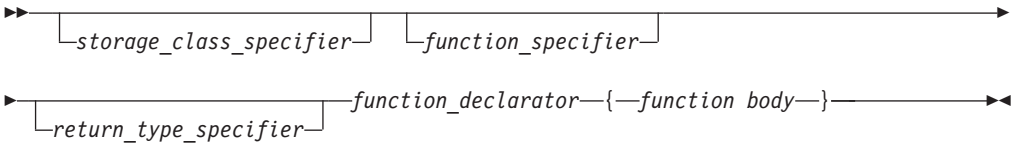
関数定義のエレメントは、次のとおりです。

- 270 ページの『関数のストレージ・クラス指定子』。これは、リンケージを指定します。
- 278 ページの『関数の戻りの型指定子』。これは、戻される値のデータ型を指定します。
- 273 ページの『関数指定子』。これは、関数の追加プロパティーを指定します。
- 280 ページの『関数宣言子』。これは、パラメーターのリストだけでなく、関数の ID を含みます。
- *関数本体*。これは、その関数が実行するアクションを表す、中括弧で囲まれた一連のステートメントです。
- C++ コンストラクター初期化指定子。これは、クラスで宣言されるコンストラクター関数でのみ使用されます。これについては、411 ページの『コンストラクター』で説明します。
- C++ try ブロック。これは、クラス関数で使用されます。これについては493 ページの『try ブロック』で説明します。

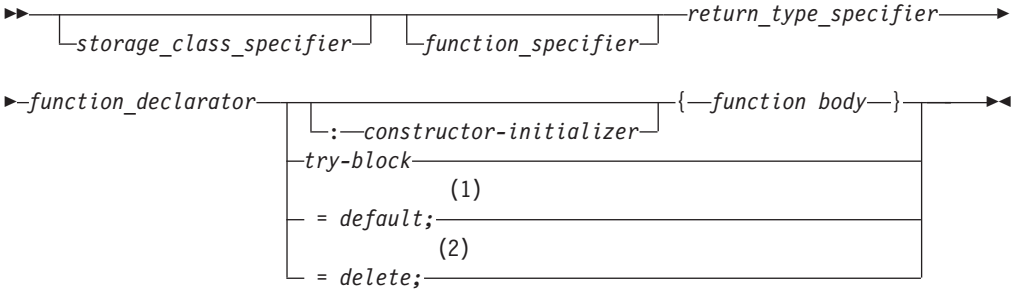
IBM さらに、GNU C および C++ との互換性を保つために、XL C/C++ では、*属性* を使用して関数のプロパティーを変更することができます。これについては、 286 ページの『関数属性 (IBM 拡張)』で説明しています。 IBM

関数定義の形式は、次のとおりです。

関数定義の構文 (C のみ)



関数定義の構文 (C++ のみ)



注:

1 この構文は、C++11 標準でのみ有効です。

2 この構文は、C++11 標準でのみ有効です。

C++11

注: *function_declarator* が後置戻り型を組み込む場合、*return_type_specifer* は `auto` でなければなりません。後置戻り型について詳しくは、283 ページの『後置戻り型 (C++11)』を参照してください。

C++11

明示的にデフォルトに設定された関数

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

明示的にデフォルト設定された関数 宣言は、C++11 標準に導入された新しい形式の関数宣言です。 `=default;` 指定子を関数宣言の末尾に追加して、その関数を明示的にデフォルト設定された関数として宣言することができます。コンパイラーは、明示的にデフォルトに設定された関数に対してデフォルトの実装を生成します。これは手動でプログラミングする関数実装よりも効率的です。明示的にデフォルト設定された関数は、特殊メンバー関数でなければならず、デフォルトの引数は持ちません。明示的にデフォルトに設定された関数により、関数を手動で定義する手間を省くことができます。

明示的にデフォルトに設定された関数は、インラインまたはアウト・オブ・ラインのどちらででも宣言できます。次に例を示します。

```
class A{
public:
   A() = default;            // Inline explicitly defaulted constructor definition
   A(const A&);
   ~A() = default;           // Inline explicitly defaulted destructor definition
};

A::A(const A&) = default;    // Out-of-line explicitly defaulted constructor definition
```

関数が特殊メンバー関数であり、デフォルトの引数がない場合にのみ、関数を明示的にデフォルト設定された関数として宣言できます。次に例を示します。

```
class B {
public:
  int func() = default;     // Error, func is not a special member function.
  B(int, int) = default;    // Error, constructor B(int, int) is not
                            // a special member function.
  B(int=0) = default;       // Error, constructor B(int=0) has a default argument.
};
```

明示的にデフォルトに設定された関数宣言により、コンパイラーが明示的にデフォルトに設定された関数を単純な関数として扱うことができるので、最適化の機会をさらに拡張できます。

関連資料:

削除済み関数 (C++11)

409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

削除済み関数

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

削除済み関数 宣言は、C++11 標準に導入された新しい形式の関数宣言です。削除済み関数として関数を宣言するためには、`=delete;` 指定子を関数宣言の末尾に付加することができます。コンパイラーは、削除済み関数を使用できないようにします。

暗黙的に定義された関数は、その関数を使用されないようにする場合に、削除済み関数として宣言できます。例えば、暗黙的に定義されたクラスのコピー割り当て演算子とコピー・コンストラクターを、削除済み関数として宣言して、そのクラスのオブジェクト・コピーが行われないようにすることができます。

```
class A{
public:
  A(int x) : m(x) {}
  A& operator = (const A &) = delete;     // Declare the copy assignment operator
                                          // as a deleted function.
  A(const A&) = delete;                   // Declare the copy constructor
                                          // as a deleted function.

private:
  int m;
};

int main(){
  A a1(1), a2(2), a3(3);
  a1 = a2;            // Error, the usage of the copy assignment operator is disabled.
  a3 = A(a2);         // Error, the usage of the copy constructor is disabled.
}
```

さらに、望ましくない変換コンストラクターや演算子を削除済み関数と宣言することで、問題のある変換を回避することもできます。以下の例は、`double` からクラス型への望ましくない変換を回避する方法を示しています。

```
class B{
public:
  B(int){}
  B(double) = delete;   // Declare the conversioin constructor as a deleted function
};
```

```
int main(){
  B b1(1);
  B b2(100.1);          // Error, conversion from double to class B is disabled.
}
```

削除済み関数は暗黙的にインラインになります。関数の削除済み定義は、関数の最初の宣言でなければなりません。次に例を示します。

```
class C {
public:
  C();
};

C::C() = delete;    // Error, the deleted definition of function C must be
                    // the first declaration of the function.
```

関連資料:

明示的にデフォルトに設定された関数 (C++11)

関数宣言の例

以下のコード・フラグメントは、それぞれ、関数宣言 (または *プロトタイプ*) を示しています。最初の部分は、2 つの整数の引数を採用し、戻りの型が `void` である関数 `f` を宣言しています。

```
void f(int, int);
```

このフラグメントは、固定文字を指すポインターを取り、整数を戻す関数を指すポインター `p1` を宣言しています。

```
int (*p1) (const char*);
```

次のコードは、関数 `f1` を宣言し、関数 `f1` は、1 つの整数の引数を取り、整数の引数を取って整数を戻す関数を指すポインターを戻します。

```
int (*f1(int)) (int);
```

関数 `f1` の複雑な戻りの型に対して、上記の関数の代わりに、`typedef` を使用することができます。

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

次の宣言には、最初の引数として定数整数を取り、可変数の、可変の型の他の属性を持つことができ、型 `int` を戻す外部関数 `f2` があります。

```
extern int  f2(const int, ...);
```

C++ 関数 `f4` は、引数を取らず、戻りの型が `void` で、`X` 型および `Y` 型のクラス・オブジェクトをスローすることができます。

```
class X;
class Y;

// ...

void f4() throw(X,Y);
```

C++

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、外部リンケージがあり、int 型のオブジェクトを戻し、 x および y と宣言された int 型の 2 つのパラメーターがあります。この関数本体には、x と y の合計を戻す 1 個のステートメントが入っています。

次の関数 set_date は、date 型の構造体を指すポインターをパラメーターとして宣言します。 date_ptr は、ストレージ・クラス指定子 register を持っています。

```
void set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

互換性のある関数 (C のみ)

互換性のある 2 つの関数型の場合、次の要件を満たす必要があります。

- パラメーターの数 (および省略符号の使用) を一致させなければなりません。
- 互換性のある戻りの型を持たなければなりません。
- 対応するパラメーターは、デフォルト引数プロモーションのアプリケーションの結果による型と、互換性がなければなりません。

2 つの互換性のある関数型の複合型は、次のように決定されます。

- 関数の型の 1 つにパラメーター型リストがある場合、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あること。
- 両方の関数型がパラメーター型リストを持つ場合、各パラメーターの複合型は次のように決定されます。
  - 異なるランクのパラメーターの複合は、デフォルト引数プロモーションのアプリケーションの結果による型です。
  - 配列または関数型を持つパラメーターの複合は、調整された型です。
  - 修飾された型を持つパラメーターの複合は、宣言された型の非修飾版です。

例えば、2 つの関数宣言は以下のようになります。

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

結果の複合型は次のようになります。

```
int f(int (*)(char *), double (*)[3]);
```

関数宣言子が関数宣言の一部でない場合、パラメーターは不完全型にすることができます。このパラメーターは、宣言子指定子のシーケンスで [*] 表記を使用して、可変長の配列型を指定することもできます。以下は、互換性のある関数プロトタイプ宣言子の例です。

```
int myMin(int n, int m, int a[n][m]);
int myMin(int n, int m, int a[*][*]);
int myMin(int n, int m, int a[ ][*]);
int myMin(int n, int m, int a[ ][m]);
```

関連資料:

互換型および複合型

多重関数宣言 (C++ のみ)

ある特定の関数に対する多重関数宣言はすべて、パラメーターの数と型が同じでなければなりません。また、戻りの型も同じでなければなりません。

これらの戻りの型およびパラメーター型は、関数型の一部ですが、デフォルトの引数と例外指定は、関数型の一部ではありません。

既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、リンケージをもたずに、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もちます。

引数のマッチングの観点では、省略符号とリンケージ・キーワードは、関数型の一部と見なされます。これらは、関数のすべての宣言において、矛盾しないように使用する必要があります。 2 つの宣言におけるパラメーター型で、typedef 名または未指定の引数の配列境界の使用だけが相違している場合、その宣言は同じです。const 型修飾子または volatile 型修飾子も関数型の一部ですが、これらは、非静的メンバー関数の宣言または定義の一部となることができるだけです。

戻りの型とパラメーター・リストの両方で 2 つの関数宣言が一致している場合は、2 番目の宣言は最初の宣言の再宣言と見なされます。以下の例では、同じ関数が宣言されています。

```
int foo(const string &bar);
int foo(const string &);
```

戻りの型が違うだけの 2 つの関数宣言は、無効な関数多重定義となり、コンパイル時エラーのフラグが付けられます。次に例を示します。

```
void f();
int f();      // error, two definitions differ only in
              // return type
int g()
{
   return f();
}
```

関連資料:

325 ページの『関数の多重定義』

関数のストレージ・クラス指定子

関数の場合は、ストレージ・クラス指定子は、関数のリンケージを決めます。デフォルトで、関数定義は、外部リンケージを持ち、他のファイルで定義された関数で呼び出すことができます。 C 1 つの例外はインライン関数です。インライン関数は、デフォルトで内部結合を持っているものとして処理されます。詳しくは、274 ページの『インライン関数のリンケージ』を参照してください。

ストレージ・クラス指定子は、関数宣言と関数定義の両方で使用できます。関数のストレージ・クラス・オプションは、次の 2 つだけです。

- static
- extern

静的ストレージ・クラス指定子

static ストレージ・クラス指定子を使って宣言された関数は、内部リンケージを持ちます。これは、この関数が、関数が定義されている変換単位内でしか呼び出しできないことを意味します。

static ストレージ・クラス指定子は、関数宣言がファイル・スコープにある場合のみ、関数宣言で使用できます。ブロック内の関数を static と宣言することはできません。

C++ この static の使用法は、C++ では推奨されません。その代わりに、関数を名前なし名前空間に入れます。

関連資料:

8 ページの『内部リンケージ』

313 ページの『第 9 章 名前空間 (C++ のみ)』

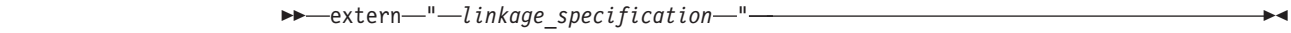
extern ストレージ・クラス指定子

extern ストレージ・クラス指定子を宣言された関数は外部リンケージを持ちます。つまり、その関数は、他の変換単位から呼び出すことができます。キーワード extern はオプションです。ストレージ・クラス指定子を指定しない場合は、関数に外部リンケージを持つと想定されています。

C++ XL 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。

C++ XL C++ では、extern キーワードを、リンケージの型を指定する引数と一緒に使用できます。

extern 関数ストレージ・クラス指定子の構文



ここで、*linkage_specification* は次のいずれかです。

- C
- C++

次のコードの部分は、extern "C" の使用を示しています。

```
extern "C" int cf();      //declare function cf to have C linkage

extern "C" int (*c_fp)(); //declare a pointer to a function,
                          // called c_fp, which has C linkage

extern "C" {
  typedef void(*cfp_T)(); //create a type pointer to function with C
                          // linkage
```

```
  void cfn();             //create a function with C linkage
  void (*cfp)();          //create a pointer to a function, with C
                          // linkage
}
```

リンケージの互換性は、qsort などのパラメーターとしてユーザー関数ポインターを受け入れるすべての C ライブラリー関数に影響を与えます。extern "C" リンケージ指定を使用して、宣言されたリンケージを同じものにする必要があります。次のフラグメント例は、extern "C" を qsort と一緒に使用しています。

```
#include <stdlib.h>

// function to compare table elements
extern "C" int TableCmp(const void *, const void *); // C linkage
extern void * GenTable();                             // C++ linkage

int main() {
  void *table;

  table = GenTable();               // generate table
  qsort(table, 100, 15, TableCmp);  // sort table, using TableCmp
                                    // and C library routine qsort();
}
```

C++ 言語は、多重定義をサポートしますが、ほかの言語では、多重定義をサポートできません。これは次のことを意味します。

- 関数に C++ (デフォルト) のリンケージが指定されている限り、その関数を多重定義できます。したがって、XL C++ は次の一連のステートメントを許可します。

  ```
  int func(int);             // function with C++ linkage
  int func(char);            // overloaded function with C++ linkage
  ```

  それに対して、C++ 以外のリンケージが指定された関数は、多重定義できません。

  ```
  extern "FORTRAN"{int func(int);}
  extern "FORTRAN"{int func(int,int);}  // not allowed
                                        //compiler will issue an error message
  ```

- 1 つの C++ 以外のリンケージ関数のみが、多重定義関数と同じ名前を持つことができます。次に例を示します。

  ```
  int func(char);
  int func(int);
  extern "FORTRAN"{int func(int,int);}
  ```

  しかし、C++ 以外のリンケージ関数は、同じ名前を持つ C++ 関数と同じパラメーターを持つことはできません。

  ```
  int func(char);      // first function with C++ linkage
  int func(int, int);  // second function with C++ linkage
  extern "FORTRAN"{int func(int,int);}  // not allowed since the parameter
                                        // list is the same as the one for
                                        // the second function with C++ linkage
                                        // compiler will issue an error message
  ```

C++

関連資料:

9 ページの『外部リンケージ』

5 ページの『クラス・スコープ (C++ のみ)』

313 ページの『第 9 章 名前空間 (C++ のみ)』

関数指定子

関数定義に使用できる関数指定子は、次のとおりです。

- C++11 constexpr。constexpr 関数および constexpr コンストラクターの宣言に使用できます。詳しくは、 99 ページの『constexpr 指定子 (C++11)』で説明しています。 C++11
- inline。コンパイラーに、関数呼び出しポイントで関数定義を拡張するよう指示します。
- C++ explicit。クラスのメンバー関数のみに使用できます。詳しくは、 429 ページの『明示的変換コンストラクター』で説明しています。 C++
- C11 _Noreturn。関数が呼び出し元に戻らないことを示します。 C11
- C++ virtual。クラスのメンバー関数のみに使用できます。 400 ページの『仮想関数』で説明します。 C++

inline 関数指定子

インライン関数とは、コンパイラーが、個別の命令セットをメモリー内に作成するのではなく、関数定義からのコードを呼び出し元関数のコードに直接コピーする関数です。関数コード・セグメントとの間で制御権を移動するのではなく、変更された関数本体のコピーを関数呼び出しの代わりに直接使用することができます。このようにすると、関数呼び出しのパフォーマンス・オーバーヘッドを回避することができます。 inline 指定子の使用は、インライン展開を実行できるという提案をコンパイラーに行うだけです。コンパイラーはその提案を自由に無視できます。

C main を除くすべての関数は、inline 関数指定子を使用してインラインとして宣言または定義することができます。静的ローカル変数を、インライン関数の本体で定義することは許可されていません。

C++ クラス宣言の中でインプリメントされた C++ 関数は自動的にインラインと定義されます。main 以外の、クラス宣言の外で宣言された通常の C++ 関数とメンバー関数は、inline 関数指定子を使用してインラインとして宣言または定義することができます。インライン関数の本体で定義される静的ローカル変数およびストリング・リテラルは、変換単位をまたがって同じオブジェクトとして処理されます。詳しくは、 274 ページの『インライン関数のリンケージ』 を参照してください。

次のコード・フラグメントはインライン関数定義を示しています。

```
inline int add(int i, int j) { return i + j; }
```

inline 指定子を使用しても、関数の意味は変わりません。ただし、関数のインライン展開を行うと、実引数の評価の順序が保持されなくなる場合があります。

インライン関数の最も効果的なコーディング方法は、インライン関数の定義をヘッダー・ファイルに配置し、次に、そのヘッダーを、インライン化したい関数への呼び出しを含む任意のファイルに組み込むことです。

注: inline 指定子は次のキーワードで表されます。

- C inline キーワードは、**xlc** または**c99** を使用するか、あるいは **-qlanglvl=stdc99** または **-qlanglvl=extc99** オプション、あるいは **-qkeyword=inline** を使用したコンパイルで認識されます。 __inline__ (または __inline) キーワードは、すべての言語レベルで認識されますが、このキーワードのセマンティクスについては後述の『インライン関数のリンケージ』を参照してください。
- C++ inline キーワード、__inline キーワード、および __inline__ キーワードは、すべての言語レベルで認識されます。

インライン関数のリンケージ

C C では、インライン関数はデフォルトで *静的* 結合を持っているものとして処理されます。すなわち、インライン関数は単一の変換単位内でのみ可視です。よって以下の例では、関数 foo がまったく同じ方法で定義されていますが、ファイル a.c の foo とファイル b.c の foo は別の関数として処理されます。2 つの関数の本体が生成され、メモリー内の異なる 2 つのアドレスが割り当てられます。

```
// a.c

#include <stdio.h>

inline int foo(){
return 3;
}

void g() {
printf("foo called from g: return value = %d, address = %p¥n", foo(), &foo);
}


// b.c

#include <stdio.h>

inline int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d, address = %p¥n", foo(), &foo);
g();
}
```

コンパイル済みプログラムの出力は、次のようになります。

```
foo called from main: return value = 3, address = 0x10000580
foo called from g: return value = 3, address = 0x10000500
```

インライン関数は内部リンケージを持っているものとして処理されるので、インライン関数の定義は、別の変換単位内で同じ名前を持つ関数の、通常の外部定義と共存できます。ただし、インライン定義を含むファイルから関数を呼び出す場合、コンパイラーはその呼び出しに対して、同じファイルに定義されたインライン・バージョンまたは 別のファイルに定義された外部バージョンのいずれか を選択できます。プログラムは、呼び出されるインライン・バージョンに依存しません。以下の例では、関数 g からの foo の呼び出しで、6 または 3 のいずれかを戻します。

```
// a.c

#include <stdio.h>

inline int foo(){
return 6;
}

void g() {
printf("foo called from g: return value = %d¥n", foo());
}


// b.c

#include <stdio.h>

int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d¥n", foo());
g();
}
```

同様に、関数を extern inline として定義する場合、または inline 関数を extern として再宣言する場合は、その関数は単に通常の外部関数となるだけで、インライン化されません。

IBM 末尾に下線を付けて __inline__ キーワードを指定すると、コンパイラーはインライン関数に GNU C のセマンティクスを使用します。 C99 のセマンティクスと対比して、__inline__ として定義された関数は外部定義のみを提供し、static __inline__ と定義された関数は内部リンケージ (C99 と同様) を持つインライン定義を提供します。さらに、extern __inline__ と定義された関数は、最適化を使用可能にしてコンパイルすると、同じ関数のインライン定義と外部定義の共存を許可します。インライン関数の GNU C インプリメンテーションについて詳しくは、 GCC 資料 (http://gcc.gnu.org/onlinedocs/ で入手可能) を参照してください。
IBM

C++ インライン関数は、その関数が使用または呼び出される各変換単位内で、まったく同一の方法で定義する必要があります。また、関数が inline として定義されていても同じ変換単位内で使用されないか、または呼び出されることがなければ、その関数はコンパイラーによって破棄されます (**-qkeepinlines** オプションを使用してコンパイルしない場合)。

それでもなお、C++ では、インライン関数はデフォルトで外部 結合を持っているものとして処理されます。つまり、プログラムは、関数のコピーが 1 つしかないかのような動作をします。関数はすべての変換単位内で同じアドレスを持ち、それぞれの変換単位はすべての静的ローカル変数とストリング・リテラルを共用します。 したがって、前述の例のコンパイルでは、出力は以下のようになります。

```
foo called from main: return value = 3, address = 0x10000580
foo called from g: return value = 3, address = 0x10000580
```

異なる関数本体に同じ名前を使用してインライン関数を再定義することは正しくありません。ただし、コンパイラーはこれにエラーとしてのフラグを立てません。単に、コンパイルのコマンド行に最初に入力されたファイルで定義済みのバージョンの関数本体を生成して、それ以外のものは破棄するだけです。 したがって、インライン関数 foo を 2 つの異なるファイル内に別々に定義している以下の例では、期待される結果を出すことはできません。

```
// a.C

#include <stdio.h>

inline int foo(){
return 6;
}

void g() {
printf("foo called from g: return value = %d, address = %p¥n", foo(), &foo);
}


// b.C

#include <stdio.h>

inline int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d, address = %p¥n", foo(), &foo);
g();
}
```

コマンド xlc++a.C b.C を使用してコンパイルすると、出力は次のようになります。

```
foo called from main: return value = 6, address = 0x10001640
foo called from g: return value = 6, address = 0x10001640
```

main から foo への呼び出しでは、b.C に提供されたインライン定義を使用せず、foo を a.C に定義された通常の外部関数として呼び出します。予期せぬ結果が出ることを回避するために、同じ名前のインライン関数定義が変換単位をまたがって厳密に一致していることを確認するのはお客様の責任です。

インライン関数は外部リンケージを持っているものとして処理されるので、インライン関数の本体で定義されるすべての静的ローカル変数またはストリング・リテラルは、変換単位をまたがって同じオブジェクトとして処理されます。次の例は、このことを示しています。

```
// a.C

#include <stdio.h>

inline int foo(){
static int x = 23;
printf("address of x = %p¥n", &x);
x++;
return x;
}
```

```
void g() {
printf("foo called from g: return value = %d¥n", foo());
}


// b.C

#include <stdio.h>

inline int foo()
{
static int x=23;
printf("address of x = %p¥n", &x);
x++;
return x;
}

void g();

int main() {
printf("foo called from main: return value = %d¥n", foo());
g();
}
```

このプログラムの出力では、foo の両方の定義内の x は、まったく同じオブジェクトであることを示しています。

```
address of x = 0x10011d5c
foo called from main: return value = 24
address of x = 0x10011d5c
foo called from g: return value = 25
```

インラインとして定義された関数の各インスタンスが別個の関数として処理されていることを確認したければ、各変換単位の関数定義に static 指定子を使用することができ、また、**-qstaticinline** オプションを使用してコンパイルすることができます。ただし、静的インライン関数はテンプレートのインスタンス生成の際に名前ルックアップから除去され、検索できないことに注意してください。 C++

関連資料:

289 ページの『always_inline』

294 ページの『noinline』

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照

271 ページの『静的ストレージ・クラス指定子』

271 ページの『extern ストレージ・クラス指定子』

「XL C/C++ コンパイラー・リファレンス」の中の『-qstaticinline』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qkeepinlines』を参照

_Noreturn 関数指定子

_Noreturn 関数指定子は、呼び出し元に戻らない関数を宣言します。この指定子を付けて関数を宣言すると、コンパイラーは、関数が戻る場合の動作を考慮せずに、コードをよりよく最適化できます。main を除くすべての関数は、_Noreturn 関数指定子を使用して宣言または定義することができます。

_Noreturn が有効な場合、__IBMC_NORETURN マクロは 1 と定義されます。

以下のコードは、_Noreturn 指定子を使用した関数定義を示しています。

```
_Noreturn void f () {
  abort();
}
```

注:

- C _Noreturn キーワードは、**-qlanglvl=extc89**、**-qlanglvl=extc99**、**-qlanglvl=extended**、または **-qlanglvl=extc1x** コンパイラー・オプションを使用してコンパイルした場合に認識されます。
- C++ _Noreturn キーワードは、**-qlanglvl=extended**、**-qlanglvl=extended0x**、または **-qlanglvl=c1xnoreturn** コンパイラー・オプションを使用してコンパイルした場合に認識されます。
- さらに、_Noreturn 関数指定子を使用して、戻ることがない独自の関数を定義することもできます。ただし、_Noreturn で宣言される関数は、以下のいずれかの関数を呼び出す必要があります。そうしない場合、関数は制御をそれぞれの呼び出し元に戻します。
  - abort
  - exit
  - _Exit
  - longjmp
  - quick_exit
  - thrd_exit

関数の戻りの型指定子

関数の結果は*戻り値* と呼ばれ、戻り値のデータ型は*戻りの型* と呼ばれます。

C++ 関数宣言および関数定義はすべて、それが実際に値を戻すかどうかに関係なく、戻りの型を指定しなければなりません。

C 関数宣言に戻りの型が指定されていない場合、コンパイラーは、暗黙の戻りの型 int を想定します。ただし、C99 に準拠するために、その関数が int を戻すかどうかに関係なく、すべての関数宣言および関数定義に戻りの型をユーザーが指定するべきです。

関数は、配列型および関数型を除いて、値の型を戻すように定義できます。これら 2 つの例外は、配列または関数にポインターを戻すことにより、処理しなければなりません。関数が値を戻さない場合、void が、関数宣言および定義での型指定子になります。

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 オブジェクトまたは const オブジェクトを指すポインターを戻すことはできます。

関数は、ユーザー定義型である戻りの型を持つことができます。次に例を示します。

```
enum count {one, two, three};
enum count counter();
```

C ユーザー定義型は、関数宣言内でも定義できます。C++ ユーザー定義型は、関数宣言内では定義できません。

```
enum count{one, two, three} counter();   // legal in C
enum count{one, two, three} counter();   // error in C++
```

C++ 参照を関数の戻りの型として使うこともできます。参照は、参照しているオブジェクトの左辺値を戻します。

関連資料:

63 ページの『型指定子』

関数からの戻り値

C 関数が戻りの型 void を持つものとして定義された場合、その関数は値を戻しません。C++ C++ では、戻りの型 void を持つものとして定義された関数、またはコンストラクターまたはデストラクターである関数は、値を戻してはなりません。

C 関数が void 以外の戻りの型を持つように定義された場合、その関数は値を戻します。C99 に厳密に準拠するコンパイルでは、戻りの型を定義された関数は、戻される値を入れる式が含まれていなければなりません。

C++ 戻りの型が定義されている関数には、戻される値を入れる式が含まれていなければなりません。

関数が値を戻す場合、その値は、それが定義されている関数の戻りの型に暗黙的に変換された後、return ステートメントを介して関数の呼び出し元に戻されます。以下のコードは、return ステートメントを含む関数定義を示しています。

```
int add(int i, int j)
{
  return i + j; // return statement
}
```

以下のコードで示すように、関数 add() を呼び出すことができます。

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

この例では、return ステートメントは、戻された型の変数を初期化します。変数 answer を int の値 30 で初期化しています。戻された式の型を、戻りの型と照らし合わせて検査します。必要に応じて、すべての標準型変換およびユーザー定義の型変換が適用されます。

関数が呼び出されるたびに、自動ストレージを持つその変数の新しいコピーが作成されます。関数が終了した後に、これらの自動変数のストレージは再利用されることがあるので、自動変数を指すポインター または参照は戻してはなりません。
C++ クラス・オブジェクトが戻された場合、そのクラスに copy コンストラクターまたはデストラクターがあるときには、一時オブジェクトを作成することがあります。

関連資料:

249 ページの『return ステートメント』

332 ページの『割り当ての多重定義』

334 ページの『添え字の多重定義』

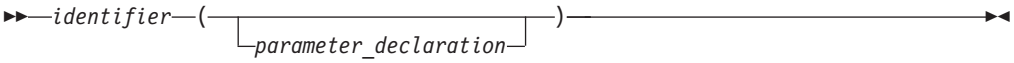
55 ページの『auto ストレージ・クラス指定子』

関数宣言子

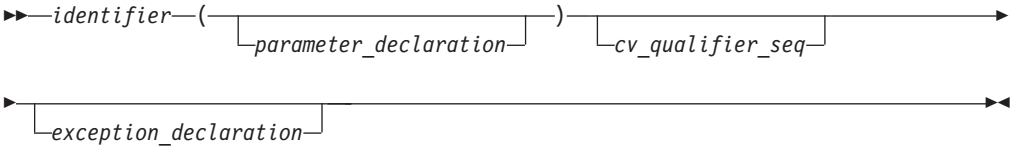
関数宣言子は以下のエレメントで構成されます。

- *ID*、すなわち、名前
- 『パラメーター宣言』。これは、関数呼び出しで関数に渡すことができるパラメーターを指定します。
- C++ 例外宣言。これは、throw 式を含みます。例外指定については、『 493 ページの『第 16 章 例外処理 (C++ のみ)』』で説明しています。 C++
- C++ *cv_qualifier_seq* は、const と volatile のいずれか、または両方を組み合わせたものを表しており、クラス・メンバー関数でのみ使用できます。定数および volatile メンバー関数について詳しくは、 358 ページの『定数および volatile メンバー関数』を参照してください。 C++

関数宣言子の構文 (C のみ)



関数宣言子の構文 (C++ のみ)



注: *identifier* の代わりに *direct_declarator* の構文を使用して、より複雑な型を形成する場合もあります。*direct_declarator* について詳しくは、 113 ページの『宣言子の概要』を参照してください。

関連資料:

304 ページの『C++ 関数内のデフォルトの引数 (C++ のみ)』

113 ページの『第 4 章 宣言子』

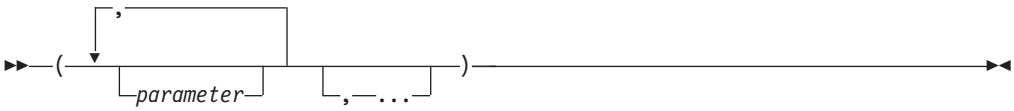
パラメーター宣言

関数宣言子は、関数が別の関数から呼び出されたとき、また自分自身で呼び出したとき、その関数に渡すことができるパラメーターのリストを含みます。

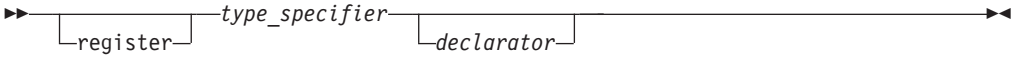
C++ C++ では、関数のパラメーター・リストは、関数のシグニチャーと呼ばれます。関数の名前とシグニチャーによって、関数は一意的に識別されます。コン

パイラーは、その言葉からも分かるように、関数シグニチャーによって、多重定義された関数の個別のインスタンスを見分けることができます。

関数仮パラメーター宣言の構文



パラメーター



C++ 関数宣言または関数定義の中の空の引数リストは、その関数は引数を受け取らないことを表します。関数が引数を取らないことを明示的に示すには、関数を次の 2 とおりの方法で宣言することができます。すなわち、空のパラメーター・リストを使用する方法と、キーワード `void` を使用する方法です。

```
int f(void);
int f();
```

C 関数定義 の中に空の引数リストがある場合、その関数が引数を取らないことを表します。関数宣言 の中の空の引数リストは、その関数は任意の数の引数、または任意の型の引数を取ることができることを表します。したがって、次の例は、

```
int f()
{
...
}
```

関数 `f` は引数を取らないことを表します。しかし、次の例は、

```
int f();
```

単に、パラメーターの数および型は不明であることを表しているだけです。関数が引数を取らないことを明示的に示すために、引数リストをキーワード `void` に置き換えることができます。

```
int f(void);
```

C

パラメーター指定の終わりにある省略符号は、関数が、可変数のパラメーターを持っていることを指定するために使用します。パラメーターの数は、パラメーター指定の数に等しいか、またはそれより多くなります。

```
int f(int, ...);
```

C++ 省略符号の前のコンマはオプションです。さらに、パラメーター宣言は、省略符号の前には必要ありません。

C 少なくとも 1 つのパラメーター宣言と省略符号の前のコンマは、C では両方とも必須です。

パラメーターの型

関数宣言、すなわち、プロトタイプでは、各パラメーターの型を指定しなければなりません。 C++ 関数定義 では、各パラメーターの型も指定しなければなりません。 C 関数定義 では、パラメーターの型が指定されていない場合、int と見なされます。

ユーザー定義型の変数を、次の例に示したように、パラメーター宣言で宣言することができます。この例では、x は、初めて宣言されています。

```
struct X { int i; };
void print(struct X x);
```

C ユーザー定義型は、パラメーター宣言内でも定義できます。 C++ ユーザー定義型はパラメーター宣言内では定義できません。

```
void print(struct X { int i; } x);   // legal in C
void print(struct X { int i; } x);   // error in C++
```

パラメーターの名前

関数定義 では、パラメーターの ID は必須です。関数宣言、すなわち、プロトタイプでは、ID の指定はオプションです。したがって、次の例は、関数宣言では有効です。

```
int func(int,long);
```

C++ 関数宣言でのパラメーターの名前の使用には、以下の制約が適用されます。

- 1 つの宣言内で 2 つのパラメーターが同じ名前を持つことはできません。
- パラメーターの名前が関数の外の名前と同じである場合、関数の外の名前は隠され、パラメーター宣言の中では使用できません。

  次の例では、3 番目のパラメーター名 intersects は、列挙型 subway_line をも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。最初のパラメーター名 subway_line が、名前空間スコープ enum 型を隠蔽し、2 番目のパラメーターで再度使用することができないからです。

  ```
  enum subway_line {yonge, university, spadina, bloor};
  int subway(char * subway_line, int stations, subway_line intersects);
  ```

C++

関数仮パラメーター宣言の中の静的配列指標 (C のみ)

一部のコンテキストの場合を除いて、添え字なし配列名 (例えば、 region であって、region[4] ではない) は、配列が既に宣言されている場合、値が配列の最初のエレメントのアドレスであるポインターを表します。関数のパラメーター・リスト内の配列型も、対応するポインター型に変換されます。配列が関数本体内からアクセスされると、引数配列のサイズに関する情報は失われます。

この情報は最適化に有用なので、この情報を保持しておくには、static キーワードを使用して引数配列の指標を宣言するとよいでしょう。定数式は、最適化の前提事

項として使用できる最小のポインター・サイズを指定します。 static キーワードの、この特殊な使用法は、厳しい規定があります。このキーワードは、最外部の配列型派生の中でのみ、そして関数仮パラメーター宣言でのみ使用できます。関数の呼び出し側がこのような制限に従っていない場合、動作は未定義です。

注: この機能は C99 固有です。

次の例は、この機能の使われ方を示しています。

```
void foo(int arr [static 10]);       /* arr points to the first of at least
                                           10 ints                          */
void foo(int arr [const 10]);        /* arr is a const pointer              */
void foo(int arr [static const i]);  /* arr points to at least i ints;
                                           i is computed at run time.       */
void foo(int arr [const static i]);  /* alternate syntax to previous example */
void foo(int arr [const]);           /* const pointer to int                */
```

関連資料:

56 ページの『静的ストレージ・クラス指定子』

123 ページの『配列』

199 ページの『配列添え字演算子 [ ]』

67 ページの『void 型』

63 ページの『型指定子』

101 ページの『型修飾子』

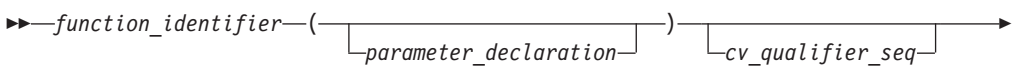
507 ページの『例外指定』

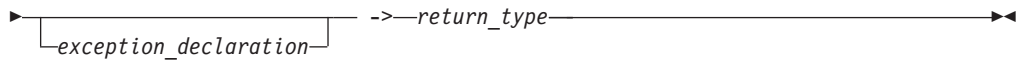
後置戻り型 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

後置戻り型機能は、戻りの型が関数引数の型に依存する場合は関数テンプレートの戻りの型を一般化できないという C++ の制限を取り除きます。例えば、a と b は関数テンプレート multiply(const A &a, const B &b) の引数であり、a と b の型は任意です。後置戻り型機能を使用しないと、a*b のすべてのケースを一般化するように multiply 関数テンプレートの戻りの型を宣言することができません。この機能を使用すると、関数引数の後に戻りの型を指定できます。これによって、関数テンプレートの戻りの型が関数引数の型に依存する場合のスコープの問題が解決されます。

後置戻り型の構文





注:

- この構文は、関数宣言または関数定義の構文ではありません。*return_type_specifier* が指定された宣言または定義の構文には、auto プレースホルダーが存在します。
- 後置戻り型を使用しない関数宣言子と同様に、この構文は関数へのポインターまたは参照の宣言に使用される場合があります。
- *function_identifier* の代わりに *direct_declarator* の構文を使用して、より複雑な型を形成する場合もあります。*direct_declarator* について詳しくは、113 ページの『宣言子の概要』を参照してください。

後置戻り型機能を使用するには、関数 ID の前に `auto` キーワードを指定して汎用的な戻りの型を宣言し、関数 ID の後に正確な戻りの型を指定します。例えば、`decltype` キーワードを使用して、正確な戻りの型を指定します。

次の例では、関数 ID `add` の前に `auto` キーワードが入っています。 `add` の戻りの型は `decltype(a + b)` であり、これは関数引数 `a` および `b` の型に依存します。

```
// Trailing return type is used to represent
// a fully generic return type for a+b.
template <typename FirstType, typename SecondType>
auto add(FirstType a, SecondType b) -> decltype(a + b){
   return a + b;
}

int main(){
   // The first template argument is of the integer type, and
   // the second template argument is of the character type.
   add(1, 'A');

   // Both the template arguments are of the integer type.
   add(3, 5);
}
```

注:

- 後置戻り型を使用する場合、プレースホルダーの戻りの型は `auto` でなければなりません。例えば、ステートメント `auto *f()->char` の場合、`auto *` はプレースホルダーの戻りの型として許可されないため、コンパイル時エラーとなります。
- `auto` 型指定子は、後置戻り型を持つ関数宣言子と共に使用できます。それ以外の場合、`auto` 型指定子は、auto 型推定機能に従って使用されます。 auto 型推定について詳しくは、91 ページの『auto 型指定子 (C++11)』を参照してください。関数宣言は auto 型推定に必要な初期化指定子を持つことができないため、関数宣言において後置戻り型なしで `auto` 型指定子を使用することはできません。関数へのポインターまたは参照の宣言の場合、`auto` 型指定子は、対応する後置戻り型または初期化指定子のいずれか一方と共に使用できます。関数へのポインターおよび参照について詳しくは、307 ページの『関数を指すポインター』を参照してください。
- 関数の戻りの型を、以下の型にすることはできません。
  - 関数

  - 配列
  - 不完全なクラス
- 関数の戻りの型で、以下の型を定義することはできません。
  - struct
  - クラス
  - 共用体
  - 列挙型

  ただし、関数宣言で型が定義されていない場合は、戻りの型を上記のいずれかの型にすることができます。

また、関数が複雑な戻りの型を持つ場合は、この機能によってプログラムがよりコンパクトで簡潔になります。この機能を使用しないと、プログラムが複雑になり、エラーが発生しやすくなる可能性があります。以下の例を参照してください。

```
template <class A, class B> class K{
   public:
      int i;
};

K<int, double> (*(*bar())())() {
   return 0;
}
```

後置戻り型機能を使用すると、コードをコンパクトにすることができます。以下の例を参照してください。

```
template <class A, class B> class K{
   public:
      int i;
};

auto bar()->auto(*)()->K<int, double>(*)(){
   return 0;
}
```

この機能は、クラスのメンバー関数にも使用できます。次の例では、後置戻り型を使用した後はメンバー関数 bar の戻りの型を修飾する必要がないため、プログラムが簡潔になります。

```
struct A{
   typedef int ret_type;
   auto bar() -> ret_type;
};

// ret_type is not qualified
auto A::bar() -> ret_type{
   return 0;
}
```

この機能のもう 1 つの使用法は、完全な転送関数を作成することです。つまり、転送関数は別の関数を呼び出し、転送関数の戻りの型は、呼び出された関数の戻りの型と同じになります。以下の例を参照してください。

```
double number (int a){
   return double(a);
}

int number(double b){
   return int(b);
```

```
}

template <class A>
auto wrapper(A a) -> decltype(number(a)){
   return number(a);
}

int main(){
   // The return value is 1.000000.
   wrapper(1);

   // The return value is 1.
   wrapper(1.5);
}
```

この例で、wrapper 関数と number 関数の戻りの型は同じです。

関連資料:

91 ページの『auto 型指定子 (C++11)』

93 ページの『decltype(*expression*) 型指定子 (C++11)』

555 ページの『C++11 互換性の拡張機能』

357 ページの『メンバー関数』

451 ページの『クラス・テンプレートのメンバー関数』

453 ページの『関数テンプレート』

264 ページの『関数宣言』

265 ページの『関数定義』

307 ページの『関数を指すポインター』

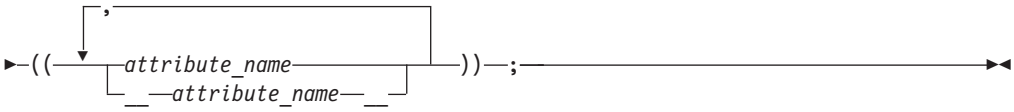
関数属性 (IBM 拡張)

関数の属性は、GNU C で開発されたプログラムの移植性を高めるためにインプリメントされた拡張機能です。関数に属性を指定することは、コンパイラーが関数呼び出しを最適化するのを支援し、コードをより多面的に検査するようにコンパイラーに指示する明示的な方法です。追加機能を提供する属性もあります。

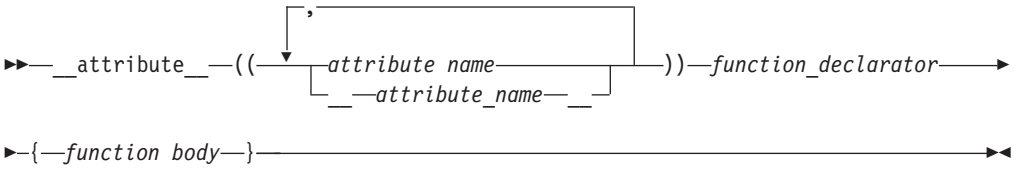
IBMC および C++ は、GNU C 関数属性のサブセットをインプリメントしています。 特定の関数属性がインプリメントされていない場合、その指定は受け入れられますが、セマンティクスは無視されます。これらの言語機能は、拡張言語レベルの 1 つでコンパイルされるとき、一括で使用可能になります。

関数属性は、キーワード __attribute__、その後に続く属性名、さらに、その属性名が必要とする追加引数によって指定されます。関数の __attribute__ 指定は、関数の宣言または定義に含まれます。構文の形式は次のとおりです。

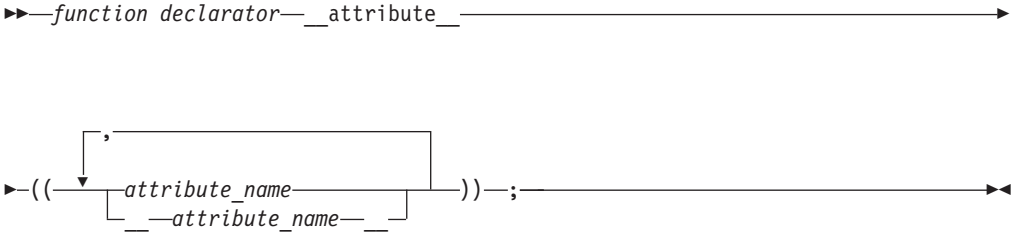
関数属性の構文: 関数宣言



関数属性の構文： 関数定義 (C のみ)



関数属性の構文： 関数定義 (C++ のみ)



関数宣言内の関数属性は、括弧に入れたパラメーター宣言を含めて、常に宣言子の後に書きます。

```
/* 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++ C++ では、属性指定も、関数に対して存在する可能性があるあらゆる例外宣言の後に行わなければなりません。 C++

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。

```
int __attribute__((individual_attribute_name)) foo(int i) { }
```

attribute_name は、前後に 2 つの下線文字を付けても付けなくても指定できます。ただし、2 つの下線文字を使用すると、同じ名前のマクロと名前が競合する可能性が低くなります。 C

IBM 次の関数属性がサポートされています。

- 別名
- always_inline
- const
- constructor および destructor
- format

- format_arg
- noinline
- noreturn
- pure
- section
- weak

IBM

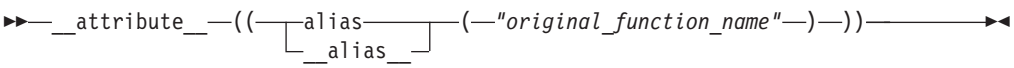
関連資料:

145 ページの『変数属性 (IBM 拡張)』

alias

alias 関数属性があると、関数宣言は、オブジェクト・ファイルの中で他のシンボルに対する別名として現れます。この言語機能は、重複した名前または扱いにくい名前に対処する手法として利用できます。

alias 関数属性の構文



C 別名関数は、その別名の指定後に、この関数属性で定義することができます。C は、同じコンパイル単位内に別名関数の定義がなくても、別名の指定を許可します。

次のコードは、func1 を __func2 の別名と宣言しています。

```
void __func2(){   /*  function body  */   }
void func1() __attribute__((alias("__func2")));
```

C

C++ *original_function_name* はマングルされた名前でなければなりません。

次のコードは、func1 を __func2 の別名と宣言しています。

```
extern "C" __func2(){   /*  function body  */   }
void func1() __attribute__((alias("__func2")));
```

C++

コンパイラーは、func1 の宣言と __func2 の定義の整合性の検査は行いません。そのような整合性の維持は、プログラマーが行うべきことです。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の #pragma weak を参照

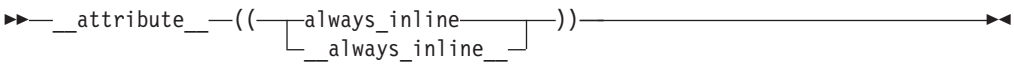
151 ページの『weak 変数属性』

always_inline

always_inline 関数属性は、関数をインライン化するようにコンパイラーに指示します。この関数は、以下のすべての条件が満たされるとインライン化することができます。

- 関数が、以下のいずれかの条件を満たすインライン関数である。
  - 関数が inline または __inline__ キーワードと一緒に指定されている。
  - オプション **-qinline+<function_name>** が指定されている。ここで *function_name* は、インライン化される関数の名前です。
  - C++ 関数がクラス宣言の範囲内で定義されている。 C++
- 関数が noinline または __noinline__ 属性と一緒に指定されていない。
- C 最適化レベルが **O2** 以上である。 C
- インライン化される関数の数が、コンパイラーがサポート可能なインライン関数の限度を超えていない。

always_inline 関数属性の構文



noinline 属性は、always_inline 属性よりも優先されます。 always_inline 属性は、インラインが使用可能である場合にのみ、インライン化コンパイラー・オプションよりも優先されます。 always_inline 属性は、インライン化が使用不可である場合には無視されます。

C++ コンパイラーは、関数が always_inline 属性と一緒に指定されている場合でも、仮想関数をインライン化できないことがあります。コンパイラーは、仮想関数がインライン化されていないことを示す通知メッセージは発行しません。

always_inline 属性を使用して指定された関数テンプレートを特殊化するときに、この属性はテンプレート仕様に伝搬されます。ここで、always_inline 属性をテンプレート仕様に適用した場合、重複した always_inline 属性は無視されます。以下の例を参照してください。

```
template<class T> inline __attribute__((always_inline)) T test( ){return (T)0;}

// The duplicate attribute "always_inline" is ignored.
template<> inline __attribute__((always_inline)) float test<float>(){return (float)0;}
```

C++

関連資料:

273 ページの『inline 関数指定子』

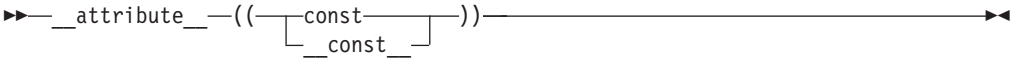
294 ページの『noinline』

const

const 関数属性は、その関数の呼び出し回数が、ソース・コードで指定された回数より少なくても問題ないことをコンパイラーに知らせます。この言語機能は、その

関数は引数以外の値を検査しないこと、およびその戻り値のため以外には効力を持たないことを明示的に指定できる機能であり、コンパイラーは、それによって、コードを最適化しやすくなります。

const 関数属性の構文



次の種類の関数には、const を宣言しないでください。

- 指し示されたデータを調べるポインター引数を持つ関数。
- const 関数以外の関数を呼び出す関数。

関連資料:

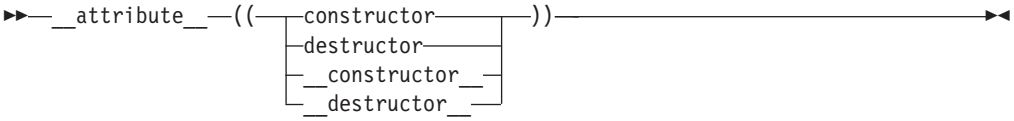
「XL C/C++ コンパイラー・リファレンス」の『-qisolated_call』を参照

constructor および destructor

constructor および destructor 関数属性は、データを初期化したり、プログラムの実行中に暗黙的に使用されるストレージを解放したりする関数を作成する機能を提供します。constructor 関数属性が適用されている関数は、実行が main に入る前に自動的に呼び出されます。同様に、destructor 属性が適用されている関数は、exit を呼び出した後、または main が完了した後に自動的に呼び出されます。

constructor または destructor 関数が自動的に呼び出されるときは、関数の戻り値は無視され、関数のパラメーターはどれも定義されません。

コンストラクターおよびデストラクターの関数属性の構文

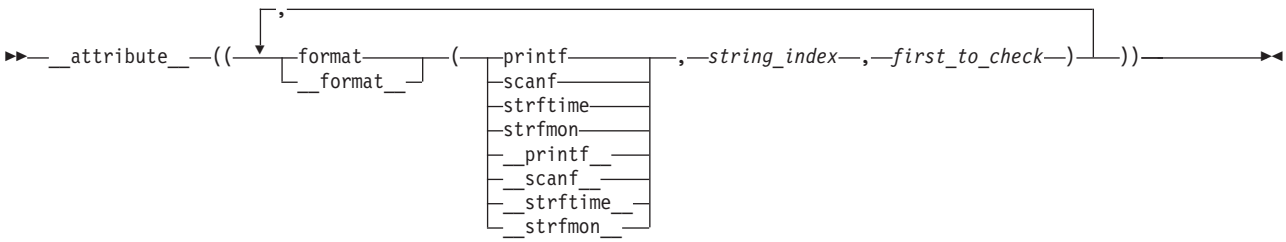


constructor または destructor 関数属性を含む関数宣言は、その他の宣言のすべてと一致していなければなりません。

format

format 関数属性を使用することにより、書式制御ストリングを引数として取るユーザー定義関数を識別することができます。これにより、これらの関数に対する呼び出しは、書式制御ストリングと対照して型検査されます。これは、コンパイラーが、関数 printf、scanf、strftime、および strfmon に対する呼び出しをエラーがないか検査する方法に似ています。

format 関数属性の構文



ここで、

string_index
: ユーザー関数の宣言内のどの引数が書式制御ストリングであるかを指定する定数整数式です。C++ C++ では、非静的メンバー関数の *string_index* の最小値は 2 です。これは、最初の引数が暗黙の this 引数であるためです。この振る舞いは、GNU C++ の振る舞いと一貫性があります。C++

first_to_check
: これは、書式制御ストリングに照らして検査する最初の引数を指定する定数整数式です。書式制御ストリングに照らして検査すべき引数がない場合 (つまり、書式制御ストリングの構文およびセマンティクスについてのみ診断を行う場合) は、*first_to_check* の値には 0 を指定します。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。

同じ関数について複数の format 属性を指定できます。その場合はそれらの属性がすべて適用されます。

```
void my_fn(const char* a, const char* b, ...)
       __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

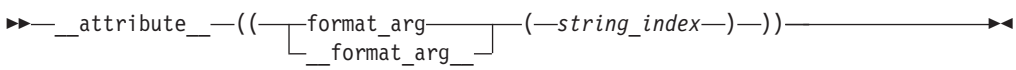
同じストリングを、いくつかの異なる書式スタイルについて診断することもできます。その場合は、すべてのスタイルが診断されます。

```
void my_fn(const char* a, const char* b, ...)
       __attribute__((__format__(__printf__,2,3),
                      __format__(__strftime__,2,0),
                      __format__(__scanf__,2,3)));
```

format_arg

format_arg 関数属性は、書式制御ストリングを変更するユーザー定義関数を識別するための手段として使用できます。該当するユーザー定義関数が識別されると、その関数に対する呼び出しをオペランドとして持つ printf、scanf、strftime、または strfmon などの関数について、エラーの有無を検査することができます。

format_arg 関数属性の構文



ここで、*string_index* は、どの引数が書式制御ストリング引数であるかを指定する定数整数式であり、値は 1 から始まります。C++ C++ での非静的メンバー関数の場合、*string_index* は 2 から始まります。これは、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です。C++

同じ関数について複数の format_arg 属性を指定でき、その場合はそのすべてが適用されます。

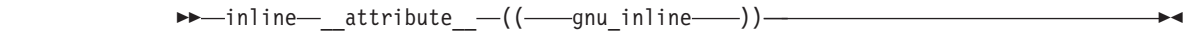
```
extern char* my_dgettext(const char* my_format, const char* my_format2)
          __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%","%"));
//printf-style format diagnostics are performed on both "%" strings
```

gnu_inline

gnu_inline 属性は、関数のインライン化動作を変更するようコンパイラーに指示します。この関数属性が使用されると、コンパイラーは C に対する GNU レガシー・インライン化拡張機能を模倣します。

この関数属性は、inline キーワード (__inline__、inline、__inline など) と組み合わせて使用する場合のみ、使用可能になります。

gnu_inline 関数属性の構文



注: inline キーワードと __inline__ キーワードのいずれと一緒に使用しても、gnu_inline 関数属性の動作は同じです。

C に対する GNU レガシー・インライン化拡張機能のセマンティクスは次のとおりです。

C

extern gnu_inline:

```
extern inline __attribute__((gnu_inline)) func() {...};
```

この func の定義は、インライン化にのみ使用されます。これは、スタンドアロン関数としてコンパイルされません。

static gnu_inline:

```
static inline __attribute__((gnu_inline)) func() {...};
```

この関数が生成される場合、関数は内部リンケージとともに生成されます。

plain gnu_inline:

```
inline __attribute__((gnu_inline)) func() {...};
```

この定義は、可能な場合、インライン化に使用されます。これは、スタンドアロン関数としてコンパイルされ (強い定義として出力され)、外部リンケージとともに出力されます。

C

C++

extern gnu_inline:

```
[extern] inline __attribute__((gnu_inline)) func() {...};
```

この func の定義は、インライン化にのみ使用されます。これは、スタンドアロン関数としてコンパイルされません。関数属性 gnu_inline でマークされたメンバー関数 (静的関数、およびリンケージを持たない関数を含む) は「extern」動作を持つことに注意してください。

static gnu_inline:

```
static inline __attribute__((gnu_inline)) func() {...};
```

この関数が生成される場合、関数は内部リンケージとともに生成されます。静的動作は非メンバーの静的関数にのみ適用されることに注意してください。

C++

gnu_inline 属性を指定するには、関数宣言で二重括弧で囲み、キーワード __attribute__ を使用します。以下の例を参照してください。

