IBM

System i

ILE C/C++ 解説書

バージョン 6 リリース 1

SC88-4026-01
(英文原典：SC09-7852-01)

System i

ILE C/C++ 解説書

バージョン 6 リリース 1

SC88-4026-01
(英文原典：SC09-7852-01)

ご注意

本書および本書で紹介する製品をご使用になる前に、 349 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 WebSphere® Development Studio for System i (プログラム 5761–WDS)、ILE C/C++ コンパイラーのバージョン 6、リリース 1、モディフィケーション 0 に適用されます。また、改訂版で断りがない限り、それ以降のすべてのリリースおよびモディフィケーションに適用されます。本書は、*ILE C/C++ Language Reference*、SC09-7852-00 の改訂版です。

IBM 発行のマニュアルに関する情報のページ

`http://www.ibm.com/jp/manuals/`

こちらから、日本語版および英語版のオンライン・ライブラリーをご利用いただけます。また、マニュアルに関するご意見やご感想を、上記ページよりお送りください。今後の参考にさせていただきます。

(URL は、変更になる場合があります)

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

原　典： SC09-7852-01
System i
ILE C/C++ Language Reference
Version 6 Release 1

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2008.2

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

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

目次

ILE C/C++ ランゲージ・リファレンス (SC09-7852-01)

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

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

本書で説明する C および C++ 言語は、以下の標準に基づきます。

- *Programming languages - C* (ISO/IEC 9899:1990) に記述されている C 言語。本書では *C89* と呼びます。これは、最初の ISO C 規格です。
- *Programming languages - C++* (ISO/IEC 14882:1998) に記述されている C++ 言語。これは、最初の正式な言語定義です。

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

本書の対象読者

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

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

言語仕様に厳密に準拠するプログラムは、異なる環境間で最大の移植性を発揮します。理論上、標準に準拠した、あるコンパイラーで正しくコンパイルされたプログラムは、ハードウェアの差異が許す範囲内で、他

のすべての標準準拠のコンパイラーの下で正確にコンパイルされ、作動します。言語インプリメンテーションによって提供される言語への拡張子を正しく活用したプログラムは、そのオブジェクト・コードの効率性を向上させることができます。

強調表示の規則

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

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

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

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

構文図の読み方

- 構文図は、左から右、上から下に、線のパスに従って読んでください。

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

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

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

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

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

  注: 以下のダイアグラムで、`statement` は、C または C++ コマンド、ディレクティブ、またはステートメントを表しています。
- 必須項目は、水平線 (メインパス) 上に記述されます。

```
▶▶─statement─required_item─────────────────────────────────────────────────▶◀
```

- オプション項目は、メインパスの下に記述されます。

```
▶▶─statement─┬──────────────┬──────────────────────────────────────────────▶◀
             └─optional_item─┘
```

- 2 つ以上の項目から選択可能な場合は、スタック内に垂直に記述されます。

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

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

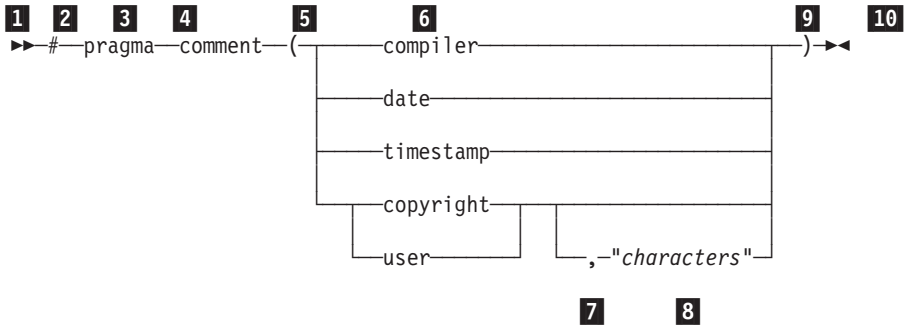
デフォルト項目は、メインパスの上に記述されます。

- メインパスの線の上の左に戻る矢印は、繰り返し可能な項目を示します。

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

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

次の構文図の例では、**#pragma comment** ディレクティブの構文を示しています。 **#pragma** ディレクティブの詳細については、 190 ページの『プラグマ・ディレクティブ (#pragma)』を参照してください。



1 構文図の始まりを示します。

2 記号 `#` を最初に記述します。

3 キーワード `pragma` は、シンボル `#` の次に記述されます。

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

5 左括弧が必要です。

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

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

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

9 右小括弧は必須です。

10 これが、構文図の終わりを示します。

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

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

前提条件および関連情報

IBM i5/OS Information Center は、IBM System i™ の技術情報の開始点として使用してください。

Information Center には、次の 2 とおりの方法でアクセスすることができます。

- 以下の Web サイトから。

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

- *i5/OS Information Center CD*、*SK88-8055* から。この CD-ROM は、新しい System i ハードウェアまたは i5/OS ソフトウェア・アップグレード版に付属されています。IBM Publications Center から CD-ROM を注文することもできます。

  http://www.ibm.com/shop/publications/order

i5/OS Information Center には、ソフトウェアおよびハードウェア・インストール、Linux、WebSphere、Java、高可用性、データベース、論理区画、CL コマンド、およびシステムのアプリケーション・プログラミング・インターフェース (API) などの新規および更新システム情報が含まれています。また、システムのハードウェアおよびソフトウェアを計画、トラブルシューティング、および構成するのに役立つアドバイザーや検索機能も提供されています。

新しいハードウェアを注文すると、*System i Access for Windows DVD*、*SK88-8058* が付属されてきます。この DVD は、IBM System i Access for Windows のライセンス・プログラムのインストール用に提供されています。System i Access Family には、System i モデルに PC を接続するためのクライアントおよびサーバー機能が装備されています。

変更の要約

このエディションでの情報の変更は、以下のとおりです。

- __thread ストレージ・クラス指定子のサポート。
- 10 進浮動小数点データ型のサポート。
- すべてのモジュールでのテラスペース・アドレス (TERASPACE(*NO) を指定してコンパイルした場合も含む) の使用可能化。

第 1 章 スコープおよびリンケージ

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

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

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

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

スコープとリンケージの違いは、スコープはコンパイラーが利用し、リンケージはバインダーが利用する点です。ソース・ファイルをオブジェクト・コードに変換するとき、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、それをモジュール内にエクスポートとして保管します。バインダーは、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。

関連参照

- 6 ページの『プログラム・リンケージ』

スコープ

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

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

スコープ

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

スコープの種類

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

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

```
int x;
void f() {

  int x = x;
}
```

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

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

グローバル・スコープ または*グローバル・ネーム・スペース・スコープ* は、オブジェクト、関数、型、およびテンプレートを定義できるプログラムの最外部のネーム・スペース・スコープです。ネーム・スペース定義を使用して、グローバル・スコープ内でユーザー定義のネーム・スペースをネストすることができます。ユーザー定義のネーム・スペースは、それぞれグローバル・スコープと区別される別のスコープになります。

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

```
int f();

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

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

クラスの暗黙宣言は、同じクラスの別の宣言が見えるまでは可視になりません。

ローカル・スコープ

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

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

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

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

関連参照
- 159 ページの『ブロック・ステートメント』

関数スコープ

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

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

関連参照
- 157 ページの『ラベル』

関数プロトタイプ・スコープ

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

関連参照
- 136 ページの『関数宣言』

グローバル・スコープ

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

C++ ID の宣言が、すべてのブロック、ネーム・スペース、およびクラスの外側で現れる場合は、名前にグローバル・ネーム・スペース・スコープがあります。 グローバル・ネーム・スペース・スコープと内部結合付きの名前は、名前が宣言された位置から変換単位の終わりまで可視になります。

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

関連参照
- 191 ページの『第 10 章 ネーム・スペース』
- 6 ページの『内部結合』

クラス・スコープ

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

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

クラス・メンバーの名前には、クラス・スコープ があり、次のケースでのみ使用できます。
- そのクラスのメンバー関数内
- そのクラスから派生したクラスのメンバー関数内
- そのクラスのインスタンスに適用された . (ドット) 演算子の後
- そのクラスから派生したクラスのインスタンスに適用された . (ドット) 演算子の後 (派生したクラスが名前を隠蔽しない場合)
- そのクラスのインスタンスを指すポインターに適用された -> (矢印) 演算子の後
- そのクラスから派生したクラスのインスタンスを指すポインターに適用された -> (矢印) 演算子の後 (派生したクラスが名前を隠蔽しない場合)
- クラスの名前に適用された :: (スコープ・レゾリューション) 演算子の後
- そのクラスから派生したクラスに適用された :: (スコープ・レゾリューション) 演算子の後

関連参照
- 213 ページの『第 12 章 クラス』
- 216 ページの『クラス名のスコープ』
- 251 ページの『規定クラス・メンバーのアクセス制御』

ID のネーム・スペース

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

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

次の 4 つの各ネーム・スペース内では、ID を固有にする必要があります。
- 次の型のタグ は、単一スコープ内で固有にする必要があります。

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

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

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

```
int get_item()
{
   struct student     /* structure tag                   */
   {
      char name[20];  /* this structure member may not be named student      */
      int section;
      int id;
   } sam;             /* this structure variable should not be named student */

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

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

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

名前の隠蔽

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

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

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

ID のネーム・スペース

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

namespace A {
  char x;
};

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

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

次に、上記の例の出力を示します。

```
int
```

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

関連参照
- 213 ページの『第 12 章 クラス』
- 224 ページの『メンバー関数』
- 227 ページの『メンバー・スコープ』
- 191 ページの『第 10 章 ネーム・スペース』

プログラム・リンケージ

リンケージ は、同一の名前を持つ複数の ID が、たとえこれらの ID が異なる変換単位に現れたとしても、同じオブジェクト、関数、または他のエンティティ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。 リンケージには外部結合、内部結合、リンケージなしの 3 つのタイプがあります。

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

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

400 異なるプログラミング言語で作成された変換単位間に、*言語リンケージ* と呼ばれるリンケージを設定することができます。言語リンケージを使用すると、ある ILE 言語のコードを別の ILE 言語で作成されたコードにリンクできるようにして、すべての ILE 言語の間の関係をクローズ状態にすることができます。 C++ では、すべての ID にデフォルトで C++ の言語リンケージが設定されています。言語リンケージは、異なる変換単位間で整合している必要があります。 C 以外または C++ 以外の言語リンケージは、ID は外部リンケージを持つことを意味します。

i5/OS 固有の使用の詳細については、「*ILE C/C++ Programmer's Guide*」の第 23 章『Using ILE C/C++ Call Conventions』を参照してください。

内部結合

次の種類の ID は、内部結合を持っています。
- **static** と明示的に宣言されたオブジェクト、参照、または関数。

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

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

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

外部結合

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

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

C++ ネーム・スペース・スコープでは、次にあげる種類のエンティティの ID は、外部結合を持っています。
- 内部結合を持たない参照またはオブジェクト。
- 内部結合を持たない関数。
- 名前付きのクラスまたは列挙。
- typedef 宣言で定義された名前なしクラスまたは列挙。
- 外部結合を持っている列挙の列挙子。
- 内部結合を持つ関数テンプレートでない場合のテンプレート。
- 名前なしネーム・スペース内で宣言されていない場合のネーム・スペース。

クラスの ID が外部結合を持っている場合は、そのクラスのインプリメンテーションでは、以下のものの ID も外部結合を持ちます。
- メンバー関数。
- 静的データ・メンバー。
- クラス・スコープのクラス。
- クラス・スコープの列挙。

リンケージなし

次の種類の ID には、リンケージがありません。
- 外部結合も内部結合も持たない名前

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

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

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

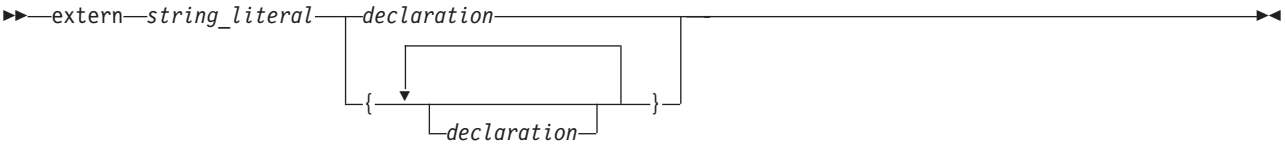
コンパイラーは、外部結合を持つ a1 の宣言を許可しません。クラス A は、リンケージを持っていません。コンパイラーは、外部結合を持つ a2 の宣言を許可しません。 A がリンケージを持っていないので、**typedef** 名 a2 は、リンケージを持っていません。

リンケージ指定 — C++ 以外のプログラムへのリンク

400 C i5/OS システムでは、#pragma 引数 を使用することで、C で言語リンケージを使用することができます。詳細については、「*ILE C/C++ コンパイラー参照*」の第 3 章『ILE C/C++ Pragmas』および「*ILE C/C++ Programmer's Guide*」の第 23 章『Using ILE C/C++ Call Conventions』を参照してください。

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

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



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

"C++"	別に指定されていなければ、オブジェクトおよび関数では、これがデフォルトのリンケージ指定です。
"C"	C プロシージャーへのリンケージを示します。

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

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

次の例では、C++ から呼び出される C 印刷関数を示します。

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

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

ネーム・マングリング

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

マングル名を作成する方法は、ソース・コードをコンパイルするために使用するオブジェクト・モデルによって異なります。特定のオブジェクト・モデルを使用してコンパイルされたクラスのオブジェクトのマングル名は、別のオブジェクト・モデルを使用してコンパイルされた同じクラスのオブジェクトのマングル名とは異なります。オブジェクト・モデルは、コンパイラー・オプションまたはプラグマによって制御されます。

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

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

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

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

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

第 2 章 字句エレメント

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

トークン

プリプロセ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 5 つの型があります。

- ID
- キーワード
- リテラル
- 演算子
- 区切り子

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

区切り子

区切り子 は、コンパイラ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。 C89 言語レベルで、区切り子はアクションを起こしません。例えば、コンマは、引数リストまたは初期化指定子リストの区切り子ですが、括弧で囲んだ式の中で使用される場合は演算子です。

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

```
[   ]       (   )       {   }       ,           :           ;
```

または、以下のもの。

```
*           =           ...         #
```

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

C++ C89 区切り子およびプリプロセッシング・トークンに加えて、C++ では、区切り子の適正なトークンまたはプリプロセッシング・トークン数は増加し、C および C++ 演算子を包含します。実行する演算を指定する区切り子は、*演算子* として認識されます。C++ では、以下のトークンを使用できます。このトークンには、プリプロセッサーによって演算子および区切り子のトークンに変換されるプリプロセッシング・トークンが含まれています。

```
.           ->          ++          --          .*          ->*
&           +           -           ~           !           ::
/           %           <<          >>          new         delete
<           >           <=          >=          ==          !=
```

^	|	&&	||	?	
*=	/=	%=	+=	-=	
<<=	>>=	&=	^=	|=	##
<:	:>	<%	%>	%:	%:%:
and	and_eq	bitand	bitor	comp	
not	not_eq	or	or_eq	xor	xor_eq

代替トークン

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

演算子または区切り子	代替表記
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:
&&	and
|	bitor
||	or
^	xor
~	compl
&	bitand
&=	and_eq
|=	or_eq
^=	xor_eq
!	not
!=	not_eq

ソース・プログラムの文字セット

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

- アルファベットの大文字および小文字

  ```
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  ```

- 0 から 9 までの 10 進数字

  ```
  0 1 2 3 4 5 6 7 8 9
  ```

- 以下の図形文字

  ```
  ! " # % & ' ( ) * + , - . / :
  ; < = > ? [ ¥ ] _ { } ~
  ```

  – ASCII の脱字記号 (^) 文字 (ビット単位の排他 OR 記号) または EBCDIC の等価否定 (¬) 文字

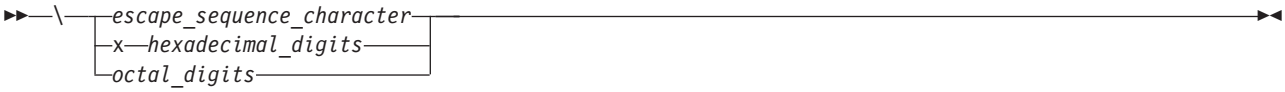
  – ASCII の分割垂直バー (¦) 文字この文字は、EBCDIC システムでは、垂直バー (|) 文字によって表すことができます

- 空白文字
- 改行、水平タブ、垂直タブ、改ページ、およびストリングの終了 (NULL 文字) を表す制御文字

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

エスケープ・シーケンス

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



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

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

エスケープ・シーケンスと表される文字は、次のとおりです。

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

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

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

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

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

このステートメントでは、次のように出力されます。

```
The escape sequence ¥n.
```

ユニコード規格

ユニコード規格 は、書かれた文字とテキストに対するコード化スキームの仕様です。ユニコード規格は、マルチリンガル・テキストのエンコードに整合性を持たせ、矛盾を起こさずにテキスト・データを国際的に交換できるようにした汎用の規格です。 C および C++ 用の ISO 規格は、 *ISO/IEC 10646-1:2000, Information Technology—Universal Multiple-Octet Coded Character Set (UCS)* です。(*octet* という用語は、ISO ではバイトを指す用語として使用されます。) ISO/IEC 10646 規格は、エンコードのフォーム数がユニコード規格より制限されています。すなわち、ISO/IEC 10646 に準ずる文字セットはユニコード規格も満たします。

ユニコード規格は、各文字に固有の数値と名前を指定しており、数値のビット表現に 3 つのエンコード方式を定義しています。名前と値のペアで文字の識別を行います。文字を表す 16 進値はコード・ポイントと呼ばれます。仕様には、全体の文字特性、すなわち、各文字の大/小文字、方向性、英字特性、その他のセマンティクス情報も記述されています。 ASCII に基づくと、ユニコード規格は英字、表意文字、およびシンボルを扱い、予約済みコード・ポイント範囲でインプリメンテーション別の文字コードを定義することができます。したがって、ユニコード規格のエンコード・スキームは、世界中のあらゆる国の歴史的な筆記体も含めて、既知のすべての文字エンコードに対応できるだけの柔軟性を十分に持っています。

C++ では、ISO/IEC 10646 で定義されている汎用文字名構成を使用して、基本のソース文字集合外の文字を表すことができます。汎用文字名は、ID、文字定数、およびストリング・リテラルで使用できます。この言語フィーチャーはコンパイル時に指定される言語レベルには依存しません。

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

汎用文字名	ISO/IEC 10646 の短縮名
¥UNNNNNNNN	NNNNNNNN
¥uNNNN	0000NNNN

ここでは、N は 16 進数字です。

C++ は、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じています。また以下の文字も使用できません。

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

3 文字表記

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

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

プリプロセッサーが、3 文字表記を対応する単一文字表示に置換します。

マルチバイト文字

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

関連参照

- 41 ページの『char および wchar_t 型指定子』

コメント

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

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

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

マルチバイト文字は、コメントにも含めることができます。

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

次のプログラムでは、2 番目の `printf()` がコメントです。

```
#include <stdio.h>

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

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

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

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

```
#include <stdio.h>

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

このプログラムの出力は、次のようになります。

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

次の例では、コメントは強調表示されています。

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

    #include <stdio.h>

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

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

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

```
    /* A program with conditional compilation to avoid nested comments.
*/
    #define TEST_FUNCTION 0
    #include <stdio.h>

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

    int test_function(void)
    {
        int number;
        char letter;
     #if TEST_FUNCTION
       number = 55;
       letter = 'A';
       /*number = 44;*/
     #endif  /*TEST_FUNCTION */
    }
```

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

```
#include <stdio.h>

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

関連参照

- 14 ページの『3 文字表記』

ID

ID は次の言語エレメントに名前を付けます。

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

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

C++ 基本ソース文字セット外の文字および数字の汎用文字名も使用できます。

予約済みの ID

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

C 1 文字の下線で始まる ID は、通常のネーム・スペースとタグ・ネーム・スペースのどちらにおいてもファイル・スコープを持つ ID として予約されています。

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

ID での大文字小文字の区別と特殊文字

コンパイラーでは、ID 内の大文字と小文字が区別されます。例えば、PROFIT と profit は、別の ID を表します。

関数名や変数名に、下線 (_) で始まる ID を使用しないでください。

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

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

標準のライブラリー関数を使用するときには常に、適切なヘッダーをインクルードする必要があります。

システム呼び出しおよびライブラリー関数の名前は、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。

キーワード

キーワード は、言語の特別な用途のために予約された ID です。キーワードは、プリプロセッサーのマクロ名に使用できますが、プログラミング・スタイルとしては良い方法ではありません。キーワードの厳密なスペルのみが予約されています。例えば、auto は予約されていますが、AUTO は予約されていません。下記に、C および C++ 言語の両方に共通するキーワードをリストします。

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

C++ C++ 言語では、次のキーワードも予約されています。

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

言語拡張のキーワード

400 ILE C/C++ は、標準言語キーワードの他に、言語拡張のため、および将来の使用のために ID を予約しています。 言語拡張での使用のために、次のキーワードが予約されています。

decimal	**_Decimal64**	**__align**	**__ptr128**	**__thread**
_Decimal	**_Decimal128**	**__alignof**	**__ptr64**	
_Decimal32		**_Packed**		

演算子および区切り子の代替表記

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

and	**bitor**	**not_eq**	**xor**
and_eq	**compl**	**or**	**xor_eq**
bitand	**not**	**or_eq**	

リテラル

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

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

- C++ ブール
- 整数
- 文字
- 浮動小数点
- 400 パック 10 進数
- ストリング

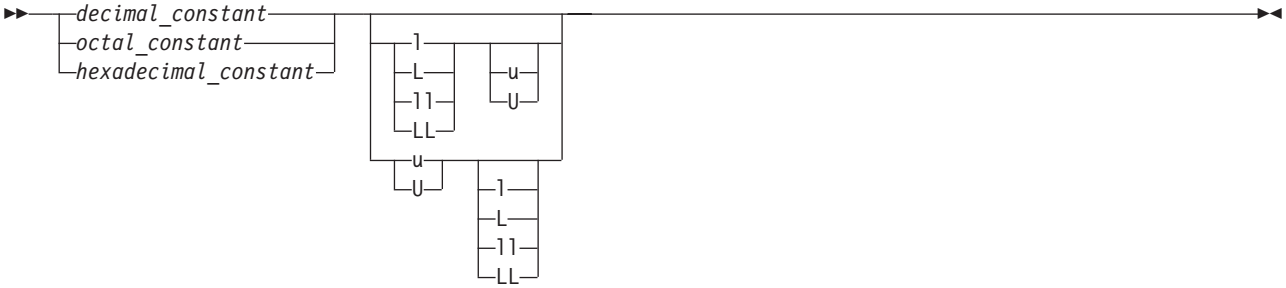
ブール・リテラル

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

関連参照
- 40 ページの『ブール変数』
- 83 ページの『左辺値と右辺値』

整数リテラル

整数リテラル は、10 進、8 進、または 16 進の値を表します。これらは、小数点または指数部を持たない数値です。ただし、整数リテラルは、基数を指定する接頭部、または型を指定するサフィックスを持つことができます。



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

整数リテラル	可能なデータ型
サフィックスがない 10 進数	**int**、**long int**、**unsigned long int**、**long long int**
サフィックスがない 8 進数	**int**、**unsigned int**、**long int**、**unsigned long int**、**long long int**、**unsigned long long int**
サフィックスがない 16 進数	**int**、**unsigned int**、**long int**、**unsigned long int**、**long long int**、**unsigned long long int**
u または **U** の接尾部が付く 10 進数、8 進数、または 16 進数	**unsigned int**、**unsigned long int**、**unsigned long long int**
サフィックスとして **l** または **L** が付く 10 進数	**long int**、**long long int**
サフィックスとして **l** または **L** が付く 8 進数または 16 進数	**long int**、**unsigned long int**、**long long int**、**unsigned long long int**
u または **U** と **l** または **L** の両方の接尾部が付く 10 進数、8 進数、または 16 進数	**unsigned long int**、**unsigned long long int**
サフィックスとして **ll** または **LL** が付く 10 進数	**long long int**
サフィックスとして **ll** または **LL** が付く 8 進数または 16 進数	**long long int**、**unsigned long long int**
u または **U** と **ll** または **LL** の両方の接尾部が付く 10 進数、8 進数、または 16 進数	**unsigned long long int**

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

10 進整数リテラル

10 進整数リテラル には、0 から 9 までの数字が含まれます。最初の数字を 0 にすることはできません。

数字 0 で始まる整数リテラルは、10 進整数リテラルではなく、8 進整数リテラルと解釈されます。

10 進リテラルの例を次に示します。

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

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

16 進整数リテラル

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

16 進整数リテラルの例を次に示します。

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

8 進整数リテラル

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

8 進整数リテラルの例を次に示します。

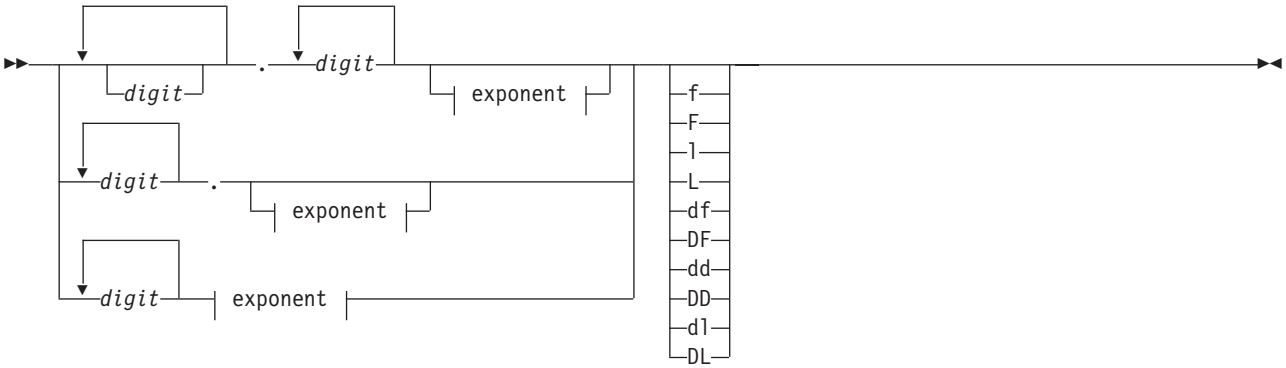
```
0
0125
034673
03245
```

浮動小数点リテラル

浮動小数点リテラル は、次のもので構成されます。

- 整数部分
- 小数点
- 小数部分
- 指数部分
- オプションの接尾部

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。



Exponent:

float の大きさの範囲は、おおよそ 1.2e-38 から 3.4e38 までです。 **double** または **long double** の大きさの範囲は、おおよそ 2.2e-308 から 1.8e308 までです。 **_Decimal32** の範囲は 1.000000e-95 から 9.999999e+96 までです。**_Decimal64** の範囲は 1.000000000000000e-383 から 9.999999999999999e+384 までです。 **_Decimal128** の範囲は 1.000000000000000000000000000000000e-6143 から 9.999999999999999999999999999999999e+6144 までです。浮動小数点定数が長すぎたり、短すぎたりする場合は、言語での結果は予期できません。

サフィックス **f** または **F** は **float** 型を示し、サフィックス **l** または **L** は **long double** 型を示します。接尾部 **df** または **DF** は **_Decimal32** 型を示します。接尾部 **dd** または **DD** は **_Decimal64** 型を示します。接尾部 **dl** または **DL** は **_Decimal128** 型を示します。大/小文字混合の 10 進浮動小数点の接尾部は無効です。例えば、dD および Df は無効な接尾部です。サフィックスを指定しないと、浮動小数点定数には、**double** 型が指定されます。

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

2 進浮動小数点リテラルの例を次に示します。

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

10 進浮動小数点データを使用する場合、10 進浮動小数点リテラルを使用するとより正確な結果が得られます。前述のとおり、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。以下のように初期化を行うとします。

- _Decimal64 rate = 0.1;

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

- _Decimal64 rate = 0.1dd;

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

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

関連参照
- 41 ページの『2 進浮動小数点変数』
- 42 ページの『10 進浮動小数点変数』
- 98 ページの『単項式』

パック 10 進数リテラル

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

パック 10 進数リテラルの形式は、次のとおりです。

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

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

文字リテラル

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

文字リテラルの形式は、次のとおりです。

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

C 文字リテラルは、**int** 型を持ちます。

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

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

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

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

改行文字は、¥n 改行エスケープ・シーケンスによって表すことができます。

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

文字リテラルの例を次に示します。

```
'a'
'¥''
L'0'
'('
```

関連参照

- 41 ページの『char および wchar_t 型指定子』

ストリング・リテラル

ストリング・リテラル には、二重引用符で囲まれた連続した文字またはエスケープ・シーケンスが含まれます。

接頭部 **L** のあるストリング・リテラルは、*ワイド・ストリング・リテラル* です。接頭部 **L** のないストリング・リテラルは、*通常の*、すなわち*狭幅ストリング・リテラル* です。

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

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

ストリング・リテラルの例を、次に示します。

```
char
titles[ ] = "Handel's ¥"Water Music¥"";
char *mail_addr = "Last Name    First Name    MI   Street Address ¥
   City     Province   Postal code ";
char *temp_string = "abc" "def" "ghi";  /* *temp_string = "abcdefghi¥0" */
wchar_t *wide_string = L"longstring";
```

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

ストリング・リテラル内に含まれている複数のスペースは保存されます。

次の行にストリングを継続するには、行継続文字 (¥ 記号) と、その後に続くオプションの空白文字と改行文字 (必須) を使用します。次の例では、ストリング・リテラル second が、コンパイル時のエラーを起こします。

```
char *first = "This string continues onto the next¥
  line, where it ends.";               /* compiles successfully.   */
char *second = "The comment makes the ¥ /* continuation symbol      */
  invisible to the compiler.";         /* compilation error.       */
```

連結

ストリングを継続する別の方法は、複数の連続ストリングを持つことです。隣接するストリング・リテラルを連結し、単一のストリングを作成します。ワイド・ストリング・リテラルと狭幅ストリング・リテラルが相互に隣接する場合、その結果の振る舞いについては未定義です。次の例は、このことを示しています。

```
"hello " "there"     /* is equivalent to "hello there"  */
"hello " L"there"    /* the behavior at the C89 language level is undefined */
"hello" "there"      /* is equivalent to "hellothere"                        */
```

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

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

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

関連参照

- 41 ページの『char および wchar_t 型指定子』
- 60 ページの『型修飾子』
- 33 ページの『static ストレージ・クラス指定子』

第 3 章 宣言

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

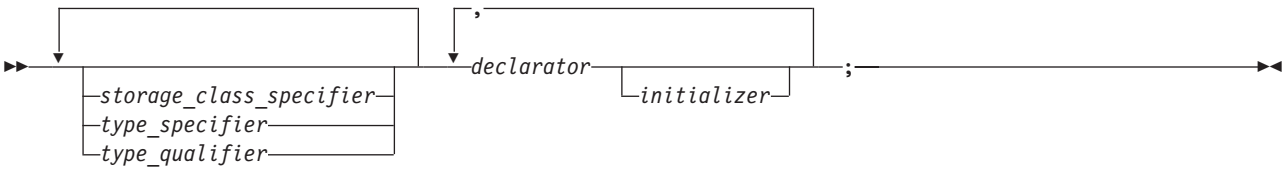
宣言によって、オブジェクトの内部関連属性 (ストレージ・クラス、型、スコープ、可視性、ストレージ期間、およびリンケージ) がさまざまな方法で決定されます。

宣言概要

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

データ・オブジェクトを宣言する際のエレメントの字句の順序は、次のとおりです。
- ストレージ期間およびリンケージ指定
- タイプ指定
- 宣言。ID を定義し、型修飾子およびストレージ修飾子を使用します
- 初期化指定子。ストレージを初期値で初期化します

すべてのデータ宣言の形式は、次のとおりです。



次の表は、宣言と定義の例を示しています。最初の列に宣言されている ID は、ストレージを割り当てません。これらの ID は、対応する定義を参照します。関数の場合は、対応する定義は、関数のコードまたは本体です。2 番目の列に宣言されている ID は、ストレージを割り当てます。これらの ID は、宣言と定義の両方になります。

宣言	宣言と定義
`extern double pi;`	`double pi = 3.14159265;`
`float square(float x);`	`float square(float x) { return x*x; }`

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

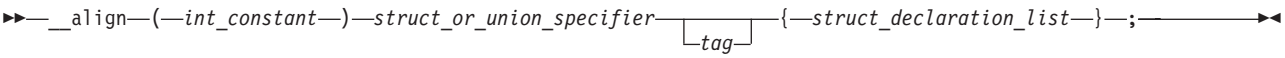
関連参照
- 65 ページの『第 4 章 宣言子』

__align 指定子

400 **__align** キーワードを使用すると、データ構造の明示的位置合わせを指定することができます。キーワードは、集合体型の定義または第 1 レベル変数の宣言において使用するための直交言語拡張です。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 指定子は、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、集合体またはクラス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。

構造体または共用体の構文は、以下のとおりです。



各値は、次のとおりです。
int_constant
　　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、1、2、4、8、または 16 です。
struct_or_union_specifier
　　構造体または共用体の指定子です。
struct_declaration_list
　　構造体宣言リストです。
tag　　構造体または共用体の ID です。

制約および制限事項

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** 指定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 指定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。

400 集合体の位置合わせは、**__align** で指定される境界上のメモリーに位置合わせされることを保証されません。これは、特定の i5/OS 型には特定の位置合わせが必要なためです。集合体がこれらのいずれかの型のメンバーを持つ場合、集合体の **__align** によって指定された位置合わせはそのメンバーが必要とする位置合わせに置き換えられます。例えば、16 バイトのポインターは、16 バイトの境界に位置合わせされる必要があります。

オブジェクト

オブジェクト は、値または値のグループを含むストレージの領域です。それぞれの値には、その ID を使用して、またはそのオブジェクトを参照する複雑な式を使用してアクセスできます。さらに、各オブジェクトには、固有の*データ型* が指定されています。オブジェクトの ID とデータ型は、両方とも、オブジェクトの*宣言* で確立されます。

オブジェクトのデータ型によって、そのオブジェクトの初期ストレージ割り当てと、以降のアクセスでの値の解釈が決まります。このデータ型は、型チェック演算でも使われます。

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

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

関連参照

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

ストレージ・クラス指定子

ストレージ・クラス指定子は、変数、関数、およびパラメーターの宣言を詳細化するために使用します。宣言で使用するストレージ・クラス指定子によって、次のことが決まります。

- オブジェクトに、内部リンケージまたは外部リンケージがあるか、またはリンケージがないか
- オブジェクトは、メモリーまたはレジスター (使用可能な場合) のどちらに格納されるか
- オブジェクトが、デフォルトの初期値 `0` または不確定デフォルトの初期値のどちらを受け取るか
- オブジェクトが、プログラム全体で参照されるか、または変数を定義した関数、ブロック、ソース・ファイル内でのみ参照されるか
- オブジェクトのストレージ期間は、*静的* (ストレージは、プログラムの実行中は保持される)、*自動* (オブジェクトが定義されているブロックが実行される間のみ、ストレージは保持される)、*スレッド* (ストレージは、現行のスレッドの実行中は保持される) のいずれかです。

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では十分でない場合、明示的なストレージ・クラス **auto**、**static**、**extern**、**register**、または **__thread** を指定することができます。C++ では、追加のオプションを使うことによって、クラス・データ・メンバーが **const** として宣言されたオブジェクトの一部であったとしても、そのメンバーにストレージ・クラス **mutable** を指定し、それを変更可能にすることができます。

関数の場合は、ストレージ・クラス指定子が関数のリンケージを決めます。唯一のオプションは、**extern** および **static** です。**extern** ストレージ・クラス指定子を使って宣言された関数は、外部結合を持ちます。これは、この関数が、他の変換単位から呼び出し可能であるということを意味します。**static** ストレージ・クラス指定子を使って宣言された関数は、内部結合を持ちます。これは、この関数が、関数が定義されている変換単位内でしか呼び出しできないことを意味します。関数のデフォルトは、外部結合です。

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

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

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

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

C および C++ のストレージ・クラス指定子は、次のとおりです。

- **auto**
- **extern**
- C++ **mutable**
- **register**
- **static**
- **__thread**
- **typedef**

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

auto ストレージ・クラス指定子

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

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

初期化

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

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

ストレージ期間

auto ストレージ・クラス指定子が指定されたオブジェクトには、自動ストレージ期間が指定されます。ブロックに入るたびに、そのブロックに定義された **auto** オブジェクトに対するストレージが使用可能になり

ます。ブロックが終了すると、そのオブジェクトは使用できなくなります。リンケージ指定がなく、**static** ストレージ・クラス指定子を使用しないで宣言されたオブジェクトは、自動ストレージ期間を持ちます。

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

リンケージ

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

関連参照
- 159 ページの『ブロック・ステートメント』
- 172 ページの『goto ステートメント』
- 136 ページの『関数宣言』

extern ストレージ・クラス指定子

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

extern 宣言は、関数の外側またはブロックの先頭に現れます。宣言が、関数を記述するか、関数の外側で現れて外部結合を持つオブジェクトを記述する場合は、キーワード **extern** はオプションです。ストレージ・クラス指定子を指定しない場合は、関数に外部結合があると想定されています。

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

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

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

初期化

extern ストレージ・クラス指定子を持つオブジェクトは、C のグローバル・スコープまたは C++ のネーム・スペースで初期化できます。 **extern** オブジェクトの初期化指定子は、以下のうちのいずれかにする必要があります。
- 定数式によって初期値が記述され、定義の一部として発生する。または、
- 静的ストレージ期間が指定された、すでに宣言済みのオブジェクトのアドレスに変換する。このオブジェクトは、ポインター演算によって変更することができます。 (言い換えると、オブジェクトは、整数定数式の加算または減算によって変更することができます。)

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

ストレージ期間

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

リンケージ

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

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

関連参照
- 7 ページの『外部結合』
- 6 ページの『内部結合』
- 33 ページの『static ストレージ・クラス指定子』
- 4 ページの『クラス・スコープ』
- 191 ページの『第 10 章 ネーム・スペース』
- 155 ページの『インライン関数』

mutable ストレージ・クラス指定子

C++ **mutable** ストレージ・クラス指定子は、クラス・データ・メンバーでのみ使用されます。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。

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

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

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

register ストレージ・クラス指定子

register ストレージ・クラス指定子は、オブジェクトの値をマシン・レジスターに常駐させるようにコンパイラーに指示します。コンパイラーは、この要求を満たす必要がありません。多くのシステムで使用できるレジスターのサイズと数は制限されているので、実際にレジスターに書き込まれる変数は少なくなります。コンパイラーが、マシン・レジスターを **register** オブジェクトに割り振らない場合、そのオブジェクトは、ストレージ・クラス指定子 **auto** を持っているものとして扱われます。 **register** ストレージ・クラス指定子を使用することで、ループ制御変数などのオブジェクトが頻繁に使用されているので、アクセス時間を最小化してパフォーマンスを上げるようプログラマーが望んでいることを知らせます。

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

初期化

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

ストレージ期間

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

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

リンケージ

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

C **制約事項**

- **register** ストレージ・クラス指定子は、ブロック内で宣言された変数についてのみ使用が許されます。これを、グローバルなスコープ・データ宣言で使用することはできません。
- register にはアドレスがありません。したがって、アドレス演算子 (**&**) を **register** 変数に適用することはできません。

C++ **制約事項**

- **register** ストレージ・クラス指定子は、ネーム・スペース・スコープでのデータ宣言には使用できません。
- C とは異なり、C++ では、**register** ストレージ・クラスが指定されているオブジェクトのアドレスを取ることができます。次に例を示します。

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

static ストレージ・クラス指定子

static ストレージ・クラス指定子を使用して、内部結合を持つオブジェクトまたは関数を定義することができます。つまり、特定の ID のそれぞれのインスタンスは、1 つのファイル内のみの同じオブジェクトま

たは関数を表します。さらに、**static** として宣言されたオブジェクトは、*静的ストレージ期間* を持ちます。つまり、これらのオブジェクトのメモリーは、プログラムの実行が開始したときに割り当てられ、プログラムが終了すると解放されます。

オブジェクトの静的ストレージ期間は、ファイル・スコープまたはグローバル・スコープと異なります。オブジェクトは静的期間を持つことができますが、ローカル・スコープは持てません。一方、**static** ストレージ・クラス指定子は、ファイル・スコープにある場合に限り、関数宣言で使用することができます。

static ストレージ・クラス指定子は、以下の名前だけに適用できます。
- オブジェクト
- 関数
- クラス・メンバー
- 無名共用体

以下のものは、**static** として宣言できません。
- 型宣言
- ブロック内の関数宣言
- 関数仮パラメーター

C キーワード **static** は、情報の隠蔽を強制するための C の主要な手段です。C++ は、情報の隠蔽をネーム・スペース言語フィーチャーとクラスのアクセス制御によって強制します。

C++ 外部変数のスコープを制限するためのキーワード **static** の使用は、ネーム・スペース・スコープ内のオブジェクトの宣言には推奨できません。

初期化

静的オブジェクトの初期化を、定数式、またはすでに **extern** または **static** と宣言されているオブジェクトのアドレスに変換する式 (多くは定数式によって修正される) によって行えます。**静的** (または外部) 変数を明示的に初期化しない場合は、その初期値は、該当する型のゼロになります。

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

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

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

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

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

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

リンケージ

static ストレージ・クラス指定子を含み、ファイル・スコープを持つオブジェクトまたはファイルの宣言は、ID の内部結合を示します。したがって、特定の ID のそれぞれのインスタンスは、1 つのファイル内のみの同じオブジェクトまたは関数を表します。

例

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

```
#include <stdio.h>

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

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

上記の例の出力は、以下のとおりです。

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

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

__thread ストレージ・クラス指定子

__thread で宣言されたオブジェクトは、*スレッド・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、スレッドが実行を開始するときに割り振られ、スレッドが終了するときに解放されます。

オブジェクトのスレッド・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。オブジェクトはスレッド期間を持ちますが、ローカル・スコープです。

__thread ストレージ・クラス指定子は、以下の名前のみに適用できます。

- ファイル・スコープで宣言されたオブジェクト
- extern または static ストレージ・クラス指定子を指定してブロック・スコープで宣言されたオブジェクト
- 静的クラス・メンバー
- 無名共用体

以下のものは、**__thread** として宣言できません。

- 型宣言
- 関数
- 関数仮パラメーター
- 非単純コンストラクター・クラスを持つクラス・オブジェクト

初期化

__thread オブジェクトの初期化を、定数式、またはすでに **__thread** と宣言されているオブジェクトのアドレスに変換する式 (多くは定数式によって修正される) によって行えます。**__thread** 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。

__thread 変数は、スレッドを実行する前に一度だけ初期化されます。

リンケージ

__thread ストレージ・クラス指定子は、オブジェクトのリンケージには影響を与えません。影響を与えるのは、オブジェクトのストレージ期間のみです。 **__thread** ストレージ・クラス指定子は、他のストレージ・クラス指定子と組み合わせて使用できるため固有です。他のストレージ・クラス指定子はオブジェクトのリンケージに影響を与え、**__thread** ストレージ・クラス指定子はオブジェクトのストレージ期間に影響を与えます。

例

__thread ストレージ指定子は、静的スコープを持つ変数でのみ使用でき、**extern** または **static** ストレージ・クラス指定子の直後に指定する必要があります。以下に、例を示します。

- 非ローカル変数

```
__thread int i;                    /* external linkage */
extern __thread struct state s;    /* external linkage */
static __thread char *p;           /* internal linkage */
```

- ローカル変数

```
int foo(void) {
  static __thread int tlsVal=0;    /* internal linkage */
  extern __thread int tlsVal2=1;   /* external linkage */
  return ++tlsVal;
}
```

- クラス・メンバー変数

```
class A {
  static __thread int tlsInstance;      /* internal linkage */
};
```

typedef

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

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

typedef 宣言の例

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

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

次の宣言は、上記の宣言と同じです。

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

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

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

そうすると、構造体 WEIGHT は、以下の宣言で使用できます。

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

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

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

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

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

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

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

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

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

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

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

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

型指定子

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

- 単純型指定子
- 列挙指定子
- **const** および **volatile** 修飾子

- C++ クラス指定子
- C++ 詳述型指定子

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

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

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

型名

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

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

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

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

以下の構造体は、型の命名規則の適用について説明しています。

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

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

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

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

互換型

C 互換型の概念は、変更することなく 2 つの型を一緒に使用できるという考え方 (代入式のように)、変更することなく一方から他方へ置換できるという考え方、およびそれらを複合型へ結び付けるという考え方を結合したものです。*複合型* は、2 つの互換型を結合した結果生まれるものです。合成された 2 つの互換型から生まれた結果の複合型を判別する方法は、整数型が算術演算と結合するときに使われる整数型の通常のバイナリー変換と似ています。

明らかに、同一の 2 つの型は互換性があり、それらの複合型は同じ型です。同一でない型、関数プロトタイプ、および型修飾型の型互換性を決定する法則は、あまり明らかではありません。 `typedef` 定義での名前は単なる型の同義語で、そのため `typedef` の名前が同じもの、ゆえに互換タイプを示す可能性があります。特定のプロパティーを持つポインター、関数、および配列もまた、互換タイプである可能性があります。

同一型

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

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

2 つの算術型は、それらが同じ型である場合に限って互換性があります。

別々にコンパイルされたソース・ファイル間の互換性

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

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

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

列挙型の場合、対応するメンバーも同じ値を持つこと。

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

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

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

単純型指定子

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

- 算術型
  - 整数型
    - C++ **bool**
    - **char**
    - **wchar_t**
    - 符号付き整数型
      - **signed char**
      - **short int**
      - **int**
      - **long int**
    - 符号なし整数型
      - **unsigned char**
      - **unsigned short int**
      - **unsigned int**
      - **unsigned long int**
  - 2 進浮動小数点型
    - **float**
    - **double**
    - **long double**
  - C 10 進浮動小数点型
    - **_Decimal32**
    - **_Decimal64**
    - **_Decimal128**
  - パック 10 進数
- **void**

ブール変数

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

同等、関連、および論理の各演算子の結果は、型 **bool** の結果になります。つまりブール定数 `true` または `false` のいずれになります。

型指定子 **bool** およびリテラル `true` および `false` を使用して、ブール論理テストを行います。*ブール論理テスト* は、論理演算の結果を表すために使用されます。次に例を示します。

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

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

char および wchar_t 型指定子

char 指定子の構文は、次のとおりです。

char 指定子は、整数型です。

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

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

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

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

char 型指定子の例

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

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

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

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

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

```
char *string_pointer;
```

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

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

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

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

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

2 進浮動小数点変数

2 進浮動小数点変数には、次の 3 つの型があります。

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

- **long double**

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

単純な 2 進浮動小数点宣言の宣言子は、ID です。単純な 2 進浮動小数点変数の初期化は、浮動定数、または、整数あるいは 2 進浮動小数点数に数値化される変数または式を使用して行います。変数のストレージ・クラスによって、変数を初期化する方法が決まります。

2 進浮動小数点データ型の例

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

```
double pi;
```

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

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

注: **f** 接尾部を 2 進浮動小数点リテラルに追加しない場合は、その数は、型 **double** になります。型 **float** のオブジェクトを、型 **double** のオブジェクトで初期化する場合、コンパイラーは暗黙的に、型 **double** のオブジェクトを型 **float** のオブジェクトに変換します。

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

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

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

```
extern long double maximum;
```

次の例では、配列 table を型 **double** の 20 個のエレメントで定義します。

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

関連参照
- 22 ページの『浮動小数点リテラル』
- 120 ページの『代入式』

10 進浮動小数点変数

400 C 10 進浮動小数点変数には、次の 3 つの型があります。

- **_Decimal32**
- **_Decimal64**
- **_Decimal128**

ビジネスおよび財務アプリケーションでは、数値データの計算は明示的に丸める以外は (丸め誤差がなく) 正確でなければなりません。また、正確な値になるように監査可能な詳細分析でサポートされている必要があります。このようなアプリケーションでは、計算によって生じる変換誤差を避ける方法が必要です。このような誤差を避ける最も効率的な方法は、10 進数演算機構を使用することです。 10 進浮動小数点は、パック 10 進および 2 進浮動小数点と類似しています。ただし、10 進浮動小数点には、パック 10 進数に極めて近い操作規則を使用して、(2 進浮動小数点と同様に) 大きな数を含めることができます。

C 言語の型階層内では、10 進浮動小数点型は、基本タイプ、実数型、および算術型です。 float 型、double 型、および long double 型は、総称浮動小数点型と呼ばれ、10 進浮動小数点型と同位です。総称浮動小数点型と 10 進浮動小数点型は、実数型です。

_Decimal32 データ型はストレージ・フォーマットのみに使用されます。Decimal32 データは直接計算されません。代わりに、_Decimal32 データ型は _Decimal64 データ型に変換されてから計算されます。また、計算結果が _Decimal32 の場合、その結果は _Decimal32 に戻されます。

10 進浮動小数点型にデフォルトの引数拡張はありません。 10 進浮動小数点データは、引数リストが省略符号で定義されている関数、またはプロトタイプを持たない関数に渡される場合、プロモートまたは拡張されません。

以下に、10 進浮動小数点型の有効および無効な使用方法を示します。また、他のデータ型との暗黙的および明示的な変換も示します。

```
_Decimal32 dfp32a = 200E-2df;  // ok
_Decimal64 dfp64a = 2.0dd;     // ok
_Decimal128 dfp128a = 2.0dl;   // ok
_Decimal32 dfp32b = 1.0;       // ok, implicit conversion of double
                               // to _Decimal32
_Decimal64 dfp64b = 1.0f;      // ok, implicit conversion of float
                               // to _Decimal64
_Decimal128 dfp128b = 1.0dL;   // error, mixed case suffix not valid

int i = 4;
double dbl = 4.0;

_Decimal64 multByTwo ( _Decimal64 );
void miscFunc ( ... );

dfp64a = dfp64a + dfp64b;      // ok
dfp32a++;                      // ok, dfp32a =
                               // (_Decimal32) ((_Decimal64)dfp32a+1.0dd)
dfp128a = dfp128b | 2.0dl;     // error, DFP data cannot be used with
                               // "|" operator
if (dfp64a>=dfp32a) dfp64a--;  // ok, _Decimal32 promoted to _Decimal64
                               // for compare
dfp128a = dfp64b - dfp32a;     // ok, _Decimal32 promoted to _Decimal64
                               // for subtraction, then implicit
                               // conversion to _Decimal128 for
                               // the assignment
dbl = dfp32b * i;              // ok, _Decimal32 converted to _Decimal64
                               // for computation, int promoted to
                               // _Decimal64, then implicit conversion
                               // to double for the assignment
dfp64a = dbl - dfp64b;         // error, double and _Decimal64 same
                               // rank, no promotion
dfp64a = (Decimal64)dbl - dfp64b;
                              // ok, explicit cast from double to
                              // _Decimal64
dfp128a += dbl;               // error, same as dfp128a = dfp128a + dbl
dfp64a = multByTwo(dfp32b);   // ok, _Decimal32 implicitly converted
                              // to _Decimal64
dfp64b = multByTwo(i);        // ok, int implicitly converted to
                              // _Decimal64
miscFunc(dfp32b);             // ok, _Decimal32 passed, no default
                              // argument promotion
miscFunc(dfp32a+dfp32b);      // ok, _Decimal32 vars converted to
                              // _Decimal64 for the add, then
                              // converted back to _Decimal32 for
                              // the call argument
```

関連参照

| • 22 ページの『浮動小数点リテラル』
| • 131 ページの『算術変換』

パック 10 進変数

400 *パック 10 進数* のデータ型は、ISO C89 および Standard C++ の拡張としてサポートされています。このデータ型は、大きな数値を正確に表すことができます。この形式は、整数部分、小数点、小数部分、および接尾部 D で構成されます。パック 10 進数型の変数では、有効数字を整数部分と小数部分を合わせて 63 桁まで指定できます。

パック 10 進数のデータ型の形式は、次のとおりです。

n は、最大値 63 の有効数字の総数を表します。*precision* は精度の桁数を表します。*precision* の値は *n* 以下でなければなりません。

例えば、次のような場合です。

```
decimal(4,2) x = 12.34D;     /* Requires #include <decimal.h>             */
_Decimal(4,2) y = 12.34d;    /* Declaration without including <decimal.h> */
```

「*ILE C/C++ Programmer's Guide*」の第 26 章『Using Packed Decimal Data in a C Program』を参照してください。

整変数

整変数は、次のカテゴリーに分かれます。

- 整数型
  - C++ **bool**
  - **char**
  - **wchar_t**
  - 符号付き整数型
    - **signed char**
    - **short int**
    - **int**
    - **long int**
  - 符号なし整数型
    - **unsigned char**
    - **unsigned short int**
    - **unsigned int**
    - **unsigned long int**

ビット・フィールドのデフォルトの整数型は、**unsigned** です。

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

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

単純な整数の定義または宣言の宣言子は、 ID です。単純整数の定義の初期化は、整数定数、または整数に割り当て可能な値に数値化される式を使用して行うことができます。変数のストレージ・クラスは、変数を初期化する方法を決めます。

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。

整数データ型の例

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

```
short int flag;
```

次の例では、**int** 変数である `result` を定義します。

```
int result;
```

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

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

void 型

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

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

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

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

void 型の例

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

注: C `find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));` の行の `sizeof` 演算子の使用は、配列のエレメントの数を決める標準的な方法です。

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

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

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

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

   return(0);
}
```

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

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

複合型

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

複合型を使用するのは、以下のいずれかを作成するときです。

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

構造体

構造体 は、データ・オブジェクトの順序付けられたグループから構成されます。配列の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。

C++ C++ では、構造体メンバーは完了型である必要があります。

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

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

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

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

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

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

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

ドット演算子 (.) 付きの構造変数の名前か、矢印演算子 (->) 付きのポインターとメンバー名を指定することによって、構造体のメンバーを参照します。例えば、次の例

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

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

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

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

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

互換構造体

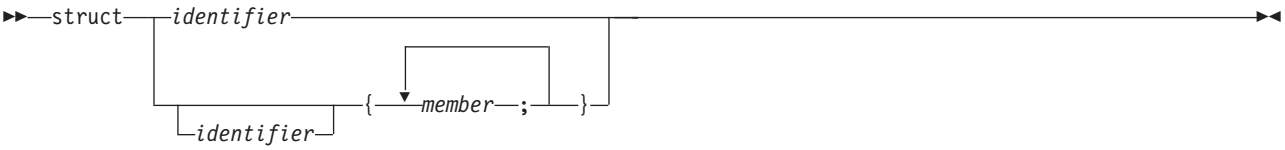
C 構造体の定義ごとに、同じソース・ファイルにある他の構造体型とは異なる、または互換性のない新規の構造体型を作成します。ただし、すでに定義された構造体型への参照である型指定子は同じ型です。構造体タグが、参照と定義を関連付け、型名として効果的に機能します。つまり、以下の例では構造体型 j および k だけが同じ型になります。

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

構造体の宣言と定義: *構造体型の定義* では、構造体の一部になっているメンバーを記述します。構造体型の定義には、**struct** キーワードと、その後に続くオプションの ID (構造体タグ) および中括弧で囲まれたメンバーのリストが含まれています。

構造データ型の宣言は、次の形式を持ちます。

型指定子



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

構造体宣言 は、宣言が中括弧で囲まれたメンバーのリストを持っていないということを除けば、構造体定義と同じ形式です。*構造体定義* は、構造体データ型の宣言と同じ形式を持ちますが、セミコロンで終了します。

構造体メンバーの定義

メンバーのリストで、構造体に保管できる値の説明と構造体データ型が提供されます。 C では、構造体メンバーは、「関数からの戻り　T」(型によっては T)、不完全型、 および void を除く、任意の型をとります。不完全型は構造体メンバーとして許可されないため、構造体型はメンバーとしてそれ自身のインスタンスを含むことはできませんが、それ自身のインスタンスを指すポインターを含むことはできます。

構造体メンバーの定義は、変数宣言の形式になっています。構造体メンバーの名前は、単一構造体内で固有でなければなりませんが、同じメンバー名は、同じスコープ内で定義された別の構造体型で使用でき、さらに、変数、関数、または型名と同じ名前であっても構いません。ビット・フィールドを表さないメンバーは、任意のデータ型にすることができ、型修飾子 **volatile** または **const** のいずれかの型で修飾することができます。結果は左辺値です。ただし、型修飾子のないビット・フィールドは、構造体メンバーとして宣言することができます。ビット・フィールドは、名前がない場合は初期化には関与せず、初期化が終わると不定値を持ちます。

C 構造体メンバーはメモリー・アドレスに昇順に割り当てられます。最初のコンポーネントは、構造体名そのものの先頭アドレスから始まります。 コンポーネントの適切な位置合わせを可能にするために、ホールまたは埋め込みが構造体レイアウト内の連続メンバー間に置くことができます。

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

C++ キーワード **struct** は、C++ ではオプションです。

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

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

C 構造体または任意の集合体型の automatic 変数の初期化指定子は、定数式でなければなりません。

例

次の定義では、完全に初期化される構造体を示します。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                };
static struct address perm_address =
                { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

perm_address の値は、次のとおりです。

メンバー	値
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

次の定義では、一部のみが初期化される構造体を示します。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                };
struct address temp_address =
                { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

temp_address の値は、次のとおりです。

メンバー	値
temp_address.street_no	44
temp_address.street_name	ストリング "Knyvet Ave." のアドレス
temp_address.city	ストリング "Hamilton" のアドレス
temp_address.prov	ストリング "Ontario" のアドレス
temp_address.postal_code	ストレージ・クラスによって異なる値

注: temp_address.postal_code のような初期化されない構造体メンバーの初期値は、メンバーに関連付けられたストレージ・クラスによって異なります。

同じステートメントでの構造体型および変数の宣言: C 1 つのステートメントで、構造体型および構造変数を定義するには、型定義の後に、宣言子とオプションの初期化指定子を置きます。変数のストレージ・クラス指定子を指定するには、ステートメントの先頭にストレージ・クラス指定子を入れる必要があります。

次に例を示します。

```
static struct {
                  int street_no;
                  char *street_name;
```

```
                char *city;
                char *prov;
                char *postal_code;
            } perm_address, temp_address;
```

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

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

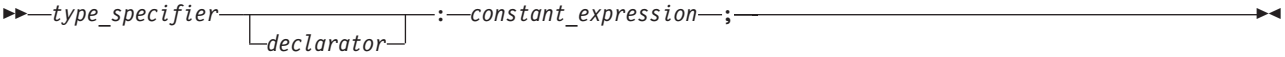
次に例を示します。

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

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

構造体でのビット・フィールドの宣言および使用: C および C++ では、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・フィールドを宣言する構文は、以下のとおりです。