```
inline int func() __attribute__((gnu_inline));
```

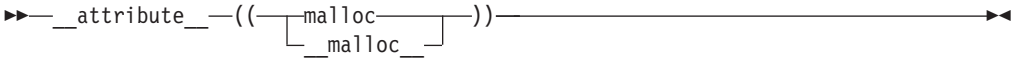
他の GCC 関数属性と同じように、属性名に付いている 2 つの連続する下線はオプションです。gnu_inline 属性は、同様に inline キーワードで宣言された関数と一緒に使用する必要があります。

malloc

関数属性 malloc を使用すると、関数が返す NULL ではないすべてのポインターは、他の有効なポインターの別名ではあり得ないとして、その関数を処理することをコンパイラーに指示できます。このタイプの関数 (malloc、calloc など) にはこの特性があるため、このような属性名になっています。サポートされるすべての属性と同じように、malloc は、特定のオプションまたは言語レベルを必要とせずにコンパイラーで受け入れられます。

malloc 関数属性を指定するには、関数宣言で二重括弧で囲み、キーワード __attribute__ を使用します。

malloc 関数属性の構文



他の GCC 関数属性と同じように、属性名に付いている 2 つの連続する下線はオプションです。

注:

- 関数によって返されるポインターが固有のストレージを指すことが確実でない限り、この関数属性を使用しないでください。そうしないと、最適化が実行されることよって実行時に誤った動作になる可能性があります。
- ポインターまたは C++ 参照の戻り型を返さない関数が、関数属性 malloc でマークされている場合は、警告が出され、この属性は無視されます。

例

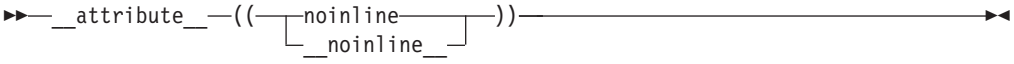
属性 malloc を使用した場合に最適化される、単純なケースを以下に示します。

```
#include <stdlib.h>
#include <stdio.h>
int a;
void* my_malloc(int size)  __attribute__ ((__malloc__))
{
  void* p = malloc(size);
  if (!p) {
    printf("my_malloc: out of memory!¥n");
    exit(1);
  }
  return p;
}
int main() {
  int* x = &a;
  int* p = (int*) my_malloc(sizeof(int));
  *x = 0;
  *p = 1;
  if (*x) printf("This printf statement to be detected as unreachable
                  and discarded during compilation process¥n");
  return 0;
}
```

noinline

noinline 関数属性を使用することにより、対象の関数について inline または non-inline のどちらが宣言されているかに関係なく、その関数がインライン化されないようにすることができます。この属性は、インライン化コンパイラー・ オプション、inline キーワード、および always_inline 関数属性より優先されます。

noinline 関数属性の構文



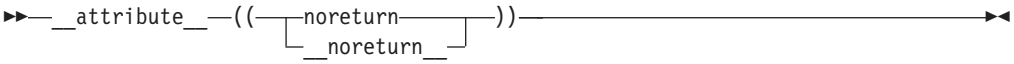
この属性はインライン化を防止するだけのものであり、インライン関数のセマンティクスは除去されません。

noreturn

noreturn 関数属性により、関数がその呼び出し元に制御を戻さないようコンパイラーに指示することができます。この言語機能は、コンパイラーがコードを最適化しやすくなるように、また初期化されていない変数に対する誤った警告を減らすように、プログラマーが明示的に指定できる機能です。

この関数の戻りの型は void でなければなりません。

noreturn 関数属性の構文



呼び出し元の関数によって保存されたレジスターは、戻らない関数を呼び出す前に必ず復元されるとは限りません。

関連資料:

「XL C/C++ コンパイラー・リファレンス」の #pragma leaves を参照

pure

pure 関数属性は、その関数の呼び出し回数が、ソース・コード明記された回数より少なくてもよい関数を宣言します。pure 属性を持つ関数を宣言することは、その関数は、パラメーター、グローバル変数、またはその両方に依存する戻り値以外は効力を持たないことを意味します。

pure 関数属性の構文

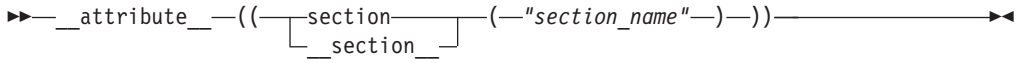
関連資料:

「XL C/C++ コンパイラー・リファレンス」の『-qisolated_call』を参照

section 関数属性

section 関数属性は、コンパイラーがその生成済みコードを配置する、オブジェクト・ファイル内のセクションを指定します。この言語機能を使用すると、関数が現れるセクションを制御できます。

section 関数属性の構文



ここで *section_name* はストリング・リテラルです。

定義済みの各関数は、1 つのセクションにのみ存在できます。関数定義に示されるこのセクションは、直前の宣言にあるセクションと一致していなければなりません。関数定義で示されているセクションは上書きできませんが、関数宣言で示されているセクションは後で指定するときに上書きできます。さらに、セクション属性が関数宣言に適用されていると、同じコンパイル単位に定義されている場合に限り、関数は指定されたセクションに配置されます。

関連資料:

149 ページの『section 変数属性』

used

関数がインライン・アセンブリーでのみ参照される場合は、used 関数属性を使用して、その関数が参照されていないように認識される場合でもその関数のコードを出力するようコンパイラーに指示できます。

used 関数属性を指定するには、関数宣言で二重括弧で囲み、キーワード __attribute__ を使用します。例えば、`int foo() __attribute__ ((__used__));` のようにします。他の GCC 関数属性と同じように、属性名に付いている 2 つの連続する下線はオプションです。

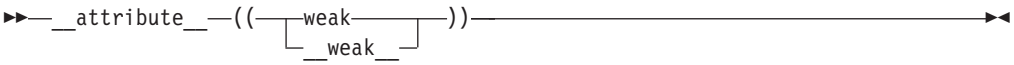
used 関数属性の構文

関数属性 gnu_inline が関数を破棄するように指定されていて、それと一緒に関数属性 used が指定されている場合、gnu_inline 属性が優先されて、関数定義は破棄されます。

weak

weak 関数属性があると、関数宣言の結果によるシンボルは、オブジェクト・ファイルの中に、グローバル・シンボルとしてではなく、弱いシンボルとして現れます。この言語機能は、ライブラリー関数を書くプログラマーが、ユーザー ・コード内の変数定義で、重複した名前エラーを起こすことなく、ライブラリー関数宣言をオーバーライドするための機能です。

weak 関数属性の構文



関連資料:


「XL C/C++ コンパイラー・リファレンス」の #pragma weak を参照

288 ページの『alias』

weakref

weakref は、オプションでターゲット名を指定できる関数宣言に付加される属性です。ターゲット名は、関数の宣言で属性 alias を通じて指定される場合もあります。

weakref 関数への参照は、ターゲット名の参照に変換されます。ターゲット名が現在の変換単位に定義されておらず、直接の参照も、あるいはターゲットの定義を必要とする方法での参照もされていない場合 (例えば、ターゲット名が weakref 関数の使用によってのみ参照される場合)、この参照は弱い参照です。現在の変換単位にターゲットの定義がある場合、weakref 関数の参照は当該の定義に直接解決されます。それ以外に、weakref 属性がターゲットの定義に影響を及ぼすことはありません。weakref 関数は内部リンケージを持っていなければなりません。

weakref 属性は、他の GCC 属性と同じく、以下のように前置構文または後置構文で表すことができます。

前置構文

```
static __attribute__((weakref("bar"))) void foo(void);
```

後置構文

```
static void foo(void) __attribute__((weakref("bar")));
```

weakref または alias 属性を持つ関数が、他の同様の関数を参照する場合があります。参照されている名前は、最後 (つまり非 weakref で非 alias) のターゲットの名前です。

規則

weakref 関数がキーワード static なしで宣言されると、GCC バージョン 4.3 以降でコンパイラーが構成されている場合、エラー・メッセージが出力されます。

weakref 関数宣言に指定されているターゲット名がそれ自身を直接的または間接的に指すことはできません。

ターゲット名を指定せずに weakref 属性を使用することは、推奨されていません。

前置構文の weakref 関数宣言で本体が提供される場合、この属性は無視されます。この状態を報告する警告メッセージが出されます。

例

以下の例は、weakref 関数のさまざまな宣言を示しています。

```
static void foo() __attribute__((weakref("bar")));

void foo() __attribute__((weakref("bar")));

static void foo() __attribute__((weakref,alias("bar")));

static void foo() __attribute__((alias("bar"),weakref));

static void foo() __attribute__((alias("bar")));
static void foo() __attribute__((weakref));

static void foo() __attribute__((alias("bar")));
void foo() __attribute__((weakref));

static void foo() __attribute__((weakref));
static void foo() __attribute__((alias("bar")));

static void foo() __attribute__((weakref));
void foo() __attribute__((alias("bar")));
```

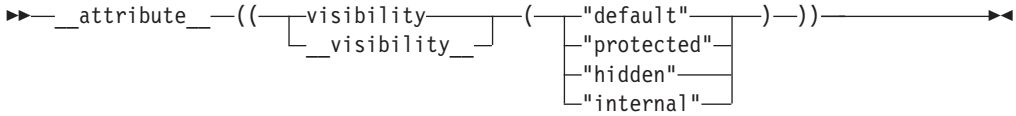
関連資料:

「XL C/C++ コンパイラー・リファレンス」の #pragma weak を参照

visibility

visibility 関数属性は、あるモジュール内で定義されている関数が、他のモジュール内で参照または使用できるかどうか、およびその方法を記述します。この機能を使用することで、共有ライブラリーを小さくして、シンボル競合の可能性を減らすことができます。詳しくは、「XL C/C++ 最適化およびプログラミング・ガイド」の『visibility 属性の使用』を参照してください。

visibility 関数属性の構文



例

以下の例で、関数 void f(int i, int j) の visibility 属性は隠されます。

```
void __attribute__((visibility("hidden"))) f(int i, int j);
```

関連資料:

9 ページの『外部リンケージ』

151 ページの『visibility 変数属性』

112 ページの『visibility 型属性 (C++ のみ)』

323 ページの『visibility 名前空間属性 (IBM 拡張)』

「XL C/C++ 最適化およびプログラミング・ガイド」の中の『visibility 属性の使用』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qvisibility』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qmkshrobj』 を参照

「XL C/C++ コンパイラー・リファレンス」の『#pragma GCC visibility push、#pragma GCC visibility pop (IBM 拡張)』を参照

main() 関数

プログラムが実行を開始すると、システムは main 関数を呼び出します。この関数は、プログラムのエントリー・ポイントを表します。デフォルトでは、main は、ストレージ・クラス extern を持ちます。どのプログラムにも、main という名前の関数が 1 つなければなりません。そして、次の制約が適用されます。

- プログラムにあるそれ以外の関数を main と呼ぶことはできません。
- main は、inline または static と定義することはできません。
- C++ main は、プログラム内から呼び出すことはできません。 C++
- C++ main のアドレスを取得することはできません。 C++
- C++ main 関数を多重定義することはできません。 C++
- C++ main 関数は、constexpr 指定子を指定して宣言することはできません。 C++11

関数 main は、次のいずれかの形式で、パラメーターを指定せずに、またはパラメーターを指定して定義することができます。

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

どのような名前もこれらのパラメーターに付けることはできますが、それらは通常、argc および argv と呼ばれています。最初のパラメーターの argc (引数カウント) は、プログラムが開始された時、コマンド行にいくつの引数が入力されたかを示す整数です。 2 番目のパラメーターの argv (引数ベクトル) は、文字オブジェ

クトの配列を指すポインターの配列です。配列オブジェクトは NULL 終了ストリングであって、プログラムが開始された時、コマンド行に入力された引数を表します。

この配列の最初のエレメント argv[0] は、コマンド行から実行されているプログラムのプログラム名または起動名が入っている文字配列を指すポインターです。argv[1] は、プログラムに渡される最初の引数を示し、argv[2] は 2 番目の引数などとなります。

次のプログラム例 backward は、コマンド行に入力された引数を、最後の引数が最初に印刷されるように印刷します。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
  printf("¥n");
}
```

以下によって、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
```

プログラムの先頭で、引数の argc と argv には以下の値が入ります。

オブジェクト	値
argc	3
argv[0]	ストリング "backward" を指すポインター
argv[1]	ストリング "string1" を指すポインター
argv[2]	ストリング "string2" を指すポインター
argv[3]	NULL

関連資料:

57 ページの『extern ストレージ・クラス指定子』

273 ページの『inline 関数指定子』

56 ページの『静的ストレージ・クラス指定子』

『関数呼び出し』

関数呼び出し

関数を宣言して定義すると、その関数をプログラム内の任意の場所から*呼び出す* ことができます。すなわち、main 関数内や別の関数から呼び出すことができ、さらにその関数自身から呼び出すことも可能です。関数を呼び出すには、関数名、その後に関数呼び出し演算子、および関数が受け取ることを期待するデータ値を指定します。これらの値は、関数に定義されたパラメーターの*引数* です。この処理は、関数への*引数の受け渡し* と呼ばれます。

以下の 3 つの方法で、呼び出された関数に引数を渡すことができます。

- 『値による受け渡し』。これは、引数の *値* を、呼び出された関数の対応するパラメーターにコピーします。
- 301 ページの『ポインターによる受け渡し』。これは、ポインター 引数を、呼び出された関数の対応するパラメーターに渡します。
- C++ 302 ページの『参照による受け渡し (C++ のみ)』。これは、引数の *参照* を、呼び出された関数の対応するパラメーターに渡します。

C++ クラスに、デストラクターまたはビット単位を超えるコピーを行うコピー・コンストラクターがある場合、クラス・オブジェクトを値で渡すと、実際には参照によって渡される一時オブジェクトが構築されます。

関数引数がクラス・オブジェクトであり、以下のすべての条件が真である場合は、コンパイラーがエラーを生成します。

- クラスが copy コンストラクターを必要としている。
- クラスに、ユーザー定義の copy コンストラクターがない。
- そのクラス用に copy コンストラクターを生成することができない。

C++

C 関数呼び出しは常に右辺値です。 C

C++ 関数呼び出しは、関数の結果の型に応じて、以下のいずれかの値のカテゴリーに属します。

- 結果の型が左辺値参照型である場合 C++11 または関数型に対する右辺値参照である場合 C++11 は左辺値
- C++11 結果の型がオブジェクト型に対する右辺値参照型である場合は xvalue C++11
- それ以外の場合は C++11 (prvalue) C++11 右辺値

C++

関連資料:

162 ページの『関数引数の変換』
175 ページの『関数呼び出し式』
411 ページの『コンストラクター』
165 ページの『左辺値と右辺値』
126 ページの『参照 (C++ のみ)』

値による受け渡し

値 による受け渡しを使用する場合、コンパイラーは、呼び出し側の関数の引数の値を、呼び出された関数定義の中のポインター以外または参照以外のパラメーターにコピーします。呼び出された関数のパラメーターは、渡された引数の値で初期化されます。そのパラメーターが定数と宣言されていなければ、パラメーターの値は変更できますが、変更は、呼び出された関数のスコープ内でのみ行われます。すなわち、この変更は、呼び出し側の関数の引数の値には、影響しません。

次の例では、main から func に、5 および 7 の 2 つの値が渡されます。関数 func は、これらの値のコピーを受け取り、ID a と b によって、それらにアクセスします。関数 func は、値 a を変更します。 main に制御が戻るときには、x と y の実際の値は、変わっていません。

```
/**
 ** This example illustrates calling a function by value
 **/

#include <stdio.h>

void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d¥n", a, b);
}

int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d¥n", x, y);
   return 0;
}
```

このプログラムの出力は、次のようになります。

```
In func, a = 12 b = 7
In main, x = 5  y = 7
```

ポインターによる受け渡し

ポインターによる受け渡し では、呼び出し元関数のポインター引数が、呼び出された関数の対応する仮パラメーターに受け渡されます。呼び出された関数は、ポインター引数が指す変数の値を変更できます。

以下の例は、ポインターによって引数がどのように渡されるかを示しています。

```
#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 の値は、ポインター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
```

ポインターによる受け渡しを使用する場合、ポインターのコピーが関数に渡されます。呼び出された関数内のポインターを変更する場合、ポインターのコピーが変更されるだけで、元のポインターは変更されず、元の変数を指し続けます。

ポインターによる受け渡しと値による受け渡しを比べた場合、呼び出された関数内のポインターによって受け渡された引数への変更が呼び出し元関数に影響を与えるのに対し、呼び出された関数内の値によって受け渡された引数への変更は呼び出し元関数に影響を与えません。呼び出し元関数の引数値を変更する場合は、ポインターによる受け渡しを使用します。それ以外の場合は、値による受け渡しを使用して引数を渡します。

関連資料:

116 ページの『ポインター』

参照による受け渡し (C++ のみ)

参照による受け渡し は、呼び出し側の関数内の引数の参照を、呼び出された関数の対応する仮パラメーターに渡すことを意味します。呼び出された関数は、渡された参照を使用して、引数の値を変更できます。

次の例は、参照によって引数がどのように渡されるかを示しています。この関数を呼び出すと、参照パラメーターが実引数によって初期化されます。

```
#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
```

const 修飾子によって修飾されている参照を変更するには、const_cast 演算子を使用して、その constness をキャストする必要があります。次に例を示します。

```
#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。

参照による受け渡しは、引数をコピーしないため、値による受け渡しよりも効率的です。仮パラメーターは、引数の別名です。呼び出された関数が仮パラメーターを読みとるかまたは書き込んだ場合、その関数は引数自体を実際に読み取るかまたは書き込みます。

参照による受け渡しと値による受け渡しの違いは、参照によって渡された引数に対して呼び出された関数で変更を加えると、それが呼び出し側の関数に影響しますが、値によって渡された引数に対して呼び出された関数で変更を加えた場合は、呼び出し側に影響はないという点です。呼び出し側の関数で引数値を変更したい場合は、参照による受け渡しを使用してください。それ以外の場合は、値による受け渡しを使用して引数を渡します。

参照による受け渡しとポインターによる受け渡しの違いは、ポインターは NULL とすることも、再割り当てすることも可能ですが、参照はそのようにすることはできないという点です。NULL が有効なパラメーター値である場合、またはポインターを再割り当てしたい場合は、ポインターによる受け渡しを使用してください。それ以外の場合は、定数参照または非定数参照を使用して引数を渡します。

関連資料:

126 ページの『参照 (C++ のみ)』

212 ページの『const_cast 演算子 (C++ のみ)』

割り振りおよび割り振り解除関数 (C++ のみ)

独自の new 演算子または割り振り関数を、クラス・メンバー関数やグローバル・ネームスペース関数として定義するには、以下の制約事項があります。

- 最初のパラメーターは、型 std::size_t でなければなりません。デフォルトのパラメーターを持つことはできません。
- 戻りの型は、型 void* でなければなりません。
- 割り振り関数はテンプレート関数でもかまいません。最初のパラメーターも、戻りの型もテンプレート・パラメーターに依存しません。
- 空の例外指定 throw() を指定して割り振り関数を宣言する場合、関数が失敗すると、割り振り関数は NULL ポインターを戻す必要があります。そうでない場合、関数が失敗すると、型 std::bad_alloc の例外、または std::bad_alloc から派生したクラスをスローする必要があります。

独自の delete 演算子または割り振り解除関数を、クラス・メンバー関数やグローバル・ネームスペース関数として定義するには、以下の制約事項があります。

- 最初のパラメーターの型は void* でなければなりません。
- 戻りの型は、型 void でなければなりません。
- 割り振り解除関数はテンプレート関数でもかまいません。最初のパラメーターも、戻りの型もテンプレート・パラメーターに依存しません。

以下に、グローバル・ネームスペース new と delete に置換関数を定義する例を示します。

```
#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;
}
```

上記の例の出力は、以下のとおりです。

```
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
```

関連資料:

218 ページの『new 式 (C++ のみ)』

C++ 関数内のデフォルトの引数 (C++ のみ)

関数仮パラメーターに対してデフォルト値を与えることができます。次に例を示します。

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

int a = 1;
int f(int a) { return a; }
int g(int x = f(a)) { return x; }

int h() {
  a = 2;
```

```
  {
    int a = 3;
    return g();
  }
}

int main() {
  cout << h() << endl;
}
```

この例では、標準出力に 2 を印刷します。その理由は、 g() の宣言で参照される a は、ファイル・スコープのもので、この値は、g() が呼び出されるときは 2 であるためです。

デフォルト引数は、暗黙的に、パラメーター型へ変換可能でなければなりません。

関数を指すポインターは、その関数と同じ型でなければなりません。関数の型を指定せずに参照によって関数のアドレスを得ようとすると、エラーになります。関数の型は、デフォルト値を持つ引数によって影響を受けません。

以下の例は、デフォルト引数が、関数の型の一部とは見なされていないことを示しています。デフォルト引数を使用すると、すべての引数を指定しなくても関数を呼び出すことができます。すべての引数の型を指定しない関数を指すポインターを作成することはできません。関数 f は、明示的引数がなくても呼び出すことができますが、ポインター badpointer は、引数の型を指定しないと定義することができません。

```
int f(int = 0);
void g()
{
   int a = f(1);                // ok
   int b = f();                 // ok, default argument used
}
int (*pointer)(int) = &f;       // ok, type of f() specified (int)
int (*badpointer)() = &f;       // error, badpointer and f have
                                // different types. badpointer must
                                // be initialized with a pointer to
                                // a function taking no arguments.
```

この例では、関数 f3 は、戻りの型 int を持っています。そして、その関数が取る int 引数は、関数 f2 から戻された値であるデフォルト値を持ちます。

```
      const int j = 5;
      int f3( int x = f2(j) );
```

関連資料:

307 ページの『関数を指すポインター』

デフォルト引数の制約事項 (C++ のみ)

演算子の中で、多重定義時にデフォルト引数を持つことができるのは、関数呼び出し演算子と演算子 new だけです。

デフォルト引数を持つパラメーターは、関数宣言パラメーター・リスト内の末尾のパラメーターでなければなりません。次に例を示します。

```
void f(int a, int b = 2, int c = 3);  // trailing defaults
void g(int a = 1, int b = 2, int c);  // error, leading defaults
void h(int a, int b = 3, int c);      // error, default in middle
```

いったん宣言または定義でデフォルト引数を指定すると、その引数を、同じ値にする場合であっても、再定義することはできません。ただし、それまでの宣言で指定されていないデフォルト引数を追加することはできます。例えば、次の例の最後の宣言では、a および b に対するデフォルト値を再定義しようとしています。

```
void f(int a, int b, int c=1);     // valid
void f(int a, int b=1, int c);     // valid, add another default
void f(int a=1, int b, int c);     // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

関数宣言または定義では、いかなるデフォルト引数値でも与えることができます。デフォルト引数値に続くパラメーター・リストの中のパラメーターはすべて、関数の、この宣言または前の宣言に指定されたデフォルト引数値を持っていなければなりません。

デフォルト引数の式では、ローカル変数を使用することはできません。例えば、コンパイラーは、次の g() および h() の両方の関数に対してエラーとします。

```
void f(int a)
{
      int b=4;
      void g(int c=a); // Local variable "a" cannot be used here
      void h(int d=b); // Local variable "b" cannot be used here
}
```

関連資料:

175 ページの『関数呼び出し式』

218 ページの『new 式 (C++ のみ)』

デフォルト引数の評価 (C++ のみ)

デフォルト引数を使用して定義された関数が、末尾の引数が欠落している状態で呼び出されると、デフォルト式が評価されます。次に例を示します。

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);            // same as call f(a,2,3)
f(a,10);         // same as call f(a,10,3)
f(a,10,20);      // no default arguments
```

デフォルト引数は、関数宣言に対して検査されます。そして、関数が呼び出されるときに評価されます。デフォルトの引数の評価の順序は、定義されていません。デフォルト引数式は、関数の他のパラメーターは使用できません。次に例を示します。

```
int f(int q = 3, int r = q); // error
```

q の値は、r に割り当てられるときには決まっていない場合があるので、引数 r を引数 q の値で初期化することはできません。上記の関数宣言を、次のように書き直すものとします。

```
int q=5;
int f(int q = 3, int r = q); // error
```

関数宣言内の r の値はやはりエラーとなります。それは、関数の外側で定義された変数 q が、関数に対して宣言されている引数 q によって隠されているからです。同様に、

```
typedef double D;
int f(int D, int z = D(5.3) ); // error
```

ここでは、型 D は、関数宣言内で整数の名前として解釈されます。型 D は、引数 D により隠されます。したがって、D が引数の名前であり、型ではないため、キャスト D(5.3) が、キャストとして解釈されません。

次の例では、非静的メンバー a を初期化指定子として使用できません。 a は、クラス X のオブジェクトが作成されるまで存在しないからです。 b は、クラス X のどのオブジェクトとも無関係に作成されるので、静的メンバー b を初期化指定子として使用することができます。デフォルト値は、クラス宣言の最後の大括弧 } の後まで分析されないので、メンバー b をデフォルト引数として使用した後で、それを宣言することができます。

```
class X
{
   int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

関数を指すポインター

関数を指すポインター

関数を指すポインター

関数を指すポインターは、その関数の実行可能コードのアドレスを示します。ポインターを使用して関数を呼び出し、関数を引数として他の関数に渡すことができます。関数を指す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、関数を指すポインターの型が決まります。

関数を指す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。関数仮パラメーターは、宣言のポインターよりも優先されるため、優先順位を変更して関数へのポインターを宣言するには、小括弧が必要です。小括弧がないと、コンパイラーはその宣言を、指定された戻り型へのポインターを返す関数として解釈します。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        */
```

最初の宣言では、f は、引数として int を取り、int を指すポインターを戻す関数として解釈されます。2 番目の宣言では、g は、int 引数を取り、int を戻す関数を指すポインターと解釈されます。

C++11

関数へのポインターの宣言または定義では、後置戻り型を使用できます。次に例を示します。

```
auto(*fp)()->int;
```

この例で、fp は int を返す関数へのポインターです。fp の宣言を、後置戻り型を使用しないように書き換えると、int (*fp)(void) になります。 後置戻り型につい

て詳しくは、 283 ページの『後置戻り型 (C++11)』を参照してください。

C++11

関数への参照

関数への参照とは、関数の別名、つまり代替名です。関数への参照はすべて、定義後に初期化する必要があります。関数への参照は、一度定義すると再割り当てできません。参照を使用して関数を呼び出し、関数を引数として他の関数に渡すことができます。次に例を示します。

```
int g();

// f is a reference to a function that has no parameters and returns int.
int bar(int(&f)()){

   // call function f that is passed as an argument.
   return f();
}

int x = bar(g);
```

C++11

関数の参照の宣言または定義でも、後置戻り型を使用できます。次の例で、`fp` は `int` を返す関数への参照です。後置戻り型について詳しくは、 283 ページの『後置戻り型 (C++11)』を参照してください。

```
auto(&fp)()->int;
```

C++11

関連資料:

10 ページの『言語リンケージ (C++ のみ)』

116 ページの『ポインター』

160 ページの『ポインター型変換』

126 ページの『参照 (C++ のみ)』

271 ページの『extern ストレージ・クラス指定子』

ネストされた関数 (IBM 拡張)

C ネストされた関数とは、他の関数の定義の中で定義されている関数です。ネストされた関数は、変数宣言を使用できる場所ならどこでも定義することができ、したがって、ネストされた関数の中にさらに関数をネストすることもできます。包含する関数の中で、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することができます。そうでない場合は、ネストされた関数は内部リンケージを持つことになります。この言語機能は C89 および C99 の拡張機能であって、GNU 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。

ネストされた関数は、その関数の定義より前にある、包含する関数のすべての ID にアクセスできます。

ネストされた関数は、包含する関数の終了後に呼び出すことはできません。

ネストされた関数は、goto ステートメントを使用して、包含する関数内のラベルへ、または包含する関数から継承された __label__ キーワードを宣言されたローカル・ラベルへジャンプすることはできません。 C

関連資料:

232 ページの『ローカルに宣言されたラベル (IBM 拡張)』

constexpr 関数 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

constexpr 指定子を指定して宣言される非コンストラクター関数は、constexpr コンストラクター関数です。 constexpr 関数は、定数式内で呼び出すことができる関数です。

constexpr 関数は、以下の条件を満たしている必要があります。

- 仮想ではない。
- 戻りの型がリテラル型である。
- その各パラメーターは必ずリテラル型である。
- 戻り値を初期化する際に、それぞれのコンストラクター呼び出しと暗黙的変換が定数式において有効である。
- その関数本体が = delete または = default である。そうでない場合、その関数本体は以下のステートメントのみを含んでいる必要があります。
  - null ステートメント
  - static_assert 宣言
  - クラスまたは列挙を定義していない typedef 宣言
  - using ディレクティブ
  - using 宣言
  - 1 つの return ステートメント

コンストラクターではない非静的メンバー関数が constexpr 指定子を指定して宣言されている場合、メンバー関数は定数であり、constexpr 指定子には関数型に対するその他の効果はありません。その関数がメンバーであるクラスは、リテラル型でなければなりません。

以下の例は、constexpr 関数の使用法を示しています。

```
const int array_size1 (int x) {
  return x+1;
}
```

```
// Error, constant expression required in array declaration
int array[array_size1(10)];

constexpr int array_size2 (int x) {
  return x+1;
}
// OK, constexpr functions can be evaluated at compile time
// and used in contexts that require constant expressions.
int array[array_size2(10)];

struct S {
  S() { }
  constexpr S(int) { }
  constexpr virtual int f() { // Error, f must not be virtual.
    return 55;
  }
};

struct NL {
  ~NL() { }              // The user-provided destructor (even if it is trivial)
                         // makes the type a non-literal type.
};

constexpr NL f1() {          // Error, return type of f1 must be a literal type.
  return NL();
}

constexpr int f2(NL) {       // Error, the parameter type NL is not a literal type.
  return 55;
}

constexpr S f3() {
  return S();
}

enum { val = f3() };         // Error, initialization of the return value in f3()
                             // uses a non-constexpr constructor.

constexpr void f4(int x) {   // Error, return type should not be void.
  return;
}

constexpr int f5(int x) {    // Error, function body contains more than return statement.
  if (x<0)
    x = -x;
  return x;
}
```

関数テンプレートが constexpr 関数として宣言された場合に、インスタンス化の結果として constexpr 関数の要件を満たさない関数が生成されるときには、constexpr 指定子は無視されます。次に例を示します。

```
template <class C> constexpr NL f6(C c) {    // OK, the constexpr specifier ignored
  return NL();
}
void g() {
  f6(55); // OK, not used in a constant expression
}
```

constexpr 関数への呼び出しは、あらゆる点で、同等の非 constexpr 関数への呼び出しと同じ結果を生みます。ただし、constexpr 関数への呼び出しは定数式内に指定できるという点が異なります。

constexpr 関数は暗黙的にインラインになります。

main 関数は、constexpr 指定子を指定して宣言することはできません。

関連資料:

99 ページの『constexpr 指定子 (C++11)』

174 ページの『一般化された定数式 (C++11)』

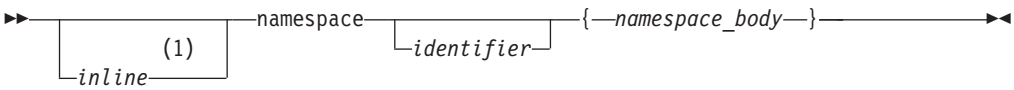
第 9 章 名前空間 (C++ のみ)

名前空間 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、名前空間内で名前を宣言します。ネスト・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、名前空間内で宣言された名前にアクセスできます。ただし、名前空間は、クラスまたは列挙型の追加機能を持ちません。名前空間の主要な目的は、追加の ID (名前空間の名前) を名前に追加することです。

名前空間の定義

名前空間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。

名前空間の構文



注:

1 この構文は、C++11 言語レベルでのみ有効です。

オリジナルの名前空間定義の中の *identifier* (ID) は、名前空間の名前です。オリジナルの名前空間定義が現れる宣言領域では、名前空間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、その名前空間は、*名前なし名前空間* です。

関連資料:

316 ページの『名前なし名前空間』

320 ページの『インライン名前空間定義 (C++11)』

名前空間の宣言

名前空間名に使用される ID は、固有である必要があります。前にグローバル ID として使用されていてはなりません。

```
namespace Raymond {
  // namespace body here...
  }
```

この例では、`Raymond` は、名前空間の ID です。名前空間のエレメントにアクセスする意図がある場合、名前空間の ID は、すべての変換単位で既知である必要があります。

関連資料:

3 ページの『ファイル/グローバル・スコープ』

名前空間の別名の作成

特定の名前空間 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、長い名前空間 ID を参照するのに有用です。

名前空間名または別名が、同じ宣言領域内の他のエンティティーの名前として宣言されると、コンパイル時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名前空間名が、プログラムのいずれかのグローバル・スコープ内の他のエンティティーの名前として宣言されると、コンパイル時エラーの結果を生じます。

関連資料:

3 ページの『ファイル/グローバル・スコープ』

ネストされた名前空間の別名の作成

名前空間定義は、宣言を持っています。名前空間定義は宣言そのものであるので、名前空間定義をネストすることができます。

ネストされた名前空間に、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、名前空間 NESTED_IBM_PRODUCT の別名です。この名前空間は、INTERNATIONAL_BUSINESS_MACHINES 名前空間内でネストされます。

関連資料:

『名前空間の別名の作成』

名前空間の拡張

名前空間は拡張可能です。前に定義された名前空間に、後続の宣言を追加できます。拡張部分は、オリジナルの名前空間定義から分離されたファイル、またはオリジナルの名前空間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 つの名前空間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。この名前空間には a、b、c、および d が含まれています。

名前空間と多重定義

複数の名前空間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:
int f(int);

// Original Y.h:
int f(char);

// Original program.c:
#include "X.h"
#include "Y.h"

int main(){
  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、名前空間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;

int main(){
  f('a');   // calls f() from Y.h
}
```

program.c で、main 関数は、名前空間 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、変更されないままになります。

関連資料:

325 ページの『第 10 章 多重定義 (C++ のみ)』

名前なし名前空間

左中括弧の前に ID のない名前空間は、*名前なし名前空間* になります。各変換単位は、それ自体の固有の名前なし名前空間を含むことができます。次の例は、名前なし名前空間がいかに有用であるかを示しています。

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   int variable;
   }

int main()
{
   cout << i << endl;
   variable = 100;
   return 0;
}
```

上記の例で、名前なし名前空間は、スコープ解決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。

名前なし名前空間を、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;
}
```

`main` の内部では、コンパイラーは、グローバル名と、同じ名前を持つ名前なし名前空間メンバーを区別できないので、`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、名前空間は 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、使用している名前空間を指定する必要があります。

名前なし名前空間は、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、同じ名前なし名前空間のメンバーです。

注: 名前なし名前空間で定義された項目は、内部リンケージを持っています。キーワード static を使用して、内部リンケージを持つ項目を定義するよりも、代わりに名前なし名前空間でそれらの項目を定義します。

関連資料:

8 ページの『プログラム・リンケージ』

8 ページの『内部リンケージ』

名前空間のメンバー定義

名前空間は、それ自体の内部で、または明示的修飾を使用して外部で、それ自体のメンバーを定義できます。以下に、名前空間が、内部的にメンバーを定義する例を示します。

```
namespace A {
  void b() { /* definition */ }
}
```

名前空間 A 内で、メンバー void b() が内部的に定義されます。

名前空間は、定義されようとしている名前に明示的修飾を使用して、外部的にそのメンバーを定義することもできます。定義されようとしているエンティティーは、名前空間内で既に宣言されている必要があります。定義は、宣言の名前空間を囲む名前空間内の、宣言のポイントの後に現れる必要があります。

以下に、名前空間が、外部的にメンバーを定義する例を示します。

```
namespace A {
  namespace B {
    void f();
  }
  void B::f() { /* defined outside of B */ }
}
```

この例では、関数 f() は、名前空間 B 内で宣言され、A 内で (B の外で) 定義されています。

名前空間とフレンド

最初に名前空間内で宣言された名前はすべて、その名前空間のメンバーです。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、最内部の囲み名前空間のメンバーです。

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、名前空間 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、コンパイル時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、最内部の囲み名前空間のメンバーです。このフレンド関数は、その名前空間によってのみアクセスすることができます。

関連資料:

372 ページの『フレンド』

using ディレクティブ

using ディレクティブは、すべての名前空間修飾子およびスコープ演算子へのアクセスを可能にします。これは、using キーワードを名前空間 ID に適用することによって行われます。

using ディレクティブの構文

name は、前に定義された名前空間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、例外を使用して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。

非修飾名参照では、スコープが、2 番目の名前空間を指名する using ディレクティブを含み、さらにその 2 番目の名前空間が別の using ディレクティブを含む場合、2 番目の名前空間の using ディレクティブは、1 番目のスコープ内にあるかのように動作します。

```
namespace A {
  int i;
}
namespace B {
  int i;
  using namespace A;
}
void f()
```

```
{
  using namespace B;
  i = 7; // error
}
```

この例で、関数 f() 内で i を初期化しようとすると、コンパイル時エラーを生じます。なぜなら、関数 f() は、どの i を呼び出すのか、つまり名前空間 A から i を呼び出すのか、または名前空間 B から i を呼び出すのか、を知らないからです。

関連資料:

389 ページの『using 宣言およびクラス・メンバー』

320 ページの『インライン名前空間定義 (C++11)』

using 宣言およびネームスペース

using 宣言は、特定の名前空間メンバーへのアクセスを可能にします。これは、対応する名前空間メンバーを持っている名前空間名に **using** キーワードを適用することによって行われます。

using 宣言の構文

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名前空間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 宣言の後に名前空間 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、名前空間 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、その特定の名前空間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、名前空間に含める必要があります。 using 宣言は、名前空間、ブロックおよびクラス・スコープに現れることができます。

関連資料:

389 ページの『using 宣言およびクラス・メンバー』

明示的アクセス

名前空間のメンバーを明示的に修飾するには、名前空間 ID を :: スコープ解決演算子と一緒に使用します。

明示的アクセス修飾の構文

次に例を示します。

```
namespace VENDITTI {
  void j()
};

VENDITTI::j();
```

この例では、スコープ解決演算子は、名前空間 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。スコープ解決演算子 :: は、グローバルおよびローカルの両方の名前空間内の 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、名前なし名前空間には適用できません。

関連資料:

173 ページの『スコープ解決演算子 :: (C++ のみ)』

インライン名前空間定義 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

インライン名前空間定義は、最初に inline キーワードを使用する名前空間定義です。このように定義された名前空間はインライン名前空間です。インライン名前空間のメンバーは、エンクロージング名前空間のメンバーでもあるかのように、定義および特殊化できます。

インライン名前空間定義の構文

インライン名前空間を定義すると、using ディレクティブがそのエンクロージング名前空間に暗黙的に挿入されます。エンクロージング名前空間で修飾名を参照するとき、その名前がエンクロージング名前空間で宣言されている場合でも、暗黙的 using ディレクティブによって、インライン名前空間のメンバーが取り込まれて検出されます。

例えば、USE_INLINE_B を定義して以下のコードをコンパイルすると、結果の実行可能モジュールの出力は 1 です。これを定義しない場合、出力は 2 になります。

```
namespace A {
#if USE_INLINE_B
   inline
#endif
   namespace B {
      int foo(bool) { return 1; }
   }
   int foo(int) { return 2; }
}

int main(void) {
   return A::foo(true);
}
```

インライン名前空間定義のプロパティーは推移的です。つまり、インライン名前空間のメンバーは、そのエンクロージング名前空間セットの名前空間のメンバーでもあるかのように使用できます。このエンクロージング名前空間セットは、インライン名前空間を囲む非インライン名前空間を最内部に持ち、これに、インライン名前空間が介在して構成されます。次に例を示します。

```
namespace L {
   inline namespace M {
      inline namespace N {
        / *...* /
      }
   }
}
```

この例では、名前空間 L は、インライン名前空間 M を含み、インライン名前空間 M は別のインライン名前空間 N を含みます。N のメンバーは、そのエンクロージング名前空間セット (つまり、L および M) の名前空間のメンバーであるかのようにも使用できます。

注:

- 名前空間 std をインライン名前空間として宣言しないでください。これは C++ 標準ライブラリー用の名前空間です。
- 名前空間が最初の定義でインラインでない場合は、それをインライン名前空間として宣言しないでください。
- 名前なし名前空間をインライン名前空間として宣言できます。

明示的インスタンス生成および特殊化でのインライン名前空間定義の使用

インライン名前空間の各メンバーは、そのエンクロージング名前空間のメンバーであるかのように、明示的にインスタンスを生成したり、特殊化したりできます。M などの名前空間での明示的インスタンス生成または特殊化の主テンプレートの名前参照は、M を含むエンクロージング名前空間セットのインライン名前空間を考慮します。以下に例を示します。

```
namespace L {
   inline namespace M {
       template <typename T> class C;
   }

   template <typename T> void f(T) { /*...*/ };
}

struct X { /*...*/ };
```

```
namespace L {
  template<> class C<X> { /*...*/ };    //template specialization
}

int main()
{
  L::C<X> r;
  f(r);  // fine, L is an associated namespace of C
}
```

この例では、M はそのエンクロージング名前空間 L のインライン名前空間であり、クラス C はインライン名前空間 M のメンバーであるため、L はクラス C の関連名前空間です。

明示的インスタンス生成および特殊化でインライン名前空間定義を使用する場合、以下の規則が適用されます。

- テンプレート名が修飾されている場合、明示的インスタンス生成は、主テンプレートのエンクロージング名前空間内に配置する必要があります。それ以外の場合は、明示的インスタンス生成は、主テンプレートの最近接エンクロージング名前空間またはエンクロージング名前空間セットの名前空間内に配置する必要があります。
- 明示的特殊化宣言は、主テンプレートの最近接エンクロージング名前空間またはエンクロージング名前空間セット内の名前空間の名前空間スコープで、最初に宣言する必要があります。その宣言が定義でない場合、後から任意のエンクロージング名前空間で定義することができます。

ライブラリー・バージョン管理でのインライン名前空間定義の使用

インライン名前空間定義を使用すると、複数のインプリメンテーションを持つライブラリーに対して、共通のソース・インターフェースを提供できるため、ライブラリーのユーザーは、1 つのインプリメンテーションを選択して、共通のインターフェースに関連付けることができます。以下の例は、明示的特殊化を含むライブラリー・バージョン管理でのインライン名前空間の使用法を示しています。

```
//foo.h
#ifndef SOME_LIBRARY_FOO_H_
   #define SOME_LIBRARY_FOO_H_
   namespace SomeLibrary
   {
      #ifdef SOME_LIBRARY_USE_VERSION_2_
         inline namespace version_2 { }
      #else
         inline namespace version_1 { }
      #endif
      namespace version_1 {
         template <typename T> int foo(T a) {return 1;}
      }
      namespace version_2 {
         template <typename T> int foo(T a) {return 2;}
      }
   }
#endif

//myFooCaller.C
#include <Foo.h>
#include <iostream>

struct MyIntWrapper { int x;};
```

```
//Specialize SomeLibrary::foo()
//Should specialize the correct version of foo()

namespace SomeLibrary {
   template <> int foo(MyIntWrapper a) { return a.x;}
}

int main(void) {
   using namespace SomeLibrary;
   MyIntWrapper intWrap = { 4 };
   std::cout << foo(intWrap) + foo(1.0) << std::endl;
}
```

SOME_LIBRARY_USE_VERSION_2_ を定義してこの例をコンパイルした場合、結果の実行可能モジュールの出力は 6 です。これを定義しない場合は、出力は 5 になります。関数呼び出し foo(intWrap) がいずれか 1 つのインライン名前空間により修飾されている場合、明示的特殊化を必ず有効にする必要があります。

関連資料:

313 ページの『名前空間の定義』

314 ページの『名前空間の拡張』

318 ページの『using ディレクティブ』

319 ページの『using 宣言およびネームスペース』

463 ページの『明示的インスタンス生成』

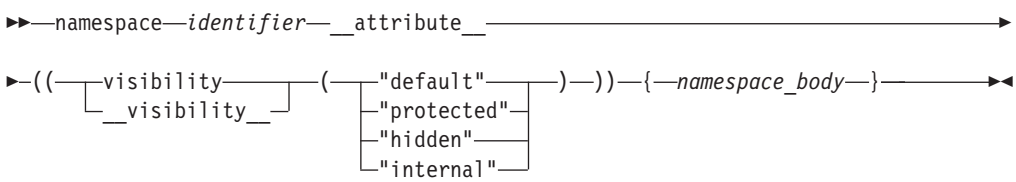
468 ページの『明示的特殊化』

555 ページの『C++11 互換性の拡張機能』

visibility 名前空間属性 (IBM 拡張)

visibility 名前空間属性は、あるモジュール内で定義されている名前空間内のエンティティーを、他のモジュール内で参照または使用できるかどうかおよびその方法について制御できる、言語拡張です。この機能を使用することで、共有ライブラリーを小さくして、シンボル競合の可能性を減らすことができます。詳しくは、「XL C/C++ 最適化およびプログラミング・ガイド」の『visibility 属性の使用』を参照してください。

visibility 名前空間属性の構文



属性名 visibility は、前後に 2 つの下線文字を付けても付けなくても指定できます。ただし、2 つの下線文字を使用すると、同じ名前のマクロと名前が競合する可能性が低くなります。

例

以下の例では、関数 fun() は名前空間 A で定義されており、fun() の visibility 属性はデフォルトです。

```
namespace A __attribute__((visibility("default"))) {
   void fun(){}
}
```

関連資料:

9 ページの『外部リンケージ』

151 ページの『visibility 変数属性』

visibility

112 ページの『visibility 型属性 (C++ のみ)』


「XL C/C++ 最適化およびプログラミング・ガイド」の中の『visibility 属性の使用』を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qvisibility』を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qmkshrobj』 を参照


「XL C/C++ コンパイラー・リファレンス」の『#pragma GCC visibility push、#pragma GCC visibility pop (IBM 拡張)』を参照

第 10 章 多重定義 (C++ のみ)

関数名または演算子に対して同じスコープ内で複数の定義を指定すると、その関数名または演算子を*多重定義した* ことになります。多重定義された関数および演算子については、それぞれ、『関数の多重定義』および 327 ページの『演算子の多重定義』で説明しています。

多重定義された宣言 は、同じスコープで前に宣言された宣言と同じ名前を使用して宣言された宣言です。ただし、両方の宣言は、異なる型を持っています。

多重定義関数名または演算子を呼び出す場合、コンパイラーは、関数または演算子を呼び出すのに使用した引数型を、定義に指定されているパラメーター型と比較することによって、使用するのに最も適切な定義を判別します。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、336 ページの『多重定義解決』での説明のように、*多重定義解決* と呼ばれています。

関数の多重定義

同じスコープ内で名前 `f` を持つ複数の関数を宣言することによって、関数名 `f` を多重定義することになります。 `f` の宣言は、型または引数リスト中の引数の数、またはその両方で、相互に異なっていなければなりません。 `f` という名前の多重定義された関数を呼び出すとき、関数呼び出しの引数リストを、名前 `f` を持つ多重定義された候補関数のそれぞれのパラメーター・リストと比較することによって、正しい関数が選択されます。*候補関数* とは、多重定義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");
}
```

上記の例の出力は、以下のとおりです。

```
Here is int 10
Here is float 10.1
Here is char* ten
```

ベクトル・パラメーター型に基づく関数多重定義がサポートされています。

関連資料:

『多重定義された関数の制約事項』

383 ページの『派生』

多重定義された関数の制約事項

以下の関数宣言は、同じスコープ内に現れても、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、以下の宣言を使用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使用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。

  ```
  void f(char*);
  void f(char[10]);
  ```

  パラメーターを差別化するときに、最初の配列次元が無意味で、他のすべての配列次元が有効であるもの。例えば、次の宣言は、同じ関数の宣言です。

  ```
  void g(char(*)[20]);
  void g(char[5][20]);
  ```

  次の 2 つの宣言は、等価ではありません。

  ```
  void g(char(*)[20]);
  void g(char(*)[40]);
  ```

- 一方は関数型、もう一方は同じ型の関数を指すポインターであることによってのみ異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

  ```
  void f(int(float));
  void f(int (*)(float));
  ```

- cv 修飾子 const、volatile、および restrict があるという点のみが異なるパラメーターを持つ関数宣言。この制約が適用されるのは、最外部のレベルのパラメーター型指定子の中に、これらの修飾子のいずれかが含まれている場合に限られます。例えば、次の宣言は、同じ関数の宣言です。

  ```
  int f(int);
  int f(const int);
  int f(volatile int);
  ```

  const 修飾子、volatile 修飾子および restrict 修飾子を、パラメーター型指定内で 適用する場合には、これらの修飾子を持つパラメーターを差別化できます。例えば、以下の宣言では、const および volatile は * ではなく int を修飾しています。したがって、これらの宣言は最外部のレベルのパラメーター型指定ではないので、それぞれ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数が異なっているということのみが異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

  ```
  void f(int);
  void f(int i = 10);
  ```

- extern "C" 言語リンケージおよび同じ名前を持つ、複数の関数 (それらのパラメーター・リストが異なっているかどうかに関係なく)。

関連資料:

319 ページの『using 宣言およびネームスペース』

89 ページの『typedef 定義』

101 ページの『型修飾子』

10 ページの『言語リンケージ (C++ のみ)』

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。

ベクトル型を含む演算子多重定義はサポートされていません。

多重定義された演算子は、*演算子関数* と呼ばれます。演算子の前にキーワード operator を置いて、演算子関数を宣言します。多重定義された演算子は、多重定義

された関数とは別個のものです。ただし、多重定義された関数と同様、演算子で使用されるオペランドの数と型によって区別されます。

標準の + (正) 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、complx 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 + (正) 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 complx
{
      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。

.	.*	::	?:

プリプロセッサー記号の # と ## は、多重定義できません。

演算子関数は、非静的メンバー関数であってもかまわないし、あるいは、クラス、クラスへの参照、列挙、または列挙型への参照であるパラメーターを少なくとも 1 つ持っている、非メンバー関数であってもかまいません。

演算子の優先順位、グループ分け、またはオペランドの数は変更できません。

多重定義された演算子 (関数呼び出し演算子を除く) は、引数リストの中にデフォルト引数や省略符号を入れることはできません。

多重定義された =、[]、()、および -> の各演算子は、第 1 オペランドとして必ず左辺値を受け取ることができるように、非静的メンバー関数として宣言する必要があります。

演算子 new、delete、new[]、および delete[] は、このセクションで説明する一般規則には従いません。

= 演算子を除く全演算子は継承されます。

単項演算子の多重定義

パラメーターを持たない非静的メンバー関数、またはパラメーターを 1 つ持っている非メンバー関数のいずれかを使用して、単項演算子を多重定義します。単項演算子 @ は、ステートメント @t の形で呼び出されるものと想定します。ここで、t は、型 T のオブジェクトです。この演算子を多重定義する非静的メンバー関数は、以下の形式になっています。

```
return_type operator@()
```

同じ演算子を多重定義する非メンバー関数は、以下の形式になっています。

```
return_type operator@(T)
```

多重定義された単項演算子は、どのような型でも戻すことができます。

次の例では、! 演算子を多重定義します。

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

struct X { };

void operator!(X) {
  cout << "void operator!(X)" << endl;
}

struct Y {
  void operator!() {
    cout << "void Y::operator!()" << endl;
  }
};

struct Z { };
```

```
int main() {
  X ox; Y oy; Z oz;
  !ox;
  !oy;
//  !oz;
}
```

上記の例の出力は、次のとおりです。

```
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、`!` 演算子がクラス `Z` に対して定義されていないので、`!oz` を許可しません。)

関連資料:

176 ページの『単項式』

増分および減分演算子の多重定義

1 個のクラス型の引数かクラス型への参照を持つ非メンバー関数演算子を使用して、あるいは引数のないメンバー関数演算子を使用して、前置増分演算子 (++) を多重定義します。

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 つの関数引数 (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 個の引数を持つ、メンバー関数演算子 `operator++()` を宣言することができます。コンパ

イラーは int 引数を使用して、前置増分演算子と後置増分演算子を区別します。暗黙の呼び出しの場合、デフォルト値はゼロです。

次に例を示します。

```
class X {
public:

  // member postfix x++
  void operator++(int) { };
};

class Y { };

// nonmember postfix y++
void operator++(Y&, int) { };

int main() {
  X x;
  Y y;

  // calls x.operator++(0)
  // default argument of zero is supplied by compiler
  x++;
  // explicit call to member postfix x++
  x.operator++(0);

  // calls operator++(y, 0)
  y++;

  // explicit call to non-member postfix y++
  operator++(y, 0);
}
```

前置減分演算子と後置減分演算子は、対応する増分演算子と同じ規則に従います。

関連資料:

177 ページの『増分演算子 ++』

178 ページの『減分演算子 --』

2 項演算子の多重定義

パラメーターを 1 つ持っている非静的メンバー関数、またはパラメーターを 2 つ持っている非メンバー関数のいずれかを使用して、2 項演算子を多重定義します。2 項演算子 @ は、ステートメント t @ u の形で呼び出されるものと想定します。ここで、t は、型 T のオブジェクト、u は、型 U のオブジェクトです。この演算子を多重定義する非静的メンバー関数は、以下の形式になっています。

return_type operator@(U)

同じ演算子を多重定義する非メンバー関数は、以下の形式になっています。

return_type operator@(T, U)

多重定義された 2 項演算子は、どのような型でも戻すことができます。

次の例では、* 演算子を多重定義します。

```
struct X {

  // member binary operator
  void operator*(int) { }
};
```

```
// non-member binary operator
void operator*(X, float) { }

int main() {
  X x;
  int y = 10;
  float z = 10;

  x * y;
  x * z;
}
```

呼び出し x * y は、x.operator*(y) と解釈されます。呼び出し x * z は、operator*(x, z) と解釈されます。

関連資料:

189 ページの『2 項式』

割り当ての多重定義

パラメーターを 1 つだけ持っている非静的メンバー関数を使用して、割り当て演算子 operator= を多重定義します。非メンバー関数である、多重定義された割り当て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、割り当て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)
}
```

割り当て x1 = x2 は、コピー割り当て演算子 X& X::operator=(X&) を呼び出します。割り当て x1 = 5 は、コピー割り当て演算子 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割り当て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。その結果、派生クラスのコピー割り当て演算子 (operator=) は、その基底クラスのコピー割り当て演算子を隠蔽します。

ただし、コピー割り当て演算子はいずれも、仮想として宣言できます。次の例は、このことを示しています。

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

struct A {
  A& operator=(char) {
    cout << "A& A::operator=(char)" << endl;
    return *this;
  }
  virtual A& operator=(const A&) {
    cout << "A& A::operator=(const A&)" << endl;
    return *this;
  }
```

```
};

struct B : A {
    B& operator=(char) {
      cout << "B& B::operator=(char)" << endl;
      return *this;
    }
    virtual B& operator=(const A&) {
      cout << "B& B::operator=(const A&)" << endl;
      return *this;
    }
};

struct C : B { };

int main() {
  B b1;
  B b2;
  A* ap1 = &b1;
  A* ap2 = &b1;
  *ap1 = 'z';
  *ap2 = b2;

  C c1;
//  c1 = 'z';
}
```

上記の例の出力は、以下のとおりです。

```
A& A::operator=(char)
B& B::operator=(const A&)
```

割り当て `*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。割り当て `*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割り当て演算子は、 `B& B::operator=(char)` を隠蔽するので、コンパイラーは、割り当て `c1 = 'z'` を許可しません。

関連資料:

435 ページの『コピー割り当て演算子』

189 ページの『割り当て演算子』

関数呼び出しの多重定義

関数呼び出し演算子は、多重定義された場合には、関数が呼び出される方法を変更しません。むしろ、演算子が与えられた型のオブジェクトに適用された場合の、演算子の解釈の仕方を変更します。

任意の数のパラメーターを持つ非静的メンバー関数を使用して、関数呼び出し演算子 `operator()` を多重定義します。あるクラスに対して関数呼び出し演算子を多重定義する場合、その宣言は以下の形式になります。

return_type `operator()(`*parameter_list*`)`

他のすべての多重定義された演算子と異なり、関数呼び出し演算子の引数リスト内に、デフォルト引数と省略符号を指定することができます。

次の例は、コンパイラーがどのように関数呼び出し演算子を解釈するかを示しています。