ビット・フィールド宣言には、型指定子と、その後に続く、オプションの宣言子、コロン、フィールド幅をビット単位で示す定数整数式、およびセミコロンが含まれます。ビット・フィールド宣言では、型修飾子 **const** または **volatile** のいずれかを使用することができません。

C++ C++ は、ビット・フィールドの許容型のリストを整数型または列挙型を含むように拡張しています。

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

長さ 0 のビット・フィールドは、名前なしのビット・フィールドにする必要があります。名前なしのビット・フィールドは、参照または初期化することはできません。幅がゼロのビット・フィールドがあると、次のフィールドは次のコンテナー境界に位置合わせされ、そこではコンテナーは、ビット・フィールドの基礎となる型と同じサイズになります。

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

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

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

ビット・フィールドの位置合わせ

一連のビット・フィールドが **int** のサイズいっぱいにならない場合は、埋め込みが行われます。埋め込みの量は、構造体メンバーの位置合わせ特性によって決定されます。

次の例は、埋め込みの例を示します。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)	1 ビット
clock	1 ビット
(埋め込み — 23 ビット)	次の **int** 境界まで (名前なしフィールド)
flag	1 ビット
(埋め込み — 31 ビット)	次の **int** 境界まで

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

次の式では、light フィールドを 1 にセットします。

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

ビット・フィールドに範囲外の値を割り当てると、ビット・パターンは保存され、適切なビットが割り当てられます。次の式では、kitchen 構造体の toaster フィールドに 0 を割り当てます。これは、最下位桁のビットのみが toaster フィールドに割り当てられるからです。

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

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

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

#include <stdio.h>

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

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

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

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

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

   return(0);
}
```

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

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

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

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

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

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

では、ポインターをリスト内の次のオブジェクトに進めます。

関連参照
- 64 ページの『不完全型』

共用体

共用体 は、そのすべてのメンバーが、メモリー内の同じ場所から開始するということを除けは、構造体に類似しているオブジェクトです。共用体は、一度に、そのメンバーのうちの 1 つの値しか含むことができません。 static ストレージを持つ共用体のデフォルトの初期化指定子は最初のコンポーネントのデフォルトであり、automatic ストレージを持つ共用体には何も含まれません。

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

C 共用体の最初のメンバーのみを初期化できます。

互換共用体

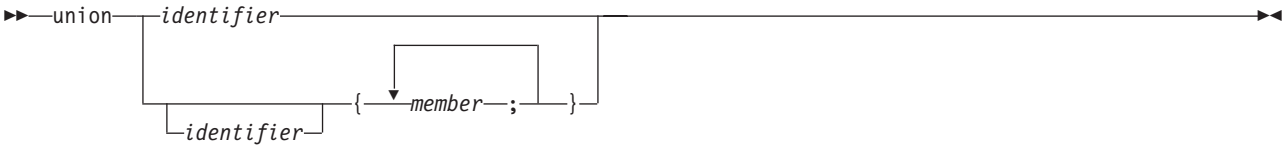
C 共用体の定義ごとに、同じソース・ファイルにある他の共用体型とは異なるか、または互換性のない新規の構造体型を作成します。ただし、すでに定義された共用体型への参照である型指定子は同じ型です。共用体タグが、参照と定義を関連付け、型名として効果的に機能します。

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

C++ は、共用体メンバーの許容データ型に制限を追加します。C++ の共用体メンバーは、コンストラクター、デストラクター、または多重定義コピー代入演算子を持つクラス・オブジェクトにすることはできません。共用体メンバーは、**static** というキーワードでは宣言できません。

共用体の宣言: *共用体型の定義* には、**union** キーワードと、その後に続く、オプションの ID (タグ) および中括弧で囲まれたメンバーのリストが含まれます。

共用体定義の形式は、次のとおりです。



共用体宣言 は、宣言が中括弧で囲まれたメンバーのリストを持っていないということを除けば、共用体定義と同じ形式です。

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

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

共用体メンバー定義の形式は、変数宣言の形式と同じです。

共用体のメンバーは、構造体のメンバーの場合と同じ方法で参照できます。

次に例を示します。

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

people.birthday[0] = '¥n';
```

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

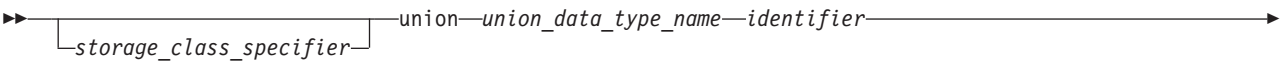
共用体は、一度に、そのメンバーのうちの 1 つのみしか表すことができません。この例では、共用体 people には、age、birthday、または weight のいずれか 1 つを含むことができますが、2 つ以上を含むことはできません。次の例では、printf ステートメントの結果は正しくありません。これは、people.age が、1 行目で people.birthday に代入された値を置換するからです。

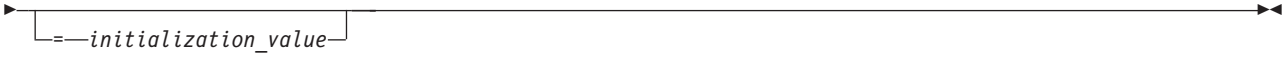
```
#include <stdio.h>
#include <string.h>

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

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

上記の例の出力は、次の出力に似ています。

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

共用体変数の定義: C 共用体変数定義の形式は、次のとおりです。





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

共用体の最初のメンバーのみを初期化できます。

次の例では、共用体変数 people の最初の共用体メンバーである birthday を初期化する方法を示します。

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = {"23/07/57"};
```

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

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

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

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

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

C++ 無名共用体は、protected メンバーまたは private メンバーを持つことはできません。グローバルまたはネーム・スペース無名共用体は、キーワード **static** を使って宣言される必要があります。

共用体の例: 次の例では、共用体データ型 (名前なし) と共用体変数 (名前 length 付き) を定義します。length のメンバーは、**long int**、**float**、または **double** のいずれかでかまいません。

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

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

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

次のステートメントでは、1 つの文字を input に代入します。

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

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

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

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

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

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

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

列挙型

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

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

互換列挙型

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

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

列挙データ型の宣言: 列挙型宣言には、**enum** キーワードと、その後に続くオプションの ID (*列挙型タグ*) および中括弧で囲まれた列挙子のリストが含まれます。コンマは、列挙子のリスト内で各列挙子を分離します。列挙型の宣言形式は、以下のとおりです。

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

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

列挙子の形式は、次のとおりです。

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

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

定数の値は、次の方法で判別されます。

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

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

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

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

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

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

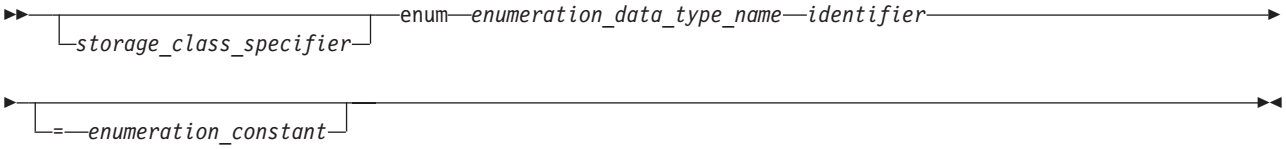
```
enum grain { oats, wheat, barley, corn, rice };
   /*         0      1      2      3     4         */

enum grain { oats=1, wheat, barley, corn, rice };
```

```
   /*         1        2      3      4     5       */

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

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

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

列挙変数の定義: 列挙型変数定義の形式は、次のとおりです。



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

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

次の例の 1 行目は、列挙型 grain を宣言します。 2 行目は、変数 g_food を定義し、g_food に初期値 barley (2) を指定します。

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

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

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

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

列挙型および列挙オブジェクトの定義: 型定義の後に、宣言子とオプションの初期化指定子を使用することによって、型と変数を 1 つのステートメントに定義できます。変数のストレージ・クラス指定子を指定するには、ストレージ・クラス指定子を宣言の先頭に入れる必要があります。次に例を示します。

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

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

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

これらの例のいずれも、次の 2 つの宣言と同じです。

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

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

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

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

この例では、変数 weekday を定義します。この変数には、指定した任意の列挙型定数を割り当てることができます。

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

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

#include <stdio.h>

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

void french(enum days);

int main(void)
{
   int num;

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

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

型修飾子

C は、C++ が *cv 修飾子* として参照する型修飾子 **const** および **volatile** を認識します。いずれの言語においても、cv 修飾子は左辺値の式でのみ意味があります。C++ では、C では許可されない関数に cv 修飾子を適用することができます。

const および volatile キーワードの構文

volatile または **const** ポインターの場合は、* と ID の間にキーワードを入れる必要があります。次に例を示します。

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

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

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

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

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

```
int * const y
```

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

```
const int * y
```

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

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

上記の例では、次の例のストレージと同じストレージを提供します。

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

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

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

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

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

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

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

宣言に複数の修飾子を入れることができますが、宣言に同じ修飾子を複数回指定することはできません。

const 型修飾子

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

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

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

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

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

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

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

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

C++ では、外部で定義された定数を参照するものを除き、すべての const 宣言に初期化指定子が必要です。

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

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、const オブジェクトはデフォルトで内部結合を持つため、const オブジェクトをヘッダー・ファイルで定義することができます。

const ポインター

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

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

オブジェクトを const となるよう宣言することは、this ポインターが const オブジェクトを指すポインターであることを意味します。const this ポインターは、const メンバー関数を使うときのみに使用できます。

const メンバー関数

メンバー関数 **const** を宣言することは、this ポインターが const オブジェクトを指すポインターであることを意味します。クラスのデータ・メンバーは、関数内で const になります。その場合でも、関数はその値を変更することができますが、そのためには、次のように const_cast が必要です。

```
void foo::p() const{
   member = 1;                        // illegal
   const_cast <int&> (member) = 1;    // a bad practice but legal
}
```

もっといい方法は、member **mutable** を宣言することです。

volatile 型修飾子

volatile 修飾子は、データ・オブジェクトに対するメモリー・アクセスの整合性を維持します。揮発性オブジェクトは、それらの値が必要になるたびにメモリーから読み取られ、変更されるたびにメモリーへ書き戻されます。 **volatile** 修飾子は、コンパイラーの制御または検出以外の方法 (システム・クロックによって更新された変数など) で、値を変更できるデータ・オブジェクトを宣言します。その結果、コンパイラーは、オブジェクトを参照するコードに対して特定の最適化を適用しないよう通知されます。

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

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

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

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

ILE 型修飾子

400 C および C++ には言語拡張として次の型修飾子が導入されており、i5/OS オペレーティング・システム固有のニーズに対応します。

詳細については、「*ILE C/C++ Programmer's Guide*」の第 22 章『Using Pointers in a Program』および第 29 章『Using Teraspace in ILE C and C++ Programs』を参照してください。

__ptr128 修飾子

400 キーワード **__ptr128** は、サイズを 16 バイトおよび 16 バイトの位置合わせに明示的に指定するためにポインター型に適用できる修飾子です。この言語拡張は、アプリケーションを i5/OS オペレーティング・システムに容易に移植するために、つまり、アプリケーションがテラスペース・ストレージ・モデルおよびランタイム環境を利用できるように提供されています。

__ptr128 修飾子は、8 バイトで特別の位置合わせ要件がないプロセス・ローカル・ポインターではなく、従来のポインターを指定します。プロセス・ローカル・ポインターは、テラスペース・ストレージにアクセスするために使用します。 **__ptr128** キーワードは、ポインター宣言の宣言子部分の、C++ cv 修飾子を指定できるところに指定できます。

例えば、次のような場合です。

```
int * __ptr128 p;
```

p を int に対する 16 バイト・ポインターとして宣言します。

```
int * __ptr128 const r;
```

これは r を、const の 16 バイト・ポインターであると宣言しています。

__ptr64 修飾子

400 キーワード **__ptr64** は、ポインター型に適用して、そのサイズを 8 バイトに制限することができる修飾子です。8 バイトのポインターは、*プロセス・ローカル・ポインター* と呼ばれ、テラスペース・ストレージのアクセスに使用されます。この言語拡張は、アプリケーションを i5/OS オペレーティング・システムに容易に移植するために、つまり、アプリケーションがテラスペース・ストレージ・モデルおよびランタイム環境を利用できるように提供されています。 ILE C/C++ コンパイラーは、割り当て演算またはパラメーター結合時に、8 バイトと 16 バイトのポインター間で、あるレベルの変換を暗黙的に実行します。

__ptr64 修飾子は、8 バイトで特別の位置合わせ要件がないプロセス・ローカル・ポインターを指定します。 **__ptr64** キーワードは、ポインター宣言の宣言子部分の、C++ cv 修飾子を指定できるところに指定できます。

---

asm 宣言

キーワード **asm** は、アセンブリー・コードを表します。このインプリメンテーションでは、コンパイラーは、宣言内のキーワード **asm** を認識して無視します。

不完全型

以下は、不完全型です。

- 型 **void**
- 不明サイズの配列
- 不完全型エレメントの配列
- 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

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

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 */
```

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

第 4 章 宣言子

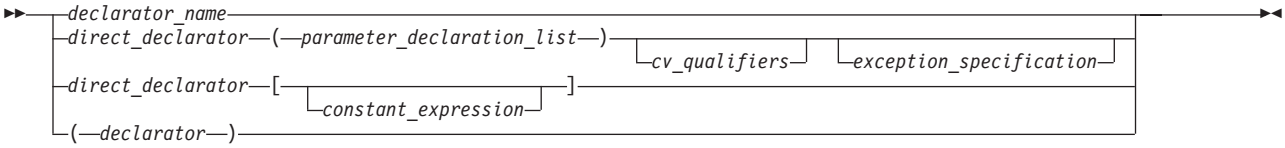
宣言子 は、データ・オブジェクトまたは関数を指定します。宣言子は、多くのデータ定義と宣言および一部の型定義で使用します。

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

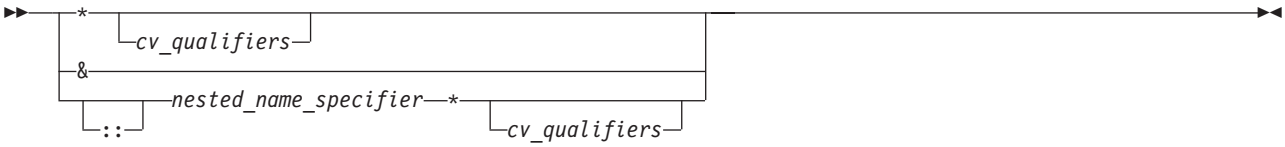
宣言子の形式は、次のとおりです。

宣言子 (declarator)

direct_declarator

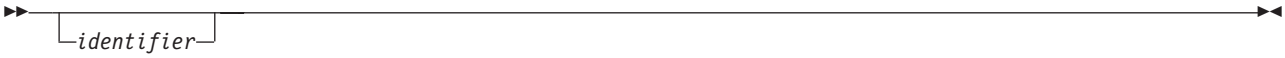


pointer_operator



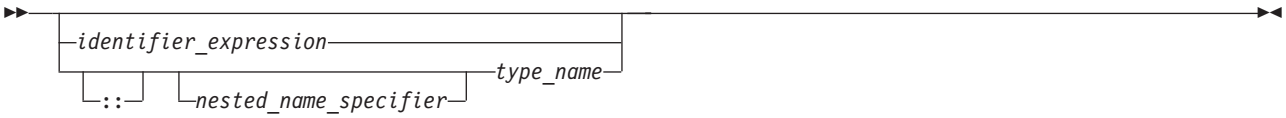
C C における宣言子名の構文

declarator_id



C++ C++ における宣言子名の構文

declarator_id



宣言子構文に関する注意

- *cv_qualifiers* 変数は、**const** および **volatile** のいずれか、またはその両方を組み合わせたものを表します。 C では、**volatile** または **const** 関数は、宣言または定義できません。ただし、C++ では、非静的メンバー関数を cv 修飾子の **const** または **volatile** で修飾することができます。
- C++ 変数 *exception_specification* と *nested_name_specifier*、およびスコープ解決演算子 **::** は、C++ においてのみ使用可能です。
- C++ *identifier_expression* は、修飾 ID または非修飾 ID のいずれにもなれます。スコープ解決演算子、テンプレート、およびその他の拡張機能は、C には存在しません。
- C++ *nested_name_specifier* は、修飾された ID 式です。

次の表に、いくつかの宣言子の例を示します。

例	説明
`int owner`	`owner` は、**int** データ・オブジェクトです。
`int *node`	`node` は、**int** データ・オブジェクトを指すポインターです。
`int names[126]`	`names` は、126 個の **int** エレメントの配列です。
`int *action( )`	`action` は、**int** を指すポインターを戻す関数です。
`volatile int min`	`min` は、**volatile** 修飾子がある **int** です。
`int * volatile volume`	`volume` は、**int** を指す **volatile** ポインターです。
`volatile int * next`	`next` は、**volatile int** を指すポインターです。
`volatile int * sequence[5]`	`sequence` は、**volatile int** オブジェクトを指す 5 つのポインターの配列です。
`extern const volatile int clock`	`clock` は、静的ストレージ期間および外部結合が指定された定数および volatile 整数です。

初期化指定子

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

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

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

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

```
int group = 3;
```

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

中括弧で囲まれた初期化リストを使用して初期化された配列、構造体、または共用体では、初期化されないメンバーまたはサブスクリプトを適切な型のゼロに暗黙で初期化します。

例

次の例では、配列 grid の最初の 8 つのエレメントのみが、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、明示的にゼロに初期化されたかのように初期化されます。

```
static short grid[3] [4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	0	grid[1] [2]	1
grid[0] [1]	0	grid[1] [3]	1
grid[0] [2]	0	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	0	grid[2] [2]	0
grid[1] [1]	0	grid[2] [3]	0

C++ では、ネーム・スペース・スコープで定数以外の式を指定して、変数を初期化できます。 C では、グローバル・スコープでこれと同じことを行うことはできません。 C または C++ のいずれかで、**__thread** 変数は、定数以外の式で初期化することはできません。

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

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

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

skiplabel: i = 4;
```

外部、静的、および自動定義内のクラスを初期化できます。初期化指定子には、= (等号) と、その後に続く、中括弧で囲まれ、コンマで分離された値のリストが含まれます。クラスのすべてのメンバーを初期化する必要はありません。

ポインター

ポインター 型変数は、データ・オブジェクトまたは関数のアドレスを保持します。ポインターは、1 つのデータ型のオブジェクトを参照できます。ビット・フィールドまたは参照は参照できません。 ポインターは、一度に 1 つの値しか保持できないという意味で、スカラー型として分類されます。

ポインターに共通な使用を次に示します。
- リンクされたリスト、ツリー、キューなどの動的データ構造にアクセスする。
- 配列のエレメント、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。

C このセクションでのここから先の説明は、C だけに適用されます。

ポインターを使用して、**register** ストレージ・クラス指定子で宣言される参照オブジェクトを指すことはできません。

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

ポインターの宣言

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

```
long *pcoat;
```

キーワード **volatile** が * の前に現れる場合は、宣言子は、**volatile** オブジェクトを指すポインターを記述します。キーワード **volatile** が、* と ID の間にある場合は、宣言子は **volatile** ポインターを記述します。キーワード **const** は、**volatile** キーワードと同じように機能します。次の例では、pvolt は、**short** 型が指定されたオブジェクトを指す定数ポインターです。

```
extern short * const pvolt;
```

次の例では、pnut を、**volatile** 修飾子が指定された **int** オブジェクトを指すポインターとして宣言します。

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

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

```
float * volatile psoup;
```

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

```
enum bird *pfowl;
```

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

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

ポインターの割り当て

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

次の例では、代入演算用の互換性がある宣言を示します。

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

次の例では、代入演算用の互換性がない宣言を示します。

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

ポインターの初期化

初期化指定子は、= (等号) と、その後に続く、ポインターに入れられるアドレスを表す式によって表されます。次の例では、変数 time と speed には **double** 型、amount には **double** を指す型ポインターが指定されるように定義します。この例では、ポインター amount が total を指すように初期化が行われています。

```
double total, speed, *amount = &total;
```

コンパイラーは、サブスクリプトがない配列名を、配列の 1 番目のエレメントを指すポインターに変換します。配列の名前を指定することによって、配列の 1 番目のエレメントのアドレスをポインターに割り当てることができます。次の 2 つの定義のセットは、同じです。定義は両方とも、ポインター student を定義し、student を section の 1 番目のエレメントのアドレスに初期化します。

```
int section[80];
int *student = section;
```

は、以下と同等です。

```
int section[80];
int *student = &section[0];
```

初期化指定子のストリング定数を指定することによって、ストリング定数の 1 番目の文字のアドレスをポインターに割り当てることができます。

次の例では、ポインター変数 string とストリング定数 "abcd" を定義します。ポインター string は、ストリング "abcd" の文字 a を指すように初期化されます。

```
char *string = "abcd";
```

次の例では、weekdays を、ストリング定数を指すポインターの配列として定義します。各エレメントは、異なるストリングを指します。例えば、ポインター weekdays[2] は、ストリング "Tuesday" を指します。

```
static char *weekdays[ ] =
      {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
      };
```

ポインターは、0 に数値化される整数定数式 (例えば、char * a=0;) を使用して、ヌルに初期化することもできます。このようなポインターは、*ヌル・ポインター* です。このヌル・ポインターは、オブジェクトを指しません。

ポインターの使用

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

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

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

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

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

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

ポインター演算

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

- 増分と減分
- 加算および減算
- 比較
- 代入

増分 (++) 演算子は、ポインターが参照するデータ・オブジェクトのサイズによって、ポインターの値を増やします。例えば、ポインターが配列の 2 番目のエレメントを参照している場合は、++ によって、ポインターに、配列の 3 番目のエレメントを参照させます。

減分 (--) 演算子は、ポインターが参照するデータ・オブジェクトのサイズによって、ポインターの値を減らします。例えば、ポインターが配列の 2 番目のエレメントを参照している場合は、-- によって、ポインターに、配列の 1 番目のエレメントを参照させます。

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

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

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

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

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

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

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

ポインターを使用したプログラムの例

次のプログラムには、ポインター配列が含まれています。

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

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

#define  SIZE  20

int main(void)
```

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

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



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

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

このプログラムとの対話により、次のセッションが作成されます。

```
出力          Enter name to be searched.

入力          Mark

出力          name Mark found
```

または

```
出力          Enter name to be searched.

入力          Deborah

出力          name Deborah not found
```

配列

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

C および C++ 言語には、個々のエレメントの読み取りおよび書き込みができる、配列型用の限定された組み込みサポートがあります。ある配列を別の配列に割り当てたり、2 つの配列を同じかどうか比較し、自己認識サイズを戻したりする操作は、いずれの言語でもサポートされていません。

配列型は、特定型のオブジェクトのセット用に隣接して割り振られたメモリーを表します。配列型は、いわゆる *配列型派生* 内で、エレメントの型から派生します。配列オブジェクトの型が不完全な場合、配列型も不完全であると見なされます。

配列エレメントは、**void** 型にも関数型にもすることができません。ただし、関数へのポインター配列は許可されます。C++ では、配列エレメントは参照型にも抽象クラス型にもすることができません。

C 同じように修飾される 2 つの配列型は、それらのエレメントの型に互換性があれば、互換性があります。次に例を示します。

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

これらには、互換性はありません。2 つの互換配列型から成る複合型は、複合エレメント型を持つ配列です。元の型のサイズが両方とも既知である場合は、両方のサイズが同じでなければなりません。

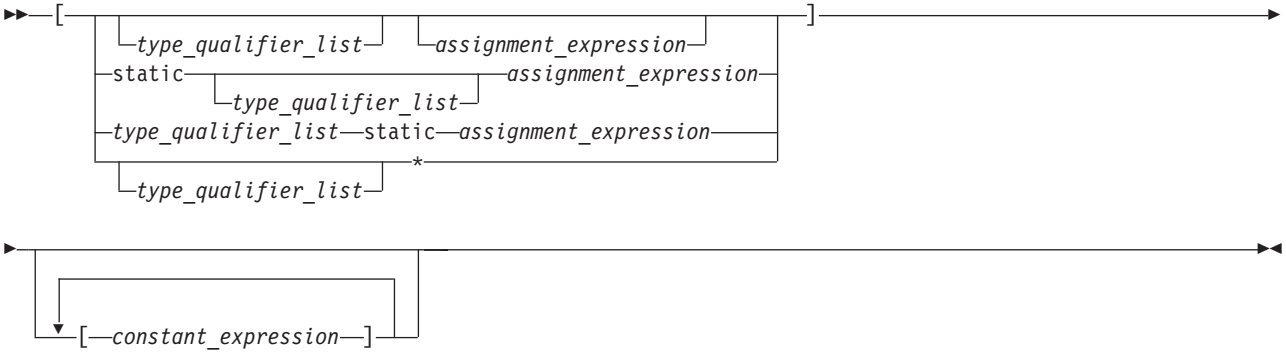
特定のコンテキスト内を除いて、サブスクリプトされていない配列名 (例えば、`region[4]` の代わりに `region`) は、配列がすでに宣言されている場合、配列の最初のエレメントのアドレスを値に持つポインターを表します。例外は、配列名が配列自身を渡す場合です。例えば、配列名は、それが **sizeof** 演算子またはアドレス (**&**) 演算子のオペランドであるときに、配列全体を渡します。

同様に、関数のパラメーター・リスト内の配列型は、対応するポインター型に変換されます。引数配列のサイズ情報は、配列が関数本体内部からアクセスされるときに失われます。

配列の宣言

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

サブスクリプト宣言子の形式は、次のとおりです。



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

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

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

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

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

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

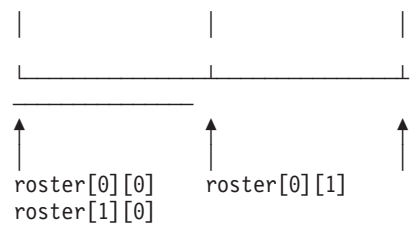
次の例では、**int** 型の 6 つのエレメントを含む 2 次配列を定義します。

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

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

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

ストレージ内では、roster のエレメントは、以下のように保管されます。



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

- 初期化を含む配列定義
- **extern** 宣言
- パラメーター宣言

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

配列の初期化

配列の初期化指定子は、中括弧 ({ }) で囲まれた定数式の、コンマで分離されたリストです。初期化指定子には、等号 (=) が前に付いています。配列内のすべてのエレメントを初期化する必要はありません。配列が部分的に初期化されている場合は、初期化されていないエレメントの値は、該当の型の値 0 となります。静的ストレージ期間を持つ配列のエレメントについても、同じことが言えます。(静的ストレージ期間を持つのは、**static** キーワードを指定して宣言されているすべてのファイル・スコープ変数および関数スコープ変数です。)

次の定義では、完全に初期化される 1 次元配列を示します。

```
static int number[3] = { 5, 7, 2 };
```

配列 number には、次の値が含まれます。number[0] は 5、number[1] は 7、number[2] は 2 です。サブスクリプト宣言子内の式がエレメント数 (この場合は 3) を定義している場合、配列内のエレメント数よりも多くの初期化指定子を持つことはできません。

初期化指定子

次の定義では、部分的に初期化される 1 次元配列を示します。

```
static int number1[3] = { 5, 7 };
```

number1 の値に関して、number1[0] および number1[1] は前の定義と同じですが、number1[2] は 0 です。

サブスクリプト宣言子の式でエレメントの数を定義する代わりに、次の 1 次元配列定義では、指定された各初期化指定子に対して 1 つのエレメントを定義します。

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

サイズが指定されていなくて、初期化エレメントが 5 つあるため、コンパイラーは item に次の 5 つの初期化されたエレメントを指定します。

次のものを指定して、1 次元の文字配列を初期化できます。
- 中括弧で囲まれ、コンマで分離された定数のリスト。各定数は、文字に含めることができます。
- ストリング定数 (定数を囲む中括弧はオプション)

ストリング定数を初期化すると、空いている場所がある場合または配列の次元が指定されていない場合は、ヌル文字 (¥0) をストリングの終わりに入れます。

以下の定義は、文字配列の初期化を示します。

```
static char name1[ ] = { 'J', 'a', 'n' };
static char name2[ ] = { "Jan" };
static char name3[4] = "Jan";
```

以下の定義では、次のエレメントを作成します。

エレメント	値	エレメント	値	エレメント	値
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	¥0	name3[3]	¥0

次の定義では、ヌル文字はなくなります。

```
static char name3[3]="Jan";
```

C++ 文字の配列をストリングで初期化する場合、ストリング内の文字数 (終端の「¥0」を含む) は、配列の中のエレメント数を超えてはなりません。

次の方法を使用して、多次元配列を初期化できます。
- 初期化するすべてのエレメントの値を、コンパイラーが値を割り当てる順序でリストする。コンパイラーは、最後の次元のサブスクリプトを最も速く増やして、値を割り当てます。この形式の多次元配列の初期化は、1 次元配列の初期化と似ています。次の定義では、配列 month_days を完全に初期化します。

```
static month_days[2][12] =
{
 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

- 中括弧を使用して、初期化するエレメントの値をグループ化します。各エレメントかエレメントのネスト・レベルを中括弧で囲むことができます。次の定義には、第 1 次元の 2 つのエレメントが含まれます (これらのエレメントは、行と見なすことができます)。初期化には、これらの 2 つの各エレメントを囲む中括弧が含まれます。

```
static int month_days[2][12] =
{
 { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
 { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
```

- ネストされた中括弧を使用して、次元および次元のエレメントを選択的に初期化する。

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

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

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

エレメント	値	エレメント	値
matrix[0][0]	1	matrix[1][2]	0
matrix[0][1]	2	matrix[1][3]	0
matrix[0][2]	0	matrix[2][0]	5
matrix[0][3]	0	matrix[2][1]	6
matrix[1][0]	3	matrix[2][2]	0
matrix[1][1]	4	matrix[2][3]	0

配列を使用したプログラムの例

次のプログラムでは、prices と呼ばれる浮動小数点配列を定義します。

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

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

#include <stdio.h>
#define  ARR_SIZE  5

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

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

  printf("¥n");

  for (i = 0; i < ARR_SIZE; i++)
  {
```

```
    total = prices[i] * 1.05;
    printf("total = $%.2f¥n", total);
  }

  return(0);
}
```

このプログラムの出力は次のようになります。

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

total = $1.48
total = $1.58
total = $3.94
total = $5.25
total = $0.90
```

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

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

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

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

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

   return(0);
}
```

このプログラムの出力は次のようになります。

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

```
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140
```

関数指定子

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

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

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

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

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

参照

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

参照型は、型指定子の後に 参照宣言子 `&` を入れることによって定義できます。関数パラメーターを除く参照はすべて、定義するときに初期化する必要があります。参照は一度定義すると再割り当てできません。参照を再割り当てしようとすると、ターゲットへ新しい値が割り当てられます。

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

次に例を示します。

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

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

NULL への参照は認められていません。

参照の初期化

参照を初期化するために使用するオブジェクトは、参照と同じ型にする必要があります。同じ型でなければ、参照型に型変換できる型でなければなりません。型変換が必要なオブジェクトを使用して定数への参照を初期化する場合は、一時オブジェクトを作成します。次の例では、型 **float** の一時オブジェクトを作成します。

```
int i;
const float& f = i; // reference to a constant float
```

オブジェクトを使用して参照を初期化する場合、その参照をそのオブジェクトにバインド します。

型変換が必要なオブジェクトを使用して、定数以外の参照を初期化しようとすると、エラーになります。

参照は初期化されると、別のオブジェクトを参照するように変更することはできません。次に例を示します。

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;          // valid
int &RefOne = num2;          // error, two definitions of RefOne
RefOne = num2;                      // assign num2 to num1
int &RefTwo;                 // error, uninitialized reference
int &RefTwo = num2;          // valid
```

参照の初期化は、参照への代入と同じではないことに注意してください。初期化は、実際の参照に対する別名であるオブジェクトを使用して参照を初期化することによって、実際の参照で動作します。代入は、参照されるオブジェクトでの参照を通して動作します。

次の場合には、初期化指定子を使用せずに参照を宣言できます。
- パラメーター宣言で使用する場合
- 関数呼び出しの戻りの型の宣言内
- クラス・メンバーのクラス宣言での宣言内
- **extern** 指定子を明示的に使用する場合

以下のものへの参照は使用できません。
- 他の参照
- ビット・フィールド
- 参照の配列
- 参照を指すポインター

直接バインディング

型 T の参照 r が、型 U の式 e によって初期化されると想定します。

以下のステートメントが真であれば、参照 r は e に*直接バインドされます*。
- 式 e は、左辺値である。
- T は U と同じ型であるか、または T は、U の基底クラスである。
- T は、U と同じ、あるいはより多くの **const** または **volatile** 修飾子を持っている。

ステートメントの前のリストが真であるように e を暗黙的に型に変換できる場合、参照 r も e に直接バインドされます。

第 5 章 式と演算子

式は、計算を指定する演算子、オペランド、および区切り子のシーケンスです。式に含まれている演算子と、それらの演算子が使われるコンテキストに基づいて式の評価が行われます。式の結果、値が生じ、副次作用が生じる場合があります。*副次作用* とは、実行環境の状態における変化のことです。

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

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

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

C++ C++ の演算子は、クラス型のオペランドに適用されるときに、異なる振る舞いを行うように定義できます。これは演算子の*多重定義* と呼ばれます。本章では、多重定義されない演算子の振る舞いについて説明します。

関連参照
- 83 ページの『左辺値と右辺値』
- 201 ページの『演算子の多重定義』

演算子優先順位と結合順序

優先順位 と *結合順序* という 2 つの演算子の特性によって、オペランドが演算子とグループ化される方法が決まります。優先順位は、型が異なる演算子をオペランドとグループ化させるときの優先順位です。結合順序は、オペランドを、同じ優先順位の演算子にグループ化するときの左から右、または右から左の順序です。演算子の優先順位は、より高いまたは低い優先順位の他の演算子がある場合にのみ、意味があります。より高い優先順位の演算子を持つ式が、最初に評価されます。小括弧を使用して、強制的にオペランドをグループ化することができます。

例えば、次のステートメントでは、値 `5` は、`=` 演算子の右から左への結合順序を使用して、`a` と `b` の両方に代入されます。最初に値 `c` が `b` に代入され、次に値 `b` が `a` に代入されます。

```
b = 9;
c = 5;
a = b = c;
```

演算子優先順位と結合順序

副次式の評価順序が指定されていないので、小括弧を使用して、演算子付きのオペランドのグループ化を明示的に強制することができます。

次の式では、

```
a + b * c / d
```

優先順位により、+ 演算の前に、* および / 演算が実行されます。結合順序により、b は d によって除算される前に、c と乗算されます。

次の表に、C および C++ 言語の演算子を優先順位の順序でリストし、各演算子の結合順序の方向を示します。

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

C および C++ 演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
1	はい	C++ グローバル・スコープ・レゾリューション	**::** *name_or_qualified_name*
1		C++ クラスまたはネーム・スペース・スコープ・レゾリューション	*class_or_namespace* **::** *member*
2		メンバー選択	*object* **.** *メンバー (member)*
2		メンバー選択	*pointer* **->** *member*
2		サブスクリプト	*pointer* **[** *expr* **]**
2		関数呼び出し	*expr* **(** *expr_list* **)**
2		値生成	*type* **(** *expr_list* **)**
2		後置増分	*lvalue* **++**
2		後置減分	*lvalue* **--**
2	はい	C++ 型の識別	**typeid** **(** *type* **)**
2	はい	C++ 実行時の型識別	**typeid** **(** *expr* **)**
2	はい	C++ コンパイル時にチェックされる型変換	**static_cast** **<** *type* **>** **(** *expr* **)**
2	はい	C++ 実行時にチェックされる型変換	**dynamic_cast** **<** *type* **>** **(** *expr* **)**
2	はい	C++ チェックされない型変換	**reinterpret_cast** **<** *type* **>** **(** *expr* **)**
2	はい	C++ **const** の変換	**const_cast** **<** *type* **>** **(** *expr* **)**
3	はい	オブジェクトのサイズ (バイト)	**sizeof** *expr*
3	はい	型のサイズ (バイト)	**sizeof** **(** *type* **)**
3	はい	接頭部増分	**++** *lvalue*
3	はい	接頭部減分	**--** *lvalue*
3	はい	ビット単位否定	**~** *expr*
3	はい	否定	**!** *expr*
3	はい	単項負	**-** *expr*
3	はい	単項正	**+** *expr*
3	はい	アドレス	**&** *左辺値 (lvalue)*
3	はい	間接または参照解除	***** *expr*
3	はい	C++ 作成 (メモリーの割り振り)	**new** *type*

C および C++ 演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
3	はい	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*
4		メンバー選択	*object* **.*** *ptr_to_member*
4		メンバー選択	*object* **->*** *ptr_to_member*
5		乗算	*expr* ***** *expr*
5		除法	*expr* **/** *expr*
5		モジュロ (剰余)	*expr* **%** *expr*
6		2 項加算	*expr* **+** *expr*
6		2 項減算	*expr* **-** *expr*
7		ビット単位シフト	*expr* **<<** *expr*
7		右へのビット単位シフト	*expr* **>>** *expr*
8		より小さい	*expr* **<** *expr*
8		より小さいまたは等しい	*expr* **<=** *expr*
8		より大きい	*expr* **>** *expr*
8		より大きいまたは等しい	*expr* **>=** *expr*
9		等しい	*expr* **==** *expr*
9		等しくない	*expr* **!=** *expr*
10		ビット単位 AND	*expr* **&** *expr*
11		ビット単位排他 OR	*expr* **^** *expr*
12		ビット単位包含 OR	*expr* **|** *expr*
13		論理 AND	*expr* **&&** *expr*
14		論理包含 OR	*expr* **||** *expr*
15		条件式	*expr* **?** *expr* **:** *expr*
16	はい	単純代入	*lvalue* **=** *expr*
16	はい	乗算および代入	*lvalue* ***=** *expr*
16	はい	除算および代入	*lvalue* **/=** *expr*
16	はい	モジュロおよび代入	*lvalue* **%=** *expr*
16	はい	加算および代入	*lvalue* **+=** *expr*
16	はい	減算および代入	*lvalue* **-=** *expr*
16	はい	左へのシフトおよび代入	*lvalue* **<<=** *expr*
16	はい	右へのシフトおよび代入	*lvalue* **>>=** *expr*
16	はい	ビット単位 AND および代入	*lvalue* **&=** *expr*
16	はい	ビット単位排他 OR および代入	*lvalue* **^=** *expr*
16	はい	ビット単位包含 OR および代入	*lvalue* **|=** *expr*
17	はい	C++ 式をスロー	**throw** *expr*
18		コンマ (順序付け)	*expr* **,** *expr*

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。次のようなあいまいな式は作成しないようにしてください。

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

上記の例では、すべての C 言語インプリメンテーションで、++y と func1(y) が同じ順序で評価されるとは限りません。最初のステートメントの前に、y に値 1 が指定されている場合は、値 1 または値 2 が func1() に渡されるかどうかは不明です。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。

式と優先順位の例

以下の式では、小括弧は、コンパイラーがオペランドや演算子をグループ化する方法を明示的に示します。

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

これらの式に小括弧がない場合、小括弧によって指示されるのと同じ方法で、オペランドと演算子がグループ化されます。例えば、次の式は同じ出力を作成します。

```
total = (4+(5*3));
total = 4+5*3;
```

結合属性と可換属性の両方がある演算子とオペランドをグループ化する順序は規定されていないので、次の式で、コンパイラーはオペランドと演算子をグループ化します。

```
total = price + prov_tax +
city_tax;
```

例えば、次のような方法が (小括弧で示されているように) 可能です。

```
total = (price + (prov_tax + city_tax));
total = ((price + prov_tax) + city_tax);
total = ((price + city_tax) + prov_tax);
```

ある順序付けがオーバーフローを引き起こし、他の順序付けがオーバーフローを引き起こさないというのでないかぎり、オペランドおよび演算子のグループ化が、結果に影響を与えることはありません。例えば、price = 32767、prov_tax = -42、および city_tax = 32767、ならびにこれら 3 つの変数すべてが整数として宣言されていた場合、3 番目のステートメント total = ((price + city_tax) + prov_tax) は、整数オーバーフローを引き起こすけれども、他のステートメントは引き起こしません。

中間値が丸められるため、浮動小数点演算子の異なるグループ化は、異なる結果になる場合があります。

ある式では、オペランドや演算子のグループ化によっては、結果に影響を与えます。例えば、次の式では、各関数呼び出しは同じグローバル変数を変更します。

```
a = b() + c() + d();
```

この式は、関数が呼び出される順序によって、異なる結果になります。

式に結合属性と可換属性の両方がある演算子が含まれており、オペランドを演算子にグループ化する順序が式の結果に影響を与える場合は、式をいくつかの式に区切ります。例えば、呼び出し先関数が変数 a に影響を与えるような副次作用を作成しない場合は、次の式によって、前の例の式を置換できます。

```
a = b();
a += c();
a += d();
```

左辺値と右辺値

オブジェクト は、検査して、保管に使用できるストレージの領域です。*左辺値* は、そのようなオブジェクトを参照する式です。左辺値は、それが指定するオブジェクトの変更を必ずしも許可するわけではありません。例えば、**const** オブジェクトは、変更が不可能な左辺値です。*変更可能な左辺値* という用語は、その左辺値は、指定されたオブジェクトを検査するだけでなく、変更できることを強調するために使用されます。次のオブジェクト・タイプは左辺値ですが、変更可能な左辺値ではありません。

- 配列型
- 不完全型
- **const** 修飾型
- オブジェクトが構造体または共用体型であり、そのメンバーの 1 つが **const** 修飾型を持つ場合。

これらの左辺値は変更可能ではないため、代入ステートメントの左辺に置くことはできません。

用語*右辺値* は、メモリー内のいずれかのアドレスに保管されるデータ値のことです。*右辺値* は、それに代入する値を持つことができない式です。リテラル定数および変数は両方とも、右辺値として使用できます。右辺値を必要とするコンテキストで左辺値が現れると、左辺値は暗黙的に右辺値に変換されます。しかし、その逆は行われません。つまり、右辺値は左辺値に変換されません。右辺値は常に、完全型または void 型を持っています。

C ISO C は、*関数指定機能* を、関数型を持つ式として定義します。関数指定機能は、オブジェクト型または左辺値とは異なります。関数指定子は関数の名前、または関数ポインターを間接参照した結果を使用できます。 C 言語でも、その関数ポインターとオブジェクト・ポインターの処理を区別しています。

C++ 一方 C++ では、参照を戻す関数呼び出しは左辺値です。それ以外の場合、関数呼び出しは右辺値式です。C++ では、式はすべて*左辺値* または*右辺値* を生成するか、値をまったく生成しません。

C および C++ の両方において、演算子の中には、それらの一部のオペランドとして左辺値を必要とするものがあります。下表は、そのような演算子と、その使い方に対する追加制限を掲載しています。

演算子	要件
& (単項)	オペランドは左辺値である必要があります。
++ --	オペランドは、左辺値でなければなりません。これは、前置および後置の両形式に適用されます。
= += -= *= %= <<= >>= &= ^= |=	左方オペランドは左辺値である必要があります。

例えば、すべての代入演算子は、それらの右方オペランドを評価し、その値を左方オペランドに代入します。左方オペランドは、変更可能な左辺値、または変更可能なオブジェクトへの参照である必要があります。

アドレス演算子 (&) には、オペランドとして左辺値が必要です。一方、増分演算子 (++) と減分演算子 (--) には、修正可能な左辺値がオペランドとして必要です。以下の例は、式およびその対応する左辺値を示します。

式	左辺値
`x = 42`	`x`
`*ptr = newvalue`	`*ptr`
`a++`	`a`

左辺値 (lvalue)

式	左辺値
C++ `int& f()`	`f()` への関数呼び出し

関連参照
- 126 ページの『左辺値から右辺値への変換』

1 次式

1 次式 は、次の一般的なカテゴリーに分かれます。
- 名前 (ID)
- リテラル (定数)
- 括弧で囲んだ式
- C++ **this** ポインター
- C++ スコープ・レゾリューション演算子 (`::`) によって修飾された名前

名前

名前の値は、その型によって異なります。型は、その名前がどのように宣言されるかによって決まります。次の表は、名前が左辺値式であるかどうかを示します。

1 次式: 名前

名前の宣言	評価対象	左辺値である
算術型、ポインター型、列挙型、構造体型、または共用体型の変数	その型のオブジェクト	左辺値
列挙型定数	関連した整数値	左辺値ではない
配列	その配列。変換の対象となるコンテキストでは、その配列の最初のオブジェクトへのポインター (`sizeof` 演算子への引数として名前が使用される場合を除く)。	C 左辺値ではない
関数	その関数。変換が必要なコンテキストでは、名前が `sizeof` 演算子の引数として使用される場合を除いて、または関数呼び出し式の中で関数として使用される場合を除いて、その関数へのポインター。	C 左辺値ではない C++ 左辺値

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

リテラル

リテラルは、数値定数またはストリング・リテラルです。リテラル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。

関連参照
- 19 ページの『リテラル』
- 229 ページの『this ポインター』

ID 式

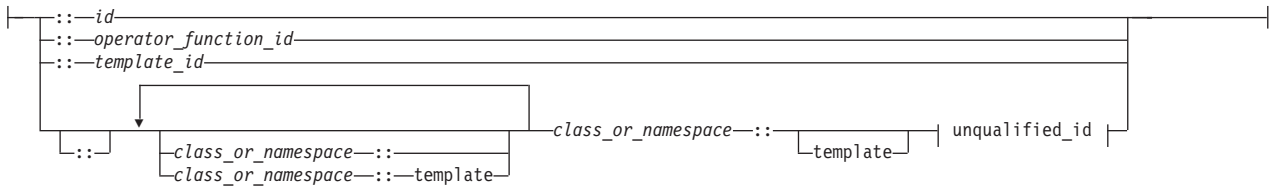
C++ ID 式 (*id-expression*) は、制限された形の 1 次式です。構文的に、*id-expression* は、C++ のすべての言語エレメントの名前を提供する上で単純な ID よりも複雑さのレベルが高くなります。

id-expression は、修飾された 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。

構文 - id-expression

unqualified_id:

qualified_id:



関連参照
- 17 ページの『ID』
- 65 ページの『第 4 章 宣言子』

整数定数式

整数コンパイル時定数 は、コンパイルの間に決定される値で、実行時に変更することはできません。*整数コンパイル時定数式* は、定数で構成されていて定数に対して評価される式です。

整数定数式は、以下の項目だけで構成される式です。
- リテラル
- 列挙子
- **const** 変数
- 整数または列挙型の**静的**データ・メンバー

- 整数型のキャスト
- **sizeof** 式

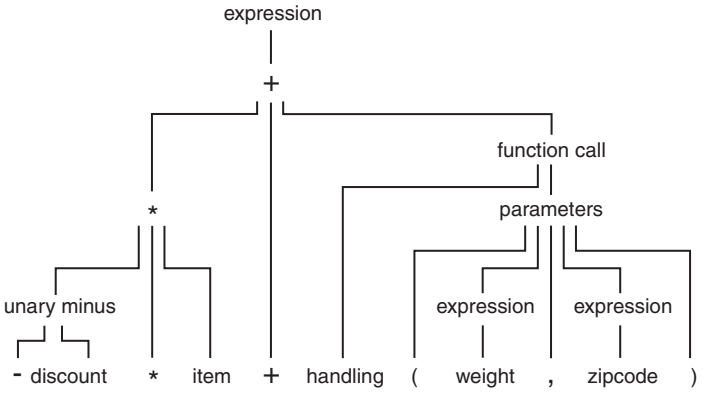
以下のような状況においては、整数定数式を使用する必要があります。
- 添え字宣言子の中 (バインドされた配列の記述として)。
- **switch** ステートメントのキーワード **case** の後。
- 列挙子の中で、enum 定数の数値として。
- ビット・フィールド幅の指定子の中。
- プリプロセッサー **#if** ステートメントの中で。 ((列挙型定数、アドレス定数、および **sizeof** は、プリプロセッサーの **#if** ステートメントでは指定できません。)

関連参照

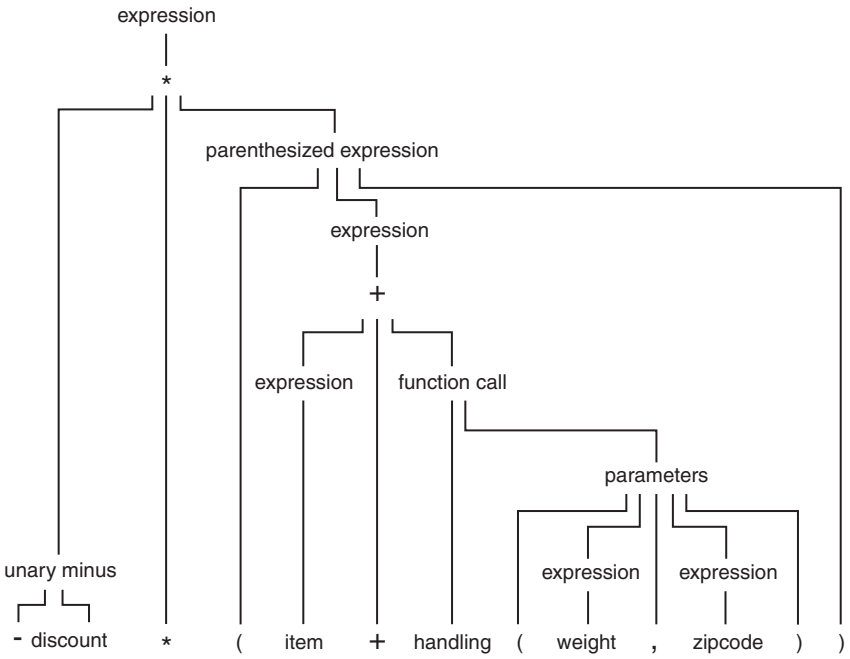
- 19 ページの『リテラル』
- 56 ページの『列挙型』

括弧で囲んだ式 ( )

小括弧を使用して、式の評価順序を明示的に強制します。次の式は、オペランドと演算子をグループ化するための小括弧は使用していません。 `weight, zipcode` を囲む式によって、関数呼び出しが形成されます。コンパイラーが式の中のオペランドと演算子を、演算子の優先順位や結合順序の規則に従って、グループ化する方法に注意してください。



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



結合属性および可換属性の両方がある演算子を含む式では、小括弧を使用して、オペランドと演算子をグループ化する方法を指定できます。次の式の小括弧は、オペランドと演算子をグループ化する順序を確実にします。

```
x = f + (g + h);
```

C++ スコープ・レゾリューション演算子 ::

C++ :: (スコープ・レゾリューション) 演算子は、隠された名前を修飾して、それらの名前を引き続き使用できるようにするために使われます。ブロックまたはクラス内の同じ名前の明示宣言によって、ネーム・スペース名またはグローバル・スコープ名が隠されている場合は、単項スコープ演算子を使用できます。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・ネーム・スペース・スコープで宣言された `count` という名前の整数を隠蔽します。ステートメント `::count = 1` は、グローバル・ネーム・スペース・スコープで宣言された `count` という名前の変数にアクセスします。

また、クラス・スコープ演算子を使用して、クラス名またはクラス・メンバーの名前を修飾することもできます。隠されているクラス・メンバー名は、そのクラス名とクラス・スコープ演算子を修飾することによって、使用することができます。

左辺値 (lvalue)

次の例では、変数 X の宣言によって、クラス型 X が隠されますが、静的クラス・メンバー count は、クラス型 X とスコープ・レゾリューション演算子で修飾することによって、まだ使用することができます。

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

class X
{
public:
      static int count;
};
int X::count = 10;                // define static data member

int main ()
{
      int X = 0;                  // hides class type X
      cout << X::count << endl;   // use static member of class X
}
```

関連参照
- 216 ページの『クラス名のスコープ』
- 191 ページの『第 10 章 ネーム・スペース』

後置式

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

後置演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
2		メンバー選択	*object* . メンバー *(member)*
2		メンバー選択	*pointer* -> *member*
2		サブスクリプト	*pointer* [*expr*]
2		関数呼び出し	*expr* (*expr_list*)
2		値生成	*type* (*expr_list*)
2		後置増分	*lvalue* ++
2		後置減分	*lvalue* --
2	はい	C++ 型の識別	**typeid** (*type*)
2	はい	C++ 実行時の型識別	**typeid** (*expr*)
2	はい	C++ コンパイル時にチェックされる型変換	**static_cast** < *type* > (*expr*)
2	はい	C++ 実行時にチェックされる型変換	**dynamic_cast** < *type* > (*expr*)
2	はい	C++ チェックなしの変換	**reinterpret_cast** < *type* > (*expr*)
2	はい	C++ **const** の変換	**const_cast** < *type* > (*expr*)

関数呼び出し演算子 ( )

関数呼び出し は、単純型の名前と括弧で囲まれた引数リストを含む式です。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。

次に例を示します。

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

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

関数呼び出しの型

関数呼び出し式の型は、その関数の戻りの型です。この型は、完全型、参照型、または型 **void** のいずれかです。関数呼び出しは、関数の型が参照である場合に限って、左辺値です。

引数およびパラメーター

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

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

リンケージおよび関数呼び出し

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

引数の型変換

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

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

コンパイラーは、呼び出し側の関数によって提供されるデータ型を、呼び出し先の関数が予想するデータ型と比較し、必要な型変換を行います。例えば次の例で、関数 funct が呼び出されると、引数 f は、**double** に、引数 c は、**int** に変換されます。

```
char * funct (double d, int i);
     /* ... */
int main(void)
{
   float f;
   char c;
   funct(f, c) /* f is converted to a double, c is converted to an int */
   return 0;
}
```

引数の評価順序

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

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

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

関数呼び出しの例

次の例では、main から func に、5 および 7 の 2 つの値が渡されます。関数 func は、これらの値のコピーを受け取り、ID a と b によって、それらにアクセスします。関数 func は、値 a を変更します。main に制御が戻るときには、x と y の実際の値は、変わっていません。呼び出し先関数 func は、変数自身ではなく、x と y の値のコピーだけを受け取ります。

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

#include <stdio.h>

void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d¥n", a, b);
}

int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d¥n", x, y);
   return 0;
}
```

このプログラムの出力は次のようになります。

```
In func, a = 12   b = 7
In main, x = 5    y = 7
```

配列サブスクリプト演算子 [ ]

後置式とその後に続く [ ] (大括弧) 内の式が、配列のエレメントを指定します。大括弧内の式は、*添え字*と呼ばれます。配列の最初のエレメントの添え字はゼロです。

定義により、式 a[b] は、式 *((a) + (b)) と等価です (また、加算は結合であるので、b[a] とも等価です)。式 a と b の間で、一方は、型 T に対するポインターでなければなりません。そして、他方は整数型または列挙型を持っていなければなりません。配列添え字の結果は、左辺値になります。次の例は、このことを示しています。

```
#include <stdio.h>

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

次に、上記の例の出力を示します。

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

C++ サブスクリプト演算子が必要とする、式の型に関する上述の制限だけでなく、サブスクリプト演算子とポインター演算の間の関係も、クラスの **operator[]** を多重定義する場合、適用されません。

各配列の最初のエレメントに、サブスクリプト 0 が付けられます。式 contract[35] は、配列 contract の 36 番目のエレメントを参照します。

多次元配列では、最も頻繁に右端サブスクリプトを増分することによって (保管場所の昇順で)、各エレメントを参照できます。

例えば、次のステートメントは、配列 code[4][3][6] の各エレメントに 100 を与えます。

```
for (first = 0; first < 4; ++first)
   {
   for (second = 0; second < 3; ++second)
      {
      for (third = 0; third < 6; ++third)
         {
         code[first][second][third] =
         100;
         }
      }
   }
```

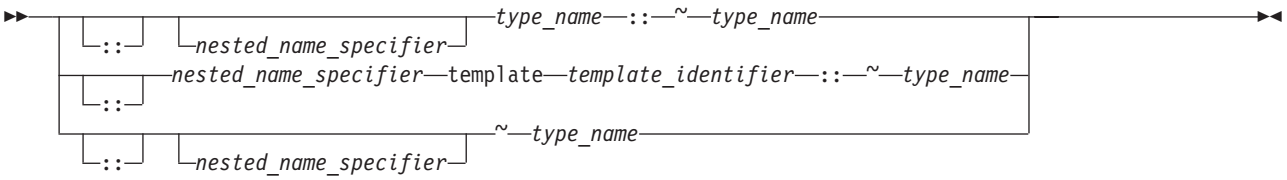
ドット演算子 .

. (ドット) 演算子は、クラス、構造体、または共用体メンバーにアクセスするために使われます。メンバーは、後置式によって指定され、その後に . (ドット) 演算子、さらにその後に、できるだけ修飾された ID または疑似デストラクター名が続きます。後置式は、**class**、**struct**、または **union** 型のオブジェクトでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。後置式と名前が左辺値の場合は、その式の値も左辺値です。後置式が型修飾されている場合、結果式内の指定メンバーには同じ型修飾子が適用されます。

疑似デストラクター

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



矢印演算子 ->

-> (矢印) 演算子は、ポインターを使用するクラス、構造体または共用体メンバーにアクセスするために使われます。後置式、その後に続く -> (矢印) 演算子、さらにその後に、できるだけ修飾された ID または疑似デストラクター名が続き、ポインターが指すオブジェクトのメンバーを指定します。 (*疑似デストラクター* は、非クラス型のデストラクターです。) 後置式は、**class**、**struct**、または **union** 型のオブジェクトを指すポインターでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。名前が左辺値の場合は、式の値も左辺値です。式が修飾型へのポインターである場合、同じ型修飾子が結果の式の中の指定されたメンバーに適用されます。

左辺値 (lvalue)

関連参照
- 91 ページの『ドット演算子 .』

typeid 演算子

C++ **typeid** 演算子によって、プログラムは、ポインターまたは参照により参照されるオブジェクトの実際の派生型を検索することができます。この演算子は、dynamic_cast 演算子とともに、C++ における RTTI (run-time type identification) サポート用に提供されています。演算子の形式は、次のとおりです。

typeid 演算子は、式 *expr* の型を表す型 **const std::type_info** の左辺値を戻します。 typeid 演算子を使用するためには、標準テンプレート・ライブラリー・ヘッダー **<typeinfo>** をインクルードしておく必要があります。

expr が、ポリモアフィック・クラスへの参照または参照解除ポインターである場合、**typeid** は、実行時に参照またはポインターが示すオブジェクトを表す **type_info** オブジェクトを戻します。ポリモアフィック・クラスでない場合、**typeid** は、参照の型または参照解除ポインターを表す **type_info** オブジェクトを戻します。次の例は、このことを示しています。

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

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };

int main() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;

  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}
```