```
struct A {
  void operator()(int a, char b, ...) { }
  void operator()(char c, int d = 20) { }
};

int main() {
  A a;
  a(5, 'z', 'a', 0);
  a('z');
//  a();
}
```

関数呼び出し `a(5, 'z', 'a', 0)` は、`a.operator()(5, 'z', 'a', 0)` と解釈されます。これは `void A::operator()(int a, char b, ...)` を呼び出します。関数呼び出し `a('z')` は、`a.operator()('z')` と解釈されます。これは、`void A::operator()(char c, int d = 20)` を呼び出します。コンパイラーは、関数呼び出し `a()` を許可しません。なぜなら、その引数リストが、クラス `A` に定義された関数呼び出しパラメーター・リストのいずれにも一致しないからです。

次の例は、多重定義された関数呼び出し演算子を示しています。

```
class Point {
private:
  int x, y;
public:
  Point() : x(0), y(0) { }
  Point& operator()(int dx, int dy) {
    x += dx;
    y += dy;
    return *this;
  }
};

int main() {
  Point pt;

  // Offset this coordinate x with 3 points
  // and coordinate y with 2 points.
  pt(3, 2);
}
```

上記の例は、クラス `Point` のオブジェクトの関数呼び出し演算子を再解釈しています。 `Point` のオブジェクトを関数のように扱って、2 つの整数引数を渡す場合、関数呼び出し演算子は、渡した引数の値を、それぞれ `Point::x` および `Point::y` に加えます。

関連資料:

175 ページの『関数呼び出し式』

添え字の多重定義

パラメーターを 1 つだけ持っている非静的メンバー関数を使用して、`operator[]` を多重定義します。次の例は、多重定義された添え字演算子を持っている単純配列クラスです。多重定義された添え字演算子は、ユーザーが指定された境界の外で配列にアクセスしようとすると、例外をスローします。

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

template <class T> class MyArray {
private:
  T* storage;
  int size;
public:
  MyArray(int arg = 10) {
    storage = new T[arg];
    size = arg;
  }

  ~MyArray() {
    delete[] storage;
    storage = 0;
  }

  T& operator[](const int location) throw (const char *);
};

template <class T> T& MyArray<T>::operator[](const int location)
  throw (const char *) {
    if (location < 0 || location >= size) throw "Invalid array access";
    else return storage[location];
}

int main() {
  try {
     MyArray<int> x(13);
     x[0] = 45;
     x[1] = 2435;
     cout << x[0] << endl;
     cout << x[1] << endl;
     x[13] = 84;
  }
  catch (const char* e) {
    cout << e << endl;
  }
}
```

上記の例の出力は、次のとおりです。

```
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。

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とき、または別のオブジェクトを指すためにポインターが使用されるとき)、あるいは参照カウント (同じオブジェクトを指すスマート・ポインターの数をカウントし、そして、そのカウントがゼロになったときに、オブジェクトを自動的に削除する) などの別の作業を実行します。

auto_ptr と呼ばれるスマート・ポインターの 1 つの例が、C++ 標準ライブラリーに含まれています。 <memory> ヘッダーの中に、それを見出すことができます。 auto_ptr クラスは、自動オブジェクト削除をインプリメントします。

関連資料:

176 ページの『矢印演算子 ->』

多重定義解決

最も適切な多重定義された関数または演算子を選択するプロセスは、*多重定義解決* と呼ばれています。

f が、多重定義関数名であると想定します。多重定義された関数 f() を呼び出す場合、コンパイラーは*候補関数* のセットを作成します。この関数のセットには、f() を呼び出したポイントからアクセスできる、f という名前のすべての関数が含まれています。コンパイラーは、多重定義解決を促進するために、f という名前のこれらのアクセス可能な関数の中の 1 つの関数の代替表記を、候補関数として含めることができます。

候補関数のセットを作成した後、コンパイラーは、*実行可能関数* のセットを作成します。この関数のセットは、候補関数のサブセットです。各実行可能関数のパラメーター数は、f() を呼び出すのに使用した引数の数に一致します。

コンパイラーは、実行可能関数のセットから*最良の実行可能関数* を選択します。これは、f() を呼び出すときに C++ ランタイム環境が使用する関数宣言です。コンパイラーは、*暗黙的変換シーケンス* によって、これを行います。暗黙的変換シーケンスは、関数呼び出しの中の引数を、関数宣言の中の対応するパラメーターの型へ変換するのに必要な変換のシーケンスです。暗黙的変換シーケンスはランク付けされます。いくつかの暗黙的変換シーケンスは、他のものより良いシーケンスです。最良の実行可能関数は、そのすべてのパラメーターが、他の実行可能関数よりも良いか等しいランク付けの暗黙的変換シーケンスを持つ関数です。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。暗黙的変換シーケンスについては、 337 ページの『暗黙の変換シーケンス』でさらに詳しく説明しています。

可変長配列が関数仮パラメーターである場合は、候補関数の中で、左端の配列ディメンションが異なっていても違う関数とはみなされません。以下の例では、`void f(int [])` が既に定義されているため、2 番目の `f` の定義は許されません。

```
void f(int a[*]) {}
void f(int a[5]) {} // illegal
```

ただし、可変長配列が関数仮パラメーターであっても、その可変長配列内の左端以外の配列ディメンションが異なれば、候補関数は区別されます。例えば、関数 `f` の多重定義セットは、以下のように指定できます。

```
void f(int a[][5]) {}
void f(int a[][4]) {}
void f(int a[][g]) {}   // assume g is a global int
```

ただし、以下の定義を含めることはできません。

```
void f(int a[][g2]) {} // illegal, assuming g2 is a global int
```

この例では、第 2 レベルの配列ディメンション `g` および `g2` を持つ候補関数があるため、どの `f` 関数を呼び出すべきかに関してあいまいさが生じます。コンパイル時には、`g` も `g2` も認識されません。

明示的キャストを使用することによって、完全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

暗黙的変換シーケンス は、関数呼び出しの中の引数を、関数宣言の中の対応するパラメーターの型へ変換するのに必要な変換のシーケンスです。

コンパイラーは、各引数に対する暗黙的変換シーケンスを決定しようとします。コンパイラーは次に、各暗黙的変換シーケンスを 3 つのカテゴリーの中の 1 つにカテゴリー化し、カテゴリーに応じてそれらをランク付けします。コンパイラーは、引数に対する暗黙的変換シーケンスを検出できないようなプログラムは、許可しません。

以下に、変換シーケンスの 3 つのカテゴリーを、最良から最悪への順序で示します。

- 338 ページの『標準変換シーケンス』
- 338 ページの『ユーザー定義の変換シーケンス』
- 338 ページの『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、最高位から最下位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
- プロモーション: このランクには整数および浮動小数点プロモーションが含まれます。
- 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 - 整数型変換
  - ポインター型変換
  - ポインターからメンバーへの変換
  - ブール型変換

コンパイラーは、標準変換シーケンスを、その最低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のランクが、B の 2 番目の標準変換シーケンスのランクよりも良ければ、A は B より良いランクの変換シーケンスです。

省略符号変換シーケンス

省略符号変換シーケンス は、コンパイラーが関数呼び出しの中の引数を、対応する省略符号パラメーターに突き合わせるときに生じます。

暗黙的変換シーケンスのランク付け

標準変換シーケンスのランク付け

S1 および S2 が 2 つの標準変換シーケンスであるとします。コンパイラーは、S1 および S2 が以下の条件を満たすかどうか順に検査します。いずれかの条件が満たされる場合、S1 は S2 より良いランクの標準変換シーケンスです。

1. S2 では修飾変換が必要だが、S1 では修飾変換が不要である。例 1 を参照してください。
2. S1 のランクが S2 のランクより高い。例 2 を参照してください。
3. S1 と S2 の両方で修飾変換が必要である。T1 が S1 のターゲット型であり、T2 が S2 のターゲット型である。T2 が T1 よりも詳細に cv 修飾されている。例 3 を参照してください。
4. C++11 S1 および S2 が右辺値に対する参照バインディングであり、いずれも非静的メンバー関数の暗黙的オブジェクト・パラメーターを参照していない。S1 が右辺値参照をバインドし、S2 が左辺値参照をバインドする。 例 4 を参照してください。 C++11
5. C++11 S1 および S2 が参照バインディングである。S1 が左辺値参照を関数左辺値にバインドし、S2 が右辺値参照を関数左辺値にバインドする。例 5 を参照してください。 C++11
6. S1 および S2 が参照バインディングである。T1 が S1 によって参照されるターゲット型であり、T2 が S2 によって参照されるターゲット型である。T1 と T2 の相違点がトップレベルの cv 修飾子のみであり、T2 が T1 よりも詳細に cv 修飾されている。例 6 を参照してください。

2 つの標準変換シーケンス S1 および S2 のランクが同じであり、以下のいずれかの条件が満たされる場合は、S1 が S2 より良いランクの標準変換シーケンスです。

- S1 がポインター、メンバーを指すポインター、または NULL ポインターを変換するが、S2 はそのような変換を行わない。例 7 を参照してください。
- クラス A がクラス B の親クラスであり、S1 が B* から A* への変換であり、S2 が B* から void* への変換である。または S1 が A* から void* への変換であり、S2 が B* から void* への変換である。 例 8 を参照してください。
- クラス A がクラス B の親クラスであり、クラス B がクラス C の親クラスである。以下のいずれかの条件が満たされる。
  - S1 が C* から B* への変換であり、S2 が C* から A* への変換である。
  - S1 が型 C の式を型 B& の参照にバインドし、S2 が型 C の式を型 A& の参照にバインドする。
  - S1 が A::* から B::* への変換であり、S2 が A::* から C::* への変換である。
  - S1 が C から B への変換であり、S2 が C から A への変換である。
  - S1 が B* から A* への変換であり、S2 が C* から A* への変換である。 例 9 を参照してください。
  - S1 が型 B の式を型 A& にバインドし、S2 が型 C の式を型 A& にバインドする。
  - S1 が B::* から C::* への変換であり、S2 が A::* から C::* への変換である。
  - S1 が B から A への変換であり、S2 が C から A への変換である。

例 1

```
void f(int*);        // #1 function
void f(const int*);  // #2 function
```

```
void test() {
   // The compiler calls #1 function
   f(static_cast<int*>(0));
}
```

この例の f(static_cast<int*>(0)) の呼び出しにおいて、f(int*) 関数の標準変換シーケンス S1 は、int* から int* です。f(const int*) 関数の標準変換シーケンス S2 は、int* から const int* です。S2 では修飾の変換が必要ですが、S1 では不要です。そのため、S1 は S2 より良いランクの標準変換シーケンスです。

例 2

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

void f(const B*);  // #1 function
void f(A*);        // #2 function

void test() {
   // The compiler calls #1 function
   f(static_cast<B*>(0));
}

struct A1 *g(int);    // #3 function
struct A2 *g(short);  // #4 function

void test2() {
  // The compiler calls #4 function
  A2* a2 = g(static_cast<short>(0));

  // The compiler calls #3 function
  A1* a1 = g('¥0');
}
```

この例の f(static_cast<B*>(0)) の呼び出しにおいて、f(const B*) 関数の標準変換シーケンスは完全一致であり、f(A*) 関数の標準変換シーケンスは変換です。完全一致のランクは変換のランクより高いため、多重定義解決によって f(const B*) が選択されます。同様に、g(static_cast<short>(0)) の呼び出しにおいて、g(short) 関数の標準変換シーケンスは完全一致であり、g(int) 関数の標準変換シーケンスはプロモーションです。完全一致のランクはプロモーションのランクより高いため、g(short) 関数が呼び出されます。g('¥0') の呼び出しの場合、g(short) 関数の標準変換シーケンスは変換であり、g(int) 関数の標準変換シーケンスはプロモーションです。プロモーションのランクは変換のランクより高いため、この場合は g(int) が呼び出されます。

例 3

```
struct A { };
struct B : A { };
void g(const A*);           // #1 function
void g(const volatile A*);  // #2 function
void test2() {
   // The compiler calls #1 function
   g(static_cast<B*>(0));
}
```

この例の g(static_cast<B*>(0)) の呼び出しにおいて、g(const A*) 関数の標準変換シーケンス S1 は B* から const A* です。g(const volatile A*) 関数の標準変換シーケンス S2 は B* から const volatile A* です。S1 と S2 は両方とも修飾変換が必要であり、const volatile A* は const A* よりも詳細に cv 修飾されて

います。したがって、S1 は S2 より良いランクの標準変換シーケンスです。

C++11

例 4

```
double f1();
int g(const double&&);  // #1 function
int g(const double&);   // #2 function

// The compiler calls #1 function
int i = g(f1());

struct A {
   int operator+(int);
};

int operator+(A &&, int);

A &&f2();

void test() {
   f2() + 0;  // error
}
```

この例の g(f1()) の呼び出しにおいて、g(const double&) 関数の標準変換シーケンスは左辺値参照をバインドし、g(const double&&) の標準変換シーケンスは右辺値参照をバインドします。これらの 2 つの標準変換シーケンスは、いずれも非静的メンバー関数の暗黙のオブジェクト・パラメーターを参照しません。g(const double&&) 関数の標準変換シーケンスのランクがより良いため、この関数が呼び出されます。式 f2() + 0 の場合は、クラス・メンバーの候補が非静的メンバー関数の暗黙のオブジェクト・パラメーターの参照バインディングを必要とするため、名前空間スコープの候補を考慮した順序付けを行えません。

例 5

```
double f();
int g(double(&&)());  // #1 function
int g(double(&)());   // #2 function

// The compiler calls #2 function
int i = g(f)
```

この例の g(f) の呼び出しにおいて、g(double(&)()) 関数の標準変換シーケンスは左辺値参照を関数左辺値にバインドし、g(double(&&)()) 関数の標準変換シーケンスは右辺値参照を関数左辺値にバインドします。g(double(&)()) 関数の標準変換シーケンスのランクがより良いため、この関数が呼び出されます。

C++11

例 6

```
void f(A&);        // #1 function
void f(const A&);  // #2 function
void test() {
   A a;
   // The compiler calls #1 function
   f(a);
}
```

この例の f(a) の呼び出しにおいて、f(A&) 関数の標準変換シーケンス S1 は左辺値参照 A& を a にバインドし、f(const A&) 関数の標準変換シーケンス S2 は const 左辺値参照 const A& を a にバインドします。const A と A はトップレベルの cv 修飾子を除いて同じであり、const A は A よりも詳細に cv 修飾されているため、S1 が S2 より良いランクの標準変換シーケンスです。

例 7

```
void f(void*);  // #1 function
void f(bool);   // #2 function
void test() {
   // The compiler calls #1 function
   f(static_cast<int*>(0));
}
```

この例の f(static_cast<int*>(0)) の呼び出しにおいて、f(void*) 関数の標準変換シーケンス S1 は int* から void* であり、f(bool) 関数の標準変換シーケンス S2 は int* から bool です。S1 と S2 のランクは同じです。しかし、S1 はポインター、メンバーを指すポインター、または NULL ポインターを bool に変換しませんが、S2 はポインターを bool に変換するため、S1 は S2 より良いランクの標準変換シーケンスです。

例 8

```
//
void f(void*);      // #1 function
void f(struct A*);  // #2 function
struct A { };
struct B : A { };
void test() {
  // The compiler calls #2 function
  f(static_cast<B*>(0));
}
```

この例の f(static_cast<B*>(0)) の呼び出しにおいて、f(void*) 関数の標準変換シーケンスは B* から void* であり、f(struct A*) の標準変換シーケンスは B* から A* です。 f(struct A*) の標準変換シーケンスのランクがより良いため、この関数が呼び出されます。

例 9

```
void f(struct A *);
struct A { };
struct B : A { };
struct C : B { };
struct S {
   operator B*();
   operator C*();
}
void test() {
   // calls S::operator B*()
   f(S());
}
```

この例の f(S()) の呼び出しにおいて、標準変換シーケンスは S() から A* であり、構造体 S には 2 つの変換演算子があります。B* から A* への変換は C* から A* への変換よりランクが良いため、演算子関数 operator B* () が呼び出されます。

ユーザー定義の変換シーケンスのランク付け

U1 および U2 が 2 つのユーザー定義の変換シーケンスであるとします。U1 と U2 は同じユーザー定義変換関数、コンストラクター、または集合体初期化を使用します。U1 の 2 番目の標準変換シーケンスのランクが U2 の 2 番目の標準変換シーケンスのランクより良い場合は、U1 が U2 より良いランクのユーザー定義変換シーケンスです。例 10 を参照してください。

例 10

```
void f(void*);  // #1 function
void f(bool);   // #2 function
struct A {
   operator int*();
}
void test() {
   // The compiler calls #1 function
   f(A());
}
```

この例の f(A()) の呼び出しにおいて、f(void*) 関数のユーザー定義変換シーケンス U1 は A から void* です。f(bool) 関数のユーザー定義変換シーケンス U2 は A から bool です。U1 と U2 は、同じユーザー定義の変換 (A から int*) を使用します。 int* から void* への標準変換シーケンスは int* から bool への標準変換シーケンスよりランクが良いため、U1 は U2 より良いランクのユーザー定義変換シーケンスです。

関連資料:

160 ページの『左辺値から右辺値への変換』

160 ページの『ポインター型変換』

154 ページの『整数型変換』

155 ページの『浮動小数点変換』

155 ページの『ブール型変換』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

多重定義された関数のアドレスの解決

引数が指定されていない多重定義関数名 f を使用する場合、その名前は、関数、関数を指すポインター、メンバー関数を指すポインター、または関数テンプレートの特殊化を参照することができます。引数が指定されなかったので、コンパイラーは、関数呼び出しまたは演算子の使用に対して実行するのと同じ方法では、多重定義解決を実行することができません。その代わりに、コンパイラーは、f を使用した場所に応じて、以下の式のうちの 1 つの式の型に一致する、最良の実行可能関数を選択しようと試みます。

- 初期化しようとしているオブジェクトまたは参照
- 割り当ての左辺
- 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を選択した場合、コンパイラーは、宣言を型「ポインターから関数へ」または「参照から関数へ」の式に突き合わせました。コンパイラーが非静的メンバー関数の宣言を選択した場合、コンパイラーは、その宣言を型「ポインターからメンバー関数へ」の式に突き合わせました。次の例は、このことを示しています。

```
struct X {
  int f(int) { return 0; }
  static int f(char) { return 0; }
};

int main() {
  int (X::*a)(int) = &X::f;
//  int (*b)(int) = &X::f;
}
```

コンパイラーは、関数ポインター b の初期化を許可しません。型 int(int) の非メンバー関数または静的関数は、宣言されていません。

f がテンプレート関数の場合、コンパイラーは、テンプレート引数推論を実行して、どのテンプレート関数を使用するかを決めます。うまくいけば、その関数を実行可能関数のリストに追加します。このセットに、非テンプレート関数を含めて、複数の関数がある場合、コンパイラーは、セットからすべてのテンプレート関数を除去し、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。

関連資料:

307 ページの『関数を指すポインター』

361 ページの『メンバーを指すポインター』

453 ページの『関数テンプレート』

468 ページの『明示的特殊化』

第 11 章 クラス (C++ のみ)

クラス は、ユーザー定義のデータ型を作成するためのメカニズムの 1 つです。これは、C 言語の構造体のデータ型と似ています。 C では、構造体は、データ・メンバーのセットで構成されます。 C++ では、クラス型は C 構造体と似ていますが、クラスは、データ・メンバーのセット、およびそのクラスで実行できるオペレーションのセットで構成される点が異なります。

C++ において、クラス型は 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、デフォルトで private になります。クラスは、デフォルトで private で継承されます。
- キーワード struct を用いて宣言されたクラスのメンバーは、デフォルトでは public になります。構造体は、デフォルトで public で継承されます。
- (キーワード union を用いて宣言された) 共用体のメンバーは、デフォルトにより public になります。共用体は、派生における基底クラスとして使用することはできません。

クラス型を作成すると、1 つ以上のそのクラス型のオブジェクトを宣言することができます。次に例を示します。

```
class X
{
     /* define class members here */
};
int main()
{
      X xobject1;       // create an object of class type X
      X xobject2;       // create another object of class type X
}
```

C++ には、ポリモアフィック・クラスがあります。ポリモアフィズムにより、コンパイル時において関数が所属するクラスを正確に把握しなくても、別々のクラス (継承に関連した) に現れる関数名を使用することができます。

C++ では、多重定義の概念から、標準の演算子および関数を再定義することができます。演算子の多重定義により、組み込み (標準装備の) 型と同じように簡単にクラスを使用できるので、データ抽出が容易になります。

関連資料:

71 ページの『構造体および共用体』

355 ページの『第 12 章 クラスのメンバーとフレンド (C++ のみ)』

381 ページの『第 13 章 継承 (C++ のみ)』

325 ページの『第 10 章 多重定義 (C++ のみ)』

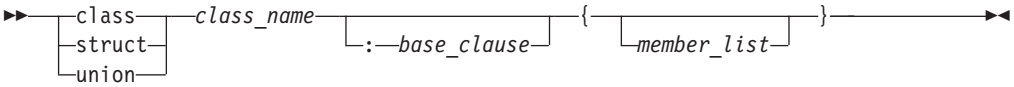
400 ページの『仮想関数』

クラス型の宣言

クラス宣言は、固有の型のクラス名を作成します。

クラス指定子 は、クラスを宣言する際に使用される型指定子です。そのクラスのメンバー関数がまだ定義されていない場合でも、クラス指定子が見つけられてそのメンバーが宣言されると、クラスは定義されていると見なされます。

クラス指定子の構文



class_name は、スコープ内で予約語となる固有の ID です。クラス名が宣言されると、そのクラス名は、囲みスコープ内にある同じ名前の他の宣言を隠蔽します。

member_list は、クラス *class_name* のクラス・メンバー (データと関数の両方) を指定します。クラスの *member_list* が空の場合、そのクラスのオブジェクトのサイズはゼロではありません。クラスのサイズが必要でなければ、クラス指定子自体の *member_list* 内で *class_name* を使用することができます。

base_clause は、クラス *class_name* がメンバーを継承する基底クラス (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 protected 非静的データ・メンバー、基底クラス、および仮想関数のないクラスです。

関連資料:

355 ページの『クラス・メンバー・リスト』

383 ページの『派生』

クラス・オブジェクトの使用

クラス型を使用して、そのクラス型のインスタンスつまりオブジェクト を作成することができます。例えば、クラス名の `X`、`Y`、および `Z` を指定して、それぞれクラス、構造体、および共用体を宣言することができます。

```
class X {
  // members of class X
};

struct Y {
  // members of struct Y
};
```

```
union Z {
  // members of union Z
};
```

これで、各クラス型のオブジェクトを宣言することができます。クラス、構造体、および共用体は、すべて C++ クラス型であることを忘れないでください。

```
int main()
{
      X xobj;      // declare a class object of class type X
      Y yobj;      // declare a struct object of class type Y
      Z zobj;      // declare a union object of class type Z
}
```

C++ では、C とは異なり、クラスの名前が隠れていない限り、クラス・オブジェクトの宣言の前に、キーワード union、struct、および class を入れる必要はありません。次に例を示します。

```
struct Y { /* ... */ };
class X { /* ... */ };
int main ()
{
      int X;             // hides the class name X
      Y yobj;            // valid
      X xobj;            // error, class name X is hidden
      class X xobj;      // valid
}
```

1 つの宣言で複数のクラス・オブジェクトを宣言すると、宣言子は個々に宣言されたように扱われます。例えば、以下のように、クラス S の 2 つのオブジェクトを単一の宣言で宣言する場合、

```
class S { /* ... */ };
int main()
{
      S S,T; // declare two objects of class type S
}
```

この宣言は、以下と等価です。

```
class S { /* ... */ };
int main()
{
      S S;
      class S T;       // keyword class is required
                       // since variable S hides class type S
}
```

しかし、以下と等価ではありません。

```
class S { /* ... */ };
int main()
{
      S S;
      S T;             // error, S class type is hidden
}
```

また、クラスへの参照、クラスを指すポインター、およびクラスの配列を宣言することもできます。次に例を示します。

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
int main()
```

```
{
      X xobj;
      X &xref = xobj;           // reference to class object of type X
      Y *yptr;                  // pointer to struct object of type Y
      Z zarray[10];             // array of 10 union objects of type Z
}
```

外部、静的、および自動定義内のクラスを初期化できます。初期化指定子には、= (等号) と、その後に続く、中括弧で囲まれ、コンマで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。

コピー制限のないクラス型のオブジェクトは、関数への引数として、割り当てまたは引き渡しを行うことができ、また関数により戻すことができます。

関連資料:

71 ページの『構造体および共用体』

126 ページの『参照 (C++ のみ)』

349 ページの『クラス名のスコープ』

クラスと構造体

C++ のクラスは、C 言語の構造体の拡張機能です。構造体とクラスの唯一の相違点は、デフォルトによるアクセスが、構造体メンバーは public アクセスで、クラス・メンバーは private アクセスであることです。したがって、キーワードの `class` または `struct` を使用して、等価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` と等価です。

```
class X {

  // private by default
  int a;

public:

  // public member function
  int f() { return a = 5; };
};

struct Y {

  // public by default
  int f() { return a = 5; };

private:

  // private data member
  int a;
};
```

構造体を定義してから、キーワード `class` を使用して、その構造体のオブジェクトを宣言すると、デフォルトによりそのオブジェクトのメンバーは、public のままです。以下の例において、`obj_X` が、クラス・キー `class` を使用する詳述型指定子の使用を宣言していますが、`main()` は、`obj_X` のメンバーへのアクセスを行います。

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

struct X {
```

```
 int a;
 int b;
};

class X obj_X;

int main() {
  obj_X.a = 0;
  obj_X.b = 1;
  cout << "Here are a and b: " << obj_X.a << " " << obj_X.b << endl;
}
```

上記の例の出力は、次のとおりです。

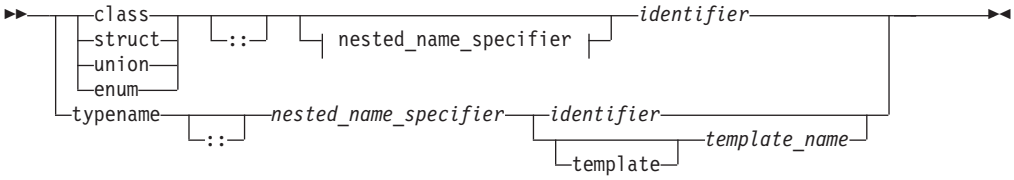
```
Here are a and b: 0 1
```

クラス名のスコープ

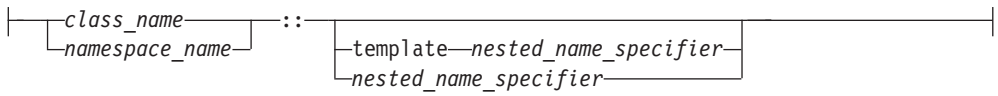
クラス宣言は、クラスが宣言されたスコープ内にクラス名を導入します。囲みスコープ内にある、その名前のクラス、オブジェクト、関数、または他の宣言はすべて隠蔽されます。

クラス名が、同じ名前を持つ関数、列挙子、またはオブジェクトと同じスコープ内で宣言される場合は、*詳述型指定子* を使用してそのクラスを参照してください。

詳述型指定子の構文



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。

関連資料:

『不完全なクラス宣言』

不完全なクラス宣言

不完全なクラス宣言 とは、クラス・メンバーを定義していないクラス宣言のことです。宣言が完全なものになるまでは、そのクラス型のオブジェクトを宣言したり、クラスのメンバーを参照することはできません。ただし、クラスのサイズが必要でなければ、不完全な宣言を使用して、クラスの定義の前にそのクラスに特定の参照を行うことはできます。

例えば、以下に示すように、構造体 `second` の定義で、構造体 `first` を指す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として構造体 `first` が宣言されています。構造体 `second` 内の `oneptr` の定義では、`first` のサイズは必要ありません。

```
struct first;           // incomplete declaration of struct first

struct second           // complete declaration of struct second
{
      first* oneptr;    // pointer to struct first refers to
                        // struct first prior to its complete
                        // declaration

      first one;        // error, you cannot declare an object of
                        // an incompletely declared class type
      int x, y;
};

struct first            // complete declaration of struct first
{
      second two;       // define an object of class type second
      int z;
};
```

しかし、空のメンバー・リストを指定してクラスを宣言すると、それは完全なクラス宣言となります。次に例を示します。

```
class X;                // incomplete class declaration
class Z {};             // empty member list
class Y
{
public:
      X yobj;           // error, cannot create an object of an
                        // incomplete class type
      Z zobj;           // valid
};
```

関連資料:

355 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、そのエンクロージング・クラスに対してローカルです。ポインター、参照、またはオブジェクト名を明示的に使用しない限り、ネスト・クラスでの宣言

で使用できるのは可視構成だけで、これには、エンクロージング・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、エンクロージング・クラスのメンバーへの特別なアクセス権を持ちません。エンクロージング・クラスのメンバー関数は、ネスト・クラスのメンバーへの特別なアクセスは行いません。次の例は、このことを示しています。

```
class A {
  int x;

  class B { };

  class C {

    // The compiler cannot allow the following
    // declaration because A::B is private:
    //   B b;

    int y;
    void f(A* p, int i) {

    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;

    }
  };

  void g(C* p) {

    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
  }
};

int main() { }
```

コンパイラーは、クラス `A::B` が private なので、オブジェクト `b` の宣言を許可しません。コンパイラーは、`A::x` が private なので、ステートメント `p->x = i` を許可しません。コンパイラーは、`C::y` が private なので、ステートメント `int i = p->y` を許可しません。

名前空間スコープで、ネスト・クラスのメンバー関数および静的データ・メンバーを定義することができます。例えば、以下のコードでは、修飾された型名を使用して、静的メンバー `x` と `y` およびネスト・クラス `nested` のメンバー関数 `f()` と `g()` にアクセスすることができます。修飾された型名を使用すると、`typedef` を定義して、修飾されたクラス名を表すことができます。その後、以下の例に示したように、`::` (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、そのエンクロージング・クラスの private メンバーから継承します。次の例は、このことを示しています。

```
class A {
private:
  class B { };
  B *z;

  class C : private B {
  private:
      B y;
//      A::B y2;
      C *x;
//      A::C *x2;
    };
};
```

ネスト・クラス A::C は A::B から継承します。コンパイラーは、A::B も A::C も private なので、宣言 A::B y2 および A::C *x2 を許可しません。

関連資料:

5 ページの『クラス・スコープ (C++ のみ)』

349 ページの『クラス名のスコープ』

370 ページの『メンバー・アクセス』

365 ページの『静的メンバー』

ローカル・クラス

ローカル・クラス は、関数定義の中で宣言されます。ローカル・クラスでの宣言が使用できるのは、外部変数および関数に加えて、囲みスコープからの型名、列挙、静的変数だけです。

次に例を示します。

```
int x;                          // global variable
void f()                        // function definition
{
      static int y;             // static variable y can be used by
                                // local class
      int x;                    // auto variable x cannot be used by
                                // local class
      extern int g();           // extern function g can be used by
                                // local class
```

```
      class local              // local class
      {
            int g() { return x; }      // error, local variable x
                                       // cannot be used by g
            int h() { return y; }      // valid,static variable y
            int k() { return ::x; }    // valid, global x
            int l() { return g(); }    // valid, extern function g
      };
}

int main()
{
      local* z;                // error: the class local is not visible
      // ...}
```

ローカル・クラスのメンバー関数は、定義される場合は、そのクラス定義の中で定義する必要があります。結果として、ローカル・クラスのメンバー関数は、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。

関連資料:

357 ページの『メンバー関数』

273 ページの『inline 関数指定子』

ローカル型名

ローカル型名は、他の名前と同じスコープ規則に従います。クラス宣言の中で定義される型名にはクラス・スコープがあり、修飾をしなければ、それらのクラスの外側で使用することはできません。

型名で使用されているクラス名、typedef 名、または定数名をクラス宣言で使用すると、その名前をクラス宣言で再定義することはできません。

次に例を示します。

```
int main ()
{
      typedef double db;
      struct st
      {
            db x;
```

```
            typedef int db; // error
            db y;
      };
}
```

次の宣言は有効です。

```
typedef float T;
class s {
      typedef int T;
      void f(const T);
};
```

したがって、関数 f() は型 s::T の引数を取ります。ただし、s のメンバーの順序が逆になっている以下の宣言ではエラーが発生します。

```
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。

関連資料:

2 ページの『スコープ』

89 ページの『typedef 定義』

第 12 章 クラスのメンバーとフレンド (C++ のみ)

このセクションでは、情報隠蔽メカニズムに関するクラス・メンバーの宣言と、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、フレンド・メカニズムの使用により、クラスがどのように認可するのかについて説明します。 C++ は、private インプリメンテーションを除く public クラス・インターフェースを持つという考え方を取り入れるように、情報隠蔽の概念を拡張しました。これは、プログラム内の関数による、クラス型の内部表記への直接アクセスを制限するためのメカニズムです。

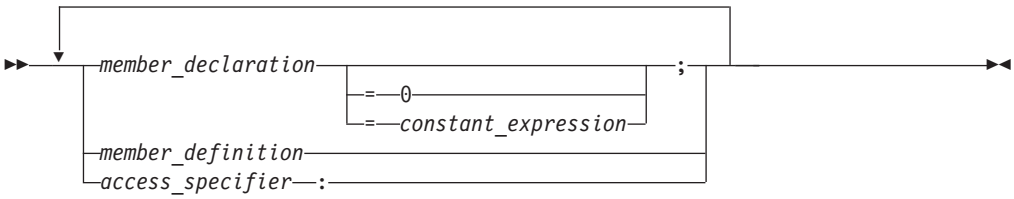
関連資料:

386 ページの『継承されるメンバー・アクセス』

クラス・メンバー・リスト

オプションのメンバー・リスト は、クラス・メンバー と呼ばれるサブオブジェクトを宣言します。クラス・メンバーとしては、データ、関数、ネストされた型、および列挙子が可能です。

クラス・メンバー・リストの構文



メンバー・リストは、クラス名の後に続き、中括弧の中に入れられます。以下が、メンバー・リストおよびメンバー・リストのメンバーに適用されます。

- *member_declaration* または *member_definition* は、データ・メンバー、メンバー関数、ネスト型、または列挙型の宣言または定義です。 (また、クラス・メンバー・リストに定義されている列挙型の列挙子も、クラスのメンバーです。)
- メンバー・リストは、ユーザーがクラス・メンバーを宣言できる唯一の場所です。
- フレンド宣言は、クラス・メンバーではありませんが、メンバー・リストに入っていることが必要です。
- クラス定義でのメンバー・リストには、クラスのメンバーをすべて宣言します。メンバーを他の場所で追加することはできません。
- メンバー・リストに、同じメンバーを 2 回宣言することはできません。
- データ・メンバー、またはメンバー関数を `static` として宣言できますが、`auto`、`extern`、または `register` としては宣言できません。
- ネスト・クラス、メンバー・クラス・テンプレート、またはメンバー関数を宣言し、クラスの外側でそれらを定義できます。

- 静的データ・メンバーは、クラスの外側で定義する必要があります。
- クラス・オブジェクトである非静的メンバーは、事前に定義されたクラスのオブジェクトでなければなりません。つまり、クラス A に A のオブジェクトを含めることはできませんが、クラス A のオブジェクトを指すポインターや参照を含めることはできます。
- 非静的配列メンバーの寸法は、すべて指定する必要があります。

定数初期化指定子 (= *constant_expression*) は、static と宣言された整数または列挙型のクラス・メンバー内にのみ現れます。

純粋指定子 (= 0) は、関数に定義がないことを示します。これは、virtual として宣言されたメンバー関数のみと一緒に使用され、メンバー・リスト内のメンバー関数の関数定義を置き換えます。

アクセス指定子 は、public、private、または protected のいずれかです。

メンバー宣言 は、宣言が入っているクラスのクラス・メンバーを宣言します。

access_specifier が分離する非静的クラス・メンバーの割り当て順は、インプリメンテーションに依存します。

access_specifier が分離する非静的クラス・メンバーの割り当て順は、インプリメンテーションに依存します。コンパイラーでは、クラス・メンバーが宣言された順序に従ってそれらを割り振ります。

A がクラスの名前だとします。以下のような A のクラス・メンバーは、A と異なる名前にする必要があります。

- すべてのデータ・メンバー
- すべての型のメンバー
- すべての列挙型メンバーの列挙子
- すべての無名共用体メンバーのメンバーすべて

関連資料:

346 ページの『クラス型の宣言』

370 ページの『メンバー・アクセス』

386 ページの『継承されるメンバー・アクセス』

365 ページの『静的メンバー』

データ・メンバー

データ・メンバーには、基本型だけでなく、ポインター、参照、配列型、ビット・フィールド、およびユーザー定義の型を含む他の型を用いて宣言されるメンバーが含まれます。データ・メンバーは、変数と同じ方法で宣言できますが、明示的初期化指定子をクラス定義の内部に設定することはできません。しかし、整数型または列挙型の const 静的データ・メンバーは、明示的初期化指定子を持つこともあります。

配列を非静的クラス・メンバーとして宣言する場合には、その配列のすべての次元を指定する必要があります。

クラスには、クラス型のメンバー、またはクラス型を指す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、不完全なクラス型を指すポインターであるメンバーを宣言することができます。

クラス X には、型 X のメンバーを入れることはできません。ただし、X へのポインター、X への参照、および X の静的オブジェクトは入れることができます。 X のメンバー関数は、型 X の引数を取り、X 型を戻します。例えば、次のようになります。

```
class X
{
      X();
      X *xptr;
      X &xlref;
      X &&xrref;
      static X xcount;
      X xfunc(X);
};
```

関連資料:

370 ページの『メンバー・アクセス』

386 ページの『継承されるメンバー・アクセス』

365 ページの『静的メンバー』

メンバー関数

メンバー関数 とは、クラスのメンバーとして宣言される演算子および関数のことです。メンバー関数には、friend 指定子を用いて宣言される演算子および関数は含まれません。これらは、クラスのフレンド と呼ばれます。メンバー関数を static として宣言できます。これは、*静的メンバー関数* と呼ばれます。 static として宣言されていないメンバー関数は、*非静的メンバー関数* と呼ばれます。

メンバー関数の定義は、そのエンクロージング・クラスのスコープ内にあります。メンバー関数の定義が、クラス・メンバー・リストのそのメンバーの宣言の前にある場合であっても、メンバー関数の本体は、クラス宣言の後で分析され、そのクラスのメンバーを、メンバー関数の本体で使用できるようにします。以下の例では、関数 add() が呼び出されるとき、データ変数の a、b、および c を add() の本体で使用することができます。

```
class x
{
public:
      int add()             // inline member function add
      {return a+b+c;};
private:
      int a,b,c;
};
```

C++11

後置戻り型をメンバー関数 (複雑な戻り型を持つものを含む) に使用できます。詳しくは、 283 ページの『後置戻り型 (C++11)』を参照してください。

C++11

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既に宣言している (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、クラス定義を囲む名前空間スコープ内に入れる必要があります。スコープ・レゾリューション (::) 演算子を使用して、メンバー関数名を修飾することもできます。

インライン・メンバー関数を宣言するのと同等な方法は、inline キーワードを指定してクラス内でその関数を宣言するか (そしてクラスの外側で関数を定義する)、または inline キーワードを使用して、クラス宣言の外側でその関数を定義することです。

次の例では、メンバー関数 Y::f() は、インライン・メンバー関数です。

```
struct Y {
private:
  char* a;
public:
  char* f() { return a; }
};
```

次の例は、直前の例と等価です。Y::f() が、インライン・メンバー関数です。

```
struct Y {
private:
  char* a;
public:
  char* f();
};

inline char* Y::f() { return a; }
```

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影響を与えません。リンケージは、デフォルトでは外部です。

ローカル・クラスのメンバー関数は、そのクラス定義の中で定義する必要があります。結果として、ローカル・クラスのメンバー関数は、暗黙的にインライン関数です。これらのインライン・メンバー関数には、リンケージはありません。

定数および volatile メンバー関数

const 修飾子を指定して宣言されたメンバー関数は、定数オブジェクトおよび非定数オブジェクトに対して呼び出すことができます。非定数メンバー関数は、非定数オブジェクトに対してのみ呼び出すことができます。同様に、volatile 修飾子を指定して宣言されたメンバー関数は、volatile オブジェクトおよび非 volatile オブジェクトに対して呼び出すことができます。非 volatile メンバー関数は、非 volatile オブジェクトに対してのみ呼び出すことができます。

関連資料:

362 ページの『this ポインター』

仮想メンバー関数

仮想メンバー関数は、キーワード virtual によって宣言されます。この関数を使用すると、メンバー関数の動的バインディングが可能となります。仮想関数は、すべてメンバー関数でなければならないため、仮想メンバー関数は、単に*仮想関数* と呼ばれます。

関数の宣言内の純粋指定子によって仮想関数の定義が置き換えられた場合、その関数は純粋を宣言されたと言います。純粋仮想関数を 1 つでも持つクラスは、*抽象クラス* と呼ばれます。

関連資料:

400 ページの『仮想関数』

406 ページの『抽象クラス』

特殊メンバー関数

特殊メンバー関数 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。これらには、以下のエレメントが含まれます。

- 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
- コピー割り当て演算子

これらの関数の詳しい説明については、409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』を参照してください。

メンバー・スコープ

クラス・メンバー・リスト内で既に宣言してあるが、定義はしていないメンバー関数と静的メンバーは、それらのクラス宣言の外側で定義することができます。非静的データ・メンバーは、それらのクラスのオブジェクトが作成されたときに定義されます。静的データ・メンバーの宣言は、定義ではありません。メンバー関数の宣言は、関数の本体も指定されている場合には、定義になります。

クラス・メンバーの定義がクラス宣言の外側にある場合、メンバー名は、:: (スコープ・レゾリューション) 演算子を使用して、クラス名によって修飾する必要があります。

以下の例では、クラス宣言の外側でメンバー関数を定義しています。

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

struct X {
  int a, b ;

  // member function declaration only
  int add();
};
```

```
// global variable
int a  = 10;

// define member function outside its class declaration
int X::add() { return a + b; }

int main() {
  int answer;
  X xobject;
  xobject.a = 1;
  xobject.b = 2;
  answer = xobject.add();
  cout << xobject.a << " + " << xobject.b << " = " << answer << endl;
}
```

この例の出力は 1 + 2 = 3 となります。

すべてのメンバー関数は、それらがクラス宣言の外側で定義されている場合であっても、クラス・スコープ内にあります。上記の例では、メンバー関数 add() はデータ・メンバー a を戻しますが、グローバル変数 a は戻しません。

クラス・メンバーの名前は、そのクラスに対してローカルなものです。いずれのクラス・アクセス演算子 (. (ドット)、-> (矢印)、または :: (スコープ・レゾリューション) も使用しない場合、クラス・メンバーを使用できるのは、そのクラスおよびネスト・クラスのメンバー関数内のみです。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. すべてのエンクロージング・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、エンクロージング・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。

メンバーを指すポインター

メンバーを指す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。メンバーを指す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ことが必要です。

メンバー関数を指すポインターは、関数を指すポインターと同じ方法で使用することができます。メンバー関数を指すポインターを比較し、値を割り当て、さらにそれらを使用してメンバー関数を呼び出すことができます。メンバー関数の型は、番号、引数の型、および戻りの型が同じ非メンバー関数と同じではないことに注意してください。

メンバーを指すポインターは、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とおりです。

```
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 がメンバーを指すポインターであると宣言することができます。メンバーを指すポインターは、以下のコード・フラグメントに示すように宣言し、使用することができます。

```
typedef int X::*my_pointer_to_member;
typedef void (X::*my_pointer_to_function) (int);

int main() {
  my_pointer_to_member ptiptr = &X::a;
  my_pointer_to_function ptfptr = &X::f;
  X xobject;
  xobject.*ptiptr = 10;
  cout << "The value of a is " << xobject.*ptiptr << endl;
  (xobject.*ptfptr) (20);
}
```

メンバーを指すポインター演算子 .* および ->* は、特定のクラス・オブジェクトのメンバーを指すポインターをバインドする際に用いられます。 () (関数呼び出し演算子) の優先順位の方が .* および ->* よりも高いため、ptf によって指示される関数を呼び出す際は小括弧を使用することが必要です。

メンバーを指すポインターの変換は、メンバーを指すポインターの変換が初期化、割り当て、または比較されるときに行われます。メンバーを指すポインターは、オブジェクトを指すポインターまたは関数を指すポインターと同じではありません。

this ポインター

キーワード this は、特定の型のポインターを識別します。クラス A の x という名前のオブジェクトを作成し、クラス A には、非静的メンバー関数 f() があるとします。関数 x.f() を呼び出す場合、f() の本体にあるキーワード this は、x のアドレスを保管します。this ポインターを宣言したり、またはそれに割り当てたりすることはできません。

静的メンバー関数は、this ポインターを持ちません。

クラス型 X のメンバー関数に対する this ポインターの型は、X* 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* です。

const this ポインターは、const メンバー関数内でのみ使用できます。そのクラスのデータ・メンバーは、その関数内で固定です。その場合でも、関数はその値を変更することができますが、そのためには、次のように const_cast が必要です。

```
void foo::p() const{
member = 1;                         // illegal
const_cast <int&> (member) = 1;   // a bad practice but legal
}
```

それよりも、member を mutable (可変) と宣言する方法のほうが良いでしょう。

メンバー関数が **volatile** 修飾子を用いて宣言されている場合、クラス X のそのメンバー関数に対する this ポインターの型は、volatile X* const です。例えば、コンパイラーは次のコードを許可しません。

```
struct A {
  int a;
  int f() const { return a++; }
};
```

コンパイラーは、関数 f() の本体で、ステートメント a++ を許可しません。関数 f() では、this ポインターは、A* const 型です。関数 f() は、this が指すオブジェクトの一部を変更しようとしています。

this ポインターは、すべての非静的メンバー関数呼び出しに隠れた引数として渡され、すべての非静的関数本体の中のローカル変数として使用することができます。

例えば、メンバー関数本体内で this ポインターを使用することによって、メンバー関数が呼び出される特定のクラス・オブジェクトを参照することができます。以下の例で示すコードによって作成される出力は、a = 5 です。

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

struct X {
private:
  int a;
public:
  void Set_a(int a) {

    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

メンバー関数 Set_a() では、ステートメント this->a = a は、this ポインターを使用して、自動変数 a によって隠された xobj.a を検索します。

クラス・メンバー名が隠蔽されていない限り、クラス・メンバー名の使用は、this ポインターとクラス・メンバー・アクセス演算子 (->) を用いたクラス・メンバー名の使用と同じです。

以下の表の最初の列は、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。

`#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();`
`}`
	`#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);`
	`}`

両方の例は、以下の出力を作成します。

```
abcdefgh
abcdefghijkl
```

関連資料:

332 ページの『割り当ての多重定義』

433 ページの『コピー・コンストラクター』

静的メンバー

クラス・メンバーは、クラス・メンバー・リストで、ストレージ・クラス指定子 static を使用して宣言することができます。プログラム中の 1 つのクラスのすべてのオブジェクトが、静的メンバーの 1 つのコピーのみを共有します。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();
}
```

関連資料:

358 ページの『定数および volatile メンバー関数』

355 ページの『クラス・メンバー・リスト』

静的メンバーでのクラス・アクセス演算子の使用

静的メンバーを参照するのに、クラス・メンバー・アクセス構文を使用する必要はありません。つまり、クラス X の静的メンバー s にアクセスするために、式 X::s が使用できます。以下の例は、静的メンバーへのアクセスを説明しています。

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

struct A {
  static void f() { cout << "In static function A::f()" << endl; }
};

int main() {

  // no object required for static member
  A::f();

  A a;
  A* ap = &a;
  a.f();
  ap->f();
}
```

ステートメント A::f()、a.f()、および ap->f() の 3 つはすべて、同じ静的メンバー関数 A::f() を呼び出します。

そのクラスの同じスコープ内、または静的メンバーのクラスから派生したクラスのスコープ内にある、静的メンバーを直接参照することができます。次の例は、後者のケースを説明しています (静的メンバーのクラスから派生したクラスのスコープ内にある静的メンバーを直接参照する)。

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

int g() {
   cout << "In function g()" << endl;
   return 0;
}

class X {
   public:
      static int g() {
         cout << "In static member function X::g()" << endl;
         return 1;
      }
};

class Y: public X {
   public:
      static int i;
};

int Y::i = g();

int main() { }
```

上記のコードの出力は、以下のとおりです。

```
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、グローバル名前空間に宣言されている関数、g() は呼び出しません。

関連資料:

56 ページの『静的ストレージ・クラス指定子』

173 ページの『スコープ解決演算子 :: (C++ のみ)』

176 ページの『ドット演算子 .』

176 ページの『矢印演算子 ->』

静的データ・メンバー

クラスのメンバー・リストにおける静的データ・メンバーの宣言は、定義ではありません。静的メンバーは、名前空間スコープのクラス宣言の外側で定義することが必要です。次に例を示します。

```
class X
{
public:
      static int i;
};
int X::i = 0; // definition outside class declaration
```

静的データ・メンバーを一度定義してしまえば、そのメンバーは、静的データ・メンバー・クラスのオブジェクトがなくても存在します。上記の例では、静的データ・メンバー X::i が定義されていても、クラス X のオブジェクトは、存在しません。

名前空間スコープのクラスの静的データ・メンバーには、外部リンケージがあります。静的データ・メンバー用の初期化指定子は、メンバーを宣言するクラスのスコープ内にあります。

静的データ・メンバーは、void、あるいは const または volatile で修飾された void を除く、あらゆる型とすることができます。静的データ・メンバーを mutable として宣言できません。

プログラム内には、静的メンバーの定義は 1 つしか入れられません。名前のないクラス、名前のないクラスに含まれるクラス、およびローカル・クラスは、静的データ・メンバーを持つことができません。

静的データ・メンバーとその初期化指定子は、そのクラスの、他の静的な private メンバーおよび protected メンバーにアクセスすることができます。以下の例は、他の静的メンバー (そのメンバーが private であっても) を使用して、どのように静的メンバーを初期化できるかを示しています。

```
class C {
      static int i;
      static int j;
      static int k;
      static int l;
      static int m;
      static int n;
      static int p;
      static int q;
      static int r;
      static int s;
      static int f() { return 0; }
      int a;
public:
      C() { a = 0; }
      };

C c;
int C::i = C::f();    // initialize with static member function
int C::j = C::i;      // initialize with another static data member
int C::k = c.f();     // initialize with member function from an object
int C::l = c.j;       // initialize with data member from an object
int C::s = c.a;       // initialize with nonstatic data member
int C::r = 1;         // initialize with a constant value

class Y : private C {} y;

int C::m = Y::f();    // error
int C::n = Y::r;      // error
int C::p = y.r;       // error
int C::q = y.f();     // error
```

C::m、C::n、C::p、および C::q の初期化ではエラーが発生します。これは初期化に使用される値が、アクセスできないクラス Y の private メンバーであるためです。

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。

C++11 リテラル型の静的データ・メンバーは、クラス定義の constexpr 指定子で宣言できます。データ・メンバー宣言は、定数初期化指定子を指定する必要があります。次に例を示します。

```
struct Constants {
  static constexpr int bounds[] = { 42, 56 };
};

float a[Constants::bounds[0]][Constants::bounds[1]];
```

C++11

定数初期化指定子は、定義ではないことに注意してください。まだ、囲み名前空間に静的メンバーを定義する必要があります。次の例は、このことを示しています。

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

struct X {
  static const int a = 76;
};

const int X::a;

int main() {
  cout << X::a << endl;
}
```

静的データ・メンバー a の最後に宣言されているトークン = 76 が、定数初期化指定子です。

関連資料:

9 ページの『外部リンケージ』

370 ページの『メンバー・アクセス』

352 ページの『ローカル・クラス』

静的メンバー関数

同じ名前、および引数の数と型が同じである、静的メンバー関数と非静的メンバー関数を持つことはできません。

静的データ・メンバーのように、クラス A のオブジェクトを使用しないで、クラス A の静的メンバー関数 f() にアクセスできます。

静的メンバー関数は、this ポインターを持ちません。次の例は、このことを示しています。

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

struct X {
private:
  int i;
  static int si;
```

```
public:
  void set_i(int arg) { i = arg; }
  static void set_si(int arg) { si = arg; }

  void print_i() {
    cout << "Value of i = " << i << endl;
    cout << "Again, value of i = " << this->i << endl;
  }

  static void print_si() {
    cout << "Value of si = " << si << endl;
    cout << "Again, value of si = " << this->si << endl; // error
  }

};

int X::si = 77;       // Initialize static data member

int main() {
  X xobj;
  xobj.set_i(11);
  xobj.print_i();

  // static data members and functions belong to the class and
  // can be accessed without using an object of class X
  X::print_si();
  X::set_si(22);
  X::print_si();
}
```

上記の例の出力は、以下のとおりです。

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

コンパイラーは、このメンバー関数が静的として宣言されていて、そのためメンバー関数が `this` ポインターを持っていないので、関数 `A::print_si()` で、メンバー・アクセス操作 `this->si` を認めません。

非静的メンバー関数の `this` ポインターを使用して、静的メンバー関数を呼び出すことができます。以下の例では、非静的メンバー関数の `printall()` が、`this` ポインターを使用して静的メンバー関数の `f()` を呼び出します。

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

class C {
  static void f() {
    cout << "Here is i: " << i << endl;
  }
  static int i;
  int j;
public:
  C(int firstj): j(firstj) { }
  void printall();
};

void C::printall() {
  cout << "Here is j: " << this->j << endl;
  this->f();
}

int C::i = 3;
```

```
int main() {
  C obj_C(0);
  obj_C.printall();
}
```

上記の例の出力は、以下のとおりです。

```
Here is j: 0
Here is i: 3
```

キーワード virtual、const、volatile、または const volatile を使用して、静的メンバー関数の宣言はできません。

静的メンバー関数がアクセスできるのは、その関数が宣言されているクラスの静的メンバー、列挙子、およびネストされた型の名前だけです。静的メンバー関数 f() が、クラス X のメンバーであるとします。静的メンバー関数 f() は、非静的メンバー X または基底クラス X の非静的メンバーにアクセスできません。

関連資料:

362 ページの『this ポインター』

メンバー・アクセス

メンバー・アクセス は、式または宣言内で、クラス・メンバーがアクセス可能かどうかを判別します。 x がクラス A のメンバーであるとします。このクラス・メンバー x は、以下のいずれかのレベルのアクセス可能性を持つものとして宣言することができます。

- public: x は、private や protected で定義したアクセス制限以外の場所なら、どこででも使用できます。
- private: x は、クラス A のメンバーとフレンドだけが使用できます。
- protected: x は、クラス A のメンバーとフレンド、およびクラス A から派生したクラスのメンバーとフレンドだけが使用できます。

キーワード class を指定して宣言されたクラスのメンバーのデフォルトは、private です。キーワード struct または union を指定して宣言されたクラスのメンバーのデフォルトは、public です。

クラス・メンバーのアクセスを制御するには、*アクセス指定子* public、 private、または protected をクラス・メンバー・リストのラベルとして使用します。次の例はこれらのアクセス指定子を示しています。

```
struct A {
  friend class C;
private:
  int a;
public:
  int b;
protected:
  int c;
};

struct B : A {
  void f() {
    // a = 1;
    b = 2;
    c = 3;
```

```
  }
};

struct C {
  void f(A x) {
    x.a = 4;
    x.b = 5;
    x.c = 6;
  }
};

int main() {
  A y;
//  y.a = 7;
  y.b = 8;
//  y.c = 9;

  B z;
//  z.a = 10;
  z.b = 11;
//  z.c = 12;
}
```

次の表は、上記の例のようなさまざまなスコープ内のデータ・メンバー A::a、A::b、および A::c へのアクセスについてリストしています。

スコープ	A::a	A::b	A::c
関数 B::f()	アクセス不可。メンバー A::a は、private です。	アクセス。メンバー A::b は、public です。	アクセス。クラス B は、A から継承します。
関数 C::f()	アクセス。クラス C は、A のフレンドです。	アクセス。メンバー A::b は、public です。	アクセス。クラス C は、A のフレンドです。
main() のオブジェクト y	アクセス不可。メンバー y.a は、private です。	アクセス。メンバー y.a は、public です。	アクセス不可。メンバー y.c は、protected です。
main() のオブジェクト z	アクセス不可。メンバー z.a は、private です。	アクセス。メンバー z.a は、public です。	アクセス不可。メンバー z.c は、protected です。

アクセス指定子は、次のアクセス指定子まで、またはクラス定義の終わりまでその後に続くメンバーのアクセス可能性を指定します。任意の数のアクセス指定子を、任意の順序で使用することができます。クラス定義内に後からクラス・メンバーを定義する場合、そのアクセス指定は、その宣言と同一にする必要があります。次の例は、このことを示しています。