次に、上記の例の出力を示します。

```
ap: B
ar: B
cp: C
cr: C
```

クラス A と B はポリモアフィックで、クラス C と D はそうではありません。 cp と cr は、型 D のオブジェクトを参照していますが、typeid(*cp) と typeid(cr) は、クラス C を表すオブジェクトを戻します。

左辺値から右辺値へ、配列からポインターへ、および関数からポインターへの変換は、*expr* へは適用されません。例えば、次の例の出力は int [10] であって、int * ではありません。

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

int main() {
  int myArray[10];
  cout << typeid(myArray).name() << endl;
}
```

expr がクラス型であれば、そのクラスは、完全に定義される必要があります。

typeid 演算子は、トップレベルの **const** または **volatile** 修飾子を無視します。

static_cast 演算子

C++ *static_cast 演算子* は、与えられた式を指定された型に変換します。

構文 - static_cast

次に **static_cast** 演算子の例を示します。

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

int main() {
  int j = 41;
  int v = 4;
  float m = j/v;
  float d = static_cast<float>(j)/v;
  cout << "m = " << m << endl;
  cout << "d = " << d << endl;
}
```

次に、上記の例の出力を示します。

```
m = 10
d = 10.25
```

この例では、`m = j/v;` は、型 int の答えを作成します。なぜなら、`j` と `v` の両方とも整数であるからです。逆に、`d = static_cast<float>(j)/v;` は、型 float の答えを作成します。 **static_cast** 演算子は、変数 `j` を、型 **float** に変換します。これによって、コンパイラーは、型 **float** の答えを持つ割り算を生成できます。すべての **static_cast** 演算子は、コンパイル時に解決します。そして、どの **const** または **volatile** 修飾子も除去しません。

static_cast 演算子をヌル・ポインターに適用すると、その演算子は、ターゲット型のヌル・ポインター値に変換されます。

`A` が `B` の基底クラスであれば、型 `A` のポインターを型 `B` のポインターに、明示的に変換できます。 `A` が `B` の基底クラスでなければ、コンパイラー・エラーになります。

以下が真であれば、型 `A` の左辺値を、型 `B&` にキャストすることができます。

- `A` は、`B` の基底クラスである。
- 型 `A` のポインターを、型 `B` のポインターへ変換できる。
- 型 `B` は、型 `A` と同じまたはより大きい、**const** または **volatile** 修飾子を持っている。

- A は、B の仮想基底クラスではない。

結果は、型 B の左辺値になります。

メンバー型を指すポインターは、別のメンバー型を指すポインターへ明示的に変換できます。ただし、両方の型が、同じクラスのメンバーを指すポインターである場合に限ります。明示的変換のこの形式は、メンバー型を指すポインターが、別のクラスからのものである場合に行われることがあります。ただし、クラス型のうちの 1 つは、他のものから派生していなければなりません。

reinterpret_cast 演算子

C++ *reinterpret_cast 演算子* は、無関係の型の間の変換を扱います。

構文 - reinterpret_cast

reinterpret_cast 演算子は、引数と同じビット・パターンを持っている、新規の型の値を作成します。 **const** または **volatile** 修飾をキャストすることはできません。以下の変換を明示的に実行することができます。
- ポインターから、それを保持するのに十分に大きい任意の整数型へ
- 整数値または列挙型から、ポインターへ
- 関数を指すポインターから、別の型の関数を指すポインターへ
- オブジェクトを指すポインターから、別の型のオブジェクトを指すポインターへ
- メンバーを指すポインターから、別のクラスまたは型のメンバーを指すポインターへ。ただし、メンバーの型が両方の関数型またはオブジェクト型である場合。

ヌル・ポインター値は、宛先型のヌル・ポインター値に変換されます。

型 T の左辺値式およびオブジェクト x が与えられているとして、以下の 2 つの変換は同義です。
- `reinterpret_cast<T&>(x)`
- `*reinterpret_cast<T*>(&x)`

ISO C++ は、C スタイル・キャストもサポートします。明示的なキャストの 2 つのスタイルは、構文は異なりますが同じセマンティクスを持っています。ポインターの一方の型をポインターの非互換型であるとする、どちら側からの再解釈も、通常無効です。 reinterpret_cast 演算子は、他の名前付きキャスト演算子と同様に、C スタイル・キャストよりも容易にスポットされ、明示的キャストを可能にする、強くタイプされた言語の矛盾をハイライトします。

C++ コンパイラーは、全部ではないがほとんどの違反を検出し、修正します。プログラムがコンパイルされても、そのソース・コードが、完全には正しくない場合があるということを覚えておくことは重要です。プラットフォームによっては、パフォーマンスの最適化が、ISO 別名割り当て規則へ厳格に準拠して行われます。 C++ コンパイラーは、型ベースの別名割り当て違反についてヘルプしようと試みるけれども、すべての可能なケースを検出することはできません。

次の例は、別名割り当て規則に違反しています。しかし、C++ または K&R C で、最適化されずにコンパイルされると、期待通りに実行されます。またそれは、C++ で、最適化して正常にコンパイルできますが、必ずしも期待通りには実行されません。問題の 7 行目は、x の古いまたは未初期化の値を印刷してしまいます。

```
1  extern int y = 7.;
2
3  int main() {
```

```
4       float x;
5       int i;
6       x = y;
7       i = *(int *) &x;
8       printf("i=%d. x=%f.¥n", i, x);
9   }
```

次のコードの例は、キャストが 2 つの異なるファイルにまたがっているので、コンパイラーが検出すらもできない、誤ったキャストを含んでいます。

```
1   /* separately compiled file 1 */
2       extern float f;
3       extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */
4
5   /* separately compiled file 2 */
6       extern float f;
7       extern int * int_pointer_to_f;
8       f = 1.0;
9       int i = *int_pointer_to_f;           /* no suspicious cast but wrong */
```

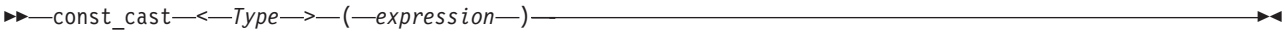
8 行目において、`int i = *int_pointer_to_f` がロード元としている同じオブジェクトを、 `f = 1.0` が保管先としていることを、コンパイラーが知る方法はありません。

関連参照
- 126 ページの『標準の型変換』
- 288 ページの『ユーザー定義の型変換』

const_cast 演算子

C++ *const_cast 演算子* は、**const** または **volatile** 修飾子を、型へ追加または型から除去するために使用されます。

構文 - const_cast



Type と *expression* の型は、それらの **const** および **volatile** 修飾子に関してのみ異なります。それらのキャストは、コンパイル時に解決されます。単一の **const_cast** 式で、任意の数の **const** または **volatile** 修飾子を追加または除去できます。

const_cast 式の結果は、*Type* が参照型でない限り、右辺値です。この場合、結果は左辺値です。

型は **const_cast** 内では定義できません。

以下は、**const_cast** 演算子の使用法を示したものです。

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

void f(int* p) {
  cout << *p << endl;
}

int main(void) {
  const int a = 10;
  const int* b = &a;

  // Function f() expects int*, not const int*
  //   f(b);
```

```
  int* c = const_cast<int>(b);
  f(c);

  // Lvalue is const
  //  *b = 20;

  // Undefined behavior
  //  *c = 30;

  int a1 = 40;
  const int* b1 = &a1;
  int* c1 = const_cast<int>(b1);

  // Integer a1, the object referred to by c1, has
  // not been declared const
  *c1 = 50;

  return 0;
}
```

コンパイラーは、関数呼び出し f(b) を許可しません。関数 f() は、**const int** ではなく **int** を指すポインターを期待します。ステートメント int* c = const_cast<int>(b) は、a の **const** 修飾なしに a を指すポインター c を戻します。 **const_cast** を使用してオブジェクトの **const** 修飾を除去するこのプロセスは、 *casting away constness* と呼ばれています。したがって、コンパイラーは、関数呼び出し f(c) を行うことができます。

コンパイラーは、代入 *b = 20 を許可しません。なぜなら、b は、型 **const int** のオブジェクトを指すからです。コンパイラーは *c = 30 を許可します。しかし、このステートメントの振る舞いは未定義です。**const** として明示的に宣言されているオブジェクトの constness をキャストし、それを変更しようとする場合、結果は予期できません。

しかし、**const** として明示的に宣言されていないオブジェクトの constness をキャストする場合、それを安全に変更することができます。上記の例では、b1 が参照しているオブジェクトは、**const** と宣言されていません。しかし、このオブジェクトを b1 によって変更することはできません。 b1 の constness をキャストし、それが参照している値を変更できます。

関連参照
- 60 ページの『型修飾子』

dynamic_cast 演算子

C++ *dynamic_cast* 演算子は、実行時に型変換を実行します。 **dynamic_cast** 演算子によって、基底クラスへのポインターが派生クラスへのポインターへと確実に変換されるか、または基底クラスを参照する左辺値が派生クラスへの参照へと確実に変換されます。それによって、プログラムはクラス階層を安全に使用することができます。この演算子と typeid 演算子は、C++ における RTTI (run-time type information) サポートを提供します。

式 dynamic_cast<T>(v) は、式 v を型 T に変換します。型 T は、完全クラス型を指すポインターまたは参照、あるいは void を指すポインターでなければなりません。 T がポインターであって、**dynamic_cast** 演算子が失敗した場合、演算子は、型 T のヌル・ポインターを戻します。 T が参照であって、**dynamic_cast** 演算子が失敗した場合、演算子は、例外 **std::bad_cast** を throw します。このクラスは、標準ライブラリー・ヘッダー **<typeinfo>** の中で検出することができます。

T が void ポインターの場合、**dynamic_cast** は、v が指すオブジェクトの開始アドレスを戻します。次の例がこのことを示しています。

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

struct A {
  virtual ~A() { };
};

struct B : A { };

int main() {
  B bobj;
  A* ap = &bobj;
  void * vp = dynamic_cast<void *>(ap);
  cout << "Address of vp  : " << vp << endl;
  cout << "Address of bobj: " << &bobj << endl;
}
```

この例の出力は、次の出力に似ています。 vp および &bobj の両方とも同じアドレスを参照します。

```
Address of vp  : 12FF6C
Address of bobj: 12FF6C
```

dynamic_cast 演算子の主目的は、型が安全な *downcasts* を実行することです。 downcast は、クラス A を指すポインターまたは参照を、クラス B を指すポインターまたは参照へ変換することです。ここで、クラス A は、B の基底クラスです。downcast についての問題は、型 A* のポインターは、A から派生したクラスの任意のオブジェクトを指すことができ、また指す必要があるということです。 **dynamic_cast** 演算子は、クラス A のポインターをクラス B のポインターに変換する場合、A が指すオブジェクトが、クラス B または B から派生したクラスに属することを保証します。

以下の例は、**dynamic_cast** 演算子の使用法を示したものです。

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

struct A {
  virtual void f() { cout << "Class A" << endl; }
};

struct B : A {
  virtual void f() { cout << "Class B" << endl; }
};

struct C : A {
  virtual void f() { cout << "Class C" << endl; }
};

void f(A* arg) {
  B* bp = dynamic_cast<B*>(arg);
  C* cp = dynamic_cast<C*>(arg);

  if (bp)
    bp->f();
  else if (cp)
    cp->f();
  else
    arg->f();
};

int main() {
  A aobj;
  C cobj;
  A* ap = &cobj;
```

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

次に、上記の例の出力を示します。

```
Class C
Class A
```

関数 f() は、ポインター arg が、型 A、B、または C のオブジェクトを指すかどうかを判別します。関数は、**dynamic_cast** 演算子を使用して、arg を、型 B のポインターへ、次に型 C のポインターに変換しようと試みることによって、この判別を行います。 **dynamic_cast** 演算子が正常に行われると、arg によって表されるオブジェクトを指すポインターを戻します。 **dynamic_cast** が失敗すると、0 が戻されます。

downcast は、ポリモアフィック・クラスにおいてのみ、**dynamic_cast** 演算子を使用して、実行することができます。上記の例では、クラス A は、仮想関数を持っているので、すべてのクラスはポリモアフィックです。 **dynamic_cast** 演算子は、ポリモアフィック・クラスから生成された実行時の型情報を使用します。

関連参照
- 247 ページの『派生』
- 288 ページの『ユーザー定義の型変換』

単項式

単項式 には、1 つのオペランドと単項演算子が含まれています。すべての単項演算子には、同じ優先順位が付けられ、右から左の結合順序が指定されます。このため、単項式は後置式です。

次の説明で示されているように、ほとんどの単項式のオペランドで、通常の算術変換を実行することができます。

次の表に、単項式の演算子が要約されています。

単項演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
3	はい	オブジェクトのサイズ (バイト)	**sizeof** (*expr*)
3	はい	型のサイズ (バイト)	**sizeof** *type*
3	はい	接頭部増分	**++** *lvalue*
3	はい	接頭部減分	**--** *lvalue*
3	はい	補数	**~** *expr*
3	はい	否定	**!** *expr*
3	はい	単項負	**-** *expr*
3	はい	単項正	**+** *expr*
3	はい	アドレス	**&** *左辺値 (lvalue)*
3	はい	間接または参照解除	* *expr*
3	はい	C++ 作成 (メモリーの割り振り)	**new** *type*
3	はい	C++ 作成 (メモリーの割り振りと初期化)	**new** *type* (*expr_list*) *type*
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*

単項演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
3	はい	型変換 (キャスト)	(*type*) *expr*

増分 ++

++ (増分) 演算子は、スカラー・オペランドの値に 1 を加えます。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ増分します。オペランドは、増分演算の結果を受け取ります。オペランドは、算術型またはポインター型の変更可能な左辺値でなければなりません。

++ は、オペランドの前にも後にも置くことができます。それがオペランドの前にくると、オペランドは増分されます。増分された値が、式の中で使用されます。オペランドの後に ++ を置くと、オペランドを増分する*前*に、そのオペランドの値が使用されます。次に例を示します。

```
play = ++play1 + play2++;
```

これは、以下の式に似ています。play2 は play の前で変更されます。

```
int temp, temp1, temp2;

temp1 = play1 + 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 + 1;
play = temp;
```

結果は、整数拡張後のオペランドと同じ型となります。

オペランドには、通常の算術変換が実行されます。

減分 --

-- (減分) 演算子は、スカラー・オペランドの値から 1 を減算します。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ、減じます。オペランドは、減分演算の結果を受け取ります。オペランドは、変更可能な左辺値でなければなりません。

減分演算子は、オペランドの前後に -- を入れることができます。この演算子がオペランドの前にあると、オペランドを減分し、減らした値が式で使われます。 -- がオペランドの後にある場合は、オペランドの現行値が式で使われ、その後でオペランドが減らされます。

次に例を示します。

```
play = --play1 + play2--;
```

これは、以下の式に似ています。play2 は play の前で変更されます。

```
int temp, temp1, temp2;

temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;
```

その結果には、オペランドと同じ型が指定されます (可能な整数拡張の場合を除く) が、左辺値ではありません。

オペランドには、通常の算術変換が実行されます。

単項正 +

+ (単項正) 演算子は、オペランドの値を保持します。オペランドには、任意の算術型またはポインター型を指定できます。結果は、左辺値ではありません。

結果は、整数拡張後のオペランドと同じ型となります。

注: 定数の前の正符号は、定数の一部ではありません。

単項負 -

- (単項負) 演算子は、オペランドの値を否定します。オペランドには、任意の算術型を指定できます。結果は、左辺値ではありません。

例えば、`quality` に値 `100` が指定された場合は、`-quality` は 値 `-100` になります。

結果は、整数拡張後のオペランドと同じ型となります。

注: 定数の前の負符号は、定数の一部ではありません。

論理否定 !

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

C オペランドの評価の結果が `0` になる場合は、式の値は `1` (真) になり、オペランドの評価の結果がゼロ以外の値になる場合は、式の値は `0` (偽) になります。

C++ オペランドの評価の結果が偽 (0) になる場合は、式の値は**真**になり、オペランドの評価の結果が真 (ゼロ以外) の値になる場合は、式の値は**偽**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;
```

ビット単位否定 ~

~ (ビット単位否定) 演算子は、オペランドのビット単位の補数を生成します。結果の 2 進表示では、すべてのビットは、オペランドの 2 進表示の同じビットの値と反対の値を保持します。オペランドには、整数型が指定されている必要があります。結果には、オペランドと同じ型が指定され、左辺値ではありません。

x が、10 進数の値 5 を表すとします。x の 16 ビットの 2 進表示は次のとおりです。

```
0000000000000101
```

式 ~x の結果は、次のようになります (ここでは、16 ビットの 2 進数で表されます)。

```
1111111111111010
```

~ 文字は、三文字表記文字の `??-` によって表されることに注意してください。

~0 の 16 ビットの 2 進表示は、次のとおりです。

1111111111111111

アドレス &

& (アドレス) 演算子は、そのオペランドを指すポインターを生成します。オペランドは、左辺値、関数指定機能、または修飾名でなければなりません。オペランドは、ビット・フィールドであることも、ストレージ・クラス **register** を指定することもできません。

オペランドが左辺値または関数である場合は、結果の型は式型を指すポインターです。例えば、式に型 **int** が指定されている場合、結果は、型 **int** が指定されたオブジェクトを指すポインターになります。

オペランドが修飾名で、メンバーが静的でない場合は、結果は、クラスのメンバーを指すポインターになり、メンバーと同じ型になります。結果は、左辺値ではありません。

p_to_y が **int** を指すポインターとして定義され、 y が **int** として定義されている場合、次の式では、変数 y のアドレスをポインター p_to_y に代入します。

```
p_to_y = &y;
```

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

アンパーサンド記号 & は、C++ では、アドレス演算子のほかに、参照宣言子として使用されます。意味は関連していますが、同一ではありません。

```
int target;
int &rTarg = target;  // rTarg is a reference to an integer.
                      // The reference is initialized to refer to target.
void f(int*& p);      // p is a reference to a pointer
```

参照のアドレスを取ると、そのターゲットのアドレスが戻されます。直前の宣言を使用して、&rTarg は &target と同じメモリー・アドレスとなります。

レジスター変数のアドレスを取ることができます。

使用する多重定義関数のバージョンを、左側が固有に判別する初期化または代入の場合に限り、多重定義関数で & 演算子を使用することができます。

関連参照
- 67 ページの『ポインター』
- 77 ページの『参照』

間接 *

* (間接) 演算子は、ポインター型オペランドによって参照される値を決めます。オペランドは、不完全型を指すポインターであってはなりません。オペランドがオブジェクトを指し示している場合は、演算子の結果、そのオブジェクトを参照する左辺値が導き出されます。オペランドが関数を指している場合、結果は C では関数指定子です。C++ では、そのオペランドが指しているオブジェクトを参照する左辺値です。配列と関数はポインターに変換されます。

オペランドの型は、結果の型を決定します。例えば、オペランドが **int** を指すポインターの場合は、結果は、**int** 型になります。

無効なアドレス (NULL など) を含むポインターに間接演算子を適用しないでください。結果は、予測できません。

単項式

p_to_y が **int** を指すポインターとして定義され、 y が **int** として定義されている場合、式は次のようになります。

```
p_to_y = &y;
*p_to_y = 3;
```

変数 y は値 3 を受け取ります。

関連参照
- 67 ページの『ポインター』

sizeof 演算子

sizeof 演算子は、オペランドのサイズを バイト で生成します。オペランドは式、または型の括弧付きの名前とすることができます。**sizeof** 式の形式は、次のとおりです。

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。

型名に適用された **sizeof** 演算子は、その型のオブジェクトによって使用されるメモリー量 (内部または末尾の埋め込みを含む) を結果として出します。**char** オブジェクトの 3 つの型 (**符号なし**、**符号付き**、またはプレーン) は、いずれもサイズがバイト 1 となります。 **sizeof** 演算子を次の項目に適用することはできません。
- ビット・フィールド
- 関数型
- 未定義の構造体またはクラス
- 不完全型 (**void** など)

式に適用された **sizeof** 演算子は、式の型の名前にのみ適用された場合と同じ結果を出します。コンパイル時に、コンパイラーは式を分析してその型を判別しますが、評価は行いません。式の型の分析で行われる通常の型変換は、いずれも **sizeof** 式に直接付随するものではありません。ただし、オペランドに型変換を行う演算子が含まれている場合、コンパイラーは、型を判別するときに、これらの型変換を考慮します。

以下の例の 2 行目では、通常の算術変換を行います。**short** は 2 バイトのストレージを使用し、**int** は 4 バイト使用しています。

```
short x; ... sizeof (x)         /* the value of sizeof operator is 2 */
short x; ... sizeof (x + 1)     /* value is 4, result of addition is type int */
```

式 x + 1 の結果は **int** 型で、sizeof(int) と同じです。値は、x に **char**、**short**、または **int** 型、あるいは任意の列挙型を指定する場合も 4 です。

型は、**sizeof** 式では定義できません。

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

```
#include <stdio.h>

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

プリプロセッサー・ディレクティブの中以外では、整数定数が必要なときは、**sizeof** 式を使用できます。**sizeof** 演算子がよく使われる 1 つの例は、ストレージ割り当て時、入力関数、および出力関数で参照されるオブジェクトのサイズを決める場合です。

もう 1 つの **sizeof** の使い方は、プラットフォームをまたがってコードを移植する場合です。データ型が表すサイズを決めるために、**sizeof** 演算子を使用します。次に例を示します。

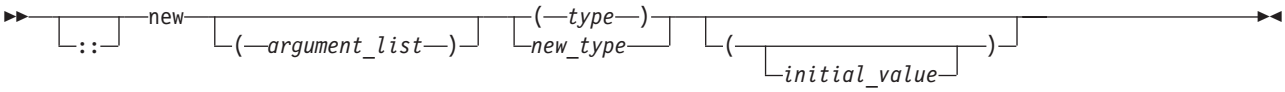
```
sizeof(int);
```

sizeof 式の結果は、それが適用される型によって異なります。

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

C++ の new 演算子

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



スコープ・レゾリューション演算子 (**::**) に **new** を接頭部として付けると、global **operator new()** が使用されます。 *argument_list* を指定した場合は、その *argument_list* に対応する、多重定義された **new** 演算子が使われます。 *type* は、既存のインクルード型またはユーザー定義の型です。 *new_type* は、まだ定義されていない型で、型指定子と宣言子をインクルードすることができます。

new 演算子を含む割り当て式は、作成されたオブジェクトのフリー・ストレージを検出するために使われます。 *new* 式 は、作成されたオブジェクトを指すポインターを戻し、これを使用してオブジェクトを初期化することができます。オブジェクトが配列の場合は、最初のエレメントを指すポインターが戻されます。

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

関数型、**void**、または不完全クラス型はオブジェクトの型ではないので、**new** 演算子を使用してこれらの型を割り振ることはできません。ただし、**new** 演算子を使用して、関数を指すポインターを割り振ることはできます。**new** 演算子を使用して、参照を作成することはできません。

作成されるオブジェクトが配列の場合は、最初の次元だけが汎用式になります。以降のすべての次元は、整数定数式でなければなりません。最初の次元は、既存の *type* が使われているときにも、汎用式になります。 **new** 演算子を使用して、ゼロ境界付きの配列を作成できます。次に例を示します。

```
char * c = new char[0];
```

この場合、固有なオブジェクトへのポインターが戻されます。

operator new() または **operator new[]()** を使用して作成されたオブジェクトは、**operator delete()** または **operator delete[]()** が、オブジェクトのメモリーを割り振り解除するために呼び出されるまで、存在します。 **delete** 演算子またはデストラクターは、**new** を使用して作成されたオブジェクトで、プログラムの終了の前に明示的に割り振り解除されていないものに対して、暗黙的に呼び出されることはありません。

小括弧が new 型内で使用される場合、構文エラーを避けるために、その new 型も小括弧で囲む必要があります。

次の例では、関数を指すポインターの配列用ストレージが割り当てられます。

```
void f();
void g();

int main(void)
{
      void (**p)(), (**q)();
      // declare p and q as pointers to pointers to void functions
      p = new (void (*[3])());
      // p now points to an array of pointers to functions
      q = new void(*[3])(); // error
      // error - bound as 'q = (new void) (*[3])();'
      p[0] = f;  // p[0] to point to function f
      q[2] = g;  // q[2] to point to function g
      p[0]();    // call f()
      q[2]();    // call g()
      return (0);
}
```

ただし、2 番目の **new** の使用では、 `q = (new void) (*[3])()` のように間違ったバインディングになります。

作成されるオブジェクトの型には、クラス宣言、列挙宣言、**const** 型、または **volatile** 型を含めることはできません。**const** または **volatile** オブジェクトを指すポインターは含めることができます。

例えば、`const char*` は使用できますが、`char* const` は使用できません。

配置構文

追加引数は、*argument_list* を使用することによって、**new** に引数を追加することができます (*配置構文* とも呼ばれます)。配置引数を使う場合は、それらの引数が指定された **operator new()** または **operator new[]()** が宣言されていなければなりません。次に例を示します。

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

class X
{
public:
      void* operator new(size_t,int, int){ /* ... */ }
};

// ...

int main ()
{
      X* ptr = new(1,2) X;
}
```

配置構文は通常、グローバル配置 new 関数を呼び出すときに使用されます。グローバル配置 new 関数は、配置 new 式の中で配置引数によって指定されたロケーションにあるオブジェクト (1 つまたは複数) を初期化します。グローバル配置 new 関数はそれ自体にメモリーを割り当てることはしないので、このロケーションには他の方法で事前に割り当てられていたストレージをアドレッシングしなければなりません。次の例では、new(whole) X(8);、new(seg2) X(9);、または new(seg3) X(10); を呼び出しても新規メモリーは割り当てられません。代わりに、コンストラクター X(8)、X(9)、および X(10) を呼び出して、バッファー whole に割り当てられたメモリーを再初期化します。

配置 new はメモリーを割り振らないので、配置構文で作成されたオブジェクトを割り振り解除するために delete を使わないでください。削除できるのは、メモリー・プール全体だけです (delete whole)。次の例では、メモリー・バッファーを残しておいて、デストラクターを明示的に呼び出すことにより、そこに保管されていたオブジェクトを破棄することができます。

```
#include <new>
class X
{
   public:
      X(int n): id(n){ }
      ~X(){ }
   private:
      int id;
      // ...
};

int main()
{
   char* whole = new char[ 3 * sizeof(X) ];  // a 3-part buffer
   X * p1 = new(whole) X(8);                 // fill the front
   char* seg2 = &whole[ sizeof(X) ];         // mark second segment
   X * p2 = new(seg2) X(9);                  // fill second segment
   char* seg3 = &whole[ 2 * sizeof(X) ];     // mark third segment
   X * p3 = new(seg3) X(10);                 // fill third segment

   p2->~X();   // clear only middle segment, but keep the buffer
   // ...
   return 0;
}
```

配置 new 構文は、コンストラクターではなく、割り振りルーチンにパラメーターを渡すためにも使用できます。

関連参照

- 283 ページの『フリー・ストレージ』
- 106 ページの『set_new_handler() — new 障害のための振る舞いのセット』
- 107 ページの『C++ の delete 演算子』
- 87 ページの『C++ スコープ・レゾリューション演算子 ::』
- 271 ページの『コンストラクターとデストラクターの概要』
- 29 ページの『オブジェクト』

new 演算子を使用して作成されたオブジェクトの初期化

C++ **new** 演算子を使用して作成されたオブジェクトは、いくつかの方法で初期化できます。クラス以外のオブジェクトまたはコンストラクターなしのクラス・オブジェクトの場合は、(式) または () を指定することによって、*new 初期化指定子* の式が new 式に提供されます。次に例を示します。

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

単項式

クラスにデフォルトのコンストラクターが指定されていない場合は、そのクラスのオブジェクトが割り振られる際に new 初期化指定子を指定する必要があります。new 初期化指定子の引数は、コンストラクターの引数と一致している必要があります。

配列に初期化指定子を指定することはできません。クラスにデフォルトのコンストラクターが指定されている場合のみ、クラス・オブジェクトの配列を初期化することができます。コンストラクターを呼び出して、各配列エレメント (クラス・オブジェクト) を初期化します。

new 初期化指定子を使用した初期化は、**new** がストレージを正常に割り振った場合にのみ実行されます。

関連参照
- 283 ページの『フリー・ストレージ』
- 271 ページの『コンストラクターとデストラクターの概要』

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

C++ **new** 演算子が新しいオブジェクトを作成すると、この演算子は、**operator new()** または **operator new[]()** 関数を呼び出して、必要なストレージを獲得します。

new は、新しいオブジェクトを作成するためのストレージを割り当てできないときには、`set_new_handler()` への呼び出しによって *new* ハンドラー 関数 (インストールされている場合) を呼び出します。`std::set_new_handler()` 関数は、ヘッダー **<new>** で宣言されます。この関数を使用して、定義済みの new ハンドラーまたはデフォルトの new ハンドラーを呼び出します。

new ハンドラーは、次のうちのどれかを実行する必要があります。
- メモリー割り振りのためにさらにストレージを取得し、それから戻ります。
- 型 `std::bad_alloc` の例外、または `std::bad_alloc` から派生したクラスをスローします。
- `abort()` または `exit()` のどちらかを呼び出します。

`set_new_handler()` 関数はプロトタイプを持ちます。

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

`set_new_handler()` は、引数として関数 (new handler) を指すポインターを取りますが、この関数は引数がなく void を戻します。`set_new_handler()` は、直前の new ハンドラー関数を指すポインターを戻します。

独自の `set_new_handler()` 関数を指定しない場合は、 **new** は、型 `std::bad_alloc` の例外をスローします。

次のプログラムでは、**new** 演算子がストレージを割り当てできない場合に、`set_new_handler()` を使用してメッセージを戻す方法を示します。