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

コンパイラーは、クラス B が既に private として宣言されているため、このクラスの定義を許可しません。

クラス・メンバーには、それがそのクラス内、またはクラスの外側に定義されたかどうかには関係なく、同じアクセス制御があります。

アクセス制御は、名前に対応しています。特に、アクセス制御を typedef 名に追加する場合、それは typedef 名だけに影響します。次の例は、このことを示しています。

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

int main() {
  A::C x;
//  A::B y;
}
```

コンパイラーは、typedef 名 A::C が public なので、宣言 A::C x を許可します。コンパイラーは、A::B は、private なので、宣言 A::B y を認めません。

アクセス可能性と可視性は、別々のものであることに注意してください。可視性は、C++ のスコープ規則に基づきます。クラス・メンバーが、可視であり、同時にアクセス不能ということはあり得ます。

関連資料:

2 ページの『スコープ』

355 ページの『クラス・メンバー・リスト』

386 ページの『継承されるメンバー・アクセス』

フレンド

クラス X のフレンドは、X のメンバーではなく、 X のメンバーと同じ X へのアクセスを認可されている関数またはクラスです。クラス・メンバー・リスト内で、friend 指定子を使用して宣言された関数は、そのクラスの*フレンド関数* と呼ばれます。別のクラスのメンバー・リスト内で friend 指定子を用いて宣言されたクラスは、そのクラスのフレンド・クラス と呼ばれます。

クラス Y を定義してからでなければ、Y の任意のメンバーを、別のクラスのフレンドとして宣言することはできません。以下の例では、フレンド関数 print は、クラス Y のメンバーであり、クラス X の private データ・メンバー a および b にアクセスします。

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

class X;

class Y {
public:
  void print(X& x);
};

class X {
  int a, b;
  friend void Y::print(X& x);
public:
  X() : a(1), b(2) { }
};

void Y::print(X& x) {
  cout << "a is " << x.a << endl;
```

```
  cout << "b is " << x.b << endl;
}

int main() {
  X xobj;
  Y yobj;
  yobj.print(xobj);
}
```

上記の例の出力は、次のとおりです。

```
a is 1
b is 2
```

クラス全体をフレンドとして宣言することができます。クラス F が、クラス A のフレンドだとします。メンバー関数、およびクラス F の静的データ・メンバー定義はいずれも、クラス A へのアクセスを行えます。

次の例では、フレンド・クラス F には、クラス X の private データ・メンバー a および b にアクセスする、メンバー関数 print があります。これは、前述の例のフレンド関数 print と同じタスクを実行します。また、クラス F に宣言されたその他のメンバーも、クラス X のメンバーすべてにアクセスできます。

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

class X {
  int a, b;
  friend class F;
public:
  X() : a(1), b(2) { }
};

class F {
public:
  void print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
  }
};

int main() {
  X xobj;
  F fobj;
  fobj.print(xobj);
}
```

上記の例の出力は、次のとおりです。

```
a is 1
b is 2
```

フレンド宣言では、クラスを定義できません。例えば、コンパイラーは以下のコードを受け入れません。

```
class F;
class X {
  friend class F { };
};
```

しかし、フレンド宣言で関数を定義できます。クラスは、非ローカル・クラスであることが必要です。関数は名前空間スコープでなければならず、関数名は未修飾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。

```
class A {
  void g();
};

void z() {
  class B {
    friend void f() { }; // error
  };
}

class C {
  friend void A::g() { } // error
  friend void h() { }
};
```

コンパイラーは、h() の定義は受け入れますが、f() または g() の関数定義は受け入れません。

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。

C++11

拡張フレンド宣言

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。
C++11 標準では、*拡張フレンド宣言* 機能によって追加的な非関数フレンド宣言の形式を受け入れます。

注: 拡張フレンド宣言の構文形式は、IBM の以前のフレンド宣言構文と部分的に一致します。このセクションでは、C++11 標準と以前の ISO C++ 標準の違いに重点を置いて説明します。

この機能が使用可能になっているため、フレンド宣言のコンテキストでクラス・キーは不要になりました。この新しい構文は、詳述型指定子の一部としてクラス・キーが必要な C++98 のフレンド・クラス宣言構文とは異なります。以下の例を参照してください。

```
class F;
class G;

class X1 {
  //C++98 friend declarations remain valid in C++11.
  friend class F;

  //Error in C++98 for missing the class-key.
  friend G;
};

class X2 {
  //Error in C++98 for missing the class-key.
```

```
  //Error in C++11 for lookup failure (no previous class D declaration).
  friend D;

  friend class D;
};
```

関数とクラスに加えて、テンプレート・パラメーターおよび基本型をフレンドとして宣言することもできます。この場合、フレンド宣言で詳述型指定子を使用できません。以下の例では、テンプレート・パラメーター T をクラス F のフレンドとして宣言することができ、基本型 char をフレンド宣言で使用できます。

```
class C;

template <typename T, typename U> class F {
   //C++11 compiles sucessfully.
   //Error in C++98 for missing the class-key.
   friend T;

   //Error in both C++98 and C++11: a template parameter
   //must not be used in an elaborated type specifier.
   friend class U;

};

F<C> rc;
F<char> Ri;
```

typedef 名をフレンドとして宣言することもできますが、この場合もフレンド宣言で詳述型指定子を指定することはできません。以下の例は、typedef 名 D をクラス Base のフレンドとして宣言したものです。

```
class Derived;
typedef Derived D;

class C;
typedef C Ct;

class Base{
public:
   Base() : x(55) {}

   //C++11 compiles sucessfully.
   //Error in C++98 for missing the class-key.
   friend D;

   //Error in both C++98 and C++11: a typedef name
   //must not be used in an elaborated type specifier.
   friend class Ct;

private:
   int x;
};

struct Derived : public Base {
   int foo() { return this->x; }
};

int main() {
   Derived d;
   return d.foo();
}
```

この機能では、フレンド宣言用の新しい名前参照規則も導入しています。フレンド・クラス宣言で詳述型指定子を使用しない場合、コンパイラーは、フレンド宣言を囲む最も内側の名前空間の外部のスコープにあるエンティティー名も検索します。次の例を検討してみます。

```
struct T { };

namespace N {
   struct A {
      friend T;
   };
}
```

この例では、この機能が有効である場合、フレンド宣言ステートメントは、新規エンティティー T を宣言しませんが、T を検索します。T が見つからない場合、コンパイラーはエラーを出します。別の例を検討してみます。

```
struct T { };

namespace N {
   struct A {
        friend class T;  //fine, no error
   };
}
```

この例では、フレンド宣言ステートメントは、名前空間 N の外部の T を検索せず、::T も検出しません。代わりに、このステートメントは名前空間 N で新規クラス T を宣言します。

C++11

関連資料:

368 ページの『静的メンバー関数』

273 ページの『inline 関数指定子』

352 ページの『ローカル・クラス』

370 ページの『メンバー・アクセス』

386 ページの『継承されるメンバー・アクセス』

555 ページの『C++11 互換性の拡張機能』

フレンド・スコープ

フレンド宣言で最初に導入されるフレンド関数またはフレンド・クラスの名前は、フレンド関係を認可するクラス (エンクロージング・クラスとも*呼ばれます*) のスコープ内にはなく、フレンド関係を認可するクラスのメンバーでもありません。

フレンド宣言で最初に導入された関数の名前は、エンクロージング・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。フレンド宣言内部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、最外部のエンクロージング・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。

最初にフレンド宣言によって導入されたフレンド・クラス名は、最初の非クラスの囲みスコープに属するものと見なされます。そのスコープで一致する宣言が提供されるまで、名前参照でクラス名は検出されません。 次に例を示します。

```
namespace A {    //the first nonclass scope
   class B {
        class C {
            friend class D;
        }
   };
};
```

この例では、クラス D のフレンド宣言を囲む最初の非クラス・スコープは、名前空間 A であるため、フレンド・クラス D は、名前空間 A のスコープ内にあります。

フレンド・クラスの名前が、フレンド宣言よりも前に導入されている場合、コンパイラーは、そのフレンド・クラスの名前と一致するクラス名の検索を、フレンド宣言のスコープの先頭から開始します。ネスト・クラスの宣言の後に同じ名前のフレンド・クラスの宣言が続いている場合、そのネスト・クラスは、エンクロージング・クラスのフレンドです。

フレンド関数が別のクラスのメンバーである場合には、スコープ解決演算子 (::) を使用することが必要です。次に例を示します。

```
class A {
public:
  int f() { }
};

class B {
  friend int A::f();
};
```

基底クラスのフレンドは、その基底クラスから派生したクラスには、継承されません。次の例は、このことを示しています。

```
class A {
  friend class B;
  int a;
};

class B { };

class C : public B {
  void f(A* p) {
    p->a = 2; // error
  }
};
```

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ため、コンパイラーは、ステートメント p->a = 2 をサポートしません。

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; // error
  }
};
```

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ため、コンパイラーは、ステートメント p->a = 2 を受け入れません。

ローカル・クラスにフレンドを宣言し、フレンド名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・クラス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。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(); // error
    friend void b();
    friend void c(); // error
  };
  ::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受け入れます。

- 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。
- X moocow2: この宣言は、非ローカル・クラス ::X のオブジェクトも作成します。

コンパイラーは、次のステートメントを受け入れません。

- friend void a(): このステートメントは、関数 a() を、名前空間スコープに宣言されたと認めません。関数 a() が、f() のスコープで宣言されていないため、コンパイラーはこのステートメントを受け入れません。
- friend void c(): 関数 c() が、f() のスコープで宣言されていないため、コンパイラーはこのステートメントを受け入れません。

関連資料:

349 ページの『クラス名のスコープ』
350 ページの『ネスト・クラス』
352 ページの『ローカル・クラス』

フレンド・アクセス

クラスのフレンドは、そのクラスの private メンバーおよび protected 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 protected メンバーにはクラスのメンバー関数、またはクラスから派生したクラスを介してのみ、アクセスすることができます。

フレンド宣言は、アクセス指定子には影響されません。

関連資料:

370 ページの『メンバー・アクセス』

第 13 章 継承 (C++ のみ)

継承 とは、既存のクラスを変更することなく、既存のクラスを再利用したり拡張したりするメカニズムのことであり、その結果、クラス間に階層関係が生じます。

継承は、オブジェクトをクラスに組み込むのとほぼ同じです。クラス A のオブジェクト x を、B のクラス定義に宣言するとします。その結果、クラス B は、クラス A の public データ・メンバーおよびメンバー関数のすべてにアクセスできます。しかし、クラス B では、オブジェクト x を介してクラス 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。新規の派生クラスで基底クラスのメンバー関数やデータをオーバーライドすることによって、既存メンバー関数やデータのインプリメンテーションを変更することができます。

別の派生クラスからもクラスを派生できます。その結果、別のレベルの継承を作成します。次の例は、このことを示しています。

```
struct A { };
struct B : A { };
struct C : B { };
```

クラス B は、A の派生クラスでもあり、同時に、C の基底クラスでもあります。継承のレベル数は、リソースによってのみ限定されます。

多重継承 を使用すると、複数の基底クラスの属性を継承する派生クラスを作成することができます。派生クラスは、その全基底クラスからメンバーを継承するので、その結果あいまいさが生じる可能性があります。例えば、2 つの基底クラスに同じ名前のメンバーがある場合、派生クラスでは 2 つのメンバーを暗黙的に区別することができません。多重継承を使用するときは、基底クラスの名前へのアクセスがあいまいにならないように注意してください。詳細については、393 ページの『多重継承』を参照してください。

直接基底クラス とは、その派生クラスの宣言の中に、基底指定子として直接現れる基底クラスのことです。

間接基底クラス とは、派生クラスの宣言の中には直接出てこないが、その基底クラスの 1 つを介して派生クラスで使用できる基底クラスのことです。あるクラスについて、直接基底クラスでない基底クラスは、すべて間接基底クラスです。次の例は、直接基底クラスおよび間接基底クラスを示しています。

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

```
  public:
    int y;
};
class C : public B { };
```

クラス B は、C の直接基底クラスです。クラス A は、B の直接基底クラスです。クラス A は、C の間接基底クラスです。 (x および y は、クラス C のデータ・メンバーになります。)

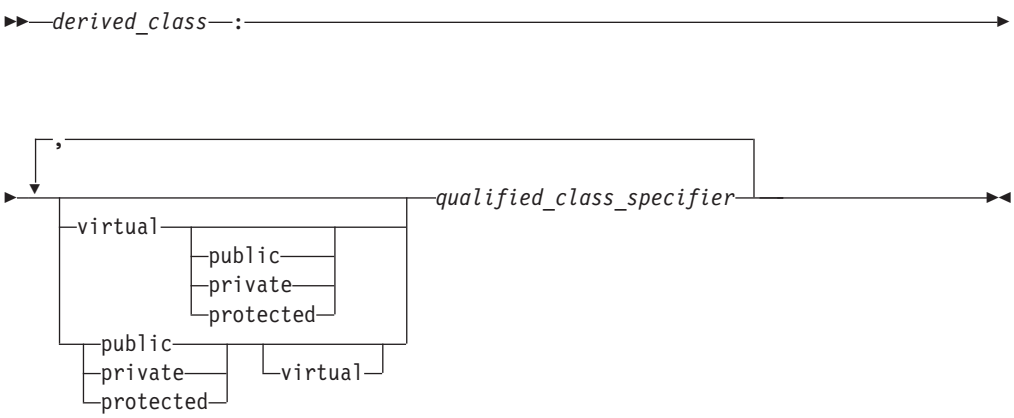
ポリモアフィック関数 は、複数の型のオブジェクトに適用できる関数です。 C++ では、ポリモアフィック関数は、2 つの方法でインプリメントできます。

- 多重定義された関数は、コンパイル時に静的にバインドされます。
- C++ が、仮想関数を提供します。*仮想関数* は、派生を介して関連付けられている、いくつかの様々なユーザー定義の型について呼び出すことができる関数です。仮想関数は、実行時に動的にバインドされます。これについては、 400 ページの『仮想関数』でさらに詳しく説明しています。

派生

C++ では、継承は、派生のメカニズムによって実現されます。派生を使用すると、*派生クラス* と呼ばれるクラスを、*基底クラス* と呼ばれる別のクラスから派生させることができます。

派生クラスの構文



派生クラスの宣言の中で、派生クラスの基底クラスをリストします。派生クラスは、これらの基底クラスからそのメンバーを継承します。

qualified_class_specifier は、クラス宣言で事前に宣言されているクラスである必要があります。

アクセス指定子 は、public、private、または protected のいずれかです。

virtual キーワードは、仮想基底クラスの宣言に使用できます。

次の例は、派生クラス D と、基底クラス V、B1、および B2 の宣言を示しています。クラス B1 は、クラス V から派生し、D の基底クラスなので、基底クラスでもあり派生クラスでもあります。

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

宣言されているが定義されていないクラスは、基底リストに入れることができません。

次に例を示します。

```
class X;

// error
class Y: public X { };
```

コンパイラーは、X が定義されていないので、クラス Y の宣言を許可しません。

クラスを派生させると、派生クラスは基底クラスの非静的データ・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。派生クラスには新しいクラス・メンバーを追加できます。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
}
```

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 に値が割り当てられます。

さらに派生クラスは、既存の基底クラス・メンバーと同じ名前を持つクラス・メンバーを宣言できます。 :: (スコープ・レゾリューション) 演算子を使用して、基底クラス・メンバーを参照できます。次に例を示します。

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

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
```

```
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  // call Derived::display()
  d.display();

  // call Base::display()
  d.Base::display();
}
```

上記の例の出力は、以下のとおりです。

```
Derived Class, Base Class
Base Class
```

派生クラスのオブジェクトは、基底クラスのオブジェクトと同様に操作できます。派生クラスのオブジェクトに対するポインターや参照を、その基底クラスに対するポインターや参照の代わりに使用することができます。例えば、派生クラスのオブジェクト D に対するポインターまたは参照を、D の基底クラスに対するポインターまたは参照を予期している関数に渡すことができます。これを実行するために明示的キャストを使用する必要はありません。標準型変換が実行されます。派生クラスに対するポインターを、明らかにアクセス可能な基底クラスを指すように、暗黙的に変換することができます。また派生クラスに対する参照を、基底クラスに対する参照に暗黙的に変換することもできます。

次の例では、派生クラスを指すポインターを基底クラスを指すポインターに変換する、標準型変換を示します。

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

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  Derived* dptr = &d;

  // standard conversion from Derived* to Base*
  Base* bptr = dptr;
```

```
  // call Base::display()
  bptr->display();
}
```

上記の例の出力は、以下のとおりです。

```
Base Class
```

ステートメント Base* bptr = dptr は、Derived 型のポインターを Base 型のポインターに変換します。

この逆は認められていません。基底クラスのオブジェクトを指すポインターや参照を、派生クラスを指す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。

関連資料:

394 ページの『仮想基底クラス』

『継承されるメンバー・アクセス』

350 ページの『不完全なクラス宣言』

173 ページの『スコープ解決演算子　:: (C++ のみ)』

継承されるメンバー・アクセス

以下のセクションでは、保護非静的基底クラス・メンバーに影響を与えるアクセス規則と、アクセス指定子を使用して派生クラスを宣言する方法について説明します。

- 『protected メンバー』
- 388 ページの『基底クラス・メンバーのアクセス制御』

関連資料:

370 ページの『メンバー・アクセス』

protected メンバー

基底クラスから派生したどのクラスのメンバーおよびフレンドも、次のいずれかの方法を使用して、protected 非静的基底クラス・メンバーにアクセスすることができます。

- 直接または間接の派生クラスを指すポインター

- 直接または間接の派生クラスへの参照
- 直接または間接の派生クラスのオブジェクト

基底クラスから private にクラスを派生させた場合、基底クラスの全 protected メンバーは、派生クラスの private メンバーになります。

派生クラス B のフレンドまたはメンバー関数で、基底クラスの A の protected 非静的メンバー x を参照する場合、A から派生したクラスに対するポインター、参照、またはオブジェクトを介して x にアクセスする必要があります。しかし、x にアクセスし、メンバーに対するポインターを作成している場合は、派生クラス B の名前を付けるネスト名前指定子で x を修飾する必要があります。

```
class A {
public:
protected:
  int i;
};


class B : public A {
  friend void f(A*, B*);
  void g(A*);
};

void f(A* pa, B* pb) {
//  pa->i = 1;
  pb->i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}

void B::g(A* pa) {
//  pa->i = 1;
  i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}

void h(A* pa, B* pb) {
//  pa->i = 1;
//  pb->i = 2;
}

int main() { }
```

クラス A には、protected データ・メンバーである、整数 i が入っています。 B は A から派生するので、B のメンバーは、 A の protected メンバーへのアクセスが可能です。関数 f() はクラス B のフレンドです。

- コンパイラーは、pa が派生クラス B に対するポインターでないので、pa->i = 1 を許可しません。
- コンパイラーは、i が派生クラス B の名前で修飾されていないので、int A::* point_i = &A::i を許可しません。

関数 g() は、クラス B のメンバー関数です。コンパイラーが許可するあるいは許可しないステートメントについての直前の注釈のリストは、次の点を除き、g() にも適用できます。

- コンパイラーは、i = 2 は、this->i = 2 と等価なので、認めます。

関数 h() は、A の派生クラスのフレンドでもメンバーでもないので、h() は、A のどの protected メンバーにもアクセスすることはできません。

基底クラス・メンバーのアクセス制御

派生クラスの宣言においては、派生クラスの基底リストの中の各基底クラスの前に、アクセス指定子を置くことができます。これによって、基底クラスから見たときの基底クラスの各メンバーのアクセス属性は、変更されませんが、派生クラスが、基底クラスのメンバーへのアクセス制御を制限できるようになります。

3 つのアクセス指定子のいずれかを使用して、クラスを派生させることができます。

- public 基底クラスでは、基底クラスの public および protected メンバーは、派生クラスにおいても public および protected メンバーです。
- protected 基底クラスにおいては、基底クラスの public および protected メンバーは、派生クラスの protected メンバーになります。
- private 基底クラスにおいては、基底クラスの public および protected メンバーは、派生クラスでは private メンバーになります。

すべての場合において、基底クラスの private メンバーは private のままです。基底クラスの private メンバーは、基底クラス内のフレンド宣言において、明示的にアクセスを認可されている場合でなければ、派生クラスから使用することはできません。

次の例では、クラス D は、クラス B から public に派生します。クラス B は、この宣言により、public 基底クラスに宣言されます。

```
class B { };
class D : public B   // public derivation
{ };
```

構造体とクラスの両方を、派生クラス宣言の基底リストの中の基底クラスとして使用することができます。

- 派生クラスがキーワード class で宣言される場合、その基本リスト指定子にあるデフォルトのアクセス指定子は、private です。
- 派生クラスがキーワード struct で宣言される場合、その基本リスト指定子にあるデフォルトのアクセス指定子は、public です。

以下の例を参照してください。

```
struct B{
};

class D : B {        // private derivation
};

struct E : B{        // public 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)

関連資料:

370 ページの『メンバー・アクセス』

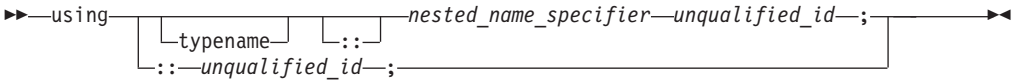
359 ページの『メンバー・スコープ』

using 宣言およびクラス・メンバー

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。

基底クラスおよび派生クラスからメンバー関数のセットを作成したい場合、またはクラス・メンバーのアクセスを変更したい場合は、クラス定義で using 宣言が必要です。

using 宣言の構文



クラス A の using 宣言は、次のオプションのいずれかに名前を付けることがあります。

- A の基底クラスのメンバー
- A の基底クラスのメンバーである無名共用体のメンバー
- A の基底クラスのメンバーである列挙型の列挙子

次の例は、このことを示しています。

```
struct Z {
  int g();
};

struct A {
  void f();
  enum E { e };
  union { int u; };
};

struct B : A {
  using A::f;
  using A::e;
  using A::u;
//  using Z::g;
};
```

コンパイラーは、Z が A の基底クラスでないので、using 宣言 using Z::g を許可しません。

using 宣言は、テンプレートに名前を付けることはできません。例えば、コンパイラーは次のコードを許可しません。

```
struct A {
  template<class T> void f(T);
};

struct B : A {
  using A::f<int>;
};
```

using 宣言で示されている名前のインスタンスは、いずれもアクセス可能でなければなりません。次の例は、このことを示しています。

```
struct A {
private:
  void f(int);
public:
  int f();
protected:
  void g();
};

struct B : A {
//  using A::f;
  using A::g;
};
```

コンパイラーは、int A::f() はアクセス可能ですが、void A::f(int) が B からアクセス不可能なので、using 宣言 using A::f を許可しません。

関連資料:

349 ページの『クラス名のスコープ』

319 ページの『using 宣言およびネームスペース』

基底クラスと派生クラスからのメンバー関数の多重定義

クラス A で f と名付けられたメンバー関数は、戻りの型や引数に関係なく、A の基底クラスで、他の f という名前のメンバーをすべて隠します。次の例は、このことを示しています。

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

struct B : A {
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
//  obj_B.f();
}
```

コンパイラーは、void B::f(int) の宣言が A::f() を隠しているので、関数呼び出し obj_B.f() を許可しません。

基底クラス A の関数を、派生クラス B で、隠蔽ではなく多重定義するには、using 宣言を使用して、関数の名前を B のスコープに導入します。以下の例は、using 宣言 using A::f を除いては、直前の例と同じです。

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

struct B : A {
  using A::f;
  void f(int) { }
};

int main() {
```

```
  B obj_B;
  obj_B.f(3);
  obj_B.f();
}
```

クラス B に using 宣言があるので、名前 f は 2 つの関数で多重定義されます。これで、コンパイラーは、関数呼び出し obj_B.f() を許可します。

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、むしろそれを隠します。次の例は、このことを示しています。

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

struct A {
  void f() { }
  void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
  B obj_B;
  obj_B.f(3);
}
```

上記の例の出力は、以下のとおりです。

```
void B::f(int)
```

仮想関数を using 宣言を使用して多重定義できます。次に例を示します。

```
#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;
  A* pa = &obj_B;
  pa->f(3);
  pa->f();
}
```

この例で、B::f(int) は仮想関数であり、using A::f; 宣言が指定されていても、A::f(int) をオーバーライドします。この出力は以下のとおりです。

```
void B::f(int)
void A::f()
```

関連資料:

325 ページの『第 10 章 多重定義 (C++ のみ)』
7 ページの『名前の隠蔽 (C++ のみ)』
389 ページの『using 宣言およびクラス・メンバー』

クラス・メンバーのアクセス権の変更

クラス B がクラス A の直接基底クラスである場合に、クラス A のメンバーに対するクラス B のアクセスを制限するには、アクセス指定子 protected または private のいずれかを使用して、B を A から派生させます。

クラス B から継承された、クラス A のメンバー x のアクセス権を拡大するには、using 宣言を使用してください。 using 宣言を指定して x へのアクセス権を制限することはできません。次のメンバーのアクセス権を拡大することができます。

- private として継承されたメンバー。 (using 宣言は、メンバーの名前に対するアクセス権なので、private として宣言されたメンバーのアクセス権を拡大することはできません。)
- protected として継承、または宣言されたメンバー。

次の例は、このことを示しています。

```
struct A {
protected:
  int y;
public:
  int z;
};

struct B : private A { };

struct C : private A {
public:
  using A::y;
  using A::z;
};

struct D : private A {
protected:
  using A::y;
  using A::z;
};

struct E : D {
  void f() {
    y = 1;
    z = 2;
  }
};

struct F : A {
public:
  using A::y;
private:
  using A::z;
};

int main() {
  B obj_B;
//  obj_B.y = 3;
//  obj_B.z = 4;

  C obj_C;
```

```
  obj_C.y = 5;
  obj_C.z = 6;

  D obj_D;
//  obj_D.y = 7;
//  obj_D.z = 8;

  F obj_F;
  obj_F.y = 9;
  obj_F.z = 10;
}
```

コンパイラーは、上記の例から、次の割り当てを許可しません。

- obj_B.y = 3 および obj_B.z = 4: メンバー y および z は、private として継承されました。
- obj_D.y = 7 および obj_D.z = 8: メンバー y および z は、private として継承されましたが、それらのアクセスは protected に変更されました。

コンパイラーは、上記の例から、次のステートメントを許可します。

- D::f() の y = 1 および z = 2: メンバー y および z は、private として継承されましたが、それらのアクセスは protected に変更されました。
- obj_C.y = 5 および obj_C.z = 6: メンバー y および z は private として継承されましたが、それらのアクセスは public に変更されました。
- obj_F.y = 9: メンバー y のアクセスは、protected から public に変更されました。
- obj_F.z = 10: メンバー z のアクセスは、public のままです。 private の using 宣言 using A::z は、z のアクセスに影響を与えません。

関連資料:

370 ページの『メンバー・アクセス』

386 ページの『継承されるメンバー・アクセス』

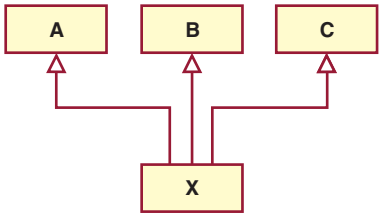
多重継承

1 つのクラスを複数の基底クラスから派生できます。複数の直接基底クラスから 1 つのクラスが派生することを、*多重継承* と呼びます。

次の例では、クラス A、B、および C は、派生クラス X の直接の基底クラスです。

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

次の*継承グラフ* は、上記の例で示した継承の関係を説明しています。矢印は、矢印の尾部の地点にあるクラスの直接基底クラスを指し示します。

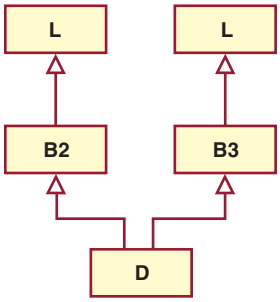


派生の順序は、コンストラクターによるデフォルト初期化およびデストラクターによる終結処理の順序の決定にのみ関係します。

直接の基底クラスは、派生クラスの基底リストに 2 回以上出現させ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.
```

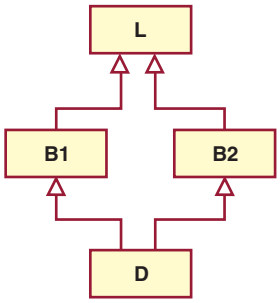
また、 383 ページの『派生』での説明のように、基底クラスの宣言に基底指定子 virtual を使用しても、このようなあいまいさを避けることができます。

仮想基底クラス

共通の基底クラス A を持つ 2 つの派生クラス B および C があり、さらに B および C から継承した別のクラス D があるとします。基底クラス A を*仮想* として宣言することで、B および C が、同じ A のサブオブジェクトを共用していることを保証できます。

次の例では、クラス D のオブジェクトには、クラス L の 2 つの別個のサブオブジェクトがあり、一方はクラス B1 を介し、もう一方はクラス B2 を介しています。クラス B1 および B2 の*基底リスト* の基底クラス指定子に、キーワード virtual を使用することで、クラス B1 およびクラス B2 が共用している、型 L のただ 1 つのサブオブジェクトが存在することを指示できます。

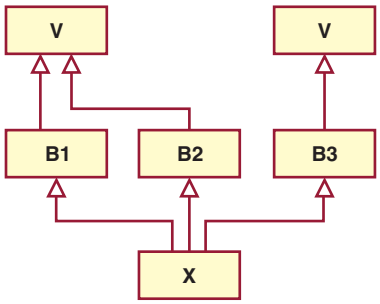
次に例を示します。




```
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、 1 つはクラス B1 とクラス B2 で共有され、もう一方はクラス B3 を介してされます。

関連資料:

383 ページの『派生』

マルチアクセス

仮想基底クラスを含む継承グラフでは、複数のパスを経由して到達できる名前は、最大広範囲のアクセスを提供するパスを介してアクセスされます。

次に例を示します。

```
class L {
public:
  void f();
};

class B1 : private virtual L { };
```

```
class B2 : public virtual L { };

class D : public B1, public B2 {
public:
  void f() {
    // L::f() is accessed through B2
    // and is public
    L::f();
  }
};
```

上記の例では、関数 f() はクラス B2 を介してアクセスされます。クラス B2 は公的に継承され、クラス B1 は私的に継承されているので、クラス B2 の方がより多くのアクセスを提供します。

関連資料:

370 ページの『メンバー・アクセス』

386 ページの『protected メンバー』

388 ページの『基底クラス・メンバーのアクセス制御』

あいまいな基底クラス

クラスを派生させるとき、基底クラスと派生クラスとに同じ名前のメンバーがあると、あいまいさが生じる可能性があります。固有な関数、またはオブジェクトを参照しない名前または修飾名を使用すると、基底クラス・メンバーへのアクセスがあいまいになります。派生クラス内であいまいな名前のメンバーを宣言してもエラーではありません。あいまいなメンバー名を使用すると、そのあいまいさにエラーとしてフラグを付けます。

例えば、A と B という名前の 2 つのクラスが、両方とも x という名前のメンバーを持ち、C という名前のクラスは、A と B の両方から継承するとします。クラス C から x にアクセスする試みは、あいまいになります。スコープ・レゾリューション (::) 演算子を使用して、そのクラス名でメンバーを修飾することによって、あいまいさを解消することができます。

```
class B1 {
public:
  int i;
  int j;
  void g(int) { }
};

class B2 {
public:
  int j;
  void g() { }
};

class D : public B1, public B2 {
public:
  int i;
};

int main() {
  D dobj;
  D *dptr = &dobj;
  dptr->i = 5;
//  dptr->j = 10;
```

```
  dptr->B1::j = 10;
//  dobj.g();
  dobj.B2::g();
}
```

ステートメント dptr->j = 10 は、B1 と B2 の両方に名前 j が現れるので、あいまいになります。B1::g(int) および B2::g() が異なるパラメーターを持っていますが、名前 g が B1 および B2 の両方に現れるので、ステートメント dobj.g() は、あいまいになります。

コンパイラーはコンパイル時にあいまいさを検査します。あいまいさの検査は、アクセス制御や型検査の前に行われるので、同じ名前の複数のメンバーの中の 1 つだけが派生クラスからアクセス可能である場合でも、あいまいさが検出される可能性があります。

名前の隠蔽

A と B という名前の 2 つのサブオブジェクトには、両方とも x という名前のメンバーがあるとします。A が B の基底クラスである場合、サブオブジェクト B のメンバー名 x は、サブオブジェクト A のメンバー名 x を隠します。次の例は、このことを示しています。

```
struct A {
   int x;
};

struct B: A {
   int x;
   void f() { x = 0; }
};

int main() {
   B b;
   b.f();
}
```

関数 B::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。

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 上の A::x を隠しますが、パス C::A::x 上の A::x は隠しません。したがって、変数 x は、D::x または A::x のどちらでも参照できます。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。

あいまいでないクラス・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
}
```

以下の条件が真である場合、基底クラスのメンバーを指すポインターは、派生クラスのメンバーを指すポインターに変換することができます。

- 型変換が、あいまいではない。基底クラスの複数インスタンスが派生クラス内にあると、変換はあいまいになります。
- 派生クラスを指すポインターは、基底クラスを指すポインターに変換することができる。その場合、その基底クラスは、*アクセス可能* であるといいます。
- メンバー型が一致する。例えば、クラス A は、クラス B の基底クラスであると想定します。型 int の A のメンバーを指すポインターを、型 float の型 B のメンバーを指すポインターには、変換できません。
- 基底クラスが、仮想ではない。

多重定義解決

多重定義解決は、コンパイラーが任意の関数名をあいまいさなく検出した後で 行われます。次の例は、このことを示しています。

```
struct A {
   int f() { return 1; }
};

struct B {
   int f(int arg) { return arg; }
};

struct C: A, B {
   int g() { return f(); }
};
```

コンパイラーは、名前 `f` が `A` および `B` の両方で宣言されているので、`C::g()` の `f()` の関数呼び出しを許可しません。コンパイラーは、多重定義解決が基底一致 `A::f()` を選択する前に、あいまいさエラーを検出します。

関連資料:

173 ページの『スコープ解決演算子　:: (C++ のみ)』

394 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);
}
```

上記の例の出力は、以下のとおりです。

```
Class A
```

関数 g() が呼び出されると、引数は、型 B のオブジェクトを参照しますが、関数 A::f() がコールされます。コンパイル時にコンパイラーが唯一認識できるのは、関数 g() の引数が、A から派生したオブジェクトの左辺値参照であることです。引数が型 A と型 B のいずれのオブジェクトに対する左辺値参照かを判別することはできません。しかし、これは実行時に判別されます。次の例は、A::f() が virtual キーワードで宣言されている点を除いては、直前の例と同じです。

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

struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
   arg.f();
}

int main() {
   B x;
   g(x);
}
```

上記の例の出力は、以下のとおりです。

```
Class B
```

virtual キーワードは、左辺値参照の型ではなく、左辺値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 {
   virtual void f(int) { cout << "Class B" << endl; }
};

struct C: B {
   void f() { cout << "Class C" << endl; }
};

int main() {
   B b; C c;
   A* pa1 = &b;
   A* pa2 = &c;
//   b.f();
   pa1->f();
   pa2->f();
}
```

上記の例の出力は、以下のとおりです。

```
Class A
Class C
```

関数 B::f は、仮想ではありません。これは A::f を隠蔽します。したがって、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 が、型 T のクラスを指すポインターまたは参照を戻し、A::f が、T のあいまいでない直接あるいは間接基底クラスを指すポインターまたは参照を戻す。
- B::f が戻すポインター、あるいは参照における const または volatile 修飾は、A::f が戻すポインター、または参照と同等な、あるいはより低い const または volatile 修飾を持っている。
- B::f の戻りの型は、B::f の宣言ポイントで完了する必要がある。または、型 B となる。
- A::f が左辺値参照を戻すのは、B::f が左辺値参照を戻す場合に限られる。

次の例は、このことを示しています。

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

struct A { };

class B : private A {
   friend class D;
   friend class F;
};
```

```
A global_A;
B global_B;

struct C {
   virtual A* f() {
      cout << "A* C::f()" << endl;
      return &global_A;
   }
};

struct D : C {
   B* f() {
      cout << "B* D::f()" << endl;
      return &global_B;
   }
};

struct E;

struct F : C {

//   Error:
//   E is incomplete
//   E* f();
};

struct G : C {

//   Error:
//   A is an inaccessible base class of B
//   B* f();
};

int main() {
   D d;
   C* cp = &d;
   D* dp = &d;

   A* ap = cp->f();
   B* bp = dp->f();
};
```

上記の例の出力は、以下のとおりです。

```
B* D::f()
B* D::f()
```

ステートメント `A* ap = cp->f()` は、`D::f()` を呼び出して、戻されるポインターを型 `A*` に変換します。ステートメント `B* bp = dp->f()` は、`D::f()` も呼び出しますが、戻されるポインターを変換しません。戻りの型は `B*` となります。コンパイラーは、`E` が完全なクラスではないので、仮想関数 `F::f()` の宣言を許可しません。コンパイラーは、クラス `A` が `B` にアクセス可能な基底クラスではないので、仮想関数 `G::f()` の宣言を許可しません (フレンド・クラス `D` および `F` と異なり、`B` の定義は、クラス `G` のメンバーにアクセス権を与えません)。

定義によると、仮想関数は、基底クラスのメンバー関数であり、特定のオブジェクトに従って、関数のどのインプリメンテーションを呼び出すかを決めるので、仮想関数をグローバルにも静的にもすることはできません。仮想関数を別のクラスのフレンドとして宣言することができます。

基底クラスにおいて仮想と宣言した関数の場合でも、スコープ・レゾリューション (::) 演算子を使用すれば、それを直接にアクセスすることができます。この場合、仮想関数呼び出しのメカニズムを抑止し、基底クラスで定義された関数インプリメンテーションが使用されます。さらに、派生クラスで仮想メンバー関数を再オーバーライドしなければ、その関数に対する呼び出しでは、基底クラスで定義された関数インプリメンテーションが使用されます。

C++11 削除された定義がある関数は、削除された定義がない関数をオーバーライドできません。同様に、削除された定義がない関数は、削除された定義がある関数をオーバーライドできません。 C++11

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
- 宣言された純粋
- 定義され宣言された純粋

1 つ以上の純粋仮想メンバー関数を含むクラスは、*抽象クラス* と呼ばれます。

関連資料:

406 ページの『抽象クラス』

267 ページの『削除済み関数』

あいまいな仮想関数呼び出し

1 つの仮想関数を、2 つ以上のあいまいな仮想関数でオーバーライドすることはできません。これは、仮想基底クラスから派生した 2 つの非仮想基底から継承する派生クラスで発生する可能性があります。

次に例を示します。

```
class V {
public:
 virtual void f() { }
};

class A : virtual public V {
  void f() { }
};

class B : virtual public V {
 void f() { }
};

// Error:
// Both A::f() and B::f() try to override V::f()
class D : public A, public B { };

int main() {
  D d;
  V* vptr = &d;

  // which f(), A::f() or B::f()?
 vptr->f();
}
```

コンパイラーは、クラス D の定義を許可しません。クラス A では、A::f() のみが V::f() をオーバーライドします。同様にクラス B でも、B::f() のみが V::f() をオーバーライドします。ただし、クラス D では、A::f() と B::f() の両方が、

V::f() をオーバーライドしようとします。上記の例で示すように、D オブジェクトが、クラス V を指すポインターを使用して参照される場合、コンパイラーは、どちらの関数を呼び出すべきかを決めることができないので、このような試みは許されません。 1 つの関数のみが仮想関数をオーバーライドできます。

同じクラス型の別個のインスタンスがあることによって、仮想関数のあいまいなオーバーライドが起きた場合、特殊なケースになります。次の例では、クラス D には、クラス A の 2 つの別々のサブオブジェクトがあります。

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

struct A {
   virtual void f() { cout << "A::f()" << endl; };
};

struct B : A {
   void f() { cout << "B::f()" << endl;};
};

struct C : A {
   void f() { cout << "C::f()" << endl;};
};

struct D : B, C { };

int main() {
   D d;

   B* bp = &d;
   A* ap = bp;
   D* dp = &d;

   ap->f();
//   dp->f();
}
```

クラス D にはクラス A の 2 つのオカレンスがあり、1 つは B から継承されたものであり、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。しかし、コンパイラーは、D::B::f または D::C::f のどちらも呼び出すことができるので、ステートメント dp->f() を許可しません。

仮想関数のアクセス

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に設計されたクラスです。抽象クラスには、少なくとも 1 つの*純粋仮想関数* が含まれています。クラス宣言の中の仮想メンバー関数の宣言で、*純粋指定子* (= 0) を使用することによって、純粋仮想関数を宣言することができます。

次に抽象クラスの例を示します。

```
class AB {
public:
  virtual void f() = 0;
};
```

関数 AB::f は、純粋仮想関数です。関数宣言に、純粋指定子と定義の両方を入れることはできません。例えば、コンパイラーは次のコードを許可しません。

```
struct A {
  virtual void g() { } = 0;
};
```

抽象クラスをパラメーター型、関数からの戻りの型、または明示型変換の型として使用することはできませんし、抽象クラスのオブジェクトも宣言することはできません。ただし、抽象クラスを指すポインター、および参照を宣言することは可能です。次の例は、このことを示しています。

```
struct A {
  virtual void f() = 0;
};

struct B : A {
  virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();
```

```
// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

// Error:
// Class A is an abstract class
//   A a;

   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

クラス A は、抽象クラスです。コンパイラーは、関数宣言 A g() または void h(A)、オブジェクト a の宣言、そして b の型 A への静的キャストも許可しません。

仮想メンバー関数は、継承されます。派生クラスにある各純粋仮想関数をオーバーライドしない限り、抽象基底クラスから派生するクラスも抽象になります。

次に例を示します。

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
  D2 d;
}
```

コンパイラーは、オブジェクト d の宣言を許可しません。 D2 が、AB から純粋仮想関数 f() を継承した抽象クラスだからです。関数 D2::f() を定義すると、コンパイラーは、オブジェクト d の宣言を許可します。これが、継承された純粋仮想関数 AB::f() をオーバーライドするからです。 D2 の抽象化を避けたい場合には、関数 AB::f() をオーバーライドする必要があります。

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、直接呼び出しの警告を純粋仮想関数に発行しますが、間接呼び出しには、警告を発行しません。

関連資料:

400 ページの『仮想関数』

405 ページの『仮想関数のアクセス』

第 14 章 特殊メンバー関数 (C++ のみ)

デフォルトのコンストラクター、デストラクター、コピー・コンストラクター、およびコピー割り当て演算子は、*特殊メンバー関数* 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。

C++11 特殊メンバー関数は、ユーザー宣言であるものの明示的にデフォルト設定されていない場合、またはその最初の宣言で削除済みになっている場合は、ユーザー提供です。 C++11

コンストラクターとデストラクターの概要

データや関数を含むクラスの内部構造は複雑なので、オブジェクトの初期化やクラス終結処理は、単純なデータ構造の場合より格段に複雑です。コンストラクターやデストラクターは、クラス・オブジェクトの構成や破棄に使用されるクラスの特殊メンバー関数です。構成では、オブジェクトのメモリー割り振りや初期化も併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 アクセス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の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当てられたオブジェクトを解放します。オブジェクトを解放するには、delete 演算子を使用します。

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たり、多重定義したりしませんが、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;
}
```

上記の例の出力は、以下のとおりです。

```
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、いず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。

関連資料:

290 ページの『constructor および destructor』

218 ページの『new 式 (C++ のみ)』

コンストラクター

A コンストラクター は、そのクラスと同じ名前をもつメンバー関数です。次に例を示します。

```
class X {
public:
  X();      // constructor for class X
};
```

コンストラクターは、そのクラス型のオブジェクトの作成に使用され、オブジェクトを初期化できます。

コンストラクターは、virtual または static として宣言できませんし、const、volatile、または const volatile として宣言することもできません。

コンストラクターの戻りの型は、指定しません。コンストラクター本体にあるリターン・ステートメントは、戻り値を持てません。

関連資料:

290 ページの『constructor および destructor』

デフォルトのコンストラクター

デフォルト・コンストラクター とは、パラメーターがないか、ある場合でも、すべての パラメーターにデフォルト値があるコンストラクターです。

クラス A にユーザー定義のコンストラクターが必要であるにもかかわらず、それが存在しない場合、コンパイラーは、デフォルトのパラメーターなし 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 AA 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 NULL ボディも持つようにはなりません。

コンパイラーは、最初に暗黙的に宣言された C++11 または明示的にデフォルト設定された C++11 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 C++11 または明示的にデフォルト設定された C++11 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、次のことがすべてに真であれば、*単純* です。

- これは暗黙的に宣言されるか、C++11 または明示的にデフォルト設定されます C++11 。
- A に仮想関数がなく、仮想基底クラスもない

- A の直接基底クラスが、すべて単純コンストラクターを持っている
- A のすべての非静的データ・メンバーに関するクラスが、単純コンストラクターを持っている

上記のいずれかが偽であれば、コンストラクターは、*非単純* です。

共用体メンバーは、非単純コンストラクターを持つクラス型にはできません。

すべての関数と同様、コンストラクターは、デフォルト引数を持つことができます。これらは、メンバー・オブジェクトの初期化に使用されます。デフォルト値が提供される場合、末尾引数は、コンストラクターの式リストで省略できます。コンストラクターにデフォルト値を持たない引数がある場合、それはデフォルト・コンストラクターではないことに注意してください。

以下に示すのは、1 つのコンストラクターと、2 つのデフォルト・コンストラクターを持つクラスを定義する例です。

```
class X {
public:
  X();                        // Default constructor with no arguments
  X(int = 0);                 // Default constructor with one default argument
  X(int, int , int = 0);      // Constructor
};
```

注: C++11 デフォルトのコンストラクターは、明示的にデフォルトに設定された関数または削除済み関数として宣言できます。詳しくは、 266 ページの『明示的にデフォルトに設定された関数』および 267 ページの『削除済み関数』 を参照してください。 C++11

関連資料:

290 ページの『constructor および destructor』

433 ページの『コピー・コンストラクター』

409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

委任コンストラクター (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 の前は、同じクラスの複数のコンストラクター内の共通の初期化を、堅固かつ保守可能な方法で、1 つの場所にまとめることはできませんでした。既存の C++ プログラムでは、この問題を部分的に改善するために、初期化の代わりに代入 を使用するか、または共通の初期化関数を追加することができました。

委任コンストラクター機能を使用すると、共通の初期化をまとめて、ターゲット・コンストラクター と呼ばれる 1 つのコンストラクターに初期化を任せることがで

きます。委任コンストラクターは、ターゲット・コンストラクターを呼び出して初期化を実行できます。委任コンストラクターは、1 つ以上の委任コンストラクターのターゲット・コンストラクター としても使用できます。この機能を使用すると、プログラムの可読性と保守性を向上させることができます。

委任コンストラクターおよびターゲット・コンストラクターは、他のコンストラクターと同じインターフェースを提供します。ターゲット・コンストラクターは、委任コンストラクターのターゲットになるための特別な処理を必要としません。これらは、多重定義解決またはテンプレート引数の推定によって選択されます。ターゲット・コンストラクターの実行完了後に、制御は委任コンストラクターに戻ります。

以下の例では、`X(int x, int y)` および `X(int x, int y, int z)` は両方とも `X(int x)` に委任します。この例は、単一コンストラクターで共通の初期化を配置する場合の一般的な使用法を示します。

```
#include <cstdio>

struct X {
   const int i;
   X(int x, int y) : X(x+y) { }
   X(int x, int y, int z) : X(x*y*z) {}
   X(int x) : i(x) { }
};

int main(void){
   X var1(55,11);
   X var2(2,4,6);
   std::printf("%d, %d¥n", var1.i, var2.i);
   return 0;
}
```

この例の出力は次のとおりです。

```
66,48
```

委任コンストラクターは別のコンストラクターのターゲット・コンストラクターになり、委任チェーンが形成される場合があります。オブジェクトの構造体で呼び出される最初のコンストラクターは、*基本コンストラクター* と呼ばれます。コンストラクターは、直接および間接に関わらず、自分自身に委任することはできません。コンパイラーは、委任の再帰的チェーンに関係するコンストラクターが、すべて 1 つの変換単位で定義されている場合に、この違反を検出できます。次の例を検討してみます。

```
struct A{
  int x,y;
  A():A(42){}
  A(int x_):A() {x = x_;}
};
```

例では、コンストラクター `A()` がコンストラクター `A(int x_)` に委任し、`A(int x_)` がまた `A()` に委任する無限の再帰的循環が存在します。コンパイラーはエラーを出して、違反を示します。

以下に示す他の既存の技法と相互作用する委任コンストラクター機能を使用できます。

- 複数のコンストラクターが同じ名前を持つ場合、名前および多重定義解決で、ターゲット・コンストラクターであるコンストラクターを特定できます。
- 委任コンストラクターをテンプレート・クラスで使用する場合、テンプレート・パラメーターの推定は正常に機能します。

関連資料:

555 ページの『C++11 互換性の拡張機能』

constexpr コンストラクター (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

constexpr 指定子を指定して宣言されるコンストラクターは、constexpr コンストラクターです。以前は、組み込み型の式のみが、有効な定数式でした。 constexpr コンストラクターを使用すると、ユーザー定義型のオブジェクトを、有効な定数式に組み込むことができます。

constexpr コンストラクターの定義は、以下の要件を満たしていなければなりません。

- 含むクラスが、仮想基底クラスを持っていてはならない。
- 各パラメーター型がリテラル型である。
- その関数本体が = delete または = default である。そうでない場合、それは以下の制約を満たしている必要があります。
  - それが関数 try ブロックではない。
  - その中の複合ステートメントに、必ず以下のステートメントのみが含まれている。
    - null ステートメント
    - static_assert 宣言
    - クラスまたは列挙を定義していない typedef 宣言
    - using ディレクティブ
    - using 宣言
- 各非静的なデータ・メンバーおよび基底クラス・サブオブジェクトが初期化されている。
- 非静的データ・メンバーおよび基底クラス・サブオブジェクトを初期化するために使用する各コンストラクターが、constexpr コンストラクターである。
- メンバー初期化指定子 ID によって指定されていないすべての非静的データ・メンバーの初期化指定子が定数式である。
- データ・メンバーを初期化するときに、以下のコンテキストに関係するすべての暗黙的な変換が、以下の定数式内で有効でなければならない。

- 任意のコンストラクターの呼び出し
- 任意の式のデータ・メンバー型への変換

暗黙的に定義されたデフォルトのコンストラクターは、クラスの一連の初期化を実行します。これは初期化指定子および空の複合ステートメントなしで、そのクラス用のユーザー作成のデフォルト・コンストラクターによって実行されます。そのユーザー定義のデフォルトのコンストラクターが constexpr コンストラクターの要件を満たす場合、暗黙的に定義されたデフォルトのコンストラクターは、constexpr コンストラクターです。

constexpr コンストラクターは暗黙的にインラインになります。

以下の例は、constexpr コンストラクターの使用法を示しています。

```
struct BASE {
};

struct B2 {
  int i;
};

//NL is a non-literal type.
struct NL {
  virtual ~NL() {
  }
};

int i = 11;

struct D1 : public BASE {
  //OK, the implicit default constructor of BASE is a constexpr constructor.
  constexpr D1() : BASE(), mem(55) { }

  //OK, the implicit copy constructor of BASE is a constexpr constructor.
  constexpr D1(const D1& d) : BASE(d), mem(55) { }

  //OK, all reference types are literal types.
  constexpr D1(NL &n) : BASE(), mem(55) { }

  //The conversion operator is not constexpr.
  operator int() const { return 55; }

private:
  int mem;
};

struct D2 : virtual BASE {
  //error, D2 must not have virtual base class.
  constexpr D2() : BASE(), mem(55) { }

private:
  int mem;
};

struct D3 : B2 {
  //error, D3 must not be a function try block.
  constexpr D3(int) try : B2(), mem(55) { } catch(int) { }

  //error, illegal statement is in body.
  constexpr D3(char) : B2(), mem(55) { mem = 55; }

  //error, initializer for mem is not a constant expression.
  constexpr D3(double) : B2(), mem(i) { }
```

```
  //error, implicit conversion is not constexpr.
  constexpr D3(const D1 &d) : B2(), mem(d) { }

  //error, parameter NL is a non-literal type.
  constexpr D3(NL) : B2(), mem(55) { }

private:
  int mem;
};
```

関連資料:

174 ページの『一般化された定数式 (C++11)』

99 ページの『constexpr 指定子 (C++11)』

63 ページの『型指定子』

266 ページの『明示的にデフォルトに設定された関数』

267 ページの『削除済み関数』

コンストラクターでの明示的初期化

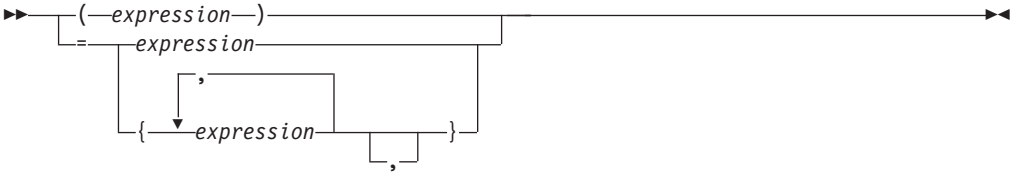
クラス・オブジェクトは、コンストラクターを用いて明示的に初期化されるか、またはデフォルト・コンストラクターを持っていなければなりません。コンストラクターを使用する明示的初期化は、集合体初期化の場合を除き、非静的定数および参照クラス・メンバーを初期化する唯一の方法です。

暗黙的に宣言された C++11 または明示的にデフォルトに設定された C++11 コンストラクター、仮想関数、private または protected 非静的データ・メンバー、および基底クラスのいずれも持たないクラス・オブジェクトは、*集合体* と呼ばれます。集合体の例としては、C 形式の構造体および共用体があります。

クラス・オブジェクトを作成する場合、そのオブジェクトを明示的に初期化します。クラス・オブジェクトを初期化するには、次の 2 つの方法があります。

- 括弧で囲んだ式のリストの使用。コンパイラーは、このリストをコンストラクターの引数リストとして使用し、クラスのコンストラクターを呼び出します。
- 単一初期化値、および = 演算子の使用。このような型の式は、割り当てでなく初期化なので、割り当て演算子関数 (これが存在する場合でも) は、呼び出されません。単一引数の型は、コンストラクターに対する最初の引数の型と一致していなければなりません。コンストラクターに残りの引数がある場合、これらの引数はデフォルト値を持っている必要があります。

初期化指定子の構文



次の例は、クラス・オブジェクトを明示的に初期化するいくつかのコンストラクターの宣言および使用の方法を示します。

```
// This example illustrates explicit initialization
// by constructor.
#include <iostream>
using namespace std;

class complx {
  double re, im;
public:

  // default constructor
  complx() : re(0), im(0) { }

  // copy constructor
  complx(const complx& c) { re = c.re; im = c.im; }

  // constructor with default trailing argument
  complx( double r, double i = 0.0) { re = r; im = i; }

  void display() {
    cout << "re = "<< re << " im = " << im << endl;
  }
};

int main() {

  // initialize with complx(double, double)
  complx one(1);

  // initialize with a copy of one
  // using complx::complx(const complx&)
  complx two = one;

  // construct complx(3,4)
  // directly into three
  complx three = complx(3,4);

  // initialize with default constructor
  complx four;

  // complx(double, double) and construct
  // directly into five
  complx five = 5;

  one.display();
  two.display();
  three.display();
  four.display();
  five.display();
}
```

上記の例で作成される出力は次のようになります。

```
re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0
```

関連資料:

290 ページの『constructor および destructor』

127 ページの『初期化指定子』

409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

基底クラスおよびメンバーの初期化

コンストラクターは、次に示す 2 とおりの異なった方法でメンバーを初期化できます。コンストラクターは渡された引数を使用して、コンストラクター定義内のメンバー変数を初期化することができます。

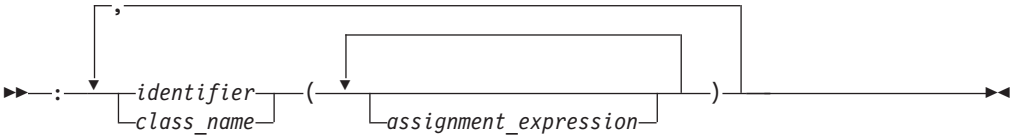
```
complx(double r, double i = 0.0) { re = r; im = i; }
```

またはコンストラクターは、定義の中に*初期化指定子リスト* を含めることができますが、それらは、コンストラクター本体の前に置く必要があります。