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

void no_storage()
{
    std::cerr << "Operator new failed: no storage is
    available.\n";
      std::exit(1);
}
int main(void)
```

```
{
     std::set_new_handler(&no_storage);
   // Rest of program ...
}
```

new がストレージを割り当てできないためにプログラムが失敗した場合は、プログラムは次のメッセージを出して終了します。

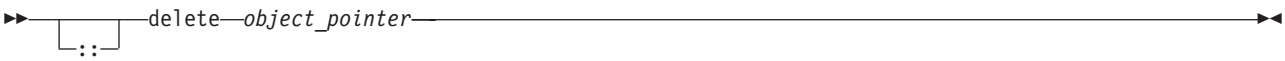
```
  Operator new failed:
no storage is available.
```

関連参照
- 103 ページの『C++ の new 演算子』
- 283 ページの『フリー・ストレージ』

C++ の delete 演算子

C++ **delete** 演算子は、オブジェクトに関連付けられたメモリーを割り当て解除することによって、**new** を使用して作成されたオブジェクトを破棄します。

delete 演算子には、**void** 戻り型があります。この演算子の構文は、次のとおりです。



delete のオペランドは、**new** によって戻されるポインターでなければなりません。定数を指すポインターであってはなりません。ヌル・ポインターを削除しても影響はありません。

delete[] 演算子は、**new[]** 演算子を使用して作成された配列オブジェクトに割り当てられたストレージを解放します。**delete** 演算子は、**new** を使用して作成された個々のオブジェクトに割り当てられたストレージを解放します。

この演算子の構文は、次のとおりです。

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

単項式

- オブジェクトがクラス以外の型の場合
- **::delete[]** 式によってオブジェクトが削除される場合

デフォルトのグローバル **operator delete()** だけが、デフォルトのグローバル **operator new()** によって割り当てられたストレージを解放することができます。デフォルトのグローバル **operator delete[]()** のみが、デフォルトのグローバル **operator new[]()** によって配列に割り当てられたストレージを解放することができます。

関連参照

- 283 ページの『フリー・ストレージ』
- 271 ページの『コンストラクターとデストラクターの概要』
- 45 ページの『void 型』

キャスト式

キャスト演算子は、*明示的型変換* のために使用されます。この演算子は、次のような形式です。ここで、*T* は型、*expr* は式です。

(*T*) *expr*

これは *expr* の値を、型 *T* に変換します。C の場合、この演算の結果は左辺値にはなりません。C++ の場合、この演算の結果は、*T* が参照であれば、左辺値です。そうでない場合、すべて結果は右辺値です。

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

キャストは、そのオペランドが左辺値の場合、有効な左辺値です。以下の単純な代入式では、最初に右側が、指定された型に変換され、次に内側の左側の式の型に変換され、結果が保管されます。その値が、指定された型に変換し直されて、それが代入の値になります。次の例では、`i` は `char *` 型です。

```
(int)i = 8     // This is equivalent to the following expression
(int)(i = (char*) (int)(8))
```

キャストに適用される複合代入演算の場合、複合代入の算術演算子は、キャスト結果の型を使って実行され、それから、単純な代入の場合と同じように進みます。次の式は同等です。また、`i` の型は `char *` です。

```
(int)i += 8     // This is equivalent to the following expression
(int)(i = (char*) (int)((int)i = 8))
```

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

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

expr(*T*)

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

C++ の場合は、オペランドには、クラス型を指定できます。オペランドにクラス型が指定された場合は、クラスにユーザー定義の型変換関数がある任意の型にキャストすることができます。これらのキャストは、

ターゲット型がクラスであれば、コンストラクターを呼び出すことができますし、ソース型がクラスであれば、型変換関数を呼び出すことができます。これらのキャストは、両方の条件が該当する場合は、不明瞭になります。

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

例

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

```
#include <stdlib.h>

int main(void) {
   int* myArray = (int*) malloc(10 * sizeof(int));
   free(myArray);
   return 0;
}
```

malloc() ライブラリー関数は、その引数のサイズのオブジェクトを保持するメモリーを指す **void** ポインターを戻します。ステートメント `int* myArray = (int*) malloc(10 * sizeof(int))` は、以下のことを行います。

- 10 個の整数を保持できるメモリーを指す void ポインターを作成します。
- その **void** ポインターを、キャスト演算子を使用して整数ポインターに変換します。
- その整数ポインターを `myArray` に代入します。配列の名前は、配列の初期のエレメントを指すポインターと同じなので、`myArray` は、`malloc()` への呼び出しによって作成されたメモリーに保管されている、10 個の整数の配列です。

2 項式

2 項式 には、1 つの演算子によって分離される 2 つのオペランドが含まれます。

すべての 2 項演算子に、同じ優先順位が付けられるわけではありません。

すべての 2 項演算子は、左から右への結合順序が指定されます。

ほとんどの 2 項演算子のオペランドが評価される順序は、指定されていません。正しい結果を得るためには、コンパイラーがオペランドを評価する順序に依存する 2 項式を作成しないようにします。

次の説明で示されているように、ほとんどの 2 項式のオペランドで、通常の算術変換を実行することができます。

次の表に、2 項式の演算子が要約されています。

2 項演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
5		乗算	*expr* * *expr*
5		除法	*expr* / *expr*
5		モジュロ (剰余)	*expr* % *expr*
6		2 項加算	*expr* + *expr*
6		2 項減算	*expr* - *expr*
7		ビット単位シフト	*expr* << *expr*
7		右へのビット単位シフト	*expr* >> *expr*
8		より小さい	*expr* < *expr*

2 項演算子の優先順位と結合順序

ランク	右結合 ?	演算子関数	使用法
8		より小さいまたは等しい	*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*
16	はい	単純代入	*lvalue* **=** *expr*
16	はい	乗算および代入	*lvalue* ***=** *expr*
16	はい	除算および代入	*lvalue* **/=** *expr*
16	はい	モジュロおよび代入	*lvalue* **%=** *expr*
16	はい	加算および代入	*lvalue* **+=** *expr*
16	はい	減算および代入	*lvalue* **-=** *expr*
16	はい	左へのシフトおよび代入	*lvalue* **<<=** *expr*
16	はい	右へのシフトおよび代入	*lvalue* **>>=** *expr*
16	はい	ビット単位 AND および代入	*lvalue* **&=** *expr*
16	はい	ビット単位排他 OR および代入	*lvalue* **^=** *expr*
16	はい	ビット単位包含 OR および代入	*lvalue* **|=** *expr*
18		コンマ (順序付け)	*expr* **,** *expr*

関連参照
- 79 ページの『演算子優先順位と結合順序』
- 131 ページの『算術変換』

乗算 *

* (乗算) 演算子は、そのオペランドの積を生成します。オペランドは、算術型または列挙型でなければなりません。結果は、左辺値ではありません。オペランドには、通常の算術変換が実行されます。

乗算演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数の乗算演算子を含む式の中でオペランドの再配置を行うことができます。例えば、次の式

```
sites * number * cost
```

は、次のいずれかの方法に解釈できます。

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

除法 /

/ (除法) 演算子はそのオペランドの商を生成します。小数部分の破棄は、多くの場合、*ゼロに切り捨て* と呼ばれます。オペランドは、算術型または列挙型でなければなりません。右方オペランドはゼロにできません。右方オペランドが 0 の場合、結果は未定義となります。例えば、式 7 / 4 は、値 1 を生成します (1.75 または 2 ではありません)。結果は、左辺値ではありません。

オペランドには、通常の算術変換が実行されます。

剰余 %

% (剰余) 演算子は、左方オペランドを右方オペランドで割り算した剰余を生成します。例えば、式 5 % 3 は 2 を生成します。結果は左辺値にはなりません。

オペランドは両方とも、整数型または列挙型でなければなりません。右方オペランドが 0 になる場合は、結果は予期できません。いずれかのオペランドに負の値がある場合は、b が 0 でなく、a/b が表示可能な場合は、結果は次の式のように、常に値 a になります。

```
( a / b ) * b + a %b;
```

オペランドには、通常の算術変換が実行されます。

加法 +

+ (加法) 演算子は、そのオペランドの合計を生成します。オペランドは両方とも算術型を保持するか、一方のオペランドがオブジェクト型を指すポインターで、もう一方のオペランドが整数型または列挙型を保持する必要があります。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

配列内のオブジェクトを指すポインターは、整数型を持つ値に加算できます。結果は、ポインター・オペランドと同じ型のポインターになります。結果は、オリジナルのエレメントから、添え字として扱われる整数値の量だけオフセットされた、配列の中の別のエレメントを参照します。結果のポインターが、配列の外側のストレージ (配列の外側の最初のロケーション以外) を指す場合、結果は予期できません。配列の終わりより 1 つ後のエレメントを指すポインターを使用して、そのアドレスにあるメモリーの内容にアクセスすることはできません。コンパイラーは、ポインターの境界検査は行いません。例えば、以下の例で、加算の後で、ptr は配列の 3 番目のエレメントを指します。

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

減法 -

- (減法) 演算子は、そのオペランドの差を生成します。オペランドは両方とも算術型または列挙型を保持するか、左方オペランドがポインター型で、右方オペランドがポインターの型と同じ型か整数型または列挙型を保持する必要があります。整数値からポインターを減算することはできません。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

左方オペランドがポインターで、右方オペランドが整数型を保持するときは、コンパイラーは、右方の値を相対位置のアドレスに変換します。結果は、ポインター・オペランドと同じ型のポインターになります。

オペランドが両方とも同じ配列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、振る舞いは未定義です。

400 ポインター差分演算の結果の型は、TERASPACE(*NO) コンパイラー・オプションを指定した場合、**ptrdiff_t** になります。 TERASPACE(*YES) コンパイラー・オプションを指定すると、ポインター差分演算の結果は signed long long 型になります。

ビット単位左シフトと右シフト << >>

ビット単位シフト演算子は、2 進数オブジェクトのビット値を移動します。左方オペランドには、シフトされる値を指定します。右方オペランドには、値のビットがシフトされる桁数を指定します。結果は、左辺値ではありません。両方のオペランドに同じ優先順位が付けられ、左から右の結合順序が指定されます。

演算子	使用法
<<	ビットが左方にシフトされることを指示します。
>>	ビットが右方にシフトされることを指示します。

各オペランドは、整数型または列挙型でなければなりません。コンパイラーは、オペランドの整数拡張を実行し、その後、右方オペランドが **int** 型に変換されます。結果には、左方オペランドと同じ型が指定されます (算術変換の後)。

右方オペランドが負の値、またはシフトされる式のビット幅より大きいかまたは等しい値を保持しないようにする必要があります。このような値でビット単位シフトを行うと、結果は予測不能な値になります。

右方オペランドに `0` がある場合は、結果は左方オペランドの値になります (通常の算術変換が実行された後)。

`<<` 演算子は、空になったビットをゼロで埋めます。例えば、`left_op` が値 `4019` を持っている場合、`left_op` のビット・パターン (16 ビットの形式) は次のようになります。

```
0000111110110011
```

式 `left_op << 3` は、次のビット・パターンを生成します。

```
0111110110011000
```

式 `left_op >> 3` は、次のビット・パターンを生成します。

```
0000000111110110
```

関係 < > <= >=

関係演算子は、2 つのオペランドを比較して、リレーションシップの妥当性を判別します。

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

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

結果は、左辺値ではありません。

次の表では、4 つの関係演算子を説明します。

演算子	使用法
<	左方オペランドの値が、右方オペランドの値より小さいかどうかを示します。
>	左方オペランドの値が、右方オペランドの値より大きいかどうかを示します。
<=	左方オペランドの値が、右方オペランドの値より小さいまたは等しいかどうかを示します。
>=	左方オペランドの値が、右方オペランドの値より大きいまたは等しいかどうかを示します。

オペランドは両方とも、算術型または列挙型を保持するか、同じ型を指すポインターでなければなりません。

C 結果は、**int** 型を持ちます。

C++ 結果は、**bool** 型を持ちます。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。ポインターが同じ配列のオブジェクトを参照しない場合は、結果は未定義になります。

ポインターは、0 に評価される定数式と比較することができます。また、ポインターを **void*** 型のポインターと比較することもできます。ポインターを **void*** 型のポインターに変換します。

2 つのポインターが同じオブジェクトを参照する場合は、この 2 つのポインターは等しいと見なされます。 2 つのポインターが同一オブジェクトの非静的メンバーを参照する場合、これらのポインターがアクセス指定子によって分離されていなければ、後で宣言されるオブジェクトを指すポインターの方がより大きいです。分離されていれば、比較は未定義です。 2 つのポインターが同じ共用体のデータ・メンバーを参照する場合は、この 2 つのポインターは同じアドレス値を保持します。

2 つのポインターが同じ配列のエレメント、または配列の最後のエレメントを超えて最初のエレメントを参照する場合、より大きいサブスクリプト値を持っているエレメントを指すポインターの方が、より大きいです。

関係演算子では、同じオブジェクトのメンバーだけを比較できます。

関係演算子には、左から右の結合順序が指定されます。例えば、次の式

```
a < b <= c
```

は、次のように解釈されます。

```
(a < b) <= c
```

a の値が b の値よりも小さい場合は、最初の関係演算は、C では値 1 を、C++ では **true** を生成します。その後で、コンパイラーは、値 **true** (または 1) を c の値と比較します (必要であれば、整数拡張が実行されます)。

等価 == !=

等価演算子は、関係演算子と同様に、リレーションシップの妥当性について 2 つのオペランドを比較します。ただし、等価演算子には、関係演算子よりも低い優先順位が付けられます。

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

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

次の表で、2 つの等価演算子を説明します。

演算子	使用法
==	左方オペランドの値が、右方オペランドの値と等価かどうかを示します。
!=	左方オペランドの値が、右方オペランドの値と等価でないかどうかを示します。

オペランドは両方とも算術型または列挙型を保持するか、同じ型を指すポインターである必要があります。あるいは、一方のオペランドがポインター型を保持し、もう一方のオペランドが void を指すポインターまたはヌル・ポインターである必要があります。結果は、C では型 **int**、C++ では **bool** です。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。

一方のオペランドがポインターで、もう一方のオペランドが値 0 の整数の場合、== 式は、ポインターのオペランドが NULL に評価される場合にだけ真になります。!= 演算子は、ポインター・オペランドが NULL に評価されない 場合に、真になります。

等価演算子を使用して、型が同じでも、同じオブジェクトに属さないメンバーを指すポインターを比較することもできます。次の式には、等価演算子と関係演算子の例が含まれています。

```
time < max_time == status < complete
letter != EOF
```

注: 等価演算子 (==) を、代入 (=) 演算子と混同しないでください。

例えば、次のような場合です。

`if (x == 3)` x が 3 の場合**真** (または 1) になります。このような等価テストでは、意図しない代入を防ぐために、演算子とオペランドの間にスペースを入れてコーディングする必要があります。

一方、

`if (x = 3)` (x = 3) がゼロ以外の値 (3) になるので、真になります。また、この式では、3 が x に代入されます。

関連参照

• 120 ページの『単純代入 =』

ビット単位 AND &

& (ビット単位 AND) 演算子は、第 1 オペランドの各ビットと第 2 オペランドの対応するビットを比較します。ビットが両方とも 1 であれば、対応する結果のビットを 1 にセットします。 1 でなければ、対応する結果のビットを 0 にセットします。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。

ビット単位 AND 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位 AND 演算子を含む式の中でオペランドの再配置を行うことができます。

次の例では、16 ビットの 2 進数で表された、a と b の値、および、a & b の結果を示します。

```
a のビット・パターン          0000000001011100
b のビット・パターン          0000000000101110
a & b のビット・パターン      0000000000001100
```

注: ビット単位 AND (&) を、論理 AND (&&) 演算子と混同しないでください。例えば、次のような場合です。

1 & 4 は 0 になります。
一方、
1 && 4 は、真になります。

関連参照
- 116 ページの『論理 AND &&』

ビット単位排他 OR ^

ビット単位排他 OR 演算子 (EBCDIC では、^ シンボルは ¬ シンボルで表します) は、第 1 オペランドの各ビットと第 2 オペランドの対応するビットを比較します。ビットが両方とも 1 か、両方とも 0 の場合、対応する結果ビットを 0 にセットします。それ以外の場合は、対応する結果ビットを 1 にセットします。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。結果は、左辺値ではありません。

ビット単位排他 OR 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位排他 OR 演算子を含む式の中でオペランドの再配置を行うことができます。 ^ 文字は、3 文字表記文字の ??' によって表されることに注意してください。

次の例では、16 ビットの 2 進数で表された a と b の値と、a ^ b の結果を示します。

```
a のビット・パターン                0000000001011100
b のビット・パターン                0000000000101110
a ^ b のビット・パターン            0000000001110010
```

関連参照
- 14 ページの『3 文字表記』

ビット単位包含 OR |

| (ビット単位包含 OR) 演算子は、各オペランドの値 (2 進形式) を比較し、いずれかのオペランドのどのビットの値が 1 であるかを示すビット・パターンの値を生成します。両方のビットが 0 であると、その結果のビットは 0 になり、それ以外の結果は 1 になります。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。結果は、左辺値ではありません。

ビット単位包含 OR 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位包含 OR 演算子を含む式の中でオペランドの再配置を行うことができます。 | 文字は、3 文字表記文字の ??! によって表されることに注意してください。

次の例では、16 ビットの 2 進数で表された、a と b の値と、a | b の結果を示します。

```
a のビット・パターン                0000000001011100
b のビット・パターン                0000000000101110
a | b のビット・パターン            0000000001111110
```

注: ビット単位 OR (|) を、論理 OR (||) 演算子と混同しないでください。例えば、次のような場合です。

```
1 | 4 は 5 になります。
```

一方、

```
1 || 4 は、真になります。
```

関連参照

- 14 ページの『3 文字表記』
- 『論理 OR ||』

論理 AND &&

&& (論理 AND) 演算子は、両方のオペランドが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は、結果の値は 1 になります。そうでない場合は、結果の値は 0 になります。その結果の型は **int** です。オペランドは両方とも、算術型またはポインター型でなければなりません。各オペランドには、通常の算術変換が実行されます。

C++ 両方のオペランドの値が**真**である場合は、その結果の値は、**真**になります。そうでない場合は、結果の値は**偽**になります。両オペランドは、暗黙的に **bool** に変換されます。結果型は **bool** です。

& (ビット単位 AND) 演算子と異なり、&& 演算子では、オペランドは必ず左から右に評価されます。左方オペランドが 0 (または**偽**) になる場合、右方オペランドは評価されません。

次の例では、論理 AND 演算子を含む式が評価される方法を示します。

式	結果
1 && 0	偽または 0
1 && 4	真または 1
0 && 0	偽または 0

次の例では、論理 AND 演算子を使用して、ゼロによる割り算を行わないようにします。

```
(y != 0) && (x / y)
```

y != 0 が 0 (または **偽**) に評価されるときは、式 x / y は評価されません。

注: 論理 AND 演算子 (&&) を、ビット単位 AND 演算子 (&) と混同しないでください。次に例を示します。

```
1 && 4 は 1 (または真) になります。
```

一方、

```
1 & 4 は 0 になります。
```

関連参照

- 114 ページの『ビット単位 AND &』

論理 OR ||

|| (論理 OR) 演算子は、いずれかのオペランドが真であるかどうかを示します。

C オペランドのいずれかにゼロ以外の値がある場合は、結果の値は 1 になります。そうでない場合は、結果の値は 0 になります。その結果の型は、**int** です。オペランドは両方とも、算術型またはポインター型でなければなりません。各オペランドには、通常の算術変換が実行されます。

C++ オペランドのいずれかの値が**真**である場合は、その結果の値は、**真**になります。そうでない場合は、結果の値は**偽**になります。両オペランドは、暗黙的に **bool** に変換されます。結果型は **bool** です。

| (ビット単位包含 OR) 演算子と異なり、|| 演算子では、オペランドは必ず左から右に評価されます。左方オペランドがゼロ以外の値 (または**真**) である場合は、右方オペランドは評価されません。

次の例では、論理 OR 演算子を含む式が評価される方法を示します。

式	結果
1 || 0	真または 1
1 || 4	真または 1
0 || 0	偽または 0

次の例では、論理 OR 演算子を使用して、y を条件付きで増分します。

```
++x || ++y;
```

式 ++x が、ゼロ以外 (または**真**) の値になる場合、式 ++y は、評価されません。

注: 論理 OR 演算子 (||) を、ビット単位 OR 演算子 (|) と混同しないでください。次に例を示します。

1 || 4 は 1 (または**真**) になります。
一方、
1 | 4 は 5 になります。

関連参照
- 115 ページの『ビット単位包含 OR |』

メンバーを指す C++ ポインター演算子 (.* ->*)

C++ メンバーを指すポインター演算子には、.* と ->* の 2 つがあります。

クラス・メンバーを指すポインターを間接参照するには、.* 演算子を使用します。第 1 オペランドは、クラス型でなければなりません。第 1 オペランドの型がクラス型 T、またはクラス型 T から派生したクラスの場合は、第 2 オペランドはクラス型 T のメンバーを指すポインターでなければなりません。

クラス・メンバーを指すポインターを間接参照するには、->* 演算子を使用します。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。

関連参照

- 83 ページの『左辺値と右辺値』
- 228 ページの『メンバーへのポインター』

条件式

条件式 は、C++ では暗黙的にタイプ bool へ変換される条件 (*operand*$_1$)、条件が true に評価される場合に評価される式 (*operand*$_2$)、および条件が値 false を持っている場合に評価される式 (*operand*$_3$) を含む複合式です。

条件式には、2 つの部分で構成される 1 つの演算子があります。 ? 記号は、条件の後に続き、 : 記号は 2 つのアクション式の間に使用されます。 ? と : の間の式は、すべて 1 つの式として扱われます。

第 1 オペランドは、スカラー型を持つ必要があります。第 2 オペランドと第 3 オペランドの型は、次のいずれかでなければなりません。
- 算術型
- 互換ポインター、構造体、または共用体型
- void

第 2 オペランドと第 3 オペランドは、ポインターまたはヌル・ポインター定数であってもかまいません。

2 つのオブジェクトが同じ型を持つが、必ずしも同じ型の修飾子 (**volatile** または **const**) でない場合、この 2 つのオブジェクトは互換性があります。ポインター・オブジェクトが同じ型を持つか、 void を指すポインターの場合は、これらのポインター・オブジェクトには互換性があります。

第 1 オペランドが評価され、その値によって第 2 オペランドまたは第 3 オペランドを評価するかどうかが判別されます。
- 値が真の場合は、第 2 オペランドが評価されます。
- 値が偽の場合は、第 3 オペランドが評価されます。

結果は、第 2 オペランドまたは第 3 オペランドの値になります。

2 番目または 3 番目の式が算術型になる場合、値に通常の算術変換を実行します。次の表は、第 2 オペランドと第 3 オペランドの型により結果の型がどのように決まるかを示します。

条件式は、第 1 オペランドと第 3 オペランドについては右から左の結合順序が適用されます。左端のオペランドが最初に評価され、次に、残りの 2 つのオペランドのいずれか 1 つだけが評価されます。次の式は、同等です。

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

C の条件式の型

C では、条件式は左辺値またはその結果ではありません。

一方のオペランドの型	もう一方のオペランドの型	結果の型
算術	算術	通常の算術変換後の算術型
構造体または共用体型	互換構造体または共用体型	両方のオペランドにすべての修飾子が付く構造体または共用体型
void	**void**	**void**
互換型を指すポインター	互換型を指すポインター	型に指定されたすべての修飾子が付く型を指すポインター

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
型を指すポインター	C: **void*** に定数 0 をキャスト C++: NULL ポインター (定数 0)	型を指すポインター
オブジェクトまたは不完全型を指すポインター	**void** を指すポインター	型に指定されたすべての修飾子が付く **void** を指すポインター

C++ の条件式の型

C++ では、条件式は、その型が **void** でなく、その結果が左辺値の場合、有効な左辺値となります。

一方のオペランドの型	もう一方のオペランドの型	結果の型
型への参照	型への参照	通常の参照変換後の参照
クラス T	クラス T	クラス T
クラス T	クラス X	型変換が存在する場合のクラス型。可能な型変換が複数ある場合は、結果は不確定になります。
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 に代入します。

```
x = (y > z) ? y : z;
```

次に等価なステートメントを示します。

```
if (y > z)
   x = y;
else
   x = z;
```

次の式では、関数 printf を呼び出し、c が数字に評価される場合に、この関数が、変数 c の値を受け取ります。そうでない場合は、printf は文字定数 'x' を受け取ります。

```
printf(" c = %c¥n", isdigit(c) ? c : 'x');
```

条件式の最後のオペランドに代入演算子が含まれる場合は、小括弧を使用して、式が正しい評価を行うようにします。例えば、次の式では、= 演算子には ?: 演算子より高い優先順位が付けられます。

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

このコンパイラーは、次のように括弧で囲まれているように解釈されるので、エラーになります。

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

つまり、k が代入式 k = j 全体としてではなく、第 3 オペランドとして扱われます。

j の値を k に代入するのは、i == 7 が偽のときで、最後のオペランドを小括弧で囲みます。

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

代入式

代入式 は、左方オペランドによって指定されたオブジェクトに値を保管します。代入演算子には、単純代入と複合代入の 2 種類があります。

すべての代入式の左方オペランドは、変更可能な左辺値でなければなりません。式の型は、左方オペランドの型です。式の値は、代入が完了した後の左方オペランドの値です。

代入式の結果は、C では左辺値ではありませんが、C++ では左辺値です。

すべての代入演算子には、同じ優先順位が付けられ、右から左の結合順序が指定されます。

単純代入 =

単純代入演算子の形式は、次のとおりです。

lvalue = *expr*

演算子は、右方オペランド *expr* の値を、左方オペランド *lvalue* によって指定されたオブジェクトに保管します。

左方オペランドは、変更可能な左辺値でなければなりません。割り当て演算の型は、左方オペランドの型です。

左方オペランドがクラス型ではない場合は、右方オペランドは暗黙的に左方オペランドの型に変換されます。この変換された型は、**const** または **volatile** によって修飾されることはありません。

左方オペランドがクラス型である場合、その型は完全でなければなりません。左方オペランドのコピー代入演算子が呼び出されます。

左方オペランドが参照型のオブジェクトの場合は、参照によって示されたオブジェクトに右方オペランドの値を代入します。

関連参照

- 83 ページの『左辺値と右辺値』
- 67 ページの『ポインター』
- 60 ページの『型修飾子』

複合代入

複合代入演算子は、2 項演算子と単純代入演算子で構成されます。複合代入演算子は、両方のオペランドに 2 項演算子の演算を実行し、その演算の結果を左方オペランドに保管します。左方オペランドは変更可能な左辺値でなければなりません。

次の表では、複合代入式のオペランドの型を示します。

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、以下と同等です。

```
a = a * (b + c)
```

そして、次の式とは同等でない ことに注意してください。

```
a = a * b + c
```

次の表では、複合代入演算子をリストし、各演算子を使用した式を示します。

演算子	例	等価な式
+=	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++ オペランド型のテーブルに加え、式は、暗黙的に左方オペランドの cv 非修飾型に変換されます (クラス型でない場合)。しかし、左方オペランドがクラス型の場合、そのクラスは完全になり、そのクラスのオブジェクトへの代入は、コピー代入操作として行われます。 C++ では、複合式および条件式は左辺値であり、複合代入式の左方オペランドとすることができます。

コンマ式

コンマ式 には、任意の型の 2 つのオペランドがコンマで区切られて含まれており、左から右の結合順序が適用されます。左方オペランドは評価されますが、副次作用が生じる可能性があり、値がある場合、その値は廃棄されます。次に、右方オペランドが評価されます。コンマ式の結果の型および値は、通常の単項変換後の右方オペランドの型および値です。 C の場合、コンマ式の結果は左辺値ではありません。 C++ では、結果は、右方オペランドが左辺値であれば、左辺値です。次のステートメントは同じです。

```
r = (a,b,...,c);
a; b; r = c;
```

相違点として、コンマ演算子は、ループ制御式などの式のコンテキストに適したものとすることができます。

右方オペランドが左辺値である場合、複合式のアドレスを取ることができます。

```
&(a, b)
a, &b
```

コンマ式

コンマ演算子は結合するので、コンマで区切られた任意の数の式は単一の式を形成します。コンマ演算子を使用すると、副次式は左から右の順に評価されることが保証され、最後の副次式の値が式全体の値になります。

次の例では、omega が 11 の場合は、式は delta を増分し、値 3 を alpha に代入します。

```
alpha = (delta++, omega % 4);
```

評価順序点は、第 1 オペランドの評価後に生じます。delta の値は廃棄されます。

例えば、次の式

```
intensity++, shade * increment, rotate(direction);
```

の値は、次の式の値になります。

```
rotate(direction)
```

コンマ演算子の基本的な使用目的は、次のような状況で、副次作用をもたらすことです。
- 関数の呼び出し
- 反復ループへの入力または繰り返し
- 条件のテスト
- 副次作用は必要であるが、式の結果は今すぐに必要ではないその他の状況

コンマ文字が使われているコンテキストによっては、あいまいさを避けるために括弧が必要な場合があります。例えば、次の関数