```
complx(double r, double i = 0) : re(r), im(i) { /* ... */ }
```

どちらの方法でも、引数値は適切なクラスのデータ・メンバーに割り当てられます。

初期化指定子リストの構文



コンストラクター宣言の一部ではなく、コンストラクター定義の一部として初期化リストをインクルードします。次に例を示します。

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

class B1 {
  int b;
public:
  B1() { cout << "B1::B1()" << endl; };

  // inline constructor
  B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};
class B2 {
  int b;
protected:
  B2() { cout << "B2::B2()" << endl; }

  // noninline constructor
  B2(int i);
};

// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {
  int d1, d2;
public:
  D(int i, int j) : B1(i+1), B2(), d1(i) {
    cout << "D1::D1(int, int)" << endl;
    d2 = j;}
};

int main() {
  D obj(1, 2);
}
```

この例の出力は次のようになります。

```
B1::B1(int)
B2::B2()
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、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型、といったケースを初期化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 {
public:
  A(int) { }
};

class B : public A {
  static const int i;
  const int j;
  int &k;
public:
  B(int& arg) : A(0), j(1), k(arg) { }
};
```

```
int main() {
  int x = 0;
  B obj(x);
};
```

データ・メンバー j および k、さらに基底クラス A は、B のコンストラクターの初期化指定子リストで初期化される必要があります。

クラスのメンバーを初期化する際、データ・メンバーを使用できます。次の例は、このことを示しています。

```
struct A {
  int k;
  A(int i) : k(i) { }
};
struct B: A {
  int x;
  int i;
  int j;
  int& r;
  B(int i): r(x), A(i), j(this->i), i(i) { }
};
```

コンストラクター B(int i) は、次の項目を初期化します。

- B::x を参照するための B::r
- B(int i) への引数の値を指定したクラス A
- B::i の値を指定した B::j
- B(int i) への引数の値を指定した B::i

クラスのメンバーを初期化する場合、メンバー関数 (仮想メンバー関数を含む) を呼び出したり、あるいは演算子 typeid または dynamic_cast を使用することもできます。しかし、すべての基底クラスが初期化される前に、メンバー初期化リストにあるこれらの演算を実行する場合、その動作は、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、次の結果のようになります。

```
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()) の動作は、未定義です。ランタイムは、B::f() を呼び出し、基底 A が初期化されていなくても、A::i にアクセスしようとします。

次の例は、B::B() の初期化指定子が異なる引数を持つ点を除いては、直前の例と同じです。

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

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(5678), j(f()) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

上記の例の出力は、次のとおりです。

```
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()) の動作は、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。

C++11

委任コンストラクター機能が使用可能な場合、初期化は非委任コンストラクター内でのみ実行できます。すなわち、委任コンストラクターは委任および初期化の両方は実行できません。次の例を検討してみます。

```
struct A{
  int x,y;
  A(int x):x(x),y(0){}

  /* the following statement is not allowed */
  A():y(0),A(42) {}
}
```

コンストラクター A() は、A(int x) に委任しますが、A() は初期化も実行します。これは許可されません。コンパイラーはエラーを出して、違反を示します。

詳細については、 412 ページの『委任コンストラクター (C++11)』を参照してください。

C++11

関連資料:

181 ページの『typeid 演算子 (C++ のみ)』

214 ページの『dynamic_cast 演算子 (C++ のみ)』

クラス・オブジェクトのコンストラクター実行順序

コンストラクターを使用してクラス・オブジェクトを作成する場合、コンストラクターの実行順序は以下のとおりです。

1. 仮想基底クラスのコンストラクターが、基底リストに示されている順序で実行される。
2. 非仮想基底クラスのコンストラクターが、宣言の順序で実行される。
3. クラス・メンバーのコンストラクターが、初期化リストの順序に関係なく、宣言の順序で実行される。
4. コンストラクターの本体が、実行される。

次の例は、これらを示しています。

```
#include <iostream>
using namespace std;
struct V {
  V() { cout << "V()" << endl; }
};
struct V2 {
  V2() { cout << "V2()" << endl; }
};
struct A {
  A() { cout << "A()" << endl; }
};
struct B : virtual V {
  B() { cout << "B()" << endl; }
};
struct C : B, virtual V2 {
  C() { cout << "C()" << endl; }
};
struct D : C, virtual V {
  A obj_A;
  D() { cout << "D()" << endl; }
};
int main() {
  D c;
}
```

上記の例の出力は、以下のとおりです。

```
V()
V2()
B()
C()
A()
D()
```

上記の出力は、型 D のオブジェクトを作成するために、C++ ランタイムがコンストラクターを呼び出す順序をリストしています。

C++11

クラス・オブジェクトの構築が委任コンストラクターによって実行される場合、委任コンストラクターの本体は、そのターゲット・コンストラクターの実行後に処理されます。この規則はさらに、ターゲット・コンストラクターが別の委任コンストラクターである場合に、ターゲット・コンストラクターに適用されます。

例:

```
#include <cstdio>
using std::printf;

class X{
public:
    int i,j;
    X();
    X(int x);
    X(int x, int y);
    ~X();
}

X::X(int x):i(x),j(23) {printf("X:X(int)¥n");}
X::X(int x, int y): X(x+y) { printf("X::X(int,int)¥n");}
X::X():X(44,11) {printf("X:X()¥n");}
X::~X() {printf("X::~X()¥n");}

int main(void){
  X x;
}
```

この例の出力は次のとおりです。

```
X::X(int)
X::X(int,int)
X:X()
X::~X()
```

詳細については、 412 ページの『委任コンストラクター (C++11)』を参照してください。

C++11

関連資料:

394 ページの『仮想基底クラス』

デストラクター

デストラクター は、オブジェクトを破棄するときに、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もつメンバー関数で、接頭部として ~ (波形記号) が付きます。次に例を示します。

```
class X {
public:
  // Constructor for class X
```

```
  X();
  // Destructor for class X
  ~X();
};
```

デストラクターは、引数を使用せず、戻りの型もありません。そのアドレスを取得することはできません。デストラクターを const、volatile、const volatile、または static で宣言できません。デストラクターは、virtual、または純粋 virtual で宣言できます。

あるクラスにユーザー定義のデストラクターが必要であるにもかかわらず、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等価です。

```
A::~A() { }
```

コンパイラーは、暗黙的に宣言された C++11 または明示的にデフォルト設定された C++11 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・メンバーを最初に暗黙的に定義してから、暗黙的に宣言された C++11 または明示的にデフォルト設定された C++11 A のデストラクターを定義します。

クラス A のデストラクターは、次のことがすべて真であれば*単純* です。

- これは暗黙的に宣言されるか、C++11 または明示的にデフォルト設定されます C++11 。
- A の直接基底クラスは、すべて単純デストラクターを持っている。
- A のすべての非静的データ・メンバーに関するクラスが、単純デストラクターを持っている。

上記のいずれかが偽であれば、デストラクターは*非単純* です。

共用体メンバーは、非単純デストラクターを持つクラス型にはできません。

クラス型であるクラス・メンバーは、独自のデストラクターを持つことができます。基底クラスと派生クラスは、両方ともデストラクターを持つことができますが、デストラクターは継承されません。基底クラス A または A のメンバーがデストラクターを持っており、A から派生したクラスがデストラクターを宣言しない場合、デフォルトのデストラクターが生成されます。

デフォルト・デストラクターは、基底クラスおよび派生クラスのメンバーのデストラクターを呼び出します。

基底クラスおよびメンバーのデストラクターは、それらのコンストラクターが完了する順番の逆順で呼び出されます。

1. クラス・オブジェクト用のデストラクターは、メンバーおよび基底用のデストラクターより前に呼び出されます。

2. 非静的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。作成された外部オブジェクトや静的オブジェクトのプログラム終了処理時にも、暗黙的にデストラクターを呼び出します。デストラクターは、new 演算子によって作成されたオブジェクトに対してユーザーが delete 演算子を使用すると呼び出されます。

次に例を示します。

```
#include <string>

class Y {
private:
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
  // Destructor
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // ...

  // Destructor ~Y is called before
  // control returns from main()
}
```

デストラクターを明示的に使用してオブジェクトを破棄することもできますが、この方法はお勧めできません。しかし、配置 new 演算子を使用して作成されたオブジェクトを破棄するために、オブジェクトのデストラクターを明示的に呼び出すことができます。次の例は、このことを示しています。

```
#include <new>
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A::A()" << endl; }
```

```
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
  char* p = new char[sizeof(A)];
  A* ap = new (p) A;
  ap->A::~A();
  delete [] p;
}
```

ステートメント A* ap = new (p) A は、型 A の新規のオブジェクトを、フリー・ストレージにではなく、p が割り振ったメモリーに動的に作成します。ステートメント delete [] p は、p が割り振ったストレージを削除します。しかし、ランタイムは、A のデストラクターを明示的に呼び出すまでは (ステートメント ap->A::~A() と指定して)、ap が指すオブジェクトはまだ存在していると判断しています。

注: C++11 デストラクターは、明示的にデフォルトに設定された関数または削除済み関数として宣言できます。詳しくは、266 ページの『明示的にデフォルトに設定された関数』および 267 ページの『削除済み関数』 を参照してください。

C++11

関連資料:

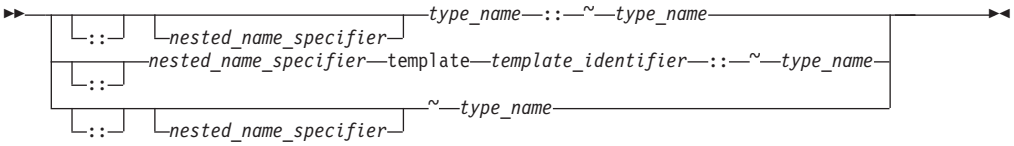
290 ページの『constructor および destructor』

409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

疑似デストラクター

疑似デストラクター は、非クラス型のデストラクターです。

疑似デストラクターの構文



次の例は、整数型の疑似デストラクターを呼び出します。

```
typedef int I;
int main() {
  I x = 10;
  x.I::~I();
  x = 20;
}
```

疑似デストラクターの呼び出し x.I::~I() は、まったく効果はありません。オブジェクト x は、破棄されておらず、割り当て 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。

関連資料:

355 ページの『第 12 章 クラスのメンバーとフレンド (C++ のみ)』

349 ページの『クラス名のスコープ』

ユーザー定義の変換

ユーザー定義の変換 を使用すると、コンストラクターまたは変換関数を使用したオブジェクト変換を指定することができます。初期化指定子、関数引数、関数からの戻り値、式のオペランド、式の反復制御、選択ステートメントの変換、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。

ユーザー定義の変換には次の 2 つのタイプがあります。

- 変換コンストラクター
- 変換関数

単一の値を暗黙的に変換する場合、コンパイラーは、ユーザー定義の型変換を 1 つだけ (型変換コンストラクターまたは型変換関数のどちらか) を使用することができます。次の例は、このことを示しています。

```
class A {
  int x;
public:
  operator int() { return x; };
};

class B {
  A y;
public:
  operator A() { return y; };
};

int main () {
  B b_obj;
//  int i = b_obj;
  int j = A(b_obj);
}
```

コンパイラーは、ステートメント `int i = b_obj` を許可しません。コンパイラーは、暗黙的に `b_obj` を型 `A` のオブジェクトに変換してから (`B::operator A()` を使用して)、暗黙的にそのオブジェクトを整数に変換しなければなりません (`A::operator int()` を使用して)。ステートメント `int j = A(b_obj)` は、暗黙的に `b_obj` を型 `A` のオブジェクトに変換してから、暗黙的にそのオブジェクトを整数に変換します。

ユーザー定義の型変換は、明確でなければなりません。さもないとそれらは呼び出されません。派生クラス型変換関数は、両方の型変換関数が同じ型に変換されない限り、基底クラスにある別の型変換関数を隠しません。関数の多重定義解決は、最適な型変換関数を選択します。次の例は、このことを示しています。

```
class A {
  int a_int;
  char* a_carp;
public:
  operator int() { return a_int; }
  operator char*() { return a_carp; }
};

class B : public A {
  float b_float;
  char* b_carp;
public:
  operator float() { return b_float; }
  operator char*() { return b_carp; }
```

```
};

int main () {
  B b_obj;
//  long a = b_obj;
  char* c_p = b_obj;
}
```

コンパイラーは、ステートメント long a = b_obj を許可しません。コンパイラーは、A::operator int() または B::operator float() のどちらかを使用して、b_obj を long に変換できます。 B::operator char*() が A::operator char*() を隠すので、ステートメント char* c_p = b_obj は、B::operator char*() を使用して、b_obj を char* に変換します。

引数を持つコンストラクターを呼び出し、その引数型を受け入れるコンストラクターが定義されていない場合、標準型変換だけを使用して、その引数を、そのクラスのコンストラクターによって受け入れ可能な別の引数型に変換します。そのクラス用に定義されたコンストラクターに受け入れられる型に引数を変換するために、他のコンストラクターや変換関数を呼び出すことはありません。次の例は、このことを示しています。

```
class A {
public:
  A() { }
  A(int) { }
};

int main() {
   A a1 = 1.234;
//   A moocow = "text string";
}
```

コンパイラーは、ステートメント A a1 = 1.234 を認めます。コンパイラーは、標準型変換を使用して、1.234 を int に変換してから、暗黙的に変換コンストラクター A(int) を呼び出します。コンパイラーは、ステートメント A moocow = "text string" を許可しません。テキスト・ストリングを整数に変換するのは、標準の型変換ではありません。

注: C++11 ユーザー定義の変換を削除済み関数として宣言できます。詳しくは、267 ページの『削除済み関数』を参照してください。 C++11

変換コンストラクター

変換コンストラクター は、関数指定子 explicit を指定せずに宣言される単一パラメーター・コンストラクターです。コンパイラーは、変換コンストラクターを使用して、オブジェクトを第 1 パラメーターの型から、変換コンストラクターのクラス型に変換します。次の例は、このことを示しています。

```
class Y {
  int a, b;
  char* name;
public:
  Y(int i) { };
  Y(const char* n, int j = 0) { };
};

void add(Y) { };

int main() {
```

```
  // equivalent to
  // obj1 = Y(2)
  Y obj1 = 2;

  // equivalent to
  // obj2 = Y("somestring",0)
  Y obj2 = "somestring";

  // equivalent to
  // obj1 = Y(10)
  obj1 = 10;

  // equivalent to
  // add(Y(5))
  add(5);
}
```

上記の例には、次の 2 つの変換コンストラクターがあります。

- `Y(int i)` は、整数をクラス `Y` のオブジェクトに変換するために使用。
- `Y(const char* n, int j = 0)` は、ストリングを指すポインターを、クラス `Y` のオブジェクトに変換するために使用。

コンパイラーは、上記で説明したような型を、`explicit` キーワードを使用して宣言されたコンストラクターで暗黙的に変換しません。コンパイラーは、`new` 式、`static_cast` 式と明示キャスト、および基底とメンバーの初期化で明示的に宣言されたコンストラクターだけを使用します。次の例は、このことを示しています。

```
class A {
public:
  explicit A() { };
  explicit A(int) { };
};

int main() {
  A z;
//  A y = 1;
  A x = A(1);
  A w(1);
  A* v = new A(1);
  A u = (A)1;
  A t = static_cast<A>(1);
}
```

コンパイラーは、これが暗黙の型変換なので、ステートメント `A y = 1` を許可しません。クラス `A` は、型変換コンストラクターを持っていません。

コピー・コンストラクターは、変換コンストラクターです。

関連資料:

218 ページの『new 式 (C++ のみ)』

209 ページの『static_cast 演算子 (C++ のみ)』

明示的変換コンストラクター

`explicit` 関数指定子は、望ましくない暗黙的型変換を制御します。それは、クラス宣言内のコンストラクターの宣言にだけ使用されます。例えば、デフォルトのコンストラクターを除いて、以下のクラスのコンストラクターは、変換コンストラクターです。

```
class A
{  public:
   A();
   A(int);
   A(const char*, int = 0);
};
```

以下の宣言は、正しい宣言です。

```
A c = 1;
A d = "Venditti";
```

最初の宣言は `A c = A(1)` に等価です。

`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割り当てできます。

例えば、以下のステートメントは正しいステートメントです。

```
  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);
```

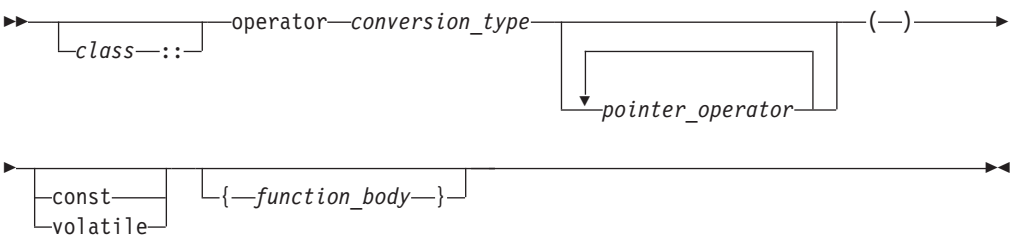
関連資料:

428 ページの『変換コンストラクター』

変換関数

変換関数 と呼ばれる、クラスのメンバー関数を定義することができ、これは、そのクラス型を指定された別の型に変換するものです。

変換関数の構文



クラス `X` に所属する型変換関数は、クラス型 `X` から *conversion_type* で指定する型に変換を指定します。次のコードは `operator int()` という変換関数を示しています。

```
class Y {
  int b;
public:
  operator int();
};
Y::operator int() {
  return b;
}
void f(Y obj) {
  int i = int(obj);
  int j = (int)obj;
  int k = i + obj;
}
```

関数 f(Y) にある 3 つのステートメントはすべて、型変換関数 Y::operator int() を使用します。

クラス、列挙型、typedef 名、関数型、または配列型は、*conversion_type* で宣言、または定義することはできません。型変換関数は、型 A のオブジェクトを、型 A、A の基底クラス、または void に変換するためには使用できません。

変換関数には引数がなく、戻りの型が暗黙的に変換の型になります。変換関数は継承することができます。仮想変換関数は使用できますが、静的変換関数は使用できません。

関連資料:

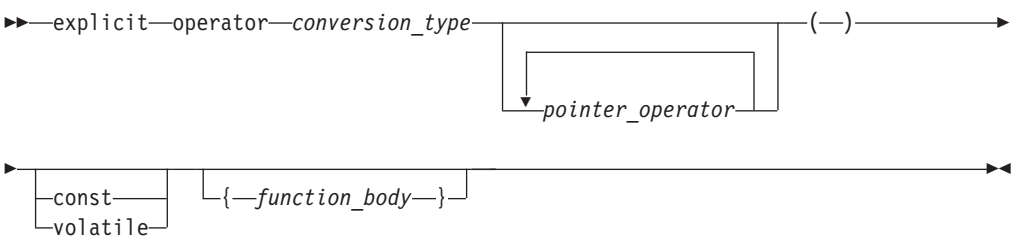
428 ページの『変換コンストラクター』

明示的型変換演算子 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

ユーザー定義の型変換関数の定義に explicit 関数指定子を適用して、意図しない暗黙的変換を適用しないようにできます。このような型変換関数は、明示的型変換演算子と呼ばれます。

明示的型変換演算子の構文



次の例は、explicit 関数指定子で修飾されていないユーザー定義の型変換関数を使用した場合の、意図した暗黙的変換と、意図しない暗黙的変換の両方を示しています。

例 1

```
#include <iostream>

template <class T> struct S {
  operator bool() const;   // conversion function
};

void func(S<int>& s) {

  // The compiler converts s to the bool type implicitly through
  // the conversion function. This conversion might be intended.
  if (s) { }
}

void bar(S<int>& p1, S<float>& p2) {
  // The compiler converts both p1 and p2 to the bool type implicitly
  // through the conversion function. This conversion might be unintended.
  std::cout << p1+p2 << std::endl;

  // The compiler converts both p1 and p2 to the bool type implicitly
  // through the conversion function and compares results.
  // This conversion might be unintended.
  if (p1==p2) { }
}
```

意図しない暗黙的変換を適用しないようにするために、以下のように explicit 関数指定子を使用して例 1 の変換関数を修飾することにより、明示的型変換演算子を定義できます。

```
explicit operator bool() const;
```

例 1 の同じコードに対して明示的型変換演算子を使用したコードをコンパイルすると、コンパイラーは以下のステートメントのエラー・メッセージを発行します。

```
// Error: The call does not match any parameter list for "operator+".
std::cout << p1+p2 << std::endl;

// Error: The call does not match any parameter list for "operator==".
if(p1==p2)
```

明示的型変換演算子による変換を適用する予定の場合は、明示的型変換演算子を以下のステートメントのように明示的に呼び出す必要があります。これによって、例 1 と同じ結果が得られます。

```
std::cout << bool(p1)+bool(p2) << std::endl;

if(bool(p1)==bool(p2))
```

ブール値が期待されるコンテキスト (&&、||、条件演算子が使用される場合、if ステートメントの条件式が評価される場合など) では、明示的 bool 型変換演算子を暗黙的に呼び出すことができます。これにより、前記の明示的型変換演算子を使用した例 1 をコンパイルすると、コンパイラーは、明示的 bool 型変換演算子を使用して、func 関数の s も bool 型に暗黙的に変換します。例 2 も、このことを示しています。

例 2

```
struct T {
  explicit operator bool();   //explicit bool conversion operator
};

int main() {
  T t1;
  bool t2;

  // The compiler converts t1 to the bool type through
  // the explicit bool conversion operator implicitly.
  t1 && t2;

  return 0;
}
```

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、そのパラメーターの残りは (残りがあれば)、デフォルト値を持っています。

クラス A に対してコピー・コンストラクターを宣言しない場合、コンパイラーは、コピー・コンストラクターを暗黙的に宣言し、それはインライン・パブリック・メンバーとなります。

次の例は、暗黙的に定義されたコピー・コンストラクターと、暗黙的なユーザー定義のコピー・コンストラクターを示しています。

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

struct A {
  int i;
  A() : i(10) { }
};

struct B {
  int j;
  B() : j(20) {
    cout << "Constructor B(), j = " << j << endl;
  }

  B(B& arg) : j(arg.j) {
    cout << "Copy constructor B(B&), j = " << j << endl;
  }

  B(const B&, int val = 30) : j(val) {
    cout << "Copy constructor B(const B&, int), j = " << j << endl;
  }
};

struct C {
  C() { }
  C(C&) { }
};

int main() {
  A a;
  A a1(a);
  B b;
```

```
  const B b_const;
  B b1(b);
  B b2(b_const);
  const C c_const;
//  C c1(c_const);
}
```

上記の例の出力は、以下のとおりです。

```
Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30
```

ステートメント A a1(a) は、暗黙的定義のコピー・コンストラクターを使用して、a から新規オブジェクトを作成します。ステートメント B b1(b) は、ユーザー定義のコピー・コンストラクター B::B(B&) を使用して、b から新規オブジェクトを作成します。ステートメント B b2(b_const) は、コピー・コンストラクター B::B(const B&, int) を使用して、新規オブジェクトを作成します。コンパイラーは、第 1 パラメーターとして型 const C& のオブジェクトを取得するコピー・コンストラクターが定義されていないので、ステートメント C c1(c_const) を許可しません。

次のことが真の場合、暗黙的に宣言されたクラス A のコピー・コンストラクターは、A::A(const A&) の書式を持ちます。

- A の直接基底および仮想基底は、第 1 パラメーターが、const または const volatile で修飾されたコピー・コンストラクターを持っている。
- A の非静的クラス型、またはクラス型データ・メンバーの配列は、第 1 パラメーターが、const または const volatile で修飾されたコピー・コンストラクターを持っている。

クラス A に対して上記のことが真でない場合、コンパイラーは、A::A(A&) の書式のコピー・コンストラクターを暗黙的に宣言します。

以下の 1 つ以上の条件が当てはまる場合に、コピー・コンストラクターが暗黙的に定義されているか、C++11 または明示的にデフォルト設定されている C++11 クラス A がプログラムに含まれていれば、そのプログラムは不適格です。

- 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 C++11 または明示的にデフォルト設定された C++11 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 C++11 または明示的にデフォルト設定された C++11 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。

注: C++11 コピー・コンストラクターは、明示的にデフォルトに設定された関数または削除済み関数として宣言できます。詳しくは、 266 ページの『明示的にデフォルトに設定された関数』および 267 ページの『削除済み関数』 を参照してください。 C++11

関連資料:

409 ページの『コンストラクターとデストラクターの概要』

409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

コピー割り当て演算子

コピー割り当て演算子 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 のコピー割り当て演算子は、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割り当て演算子と、暗黙的なユーザー定義の割り当て演算子を示しています。

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

struct A {
  A& operator=(const A&) {
    cout << "A::operator=(const A&)" << endl;
    return *this;
  }

  A& operator=(A&) {
    cout << "A::operator=(A&)" << endl;
    return *this;
  }

};
class B {
  A a;
};

struct C {
  C& operator=(C&) {
    cout << "C::operator=(C&)" << endl;
    return *this;
  }
  C() { }
};

int main() {
  B x, y;
  x = y;

  A w, z;
```

```
  w = z;

  C i;
  const C j();
//  i = j;
}
```

上記の例の出力は、次のとおりです。

```
A::operator=(const A&)
A::operator=(A&)
```

割り当て x = y は、暗黙的に定義された B のコピー割り当て演算子を呼び出します。つまり、ユーザー定義のコピー割り当て演算子 A::operator=(const A&) を呼び出します。割り当て 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、割り当て i = j を許可しません。

次のステートメントが真の場合、クラス A の暗黙的に宣言されたコピー割り当て演算子の書式は、A& A::operator=(const A&) になります。

- 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 であるコピー割り当て演算子を持っている。
- クラス A に属する、型 X の非静的クラス型データ・メンバーが、パラメーターが、型 const X&、const volatile X&、または X であるコピー・コンストラクターを持っている。

クラス A に対して上記のことが真でない場合、コンパイラーは、A& A::operator=(A&) の書式を使用したコピー割り当て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割り当て演算子は、演算子の引数への左辺値参照を戻します。

派生クラスのコピー割り当て演算子は、その基底クラスのコピー割り当て演算子を隠します。

以下の 1 つ以上の条件が当てはまる場合に、コンパイラーは、クラス A のコピー代入演算子が暗黙的に定義されているか、C++11 または明示的にデフォルト設定されている C++11 プログラムを許可できません。

- 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割り当て演算子を持つ。
- クラス A は、アクセス不能なコピー割り当て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 のコピー割り当て演算子は、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割り当て演算子は、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。

注: C++11 コピー割り当て演算子は、明示的にデフォルトに設定された関数または削除済み関数として宣言できます。詳しくは、 266 ページの『明示的にデフォルトに設定された関数』および 267 ページの『削除済み関数』 を参照してください。 C++11

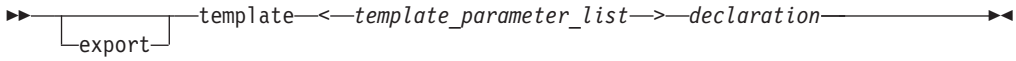
関連資料:

189 ページの『割り当て演算子』

第 15 章 テンプレート (C++ のみ)

テンプレート では、関連するクラスのセット、または関連する関数のセットについて記述し、その宣言のパラメーターのリストでは、そのセットのメンバーが、どのように異なるかを記述します。これらのパラメーターに引数を提供すると、コンパイラーは新規のクラスまたは関数を生成します。このプロセスは、テンプレートの *インスタンス生成* と呼ばれます。これについては、463 ページの『テンプレートのインスタンス生成』で詳しく説明しています。テンプレート、およびテンプレート・パラメーターのセットから生成されたこのクラスまたは関数定義は、468 ページの『テンプレートの特殊化』で説明しているように、*特殊化* と呼ばれます。

テンプレート宣言の構文



コンパイラーは、テンプレートで `export` キーワードを受諾し、暗黙に無視します。

template_parameter_list は、テンプレート・パラメーターをコンマで区切って並べたリストです。これについては、440 ページの『テンプレート・パラメーター』 で説明しています。

宣言 には、次のオプションがあります。

- 関数またはクラスの宣言または定義
- メンバー関数またはクラス・テンプレートのメンバー・クラスの定義
- クラス・テンプレートの静的データ・メンバーの定義
- クラス・テンプレート内のネスト・クラスの静的データ・メンバーの定義
- クラスまたはクラス・テンプレートのメンバー・テンプレートの定義

型 の *ID* が、テンプレート宣言のスコープ内の *type_name* であると定義されます。テンプレート宣言は、名前空間スコープ宣言またはクラス・スコープ宣言として現れます。

次の例は、クラス・テンプレートの使用法を示しています。

```
template<class T> class Key
{
    T k;
    T* kptr;
    int length;
public:
    Key(T);
    // ...
};
```

その後、次の宣言が現れるとします。

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

コンパイラーは、class Key の 3 つのインスタンスを作成します。下表に、これら 3 つのクラス・インスタンスが、テンプレートとしてではなく、通常のクラスとしてソース形式で書き出された場合の、それらクラス・インスタンスの定義を示します。

class Key<int> i;	class Key<char*> c;	class Key<mytype> m;

class Key
{
  int k;
  int * kptr;
  int length;
public:
  Key(int);
  // ...
};
	class Key
	{
	  char* k;
	  char** kptr;
	  int length;
	public:
	  Key(char*);
	  // ...
	};
		class Key
		{
		  mytype k;
		  mytype* kptr;
		  int length;
		public:
		  Key(mytype);
		  // ...
		};

これらの 3 つのクラスには、それぞれ名前があることに注意してください。不等号中括弧の中に含まれている引数は、単にクラス名に対する引数ではなく、クラス名自体の一部です。 Key<int> と Key<char*> は、クラス名です。

テンプレート・パラメーター

テンプレート・パラメーターには、次の 3 つの種類があります。

- 『「型」テンプレート・パラメーター』
- 441 ページの『「非型」テンプレート・パラメーター』
- 441 ページの『「テンプレート」テンプレート・パラメーター』

C++11

テンプレート・パラメーター・パック もテンプレート・パラメーターの一種にすることができます。詳しくは、 476 ページの『可変数引数テンプレート (C++11)』を参照してください。

C++11

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。

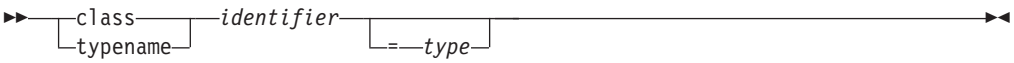
関連資料:

101 ページの『型修飾子』

165 ページの『左辺値と右辺値』

「型」テンプレート・パラメーター

「型」テンプレート・パラメーター宣言の構文



identifier は、型の名前です。

関連資料:

491 ページの『typename キーワード』

「非型」テンプレート・パラメーター

「非型」テンプレート・パラメーターの構文は、次のいずれかの型の宣言と同じです。

- 整数または列挙型
- オブジェクトを指すポインターまたは関数を指すポインター
- オブジェクトに対する左辺値参照または関数に対する左辺値参照
- メンバーを指すポインター
- C++11 std::nullptr_t C++11

配列として宣言された「非型」テンプレート・パラメーターはポインターに変換されます。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、関数を指すポインターに変換されます。以下の例は、これらの規則を示しています。

```
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 *` であり、`&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」non-reference テンプレート・パラメーターは、左辺値ではありません。

関連資料:

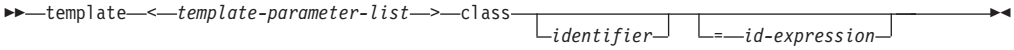
101 ページの『型修飾子』

165 ページの『左辺値と右辺値』

126 ページの『参照 (C++ のみ)』

「テンプレート」テンプレート・パラメーター

「テンプレート」テンプレート・パラメーター宣言の構文



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用法を示しています。

```
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引数

テンプレート・パラメーターは、デフォルトの引数を持っています。デフォルトのテンプレート引数のセットは、任意のテンプレートの宣言すべてに累積していきます。次の例は、このことを示しています。

```
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
   public:
      T x;
      U y;
};

A<> a;
```

メンバー a.x の型は float で、a.y の型は int です。

デフォルトの引数を、同じスコープ内の異なる宣言にある、同じテンプレート・パラメーターに与えることはできません。コンパイラーは、次の例を許可しません。

```
template<class T = char> class X;
template<class T = char> class X { };
```

あるテンプレート・パラメーターが、デフォルトの引数を持つ場合、それに続くテンプレート・パラメーターも、すべてデフォルトの引数を持つはずです。例えば、コンパイラーは次のコードを許可しません。

```
template<class T = char, class U, class V = int> class X { };
```

テンプレート・パラメーター U は、デフォルトの引数が必要です。あるいは T のデフォルトを除去する必要があります。

テンプレート・パラメーターのスコープは、その宣言のポイントから始まって、そのテンプレート定義の終わりで終了します。つまり、他のテンプレート・パラメーター宣言内のテンプレート・パラメーターの名前、およびそれらのデフォルトの引数を使用できるということです。次の例は、このことを示しています。

```
template<class T = int> class A;
template<class T = float> class B;
template<class V, V obj> class C;
// a template parameter (T) used as the default argument
// to another template parameter (U)
template<class T, class U = T> class D { };
```

フレンドとしてのテンプレート・パラメーターの命名 (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 標準では、テンプレート・パラメーターをフレンドとして宣言できる拡張フレンド宣言機能が導入されました。これにより、テンプレート内部のフレンド宣言を簡単に使用できます。

フレンド宣言でテンプレート・パラメーターが解決される場合、このフレンド宣言で詳述型指定子は使用できません。使用すると、コンパイラーがエラーを出します。

関連資料:

372 ページの『フレンド』

テンプレート引数

下記の 3 つのテンプレート・パラメーター型に対応する、3 種類のテンプレート引数があります。

- 『テンプレート「型」引数』
- 444 ページの『テンプレート「非型」引数』
- 446 ページの『テンプレート「テンプレート」引数』

テンプレート引数は、テンプレートに宣言された対応パラメーターが指定する型、およびフォームと一致しなければなりません。

C++11

テンプレートで宣言されるパラメーターがテンプレート・パラメーター・パックである場合、0 個以上のテンプレート引数に対応します。詳細については、476 ページの『可変数引数テンプレート (C++11)』を参照してください。

C++11

テンプレート・パラメーターのデフォルト値を使用するには、対応するテンプレート引数を省略します。ただし、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る場合でも、不等号括弧 <> 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。

関連資料:

2 ページの『ブロック/ローカル・スコープ』

10 ページの『リンケージなし』

ビット・フィールド・メンバー

89 ページの『typedef 定義』

テンプレート「型」引数

次のいずれかの型 を、「型」テンプレート・パラメーターのテンプレート引数として使用することはできません。

- ローカル型

- リンケージなしの型
- 無名型
- 上記の型のいずれかを複合した型

テンプレート引数が、型なのか、式なのかあいまいな場合は、テンプレート引数は、型であると見なされます。次の例は、このことを示しています。

```
template<class T> void f() { };
template<int i> void f() { };

int main() {
  f<int()>();
}
```

関数呼び出し `f<int()>()` は、`T` をテンプレート引数として、関数を呼び出します。このときコンパイラーは、`int()` を型と見なし、したがって暗黙的に最初の `f()` をインスタンス化し、呼び出します。

関連資料:

2 ページの『ブロック/ローカル・スコープ』

10 ページの『リンケージなし』

ビット・フィールド・メンバー

89 ページの『typedef 定義』

テンプレート「非型」引数

テンプレート引数リスト内で提供されている「非型」テンプレート引数は、コンパイル時に値が決められる式です。このような引数は、定数式、関数のアドレス、外部リンケージのあるオブジェクト、または静的クラス・メンバーのアドレスでなければなりません。通常は、「非型」テンプレート引数を使用して、クラスの初期化またはクラス・メンバーのサイズを指定します。

非型整数引数の場合、インスタンス引数は、そのパラメーター型に適した値と符号がある限りは、対応するテンプレート・パラメーターと一致します。

非型アドレス引数の場合、インスタンス引数の型は、*identifier* または *&identifier* の形式でなければなりません。また、インスタンス引数の型は、マッチングの前に、関数名が関数型を指すポインターに変更される点以外は、正確にテンプレート・パラメーターと一致していなければなりません。

テンプレート引数リスト内に「非型」テンプレート引数がある場合、結果として得られる値は、そのテンプレート・クラス型の一部を形成します。 2 つのテンプレート・クラス名が同じテンプレート名を持っており、それらの引数の値が同じ場合、それらは同じクラスであるといいます。

次の例では、クラス・テンプレートが、型引数だけでなく、「非型」テンプレート `int` 引数も必要であると定義されています。

```
template<class T, int size> class Myfilebuf
{
      T* filepos;
      static int array[size];
public:
```

```
      Myfilebuf() { /* ... */ }
      ~Myfilebuf();
      advance(); // function defined elsewhere in program
};
```

この例では、テンプレート引数 size が、テンプレート・クラス名の一部になります。このようなテンプレート・クラスのオブジェクトは、クラス型引数 T と「非型」テンプレート引数 size の値の両方を指定して作成されます。

オブジェクト x、およびそれに対応するテンプレート・クラス (引数 double と size=200 を持つ) は、2 番目のテンプレート引数の値を指定することにより、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 は、演算式を使用しても作成できます。

```
Myfilebuf<double,10*20> x;
```

これらの式によって作成されるオブジェクトは、テンプレート引数の評価が同じになるので、同一になります。最初の式の中の値 200 は、2 番目の構成に示すように、コンパイル時の結果が 200 に等しいということがわかっている式によって表すこともできます。

注: < シンボル、または > シンボルを含む引数が、実際には関係演算子として使用される場合は、それらの引数がテンプレート引数リスト区切り文字として解析されないように、括弧で囲む必要があります。 例えば、次の定義の中の引数は有効です。

```
Myfilebuf<double, (75>25)> x;       // valid
```

ただし次の定義は、より大の演算子 (>) がテンプレート引数リストの終了区切り文字と解釈されるので、有効ではありません。

```
Myfilebuf<double, 75>25> x;         // error
```

テンプレート引数の評価結果が同一でなければ、作成されたオブジェクトは異なる型になります。

```
Myfilebuf<double,200> x;            // create object x of class
                                    // Myfilebuf<double,200>
Myfilebuf<double,200.0> y;          // error, 200.0 is a double,
                                    // not an int
```

y のインスタンス生成は、値 200.0 の型が double で、テンプレート引数の型が int であるために失敗します。

次の 2 つのオブジェクトは、

```
      Myfilebuf<double, 128> x
      Myfilebuf<double, 512> y
```

分離テンプレート特殊化のオブジェクトです。後でこれらのオブジェクトのいずれかを Myfilebuf<double> で参照するとエラーになります。

クラス・テンプレートは、非型引数を持つ場合は、型引数を持つ必要がありません。例えば、次のテンプレートは有効なクラス・テンプレートです。

```
template<int i> class C
{
      public:
            int k;
            C() { k = i; }
};
```

このクラス・テンプレートは、次のような宣言でインスタンスを生成できます。

```
class C<100>;
class C<200>;
```

繰り返しですが、これら 2 つの宣言は、これらの非型引数の値が異なるので、別個のクラスを参照しています。

関連資料:

169 ページの『整数定数式』

126 ページの『参照 (C++ のみ)』

9 ページの『外部リンケージ』

365 ページの『静的メンバー』

テンプレート「テンプレート」引数

「テンプレート」テンプレート・パラメーターのテンプレート引数は、クラス・テンプレートの名前です。

コンパイラーが「テンプレート」テンプレート引数と一致するテンプレートの検索を試みる場合、それは主クラス・テンプレートだけを検索します。 (主テンプレート は、特殊化されているテンプレートです。) 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>
#include <typeinfo>
```

```
using namespace std;

template<class T, class U> class A {
public:
   int x;
};

template<class U> class A<int, U> {
public:
   short x;
};

template<template<class T, class U> class V> class B {
   V<int, char> i;
   V<char, char> j;
};

B<A> c;

int main() {
   cout << typeid(c.i.x).name() << endl;
   cout << typeid(c.j.x).name() << endl;
}
```

上記の例の出力は、以下のとおりです。

```
short
int
```

宣言 `V<int, char> i` は、部分的な特殊化を使用しますが、宣言 `V<char, char> j` は、主テンプレートを使用します。

関連資料:

473 ページの『部分的特殊化』

463 ページの『テンプレートのインスタンス生成』

クラス・テンプレート

クラス・テンプレートと個々のクラスとの間の関係は、クラスと個々のオブジェクトとの間の関係に似ています。個々のクラスがオブジェクトのグループの構成方法を定義し、一方、クラス・テンプレートがクラスのグループの生成方法を定義します。

クラス・テンプレート とテンプレート・クラス という用語の間の区別に注意してください。

クラス・テンプレート
: これは、テンプレート・クラスの生成に使用されるテンプレートです。クラス・テンプレートのオブジェクトは、宣言できません。

テンプレート・クラス
: クラス・テンプレートのインスタンスです。

テンプレート定義は、下記の点を除いて、テンプレートが生成し得る有効なクラス定義のいずれとも同一です。

- クラス・テンプレート定義には、次の語が先行します。

  `template< `*`template-parameter-list`*` >`

ここで、*template-parameter-list* は、次に示す種類のテンプレート・パラメーターの 1 つ以上を、コンマで区切ったリストです。

- 型
- 非型
- テンプレート

- 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。

C++11

テンプレート・パラメーター・パック もクラス・テンプレートのパラメーターの一種にすることができます。詳しくは、476 ページの『可変数引数テンプレート (C++11)』を参照してください。

C++11

詳述型指定子を使用して定義しなくても、クラス・テンプレートを宣言することができます。次に例を示します。

```
template<class L, class T> class Key;
```

これにより、名前がクラス・テンプレート名として予約されます。クラス・テンプレートのテンプレート宣言は、すべてが、同じ型と同じ数のテンプレート引数を持っていなければなりません。クラス定義を含む 1 つのテンプレート宣言だけが許可されます。

C++11

テンプレート・パラメーター・パックを使用することで、クラス・テンプレートのテンプレート宣言は、クラス・テンプレートで指定されたパラメーター数よりも少ない引数、または多くの引数を持つことが可能です。

C++11

注: テンプレート引数リストがネストされている場合は、内側のリストの終わりの `>` と外側のリストの終わりの `>` の間に分離スペースが必要です。これがなければ、抽出演算子 `>>` と 2 つのテンプレート・リスト区切り文字 `>` との区別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
}
```

C++11

右不等号括弧の機能を有効にすると、`>>` トークンは、以下の両方の条件が真の場合、2 つの連続する `>` トークンとして扱われます。

- >> トークンが、1 つ以上の左不等号括弧がアクティブであるコンテキスト内にある。左不等号括弧はアクティブであるのは、左不等号括弧に対応する右不等号括弧がまだない場合です。
- >> トークンが、区切り文字で区切られた式コンテキスト内にネストされていない。

最初の > トークンが template_parameter_list のコンテキスト内にある場合、そのトークンは template_parameter_list の終了区切り文字として扱われます。それ以外の場合は、より大演算子として扱われます。2 番目の > トークンは、囲んでいる template_id 構成体、または別の構成体 (const_cast、dynamic_cast reinterpret_cast、static_cast 演算子など) を終了します。次に例を示します。

```
template<typename T> struct list {};
template<typename T>

struct vector
{
  operator T() const;
};

int main()
{
  // Valid, same as vector<vector<int> > v;
  vector<vector<int>> v;

  // Valid, treat the >> token as two consecutive > tokens.
  // The first > token is treated as the ending delimiter for the
  // template_parameter_list, and the second > token is treated as
  // the ending delimiter for the static_cast operator.
  const vector<int> vi = static_cast<vector<int>>(v);
}
```

括弧で囲んだ式は、区切り文字で区切られた式コンテキストです。
template-argument-list 内でビット単位のシフト演算子を使用するには、括弧を使用して演算子を囲みます。次に例を示します。

```
template <int i> class X {};
template <class T> class Y {};

Y<X<(6>>1)>> y;          //Valid: 6>>1 uses the right shift operator
```

C++11

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にアクセスすることができます。次のクラス・テンプレートがあるとします。

```
template<class T> class Vehicle
{
public:
    Vehicle() { /* ... */ }    // constructor
    ~Vehicle() {};             // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

そして、次の宣言を行います。

```
Vehicle<char> bicycle; // instantiates the template
```

コンストラクター、構成オブジェクト、およびメンバー関数 drive() は、次のいずれかを指定してアクセスできます (標準ヘッダー・ファイル string.h が、プログラム・ファイルに含まれているとします)。

コンストラクター	`Vehicle<char> bicycle;`
	
	`// constructor called automatically,`
	`// object bicycle created`

オブジェクト bicycle	`strcpy (bicycle.kind, "10 speed");` `bicycle.kind[0] = '2';`
関数 drive()	`char* n = bicycle.drive();`
関数 roadmap()	`Vehicle<char>::roadmap();`

関連資料:

346 ページの『クラス型の宣言』

349 ページの『クラス名のスコープ』

357 ページの『メンバー関数』

209 ページの『static_cast 演算子 (C++ のみ)』

214 ページの『dynamic_cast 演算子 (C++ のみ)』

212 ページの『const_cast 演算子 (C++ のみ)』

211 ページの『reinterpret_cast 演算子 (C++ のみ)』

クラス・テンプレートの宣言と定義

クラス・テンプレートを宣言してから、対応するテンプレート・クラスのインスタンス生成を行う必要があります。クラス・テンプレートの定義は、1 つの変換単位内で 1 回しか使用できません。クラス・テンプレートは、クラスのサイズを必要とする、またはクラスのメンバーを参照する、テンプレート・クラスを使用する前に定義する必要があります。

次の例では、クラス・テンプレート Key は、定義の前に宣言されます。クラスのサイズは必要ないので、ポインター keyiptr の宣言は有効です。ただし、keyi の宣言はエラーになります。

```
template  <class L> class Key;      // class template declared,
                                    // not defined yet
                                    //
class Key<int> *keyiptr;            // declaration of pointer
                                    //
class Key<int> keyi;                // error, cannot declare keyi
                                    // without knowing size
                                    //
template <class L> class Key        // now class template defined
{ /* ... */ };
```

クラス・テンプレートを定義する前に、対応するテンプレート・クラスを使用すると、コンパイラーはエラーを発行します。テンプレート・クラス名の形式をもつクラス名は、テンプレート・クラスであると見なされます。言い換えれば、テンプレート・クラスの場合は、不等号括弧が有効なのは、クラス名の中だけです。

上記の例では、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 を宣言しています。 keyiptr は、詳述型指定子なしで宣言することもできます。

```
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
{ /* ... */ };
```

関連資料:

447 ページの『クラス・テンプレート』


「XL C/C++ コンパイラー・リファレンス」の中の『-qtmplparse』を参照

静的データ・メンバーとテンプレート

各クラス・テンプレートのインスタンス生成は、静的データ・メンバーのそれ自身のコピーを所有しています。静的宣言は、テンプレート引数型または任意の定義された型です。

別々に静的メンバーを定義する必要があります。次の例は、このことを示しています。

```
template <class T> class K
{
public:
      static T x;
};
template <class T> T K<T> ::x;

int main()
{
      K<int>::x = 0;
}
```

ステートメント template T K::x は、クラス K の静的メンバーを定義し、main() 関数内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。

関連資料:

365 ページの『静的メンバー』

クラス・テンプレートのメンバー関数

テンプレートのメンバー関数を、そのクラス・テンプレート定義の外側に定義できます。

クラス・テンプレート特殊化のメンバー関数を呼び出す場合、コンパイラーは、以前、クラス・テンプレートの作成に使用したテンプレート引数を使用します。次の例は、このことを示しています。

```
template<class T> class X {
   public:
      T operator+(T);
};
```

```
template<class T> T X<T>::operator+(T arg1) {
   return arg1;
};

int main() {
   X<char> a;
   X<int> b;
   a +'z';
   b + 4;
}
```

多重定義された加算演算子は、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') と等価です。ステートメント b + 4 は、b.operator+(4) と等価です。

C++11

後置戻り型をテンプレートのメンバー関数に使用できます。これには、以下の種類の戻りの型を持つものが含まれます。

- 関数引数の型に依存する戻りの型
- 複雑な戻りの型

詳しくは、 283 ページの『後置戻り型 (C++11)』を参照してください。

C++11

関連資料:

357 ページの『メンバー関数』

フレンドとテンプレート

テンプレートを必要とする場合、クラスとそれらのフレンドとの間には 4 種類の関係があります。

- *1 対多*: 非テンプレート関数は、すべてのテンプレート・クラスのインスタンス生成へのフレンドです。
- *多対 1*: テンプレート関数のすべてのインスタンス生成は、通常の非テンプレート・クラスへのフレンドです。
- *1 対 1*: テンプレート引数の 1 セットを使用したテンプレート関数のインスタンス生成は、同じテンプレート引数のセットを使用してインスタンス生成された 1 つのテンプレート・クラスのフレンドです。これは、通常の非テンプレート・クラスと通常の非テンプレート・フレンド関数との間の関係でもあります。
- *多対多*: テンプレート関数のすべてのインスタンス生成は、テンプレート・クラスのすべてのインスタンス生成へのフレンドです。

次の例は、これらの関係を示しています。

```
class B{
   template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
   friend int e();
```

```
   friend int f(T);
   friend int g<T>();
   template<class U> friend int h();
};
```

- 関数 e() は、クラス A と 1 対多の関係を持ちます。関数 e() は、クラス A のすべてのインスタンス生成のフレンドです。
- 関数 f() は、クラス A と 1 対 1 の関係を持ちます。コンパイラーは、この種の宣言に対して、以下に類似する警告を出します。

  ```
  The friend function declaration "f" will cause an error when the enclosing
  template class is instantiated with arguments that declare a friend function
  that does not match an existing definition. The function declares only one
  function because it is not a template but the function type depends on
  one or more template parameters.
  ```

- 関数 g() は、クラス A と 1 対 1 の関係を持ちます。関数 g() は、関数テンプレートです。この前に宣言する必要があります。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 のインスタンス生成ごとに、g() にマッチングするインスタンス生成が 1 つあります。例えば、g<int> は、A<int> のフレンドです。
- 関数 h() は、クラス A と多対多の関係を持ちます。関数 h() は、関数テンプレートです。 A のすべてのインスタンス生成にとって、h() のインスタンス生成は、すべてフレンドです。
- 関数 j() は、クラス B と多対 1 の関係を持ちます。

これらの関係は、フレンド・クラスにも適用します。

関連資料:

372 ページの『フレンド』

関数テンプレート

関数テンプレート は、関数のグループの生成方法を定義します。

非テンプレート関数は、テンプレートから生成された特殊化の関数と同じ名前、およびパラメーター・プロファイルを持っていたとしても、非テンプレート関数は、関数テンプレートとは関連がありません。非テンプレート関数は、関数テンプレートの特殊化と見なされることは、ありません。

次の例は、quicksort という名前の関数テンプレートを使用した、クイック・ソート・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
```

クイック・ソート・アルゴリズムは、型 T の配列 (これの関係演算子および割り当て演算子は、定義されている) をソートします。テンプレート関数は、1 つのテンプレート引数と 3 つの関数引数を取ります。

- ソートされる配列の型、T
- ソートされる配列の名前、a
- 配列の下限、leftarg
- 配列の上限、rightarg

上記の例では、次のステートメントを使用して、quicksort() テンプレート関数を呼び出すこともできます。

```
  quicksort(sortme, 0, 10 - 1);
```

コンパイラーが、テンプレート関数呼び出しの使用法とコンテキストにより、テンプレート引数を推定できる場合、テンプレート引数を省略できます。ここでは、コンパイラーは、sortme が、型 int の配列であると推定します。

C++11

テンプレート・パラメーター・パック は、関数テンプレートのテンプレート・パラメーターの一種であり、*関数仮パラメーター・パック* は、関数テンプレートの関数仮パラメーターの一種です。詳しくは、476 ページの『可変数引数テンプレート (C++11)』を参照してください。

後置戻り型を関数テンプレートに使用できます。これには、以下の種類の戻りの型を持つものが含まれます。

- 関数引数の型に依存する戻りの型
- 複雑な戻りの型

詳しくは、 283 ページの『後置戻り型 (C++11)』を参照してください。

C++11 ◀

テンプレート引数の推定

テンプレート関数を呼び出す場合、テンプレート関数呼び出しの使用法とそのコンテキストにより、コンパイラーが決定または*推定* できるテンプレート引数は、いずれでも省略できます。

コンパイラーは、対応するテンプレート・パラメーターの型と、関数呼び出しで使用される引数の型を比較することにより、テンプレート引数を推定しようとします。テンプレート引数の推定を行うためには、コンパイラーが比較する 2 つの型 (テンプレート・パラメーターと関数呼び出しで使用される引数) は、ある特定の構造体でなければなりません。下記に、これらの型構造体をリストします。

```
T
const T
volatile T
T&
T&&
T*
T[10]
A<T>
C(*)(T)
T(*)()
T(*)(U)
T C::*
C T::*
T U::*
T (C::*)()
C (T::*)()
D (C::*)(T)
C (T::*)(U)
T (C::*)(U)
T (U::*)()
T (U::*)(V)
E[10][i]
B<i>
TT<T>
TT<i>
TT<C>
```

- T、U、および V は、テンプレート型引数を表す
- 10 は、整数定数を示す
- i は、テンプレート非型引数を示す
- [i] は、参照またはポインター型の配列境界、あるいは標準配列の非主配列の境界を示す
- TT は、「テンプレート」テンプレート引数を示す
- (T)、(U)、および (V) は、少なくとも 1 つのテンプレート型引数を持つ引数リストを示す

- () は、テンプレート引数を持っていない引数リストを示す
- <T> は、少なくとも 1 つのテンプレート型引数を持つテンプレート引数リストを示す
- <i> は、少なくとも 1 つのテンプレート非型引数を持つテンプレート引数リストを示す
- <C> は、テンプレート・パラメーターに従属するテンプレート引数を持っていない、テンプレート引数リストを示す

次の例は、これら型構造体のそれぞれの使用法を示しています。例では、引数として上記の各構造体を使用して、テンプレート関数を宣言しています。そして、これらの関数が、宣言の順に (テンプレート引数を使用せずに) 呼び出されます。この例は、型構造体のリストと同様なものを出力します。

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

template<class T> class A { };
template<int i> class B { };

class C {
   public:
      int x;
};

class D {
   public:
      C y;
      int z;
};

template<class T> void f (T)          { cout << "T" << endl; };
template<class T> void f1(const T)    { cout << "const T" << endl; };
template<class T> void f2(volatile T) { cout << "volatile T" << endl;  };
template<class T> void g (T*)         { cout << "T*" << endl; };
template<class T> void g (T&)         { cout << "T&" << endl; };
template<class T> void g1(T[10])      { cout << "T[10]" << endl;};
template<class T> void h1(A<T>)       { cout << "A<T>" << endl; };

void test_1() {
   A<char> a;
   C c;

   f(c);   f1(c);   f2(c);
   g(c);   g(&c);   g1(&c);
   h1(a);
}

template<class T>          void j(C(*)(T)) { cout << "C(*) (T)" << endl; };
template<class T>          void j(T(*)())  { cout << "T(*) ()" << endl; }
template<class T, class U> void j(T(*)(U)) { cout << "T(*) (U)" << endl; };

void test_2() {
   C (*c_pfunct1)(int);
   C (*c_pfunct2)(void);
   int (*c_pfunct3)(int);
   j(c_pfunct1);
   j(c_pfunct2);
   j(c_pfunct3);
}

template<class T>          void k(T C::*) { cout << "T C::*" << endl; };
template<class T>          void k(C T::*) { cout << "C T::*" << endl; };
template<class T, class U> void k(T U::*) { cout << "T U::*" << endl; };
```

```
void test_3() {
   k(&C::x);
   k(&D::y);
   k(&D::z);
}

template<class T>     void m(T (C::*)() )
   { cout << "T (C::*)()" << endl; };
template<class T>     void m(C (T::*)() )
   { cout << "C (T::*)()" << endl; };
template<class T>     void m(D (C::*)(T))
   { cout << "D (C::*)(T)" << endl; };
template<class T, class U>  void m(C (T::*)(U))
   { cout << "C (T::*)(U)" << endl; };
template<class T, class U>  void m(T (C::*)(U))
   { cout << "T (C::*)(U)" << endl; };
template<class T, class U>  void m(T (U::*)() )
   { cout << "T (U::*)()" << endl; };
template<class T, class U, class V> void m(T (U::*)(V))
   { cout << "T (U::*)(V)" << endl; };

void test_4() {
   int (C::*f_membp1)(void);
   C (D::*f_membp2)(void);
   D (C::*f_membp3)(int);
   m(f_membp1);
   m(f_membp2);
   m(f_membp3);

   C (D::*f_membp4)(int);
   int (C::*f_membp5)(int);
   int (D::*f_membp6)(void);
   m(f_membp4);
   m(f_membp5);
   m(f_membp6);

   int (D::*f_membp7)(int);
   m(f_membp7);
}

template<int i> void n(C[10][i]) { cout << "E[10][i]" << endl; };
template<int i> void n(B<i>)     { cout << "B<i>" << endl; };

void test_5() {
   C array[10][20];
   n(array);
   B<20> b;
   n(b);
}

template<template<class> class TT, class T> void p1(TT<T>)
   { cout << "TT<T>" << endl; };
template<template<int> class TT, int i>     void p2(TT<i>)
   { cout << "TT<i>" << endl; };
template<template<class> class TT>          void p3(TT<C>)
   { cout << "TT<C>" << endl; };

void test_6() {
   A<char> a;
   B<20> b;
   A<C> c;
   p1(a);
   p2(b);
   p3(c);
```

```
}

int main() { test_1(); test_2(); test_3(); test_4(); test_5(); test_6(); }
```

「型」テンプレート引数の推定

コンパイラーは、リストされたいくつかの型構造体で構成される型から、テンプレート引数を推定できます。次の例は、いくつかの型構造体で構成される型からのテンプレート引数の推定を示しています。

```
template<class T> class Y { };

template<class T, int i> class X {
   public:
      Y<T> f(char[20][i]) { return x; };
      Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
   void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
   Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
   g(p);
}
```

型 `Y<int> (X<int, 20>::*p)(char[20][20]) T<U> (V::*)(W[20][i])` は、型構造体 `T (U::*)(V)` に基づいています。

- `T` は、`Y<int>` です
- `U` は、`X<int, 20>` です。
- `V` は、`char[20][20]` です

型が属するクラスを使用してその型を修飾し、そのクラス (*ネストされた名前指定子*) がテンプレート・パラメーターに依存する場合、コンパイラーは、そのパラメーターのテンプレート引数を推定できません。型が、この理由により推定できないテンプレート引数を含んでいる場合、その型にあるすべてのテンプレート引数は、推定されません。次の例は、このことを示しています。

```
template<class T, class U, class V>
  void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
  Y<int>::Z<char> a;
  Y<int> b;
  Y<float> c;

  h<int, char, float>(a, b, c);
  h<int, char>(a, b, c);
  // h<int>(a, b, c);
}
```

コンパイラーは、`typename Y<T>::template Z<U>` のテンプレート引数 `T` および `U` を推定できません (しかし、`Y<T>` の `T` は推定します)。コンパイラーは、`U` がそのコンパイラーによって推定されないので、テンプレート関数呼び出し `h<int>(a, b, c)` を許可しません。

コンパイラーは、関数を指すポインターから、またはいくつかの多重定義関数名を与えられたメンバー関数引数を指すポインターから、関数テンプレート引数を推定

できます。しかし、多重定義関数が、いずれも関数テンプレートではないこともあるし、複数の多重定義関数が、要求される型と一致しないこともあります。次の例は、このことを示しています。

```
template<class T> void f(void(*) (T,int)) { };

template<class T> void g1(T, int) { };

void g2(int, int) { };
void g2(char, int) { };

void g3(int, int, int) { };
void g3(float, int) { };

int main() {
//   f(&g1);
//   f(&g2);
   f(&g3);
}
```

コンパイラーは、g1() が関数テンプレートなので、呼び出し f(&g1) を許可しません。コンパイラーは、g2() という名前の関数が、両方とも f() が必要とする型と一致するので、呼び出し f(&g2) を許可しません。

コンパイラーは、デフォルト引数の型からテンプレート引数を推定できません。次の例は、このことを示しています。

```
template<class T> void f(T = 2, T = 3) { };

int main() {
   f(6);
//   f();
   f<int>();
}
```

コンパイラーは、関数呼び出しの引数値からテンプレート引数 (int) を推定できるので、呼び出し f(6) を許可します。コンパイラーは、f() のデフォルトの引数からテンプレート引数を推定できないので、呼び出し f() を許可しません。

コンパイラーは、「非型」テンプレート引数の型からテンプレート型引数を推定できません。例えば、コンパイラーは次のコードを許可しません。

```
template<class T, T i> void f(int[20][i]) { };

int main() {
   int a[20][30];
   f(a);
}
```

コンパイラーは、テンプレート・パラメーター T の型を推定できません。

C++11

関数テンプレートのテンプレート型パラメーターが cv 修飾されていない右辺値参照であるが、関数呼び出しの引数が左辺値である場合は、右辺値参照ではなく、対応する左辺値参照が使用されます。しかし、テンプレート型パラメーターが cv 修飾された右辺値参照であり、関数呼び出しの引数が左辺値である場合は、テンプレートのインスタンス生成に失敗します。次に例を示します。

```
template <class T> double func1(T&&);
template <class T> double func2(const T&&);

int var;

// The compiler calls func1<int&>(int&)
double a = func1(var);

// The compiler calls func1<int>(int&&)
double b = func1(1);

// error
double c = func2(var);

// The compiler calls func2<int>(const int&&)
double d = func2(1);
```

この例において、関数テンプレート func1 のテンプレート型パラメーターは cv 修飾されていない右辺値参照であり、関数テンプレート func2 のテンプレート型パラメーターは cv 修飾された右辺値参照です。変数 a の初期化において、テンプレート引数 var は左辺値です。したがって、関数テンプレート func1 のインスタンス生成では左辺値参照型 int& が使用されます。変数 b の初期化において、テンプレート引数 1 は右辺値です。したがって、テンプレートのインスタンス生成では右辺値参照型 int&& のままになります。c の初期化では、テンプレート型パラメーター T&& が cv 修飾されていますが、var は左辺値です。したがって、var を右辺値参照 T&& にバインドすることはできません。

C++11

「非型」テンプレート引数の推定

コンパイラーは、境界が参照またはポインター型を参照しない限り、主配列の境界の値を推定できません。主配列の境界は、関数仮パラメーター型の一部ではありません。次のコードは、このことを示しています。

```
template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
   int b[10][20];
   int c[10];
   f(b);
   // g(c);
   h(c);
}
```

コンパイラーは、呼び出し g(c) を許可しません。コンパイラーは、テンプレート引数 i を推定できません。

コンパイラーは、テンプレート関数のパラメーター・リストの式で使用されている、「非型」テンプレート引数の値を推定できません。次の例は、このことを示しています。

```
template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
```

```
  X<0> a;
  f<1>(a);
  // f(a);
}
```

関数 f() をオブジェクト a で呼び出すためには、関数が、型 X<0> の引数を受諾する必要があります。しかし、コンパイラーは、関数テンプレート引数型 X<i - 1> が X<0> と等価であるためには、テンプレート引数 i が、1 と等しい必要があるということを推定できません。したがって、コンパイラーは、関数呼び出し f(a) を許可しません。

コンパイラーに「非型」テンプレート引数を推定させたい場合、パラメーターの型が、関数呼び出しで使用される値の型と正確に一致しなければなりません。例えば、コンパイラーは次のコードを許可しません。

```
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。

C++11

テンプレート引数の推定は、可変数引数テンプレート機能にも適用されます。詳しくは、476 ページの『可変数引数テンプレート (C++11)』を参照してください。

C++11

関連資料:

126 ページの『参照 (C++ のみ)』

165 ページの『左辺値と右辺値』

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数の推定を試み、明示的に宣言されたテンプレート引数をチェックします。成功すれば、コンパイラーは、関数テンプレート特殊化のインスタンスを作成してから、この特殊化を多重定義解決で使用する*候補関数* のセットに追加します。コンパイラーは、多重定義解決を続け、候補関数のセットから最も適切な関数を選択します。非テンプレート関数は、テンプレート関数より優先順位があります。次の例は、このことを説明しています。

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

template<class T> void f(T x, T y) { cout << "Template" << endl; }
```

```
void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');
}
```

上記の例の出力は、以下のとおりです。

```
Non-template
Template
Non-template
```

関数呼び出し `f(1, 2)` は、テンプレート関数と非テンプレート関数の両方の引数型と一致します。多重定義の解決では非テンプレート関数の方が優先順位が高いと見なされるため、非テンプレート関数が呼び出されます。

関数呼び出し `f('a', 'b')` は、テンプレート関数の引数型とだけ一致します。テンプレート関数が呼び出されます。

関数呼び出し `f(1, 'b')` では、引数の推定は失敗します。コンパイラーは、テンプレート関数特殊化を生成せず、また、多重定義解決も生じません。非テンプレート関数は、関数引数 `'b'` に、標準型変換を使用して `char` から `int` に変換した後で、この関数呼び出しを解決します。

関連資料:

336 ページの『多重定義解決』

関数テンプレートの部分選択

関数テンプレート特殊化は、テンプレート引数の推定が、特殊化と複数の多重定義とを関連付けるので、あいまいになります。そのため、コンパイラーは、最も特殊化された定義を選択します。関数テンプレート定義を選択するこの処理は、*部分選択* と呼ばれます。

`X` からの特殊化と一致する引数リストは、いずれも `Y` からの特殊化と一致するが、その逆では一致しないという場合は、テンプレート `X` は、テンプレート `Y` よりもさらに特殊化されています。次の例は、部分選択を示しています。

```
template<class T> void f(T)  { }
template<class T> void f(T*) { }
template<class T> void f(const T*) { }

template<class T> void g(T) { }
template<class T> void g(T&) { }

template<class T> void h(T) { }
template<class T> void h(T, ...) { }

int main() {
   const int *p;
   f(p);

   int q;
//  g(q);
//  h(q);
}
```

宣言 `template<class T> void f(const T*)` は、 `template<class T> void f(T*)` よりもさらに特殊化されています。したがって、関数呼び出し `f(p)` は、`template<class T> void f(const T*)` を呼び出します。しかし、`void g(T)` または `void g(T&)` のいずれも、他のものより特殊化されていません。したがって、関数呼び出し `g(q)` は、あいまいです。

省略符号は、部分選択に影響を与えません。したがって、関数呼び出し `h(q)` もあいまいです。

コンパイラーは、次の場合に、部分選択を使用します。

- 多重定義解決を必要とする関数テンプレート特殊化の呼び出し
- 関数テンプレート特殊化のアドレスの取得
- フレンド関数宣言、明示的インスタンス生成、または明示的特殊化が、関数テンプレート特殊化を参照するとき
- 任意の割り振り解除 new の関数テンプレートでもある適切な配置解除関数の決定

関連資料:

468 ページの『テンプレートの特殊化』

218 ページの『new 式 (C++ のみ)』

テンプレートのインスタンス生成

関数、クラス、またはクラスのメンバーの新規定義を、テンプレート宣言から、または 1 つ以上のテンプレート引数から作成する処理は、*テンプレートのインスタンス生成* と呼ばれます。テンプレートの引数の特定のセットを処理するためにテンプレートのインスタンス生成から作成された定義は、*特殊化* と呼ばれます。

テンプレートのインスタンス生成には、明示的インスタンス生成および暗黙のインスタンス生成という 2 つのフォームがあります。

関連資料:

468 ページの『テンプレートの特殊化』

明示的インスタンス生成

コンパイラーに、テンプレートから定義をいつ生成するのかを明示的に指示できます。これは、*明示的インスタンス生成* と呼ばれます。明示的インスタンス生成には、*明示的インスタンス生成宣言* および*明示的インスタンス生成定義* という 2 つのフォームが含まれます。

C++11

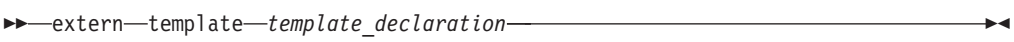
注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11

機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

明示的インスタンス生成宣言

C++11 標準では、明示的インスタンス生成宣言機能が導入されています。この機能を使用すると、テンプレート特殊化またはそのメンバーの暗黙のインスタンス生成を抑制できます。明示的インスタンス生成宣言を示すには、extern キーワードを使用します。ここでの extern の使用法は、ストレージ・クラス指定子の場合とは異なります。

明示的インスタンス生成宣言の構文



テンプレートの明示的インスタンス生成定義が、他の変換単位で存在するか、または後から同じファイル内に存在する場合は、テンプレート特殊化に対して明示的インスタンス生成宣言を指定できます。1 つの変換単位に明示的インスタンス生成定義が含まれる場合、他の変換単位は特殊化で複数回インスタンス生成を行うことなく、特殊化を使用できます。次の例は、このコンセプトを示しています。

```
//sample1.h:
template<typename T, T val>
union A{
   T func();
};
extern template union A<int, 55>;

template<class T, T val>
T A<T,val>::func(void){
   return val;
}

//sampleA.C"
#include "sample1.h"

template union A<int,55>;

//sampleB.C:
#include "sample1.h"

int main(void){
   return A<int, 55>().func();
}
```

sampleB.C は、sampleA.C の A<int, 55>().func() の明示的インスタンス生成定義を使用します。

関数またはクラスの明示的インスタンス生成宣言が宣言されているが、プログラム内に対応する明示的インスタンス生成定義が存在しない場合、コンパイラーはエラー・メッセージを出します。以下の例を参照してください。

```
// sample2.C
template <typename T, T val>
struct A{
   virtual T func();
   virtual T bar();
}
```

```
extern template int A<int,55>::func();

template <class T, T val>
T A<T,val>::func(void){
   return val;
}

template <class T, T val>
T A<T,val>::bar(void){
   return val;
}

int main(void){
   return A<int,55>().bar();
}
```

明示的インスタンス生成宣言を使用する場合、以下の制約事項に注意してください。

- 明示的インスタンス生成宣言では静的クラス・メンバーに命名できますが、静的関数には命名できません。これは、静的関数は他の変換単位では名前でアクセスできないためです。
- クラスの明示的インスタンス生成宣言は、その各メンバーの明示的インスタンス生成宣言と等価ではありません。

C++11

明示的インスタンス生成定義

明示的インスタンス生成定義は、テンプレート特殊化またはそのメンバーのインスタンス生成です。

明示的インスタンス生成定義の構文

以下に、明示的インスタンス生成定義の例を示します。

```
template<class T> class Array { void mf(); };
template class Array<char>;       /* explicit instantiation definition */
template void Array<int>::mf();   /* explicit instantiation definition */

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); /* explicit instantiation definition */

namespace N {
   template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation definition is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);           /* explicit instantiation definition */

template <class T> class X {
   private:
      T v(T arg) { return arg; };
```

```
};

template int X<int>::v(int);    /* explicit instantiation definition */

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

テンプレートの明示的インスタンス生成定義は、テンプレートを定義した場所と同じ名前空間にあります。

アクセス検査規則は、明示的インスタンス生成定義の引数には適用されません。明示的インスタンス生成定義のテンプレート引数には、private 型または private オブジェクトを使用できます。この例では、メンバー関数が private で宣言されている場合でも、明示的インスタンス生成定義 `template int X<int>::v(int)` を使用できます。

テンプレートのインスタンスを明示的に生成する場合、コンパイラーは、デフォルトの引数を使用しません。この例では、デフォルトの引数が型 `int` のアドレスである場合でも、明示的インスタンス生成定義 `template char g(char)` を使用できます。

注: `inline` または C++11 `constexpr` C++11 指定子を、関数テンプレートまたはクラス・テンプレートのメンバー関数の明示的インスタンス生成で使用することはできません。

C++11

明示的インスタンス生成定義およびインライン名前空間定義

インライン名前空間定義は、最初に `inline` キーワードを使用する名前空間定義です。インライン名前空間のメンバーは、エンクロージング名前空間のメンバーでもあるかのように、インスタンス生成または特殊化を明示的に実行できます。詳しくは、320 ページの『インライン名前空間定義 (C++11)』を参照してください。

C++11

関連資料:

555 ページの『C++11 互換性の拡張機能』

暗黙のインスタンス生成

テンプレート特殊化が明示的にインスタンス生成されない限り、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生成* と呼ばれます。

C++11

コンパイラーは、明示的インスタンス生成宣言が存在する場合、非クラス・エンティティー、非インライン・エンティティーの特殊化を生成する必要がありません。

C++11

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();
1};

X<int>* q;
X<int> r;
X<float>* s;
r.f();
s->g();
```

コンパイラーは、次のクラスおよび関数のインスタンス生成を必要とします。

- オブジェクト `r` が宣言されたときの `X<int>`
- メンバー関数呼び出し `r.f()` での `X<int>::f()`
- クラス・メンバー・アクセス関数呼び出し `s->g()` での `X<float>` および `X<float>::g()`

したがって、上記の例をコンパイルするには、関数 `X<T>::f()` および `X<T>::g()` を定義する必要があります。(コンパイラーは、オブジェクト `r` を作成する場合、クラス `X` のデフォルトのコンストラクターを使用します。) コンパイラーは、次の定義のインスタンス生成を必要としません。

- ポインター `p` が宣言されたときのクラス `X`
- ポインター `q` が宣言されたときの `X<int>`
- ポインター `s` が宣言されたときの `X<float>`

コンパイラーが、ポインター型変換、またはメンバー型変換を指すポインターに関係する場合、それはクラス・テンプレート特殊化のインスタンスを暗黙的に生成します。次の例は、このことを示しています。

```
template<class T> class B { };
template<class T> class D : public B<T> { };

void g(D<double>* p, D<int>* q)
{
  B<double>* r = p;
  delete q;
}
```

割り当て `B<double>* r = p` は、型 `D<double>*` の `p` を `B<double>*` の型に変換します。コンパイラーは、`D<double>` のインスタンスを生成する必要があります。コンパイラーは、`q` の削除を試みる際に、 `D<int>` のインスタンスを生成しなければなりません。

コンパイラーが、静的メンバーを含むクラス・テンプレートのインスタンスを暗黙的に生成する場合、それらの静的メンバーのインスタンスは、暗黙的には生成されません。コンパイラーは、静的メンバーの定義を必要とする場合のみ、静的メンバ

ーのインスタンスを生成します。インスタンスを生成されたクラス・テンプレートは、いずれも静的メンバーのそれ自身のコピーを所有しています。次の例は、このことを示しています。

```
template<class T> class X {
public:
   static T v;
};

template<class T> T X<T>::v = 0;

X<char*> a;
X<float> b;
X<float> c;
```

オブジェクト a には、型 char* の静的メンバー変数 v があります。オブジェクト b には、型 float の静的変数 v があります。オブジェクト b と c は、単一静的データ・メンバー v を共有します。

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名前空間にあります。

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。

テンプレートの特殊化

関数、クラス、またはクラスのメンバーの新規定義を、テンプレート宣言から、または 1 つ以上のテンプレート引数から作成する処理は、*テンプレートのインスタンス生成* と呼ばれます。テンプレートのインスタンス生成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* は、特殊化されているテンプレートです。

関連資料:

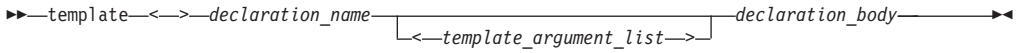
463 ページの『テンプレートのインスタンス生成』

明示的特殊化

テンプレート引数の任意のセットでテンプレートのインスタンスを生成する場合、コンパイラーは、それらのテンプレート引数に基づいて新規の定義を生成します。この定義生成の動作をオーバーライドできます。その代わりに、コンパイラーがテンプレート引数の任意のセットで使用する定義を、指定することができます。これは、*明示的特殊化* と呼ばれます。次のテンプレートは、いずれも明示的特殊化が可能です。

- 関数テンプレート
- クラス・テンプレート
- クラス・テンプレートのメンバー関数
- クラス・テンプレートの静的データ・メンバー
- クラス・テンプレートのメンバー・クラス
- クラス・テンプレートのメンバー関数テンプレート
- クラス・テンプレートのメンバー・クラス・テンプレート

明示的特殊化宣言の構文



template<> 接頭部は、次のテンプレート宣言が、テンプレート・パラメーターを取得しないということを示しています。 *declaration_name* は、以前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;
}
```

上記の例の出力は、次のとおりです。

```
Primary template non-type argument is: 6
Explicit specialization default arguments
Explicit specialization <double, 10>
```

この例では、主テンプレート (特殊化され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

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)。以下の例を参照してください。

```
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、不完全に定義されたクラスと同じようには定義されていない、明示的特殊化の名前を使用できます。次の例は、このことを示しています。

```
template<class T> class X { };
template<>  class X<char>;
X<char>* p;
X<int> i;
// X<char> j;
```

コンパイラーは、宣言 `X<char>` の明示的特殊化が定義されていないので、`X<char> j` を許可しません。

明示的特殊化とスコープ

主テンプレートの宣言は、明示的特殊化を*宣言するポイント* のあるスコープ内に存在する必要があります。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コードを許可しません。

```
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名前空間に存在します。

明示的特殊化のクラス・メンバー

明示的特殊化クラスのメンバーは、主テンプレートのメンバー宣言から暗黙的にはインスタンス作成されません。クラス・テンプレート特殊化のメンバーを明示的に定義する必要があります。明示的特殊化テンプレート・クラスのメンバーを、`template<>` 接頭部を使用せずに、標準クラスのメンバーを定義するのと同じように定義します。さらに、明示的特殊化のメンバーをインラインに定義できます。ここでは、特別なテンプレート構文は使用されていません。次の例は、クラス・テンプレート特殊化を示しています。

```
template<class T> class A {
   public:
      void f(T);
};

template<> class A<int> {
   public:
      int g(int);
};

int A<int>::g(int arg) { return 0; }
```

```
int main() {
   A<int> a;
   a.g(1234);
}
```

明示的特殊化 A<int> は、メンバー関数 g() を含んでいますが、主テンプレートは、これを含んでいません。

テンプレート、メンバー・テンプレート、またはクラス・テンプレートのメンバーを明示的に特殊化する場合、この特殊化を宣言してから、特殊化のインスタンスを暗黙的に生成する必要があります。例えば、コンパイラーは次のコードを許可しません。

```
template<class T> class A { };

void f() { A<int> x; }
template<> class A<int> { };

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

コンパイラーは、関数 f() が、特殊化の前に、この特殊化 (x の構造体にある) を使用するので、明示的特殊化 template<> class A<int> { }; を許可しません。

関数テンプレートの明示的特殊化

関数テンプレート特殊化では、コンパイラーが関数引数の型からテンプレート引数を推定できる場合、テンプレート引数はオプションです。次の例は、このことを示しています。

```
template<class T> class X { };
template<class T> void f(X<T>);
template<> void f(X<int>);
```

明示的特殊化 template<> void f(X<int>) は、 template<> void f<int>(X<int>) と等価です。

次の場合の宣言または定義では、デフォルトの関数引数を指定することはできません。

- 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

インスタンスが生成された各クラス・テンプレート特殊化は、静的メンバーのそれ自身のコピーを所有しています。静的メンバーを明示的に特殊化できます。次の例は、このことを示しています。

```
template<class T> class X {
public:
   static T v;
   static void f(T);
};

template<class T> T X<T>::v = 0;
template<class T> void X<T>::f(T arg) { v = arg; }

template<> char* X<char*>::v = "Hello";
template<> void X<float>::f(float arg) { v = arg * 2; }

int main() {
   X<char*> a, b;
   X<float> c;
   c.f(10);
}
```

このコードは、テンプレート引数 char* がストリング "Hello" を指すように、静的データ・メンバー X::v の初期化を明示的に特殊化します。関数 X::f() は、テンプレート引数 float に対して明示的に特殊化されます。オブジェクト a および b の静的データ・メンバー v は、同じストリング、つまり "Hello" を指します。c.v の値は、関数呼び出し c.f(10) の後で 20 になります。

メンバー・テンプレートを、複数のエンクロージング・クラス・テンプレート内でネストできます。いくつかのエンクロージング・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エンクロージング・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エンクロージング・クラス・テンプレートも残すことはできますが、そのエンクロージング・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');
}
```

上記のプログラムの出力は、次のとおりです。

```
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 のエンクロージング・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。

C++11

明示的特殊化定義およびインライン名前空間定義

インライン名前空間定義は、最初に inline キーワードを使用する名前空間定義です。インライン名前空間のメンバーは、エンクロージング名前空間のメンバーでもあるかのように、インスタンス生成または特殊化を明示的に実行できます。詳しくは、320 ページの『インライン名前空間定義 (C++11)』を参照してください。

C++11

関連資料:

453 ページの『関数テンプレート』

447 ページの『クラス・テンプレート』

451 ページの『クラス・テンプレートのメンバー関数』

451 ページの『静的データ・メンバーとテンプレート』

267 ページの『削除済み関数』

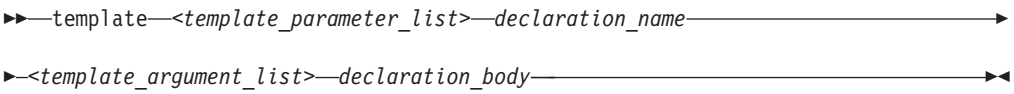
部分的特殊化

クラス・テンプレートのインスタンスを生成する場合、コンパイラーは、受け渡したテンプレート引数に基づいて定義を作成します。代替として、それらすべてのテンプレート引数が、明示的特殊化のものと一致する場合、コンパイラーは、明示的特殊化が定義した定義を使用します。

部分的特殊化 は、明示的特殊化に汎用性を持たせたものです。明示的特殊化は、テンプレート引数リストだけを持っています。部分的特殊化は、テンプレート引数リストとテンプレート・パラメーター・リストの両方を持っています。コンパイラーは、テンプレート引数リストが、テンプレートのインスタンス生成のテンプレート引数のサブセットと一致する場合、部分的特殊化を使用します。そして、コンパイラーは、テンプレートのインスタンス生成の一致しない残りのテンプレート引数を使用して、部分的特殊化から新規定義を生成します。

関数テンプレートは、部分的に特殊化することはできません。

部分的特殊化の構文



declaration_name は、以前宣言されたテンプレートの名前です。 *declaration_body* はオプションなので、部分的特殊化を前もって宣言できることに注意してください。

以下は、部分的特殊化の使用法を示したものです。

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

template<class T, class U, int I> struct X
  { void f() { cout << "Primary template" << endl; } };

template<class T, int I> struct X<T, T*, I>
  { void f() { cout << "Partial specialization 1" << endl;
  } };

template<class T, class U, int I> struct X<T*, U, I>
  { void f() { cout << "Partial specialization 2" << endl;
  } };

template<class T> struct X<int, T*, 10>
  { void f() { cout << "Partial specialization 3" << endl;
  } };

template<class T, class U, int I> struct X<T, U*, I>
  { void f() { cout << "Partial specialization 4" << endl;
  } };

int main() {
   X<int, int, 10> a;
   X<int, int*, 5> b;
   X<int*, float, 10> c;
   X<int, char*, 10> d;
   X<float, int*, 10> e;
//   X<int, int*, 10> f;
   a.f(); b.f(); c.f(); d.f(); e.f();
}
```

上記の例の出力は、以下のとおりです。

```
Primary template
Partial specialization 1
Partial specialization 2
Partial specialization 3
Partial specialization 4
```

コンパイラーは、宣言 X<int, int*, 10> f が、 template struct X<T, T*, I>、template struct X<int, T*, 10>、または template struct X<T, U*, I> と一致し、どれも他よりうまく一致する訳ではないので、この宣言を許可しません。

各クラス・テンプレートの部分的特殊化は、別々のテンプレートです。クラス・テンプレートの部分的特殊化のメンバーごとに、定義が必要です。

部分的特殊化のテンプレート・パラメーターと引数リスト

主テンプレートは、テンプレート引数リストを持っていません。このリストは、テンプレート・パラメーター・リストに暗黙で存在します。

テンプレート・パラメーターを主テンプレートでは指定しているが、部分的特殊化で使用していなければ、それを部分的特殊化のテンプレート・パラメーター・リストから省略できます。部分的特殊化の引数リストの順序は、主テンプレートの暗黙の引数リストの順序と同じです。

部分的テンプレート・パラメーターのテンプレート引数リストでは、式が ID のみとなる場合を除いて、非型引数を含む式を持つことはできません。次の例では、コンパイラーは、最初の部分的特殊化を許可しませんが、2 番目のものは許可します。

```
template<int I, int J> class X { };

// Invalid partial specialization
template<int I> class X <I * 4, I + 3> { };

// Valid partial specialization
template <int I> class X <I, I> { };
```

「非型」テンプレート引数の型は、部分的特殊化のテンプレート・パラメーターに依存できません。コンパイラーは、次の部分的特殊化を許可しません。

```
template<class T, T i> class X { };

// Invalid partial specialization
template<class T> class X<T, 25> { };
```

部分的特殊化のテンプレート引数リストは、主テンプレートによる暗黙のリストと同じにすることはできません。

部分的特殊化のテンプレート・パラメーター・リストに、デフォルト値を持つことができません。

クラス・テンプレート部分的特殊化のマッチング

コンパイラーは、クラス・テンプレート特殊化のテンプレート引数と、主テンプレートおよび部分的特殊化のテンプレート引数リストを突き合わせて、主テンプレートを使用するのか、その部分的特殊化の 1 つを使用するのかを決定します。

- コンパイラーが特殊化を 1 つだけ検出する場合、コンパイラーは、その特殊化から定義を生成します。
- コンパイラーが複数の特殊化を検出する場合、コンパイラーは、どの特殊化が最も特殊化されているのかを判別します。 X からの特殊化と一致する引数リストは、いずれも Y からの特殊化と一致するが、その逆では一致しないという場合は、テンプレート X は、テンプレート Y よりもさらに特殊化されています。コ

ンパイラーが最も特殊化された特殊化を検出できない場合は、クラス・テンプレートの使用は、あいまいになります。つまり、コンパイラーは、プログラムを許可しません。

- コンパイラーがどのような一致も検出しない場合、コンパイラーは、主テンプレートから定義を生成します。

C++11

部分的特殊化は、可変数引数テンプレート機能にも適用されます。詳しくは、『可変数引数テンプレート (C++11)』を参照してください。

関連資料:

440 ページの『テンプレート・パラメーター』

443 ページの『テンプレート引数』

可変数引数テンプレート (C++11)

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 の前は、テンプレートは、テンプレートの宣言で指定する必要のある固定数のパラメーターを持っていました。テンプレートは、可変数のパラメーターを持つクラスまたは関数テンプレートを直接表すことができませんでした。既存の C++ プログラムでは、この問題を部分的に改善するために、異なる数のパラメーターまたは特別なデフォルト・テンプレート・パラメーターを持つ多重定義関数テンプレートを使用できました。

可変数引数テンプレート機能を使用すると、ゼロ個を含む、あらゆる数のパラメーターを持つクラスまたは関数テンプレートを定義できます。これを実現するために、この機能には、*パラメーター・パック* と呼ばれるパラメーターが導入されており、テンプレートのゼロ個以上のパラメーターのリストを表すことができます。

可変数引数テンプレート機能には、パラメーター・パックが展開されていることを示す*パック拡張* も導入されています。

2 つの技法、*テンプレート引数の推定* および*部分的特殊化* も、そのパラメーター・リストにパラメーター・パックを持つテンプレートに適用できます。

パラメーター・パック

パラメーター・パック は、テンプレートのパラメーターの型に設定できます。単一の引数にのみバインドできる、これまでのパラメーターとは異なり、パラメーター・パックは、省略符号をパラメーター名の左側に配置することで、複数のパラメーターを単一のパラメーターにパックできます。

テンプレート定義では、パラメーター・パックは単一のパラメーターとして取り扱われます。テンプレートのインスタンス生成では、パラメーター・パックは展開され、正しい数のパラメーターが作成されます。

パラメーター・パックが使用されるコンテキストに応じて、パラメーター・パックをテンプレート・パラメーター・パック または*関数仮*パラメーター・パック のいずれかに設定できます。

テンプレート・パラメーター・パック

テンプレート・パラメーター・パック は、ゼロ個を含む、あらゆる数のテンプレート・パラメーターを表すテンプレート・パラメーターです。構文上は、テンプレート・パラメーター・パックは、省略符号を使用して指定するテンプレート・パラメーターです。次の例を検討してみます。

```
template<class...A> struct container{};
template<class...B> void func();
```

この例では、A および B は、テンプレート・パラメーター・パックです。

テンプレート・パラメーター・パックに含まれるパラメーターの型に応じて、以下の 3 種類のテンプレート・パラメーター・パックがあります。

- 「型」パラメーター・パック
- 「非型」パラメーター・パック
- 「テンプレート」テンプレート・パラメーター・パック

「型」パラメーター・パックは、ゼロ個以上の「型」テンプレート・パラメーターを表します。同様に、「非型」パラメーター・パックは、ゼロ個以上の「非型」テンプレート・パラメーターを表します。

注: 「テンプレート」テンプレート・パラメーター・パックは、XL C/C++ V13.1 ではサポートされていません。

以下の例は、「型」パラメーター・パックを示しています。

```
template<class...T> class X{};

X<> a;                    // the parameter list is empty
X<int> b;                 // the parameter list has one item
X<int, char, float> c;    // the parameter list has three items
```

この例では、「型」パラメーター・パック T は、ゼロ個以上の「型」テンプレート・パラメーターのリストに展開されます。

以下の例は、「非型」パラメーター・パックを示しています。

```
template<bool...A> class X{};

X<> a;                    // the parameter list is empty
X<true> b;                // the parameter list has one item
X<true, false, true> c;   // the parameter list has three items
```

この例では、「非型」パラメーター・パック A は、ゼロ個以上の「非型」テンプレート・パラメーターのリストに展開されます。

テンプレート引数を推定できる場合、例えば、関数テンプレートおよびクラス・テンプレートの部分的特殊化の場合は、テンプレート・パラメーター・パックをテン

プレートの最後のテンプレート・パラメーターにする必要はありません。この場合、テンプレート・パラメーター・リストで複数のテンプレート・パラメーター・パックを宣言できます。ただし、テンプレート引数を推定できない場合、テンプレート・パラメーター・リストで宣言できるテンプレート・パラメーター・パックは最大で 1 つであり、そのテンプレート・パラメーターパックを最後のテンプレート・パラメーターにする必要があります。次の例を検討してみます。

```
// error
template<class...A, class...B>struct container1{};

// error
template<class...A,class B>struct container2{};
```

この例では、コンパイラーは 2 つのエラー・メッセージを出します。 1 つ目のエラー・メッセージは、クラス・テンプレート container1 に対してのエラーで、container1 に、A および B という推定できない 2 つテンプレート・パラメーター・パックが含まれていることによるものです。もう 1 つのエラー・メッセージは、クラス・テンプレート container2 に対してのエラーで、テンプレート・パラメーター・パック A が container2 の最後のテンプレート・パラメーターではなく、A を推定できないことによるものです。

テンプレート・パラメーター・パックでは、デフォルトの引数は使用できません。次の例を検討してみます。

```
template<typename...T=int> struct foo1{};
```

この例では、コンパイラーはエラー・メッセージを出します。これは、テンプレート・パラメーター・パック T に、デフォルトの引数 int が指定されたためです。

関数仮パラメーター・パック

> *関数仮パラメーター・パック* は、ゼロ個以上の関数仮パラメーターを表す関数仮パラメーターです。構文上は、関数仮パラメーター・パックは、省略符号を使用して指定する関数仮パラメーターです。

関数テンプレートの定義では、関数仮パラメーター・パックは、関数仮パラメーターのテンプレート・パラメーター・パックを使用します。テンプレート・パラメーター・パックは、関数仮パラメーター・パックによって展開されます。次の例を検討してみます。

```
template<class...A> void func(A...args)
```

この例では、A はテンプレート・パラメーター・パックであり、args は関数仮パラメーター・パックです。以下のように、ゼロ個を含む、あらゆる数の引数を使用して、関数を呼び出すことができます。

```
func();                 // void func();
func(1);                // void func(int);
func(1,2,3,4,5);        // void func(int,int,int,int,int);
func(1,'x', aWidget);   // void func(int,char,widget);
```

関数仮パラメーター・パックは、関数テンプレートの最後の関数仮パラメーターである場合、*後置関数仮パラメーター・パック* になります。そうでない場合、*非後置関数仮パラメーター・パック* となります。関数テンプレートは、後置関数仮パラメーター・パックと非後置関数仮パラメーター・パックを保有できます。非後置関数仮パラメーター・パックは、関数テンプレートが呼び出されたときに、明示的に指

定された引数によってのみ推定できます。関数テンプレートが明示的な引数なしで呼び出された場合、非後置関数仮パラメーター・パックは、以下の例に示すように、空になります。

```
#include <cassert>

template<class...A, class...B> void func(A...arg1,int sz1, int sz2, B...arg2)
{
   assert( sizeof...(arg1) == sz1);
   assert( sizeof...(arg2) == sz2);
}

int main(void)
{
   //A:(int, int, int), B:(int, int, int, int, int)
   func<int,int,int>(1,2,3,3,5,1,2,3,4,5);

   //A: empty, B:(int, int, int, int, int)
   func(0,5,1,2,3,4,5);
   return 0;
}
```

この例では、関数テンプレート `func` には、`arg1` および `arg2` の 2 つの関数仮パラメーター・パックがあります。`arg1` は、非後置関数仮パラメーター・パックで、`arg2` は、後置関数仮パラメーター・パックです。`func` が、`func<int,int,int>(1,2,3,3,5,1,2,3,4,5)` という形式で、3 つの明示的に指定された引数を指定して呼び出された場合、`arg1` および `arg2` は両方とも正常に推定されます。 `func` が `func(0,5,1,2,3,4,5)` という形式で、明示的に指定された引数なしで呼び出された場合、`arg2` は正常に推定されますが、`arg1` は空になります。この例では、関数テンプレート `func` のテンプレート・パラメーター・パックは推定可能であるため、`func` には、複数のテンプレート・パラメーター・パックを保有することができます。

パック拡張

パック拡張 は、後に省略符号が続く 1 つ以上のパラメーター・パックを含む式であり、パラメーター・パックが展開されていることを示します。次の例を検討してみます。

```
template<class...T> void func(T...a){};
template<class...U> void func1(U...b){
    func(b...);
}
```

この例では、`T...` および `U...` は、テンプレート・パラメーター・パック `T` および `U` の対応するパック拡張であり、`b...` は関数仮パラメーター・パック `b` のパック拡張です。

パック拡張は、以下のコンテキストで使用できます。

- 式リスト
- 初期化指定子リスト
- 基本指定子リスト
- メンバー初期化指定子リスト
- テンプレート引数リスト
- 例外指定リスト

式リスト

例:

```
#include <cstdio>
#include <cassert>

template<class...A> void func1(A...arg){
    assert(false);
}

void func1(int a1, int a2, int a3, int a4, int a5, int a6){
    printf("call with(%d,%d,%d,%d,%d,%d)¥n",a1,a2,a3,a4,a5,a6);
}

template<class...A> int func(A...args){
    int size = sizeof...(A);
    switch(size){
        case 0: func1(99,99,99,99,99,99);
        break;
        case 1: func1(99,99,args...,99,99,99);
        break;
        case 2: func1(99,99,args...,99,99);
        break;
        case 3: func1(args...,99,99,99);
        break;
        case 4: func1(99,args...,99);
        break;
        case 5: func1(99,args...);
        break;
        case 6: func1(args...);
        break;
        default:
        func1(0,0,0,0,0,0);
    }
    return size;
}

int main(void){
    func();
    func(1);
    func(1,2);
    func(1,2,3);
    func(1,2,3,4);
    func(1,2,3,4,5);
    func(1,2,3,4,5,6);
    func(1,2,3,4,5,6,7);
    return 0;
}
```

この例の出力は次のようになります。

```
call with (99,99,99,99,99,99)
call with (99,99,1,99,99,99)
call with (99,99,1,2,99,99)
call with (1,2,3,99,99,99)
call with (99,1,2,3,4,99)
call with (99,1,2,3,4,5)
call with (1,2,3,4,5,6)
call with (0,0,0,0,0,0)
```

この例では、switch ステートメントは、異なる位置のパック拡張 `args...` を関数 `func1` の式リスト内に示します。出力は、関数 `func1` の各呼び出しを示し、その展開を示します。

初期化指定子リスト

例:

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

void printarray(int arg[], int length){
    for(int n=0; n<length; n++){
        printf("%d ",arg[n]);
    }
    printf("¥n");
}

template<class...A> void func(A...args){
    const int size = sizeof...(args) +5;
    printf("size %d¥n", size);
    int res[sizeof...(args)+5]={99,98,args...,97,96,95};
    printarray(res,size);
}

int main(void)
{
    func();
    func(1);
    func(1,2);
    func(1,2,3);
    func(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20);
    return 0;
}
```

この例の出力は次のようになります。

```
size 5
99 98 97 96 95
size 6
99 98 1 97 96 95
size 7
99 98 1 2 97 96 95
size 8
99 98 1 2 3 97 96 95
size 25
99 98 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 97 96 95
```

この例では、パック拡張 `args...` は配列 `res` の初期化指定子リスト内にあります。

基本指定子リスト

例:

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

struct a1{};
struct a2{};
struct a3{};
struct a4{};

template<class X> struct baseC{
    baseC() {printf("baseC primary ctor¥n");}
};
template<> struct baseC<a1>{
    baseC() {printf("baseC a1 ctor¥n");}
};
template<> struct baseC<a2>{
```

```
    baseC() {printf("baseC a2 ctor¥n");}
};
template<> struct baseC<a3>{
    baseC() {printf("baseC a3 ctor¥n");}
};
template<> struct baseC<a4>{
    baseC() {printf("baseC a4 ctor¥n");}
};

template<class...A> struct container : public baseC<A>...{
    container(){
        printf("container ctor¥n");
    }
};

int main(void){
    container<a1,a2,a3,a4> test;
    return 0;
}
```

この例の出力は次のようになります。

```
baseC a1 ctor
baseC a2 ctor
baseC a3 ctor
baseC a4 ctor
container ctor
```

この例では、パック拡張 baseC<A>... はクラス・テンプレート container の基本指定子リスト内にあります。パック拡張は、4 つの基底クラス baseC<a1>、baseC<a2>、baseC<a3>、および baseC<a4> に展開されます。出力は、4 つの基底クラス・テンプレートがすべて、クラス・テンプレート container のインスタンス生成の前に初期化されることを示します。

メンバー初期化指定子リスト

例:

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

struct a1{};
struct a2{};
struct a3{};
struct a4{};

template<class X> struct baseC{
    baseC(int a) {printf("baseC primary ctor: %d¥n", a);}
};
template<> struct baseC<a1>{
    baseC(int a) {printf("baseC a1 ctor: %d¥n", a);}
};
template<> struct baseC<a2>{
    baseC(int a) {printf("baseC a2 ctor: %d¥n", a);}
};
template<> struct baseC<a3>{
    baseC(int a) {printf("baseC a3 ctor: %d¥n", a);}
};
template<> struct baseC<a4>{
    baseC(int a) {printf("baseC a4 ctor: %d¥n", a);}
};

template<class...A> struct container : public baseC<A>...{
    container(): baseC<A>(12)...{
        printf("container ctor¥n");
```

```
    }
};

int main(void){
    container<a1,a2,a3,a4> test;
    return 0;
}
```

この例の出力は次のようになります。

```
baseC a1 ctor:12
baseC a2 ctor:12
baseC a3 ctor:12
baseC a4 ctor:12
container ctor
```

この例では、パック拡張 baseC<A>(12)... はクラス・テンプレート container のメンバー初期化指定子リスト内にあります。コンストラクター初期化指定子リストは展開されて、各基底クラス baseC<a1>(12)、baseC<a2>(12)、baseC<a3>(12)、および baseC<a4>(12) の呼び出しを含みます。

テンプレート引数リスト

例:

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

template<int val> struct value{
    operator int(){return val;}
};

template <typename...I> struct container{
    container(){
        int array[sizeof...(I)]={I()...};
        printf("container<");
        for(int count = 0; count<sizeof...(I); count++){
            if(count>0){
                printf(",");
            }
            printf("%d", array[count]);
        }
        printf(">¥n");
    }
};

template<class A, class B, class...C> void func(A arg1, B arg2, C...arg3){
    container<A,B,C...> t1;  // container<99,98,3,4,5,6>
    container<C...,A,B> t2;  // container<3,4,5,6,99,98>
    container<A,C...,B> t3;  // container<99,3,4,5,6,98>
}

int main(void){
    value<99> v99;
    value<98> v98;
    value<3> v3;
    value<4> v4;
    value<5> v5;
    value<6> v6;
    func(v99,v98,v3,v4,v5,v6);
    return 0;
}
```

この例の出力は次のようになります。

```
container<99,98,3,4,5,6>
container<3,4,5,6,99,98>
container<99,3,4,5,6,98>
```

この例では、パック拡張 C... はクラス・テンプレート container のテンプレート引数リストのコンテキストで展開されます。

例外指定リスト

例:

```
struct a1{};
struct a2{};
struct a3{};
struct a4{};
struct a5{};
struct stuff{};

template<class...X> void func(int arg) throw(X...){
    a1 t1;
    a2 t2;
    a3 t3;
    a4 t4;
    a5 t5;
    stuff st;

    switch(arg){
        case 1:
            throw t1;
            break;
        case 2:
            throw t2;
            break;
        case 3:
            throw t3;
            break;
        case 4:
            throw t4;
            break;
        case 5:
            throw t5;
            break;
        default:
            throw st;
            break;
    }
}

int main(void){
    try{
        // if the throw specification is correctly expanded, none of
        // these calls should trigger an exception that is not expected
        func<a1,a2,a3,a4,a5,stuff>(1);
        func<a1,a2,a3,a4,a5,stuff>(2);
        func<a1,a2,a3,a4,a5,stuff>(3);
        func<a1,a2,a3,a4,a5,stuff>(4);
        func<a1,a2,a3,a4,a5,stuff>(5);
        func<a1,a2,a3,a4,a5,stuff>(99);
    }
    catch(...){
        return 0;
    }
    return 1;
}
```

この例では、パック拡張 X... は関数テンプレート func の例外指定リストのコンテキストで展開されます。

パラメーター・パックを宣言する場合、それをパック拡張によって展開する必要があります。展開されていないパラメーター・パックの名前を指定することは誤りです。次の例を検討してみます。

```
template<class...A> struct container;
template<class...B> struct container<B>{}
```

この例では、コンパイラーはエラー・メッセージを出します。これは、テンプレート・パラメーター・パック B が展開されていないためです。

パック拡張は、パラメーター・パックでないパラメーターには一致できません。次の例を検討してみます。

```
template<class X> struct container{};

template<class A, class...B>
// Error, parameter A is not a parameter pack
void func1(container<A>...args){};

template<class A, class...B>
// Error, 1 is not a parameter pack
void func2(1...){};
```

複数のパラメーター・パックがパック拡張で参照される場合、各拡張では、これらのパラメーター・パックから同じ数の引数が展開される必要があります。次の例を検討してみます。

```
struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};

template<class...X> struct baseC{};
template<class...A1> struct container{};
template<class...A, class...B, class...C>
struct container<baseC<A,B,C...>...>:public baseC<A,B...,C>{};

int main(void){
    container<baseC<a1,a4,a5,a5,a5>, baseC<a2,a3,a5,a5,a5>,
              baseC<a3,a2,a5,a5,a5>,baseC<a4,a1,a5,a5,a5> > test;
    return 0;
}
```

この例では、テンプレート・パラメーター・パック A、B、および C は、同じパック拡張 baseC<A,B,C...>... で参照されます。コンパイラーはエラー・メッセージを出し、クラス・テンプレート container のテンプレートのインスタンス生成中に、これらのテンプレート・パラメーター・パックを展開したときに、それらの長さが一致しなかったことを示します。

部分的特殊化

部分的特殊化 は、可変数引数テンプレート機能の基本部分です。基本的な部分的特殊化を使用すると、パラメーター・パックの個々の引数にアクセスできます。以下の例は、可変数引数テンプレートの部分的特殊化の使用法を示しています。

```
// primary template
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<B,C...>{};
```

引数リストを使用してクラス・テンプレート container のインスタンスを生成する場合、部分的特殊化は、1 つ以上の引数が存在するすべての場合に一致します。その場合、テンプレート・パラメーター B は最初のパラメーターを保持し、パック拡張 C... は引数リストの残りを含みます。リストが空の場合、部分的特殊化は一致しないため、インスタンス生成は主テンプレートと一致します。

パック拡張は、部分的特殊化の引数リストの最後の引数にする必要があります。次の例を検討してみます。

```
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<C...,B>{};
```

この例では、コンパイラーはエラー・メッセージを出します。これは、パック拡張 C... が部分的特殊化の引数リストの最後の引数でないためです。

部分的特殊化は、パラメーター・リストに複数のテンプレート・パラメーター・パックを持つことができます。次の例を検討してみます。

```
template<typename T1, typename T2> struct foo{};
template<typename...T> struct bar{};

// partial specialization
template<typename...T1,typename...T2> struct bar<foo<T1,T2>...>{};
```

この例では、部分的特殊化は、パラメーター・リストに 2 つのテンプレート・パラメーター・パック T1 および T2 を持ちます。

パラメーター・パックの引数にアクセスするために、部分的特殊化を使用して、最初のステップでパラメーター・パックの 1 つのメンバーにアクセスし、引数リストの残りのインスタンス生成を再帰的に行い、すべてのエレメントを取得できます。以下に例を示します。

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

struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};
struct a6{}; struct a7{}; struct a8{}; struct a9{}; struct a10{};

template<typename X1, typename X2> struct foo{foo();};
template<typename X3, typename X4> foo<X3,X4>::foo(){cout<<"primary foo"<<endl;};
template<> struct foo<a1,a2>{foo(){cout<<"ctor foo<a1,a2>"<<endl;}};
template<> struct foo<a3,a4>{foo(){cout<<"ctor foo<a3,a4>"<<endl;}};
template<> struct foo<a5,a6>{foo(){cout<<"ctor foo<a5,a6>"<<endl;}};
template<> struct foo<a7,a8>{foo(){cout<<"ctor foo<a7,a8>"<<endl;}};
template<> struct foo<a9,a10>{foo(){cout<<"ctor foo<a9,a10>"<<endl;}};

template<typename...T>struct bar{bar{}{cout<<"bar primary"<<endl;}};

template<typename A, typename B, typename...T1, typename...T2>
struct bar<foo<A,B>,foo<T1,T2>...>{
    foo<A,B> data;
    bar<foo<T1,T2>...>data1;
};

template<> struct bar<foo<a9,a10> > {bar(){cout<<"ctor bar<foo<a9,a10>>"<<endl;}};

int main(){
    bar<foo<a1,a2>,foo<a3,a4>,foo<a5,a6>,foo<a7,a8>,foo<a9,a10> > t2;
    return 0;
}
```

この例の出力は次のとおりです。

```
ctor foo<a1,a2>
ctor foo<a3,a4>
ctor foo<a5,a6>
ctor foo<a7,a8>
ctor bar<foo<a9,a10>
```

テンプレート引数の推定

パラメーター・パックは、他の通常のテンプレート・パラメーターと同様に、テンプレート引数の推定によって推定することができます。以下の例は、テンプレート引数の推定により関数呼び出しからパックを展開する方法を示しています。

```
template<class...A> void func(A...args){}

int main(void){
    func(1,2,3,4,5,6);
    return 0;
}
```

この例では、関数引数リストは (1,2,3,4,5,6) です。各関数引数は、型 int に推定されるため、テンプレート・パラメーター・パック A の推定される型のリストは、 (int,int,int,int,int,int) になります。すべて展開されると、関数テンプレート func は void func(int,int,int,int,int,int) としてインスタンスが生成されます。これは、展開された関数仮パラメーター・パックを持つテンプレート関数です。

この例で、関数呼び出しステートメント func(1,2,3,4,5,6) を func() に変更すると、テンプレート引数の推定では、テンプレート・パラメーター・パック A が空であると推定します。

```
template<class...A> void func(A...args){}

int main(void){
    func();
    return 0;
}
```

テンプレート引数の推定では、以下の例に示すように、テンプレートのインスタンス生成からパックを展開できます。

```
#include <cstdio>

template<int...A> struct container{
    void display(){printf("YIKES¥n");}
};

template<int B, int...C> struct container<B,C...>{
    void display(){
        printf("spec %d¥n",B);
        container<C...>test;
        test.display();
    }
 };

template<int C> struct container<C>{
    void display(){printf("spec %d¥n",C);}
};

int main(void)
{
   printf("start¥n¥n");
```

```
   container<1,2,3,4,5,6,7,8,9,10> test;
   test.display();
   return 0;
}
```

この例の出力は次のようになります。

```
start

spec 1
spec 2
spec 3
spec 4
spec 5
spec 6
spec 7
spec 8
spec 9
spec 10
```

この例では、クラス・テンプレート container の部分的特殊化は、template<int B, int...C> struct container<B,C...> です。部分的特殊化は、クラス・テンプレートが container<1,2,3,4,5,6,7,8,9,10> にインスタンスを生成される場合に一致します。テンプレート引数の推定では、部分的特殊化の引数リストからテンプレート・パラメーター・パック C およびパラメーター B を推定します。テンプレート引数の推定では、パラメーター B を 1、パック拡張 C... をリスト (2,3,4,5,6,7,8,9,10)、およびテンプレート・パラメーター・パック C を (int,int,int,int,int,int,int,int,int) の型のリストと推定します。

ステートメント container<1,2,3,4,5,6,7,8,9,10> test を container<1> test に変更すると、テンプレート引数の推定では、テンプレート・パラメーター・パック C が空であると推定します。

テンプレート引数の推定では、明示的テンプレート引数の検出後にパックを展開できます。次の例を検討してみます。

```
#include <cassert>

template<class...A> int func(A...arg){
    return sizeof...(arg);
}

int main(void){
    assert(func<int>(1,2,3,4,5) == 5);
    return 0;
}
```

この例では、テンプレート・パラメーター・パック A は、明示的引数リストおよび関数呼び出し中の引数を使って、型のリスト (int,int,int,int,int) に推定されます。

関連資料:

184 ページの『sizeof 演算子』

440 ページの『テンプレート・パラメーター』

443 ページの『テンプレート引数』

447 ページの『クラス・テンプレート』

453 ページの『関数テンプレート』

455 ページの『テンプレート引数の推定』
473 ページの『部分的特殊化』
555 ページの『C++11 互換性の拡張機能』

名前のバインディングおよび従属名

名前のバインディング は、テンプレートで明示的に、または暗黙的に使用されている名前ごとに宣言を検出する処理です。コンパイラーは、テンプレートの定義で名前をバインドしたり、またはテンプレートのインスタンス生成において名前をバインドしたりすることがあります。

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。次の例を検討してみます。

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

void f(double) { cout << "Function f(double)" << endl; }

template <class A> struct container{ // point of definition of container
   void member1(){
      // This call is not template dependent,
      // because it does not make any use of a template parameter.
      // The name is resolved at the point of definition, so f(int) is not visible.
      f(1);
   }
   void member2(A arg);
};

void f(int) { cout << "Function f(int)" << endl; }

void h(double) { cout << "Function h(double)" << endl; }

template <class A> void container<A>::member2(A arg){
   // This call is template dependent, so qualified name lookup only finds
   // names visible at the point of instantiation.
   ::h(arg);
}

template struct container<int>; // point of instantiation of container<int>

void h(int) { cout << "Function h(int)" << endl; }

int main(void){
   container<int> test;
```

```
   test.member1();
   test.member2(10);
   return 0;
}
```

この例の出力は次のようになります。

```
Function f(double)
Function h(double)
```

テンプレートの定義のポイント は、それの定義の直前に配置されます。この例では、関数テンプレート container 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(1) は、テンプレート・パラメーターに依存しないので、コンパイラーは、テンプレート container 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、関数呼び出し f(1) は、f(double) を呼び出します。 f(int) のほうが的確ですが、それは、container の定義のポイントのあるスコープ内に存在していません。

テンプレートのインスタンス生成のポイント は、その使用を囲む宣言の直前に配置されます。この例では、container<int> のインスタンス生成のポイントは、明示的インスタンス生成のロケーションです。修飾された関数呼び出し ::h(arg) はテンプレート引数 arg に依存するので、コンパイラーは、container<int> のインスタンス生成の前に宣言された名前を考慮します。したがって、関数呼び出し h(arg) は h(double) を呼び出します。この関数が container<int> のインスタンス生成のポイントのあるスコープ内に存在しないので、コンパイラーは h(int) を考慮しません。

インスタンス生成バインディングのポイントは、次のことを意味しています。

- テンプレート・パラメーターは、ローカル名、またはクラス・メンバーに依存できない。
- テンプレートの非修飾名は、ローカル名、またはクラス・メンバーに依存できない。