```
f(a, (t = 3, t + 2), c);
```

の引数は、値 a、値 5、および値 c の 3 個だけです。括弧を必要とするその他のコンテキストとしては、構造および共用宣言子内のフィールド長の式、列挙宣言子リストの列挙値の式、ならびに宣言および初期化指定子の初期化式があります。

上の例では、関数呼び出しの中の引数の式を区切るためにコンマが使用されています。このコンテキストでは、コンマを使用しても、関数の引数の評価順序 (左から右) は保証されません。

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。
`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。

C++ の throw 式

C++ *throw* 式は、C++ の例外ハンドラーに例外をスロー (throw) するために使われます。 throw 式は、**void** 型です。

関連参照
- 329 ページの『第 17 章 例外処理』
- 45 ページの『void 型』

第 6 章 暗黙の型変換

与えられた型の式 e は、以下の状況のいずれかの場合で使用されると、*暗黙的に変換* されます。
- 式 e が、算術演算または論理演算のオペランドとして使用される。
- 式 e が、**if** ステートメントまたは繰り返しステートメント (**for** ループなど) の中で、条件として使用される。式 e は、**bool** (C では **int**) に変換されます。
- 式 e が、**switch** ステートメントの中で使用される。式 e は、整数型に変換されます。
- 式 e が、初期化で使用される。これには、次の場合が含まれます。
  - 代入が e とは異なる型を持つ左辺値に行われる。
  - 関数に、パラメーターとは異なる型を持つ e の引数値が提供されている。
  - 式 e が、関数の **return** ステートメントに指定されていて、e が、関数の定義済み戻り型とは異なる型を持っている。

コンパイラーが以下のステートメントを許す場合にかぎって、コンパイラーは、式 e から型 T への暗黙的な変換を許します。

```
T var = e;
```

例えば、異なるデータ型の値を足す時、まずは両方の値を同じ型に変換します。 **short int** の値と **int** の値を加算する場合、**short int** の値を **int** 型に変換します。

キャスト演算子、関数スタイル・キャスト、または C スタイル・キャストのうちのいずれかを使用して、明示的な型変換を行うことができます。

整数および浮動小数点拡張

整数拡張 とは、1 つの整数型の別の整数型への変換です。この場合、2 番目の型は、最初の型のすべての可能な値を保持することができます。整数が使用できるところではどこでも、ある種の基本型を使用することができます。整数拡張によって変換できる基本型は、以下のとおりです。
- **char**
- C++ **bool**
- **wchar_t**
- **short int**
- 列挙子
- 列挙型のオブジェクト
- 整数ビット・フィールド (符号付きおよび符号なしの両方)

wchar_t を除き、**int** で表せない値は、**unsigned int** に変換されます。 **wchar_t** では、元の型のすべての値を **int** で表せる場合、その値は、元の型のすべての値を最も適切に表すことができる型に変換されます。例えば、**long** がすべての値を表すことができる場合、その値は、**long** に変換されます。

2 進浮動小数点拡張

float 型の右辺値を、**double** 型の右辺値に変換できます。式の値は、変更されません。この変換は、*2 進浮動小数点拡張* です。

10 進浮動小数点拡張

C 型 **_Decimal32** の右辺値を型 **_Decimal64** の右辺値に、型 **_Decimal32** の右辺値を型 **_Decimal128** の右辺値に、または型 **_Decimal64** の右辺値を型 **_Decimal128** の右辺値にそれぞれ変換できます。式の値は、変更されません。この変換は、*10 進浮動小数点拡張* です。

これらの変換では、値は新しい型に正しく変換されます。

標準の型変換

多くの C および C++ 演算子によって、式の型を変更する*暗黙の型変換* が行われます。異なるデータ型の値を足す時、まず、両方の値を共通の形式に変換します。例えば、**short int** の値と **int** の値を加算する場合、**short int** の値を **int** 型に変換します。オリジナルのオブジェクトの値が、より短い型によって表すことができる範囲を超えている場合、データの損失が生じます。

暗黙の型変換は、以下の場合に行われます。
- オペランドが算術演算または論理演算用に用意される。
- 代入される値とは型が異なる左辺値に対して、代入が行われる。
- 関数に、パラメーターとは異なる型を持つ引数値が与えられる。
- 関数の **return** ステートメントに、その関数用に定義されている戻りの型とは異なる値が指定される。

C スタイル・キャスト、C++ 関数スタイル・キャスト、または C++ キャスト演算子 の 1 つを使用して、明示的な型変換を行うことができます。

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

int main() {
  float num = 98.76;
  int x1 = (int) num;
  int x2 = int(num);
  int x3 = static_cast<int>(num);

  cout << "x1 = " << x1 << endl;
  cout << "x2 = " << x2 << endl;
  cout << "x3 = " << x3 << endl;
}
```

次に、上記の例の出力を示します。

```
x1 = 98
x2 = 98
x3 = 98
```

整数 x1 には、C スタイル・キャストを使用して明示的に num を **int** に変換した値が代入されます。整数 x2 には、関数スタイル・キャストを使用して変換された値が代入されます。整数 x3 には、**static_cast** 演算子を使用して変換された値が代入されます。

関連参照

- 288 ページの『ユーザー定義の型変換』

左辺値から右辺値への変換

コンパイラーが右辺値を期待している状況で左辺値が現れた場合、コンパイラーは、左辺値を右辺値に変換します。

型 T の左辺値 e は、T が関数型または配列型でなければ、右辺値に変換できます。変換後、e の型は T になります。これに対する例外を、次の表にリストします。

変換前の状況	結果の振る舞い
T は不完全型である	コンパイル時エラー
e は、未初期化オブジェクトを参照している	未定義の振る舞い
e は、型 T でないオブジェクトを参照している	未定義の振る舞い
C++ e は、型 T でないオブジェクト、または T から派生した型でないオブジェクトを参照している	未定義の振る舞い
C++ T は非クラス型である	変換後の e の型は T で、**const** または **volatile** によって修飾されていない

関連参照
- 83 ページの『左辺値と右辺値』

ブール変換

C++ 整数、浮動小数点、算術、列挙、ポインター、およびメンバー右辺値型を指すポインターを、型 **bool** の右辺値に変換できます。ゼロ、ヌル・ポインター、またはヌル・メンバー・ポインター値は、**偽** に変換されます。他の値はすべて **真**に変換されます。

次にブール変換ステートメントを示します。

```
void f(int* a, int b)
{
  bool d = a;  // false if a == NULL
  bool e = b;  // false if b == 0
}
```

a がヌル・ポインターに等しい場合、変数 d は **偽** になります。そうでない場合には、d は **真** になります。 b がゼロに等しい場合、変数 e は **偽** になります。そうでない場合には、e は **真** になります。

関連参照
- 40 ページの『ブール変数』

整数変換

以下の変換を行うことができます。
- 整数型 (符号付きおよび符号なしの整数型を含む) の右辺値から、別の整数型の右辺値へ
- 列挙型の右辺値から、整数型の右辺値へ

整数 a を符号なし型へ変換する場合、結果値 x は、 a および x が、モジュロ 2^*n* で合同であるような、最少の符号なし整数です。ここで n は、符号なし型を表すのに使用されるビット数です。 2 つの数 a および x が、モジュロ 2^*n* で一致する場合、次の式は真です。ここで、関数 pow(m, n) は、m の n 乗の値を戻します。

```
a % pow(2, n) == x % pow(2, n)
```

整数 a を符号付き型に変換する場合、コンパイラーは、新規の型が a を保持するのに十分に大きい場合、結果の値を変更しません。新規の型が十分には大きくない場合、その振る舞いはコンパイラーで定義されます。

C++ **bool** を整数に変換する場合、**偽**の値は 0 に変換され、**真** の値は 1 に変換されます。

整数拡張は、変換の別のカテゴリーに属します。整数変換ではありません。

関連参照
- 44 ページの『整変数』

浮動小数点の型変換

浮動小数点型の右辺値を、別の浮動小数点型の右辺値に変換できます。

浮動小数点拡張 (**float** から **double** への変換など) は、変換の別のカテゴリーに属します。浮動小数点変換ではありません。

関連参照
- 41 ページの『2 進浮動小数点変数』
- 125 ページの『整数および浮動小数点拡張』
- 42 ページの『10 進浮動小数点変数』

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。

400 7 種類のポインター間の変換が、「*ILE C/C++ Programmer's Guide*」の第 29 章『Using Teraspace in ILE C and C++ Programs』の『C/C++ Pointer Support』に記載されています。

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代入変換です。次の表の中で、`const` 修飾された左辺値は、代入の左方オペランドとして使用できません。

表 1. C ポインターに関する有効な代入変換

左方オペランドの型	許可される右方オペランドの型
(オブジェクト) `T` へのポインター	定数 `0` `T` と互換性のある型へのポインター `void` へのポインター (**void***)
(関数) `F` へのポインター	定数 `0` `F` と互換性のある関数へのポインター

左方オペランドの参照された型は、右方オペランドと同じ修飾子を持ちます。他のポインターの型が **void*** の場合は、オブジェクト・ポインターの型が不完全になる場合があります。

関数に渡されるポインター引数は、その関数によって期待される正しい型が必ず渡されるように、明示的にキャストでなければなりません。 C の汎用オブジェクト・ポインターは **void*** ですが、汎用関数ポインターは存在しません。

void* への変換

オプションとして型が修飾された、型 `T` のオブジェクトを指すすべてのポインターは、同じ **const** または **volatile** 修飾を保持しながら、**void*** に変換できます。

C 以下の表に、左方オペランドとして **void*** を持つ許容代入変換を示します。

表 2. C での void に関する有効な代入変換*

左方オペランドの型	許可される右方オペランドの型
(void*)	定数 0 (オブジェクト) T へのポインター **(void*)**
400 8 バイトのポインター	16 ポインター
400 16 ポインター	8 バイトのポインター

オブジェクト T は型が不完全な場合があります。

C++ 標準型変換を使用して、関数へのポインターを型 **void*** に変換することはできません。 **void*** に関数を保持するだけの十分なビットがある場合には、明示的に行うことができます。

派生から基底への変換

C++ 変換があいまいでない限り、A が B のアクセス可能な基底クラスであれば、型 B* の右辺値ポインターを、クラス A* の右辺値ポインターに変換できます。アクセス可能な基底クラスに対する式が、複数の別個のクラスを参照できる場合には、変換はあいまいなものになります。結果として得られる値は、派生クラス・オブジェクトの基底クラス・サブオブジェクトを指します。型 B* のポインターがヌルの場合、そのポインターは型 A* のヌル・ポインターに変換されます。基底クラスが、派生クラスの仮想基底クラスである場合、クラスを指すポインターを、その基底クラスを指すポインターに変換できないことに注意してください。

NULL ポインター定数

評価がゼロになる定数式は、*ヌル・ポインター定数* です。この式をポインターに変換することができます。このポインターは、ヌル・ポインター (ゼロ値を持つポインター) となり、どのオブジェクトをも指さないようになります。

配列からポインターへの変換

型「N の配列」(N は、配列の単一エレメントの型です) での左辺値または右辺値を、N* に変換できます。結果は、配列の初期エレメントを指すポインターです。しかし、式が & (アドレス) 演算子または **sizeof** 演算子のオペランドとして使用される場合には、この変換は行うことができません。

関数からポインターへの変換

型 T の関数である左辺値を、型 T の関数を指すポインターである右辺値に変換できます。ただし、式が、& (アドレス) 演算子、() (関数呼び出し) 演算子、または **sizeof** 演算子のオペランドとして使用される場合は除きます。

参照変換

C++ 参照変換は、参照初期化が行われる場合には、いつでも実行することができます (引数の受け渡しおよび関数からの戻り値で実行される参照初期化の場合も含めて)。変換があいまいでなければ、クラスへの参照を、そのクラスのアクセス可能な基底クラスへの参照に変換することができます。変換の結果は、派生クラス・オブジェクトの基底クラス・サブオブジェクトへの参照になります。

参照変換は、対応するポインター型変換が許される場合に許されます。

メンバーを指すポインターの変換

C++ メンバーを指すポインターの変換は、メンバーを指すポインターの変換が初期化、代入、または比較されるときに行われます。メンバーへのポインターは、オブジェクトへのポインターまたは関数へのポインターと同じではないことに注意してください。

評価がゼロになる定数式は、メンバーへのヌル・ポインターに変換されます。

以下の条件が真である場合、基底クラスのメンバーへのポインターは、派生クラスのメンバーへのポインターに変換することができます。
- 型変換が、あいまいではない。基底クラスの複数インスタンスが派生クラス内にあると、変換はあいまいになります。
- 派生クラスへのポインターは、基底クラスへのポインターに変換することができる。その場合、その基底クラスは、*アクセス可能* であるといいます。
- メンバー型が一致する。例えば、クラス `A` は、クラス `B` の基底クラスであると想定します。型 **int** の `A` のメンバーを指すポインターを、型 **float** の型 `B` のメンバーを指すポインターには、変換できません。
- 基底クラスが、仮想ではない。

修飾変換

C++ 型 *cv1* `T*` (ここで、*cv1* は、ゼロ以上の **const** または **volatile** の修飾の任意の組み合わせ) の右辺値を、型 *cv2* `T*` の右辺値に変換できます。ただし、*cv2* `T*` が *cv1* `T*` よりも、多くの **const** または **volatile** で修飾されている場合です。

cv1 `T` のクラス `X` のメンバーを指す型ポインターの右辺値を、 *cv2* `T` のクラス `X` のメンバーを指す型ポインターの右辺値に変換できます。ただし、変換できるのは、*cv2* `T` が *cv1* `T` よりも多くの **const** または **volatile** で修飾されている場合です。

関連参照
- 60 ページの『型修飾子』

関数引数変換

C 関数宣言が存在していて、宣言された引数型が含まれている場合、コンパイラーは型検査を行います。関数が呼び出されたときに可視の関数宣言がない場合、またはプロトタイプ引数リストの可変部分に式が引数として現れると、コンパイラーは、関数に引数を渡す前にデフォルトの引数拡張を行うか、または式の値を変換します。自動変換では、次のことが行われます。
- 整数拡張
- **double** 型に変換される **float** 型の引数

C++ C++ での関数宣言は、常にパラメーター型を指定する必要があります。また、関数は、まだ宣言されていない場合には、呼び出されない場合もあります。

関連参照
- 125 ページの『整数および浮動小数点拡張』
- 136 ページの『関数宣言』

その他の変換

400 7 種類のポインター間の変換が、「*ILE C/C++ Programmer's Guide*」の第 29 章『Using Teraspace in ILE C and C++ Programs』の『C/C++ Pointer Support』に記載されています。

void 型

定義上、**void** 型には、値がありません。したがって、他の型に変換できないし、代入によって、他の値を **void** に変換することもできません。ただし、明示的に値を **void** にキャストすることはできます。

構造体または共用体型

構造体型間または共用体型間の変換は、次の場合以外は実行できません。 C では、右オペランドの型が左オペランドの型と互換性がある場合は、互換性のある構造体または共用体型間で割り当て型変換を実行できます。

表 3. C における構造体または共用体型の正しい割り当て型変換

左方オペランドの型	許可される右方オペランドの型
C 構造体または共用体型	互換性のある構造体または共用体型

クラス型

C++ クラス型同士の間に標準の型変換はありませんが、クラス型用の独自の型変換オペレーターを記述することはできます。

列挙型

C C では、**enum** 型指定子を使用して値を定義すると、その値は **int** として扱われます。 **enum** 値への変換およびその値からの変換は、**int** 型に対する場合と同様に進められます。

enum から任意の整数型に変換することはできますが、整数型から **enum** に変換することはできません。

関連参照
- 45 ページの『void 型』
- 288 ページの『ユーザー定義の型変換』
- 56 ページの『列挙型』

算術変換

変換は、個々の演算子およびオペランドの型によってそれぞれ異なります。ただし、整数型および浮動小数点型のオペランドについては、多くの演算子が同様の変換を行います。これらの標準型変換は、本来は算術で使用される値の型に適用されるため、*算術変換* と呼ばれます。

算術変換は、算術演算子のオペランドをマッチングするために使用されます。

算術変換

算術変換は、以下の順序で進められます。

オペランド型	型変換
C 1 つのオペランドは **_Decimal128** 型で、他のオペランドは long double 型、double 型、または float 型ではありません。	他のオペランドは **_Decimal128** に変換されます。[1]
C 1 つのオペランドは **_Decimal64** 型で、他のオペランドは long double 型、double 型、または float 型ではありません。	他のオペランドは **_Decimal64** に変換されます。[1]
C 1 つのオペランドは **_Decimal32** 型で、他のオペランドは long double 型、double 型、または float 型ではありません。	他のオペランドは **_Decimal32** に変換されます。[1]
1 つのオペランドは **long double** 型です。	他のオペランドは **long double** に変換されます。
1 つのオペランドは **double** 型です。	他のオペランドは **double** に変換されます。
1 つのオペランドは **float** 型です。	他のオペランドは **float** に変換されます。
400 1 つのオペランドは**パック 10 進数**型です。	他のオペランドは**パック 10 進数**に変換されます。
1 つのオペランドは **unsigned long long int** 型です。	他のオペランドは **unsigned long long int** に変換されます。
1 つのオペランドは **long long** 型です。	他のオペランドは **long long** に変換されます。
1 つのオペランドは **unsigned long int** 型です。	他のオペランドは **unsigned long int** に変換されます。
1 つのオペランドは **unsigned int** 型で他のオペランドは **long int** 型です。**unsigned int** の値は **long int** で表すことができます。	**unsigned int** 型のオペランドは **long int** に変換されます。
1 つのオペランドは **unsigned int** 型で他のオペランドは **long int** 型です。**unsigned int** の値は **long int** で表すことができません。	両方のオペランドとも **unsigned long int** に変換されます。
1 つのオペランドは **long int** 型です。	他のオペランドは **long int** に変換されます。
1 つのオペランドは **unsigned int** 型です。	他のオペランドは **unsigned int** に変換されます。
両方のオペランドとも **int** 型です。	結果は **int** 型になります。

注:

1. 2 進浮動小数点型と 10 進浮動小数点型の間の変換には、明示的なキャストが必要です。

関連参照

- 79 ページの『第 5 章 式と演算子』
- 44 ページの『整変数』
- 41 ページの『2 進浮動小数点変数』
- 42 ページの『10 進浮動小数点変数』

明示的キーワード

C++ 引数を 1 つだけ使用し、明示的キーワードを使用せずに宣言されたコンストラクターは、*変換コンストラクター*です。代入演算子を使用して、変換コンストラクターでオブジェクトを構成できます。 明示的キーワードを使用してこの型のコンストラクターを宣言すると、この振る舞いを防ぎます。 *明示的*キーワードは、望ましくない暗黙的型変換を制御します。それは、クラス宣言内のコンストラクターの宣言にだけ使用されます。例えば、デフォルトのコンストラクターを除いて、以下のクラスのコンストラクターは、変換コンストラクターです。

```
class A
{  public:
   A();
   A(int);
   A(const char*, int = 0);
};
```

以下の宣言は、正しい宣言です。

```
A c = 1;
A d = "Venditti";
```

最初の宣言は `A c = A(1)` に等価です。

明示的 キーワードを使用してこのクラスのコンストラクターを宣言すると、前の宣言は正しくなくなります。

例えば、クラスを以下のようなクラスとして宣言する場合、

```
class A
{  public:
   explicit A();
   explicit A(int);
   explicit A(const char*, int = 0);
};
```

クラス型の値に一致する値だけを代入できます。

例えば、以下のステートメントは正しくありません。

```
  A a1;
  A a2 = A(1);
  A a3(1);
  A a4 = A("Venditti");
  A* p = new A(1);
  A a5 = (A)1;
  A a6 = static_cast<A>(1);
```

関連参照

- 290 ページの『コンストラクターによる変換』

第 7 章 関数

プログラム言語のコンテキストにおいて、*関数* という語は、出力値を計算するために使用されるステートメントの集まりを意味します。この語は、数学で使われるときほど厳密に使われていません。数学では、関数は、入力変数を出力変数に*一意的に* 関連付ける集合を意味します。 C または C++ プログラムにおける関数は、すべての入力に対して一貫性のある出力を生成するとは限らず、出力をまったく生成しないこともあり、副次作用を持つこともあります。関数は、パラメーター・リストのパラメーター (存在する場合) をオペランドとする、ユーザー定義の演算と考えることができます。

関数は、ユーザーが作成した関数と、C 言語のインプリメンテーションで提供されている関数の 2 つのカテゴリーに分類されます。後者は*ライブラリー関数* と呼ばれます。ライブラリー関数はコンパイラーが備えている関数のライブラリーに属しているからです。

関数の結果は、その関数の*戻り値* と呼ばれます。戻り値のデータ型は、*戻りの型* と呼ばれます。関数の戻りの型が先行し、関数のパラメーター・リストが後続している関数 ID を*関数宣言* または*関数プロトタイプ* と呼びます。*関数本体* という用語は、関数が実行する処理を表すステートメントのことをいいます。関数の本体は、中括弧で囲まれ、*関数ブロック* を構成します。関数の戻りの型の後には、関数の名前、パラメーター・リストが続き、本体とともに*関数定義* を構成します。

関数名の後に関数呼び出し演算子 () が続くと、その関数が評価されます。関数がパラメーターを受け取るように定義されている場合は、その関数に送られる値が関数呼び出し演算子の括弧内にリストされます。これらの値は、そのパラメーターに対する*引数* で、今説明したプロセスのことを、関数に*引数を渡す* といいます。

C++ C++ では、関数のパラメーター・リストは、その関数の*シグニチャー* と呼ばれます。関数の名前とシグニチャーにより、その関数が一意的に識別されます。この言葉自体が表しているように、関数のシグニチャーは、多重定義された関数の異なるインスタンスを区別するために、コンパイラーによって使用されます。

関連参照

- 199 ページの『第 11 章 多重定義』

C 関数に対する C++ の拡張

C++ C++ 言語では、C 関数に対して多くの拡張を行っています。その内容は次のとおりです。

- 参照引数
- デフォルト引数
- 参照の戻りの型
- インライン関数
- メンバー関数
- 多重定義されている関数
- 演算子関数
- コンストラクター関数およびデストラクター関数
- 変換関数
- 仮想関数
- 関数テンプレート

- 例外の指定
- コンストラクター初期化指定子

関連参照

- 149 ページの『参照による引数の受け渡し』
- 150 ページの『C++ 関数におけるデフォルト引数』
- 153 ページの『戻りの型としての参照の使用』
- 155 ページの『インライン関数』
- 224 ページの『メンバー関数』
- 199 ページの『関数の多重定義』
- 201 ページの『演算子の多重定義』
- 271 ページの『コンストラクターとデストラクターの概要』
- 291 ページの『変換関数』
- 262 ページの『仮想関数』

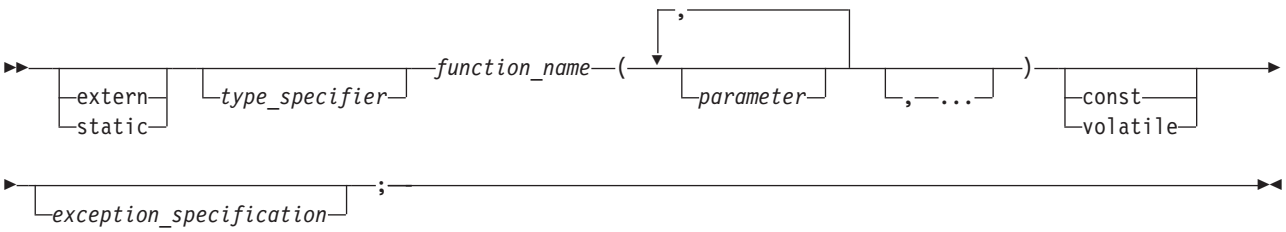
関数宣言

関数宣言 は、関数の名前およびそのパラメーターの数と型を明確にします。関数宣言は、戻りの型、名前、およびパラメーター・リストで構成されます。さらに、関数宣言はオプションで、その関数のリンケージを指定することができます。 C++ では、宣言で、例外指定、const 修飾、または volatile 修飾を指定することもできます。

宣言は、コンパイラーが関数を使用する前に、コンパイラーに対して、関数の形式と存在を通知します。すべての宣言が一致する場合は、1 つの関数を 1 つのプログラム内で複数回宣言することができます。すべての関数は、呼び出される前に暗黙的または明示的に宣言されている必要があります。C89 では、明示的なプロトタイプなしで関数が呼び出される場合、コンパイラーは暗黙宣言を提供します。 コンパイラーは、関数呼び出しのパラメーターと関数宣言におけるパラメーターとの不一致を検査することができます。コンパイラーは、引数の型の検査および引数の変換のためにも、この宣言を使います。

関数定義 には、関数宣言と関数本体が含まれます。 1 つの関数には 1 つの定義しか認められません。

通常、宣言はヘッダー・ファイルに入れますが、関数定義はソース・ファイルに入れます。



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

宣言の中には、パラメーター・リスト内にパラメーターの名前がないものもあります。つまり、下記の例のように、宣言は単にパラメーターおよび戻り値の型を指定するだけのものがあります。 これはプロトタイ

ピング と呼ばれます。関数プロトタイプは、関数の戻りの型、関数の名前、およびパラメーター・リストで構成されます。次の例は、このことを示しています。

```
int func(int,long);
```

関数プロトタイプは、C と C++ との間の互換性を保持するのに必要とされます。空のパラメーター・リストを持つ関数の非プロトタイプ形式は、C ではその関数のパラメーター数が不明であることを意味しますが、C++ ではその関数はパラメーターを取らないことを意味します。

関数の戻りの型

関数は、配列および関数呼び出しの結果以外は、任意の型の値を戻すように定義できます。配列および関数呼び出しの結果を処理するには、その配列または関数へのポインターを戻す必要があります。関数は、関数を指すポインター、または配列の最初のエレメントを指すポインターを戻すことができます。しかし、配列または関数の型を持っている値を戻すことはできません。関数が値を戻さないように指示するには、戻りの型を **void** として関数を宣言します。

関数は、**volatile** 型または **const** 型のデータ・オブジェクトを戻すものとして宣言することはできません。しかし、**volatile** または **const** オブジェクトへのポインターを戻すことはできます。

C++ での関数の宣言時の制限

すべての関数宣言では、戻りの型を指定する必要があります。

400 ILE C++ コンパイラーでは、LANGLVL(*EXTENDED) または LANGLVL(*LEGACY) が指定されている場合、関数の戻りの型として暗黙的に int がサポートされます。

メンバー関数だけが、括弧で囲まれたパラメーター・リストの後に、**const** または **volatile** 指定子を持つことができます。

exception_specification は、関数が指定されたリストの例外だけを throw するよう制限します。

関数の宣言時のその他の制限

省略符号 (**...**) を C++ における唯一の引数にすることもできます。この場合は、コンマは必要ありません。 C では、唯一の引数として省略符号を持つことはできません。

戻りまたは引数の型の中で、型は定義できません。例えば、C++ コンパイラーでは、`print()` の以下のような宣言ができます。

```
struct X { int i; };
void print(X x);
```

C コンパイラーでは、以下の宣言ができます。

```
struct X { int i; };
void print(struct X x);
```

C および C++ コンパイラーでは、同じ関数を次のように宣言することはできません。

```
void print(struct X { int i; } x);   //error
```

この例は、クラス `X` のオブジェクト `x` を引数として採用する関数 `print()` を宣言しようとしています。しかし、引数リスト内では、クラス定義はできません。

別の例では、C++ コンパイラーでは、`counter()` の以下のような宣言ができます。

```
enum count {one, two, three};
count counter();
```

同様に、C コンパイラーでは、以下の宣言ができます。

```
enum count {one, two, three};
enum count counter();
```

C および C++ のどちらのコンパイラーでも、同じ関数を次のように宣言することはできません。

```
enum count{one, two, three} counter();   //error
```

counter() の宣言の例では、関数宣言の戻りの型に列挙型定義を入れることはできません。

関連参照
- 60 ページの『型修飾子』
- 340 ページの『例外の指定』

C++ 関数の宣言

C++ C++ では、メンバー関数宣言で修飾子の **volatile** および **const** を指定することができます。また、関数宣言で例外指定を指定することもできます。すべての C++ 関数は、呼び出される前に、宣言されている必要があります。

関連参照
- 60 ページの『型修飾子』
- 226 ページの『const および volatile メンバー関数』
- 340 ページの『例外の指定』

複数の関数宣言

C++ 1 つの特定の関数に対する複数の関数宣言はすべて、パラメーターの数と型が同じでなければなりません。また、戻りの型も同じでなければなりません。

これらの戻りの型およびパラメーター型は、関数型の一部ですが、デフォルトの引数と例外指定は、関数型の一部ではありません。

すでになされたオブジェクトまたは関数の宣言が、囲みスコープで可視になっている場合は、 ID には、最初の宣言と同じリンケージが指定されています。ただし、リンケージを持たずに、後でリンケージ指定子を使用して宣言される変数または関数は、ユーザーが指定したリンケージを持ちます。

引数のマッチングの観点では、省略符号とリンケージ・キーワードは、関数型の一部と見なされます。これらは、関数のすべての宣言において、矛盾しないように使用する必要があります。 2 つの宣言におけるパラメーター型で、**typedef** 名または未指定の配列境界の使用だけが相違している場合、その宣言は同じです。 **const** または **volatile** 型修飾子も関数型の一部ですが、これらは、非静的メンバー関数の宣言または定義の一部となることができるだけです。

2 つの関数において、戻りの型およびパラメーター・リストの両方が一致している場合は、2 番目の宣言は最初の宣言の再宣言と見なされます。以下の例では、同じ関数が宣言されています。

```
int foo(const string &bar);
int foo(const string &);
```