C++11

decltype 機能は、テンプレート従属名と対話することができます。decltype(*expression*) 型指定子のオペランド *expression* 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、以下の例のように、テンプレートのインスタンス生成の前に *expression* の妥当性を判断できません。

```
template <class T, class U> int h(T t, U u, decltype(t+u) v);
```

この例では、関数テンプレート h のインスタンス生成後に、オペランド t+u が無効である場合、コンパイラーはエラー・メッセージを出します。

詳細については、 93 ページの『decltype(*expression*) 型指定子 (C++11)』を参照してください。

C++11

関連資料:

463 ページの『テンプレートのインスタンス生成』

typename キーワード

型を参照する修飾名やテンプレート・パラメーターに依存する修飾名がある場合は、キーワード typename を使用してください。キーワード typename のみを、テンプレート宣言または定義で使用してください。次の例を検討してみます。

```
template<class T> class A
{
  T::x(y);
  typedef char C;
  A::C d;
}
```

ステートメント T::x(y) は、あいまいです。そのステートメントは、非ローカル引数 y を使用した関数 x() の呼び出しや、または、型 T::x を使用して変数 y の宣言にすることができます。 C++ は、このステートメントを関数呼び出しとして解釈します。コンパイラーにこのステートメントを宣言として解釈させるには、キーワード typename を T:x(y) 開始位置に追加する必要があります。ステートメント A::C d; は、不適格です。クラス A は、A<T> も参照するので、テンプレート・パラメーターに依存します。キーワード typename をこの宣言の開始位置に追加する必要があります。

```
  typename A::C d;
```

テンプレート・パラメーター宣言で、キーワード class の代わりに、キーワード typename も使用できます。

関連資料:

440 ページの『テンプレート・パラメーター』

修飾子としての template キーワード

メンバー・フォームと他のエンティティーを区別するために、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>();  // ill-formed
}
```

この例では、変数 object_x の定義は不適格です。コンパイラーは、シンボル < が「より小演算子」であると見なします。コンパイラーにテンプレート関数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);
}
```

この例の出力は、以下のとおりです。

```
Primary: 100
Specialized, non-type argument = 20
member template's member function: 40
```

これらのケースでキーワード `template` を使用しない場合、コンパイラーは、`<` を「より小演算子」として解釈します。例えば、次のコード行は、不適格です。

```
p->f<100>();
```

コンパイラーは、`f` を非テンプレート・メンバーとして、 `<` を「より小演算子」として解釈します。

第 16 章 例外処理 (C++ のみ)

例外処理 は、例外的な状況を検出したり、処理するコードを、プログラムの他の部分から分離するメカニズムです。例外的な状況は、必ずしもエラーではないことに注意してください。

関数が例外的な状況を検出する場合、オブジェクトでこれを表します。このオブジェクトを*例外オブジェクト* と呼びます。例外的な状況を処理するには、*例外をスロー*します。これが、例外をスローした関数の直接的または間接的呼び出し側のコードの指定ブロックに、制御だけでなく例外も渡します。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、生成コードと共に、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 されます。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、次の要素で構成されます。

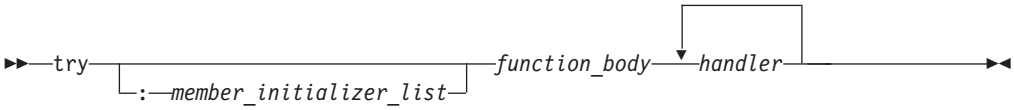
- `try` blocks
- `catch` blocks
- `throw` expressions
- 507 ページの『例外指定』

try ブロック

try ブロック を使用して、即座に処理したい例外をスローする可能性のある、プログラムのエリアを指示します。*関数 try ブロック* を使用して、関数の本体全部で例外を検出したいことを指示します。

try ブロックの構文

関数 try ブロックの構文



以下のコードは、メンバー初期化指定子、関数 try ブロック、および try ブロックを持つ関数 try ブロックの例です。

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

```
class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { }
};

class A {
   public:
      int i;

      // A function try block with a member
      // initializer
      A() try : i(0) {
         throw E("Exception thrown in A()");
      }
      catch (E& e) {
         cout << e.error << endl;
      }
};

// A function try block
void f() try {
   throw E("Exception thrown in f()");
}
catch (E& e) {
   cout << e.error << endl;
}

void g() {
   throw E("Exception thrown in g()");
}

int main() {
   f();

   // A try block
   try {
      g();
   }
   catch (E& e) {
      cout << e.error << endl;
   }
   try {
      A x;
   }
   catch(...) { }
}
```

上記の例の出力は、以下のとおりです。

```
Exception thrown in f()
Exception thrown in g()
Exception thrown in A()
```

クラス A のコンストラクターには、メンバー初期化指定子を持つ関数 try ブロックがあります。関数 f() には、関数 try ブロックがあります。 main() 関数は、try ブロックを含んでいます。

関連資料:

418 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、throw が内側の try ブロックによって呼び出された関数内で起きる場合は、制御は、引数が throw 式の引数と一致する最初の catch ブロックが見つかるまで、ネストされた try ブロックを介して外側に向けて渡されて行きます。

次に例を示します。

```
try
{
      func1();
      try
      {
            func2();
      }
      catch (spec_err) { /* ... */ }
      func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

上記の例で、spec_err が内側の try ブロック (この場合は、func2() から) 内でスローされる場合、内側の catch ブロックがこの例外をキャッチします。この catch ブロックが制御権移動を行わない場合は、func3() が呼び出されます。内側の try ブロックの後で spec_err がスローされた場合 (例えば func3() によって)、この例外はキャッチされず、関数 terminate() が呼び出されます。内側の try ブロックの中の func2() からスローされた例外が、type_err である場合、プログラムは、func3() を呼び出さずに、両方の try ブロックから出て 2 番目の catch ブロックにスキップします。内側の try ブロックの後には、適切な catch ブロックがないためです。

catch ブロック内で try ブロックをネストすることもできます。

catch ブロック

catch ブロックの構文

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オブジェクトは、catch キーワードの後に続く小括弧の中 (*exception_declaration*) で宣言されます。スカラー・オブジェクトとクラス・オブジェクトの両方をキャッチできます。cv 修飾されたオブジェクトをキャッチすることもできます。例外宣言は左辺値参照を宣言できます。その場合、例外オブジェクトは参照によってキャッチ・ハンドラーに渡されます。*exception_declaration* を不完全型、抽象クラス型 C++11 右辺値参照型 C++11 、および以下の型以外の不完全型へのポインターや参照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、スローされたオブジェクトにアクセスする直接的方法はありません。 `catch(...)` によってキャッチされた例外に関する情報は、非常に限られています。

catch ブロック内にあるスローされたオブジェクトにアクセスしたい場合は、オプションの変数名を宣言することができます。

catch ブロックはアクセス可能オブジェクトしかキャッチできません。キャッチされたオブジェクトには、アクセス可能なコピー・コンストラクターがあるはずです。

関連資料:

101 ページの『型修飾子』

370 ページの『メンバー・アクセス』

126 ページの『参照 (C++ のみ)』

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時間が、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で、または名前空間スコープ・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。この例の意図は、`~B()` の例外は `main()` の関数 try ブロックによってキャッチされるが、`~A()` の例外は、`~A()` が `main()` の完了後に実行されるため、キャッチされないことを示すことです。

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

class E {
public:
  const char* error;
```

```
  E(const char* arg) : error(arg) { }
};

class A {
  public: ~A() { throw E("Exception in ~A()"); }
};

class B {
  public: ~B() { throw E("Exception in ~B()"); }
};

int main() try {
  cout << "In main" << endl;
  static A cow;
  B bull;
}
catch (E& e) {
  cout << e.error << endl;
}
```

上記の例の出力は、次のとおりです。

```
In main
Exception in ~B()
```

ランタイムは、オブジェクト cow が、プログラムの終わりに破棄される時にスローされる例外をキャッチできません。

次の例は、名前空間スコープ・オブジェクトのコンストラクターから例外をスローします。

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

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};

namespace N {
  class C {
  public:
    C() {
      cout << "In C()" << endl;
      throw E("Exception in C()");
    }
  };

  C calf;
};

int main() try {
  cout << "In main" << endl;
}
catch (E& e) {
  cout << e.error << endl;
}
```

上記の例の出力は、次のとおりです。

```
In C()
```

コンパイラーは、オブジェクト calf の作成時にスローされる例外をキャッチできません。

関数 try ブロックのハンドラーでは、コンストラクター本体、またはデストラクター本体にジャンプすることはできません。

リターン・ステートメントは、コンストラクターの関数 try ブロック・ハンドラー内に置くことはできません。

オブジェクトのコンストラクター、またはデストラクターの関数 try ブロック・ハンドラーに入ると、そのオブジェクトの完全な構成の基底クラスおよびメンバーは、破棄されます。次の例は、このことを示しています。

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

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class B {
   public:
      B() { };
      ~B() { cout << "~B() called" << endl; };
};

class D : public B {
   public:
      D();
      ~D() { cout << "~D() called" << endl; };
};

D::D() try : B() {
   throw E("Exception in D()");
}
catch(E& e) {
   cout << "Handler of function try block of D(): " << e.error << endl;
};

int main() {
   try {
      D val;
   }
   catch(...) { }
}
```

上記の例の出力は、次のとおりです。

```
~B() called
Handler of function try block of D(): Exception in D()
```

D() の関数 try ブロックのハンドラーに入ると、ランタイムは、まず最初に D の基底クラスのデストラクター、つまり B を呼び出します。 val が完全に構成されていないので、D のデストラクターは呼び出されません。

ランタイムは、コンストラクターまたはデストラクターの関数 try ブロックのハンドラーの終了時に、例外を再スローします。その他のすべての関数は、関数 try ブロックのハンドラーの終了に到達した時点でリターンします。次の例は、このことを示しています。

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

class E {
```

```
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class A {
   public:
      A() try { throw E("Exception in A()"); }
      catch(E& e) { cout << "Handler in A(): " << e.error << endl; }
};

int f() try {
   throw E("Exception in f()");
   return 0;
}
catch(E& e) {
   cout << "Handler in f(): " << e.error << endl;
   return 1;
}

int main() {
   int i = 0;
   try { A cow; }
   catch(E& e) {
      cout << "Handler in main(): " << e.error << endl;
   }

   try { i = f(); }
   catch(E& e) {
      cout << "Another handler in main(): " << e.error << endl;
   }

   cout << "Returned value of f(): " << i << endl;
}
```

上記の例の出力は、次のとおりです。

```
Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1
```

C++11

委任プロセスが存在し、例外がターゲット・コンストラクターの本体で発生した場合、その例外は委任コンストラクターの try ブロックにある適切なハンドラーによってキャッチできます。次の例は、このことを示しています。

```
#include <cstdio>
using std::printf;

int global_argc;

struct A{
   int _x;
   A();
   A(int);
};

A::A(int x):_x((printf("In A::A(int) initializer for A::_x.¥n"),x)){
   printf("In A::A(int) constructor body.¥n");

   if(global_argc % 2 !=0){
      printf("Will throw.¥n");
      throw 0;
   }
```

```
   printf("Will not throw.¥n");
}

A::A() try:A((printf("In A::A() initializer for delegating to A::A(int).¥n"),42)){
   printf("In A::A() function-try-block body.¥n");
}
catch(...){
   printf("In catch(...) handler for A::A() function-try-block.¥n");
}

int main(int argc, char **argv){
   printf("In main().¥n");
   global_argc = argc;
   try{
      A a;
      printf("Back in main().¥n");
   }
   catch(...){
      printf("In catch(...) handler for try-block in main().¥n");
   }
   return 0;
}
```

この例は、生成されるプログラムの呼び出し時に受け渡される引数の数に応じて、異なる出力を生成できます。引数が偶数個の場合、例外がスローされます。出力は次のとおりです。

```
In main().
In A::A() initializer for delegating to A:A(int).
In A::A(int) initializer for A::_x.
In A::A(int) constructor body.
Will throw.
In catch(...) handler for A::A() function-try-block.
In catch(...) handler for try-block in main().
```

引数が奇数個の場合、例外はスローされません。出力は次のとおりです。

```
In main().
In A::A() initializer for delegating to A::A(int).
In A::A(int) initializer for A::_x.
In A::A(int) constructor body.
Will not throw.
In A::A() function-try-block body.
Back in main().
```

詳細については、 412 ページの『委任コンストラクター (C++11)』を参照してください。

C++11

関連資料:

298 ページの『main() 関数』

56 ページの『静的ストレージ・クラス指定子』

313 ページの『第 9 章 名前空間 (C++ のみ)』

423 ページの『デストラクター』

catch ブロックの引数

catch ブロックの引数に対してクラス型を指定する場合 (*exception_declaration*)、コンパイラーは、コピー・コンストラクターを使用して、その引数を初期化します。そ

の引数に名前が入っていなければ、コンパイラーは、一時オブジェクトを初期化し、ハンドラーが終了するとき、それを破棄します。

ISO C++ の仕様では、一時オブジェクトが重複している場合、コンパイラーは、重複した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生成します。プログラムをデバッグする場合、特にメモリー問題のデバッグにおいては、このことを考慮に入れてください。

スローされた例外とキャッチされた例外のマッチング

ハンドラーの catch 引数の中の引数は、次のいずれかの条件が満たされる場合、throw 式 (throw 引数) の *assignment_expression* の引数に一致します。

- catch 引数の型が、スローされたオブジェクトの型と一致する。
- catch 引数が、スローされたクラス・オブジェクトのパブリック基底クラスである。
- catch がポインターの型を指定し、スローされたオブジェクトが、標準ポインター型変換によって catch 引数のポインター型に変換できるポインター型である。

注: スローされたオブジェクトの型が `const` または `volatile` である場合、一致するには、catch 引数も `const` または `volatile` であることが必要です。ただし、`const`、`volatile`、または参照型の catch 引数が、非定数、非 volatile、または非参照オブジェクト型と一致することがあります。非参照 catch 引数型は、同じ型のオブジェクトへの参照と一致します。

関連資料:

160 ページの『ポインター型変換』

101 ページの『型修飾子』

126 ページの『参照 (C++ のみ)』

キャッチの順序

コンパイラーが try ブロックで例外と遭遇する場合、その出現順に各ハンドラーを試行します。

基底クラスのオブジェクト用の catch ブロックが、その基底クラスから派生するクラスのオブジェクト用の catch ブロックより前にある場合は、コンパイラーは警告を発行し、派生クラス・ハンドラー内に到達不能コードがあっても、プログラムのコンパイルを続行します。

`catch(...)` の書式の catch ブロックは、try ブロックの後に続く最後の catch ブロックでなければならず、そうでない場合はエラーが起こります。このように配置すると、`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のが妨げられません。

ランタイムが、現行スコープ内に一致するハンドラーを検出できない場合、ランタイムは、動的な囲み try ブロック内で一致するハンドラーの検出を継続します。次の例は、このことを示しています。

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

```
class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { };
};

class F : public E {
public:
  F(const char* arg) : E(arg) { };
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    throw E("Class E exception");
  }
  catch (F& e) {
    cout << "In handler of f()";
    cout << e.error << endl;
  }
};

int main() {
  try {
    cout << "In main" << endl;
    f();
  }
  catch (E& e) {
    cout << "In handler of main: ";
    cout << e.error << endl;
  };
  cout << "Resume execution in main" << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In main
In try block of f()
In handler of main: Class E exception
Resume execution in main
```

関数 f() では、ランタイムは、スローされた型 E の例外を処理するハンドラーを検出できません。ランタイムは、動的な囲み try ブロック内、つまり main() 関数の try ブロック内で、一致するハンドラーを検出します。

ランタイムが、プログラムで一致するハンドラーを検出できない場合、terminate() 関数を呼び出します。

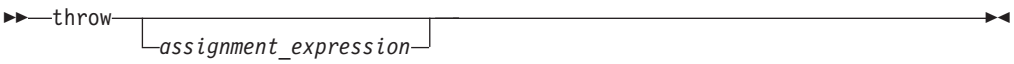
関連資料:

493 ページの『try ブロック』

throw 式

throw 式 は、プログラムで例外が生じたことを示すために使用します。

throw 式の構文



assignment_expression の型は、不完全型、抽象クラス型、または以下の型を除く不完全型を指すポインターにすることはできません。

- `void*`
- `const void*`
- `volatile void*`
- `const volatile void*`

assignment_expression は、呼び出しでの関数引数、またはリターン・ステートメントのオペランドと同じように扱われます。

assignment_expression が、クラス・オブジェクトの場合、そのオブジェクトのコピー・コンストラクターおよびデストラクターは、アクセス可能でなければなりません。例えば、プライベートとして宣言されたコピー・コンストラクターを持つクラス・オブジェクトをスローすることはできません。そのオブジェクトのコピー C++11 または移動 C++11 に用いられるコンストラクターは、多重定義の解決によって選択されます。

assignment_expression が、0 に評価する整数型の整数定数式である場合、この *assignment_expression* は、ポインターのハンドラーまたはメンバー型へのポインターに一致しません。

関連資料:

不完全型

例外の再スロー

catch ブロックが、キャッチ済みの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スローします。

例外は、rethrow 式が発生するスコープで既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 ブロックへ再スローされます。したがって、rethrow 式の発生したスコープの catch ブロックは、その例外を処理できなくなります。動的な囲み try ブロックの catch ブロックは、いずれも、例外をキャッチする機会があります。

次の例は、例外の再スローを示しています。

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

struct E {
  const char* message;
  E() : message("Class E") { }
};

struct E1 : E {
  const char* message;
  E1() : message("Class E1") { }
};

struct E2 : E {
  const char* message;
  E2() : message("Class E2") { }
};
```

```
void f() {
  try {
    cout << "In try block of f()" << endl;
    cout << "Throwing exception of type E1" << endl;
    E1 myException;
    throw myException;
  }
  catch (E2& e) {
    cout << "In handler of f(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E1& e) {
    cout << "In handler of f(), catch (E1& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E& e) {
    cout << "In handler of f(), catch (E& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
}

int main() {
  try {
    cout << "In try block of main()" << endl;
    f();
  }
  catch (E2& e) {
    cout << "In handler of main(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
  }
  catch (...) {
    cout << "In handler of main(), catch (...)" << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In try block of main()
In try block of f()
Throwing exception of type E1
In handler of f(), catch (E1& e)
Exception: Class E1
In handler of main(), catch (...)
```

main() 関数の try ブロックは、関数 f() を呼び出します。関数 f() の try ブロックは、myException という名前の、型 E1 のオブジェクトをスローします。ハンドラー catch (E1 &e) が、myException キャッチします。それからハンドラーは、ステートメント throw を使用して、myException を、次の動的な囲み try ブロック、つまり main() 関数の try ブロックに再スローします。ハンドラー catch(...) が、myException をキャッチします。

スタック・アンワインド

例外がスローされ、制御が try ブロックからハンドラーに渡されると、 C++ ランタイムは、try ブロックの開始以降に作成されたすべての自動オブジェクトに対して、デストラクターを呼び出します。この処理は、*スタック・アンワインド* と呼ばれます。自動オブジェクトは、その作成の逆順で破棄されます。 (自動オブジェクトは、auto または register と宣言されているか、あるいは static または

extern と宣言されていない、ローカル・オブジェクトです。自動オブジェクト x は、x が宣言されているブロックのプログラムの終了時に、必ず削除されます。)

例外が、サブオブジェクトまたは配列エレメントを含むオブジェクトの作成中にスローされる場合、デストラクターは、例外がスローされる前に正常に作成されたサブオブジェクトまたは配列エレメントに対してのみ、呼び出されます。ローカル静的オブジェクトに対するデストラクターは、オブジェクトが正常に作成された場合にのみ呼び出されます。

スタック・アンワインド中にデストラクターが例外をスローし、その例外が処理されない場合、terminate() 関数が呼び出されます。次の例は、このことを示しています。

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

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_terminate() {
  cout << "Call to my_terminate" << endl;
};

struct A {
  A() { cout << "In constructor of A" << endl; }
  ~A() {
    cout << "In destructor of A" << endl;
    throw E("Exception thrown in ~A()");
  }
};

struct B {
  B() { cout << "In constructor of B" << endl; }
  ~B() { cout << "In destructor of B" << endl; }
};

int main() {
  set_terminate(my_terminate);

  try {
    cout << "In try block" << endl;
    A a;
    B b;
    throw("Exception thrown in try block of main()");
  }
  catch (const char* e) {
    cout << "Exception: " << e << endl;
  }
  catch (...) {
    cout << "Some exception caught in main()" << endl;
  }

  cout << "Resume execution of main()" << endl;
}
```

この例の出力は次のようになります。

```
In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate
```

try ブロックでは、2 つの自動オブジェクト、a と b が作成されます。 try ブロックは、型 const char* の例外をスローします。ハンドラー catch (const char* e) が、この例外をキャッチします。 C++ ランタイムは、スタックをアンワインドし、a および b のデストラクターを、それらが作成された逆順で呼び出します。 a のデストラクターが、例外をスローします。プログラムにこの例外を処理できるハンドラーが存在しないので、C++ ランタイムは terminate() を呼び出します。 (関数 terminate() は、set_terminate() に引数として指定した関数を呼び出します。この例では、terminate() は、my_terminate() を呼び出すよう指定されています。)

C++11

委任コンストラクター機能が使用可能な場合に、委任コンストラクターの本体に例外がスローされると、ターゲット・コンストラクターを介して構成されるオブジェクトのデストラクターが自動的に呼び出されます。デストラクターは、適宜、サブオブジェクトのデストラクターを呼び出す方法で呼び出す必要があります。特に、仮想基底クラスがターゲット・コンストラクターを介して作成される場合、仮想基底クラスのデストラクターを呼び出す必要があります。

委任コンストラクターの本体に例外がスローされると、デストラクターはターゲット・コンストラクターによって作成されるオブジェクトに対して呼び出されます。例外が非委任コンストラクターからエスケープされる場合、アンワインド機構は完全に構成されたサブオブジェクトに対してデストラクターを呼び出します。次の例は、このことを示しています。

```
class D{
  D():D('a') { printf("D:D().¥n");}

  D:D(char) try: D(55){
    printf("D::D(char). Throws.¥n");
    throw 0;
  }
  catch(...){
    printf("D::D(char).Catch block.¥n");
  }

  D:D(int i):i(i_) {printf("D::D(int).¥n");}

  D:~D() {printf("D::~D().¥n");}
}

int main(void){
  D d;
}
```

この例の出力は次のとおりです。

```
D::D(int).
D::D(char).Throws.
D::~D().
D::D(char).Catch block.
```

この例では、例外が委任コンストラクター D:D(char) で発生するため、デストラクター D:~D() がオブジェクト d に対して呼び出されます。

詳細については、 412 ページの『委任コンストラクター (C++11)』を参照してください。

C++11

例外指定

C++ には、特定の関数が、指定されたリストの例外だけをスローするように限定するメカニズムがあります。任意の関数の先頭に例外を指定すると、関数の呼び出し元に対して、その関数が例外の指定に含まれていない例外を直接にも間接にもスローしないことを保証することができます。

例えば、次の関数を考えます。

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

これは、型が unknown_word または bad_grammar である例外オブジェクト、あるいは unknown_word または bad_grammar から派生した型の例外オブジェクトのみをスローすることを明示的に示しています。

例外指定の構文

type_id_list は、コンマで区切られた型のリストです。このリストでは、不完全型、抽象クラス型、C++11 右辺値参照型C++11 、または以下の型以外の不完全型へのポインターや参照を指定できません。

- void*
- const void*
- volatile void*
- const volatile void*

type_id_list の型は cv 修飾子で修飾できますが、例外指定で型を定義することはできません。

例外指定のない関数は、すべての例外のスローを認めます。空の *type_id_list* を持つ例外指定を使用する関数、throw() は、例外のスローを許可しません。

例外指定は関数の型の一部ではありません。

例外指定は、関数、関数を指すポインター、関数に対する参照、またはメンバー関数を指すポインターの宣言または定義におけるトップレベルの関数宣言子の終端にのみ記述できます。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 1 つだとします。その関数は、クラス A またはクラス A から public に派生したクラスの例外オブジェクトをスローします。次の例は、このことを示しています。

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

void f(int i) throw (A) {
   switch (i) {
      case 0: throw A();
      case 1: throw B();
      default: throw C();
   }
}

void g(int i) throw (A*) {
   A* a = new A();
   B* b = new B();
   C* c = new C();
   switch (i) {
      case 0: throw a;
      case 1: throw b;
      default: throw c;
   }
}
```

関数 f() は、型 A または B のオブジェクトをスローすることができます。関数が型 C のオブジェクトをスローしようとする場合、コンパイラーは、C が関数の例外指定に指定されてもいないし、それが A から public に派生したものでもないので、unexpected() を呼び出します。同様に、関数 g() は、型 C のオブジェクトを指すポインターをスローできません。関数は、型 A のポインター、または A から public に派生するオブジェクトのポインターをスローします。

仮想関数をオーバーライドする関数は、その仮想関数が指定する例外のみをスローすることができます。次の例は、このことを示しています。

```
class A {
   public:
      virtual void f() throw (int, char);
};

class B : public A{
   public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
   class C : public A {
      public: void f() { }
   };

   class D : public A {
      public: void f() throw (int, char, double) { }
   };
*/
```

コンパイラーは、メンバー関数が、型 int の例外のみをスローするので、B::f() を認めます。コンパイラーは、メンバー関数が、どの種類の例外もスローするので、C::f() を許可しません。コンパイラーは、メンバー関数が、A::f() よりも多くの型の例外 (int、char、および double) をスローするので、D::f() を許可しません。

x という名前の関数を指すポインター、または y という名前の関数を指すポインターの割り当て、または初期化を行うとします。関数 x を指すポインターは、y の例外指定が指定する例外のみをスローできます。次の例は、このことを示しています。

```
void (*f)();
void (*g)();
void (*h)() throw (int);

void i() {
   f = h;
//   h = g;  This is an error.
}
```

コンパイラーは、f がどのような種類の例外もスローできるので、割り当て f = h を認めます。コンパイラーは、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、割り当て h = g を許可しません。

暗黙的に宣言された特殊メンバー関数 (デフォルトのコンストラクター、コピー・コンストラクター、デストラクター、およびコピー割り当て演算子) には、例外指定があります。暗黙的に宣言された特殊メンバー関数の例外指定の中には、その特殊関数が起動する対象の関数の例外指定の中で宣言されている型が含められます。特別な関数を起動する関数が、例外をすべて許可する場合は、特別な関数も例外をすべて許可します。特別な関数が呼び出す関数のすべてが、例外を許可しない場合は、特別な関数も例外を許可しません。次の例は、このことを示しています。

```
class A {
   public:
      A() throw (int);
      A(const A&) throw (float);
      ~A() throw();
};

class B {
   public:
      B() throw (char);
      B(const A&);
      ~B() throw();
};

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

上記の例で示した次の特別な関数は、暗黙的に宣言されています。

```
C::C() throw (int, char);
C::C(const C&);   // Can throw any type of exception, including float
C::~C() throw();
```

デフォルトの C のコンストラクターは、型 int または char の例外をスローできます。 C のコピー・コンストラクターは、どのような種類の例外もスローできます。 C のデストラクターは、いかなる例外もスローできません。

関連資料:

不完全型

263 ページの『関数の宣言と定義』

307 ページの『関数を指すポインター』

409 ページの『第 14 章 特殊メンバー関数 (C++ のみ)』

126 ページの『参照 (C++ のみ)』

特殊な例外処理関数

スローされたすべてのエラーが catch ブロックによってキャッチされ、正常に処理されるというわけではありません。ある状況においては、例外を処理する最良の方法は、プログラムを終了することです。 C++ には、catch ブロックによって正しく処理できない例外、または有効な try ブロックの外部にスローされる例外の処理のために、 2 つの特殊なライブラリー関数がインプリメントされています。これらの関数は次のとおりです。

- 『unexpected() 関数』
- 511 ページの『terminate() 関数』

unexpected() 関数

例外指定を持つ関数が、例外指定にリストされていない例外をスローすると、 C++ ランタイムは、下記を行います。

1. unexpected() 関数が呼び出されます。
2. unexpected() 関数は、unexpected_handler によって指定された関数を呼び出します。デフォルトでは、unexpected_handler は、関数 terminate() を指します。

unexpected_handler のデフォルト値を、関数 set_unexpected() を使用して置き換えることができます。

unexpected() は、リターンできませんが、例外をスロー (または再スロー) することはできます。関数 f() の例外指定が、違反されていたとします。 unexpected() が f() の例外指定で許可された例外をスローする場合は、C++ ランタイムは、f() の呼び出しで別のハンドラーを検索します。次の例は、このことを示しています。

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

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_unexpected() {
  cout << "Call to my_unexpected" << endl;
  throw E("Exception thrown from my_unexpected");
}

void f() throw(E) {
  cout << "In function f(), throw const char* object" << endl;
  throw("Exception, type const char*, thrown from f()");
}
```

```
int main() {
  set_unexpected(my_unexpected);
  try {
    f();
  }
  catch (E& e) {
    cout << "Exception in main(): " << e.message << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In function f(), throw const char* object
Call to my_unexpected
Exception in main(): Exception thrown from my_unexpected
```

main() 関数の try ブロックは、関数 f() を呼び出します。関数 f() は、型 const char* のオブジェクトをスローします。しかし、f() の例外指定は、型 E のオブジェクトのみをスローすることを許可します。関数 unexpected() が呼び出されます。関数 unexpected() は、my_unexpected() を呼び出します。関数 my_unexpected() は、型 E のオブジェクトをスローします。 unexpected() は、f() の例外指定で許可されたオブジェクトをスローするので、 main() 関数にあるハンドラーは、その例外を処理できます。

unexpected() が、f() の例外指定で許可されたオブジェクトをスロー (または再スロー) しない場合は、C++ ランタイムは、2 つのことを行います。

- f() の例外指定にクラス std::bad_exception がある場合、unexpected() は、型 std::bad_exception のオブジェクトをスローし、C++ ランタイムは、f() の呼び出しで別のハンドラーを検索します。
- f() の例外指定にクラス std::bad_exception がない場合、関数 terminate() が呼び出されます。

関連資料:

510 ページの『特殊な例外処理関数』

513 ページの『set_unexpected() および set_terminate() 関数』

terminate() 関数

一部のケースでは、例外処理メカニズムが失敗して、void terminate() が呼び出されます。この terminate() の呼び出しは、次のいずれかの状況で行われます。

- 例外処理メカニズムが、スローされた例外のハンドラーを検出できません。以下は、より具体的なケースです。
  - スタック・アンワインド中に、デストラクターが例外をスローし、その例外が処理されません。
  - スローされた例外が、また例外をスローし、その例外が処理されません。
  - 非ローカル静的オブジェクトのコンストラクター、またはデストラクターが、例外をスローし、その例外が処理されません。
  - atexit() で登録された関数が例外をスローし、その例外が処理されません。以下に、このことを示します。
- オペランドを指定しない throw 式が、例外を再スローしようとし、現在処理されている例外がありません。

- 関数 f() が、その例外指定に違反する例外をスローします。 unexpected() 関数が、f() の例外指定に違反する例外をスローし、f() の例外指定が、クラス std::bad_exception を含んでいませんでした。
- unexpected_handler のデフォルト値が呼び出されます。

以下の例では、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ない場合に、void terminate() が呼び出されます。

```
extern "C" printf(char* ...);
#include <exception>
#include <cstdlib>
using namespace std;

void f() {
  printf("Function f()¥n");
  throw "Exception thrown from f()";
}

void g() { printf("Function g()¥n"); }
void h() { printf("Function h()¥n"); }

void my_terminate() {
  printf("Call to my_terminate¥n");
  abort();
}

int main() {
  set_terminate(my_terminate);
  atexit(f);
  atexit(g);
  atexit(h);
  printf("In main¥n");
}
```

上記の例の出力は、次のとおりです。

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

atexit() で関数を登録するには、登録したい関数を指すポインターに、atexit() へのパラメーターを渡します。標準プログラム終了処理で atexit() は、引数のない登録済み関数を逆順で呼び出します。 atexit() 関数は、<cstdlib> ライブラリーに入っています。

terminate() 関数は、terminate_handler 関数によって指定された関数を呼び出します。デフォルトで terminate_handler は、プログラムから終了する関数 abort() を指します。 terminate_handler のデフォルト値を、関数 set_terminate() に置き換えることができます。

return を使用しても、または例外をスローしても、終了関数を呼び出し元に戻すことはできません。

関連資料:

513 ページの『set_unexpected() および set_terminate() 関数』

set_unexpected() および set_terminate() 関数

関数 unexpected() は、起動されると、set_unexpected() に、一番最近に引数として供給された関数を呼び出します。 set_unexpected() がまだ呼び出されていない場合、unexpected() は terminate() を呼び出します。

関数 terminate() は、起動されると、set_terminate() に、一番最近に引数として供給された関数を呼び出します。 set_terminate() がまだ呼び出されていない場合、terminate() は abort() を呼び出し、これによってプログラムが終了します。

set_unexpected() および set_terminate() を使用して、unexpected() および terminate() によって呼び出される、ユーザー定義関数を登録することができます。 set_unexpected() と set_terminate() は、標準ヘッダー・ファイルに入っています。これらの各関数は、その戻りの型およびその引数の型として、戻りの型が void で引数なしの関数を指すポインターを持っています。引数として提供する関数を指すポインターは、対応する特殊な関数によって呼び出される関数になります。set_unexpected() への引数が unexpected() によって呼び出される関数になり、set_terminate() への引数が terminate() によって呼び出される関数になります。

set_unexpected() および set_terminate() は、前にそれぞれの特殊な関数 (unexpected() および terminate()) によって呼び出された関数を指すポインターを戻します。戻り値を保管することによって、後で元の特殊な関数を復元して、unexpected() と terminate() が、再び terminate() と abort() を呼び出すようにすることができます。

set_terminate() を使用して、ユーザー自身の関数を登録する場合、関数は、呼び出し元にはリターンせず、プログラムの実行を終了する必要があります。

例外処理関数を使用する例

次の例は、制御の流れと、例外処理で使用する特殊な関数を示しています。

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

class X { };
class Y { };
class A { };

// pfv type is pointer to function returning void
typedef void (*pfv)();

void my_terminate() {
  cout << "Call to my terminate" << endl;
  abort();
}

void my_unexpected() {
  cout << "Call to my_unexpected()" << endl;
  throw;
}

void f() throw(X,Y, bad_exception) {
  throw A();
}

void g() throw(X,Y) {
  throw A();
```

```
}

int main()
{
  pfv old_term = set_terminate(my_terminate);
  pfv old_unex = set_unexpected(my_unexpected);
  try {
    cout << "In first try block" << endl;
    f();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e1) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
    cout << "Caught some exception" << endl;
  }

  cout << endl;

  try {
    cout << "In second try block" << endl;
    g();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e2) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
    cout << "Caught some exception" << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In first try block
Call to my_unexpected()
Caught bad_exception

In second try block
Call to my_unexpected()
Call to my terminate
```

実行時に、このプログラムは次のように動作します。

1. set_terminate() を呼び出すと、old_term に、 set_terminate() が前に呼び出されたときに最後に set_terminate() に渡された関数のアドレスが割り当てられます。
2. set_unexpected() を呼び出すと、old_unex に、 set_unexpected() が前に呼び出されたときに、最後に set_unexpected() に渡された関数のアドレスが割り当てられます。
3. 最初の try ブロックの中では、関数 f() が呼び出されます。 f() が、予期しない例外をスローするので、unexpected() への呼び出しが行われます。次に、

unexpected() は、my_unexpected() を呼び出し、標準出力に対してメッセージを印刷します。関数 my_unexpected() は、型 A の例外を再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おらず、bad_exception が指定されているため、my_unexpected() がスローする例外は、型 bad_exception の例外に置換されます。

4. ハンドラー catch (bad_exception& e1) は、例外を処理できます。
5. 2 番目の try ブロックの中では、関数 g() が呼び出されます。 g() が、予期しない例外をスローするので、unexpected() への呼び出しが行われます。次に、unexpected() は、my_unexpected() を呼び出し、標準出力に対してメッセージを印刷します。関数 my_unexpected() は、型 A の例外を再スローしようとします。クラス A および bad_exception が、関数 g() の例外指定に指定されていないため、unexpected() は、terminate() を呼び出して、これが関数 my_terminate() を呼び出します。
6. my_terminate() は、メッセージを表示してから、プログラムを終了する abort() を呼び出します。

第 17 章 プリプロセッサー・ディレクティブ

プリプロセッシングは、コンパイル前の、テキストを処理するための初期フェーズです。プリプロセッサー・ディレクティブは、ソース・ファイル内で、先頭の空白以外の文字が # である行であり、この文字によりディレクティブとテキストの他の行を区別できます。各プリプロセッサー・ディレクティブにより、テキストが変更され、ディレクティブもコメントも含まれないテキストに変換されます。コンパイラーは、オプションで、プリプロセスされたテキストを、.i というサフィックスが付いたファイルに出力できます。プリプロセッシングは、常にコンパイルの初期フェーズで実行されます。既にテキストのプリプロセスが完了していても同じです。

プリプロセッサー・ディレクティブには、次のようなものがあります。

- 518 ページの『マクロ定義ディレクティブ』。このディレクティブは、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に置き換えます。
- 527 ページの『ファイル・インクルード・ディレクティブ』。このディレクティブは、現在のファイル内にファイルを組み込みます。
- 529 ページの『条件付きコンパイル・ディレクティブ』。このディレクティブは、現在のファイルのセクションを条件付きでコンパイルします。
- 534 ページの『メッセージ生成ディレクティブ』。このディレクティブは、診断メッセージの生成を制御します。
- IBM 537 ページの『アサーション・ディレクティブ (IBM 拡張)』。このディレクティブは、プログラムを実行するシステムの属性を指定します。
- 538 ページの『NULL ディレクティブ (#)』。このディレクティブは、何もアクションを行いません。
- 538 ページの『プラグマ・ディレクティブ』。このディレクティブは、コンパイラー固有の規則を、指定されたコードのセクションに適用します。

プリプロセッサー・ディレクティブは、# トークンで始まり、そのあとにプリプロセッサー・キーワードが続きます。# トークンは、空白文字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、空白文字で名前から分離することができます。

行の最後の文字が ¥ (円記号) 文字でない限り、プリプロセッサー・ディレクティブは改行文字で終了します。¥ 文字がプリプロセッサー行の最後の文字として現れると、プリプロセッサーは ¥ と改行文字を継続マーク文字として解釈します。プリプロセッサーは、¥ (およびそれに続く改行文字) を削除して、物理ソース行を継続する論理行に接合します。円記号と、行末文字または物理的なレコードの終わりとの間に、空白文字があっても構いません。ただし、通常、この空白文字は編集の際には見えません。

一部の #pragma ディレクティブを除いて、プリプロセッサー・ディレクティブはプログラム内の任意の場所に入れることができます。

マクロ定義ディレクティブ

マクロ定義ディレクティブには、次のディレクティブと演算子が含まれます。

- 『#define ディレクティブ』。マクロを定義します。
- 523 ページの『#undef ディレクティブ』。マクロ定義を除去します。

525 ページの『標準の事前定義マクロ名』で、ISO C 標準によって事前定義されたマクロを説明しています。

#define ディレクティブ

プリプロセッサー定義ディレクティブ は、これ以降のマクロの出現を、指定された置換トークンに置き換えるようプリプロセッサーに指示します。

#define ディレクティブは、次のものを指定することができます。

- 『オブジェクト類似マクロ』
- 519 ページの『関数類似マクロ』

マクロを定数および宣言済み定数に対してそれぞれ使用する場合の、いくつかの相違点を以下に示します。

- const オブジェクトは変数のスコープ規則に従いますが、 #define を使用して作成される定数はそうではありません。
- const オブジェクトと違って、マクロはインラインで展開されるので、マクロの値は、コンパイラーが使用する中間表現には含まれません。インライン展開をすると、デバッガーはマクロ値を使用できなくなります。
- C マクロは、ビット・フィールド長などのコンパイル時定数式の中で使用できますが、const オブジェクトは使用できません。
- C++ コンパイラーは、マクロ引数を含めて、マクロの型検査は行いません。

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。例えば、以下のオブジェクト類似の定義を使用すると、プリプロセッサーは、ID COUNT のこれ以降のすべてのインスタンスを、定数 1000 に置き換えます。

```
#define COUNT 1000
```

次のステートメント

```
int arry[COUNT];
```

が このマクロ定義の後の、同じコンパイル単位内にある場合、プリプロセッサーはステートメントを次のように変更します。

```
int arry[1000];
```

他の定義が ID COUNT を参照することができます。

```
#define MAX_COUNT COUNT + 100
```

プリプロセッサーは、MAX_COUNT のこれ以降の出現を COUNT + 100 で置き換えます。次に、プリプロセッサーは、これを 1000 + 100 で置き換えます。

マクロ展開によって部分的に構築された番号が作成された場合、プリプロセッサーは、その結果を単一の値であるとは見なしません。例えば、以下の結果は 10.2 という値にはならず、構文エラーになります。

```
#define a 10
doubl d = a.2
```

C++11 C++11 では、C コンパイラーおよび C++ コンパイラーに対して共通のプリプロセッサー・インターフェースを提供するために、C99 プリプロセッサーのオブジェクト類似マクロ用の診断が採用されています。マクロ定義内のオブジェクト類似マクロ名とその置換リストとの間に空白文字が存在しない場合、C++11 コンパイラーは警告メッセージを出します。詳しくは、540 ページの『C++11 に採用された C99 プリプロセッサー機能』を参照してください。 C++11

関数類似マクロ

関数類似マクロは、オブジェクト類似マクロより複雑であって、その定義は、仮パラメーターの名前をコンマで区切って、括弧で囲んで宣言します。空の正式パラメーター・リストは有効です。そのようなマクロを使用して、引数を取らない関数をシミュレートすることができます。 C99 は、可変個の引数を持つ関数類似マクロのサポートを追加しました。XL C++ は、C との互換性を保つための言語拡張機能として、C++11 および C++11 の一部として、個数が可変の引数を持つ関数類似マクロをサポートします。

関数類似マクロの定義

小括弧に囲まれたパラメーター・リストおよび置換トークンが後ろに続く ID。パラメーターを置換コード内に組み込みます。空白文字で、ID (マクロの名前) とパラメーター・リストの左括弧とを分離することはできません。コンマで各パラメーターを区切る必要があります。

移植性のため、1 つのマクロには、パラメーターが 31 を超えないようにする必要があります。パラメーター・リストは、仮パラメーターとして、省略符号 (...) で終了することができます。この場合には、置換リストに ID __VA_ARGS__ を使用できます。

関数類似マクロの呼び出し

小括弧に入れられた、コンマで区切られた引数のリストが後に続く ID。引数の数は、マクロ定義内のパラメーター・リストが省略符号で終了するのでなければ、定義内のパラメーターの数と一致しなければなりません。定義内のパラメーター・リストが省略符号で終了する場合、マクロ呼び出しでの引数の数は、定義内のパラメーターの数に一致するかそれを超えるはずです。この超過部分は、*後続引数* と呼ばれます。プリプロセッサーは、関数類似マクロの呼び出しを確認すると、引数の置換を行います。置換コード内のパラメーターは、対応する引数に置き換えられます。マクロ定義で後続引数が許可されている場合、その引数は間にコンマが挿入されて、その引数全体が単一の引数であるかのように、 ID __VA_ARGS__ を置き換えます。引数自体に含まれるマクロの呼び出しはすべて、引数が置換コード内の対応するパラメーターと置き換わる前に、完全に置き換えられます。

マクロ引数は空でもかまいません (ゼロ個のプリプロセス・トークンで構成されます)。次に例を示します。

```
#define SUM(a,b,c) a + b + c
SUM(1,,3)  /* No error message.
                1 is substituted for a, 3 is substituted for c. */
```

パラメーター・リストが省略符号で終了していない場合、マクロ呼び出しの中の引数の数は、対応するマクロ定義の中のパラメーターの数と同じでなければなりません。パラメーターの置換時、指定されたすべての引数 (区切り文字のコンマも含む) が置換された後に残っている引数は、変数引数と呼ばれる 1 つの引数にまとめられます。変数引数は、置換リストの中の ID __VA_ARGS__ のすべてのオカレンスを置き換えます。次の例は、このことを例示しています。

```
#define debug(...)   fprintf(stderr, __VA_ARGS__)

debug("flag");     /*   Becomes fprintf(stderr, "flag");   */
```

マクロ呼び出し引数リストにおけるコンマは、以下の場合には、分離文字として作用しません。

- 文字定数内にある。
- ストリング・リテラル内にある。
- 小括弧で囲まれている。

次の行は、a と b という 2 つのパラメーターと置換トークン (a + b) を持つものとしてマクロ SUM を定義します。

```
#define SUM(a,b) (a + b)
```

この定義により、プリプロセッサーは以下のステートメントを変更することになります (そのステートメントが前の定義の後に現れる場合)。

```
c = SUM(x,y);
c = d * SUM(x,y);
```

プリプロセッサーの出力においては、これらのステートメントは次のように表示されます。

```
c = (x + y);
c = d * (x + y);
```

置換テキストが正しく評価されるようにするためには、小括弧を使用してください。例えば、以下の定義では、

```
#define SQR(c)  ((c) * (c))
```

定義内の各パラメーター c の周りに小括弧を必要とします。

```
y = SQR(a + b);
```

プリプロセッサーはこのステートメントを次のように展開します。

```
y = ((a + b) * (a + b));
```

定義内に小括弧がないと、意図された評価の順序は保持されず、プリプロセッサーの出力は次のようになります。

```
y = (a + b * a + b);
```

演算子および ## 演算子の引数は、関数類似マクロのパラメーターの置換の*前*に変換されます。

プリプロセッサー ID は、いったん定義されると、言語のスコープ規則に関係なく、定義されたままとなります。マクロ定義のスコープは定義から始まり、対応する #undef ディレクティブに遭遇するまで終了しません。対応する #undef ディレクティブがない場合、そのマクロ定義のスコープは、変換単位の終わりまで続きます。

再帰マクロは、完全には展開されません。例えば、以下の定義では、

```
#define x(a,b) x(a+1,b+1) + 4
```

は、

```
x(20,10)
```

を、以下のように展開します。

```
x(20+1,10+1) + 4
```

マクロ x を、それ自体の中で繰り返し展開しようとするよりも、上述の展開を行います。マクロ x が展開された後で、そのマクロは、関数 x() の呼び出しとなります。

置換トークンを指定するのに、定義は必須ではありません。以下の定義は、現在のファイル内のこれ以降の行から、トークン debug のすべてのインスタンスを除去します。

```
#define debug
```

2 番目のプリプロセッサー #define ディレクティブを用いて、定義済みの ID またはマクロの定義を変更することができます。ただし、2 番目のプリプロセッサー #define ディレクティブの前に、プリプロセッサー #undef ディレクティブがある場合に限ります。#undef ディレクティブは、最初の定義を無効にして、同じ ID を再定義で使用できるようにします。

プログラムのテキスト内で、プリプロセッサーは、マクロ定義、マクロの未定義化、またはマクロ呼び出しについて、コメント、文字定数、またはストリング定数をスキャンすることはありません。

以下のプログラム例には、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呼び出しが含まれています。

```
/**This example illustrates #define directives.**/

void printf(const char*, ...);
#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);
}
```

プリプロセスされた後、このプログラムは、以下と等価のコードによって置き換えられます。

```
void printf(const char*, ...);

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
```

IBM

可変数引数マクロ拡張機能

可変数引数マクロ拡張機能は、個数が可変の引数を持つマクロに関連した、C99 および標準 C++ の 2 つの拡張機能です。これらの拡張の 1 つは、変数引数 ID を __VA_ARGS__ からユーザー定義の ID に名前変更するためのメカニズムです。もう 1 つの拡張機能は、変数引数が指定されていない場合に、可変数引数マクロ内のダングリング・コンマを除去するための手段を提供します。どちらの拡張機能も、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。

以下の例は、__VA_ARGS__ の代わりに ID を使用する方法を示しています。マクロ debug の最初の定義は、__VA_ARGS__ の通常の使用方法の例を示しています。 2 番目の定義は、__VA_ARGS__ の代わりの ID として args を使用する方法を示しています。

```
#define debug1(format, ...)  printf(format, ## __VA_ARGS__)
#define debug2(format, args ...)  printf(format, ## args)
```

呼び出し	マクロ展開の結果
`debug1("Hello %s/n", "World");`	`printf("Hello %s/n", "World");`
`debug2("Hello %s/n", "World");`	`printf("Hello %s/n", "World");`

プリプロセッサーは、関数マクロへの変数引数が省略されるかまたは空であり、関数マクロ定義内の変数引数 ID の前に ## を伴うコンマがある場合に、末尾のコンマを除去します。

IBM

C++11 C++11 では、C コンパイラーおよび C++ コンパイラーに対して共通のプリプロセッサー・インターフェースを提供するために、C99 プリプロセッサーからの可変数引数マクロ機能、および空のマクロ引数に関する変更が採用されています。C++11 では可変数引数マクロおよび空のマクロ引数がサポートされています。

詳しくは、 540 ページの『C++11 に採用された C99 プリプロセッサー機能』を参照してください。

関連資料:

105 ページの『const 型修飾子』

224 ページの『演算子優先順位と結合順序』

170 ページの『括弧で囲んだ式 ( )』

#undef ディレクティブ

プリプロセッサーの undef ディレクティブ により、プリプロセッサーはプリプロセッサー定義のスコープを終わらせます。

#undef ディレクティブの構文

identifier が現在マクロとして定義されていなければ、#undef は無視されます。

以下のディレクティブは BUFFER および SQR を定義します。

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

以下のディレクティブはその定義を無効にします。

```
#undef BUFFER
#undef SQR
```

これらの #undef ディレクティブの後に続いて ID BUFFER および SQR が現れても、それらは、いかなる置換トークンにも置き換えられません。 #undef ディレクティブによってマクロの定義が除去されてしまえば、新しい #define ディレクティブでその ID を使用することができます。

演算子

(単一番号記号) 演算子は、関数類似マクロのパラメーターを文字ストリング・リテラルに変換します。例えば、以下のディレクティブを使用してマクロ ABC が定義される場合、

```
#define ABC(x)   #x
```

これ以降のマクロ ABC の呼び出しはすべて、ABC に渡された引数を含む文字ストリング・リテラルに展開されます。次に例を示します。

呼び出し	マクロ展開の結果
ABC(1)	"1"
ABC(Hello there)	"Hello there"

演算子を、NULL ディレクティブと混同してはなりません。

演算子は、下記の規則に従って、関数類似マクロ定義で使用してください。

- 関数類似マクロの中の # 演算子に続くパラメーターは、マクロに渡された引数を含む文字ストリング・リテラルに変換されます。

- プリプロセッサーは、マクロに渡された引数の前または後ろにある空白文字を削除します。
- マクロに渡された引数内に組み込まれた複数の空白文字は、単一のスペース文字に置き換えられます。
- マクロに渡された引数にストリング・リテラルがある場合、およびそのリテラル内に ¥ (円記号) 文字がある場合には、マクロ展開時に、元の ¥ の前に 2 番目の ¥ 文字が挿入されます。
- マクロに渡された引数に " (二重引用符) 文字がある場合、マクロ展開時に、" の前に ¥ 文字が挿入されます。
- 引数のストリング・リテラルへの変換は、その引数でマクロが展開される前に行われます。
- マクロ定義の置換リスト内に複数の ## 演算子または # 演算子がある場合、その演算子の評価の順序は定義されていません。
- マクロ展開の結果が有効な文字ストリング・リテラルでない場合、その動作は未定義です。

以下の例は、# 演算子の使用法を示したものです。

```
#define STR(x)        #x
#define XSTR(x)       STR(x)
#define ONE           1
```

呼び出し	マクロ展開の結果
STR(¥n "¥n" '¥n')	"¥n ¥"¥¥n¥" '¥¥n'"
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"¥"hello¥""

関連資料:

538 ページの『NULL ディレクティブ (#)』

演算子

(二重番号記号) 演算子は、マクロ定義に含まれるマクロの呼び出し (テキストまたは引数、あるいはその両方) における 2 つのトークンを連結します。

以下のディレクティブを使用して、マクロ XY が定義された場合、

```
#define XY(x,y)    x##y
```

x に対する引数の最後のトークンは、y に対する引数の最初のトークンと連結されます。

演算子は、以下の規則に従って使用します。

- ## 演算子を、マクロ定義の置換リスト内の最初の項目または最後の項目にすることはできません。
- ## 演算子の前にある項目の最後のトークンは、 ## 演算子の後ろにある項目の最初のトークンに連結されます。
- 連結は、引数の中のマクロのいずれかが展開される前に行われます。

- 連結の結果が有効なマクロ名になった場合には、たとえその名前が、通常はその中では使用できないコンテキストの中に現れたとしても、その名前を、その後の置換に使用することができます。
- マクロ定義の置換リスト内に複数の ## 演算子、または # 演算子、またはその両方がある場合、その演算子の評価の順序は定義されていません。

以下の例は、## 演算子の使用法を示したものです。

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

呼び出し	マクロ展開の結果
ArgArg(lady, bug)	ladybug
ArgText(con)	conTEXT
TextArg(book)	TEXTbook
TextText	TEXTtext
ArgArg(Jitter, bug)	3

関連資料:

518 ページの『#define ディレクティブ』

標準の事前定義マクロ名

C および C++ は両方とも、ISO C 言語標準で指定されている次の事前定義マクロの名前を提供しています。 __FILE__ と __LINE__ を除いて、事前定義されたマクロの値は、変換単位全体にわたって定数のままです。事前定義マクロ名は一般に、先頭と末尾に 2 つの下線文字が付いています。

__DATE__

ソース・ファイルがプリプロセスされた日付が入っている文字ストリング・リテラル。

__DATE__ の値は、入力がプリプロセスされた日付けに従って変更されます。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。

__FILE__ の値は、ソース・プログラムの一部である組み込みファイルがプリプロセスされると変更されます。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 **classic** に設定すると、このマクロは定義されません。 (未定義のマクロは、#if ステートメントで使用されると、あたかも整数値 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__ (C のみ)
この C99 マクロの値は 1 であって、C コンパイラーはホスト型インプリメンテーションであることを示します。このマクロは、__STDC__ も定義されている場合にのみ定義されることに注意してください。

__STDC_VERSION__ (C のみ)
long int 型の整数定数: C89 言語レベルでは 199409L、C99 言語レベルでは 199901L。このマクロは、__STDC__ も定義されている場合にのみ定義されることに注意してください。

__TIME__
ソース・ファイルがプリプロセスされた時刻が入っている文字ストリング・リテラル。

__TIME__ の値は、ソース・プログラムの一部である組み込みファイルがプリプロセスされると変更されます。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++ のみ)
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。

関連資料:

535 ページの『#line ディレクティブ』

オブジェクト類似マクロ

ファイル・インクルード・ディレクティブ

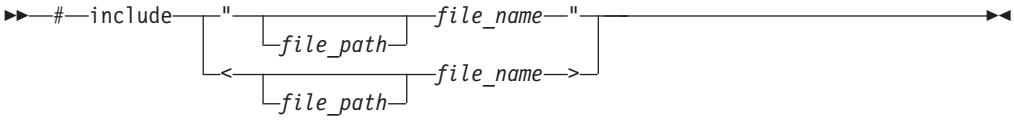
ファイル・インクルード・ディレクティブは次のもので構成されます。

- 『#include ディレクティブ』。別のソース・ファイルからテキストを挿入します。
- IBM 528 ページの『#include_next ディレクティブ (IBM 拡張)』。コンパイラーがインクルード・ファイルの検索時に、検索パスからインクルード・ファイルのディレクトリーを除外するようにします。

#include ディレクティブ

プリプロセッサーの include ディレクティブ を使用すると、プリプロセッサーは、そのディレクティブを指定されたファイルの内容に置き換えます。

#include ディレクティブの構文



例えば次のように、*file_name* が二重引用符で囲まれている場合、

```
#include "payroll.h"
```

ユーザー定義ファイルとして扱われ、ヘッダー・ファイルまたはソース・ファイルを表す場合があります。

例えば次のように、*file_name* が不等号括弧で囲まれている場合、

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

システム定義のファイルとして扱われ、必ずヘッダー・ファイルを表します。

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。

file_path は、絶対パスでも相対パスでもかまいません。二重引用符を使用する場合、*file_path* が相対パスであるかまたは指定されていなければ、プリプロセッサーはインクルードするファイルのディレクトリーを、インクルードされるファイルを検索するためのパスのリストに追加します。不等号括弧を使用する場合、*file_path* が相対パスであるか指定されていなければ、プリプロセッサーはインクルードするファイルのディレクトリーを、インクルードされるファイルを検索するためのパスのリストに追加しません。

プリプロセッサーは #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
```

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
extern int hour;
extern int min;
extern 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ー・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。

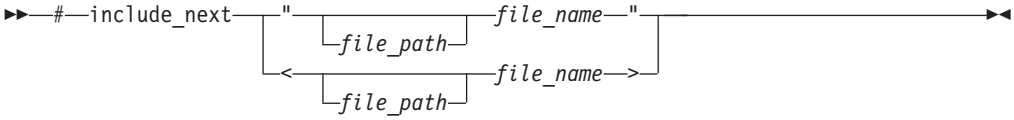
```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 *   #include <stdio.h>
 */

#include C_IO_HEADER
```

C++11 C++11 では、C および C++ コンパイラーに対して共通のプリプロセッサー・インターフェースを提供するために、C99 プリプロセッサーのヘッダー名およびインクルード・ファイル名に対し変更が採用されています。 C++11 では、#include ディレクティブ内のヘッダー・ファイル名の先頭文字に数字を使用しないでください。詳しくは、 540 ページの『C++11 に採用された C99 プリプロセッサー機能』を参照してください。

#include_next ディレクティブ (IBM 拡張)

プリプロセッサー・ディレクティブ #include_next は #include ディレクティブのように動作しますが、インクルードするファイルのディレクトリーを、指定されたファイルを検索するためのリストから、特に除外するところが異なります。インクルードするファイルのディレクトリー以下にあるすべての検索パスは、インクルードされるファイルを検索するためのパスのリストから除外されます。これにより、ファイルの複数バージョンを、同じ名前でアプリケーションの異なる部分に組み込むことができます。または、1 つのヘッダー・ファイルを別のヘッダー・ファイルに、同じ名前 (それ自身を再起的に組み込むヘッダーは付いていない) で組み込むことができます。異なるファイル・バージョンが異なるディレクトリーに保管される場合に、このディレクティブで、ファイル名を指定するために絶対パスを使用する必要なしに、ファイルの各バージョンにアクセスすることができるようになります。

#include_next ディレクティブの構文



このディレクティブはヘッダー・ファイルのみで使用する必要があり、*file_name* によって指定されたファイルがヘッダー・ファイルでなければなりません。ファイル名を囲むために使用する、二重引用符と不等号括弧との間に違いはありません。

検索パスが #include_next ディレクティブでどのように解決されるかという例として、ファイル t.h の 2 つのバージョンがあると想定します。最初のファイルは、ソース・ファイル t.c に組み込まれ、サブディレクトリー path1 にあります。2 番目のファイルは最初のファイルに組み込まれ、サブディレクトリー path2 にあります。 t.c のコンパイル時に、両方のディレクトリーがインクルード・ファイル検索パスとして指定されます。

```
/* t.c  */

#include "t.h"

int main()
{
printf(", ret_val);
}

/* t.h in path1 */

#include_next "t.h"

int ret_val = RET;

/* t.h in path2 */

#define RET 55;
```

#include_next ディレクティブは、プリプロセッサーに、 path1 ディレクトリーをスキップし、path2 ディレクトリーからインクルード・ファイルの検索を開始するよう指示します。このディレクティブにより、t.h の異なる 2 つのバージョンを使用でき、この結果 t.h が繰り返しインクルードされることがなくなります。

条件付きコンパイル・ディレクティブ

プリプロセッサーの条件付きコンパイル・ディレクティブ を使用すると、プリプロセッサーは、ソース・コードのコンパイルの一部を条件付きで抑止します。これらのディレクティブは、定数式または ID をテストして、プリプロセッサーがコンパイラーに渡すべきトークン、およびプリプロセス時に迂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。

- 531 ページの『#if ディレクティブおよび #elif ディレクティブ』。定数式の結果に基づいて、ソース・コードの部分を条件により組み込むか、抑止します。
- 532 ページの『#ifdef ディレクティブ』。マクロ名が定義されている場合に、ソース・テキストを条件により組み込みます。
- 532 ページの『#ifndef ディレクティブ』。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組み込みます。

- 533 ページの『#else ディレクティブ』。直前の #if、#ifdef、#ifndef、または #elif の検査が失敗した場合に、条件によりソース・テキストを組み込みます。
- 533 ページの『#endif ディレクティブ』。条件付きテキストを終了します。

プリプロセッサーの条件付きコンパイル・ディレクティブは、下記のいくつかの行に及びます。

- 条件指定行 (#if、#ifdef、または #ifndef で始まる)
- 条件の評価がゼロ以外の値になった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- #elif 行 (オプション)
- 条件の評価がゼロ以外の値になった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- #else 行 (オプション)
- 条件の評価がゼロになった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- プリプロセッサーの #endif ディレクティブ

#if、#ifdef、および #ifndef の各ディレクティブのそれぞれに対して、ゼロ個以上の #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・ディレクティブはネストさせることができます。 #else が存在する場合は、必須の #endif が指定されるために、あいまいさが残ることなく、その #endif と対であるものと見なされます。

```
#ifdef MACNAME
                /*  tokens added if MACNAME is defined */
#   if TEST <=10
                /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                /* tokens added if MACNAME is defined and TEST >  10 */
#   endif
#else
                /*  tokens added if MACNAME is not defined */
#endif
```

各ディレクティブは、その直後のブロックを制御します。ブロックは、ディレクティブの後の行から始まって、同じネスト・レベルにある次の条件付きコンパイル・ディレクティブで終了する、すべてのトークンで構成されます。

各ディレクティブは、検出された順序で処理されます。式の評価がゼロの場合、ディレクティブの後に続くブロックは無視されます。

プリプロセッサー・ディレクティブの後に続くブロックを無視することになっているとき、条件付きネスト・レベルが判別できるように、そのブロック内のプリプロセッサー・ディレクティブを識別するだけのために、トークンが検査されます。ディレクティブの名前以外のトークンは、すべて無視されます。

式がゼロ以外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 ディレクティブおよび #elif ディレクティブ

#if ディレクティブには、条件付きで、プリプロセッシング対象のテキストが含まれています。#if ディレクティブに続く条件がゼロ以外の値に評価された場合、関連付けられた #endif の直前までのテキストがプリプロセッシングの入力として含まれます。

#elif (else-if の短縮形) ディレクティブを使用する場合は、#if ディレクティブの影響下にあるテキストのセクション内に含める必要があります。このディレクティブは、オプションでそのディレクティブの直後の条件の評価結果に基づいて、テキストのセクションを含めることができます。#elif ディレクティブは、#if の元の条件が false と評価され、元の #if の影響下にある前の #elif ディレクティブに関連付けられたすべての条件も false と評価された場合にのみ、条件を評価します。

#if ディレクティブおよび #elif ディレクティブの構文

defined 演算子のオペランドであるマクロを除くすべてのマクロは展開されます。defined 演算子のあらゆる使用が処理され、残りのすべてのキーワードと ID が、トークン 0 に置換されます。 C++ ただし、true および false を除きます。 C++

マクロの展開により defined トークンが生成された場合、その動作は未定義です。

注:

- キャストは実行できません。例えば、以下のコードは C コンパイラーと C++ コンパイラーによって正常にコンパイルできます。

```
#if static_cast<int>(1)
#error Unexpected
#endif

int main() {
}
```

- long int 型を使用して演算を実行します。 C++11 C++11 では、long long int 型を使用して演算を実行します。詳しくは、 540 ページの『C++11 に採用された C99 プリプロセッサー機能』 を参照してください。 C++11
- 定義済みマクロを *constant_expression* に入れることはできます。
- *constant_expression* には、単項演算子 defined を入れることができます。この演算子は、プリプロセッサー・キーワードの #if または #elif を用いた場合にのみ使用できます。以下の式の評価は、プリプロセッサーに *identifier (ID)* が定義されている場合は、1 に、それ以外の場合は、0 になります。

```
defined identifier
defined(identifier)
```

次に例を示します。

```
#if defined(TEST1) || defined(TEST2)
```

- *constant_expression* は整数定数式でなければなりません。

マクロが定義されていない場合、0 (ゼロ) の値がそれに割り当てられます。以下の例では、TEST はマクロ識別子です。

```
#include <stdio.h>
int main()
{
   #if TEST != 0    // No error even when TEST is not defined.
       printf("Macro TEST is defined to a non-zero value.");
   #endif
}
```

#ifdef ディレクティブ

#ifdef ディレクティブは、マクロ定義の存在を検査します。

指定された ID がマクロとして定義されている場合、条件の直後にあるコードの行がコンパイラーに渡されます。条件付きコンパイル・ディレクティブを終了させるには、#endif ディレクティブを使用する必要があります。

#ifdef ディレクティブの構文

以下の例は、プリプロセッサーに対して EXTENDED が定義されている場合に、MAX_LEN を 75 として定義します。定義されていない場合には、MAX_LEN を 50 として定義します。

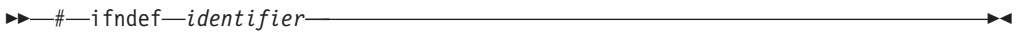
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef ディレクティブ

#ifndef ディレクティブは、マクロが定義されていないかどうかをチェックします。

指定された ID がマクロとして定義されていない場合、条件の直後にあるコードの行がコンパイラーに渡されます。

#ifndef ディレクティブの構文



ID は、#ifndef キーワードの後に続いていなければなりません。以下の例は、プリプロセッサーに対して EXTENDED が定義されていない場合に、MAX_LEN を 50 として定義します。定義されていない場合、MAX_LEN を 75 として定義します。

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

#else ディレクティブ

#if、#ifdef、または #ifndef ディレクティブで指定された条件が 0 に評価され、条件付きコンパイル・ディレクティブが、プリプロセッサー #else ディレクティブを含んでいる場合、プリプロセッサー #else ディレクティブおよびプリプロセッサー #endif ディレクティブとの間にあるコードの行が、プリプロセッサーによって選択され、コンパイラーに渡されます。

#else ディレクティブの構文

#endif ディレクティブ

プリプロセッサー・ディレクティブの #endif は、条件付きコンパイル・ディレクティブを終了します。

#endif ディレクティブの構文

#endif および #else の拡張

C/C++ 言語標準では、#endif または #else の後に追加のテキストを入れることはできません。IBM XL C および XL C/C++ コンパイラーは、標準に準拠しています。#endif または #else の後の追加テキストが許可されるコンパイラーからコードを移植する場合は、オプション **-qlanglvl=textafterendif** を指定すると、出される警告メッセージを抑制できます。

使用法の 1 つは、対応する #if または #ifdef がテストしている内容についてのコメントを入れることです。次に例を示します。

```
#ifdef MY_MACRO
...
#else MY_MACRO not defined
...
#endif MY_MACRO
```

ここで、この標準からの逸脱に関してコンパイラーをサイレントにしたい場合は、オプション **-qlanglvl=textafterendif** を指定してメッセージを抑制し、その一方で、他のコンテキスト (例えば、#undef の後に追加テキストがある場合など) ではメッセージの出力を許可することができます。

サブオプション **textafterendif** は、サポートされるどの言語レベルでも指定できます。ほとんどの場合、このサブオプションのデフォルトは **-qlanglvl=notextafterendif** であり、これは、#else または #endif の後に無関係なテキストがある場合はメッセージが出されることを示します。ただし、C コンパイラーで言語レベルが「classic」の場合だけは例外です。この場合、この言語レベルでは既に #else または #endif の後の追加テキストが許可されていて、メッセージは生成されないため、このサブオプションのデフォルトは **-qlanglvl=textafterendif** となります。

条件付きコンパイル・ディレクティブの例

以下の例は、プリプロセッサーの条件付きコンパイル・ディレクティブをどのようにネストできるかを示しています。

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

以下のプログラムには、プリプロセッサーの条件付きコンパイル・ディレクティブが含まれています。

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d¥n", i);
   printf("array[i] = %d¥n",
   array[i]);
#endif

   }
   return(0);
}
```

メッセージ生成ディレクティブ

メッセージ生成ディレクティブには、次のようなものがあります。

- 535 ページの『#error ディレクティブ』。コンパイル時のエラー・メッセージ用のテキストを定義します。
- 535 ページの『#warning ディレクティブ (IBM 拡張)』。コンパイル時の警告メッセージ用のテキストを定義します。
- 535 ページの『#line ディレクティブ』。コンパイラー・メッセージの行番号を提示します。

関連資料:

529 ページの『条件付きコンパイル・ディレクティブ』

#error ディレクティブ

プリプロセッサーの error ディレクティブ を使用すると、プリプロセッサーはエラー・メッセージを生成して、コンパイルを失敗させます。

#error ディレクティブの構文

引数 *preprocessor_token* は、マクロ展開されません。

#error ディレクティブは、多くの場合、コンパイル時の安全検査として、#if–#elif–#else 構成の #else の部分で使用されます。例えば、#error ディレクティブをソース・ファイルで使用すると、プログラムのバイパスすべき部分に到達したら、コードが生成されないようにすることできます。

例えば、次のようなディレクティブがあるとします。

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

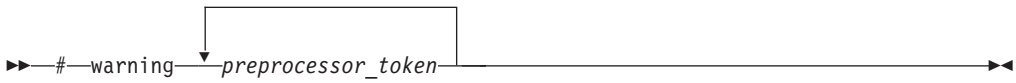
これは次のエラー・メッセージを生成します。

```
BUFFER_SIZE is too small.
```

#warning ディレクティブ (IBM 拡張)

プリプロセッサーの warning ディレクティブ を使用すると、プリプロセッサーは警告メッセージを生成します。ただし、コンパイルは続行します。#warning に対する引数は、マクロ展開されません。

#warning ディレクティブの構文

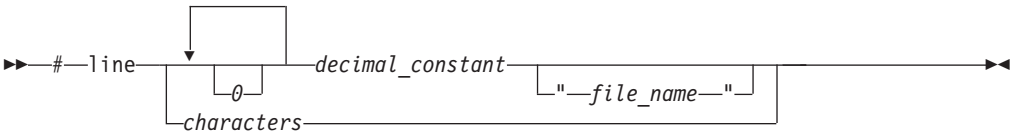


プリプロセッサー #warning ディレクティブは、GNU C で開発されたプログラムの処理を容易にするために提供される言語拡張機能です。IBM インプリメンテーションでは、複数の空白文字が保持されています。

#line ディレクティブ

プリプロセッサーの行制御ディレクティブ は、コンパイラー・メッセージに対して行番号を提供します。このディレクティブにより、コンパイラーは、次のソース行の行番号を指定された番号として表示します。

#line ディレクティブの構文



コンパイラーがプリプロセスされたソース内の行番号への参照をわかりやすく行えるようにするため、プリプロセッサーは必要な個所に (例えば、含まれているテキストの始めまたはテキストの終わりの後に)、#line ディレクティブを挿入します。

二重引用符で囲まれたファイル名の指定を行番号の後に続けることができます。ファイル名を指定すると、コンパイラーは指定されたファイルの一部として次の行を表示します。ファイル名を指定しないと、コンパイラーは現行ソース・ファイルの一部として次の行を表示します。

すべての C および C++ インプリメンテーションにおいて、#line ディレクティブのトークン・シーケンスは、マクロ置き換えをすることがあります。マクロ置き換え後に結果として得られる文字シーケンスは、10 進定数 (オプションで、二重引用符で囲まれたファイル名が後に続く) で構成されます。

#line 制御ディレクティブを使用して、コンパイラーにもっと分かりやすいエラー・メッセージを提供させることができます。以下のプログラム例は、#line 制御ディレクティブを使用して、認識しやすい行番号を各関数に提供します。

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d¥n",__LINE__);
}

#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d¥n",__LINE__);
}
```

このプログラムの出力は次のようになります。

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

C++11 C++11 では、C および C++ コンパイラーに対して共通のプリプロセッサー・インターフェースを提供するために、C99 プリプロセッサーから #line ディレクティブの制限が増大されて採用されています。#line <integer> プリプロセッサー・ディレクティブの上限が、C99 プリプロセッサーに適合する C++ プリプロ

セッサーでは、32,767 から 2,147,483,647 に増えています。詳しくは、 540 ページの『C++11 に採用された C99 プリプロセッサー機能』を参照してください。

関連資料:


「XL C/C++ コンパイラー・リファレンス」の『__C99_MAX_LINE_NUMBER』を参照

アサーション・ディレクティブ (IBM 拡張)

アサーション・ディレクティブ は、コンパイル済みプログラムが実行されるコンピューターまたはシステムを定義するために使用される、マクロ定義の代わりのものです。アサーションは、通常、事前定義されていますが、#assert プリプロセッサー・ディレクティブで定義することもできます。

#assert ディレクティブの構文

predicate は、定義しようとしているアサーション・エンティティーを表します。*answer* は、このアサーションに割り当てる値を表します。同じ述部 (predicate) と異なる応答 (answer) を使用して、複数のアサーションを作成することができます。特定の述部に対するすべての応答は、同時に真です。例えば、次のディレクティブは、フォントのプロパティーに関するアサーションを作成します。

```
#assert font(arial)
#assert font(blue)
```

アサーションが定義されると、現在のシステムをテストするために、そのアサーション述部を条件付きディレクティブの中で使用することができます。以下のディレクティブは、font について、arial が assert されるか、blue が assert されるかをテストします。

```
#if #font(arial) || #font(blue)
```

条件付きディレクティブの中で応答を省略することにより、述部について、いずれかの応答が assert されるかをテストすることができます。

```
#if #font
```

アサーションは、#unassert ディレクティブで取り消すことができます。#assert ディレクティブと同じ構文を使用すると、そのディレクティブは、指定された応答のみを取り消します。例えば、次のディレクティブは、font 述部に対して arial 応答を取り消します。

```
#unassert font(arial)
```

#unassert ディレクティブから応答を省略すると、述部全体が取り消されます。次のディレクティブは、 font 述部を完全に取り消します。

```
#unassert font
```

関連資料:

529 ページの『条件付きコンパイル・ディレクティブ』

事前定義されたアサーション

Linux プラットフォームに対しては、以下のアサーションが事前定義されています。

表 39. Linux 用に事前定義されたアサーション

#machine(powerpc)	#system(unix)	#cpu(powerpc)
#machine(bigendian)	#system(posix)	

NULL ディレクティブ (#)

NULL 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。

NULL ディレクティブを、# 演算子や、プリプロセッサー・ディレクティブの最初の文字と混同しないようにしてください。

以下の例では、MINVAL が定義済みマクロ名である場合、アクションを行いません。MINVAL が定義済み ID ではない場合は、1 に定義します。

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

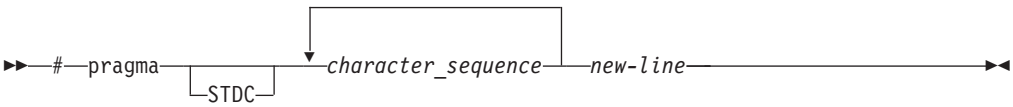
関連資料:

523 ページの『# 演算子』

プラグマ・ディレクティブ

プラグマ は、コンパイラーに対するインプリメンテーション定義の命令です。一般的な形式は次のとおりです。

#pragma ディレクティブの構文



character_sequence は、文字が書かれている場合、特定のコンパイラー命令と引数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き換えは行われません。 *new-line* 文字は、プラグマ・ディレクティブを終了させなければなりません。

プラグマの *character_sequence* は、マクロ置き換えを受けることがあります。例えば、次のような場合です。

```
#define XX_ISO_DATA isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

注: _Pragma 演算子の構文を使用してプラグマ・ディレクティブを指定することもできます。詳しくは、『_Pragma プリプロセッシング演算子』を参照してください。

1 つのプラグマ・ディレクティブで、複数のプラグマ構成を指定することができます。コンパイラーは、認識されないプラグマを無視します。

標準 C プラグマを 『標準のプラグマ (C のみ)』で説明しています。IBM XL C/C++ で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*」の『汎用プラグマ』に説明があります。

_Pragma プリプロセッシング演算子

単項演算子 _Pragma を使用すると、プラグマ・ディレクティブにプリプロセッサー・マクロを含むことができます。

_Pragma 演算子の構文

string_literal には接頭部 L を付けることができ、そうすると、それはワイド・ストリング・リテラルになります。

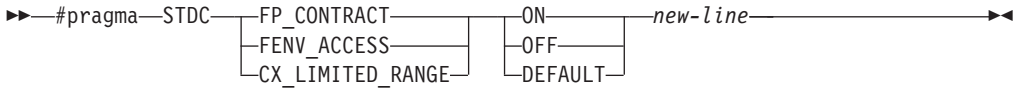
ストリング・リテラルは、ストリングが解除され、トークンになります。トークンの結果のシーケンスは、プラグマ・ディレクティブに現れたかのように処理されます。例えば、以下の 2 つのステートメントは等価です。

```
_Pragma ( "pack(full)" )
#pragma pack(full)
```

C++11 C++11 では、C コンパイラーおよび C++ コンパイラーに対して共通のプリプロセッサー・インターフェースを提供するために、C99 プリプロセッサーの *_Pragma 演算子* 機能が採用されています。_Pragma 演算子は、#pragma ディレクティブを指定する代替方式です。詳しくは、540 ページの『C++11 に採用された C99 プリプロセッサー機能』を参照してください。

標準のプラグマ (C のみ)

標準のプラグマ は、C 標準が構文およびセマンティクスを定義し、マクロ置き換えが行われないプラグマ・プリプロセッサー・ディレクティブです。標準のプラグマは、以下のいずれかでなければなりません。



C FP_CONTRACT および FENV_ACCESS プラグマは認識され、無視されます。

CX_LIMITED_RANGE については以下で説明します。

pragma STDC CX_LIMITED_RANGE

複素数の乗算、割り算、および絶対値用の通常の数学公式は、無限大の処理および不適切なオーバーフローやアンダーフローのため、問題があります。通常の公式は以下のとおりです。

$(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$

$(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$

$| x + iy | = \mathrm{sqrt}(x^2 + y^2)$

デフォルトで、コンパイラーはこれらの計算を実行するために、少々複雑ですが数学的にはより安全なアルゴリズムを使用します。通常の数学公式が安全だと判断する場合は、 STDC CX_LIMITED_RANGE プラグマを使用して、状態が「on」のときにこれらの公式が許容できるものであるとコンパイラーに対して知らせることができます。そうすることによって、コンパイラーは、これらの計算のための高速なコードを生成することができます。状態が「off」の場合は、コンパイラーは引き続き、より安全なアルゴリズムを使用します。このプラグマのインプリメンテーションについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」内の**#pragma STDC cx_limited_range**を参照してください。

C++11 に採用された C99 プリプロセッサー機能

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

C++11 標準では、C および C++ コンパイラーに対して共通プリプロセッサー・インターフェースを提供するために、複数の C99 プリプロセッサー機能が採用されています。これにより、C ソース・ファイルを C++ コンパイラーに簡単に移植でき、旧式の C プリプロセッサーと C++ プリプロセッサーの間に存在するいくつかのわずかな意味の違いを除去できるため、プリプロセッサーの互換性の問題を解決し、プリプロセッサーが異なる動作を行うことを回避できます。

C++11 では、以下の C99 プリプロセッサー機能が採用されています。

- 拡張整数型を使用したプリプロセッサー演算
- 混合ストリング・リテラルの連結
- ヘッダー・ファイルおよびインクルード名の診断
- #line ディレクティブの制限の引き上げ
- オブジェクト類似マクロ定義の診断
- _Pragma 演算子
- 可変数引数マクロおよび空のマクロ引数

- 事前定義マクロ

拡張整数型を使用したプリプロセッサー演算

C89、C++98、および C++03 プリプロセッサーでは、int 型または unsigned int 型を持つ整数リテラルは、long または unsigned long に拡張されます。ただし、C99 および C++11 プリプロセッサーでは、すべての signed 整数型および unsigned 整数型 (文字型を含む) は、XL C/C++ の通常の環境の下では long long または unsigned long long に拡張されます。

この機能を使用可能にして、**-qnolonglong** および **-qlanglvl=noc99longlong** の両方が **-q32** モードまたは **-q64** モードのいずれかで設定されている場合、プリプロセッサーはプリプロセッサー制御式のすべての整数リテラルおよび文字リテラルに対して、依然として long long または unsigned long long の表記を使用します。

以下の例は、wchar_t の基礎となる型が unsigned short であるLinux プラットフォームで有効です。

```
#if u'0' - u'1' < 0
#error non-C++11 preprocessor arithmetic
#else
#error C++11 preprocessor arithmetic
#endif
```

以下の例は、long long サポートが **-q32** モードで使用可能になっているケースを示します。この機能を使用すると、非 C++11 プリプロセッサーと C++11 プリプロセッサーの間で異なる組み込み分岐が選択されます。

```
#if ~0ull == 0u + ~0u
#error C++11 preprocessor arithmetic! 0u has the same representation as 0ull,¥
       hence ~0ull == 0u + ~0u
#else
#error non-C++11 preprocessor arithmetic. 0ul does not have the same ¥
       representation as 0ull, hence ~0ull != 0u + ~0u
#endif
```

この機能を無効にして **-qwarn0x** を設定すると、C++11 プリプロセッサーは、#if ディレクティブおよび #elif ディレクティブの制御式を評価し、評価結果を非 C++11 プリプロセッサーの結果と比較します。結果が異なる場合、コンパイラーはプリプロセッサーの制御式の評価が、C++11 と非 C++11 の言語レベル間で異なることを警告します。

混合ストリング・リテラルの連結

通常のストリングをワイド・ストリング・リテラルと連結できます。以下に例を示します。

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

int main()
{
wprintf(L"Guess what? %ls¥n", "I can now concate" L"nate regular strings¥
        and wide strings!");
printf("Guess what? %ls¥n", L"I can now concate" "nate strings¥
        this way too!");
}
```

この例を実行すると、以下を出力します。

```
Guess what? I can now concatenate regular strings and wide strings!
Guess what? I can now concatenate strings this way too!
```

ヘッダー・ファイルおよびインクルード名の診断

この機能が使用可能であるときに、#include ディレクティブ内のヘッダー・ファイル名の先頭文字が数字である場合、コンパイラーは警告メッセージを出します。次の例を検討してみます。

```
//inc.C

#include "0x/mylib.h"

int main()
{
return 0;
}
```

この機能を使用可能にして、この例をコンパイルまたはプリプロセッシングすると、コンパイラーは以下の警告メッセージを出します。

```
"inc.C", line 1.10: 1540-0893 (W) The header file name "0x/mylib.h"
in #include directive shall not start with a digit.
```

#line ディレクティブの制限の引き上げ

#line <integer> プリプロセッサー・ディレクティブの上限が、C99 プリプロセッサーに適合する C++11 プリプロセッサーでは、32,767 から 2,147,483,647 に増えています。

```
#line 1000000   //Valid in C++11, but invalid in C++98
int main()
{
    return 0;
}
```

オブジェクト類似マクロ定義の診断

マクロ定義内のオブジェクト類似マクロ名とその置換リストとの間に空白文字が存在しない場合、C++11 コンパイラーは警告メッセージを出します。次の例を検討してみます。

```
//w.C

//With -qnodollar, '$' is not part of the macro name,
//thus it begins the replacement list
#define A$B c
#define STR2( x ) # x
#define STR( x ) STR2( x )
char x[] = STR( A$B );
```

この機能を使用可能にし、**-qnodollar** を指定して、この例をコンパイルまたはプリプロセッシングすると、コンパイラーは以下の警告メッセージを出します。

```
"w.C", line 1.10: 1540-0891 (W) Missing white space between
the identifier "A" and the replacement list.
```

_Pragma 演算子

_Pragma 演算子は、#pragma ディレクティブを指定する代替方式です。例えば、以下の 2 つのステートメントは等価です。

```
#pragma comment(copyright, "IBM 2010")
_Pragma("comment(copyright, ¥"IBM 2010¥")")
```

以下のコードをコンパイルすると、ストリング IBM 2010 が C++ オブジェクト・ファイルに挿入されます。

```
_Pragma("comment(copyright, ¥"IBM 2010¥")")
int main()
{
   return 0;
}
```

可変数引数マクロおよび空のマクロ引数

C99 および C++11 では可変数引数マクロおよび空のマクロ引数がサポートされています。この機能により、変数引数 ID を __VA_ARGS__ からユーザー定義の ID に名前変更する機構が使用可能になります。次の例を検討してみます。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test): printf(__VA_ARGS__))
debug("Flag");
debug("X = %d¥n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

この例は、プリプロセッシング後に以下のコードに展開されます。

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d¥n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"): printf("x is %d but y is %d", x, y));
```

事前定義マクロ

__STDC_HOSTED__ マクロは、以下のマクロが定義されているかどうかにかかわらず、1 に事前定義されます。

- __STDC__
- __STDC_VERSION__
- __STDC_ISO_10646_

関連資料:

20 ページの『整数リテラル』

34 ページの『ストリング・リテラル』

527 ページの『#include ディレクティブ』

535 ページの『#line ディレクティブ』

518 ページの『#define ディレクティブ』

539 ページの『_Pragma プリプロセッシング演算子』

555 ページの『C++11 互換性の拡張機能』

第 18 章 IBM XL C 言語拡張機能

IBM XL C 拡張機能には、以下のカテゴリーの拡張機能としての C の機能が含まれています。

- C89
- Unicode サポート
- GNU C 互換性
- ベクトル処理のサポート

一般の IBM 拡張

以下の機能は、**extc99** または **stdc99** 言語レベルが有効でない場合、xlc、xlc++、xlC、cc、および c99 呼び出しコマンドを使用する際に、デフォルトで使用可能です。 また、以下の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可にできます。

言語機能	参照先	個々のオプション制御
C99 以外の IBM `long long` 拡張	C99 および C++11 以外の整数リテラルの型	**-q[no]longlong**

関連資料:


「XL C/C++ コンパイラー・リファレンス」の中の『-qlonglong』を参照

C99 の機能

次のコマンドのいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stdc99 | extc89 | extended | extc1x** オプション

これらのオプションについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の『**-qlanglvl**』オプションを参照してください。

表 40. C89 への拡張機能としてのデフォルトの C99 の機能

言語機能	参照先
16 進浮動小数点定数	16 進浮動小数点リテラル
`__func__` 事前定義 ID	17 ページの『__func__ 事前定義 ID』
ワイド文字ストリングおよび非ワイド文字ストリングの連結	ストリング連結
混合宣言およびコード	49 ページの『データ宣言とデータ定義の概要』
複素数データ型	65 ページの『複素数浮動小数点型』

表 40. C89 への拡張機能としてのデフォルトの C99 の機能 (続き)

言語機能	参照先
_Bool データ型	64 ページの『ブール型』
enum 宣言内で許可される末尾コンマ	82 ページの『列挙型の定義』
重複型修飾子	101 ページの『型修飾子』
可変長配列	125 ページの『可変長配列』
左辺値以外の配列添え字	199 ページの『配列添え字演算子 [ ]』
構造体または共用体の最後にある柔軟な配列メンバー	柔軟な配列メンバー
構造体または共用体の初期化指定子の非定数式	133 ページの『構造体および共用体の初期化』
指定された初期化指定子	130 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』
暗黙的関数宣言の除去	264 ページの『関数宣言』
関数宣言内の暗黙的 int 戻りの型の除去	278 ページの『関数の戻りの型指定子』
関数仮パラメーターとしての静的配列	282 ページの『関数仮パラメーター宣言の中の静的配列指標 (C のみ)』
関数類似マクロの変数引数	519 ページの『関数類似マクロ』
関数類似マクロの空の引数	519 ページの『関数類似マクロ』
追加の事前定義マクロ名	525 ページの『標準の事前定義マクロ名』
複合リテラル	217 ページの『複合リテラル式』
_Pragma 演算子	539 ページの『_Pragma プリプロセッシング演算子』
標準プラグマ	539 ページの『標準のプラグマ (C のみ)』

次のコマンドのいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stdc99 | extc89 | extended | extc1x** オプション

また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 41. C89 への拡張機能としてのデフォルトの C99 の機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
2 文字表記	43 ページの『2 文字表記文字』	**-q[no]digraph**
C++ スタイル・コメント	44 ページの『コメント』	**-q[no]cpluscmt**
inline 関数指定子	273 ページの『inline 関数指定子』	**-qkeyword=inline**

次のコマンドのいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド

- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stdc99 | extc1x** オプション

表 42. C89 への拡張機能としての厳密な C99 の機能

言語機能	参照先
C99 `long long`	C99 の整数リテラルの型
注: この機能は、**-qlonglong** オプションによって制御される `long long` 型の意味とは互換性がありません。詳しくは、**-qlonglong**を参照してください。	

次のコマンドのいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stdc99 | extc1x** オプション

また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 43. C89 への拡張機能としての厳密な C99 の機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
ユニバーサル文字名	41 ページの『ユニコード規格』	**-qlanglvl=[no]ucs**
`restrict` 型修飾子	105 ページの『restrict 型修飾子』	**-qkeyword=restrict**

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qcpluscmt』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qdigraph』 を参照

「XL C/C++ コンパイラー・リファレンス」の『コンパイラーの呼び出し』を参照

C11 との互換性のための拡張機能

注: IBM は、C11 の選択された機能 (C1X と呼ばれる) をその承認の前にサポートします。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C11 標準ライブラリーのサポートを含め、すべての C11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による C11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは特に行いません。

以下の機能は、C11 に完全に準拠するための段階を踏まえた継続的なリリース・プロセスの一部です。これらは、C コンパイラーでコンパイルする場合に、**-qlanglvl=extc1x** グループ・オプションによって使用可能にすることができま

す。 C++ コンパイラーでコンパイルする場合は、一部の C11 機能も使用できます。詳しくは、個々の機能について解説しているセクションを参照してください。

表 44. C11 との互換性のための IBM XL C 言語拡張機能

言語機能	参照先
無名構造体	無名構造体
複素数型の初期化	144 ページの『複素数型の初期化 (C11)』
静的アサーション	_Static_assert 宣言
_Noreturn 関数指定子	277 ページの『_Noreturn 関数指定子』
typedef 再宣言	89 ページの『typedef 定義』

関連資料

「XL C/C++ コンパイラー・リファレンス」の **-qlanglvl**

GNU C 互換性の拡張機能

以下の機能は、デフォルトですべての言語レベルで使用可能です。

表 45. GNU C との互換性のためのデフォルト IBMXL C 拡張機能

言語機能	参照先
#include_next プリプロセッサー・ディレクティブ	528 ページの『#include_next ディレクティブ (IBM 拡張)』

次のコマンドのいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **-qlanglvl=extc99 | extc89 | extended** オプション

表 46. GNU C との互換性のためのデフォルト IBMXL C 拡張機能

言語機能	参照先
__alignof__ 演算子	183 ページの『__alignof__ 演算子 (IBM 拡張)』
__extension__ キーワード	15 ページの『言語拡張のキーワード (IBM 拡張)』
__imag__ および __real__ 複素数型演算子	187 ページの『__real__ および __imag__ 演算子』
__inline__ 関数指定子	273 ページの『inline 関数指定子』
__typeof__ 演算子	186 ページの『typeof 演算子 (IBM 拡張)』
#assert、#unassert、#cpu、#machine、#system プリプロセッサー・ディレクティブ	537 ページの『アサーション・ディレクティブ (IBM 拡張)』
#warning プリプロセッサー・ディレクティブ	535 ページの『#warning ディレクティブ (IBM 拡張)』
asm ラベル	18 ページの『アセンブリー・ラベル (IBM 拡張)』

表 46. GNU C との互換性のためのデフォルト IBMXL C 拡張機能 (続き)

言語機能	参照先
代替キーワード	15 ページの『言語拡張のキーワード (IBM 拡張)』
共用体型へのキャスト	207 ページの『共用体型へのキャスト (C のみ) (IBM 拡張)』
複素数リテラル接尾部	複素数リテラル
単項演算子への複素数型引数	176 ページの『単項式』
計算後の `goto` ステートメント	251 ページの『計算後の goto ステートメント (IBM 拡張)』
関数の属性	286 ページの『関数属性 (IBM 拡張)』
グローバル・レジスター変数	60 ページの『指定したレジスターの変数 (IBM 拡張)』
複合リテラルによる静的変数の初期化	217 ページの『複合リテラル式』
値としてのラベル	232 ページの『値としてのラベル (IBM 拡張)』
ローカルに宣言されたラベル	232 ページの『ローカルに宣言されたラベル (IBM 拡張)』
生成された左辺値	165 ページの『左辺値と右辺値』
ネストされた関数	308 ページの『ネストされた関数 (IBM 拡張)』
構造体または共用体の任意の場所への柔軟な配列メンバーの配置	柔軟な配列メンバー
式内のステートメントおよび宣言	235 ページの『ステートメント式 (IBM 拡張)』
集合体の柔軟な配列メンバーの静的初期化	柔軟な配列メンバー
型属性	107 ページの『型属性 (IBM 拡張)』
変数属性	145 ページの『変数属性 (IBM 拡張)』
可変数引数マクロ拡張機能	可変数引数マクロ拡張機能 (IBM 拡張)
ゼロ・エクステント配列	ゼロ・エクステント配列メンバー (IBM 拡張)

次のコマンドのいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **-qlanglvl=extc99 | extc89 | extended** オプション

また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 47. GNU C との互換性のための IBM XL C 拡張機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
`__thread` ストレージ・クラス指定子	61 ページの『__thread ストレージ・クラス指定子 (IBM 拡張)』	**-qtls**

表 47. GNU C との互換性のための IBM XL C 拡張機能 (個々のオプション制御付き) (続き)

言語機能	参照先	個々のオプション制御
asm、および __asm キーワード	18 ページの『アセンブリー・ラベル (IBM 拡張)』, 252 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』	**-qkeyword=asm**, **-qasm**
asm インライン・アセンブリー言語ステートメント	252 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』	**-qasm**
visibility 関数属性	297 ページの『visibility』	**-qvisibility**
visibility 変数属性	151 ページの『visibility 変数属性』	**-qvisibility**
注: 変数および関数が、プラグマ・ディレクティブ、明示的に指定された属性、または伝搬規則から visibility 属性を取得していない場合、**-qvisibility** オプションを使用して、変数および関数に visibility 属性を指定できます。このオプションは、変数または関数に対して visibility 属性を使用不可にするために使用することはできません。		

次の機能では、追加のオプションを使用してコンパイルする必要があります。

表 48. GNU C との互換性のための IBM XL C 拡張機能 (追加のコンパイラー・オプション要)

言語機能	参照先	必須コンパイル・オプション
ID 内のドル記号	16 ページの『ID の文字』	**-qdollar**
typeof キーワード	186 ページの『typeof 演算子 (IBM 拡張)』	**-qkeyword=typeof**

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qasm』を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qtls』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qdollar』を参照

「XL C/C++ コンパイラー・リファレンス」の『コンパイラーの呼び出し』を参照

Unicode の拡張機能サポート

次の機能では、追加のオプションを使用してコンパイルする必要があります。

言語機能	参照先	必須コンパイル・オプション
UTF-16、UTF-32 リテラル	42 ページの『UTF リテラル (IBM 拡張)』	**-qutf**

関連資料:


「XL C/C++ コンパイラー・リファレンス」の中の『-qutf』 を参照

ベクトル処理の拡張機能サポート

ベクトル拡張機能は、以下の条件のすべてが満たされたときのみ受け入れられます。

- **-qarch** オプションが、ベクトル処理命令をサポートするターゲット・アーキテクチャーに設定される。例えば、POWER7 などの VSX 命令セット拡張機能をサポートするアーキテクチャーでは、**-qarch=pwr7** とする必要があります。
- **-qaltivec** オプションが有効なとき。

これらのオプションについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」を参照してください。

表 49. AltiVec Application Programming Interface 仕様をサポートするための IBM XL C 拡張機能

言語機能	参照先
ベクトル・プログラミング言語の拡張機能	67 ページの『ベクトル型 (IBM 拡張)』, 30 ページの『ベクトル・リテラル (IBM 拡張)』

以下の機能は、AltiVec Application Programming Interface 仕様に対する IBM 拡張機能です。

表 50. AltiVec Application Programming Interface 仕様に対する IBM XL C 拡張機能

言語拡張	参照先
ベクトルの初期化指定子リスト	132 ページの『ベクトルの初期化 (IBM 拡張)』
ベクトル型の `typedef` 定義	89 ページの『typedef 定義』
静的ベクトル変数の初期化指定子としての複合リテラル	217 ページの『複合リテラル式』
`__alignof__` および `typeof` 演算子への引数としてのベクトル型	183 ページの『__alignof__ 演算子 (IBM 拡張)』, 186 ページの『typeof 演算子 (IBM 拡張)』

第 19 章 IBM XL C++ 言語拡張機能

IBM XL C++ 拡張機能 (標準 C++ に対するもの) には、以下が含まれます。

- C99 互換性
- Unicode サポート
- GNU C 互換性
- GNU C++ 互換性
- C++11 互換性
- ベクトル処理のサポート

C++ コンパイラーでコンパイルする場合は、一部の C11 機能も使用できます。詳しくは、547 ページの『C11 との互換性のための拡張機能』を参照してください。

一般の IBM 拡張

以下の機能は、**-qlanglvl=compat366 | extended** オプションを使用すると使用可能になります。以下の機能は、デフォルトで使用可能です。 また、以下の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可にできます。

言語機能	参照先	個々のオプション制御
C99 以外の IBM `long long` 拡張	21 ページの『C99 および C++11 以外の整数リテラルの型』	**-q[no]longlong**

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qlonglong』を参照

C99 との互換性のための拡張機能

IBM XL C++ は、以下の C99 言語機能のサポートを追加します。これらの機能のすべては、デフォルトで使用可能です。

表 51. Standard C++ への拡張機能としてのデフォルトの C99 の機能

言語機能	参照先
重複型修飾子	101 ページの『型修飾子』
構造体または共用体の最後にある柔軟な配列メンバー	柔軟な配列メンバー
`_Pragma` 演算子	539 ページの『_Pragma プリプロセッシング演算子』
追加の事前定義マクロ名	525 ページの『標準の事前定義マクロ名』
関数類似マクロの空の引数	519 ページの『関数類似マクロ』
C 標準プラグマ	539 ページの『標準のプラグマ (C のみ)』

以下の機能は、デフォルトで使用可能です。 また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 52. Standard C++ への拡張機能としてのデフォルトの C99 の機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
`__func__` 事前定義 ID	17 ページの『__func__ 事前定義 ID』	**-qlanglvl=[no]c99__func__**
16 進浮動小数点リテラル	16 進浮動小数点リテラル	**-qlanglvl=[no]c99hexfloat**
複素数データ型	65 ページの『複素数浮動小数点型』	**-qlanglvl=[no]c99complex**
`enum` 宣言内で許可される末尾コンマ	82 ページの『列挙型の定義』	**-qlanglvl=[no]trailenum**
`restrict` 型修飾子	105 ページの『restrict 型修飾子』	**-q[no]keyword=restrict**
可変長配列	125 ページの『可変長配列』	**-qlanglvl=[no]c99vla**
複合リテラル	217 ページの『複合リテラル式』	**-qlanglvl=[no]c99compoundliteral**
関数類似マクロの変数引数	519 ページの『関数類似マクロ』	**-qlanglvl=[no]varargmacros**

以下の機能は、下記の表にリストされている特定のコンパイラー・オプションによってのみ使用可能です。

表 53. Standard C++ への拡張機能としての C99 の機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
ユニバーサル文字名	41 ページの『ユニコード規格』	**-qlanglvl=ucs**

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照

Unicode の拡張機能サポート

以下の機能は、デフォルト言語レベルの **-qlanglvl=extended** オプションで使用可能です。以下の機能は、デフォルトで使用可能です。また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

言語機能	参照先	必須コンパイル・オプション
UTF-16、UTF-32 リテラル	42 ページの『UTF リテラル (IBM 拡張)』	**-qutf**

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qutf』 を参照

C++11 互換性の拡張機能

注: IBM は、C++11 (承認前の呼称は C++0x) の、選定された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含め、すべての C++11 機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による新規 C++11 機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは、特に行いません。

以下の機能は、C++11 に完全に準拠するための段階を踏まえた継続的なリリース・プロセスの一部です。これらは、**-qlanglvl=extended0x** グループ・オプションを使用して使用可能にすることができます。特定のコンパイラー・オプションを使用してこれらの機能を有効または無効にすることもできます。以下の表を参照してください。

表 54. C++11 との互換性のための IBM XL C++ 言語拡張機能

言語機能	参照先	C++11 個別サブオプション制御
auto 型推定	91 ページの『auto 型指定子 (C++11)』	**-qlanglvl=[no]autotypedededuction**
C99 `long long`	22 ページの『C99 および C++11 の整数リテラルの型』	**-qlanglvl=[no]c99longlong** IBM **-qlanglvl=[no]extendedintegersafe**
C++11 に採用された C99 プリプロセッサー機能	540 ページの『C++11 に採用された C99 プリプロセッサー機能』	**-qlanglvl=[no]c99preprocessor**
decltype	93 ページの『decltype(*expression*) 型指定子 (C++11)』	**-qlanglvl=[no]decltype**
デフォルトに設定された関数または削除済み関数	266 ページの『明示的にデフォルトに設定された関数』 267 ページの『削除済み関数』	**-qlanglvl=[no]defaultanddelete**
委任コンストラクター	412 ページの『委任コンストラクター (C++11)』	**-qlanglvl=[no]delegatingctors**
明示的型変換演算子	431 ページの『明示的型変換演算子 (C++11)』	**-qlanglvl=[no]explicitconversionoperators**
明示的インスタンス生成宣言	464 ページの『 明示的インスタンス生成宣言』	**-qlanglvl=[no]externtemplate**

表 54. C++11 との互換性のための IBM XL C++ 言語拡張機能 (続き)

言語機能	参照先	C++11 個別サブオプション制御
拡張フレンド宣言	372 ページの『フレンド』	**-qlanglvl=[no]extendedfriend**
一般化された定数式	174 ページの『一般化された定数式 (C++11)』	**-qlanglvl=[no]constexpr**
インライン名前空間定義	320 ページの『インライン名前空間定義 (C++11)』	**-qlanglvl=[no]inlinenamespace**
Nullptr	120 ページの『NULL ポインター』	**-qlanglvl=[no]nullptr**
参照の縮約	227 ページの『参照の縮約 (reference collapsing) (C++11)』	**-qlanglvl=[no]referencecollapsing**
右不等号括弧	447 ページの『クラス・テンプレート』	**-qlanglvl=[no]rightanglebracket**
右辺値参照	**右辺値参照の使用 (C++11)**	**-qlanglvl=[no]rvaluereferences**
スコープ付き列挙型	81 ページの『列挙型』	**-qlanglvl=[no]scopedenum**
static_assert	52 ページの『static_assert 宣言 (C++11)』	**-qlanglvl=[no]static_assert**
後置戻り型	283 ページの『後置戻り型 (C++11)』	**-qlanglvl=[no]autotypededuction**
可変数引数テンプレート	476 ページの『可変数引数テンプレート (C++11)』	**-qlanglvl=[no]variadic[templates]**

注:

- **-qlanglvl=extended** グループ・オプションを使用して、明示的インスタンス生成宣言機能を使用可能にすることもできます。
- C++11 機能が有効ではないときにそれを使用しようとすると、コンパイラーは構文エラー・メッセージの後に情報メッセージを発行します。この情報メッセージは、C++11 機能をオンにして構文エラーから復旧する方法を示します。関連する C++11 機能は、以下のとおりです。
  - C++11 に採用された C99 プリプロセッサー機能
    - 混合ストリング・リテラルの連結
    - `__STDC_HOSTED__` マクロ
  - デフォルトに設定された関数または削除済み関数
  - 委任コンストラクター
  - 明示的型変換演算子
  - 一般化された定数式
  - インライン名前空間定義

– nullptr
– 参照の縮約
– 右不等号括弧
– 右辺値参照
– スコープ付き列挙型
– 可変数引数テンプレート

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

 -qlanglvl

GNU C 互換性の拡張機能

以下の GNU C 言語拡張のサブセットは、デフォルトで使用可能です。

表 55. GNU C との互換性のためのデフォルト IBMXL C++ 拡張機能

言語機能	参照先
__alignof__ 演算子	183 ページの『__alignof__ 演算子 (IBM 拡張)』
__imag__ および __real__ 複素数型演算子	187 ページの『__real__ および __imag__ 演算子』
__typeof__ 演算子	186 ページの『typeof 演算子 (IBM 拡張)』
アセンブリー・ラベル	18 ページの『アセンブリー・ラベル (IBM 拡張)』
単項演算子への複素数型引数	176 ページの『単項式』
関数の属性	286 ページの『関数属性 (IBM 拡張)』
生成された左辺値	165 ページの『左辺値と右辺値』
構造体または共用体の任意の場所への柔軟な配列メンバーの配置	柔軟な配列メンバー
式内のステートメントおよび宣言	235 ページの『ステートメント式 (IBM 拡張)』
集合体の柔軟な配列メンバーの静的初期化	柔軟な配列メンバー

以下の GNU C 言語拡張のサブセットは、デフォルトで使用可能です。 また、これらの拡張機能は、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 56. GNU C との互換性のためのデフォルト IBMXL C++ 拡張機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
__extension__ キーワード	15 ページの『言語拡張のキーワード (IBM 拡張)』	**-q[no]keyword=__extension__**
#assert、#unassert、#cpu、#machine、#system プリプロセッサー・ディレクティブ	537 ページの『アサーション・ディレクティブ (IBM 拡張)』	**-qlanglvl=[no]gnu_assert**

表 56. GNU C との互換性のためのデフォルト IBMXL C++ 拡張機能 (個々のオプション制御付き) (続き)

言語機能	参照先	個々のオプション制御
`#include_next` プリプロセッサー・ディレクティブ	528 ページの『#include_next ディレクティブ (IBM 拡張)』	**-qlanglvl=[no]gnu_include_next**
`#warning` プリプロセッサー・ディレクティブ	535 ページの『#warning ディレクティブ (IBM 拡張)』	**-qlanglvl=[no]gnu_warning**
代替キーワード	15 ページの『言語拡張のキーワード (IBM 拡張)』	**-q[no]keyword=token**
複素数リテラル接尾部	複素数リテラル	**-qlanglvl=[no]gnu_suffixij**
計算後の `goto` ステートメント	251 ページの『計算後の goto ステートメント (IBM 拡張)』	**-qlanglvl=[no]gnu_computedgoto**
関数の属性	286 ページの『関数属性 (IBM 拡張)』	**-q[no]keyword=__attribute__**
インライン・アセンブリー言語ステートメント	252 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』	**-qasm**
値としてのラベル	232 ページの『値としてのラベル (IBM 拡張)』	**-qlanglvl=[no]gnu_labelvalue**
ローカルに宣言されたラベル	232 ページの『ローカルに宣言されたラベル (IBM 拡張)』	**-qlanglvl=[no]gnu_locallabel**
型属性	107 ページの『型属性 (IBM 拡張)』	**-q[no]keyword=__attribute__**
`typeof` 演算子	186 ページの『typeof 演算子 (IBM 拡張)』	**-q[no]keyword=typeof**
変数属性	145 ページの『変数属性 (IBM 拡張)』	**-q[no]keyword=__attribute__**
可変数引数マクロ拡張機能	可変数引数マクロ拡張機能 (IBM 拡張)	**-qlanglvl=[no]gnu_varargmacros**
visibility 関数属性	297 ページの『visibility』	**-qvisibility**
visibility 変数属性	151 ページの『visibility 変数属性』	**-qvisibility**
ゼロ・エクステント配列	ゼロ・エクステント配列メンバー (IBM 拡張)	**-qlanglvl=[no]zeroextarray**
注: 変数および関数が、プラグマ・ディレクティブ、明示的に指定された属性、または伝搬規則から visibility 属性を取得していない場合、**-qvisibility** オプションを使用して、変数および関数に visibility 属性を指定できます。このオプションは、変数または関数に対して visibility 属性を使用不可にするために使用することはできません。		

以下の機能は、下記の表でリストされている追加オプションを使用してコンパイルする必要があります。

表 57. GNU C との互換性のための IBM XL C++ 拡張機能 (追加のコンパイラー・オプション要)

言語機能	参照先	必須コンパイル・オプション
ID 内のドル記号	16 ページの『ID の文字』	**-qdollar**

関連資料:


「XL C/C++ コンパイラー・リファレンス」の中の『-qkeyword』 を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qasm』を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qtls』 を参照


「XL C/C++ コンパイラー・リファレンス」の中の『-qdollar』を参照

GNU C++ 互換性の拡張機能

以下の GNU C++ 言語拡張機能は、デフォルトで使用可能です。

表 58. GNU C++ との互換性のための IBM XL C++ 言語拡張機能

言語機能	参照先
`init_priority` 変数属性	148 ページの『init_priority 変数属性 (C++ のみ)』

以下の GNU C++ 言語拡張機能は、デフォルトで使用可能です。また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 59. GNU C++ との互換性のための IBM XL C++ 言語拡張機能 (個々のオプション制御付き)

言語機能	参照先	個々のオプション制御
`extern` として宣言されるテンプレートのインスタンス化	463 ページの『テンプレートのインスタンス生成』	**-qlanglvl=[no]gnu_externtemplate**
`__thread` ストレージ・クラス指定子	61 ページの『__thread ストレージ・クラス指定子 (IBM 拡張)』	**-qtls**
visibility 名前空間属性	323 ページの『visibility 名前空間属性 (IBM 拡張)』	**-qvisibility**
visibility 型属性	112 ページの『visibility 型属性 (C++ のみ)』	**-qvisibility**

表 59. GNU C++ との互換性のための IBM XL C++ 言語拡張機能 (個々のオプション制御付き) (続き)

言語機能	参照先	個々のオプション制御
注: • オプション **-qlanglvl=[no]gnu_externtemplate** は、XL C/C++ V13.1 では推奨されません。代わりに、オプション **-qlanglvl=[no]externtemplate** を使用すると、C++11 標準に導入される明示的インスタンス生成宣言機能を制御できます。詳細については、「*XL C/C++ ランゲージ・リファレンス*」の『明示的インスタンス生成宣言』を参照してください。 • 型および名前空間が、プラグマ・ディレクティブ、明示的に指定された属性、または伝搬規則から visibility 属性を取得していない場合、**-qvisibility** オプションを使用して、型および名前空間に visibility 属性を指定できます。このオプションは、型または名前空間に対して visibility 属性を使用不可にするために使用することはできません。		

関連資料:

「XL C/C++ コンパイラー・リファレンス」の中の『-qlanglvl』 を参照

「XL C/C++ コンパイラー・リファレンス」の中の『-qtls』 を参照

ベクトル処理の拡張機能サポート

ベクトル拡張機能は、以下の条件のすべてが満たされたときのみ受け入れられます。

- **-qarch** オプションが、ベクトル処理命令をサポートするターゲット・アーキテクチャーに設定される。例えば、POWER7 などの VSX 命令セット拡張機能をサポートするアーキテクチャーでは、**-qarch=pwr7** とする必要があります。
- **-qaltivec** オプションが有効なとき。

これらのオプションについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」を参照してください。

表 60. AltiVec Application Programming Interface 仕様をサポートするための IBM XL C++ 拡張機能

言語機能	参照先
ベクトル・プログラミング言語の拡張機能	67 ページの『ベクトル型 (IBM 拡張)』, 30 ページの『ベクトル・リテラル (IBM 拡張)』

以下の機能は、AltiVec Application Programming Interface 仕様に対する IBM 拡張機能です。

表 61. AltiVec Application Programming Interface 仕様に対する IBM XL C++ 拡張機能

言語拡張	参照先
ベクトルの初期化指定子リスト	132 ページの『ベクトルの初期化 (IBM 拡張)』
ベクトル型の `typedef` 定義	89 ページの『typedef 定義』

表 61. AltiVec Application Programming Interface 仕様に対する IBM XL C++ 拡張機能 (続き)

言語拡張	参照先
静的ベクトル変数の初期化指定子としての複合リテラル	217 ページの『複合リテラル式』
`__alignof__` および `typeof` 演算子への引数としてのベクトル型	183 ページの『__alignof__ 演算子 (IBM 拡張)』, 186 ページの『typeof 演算子 (IBM 拡張)』

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。

〒103-8510
東京都中央区日本橋箱崎町19番21号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、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。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

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, 2014.

商標

IBM、IBM ロゴ および ibm.com® は、世界の多くの国で登録された International Business Machines Corporation の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Adobe および Adobe ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

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。

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]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

V

W

［特殊文字］

プログラム番号: 5765-J08; 5725-C73

Printed in Japan

SA88-5396-00

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21