戻りの型が違うだけの 2 つの関数宣言は、無効な関数多重定義となり、コンパイル時エラーのフラグが付けられます。次に例を示します。

```
void f();
int f();      // error, two definitions differ only in
              // return type
int g()
{
   return f();
}
```

関連参照

- 199 ページの『関数の多重定義』

関数宣言内のパラメーター名

C++ ユーザーが関数宣言でパラメーターの名前を付けることは可能ですが、次の 2 つの状態にある場合を除いて、コンパイラーはその名前を無視します。

1. 1 つの宣言内に同じ名前のパラメーターが 2 つある場合。これはエラーになります。
2. パラメーターの名前が、関数外の何かの名前と同じである場合。この場合、関数外の名前は隠され、パラメーター宣言でこの名前を使用することはできません。

次の例では、3 番目のパラメーター名 intersects は、列挙型 subway_line を持つことを意味します。しかし、この名前は、最初のパラメーターの名前によって隠蔽されています。関数 subway() の宣言は、subway_line が有効な型名ではないので、コンパイル時エラーを引き起こします。なぜ有効でないかというと、最初のパラメーター名 subway_line が、ネーム・スペース・スコープ **enum** 型を隠蔽し、2 番目のパラメーターで再度使用することができないからです。

```
enum subway_line {yonge,
university, spadina, bloor};
int subway(char * subway_line, int stations,
                  subway_line intersects);
```

関数宣言の例

次のコード・フラグメントは、複数の関数宣言を示しています。最初の部分は、2 つの整数の引数を採用し、戻りの型が void である関数 **f** を宣言しています。

```
void f(int, int);
```

次のコードは、固定文字へのポインターを使用して整数を戻す関数へのポインター p1 を宣言します。

```
int (*p1) (const char*);
```

次のコードは、関数 f1 を宣言し、関数 f1 は、1 つの整数の引数を取り、整数の引数を取って整数を戻す関数へのポインターを戻します。

```
int (*f1(int)) (int);
```

関数 **f1** の複雑な戻りの型に対して、上記の関数の代わりに、typedef を使用することができます。

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

次の宣言は、最初の引数として定数整数を取る外部関数 f2() の宣言です。この宣言は、可変数で可変型の他の引数を持つことができ、型 **int** を戻します。

```
int extern f2(const int ...);
```

ただし、C では、省略符号の前にコンマが必要です。

```
int extern f2(const int, ...);
```

関数 f3 は、戻りの型 **int** を持っています。そして、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) );
```

関数 f6 は、クラス X の **const** クラス・メンバー関数で、引数は取りません。**int** の戻りの型を持っています。

```
class X
{
public:
      int f6() const;
};
```

関数 f4 は、引数を取らず、戻りの型が **void** で、X 型および Y 型のクラス・オブジェクトをスロー (throw) することができます。

```
class X;
class Y;

// ...

void f4() throw(X,Y);
```

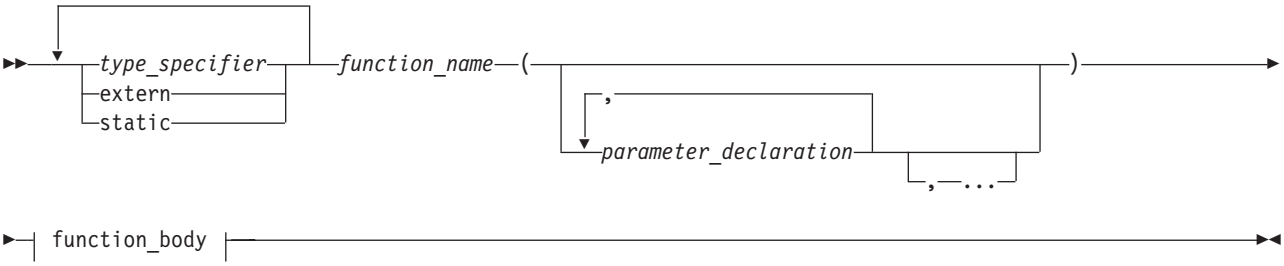
関数 f5 は、引数を取らないで、戻りの型 **void** を持っています。この関数は、任意の型の例外をスローする場合、unexpected() を呼び出します。

```
void f5() throw();
```

関数定義

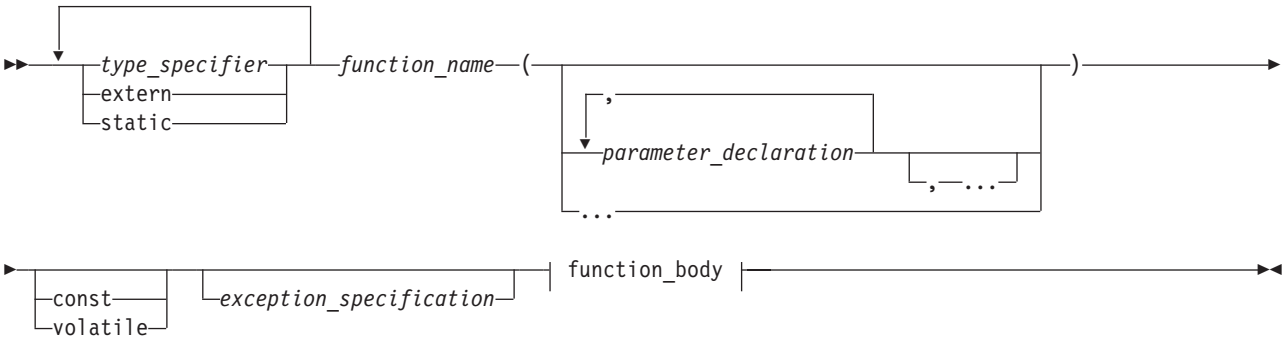
関数定義 には、関数宣言と関数本体が含まれます。

C C 関数定義の構文は以下のとおりです。



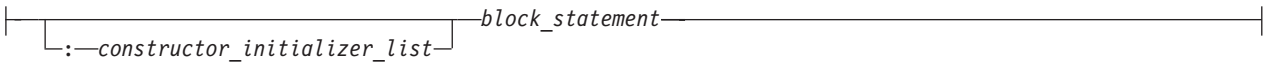
function_body:

C++ C++ 関数定義の構文は以下のとおりです。

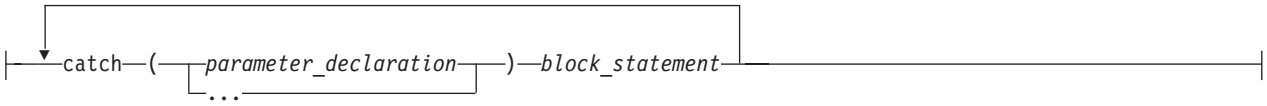


function_body:

body:



catch_handlers:



どちらの言語でも、関数定義には以下が含まれます。

- 少なくとも 1 つの*型指定子*。関数が返す値の型を決定します。例えば **unsigned long int** を返す関数の構文は、3 つの型の指定子を使用します。
- オプションのストレージ・クラス*指定子* **extern** または **static**。これは、その関数のスコープを決定します。ストレージ・クラス指定子が指定されない場合は、その関数には外部結合があります。
- *関数宣言子*は関数名であり、括弧で囲んだパラメーターの型と名前のリストが続き、各パラメーターは関数が必要とするものです。 以下の関数定義では、`f(int a, int b)` が関数宣言子です。

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

- *ブロック・ステートメント*。これには、データ定義とコードが含まれます。

C++ C++ 関数定義には、以下のものを任意に含めることができます。

- メンバー関数の関数宣言子の後の **const** または **volatile** 指定子。
- *例外指定*。これは、関数が、指定されたリストの例外だけを throw するよう制限します。
- *ブロック・ステートメント* の代わりに 1 つ以上のキャッチ・ハンドラー を持つ *try* ブロック。

- 関数定義がコンストラクターを対象としているとき、ブロック・ステートメントの前のコンストラクター*初期化指定子リスト*。クラス `A`、`x(0)`、`y('c')` の以下の定義は、コンストラクター初期化指定子リストです。

```
class A {
int x;
  char y;
public:
  A() : x(0), y('c') { }
};
```

関数は、それ自体によって、または他の関数によって呼び出すことができます。デフォルトで、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。 **static** のストレージ・クラス指定子は、関数名はグローバル・スコープだけを持っていて、同じ変換単位内からのみ直接起動できるということを意味します。

C++ **static** をこのように使用することは、C++ の場合は推奨できません。その代わりに、関数を名前なしネーム・スペースに入れます。

C C では、関数定義に外部結合と **int** 型の戻りの型がある場合は、`extern int func();` という暗黙の宣言が想定されるので、関数が可視になる前にその関数を呼び出すことができます。 C++ との互換性を保つためには、プロトタイプを使ってすべての関数を宣言する必要があります。

C 関数が値を戻さない場合は、型指定子としてキーワード **void** を使用してください。関数が引数をまったく取らない場合は、空のパラメーターではなく **void** を使用して、関数には引数が渡されないことを示してください。C では、空のパラメーター・リストを持つ関数は、受け取るパラメーター数が不明の関数を意味しますが、C++ では、その関数はパラメーターを取らないことを意味します。

C C では、関数を構造体または共用体のメンバーとして宣言することはできません。

関数宣言の互換性

所定の関数についての宣言はすべて互換性がなければなりません。つまり、戻りの型が同じで、パラメーターの型が同じです。

関数型の互換性

C 型の互換性という考え方は、C のみに関係があります。2 つの関数型に互換性を持たせるには、戻りの型に互換性を持たせる必要があります。両方の関数型がプロトタイプなしで指定されている場合は、このことが唯一の要件になります。

プロトタイプ付きで宣言されている 2 つの関数の場合は、複合型は次の追加要件を満たす必要があります。

- 関数型の 1 つがパラメーター型リストを持つ場合は、複合型は同じパラメーター型リストを持つ関数プロトタイプです。
- どちらの型もパラメーター・リストを持つ関数型の場合は、複合されるパラメーター・リスト内の各パラメーターは、対応するパラメーターの複合型です。

そして、宣言子指定子のシーケンスで [*] 表記を使用して、可変長配列型を指定することができます。

関数宣言子が関数定義の一部ではない場合は、パラメーターの型は不完全型を持つことができます。このパラメーターは、宣言子指定子のシーケンスで [*] 表記を使って、可変長の配列の型を指定することもできます。以下は、互換性のある関数プロトタイプ宣言子の例です。

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

関数定義の例

次の例は、関数 sum の定義です。

```
int sum(int x,int y)
{
   return(x + y);
}
```

関数 sum には、外部結合があり、**int** 型のオブジェクトを戻し、 x および y と宣言された **int** の 2 つのパラメーターがあります。この関数本体には、x と y の合計を戻す 1 個のステートメントが入っています。

以下の例において、ary は、2 つの関数ポインターの配列です。互換性を保つためには、この例では ary に割り当てられた値に対して、型キャストが行われます。

```
#include <stdio.h>

typedef void (*ARYTYPE)();

int func1(void);
void func2(double a);

int main(void)
{
  double num = 333.3333;
  int retnum;
  ARYTYPE ary[2];
  ary[0]=(ARYTYPE)func1;
  ary[1]=(ARYTYPE)func2;

  retnum=((int (*)())ary[0])();      /*  calls func1  */
  printf("number returned = %i¥n", retnum);
  ((void (*)(double))ary[1])(num);   /*  calls func2  */

  return(0);
}

int func1(void)
{
  int number=3;
  return number;
}

void func2(double a)
{
  printf("result of func2 = %f¥n", a);
}
```

次に、上記の例の出力を示します。

```
number
returned = 3
result of func2 = 333.333300
```

関連参照
- 31 ページの『extern ストレージ・クラス指定子』
- 33 ページの『static ストレージ・クラス指定子』
- 60 ページの『型修飾子』

省略符号および void

パラメーター指定の終わりにある省略符号は、関数が、可変数のパラメーターを持っていることを指定するために使用します。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。省略符号の前には、少なくとも 1 つのパラメーター宣言がなければなりません。

```
int f(int, ...);
```

C++ 省略符号の前のコンマは、オプションです。さらに、パラメーター宣言は、省略符号の前には必要ありません。

C 省略符号の前のコンマ、同じく省略符号の前のパラメーター宣言は、C では両方とも必須です。

C パラメーター拡張は必要に応じて行われます。ただし、型検査は、可変引数に対しては行われません。 10 進浮動小数点型では、パラメーター拡張は実行されません。

引数のない関数を、次の 2 つの方法で宣言することができます。

```
int f(void);
int f();
```

C++ 空の引数宣言や (void) の引数宣言リストは、引数を使用しない関数を示します。

C 空の引数宣言リストは、関数がパラメーターとして任意の数または型を取ることができるということを意味します。

void から派生した型 (**void** へのポインターなど) は使用できますが、型 **void** を引数型として使用することはできません。

次の例で、関数 f() は 1 つの整数引数を使用して値は戻しませんが、g() は引数がないことを予期しており、整数を戻します。

```
void f(int);
int g(void);
```

関数定義の例

以下の例には、**int** へのポインターとして宣言された table と、**int** 型として宣言された length が指定されている関数宣言子、i_sort が入っています。パラメーターとしての配列を、暗黙的にエレメント型へのポインターに変換することに注意してください。

```
/**
 ** This example illustrates function definitions.
 ** Note that arrays as parameters are implicitly
 ** converted to a pointer to the type.
 **/

#include <stdio.h>

void i_sort(int table[ ], int length);

int main(void)
```

```
{
   int table[ ]={1,5,8,4};
   int length=4;
   printf("length is %d\n",length);
   i_sort(table,length);
}
void i_sort(int table[ ], int length)
{
  int i, j, temp;

  for (i = 0; i < length -1; i++)
    for (j = i + 1; j < length; j++)
      if (table[i] > table[j])
      {
        temp = table[i];
        table[i] = table[j];
        table[j] = temp;
      }
}
```

次の例は、関数宣言 (また、*関数プロトタイプ* とも呼ばれます) の例です。

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

次の例は、関数宣言子で **typedef** ID を使用する方法を示しています。

```
typedef struct tm_fmt { int minutes;
                        int hours;
                        char am_pm;
                      } struct_t;
long time_seconds(struct_t arrival)
```

次の関数 set_date は、date 型の構造体へのポインターをパラメーターとして宣言します。 date_ptr は、ストレージ・クラス指定子 **register** を持っています。

```
void set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

main() 関数

プログラムが実行を開始すると、システムは main 関数を呼び出します。この関数は、プログラムのエントリー・ポイントを表します。どのプログラムにも、main という名前の関数が 1 つなければなりません。プログラムにあるそれ以外の関数を main と呼ぶことはできません。 main 関数の形式は、次の 2 つのうちいずれかです。

C `int main (void) block_statement`

C++ `int main ( )block_statement`

`int main (int argc, char ** argv)block_statement`

引数 argc は、プログラムに渡されたコマンド行引数の数です。引数 argv は、ストリングの配列を指すポインターです。ここで、argv[0] は、コマンド行からプログラムを実行するために使用した名前です。argv[1] はプログラムに渡された最初の引数、argv[2] は 2 番目の引数、等々です。

デフォルトでは、main は、ストレージ・クラス **extern** を持ちます。

C++ main を、**inline** または **static** として宣言することはできません。プログラムの中から main を呼び出したり、main のアドレスを使用したりすることはできません。この関数は、多重定義することはできません。

main への引数

関数 main は、パラメーターの指定ありでも、なしでも宣言することができます。

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

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、**int** 型で、コマンド行に入力する引数の数を示します。

2 番目のパラメーターの argv (引数ベクトル) の型は、**char** 配列オブジェクトへのポインターの配列型です。 **char** 配列オブジェクトは、ヌル終了ストリングです。

argc の値は、配列 argv 内のポインターの数を示します。プログラム名が使用可能な場合、argv 内の最初のエレメントは、文字配列を指し、この配列には、プログラム名または実行しているプログラムの起動名が含まれています。名前が判別不可能な場合、argv 内の最初のエレメントはヌル文字を指します。

この名前は、関数 main への引数の 1 つとしてカウントされます。例えば、プログラム名だけをコマンド行に入力すると、argc の値は 1 で、argv[0] は、そのプログラム名を指します。

コマンド行に入力された引数の数に関係なく、argv[argc] には、常に NULL が入っています。

main への引数の例

以下のプログラム backward は、コマンド行に入力された引数を、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]);
}
```

以下を指定してコマンド行からこのプログラムを呼び出します。

```
   backward string1 string2
```

出力は、次のとおりです。

```
      string2 string1
```

引数の argc と argv には以下の値が入ります。

オブジェクト	値
argc	3
argv[0]	ストリング "backward" を指すポインター
argv[1]	ストリング "string1" を指すポインター
argv[2]	ストリング "string2" へのポインター
argv[3]	NULL

注: 環境によって大文字小文字の区別をするものとしないものとがあるため、コマンド行に大文字小文字混合文字を入力する際は注意してください。また、argv[0] が指すストリングの正確な形式もシステムによって異なります。

関数呼び出しおよび引数の受け渡し

関数呼び出しの引数は、関数定義のパラメーターを初期化するために使用されます。 引数としての配列式および C の関数指定子は、呼び出しの前にポインターに変換されます。

最初に、整数拡張および浮動小数点拡張が、関数が呼び出される前に、引数の値に対して行われます。

引数の型は、関数宣言の対応するパラメーターの型に照らし合わせて検査されます。 必要に応じて、すべての標準型変換およびユーザー定義の型変換が適用されます。 各引数式の値は、代入による場合のように、対応するパラメーターの型に変換されます。

次に例を示します。

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

/* Declaration */
extern double root(double, double);

/* Definition */
double root(double value, double base) {
  double temp = exp(log(value)/base);
  return temp;
}

int main(void) {
  int value = 144;
  int base = 2;
  printf("The root is: %f¥n", root(value, base));
  return 0;
}
```

出力は、The root is: 12.000000 となります。

上記の例では、関数 root は **double** 型の引数を予期していますので、 value と base の 2 つの **int** 引数は、この関数を呼び出すと、暗黙的に **double** 型に変換されます。

引数が評価されて関数に渡される順序は、インプリメンテーションでの定義に依存します。例えば、以下の一連のステートメントで、関数 tester を呼び出します。

```
int x;
x = 1;
tester(x++, x);
```

この例のような tester の呼び出しは、別のコンパイラーでは、異なる結果を生む場合があります。インプリメンテーションによって、x++ が最初に評価されるか、または x が最初に評価されるかが決まります。このようなあいまいさを避けて、x++ が最初に評価されるようにするには、前述の一連のステートメントを次のように置き換えてください。

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

C++ このセクションでのここから先の説明は、C++ だけに適用されます。

非静的クラス・メンバー関数を引数として渡すと、この引数はメンバーへのポインターに変換されます。

クラスに、デストラクターまたはビット単位のコピー以上を行うコピー・コンストラクターがある場合、クラス・オブジェクトを値で渡すと、実際には参照によって渡される一時オブジェクトが構築されます。

関数引数がクラス・オブジェクトであり、以下の特性のすべてを持っている場合には、これはエラーです。
- クラスが copy コンストラクターを必要としている。
- クラスに、ユーザー定義の copy コンストラクターがない。
- そのクラス用に copy コンストラクターを生成することができない。

値による引数の受け渡し

非参照パラメーターに対応する引数を使用して関数を呼び出す場合、値によってその引数を渡したことになります。パラメーターは、引数の値を使用して初期化されます。関数のスコープ内のパラメーターの値を変更することができます (そのパラメーターが **const** と宣言されていなければ)。ただし、これらの変更は、呼び出す方の関数の引数の値には、影響しません。

以下は、値による引数の受け渡しの例です。

次のステートメントは、関数 **printf** を呼び出します。この関数は、文字ストリングと、(a および b の値を受け取る) 関数 sum からの戻り値を受け取ります。

```
printf("sum = %d¥n", sum(a,b));
```

以下のプログラムは、count の値を関数 increment に渡します。関数は、パラメーター x の値を 1 ずつ増やします。

```
/**
 ** An example of passing an argument to a function
 **/

#include <stdio.h>

void increment(int);

int main(void)
{
  int count = 5;

  /* value of count is passed to the function */
  increment(count);
  printf("count = %d¥n", count);

  return(0);
}

void increment(int x)
{
  ++x;
  printf("x = %d¥n", x);
}
```

出力は、main の count の値が未変更のままであることを示しています。

```
x = 6
count = 5
```

関連参照
- 88 ページの『関数呼び出し演算子 ( )』

参照による引数の受け渡し

参照による受け渡し は、呼び出す方の関数の引数の値が、呼び出される関数の中で変更できる、引数受け渡しのメソッドを指しています。

参照によって引数を渡すには、対応するパラメーターを参照型を使用して宣言します。

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ることに注意してください。

```
#include <stdio.h>

void swapnum(int &i, int &j) {
  int temp = i;
  i = j;
  j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;

  swapnum(a, b);
  printf("A is %d and B is %d¥n", a, b);
  return 0;
}
```

関数 `swapnum()` の呼び出し時に、変数 `a` および `b` の実際の値は、参照によって渡されているため、交換されます。出力は次のとおりです。

```
A is 20 and B is 10
```

実引数の値を関数 `swapnum()` で変更させるには、`swapnum()` のパラメーターを参照として定義する必要があります。

C++ **const** で修飾されている参照を変更するためには、**const_cast** 演算子を使用して、その const 型をキャストする必要があります。次の例は、このことを示しています。

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

void f(const int& x) {
  int* y = const_cast<int>(&x);
  (*y)++;
}

int main() {
  int a = 5;
  f(a);
  cout << a << endl;
}
```

この例では `6` を出力します。

ポインター・パラメーターによって、非定数オブジェクトの値を変更することができます。次の例は、このことを示しています。

```
#include <stdio.h>

int main(void)
{
  void increment(int *x);
  int count = 5;
```

```
  /* address of count is passed to the function */
  increment(&count);
  printf("count = %d¥n", count);

  return(0);
}

void increment(int *x)
{
  ++*x;
  printf("*x = %d¥n", *x);
}
```

次に、上記のコード出力を示します。

```
*x = 6
count = 6
```

例では、count のアドレスを increment() に渡します。関数 increment() は、ポインター・パラメーター x によって、count を増分します。

C++ 関数におけるデフォルト引数

C++ 関数パラメーターに対してデフォルト値を与えることができます。次に例を示します。

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

int a = 1;
int f(int a) { return a; }
int g(int x = f(a)) { return x; }

int h() {
  a = 2;
  {
    int a = 3;
    return g();
  }
}

int main() {
  cout << h() << endl;
}
```

この例では、標準出力に 2 を印刷します。その理由は、 g() の宣言で参照される a は、ファイル・スコープのもので、この値は、g() が呼び出されるときは 2 であるためです。

デフォルト引数は、暗黙的に、パラメーター型へ変換可能でなければなりません。

関数へのポインターは、その関数と同じ型でなければなりません。関数の型を指定せずに参照によって関数のアドレスを得ようとすると、エラーになります。関数の型は、デフォルト値を持つ引数によって影響を受けません。

以下の例は、デフォルト引数が、関数の型の一部とは見なされていないことを示しています。デフォルト引数を使用すると、すべての引数を指定しなくても関数を呼び出すことができます。すべての引数の型を指定しない関数へのポインターを作成することはできません。関数 f は、明示的引数がなくても呼び出すことができますが、ポインター badpointer は、引数の型を指定しないと定義することができません。

```
int f(int = 0);
void g()
{
   int a = f(1);               // ok
   int b = f();                // ok, default argument used
}
int (*pointer)(int) = &f;      // ok, type of f() specified (int)
int (*badpointer)() = &f;      // error, badpointer and f have
                               // different types. badpointer must
                               // be initialized with a pointer to
                               // a function taking no arguments.
```

関連参照

- 155 ページの『関数へのポインター』

デフォルト引数に関する制約事項

演算子の中で、多重定義時にデフォルト引数を持つことができるのは、関数呼び出し演算子と演算子 **new** だけです。

デフォルト引数を持つパラメーターは、関数宣言パラメーター・リスト内の末尾のパラメーターでなければなりません。次に例を示します。

```
void f(int a, int b = 2, int c = 3);  // trailing defaults
void g(int a = 1, int b = 2, int c);  // error, leading defaults
void h(int a, int b = 3, int c);      // error, default in middle
```

いったん宣言または定義でデフォルト引数を指定すると、その引数を、同じ値にする場合であっても、再定義することはできません。ただし、それまでの宣言で指定されていないデフォルト引数を追加することはできます。例えば、次の例の最後の宣言では、`a` および `b` に対するデフォルト値を再定義しようとしています。

```
void f(int a, int b, int c=1);     // valid
void f(int a, int b=1, int c);     // valid, add another default
void f(int a=1, int b, int c);     // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

関数宣言または定義では、いかなるデフォルト引数値でも与えることができます。デフォルト引数値に続くパラメーター・リストの中のパラメーターはすべて、関数の、この宣言または前の宣言に指定されたデフォルト引数値を持っていなければなりません。

デフォルト引数の式では、ローカル変数を使用することはできません。例えば、コンパイラーは、次の `g()` および `h()` の両方の関数に対してエラーとします。

```
void f(int a)
{
      int b=4;
      void g(int c=a); // Local variable "a" cannot be used here
      void h(int d=b); // Local variable "b" cannot be used here
}
```

関連参照

- 88 ページの『関数呼び出し演算子 ( )』
- 103 ページの『C++ の new 演算子』
- 150 ページの『C++ 関数におけるデフォルト引数』

デフォルト引数の評価

デフォルト引数を使用して定義された関数が、末尾の引数が欠落している状態で呼び出されると、デフォルト式が評価されます。次に例を示します。

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);            // same as call f(a,2,3)
f(a,10);         // same as call f(a,10,3)
f(a,10,20);      // no default arguments
```

デフォルト引数は、関数宣言に対して検査されます。そして、関数が呼び出されるときに評価されます。デフォルトの引数の評価の順序は、定義されていません。デフォルト引数式では、関数の他のパラメーターは使用できません。次に例を示します。

```
int f(int q = 3, int r = q); // error
```

q の値は、r に代入されるときには決まっていない場合があるので、引数 r を引数 q の値で初期化することはできません。上記の関数宣言を、次のように書き直すものとします。

```
int q=5;
int f(int q = 3, int r = q); // error
```

関数宣言内の r の値はやはりエラーとなります。それは、関数の外側で定義された変数 q が、関数に対して宣言されている引数 q によって隠されているからです。同様に、

```
typedef double D;
int f(int D, int z = D(5.3) ); // error
```

ここでは、型 D は、関数宣言内で整数の名前として解釈されます。型 D は、引数 D により隠されます。したがって、D が引数の名前であり、型ではないため、キャスト D(5.3) が、キャストとして解釈されません。

次の例では、非静的メンバー a を初期化指定子として使用できません。 a は、クラス X のオブジェクトが作成されるまで存在しないからです。 b は、クラス X のどのオブジェクトとも無関係に作成されるので、静的メンバー b を初期化指定子として使用することができます。デフォルト値は、クラス宣言の最後の大括弧 } の後まで分析されないので、メンバー b をデフォルト引数として使用した後で、それを宣言することができます。

```
class X
{
   int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

関連参照

- 150 ページの『C++ 関数におけるデフォルト引数』

関数からの戻り値

関数の戻りの型が **void** でない限り、関数は値を戻す必要があります。

戻り値は **return** ステートメントで指定します。以下のコードは、**return** ステートメントを含む関数定義を示しています。

```
int add(int i, int j)
{
  return i + j; // return statement
}
```

以下のコードで示すように、関数 add() を呼び出すことができます。

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

この例では、return ステートメントは、戻された型の変数を初期化します。変数 answer を **int** の値 30 で初期化しています。戻された式の型を、戻りの型と照らし合わせて検査します。必要に応じて、すべての標準型変換およびユーザー定義の型変換が適用されます。

関数が呼び出されるたびに、自動ストレージを持つその変数の新しいコピーが作成されます。関数が終了した後に、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。

関連参照
- 171 ページの『return ステートメント』
- 172 ページの『戻り式の値および関数値』
- 287 ページの『一時オブジェクト』

戻りの型としての参照の使用

関数に対する戻りの型として、参照も使用できます。参照は、参照しているオブジェクトの左辺値を戻します。これにより、関数呼び出しを代入ステートメントの左側に配置することができます。

C++ 参照された戻り値は、代入演算子およびサブスクリプト演算子が多重定義され、多重定義された演算子の結果を実際の値として使用できるようになるときに、使用されます。

注: 自動変数に参照を戻すと、予測できない結果となります。

割り当ておよび割り当て解除関数

C++ ユーザーは、クラス・メンバー関数またはグローバル・ネーム・スペース関数としての、ユーザー自身の new 演算子あるいは割り振り関数を、以下の制限の下で定義することができます。
- 最初のパラメーターは、型 **std::size_t** のパラメーターでなければなりません。デフォルトのパラメーターを持つことはできません。
- 戻りの型は、型 **void*** でなければなりません。
- 割り振り関数はテンプレート関数でもかまいません。最初のパラメーターも、戻りの型もテンプレート・パラメーターに依存しません。
- 空の例外指定 throw() を指定して割り振り関数を宣言する場合、関数が失敗すると、割り振り関数はヌル・ポインターを戻す必要があります。そうしなければ、ユーザーの関数は、失敗した場合には、型 **std::bad_alloc** の例外または **std::bad_alloc** から派生したクラスを throw する必要があります。

ユーザーは、ユーザー自身の delete 演算子、あるいはクラス・メンバー関数またはグローバル・ネーム・スペース関数としての割り振り解除関数を、以下の制限の下で定義することができます。

- この最初のパラメーターの型は **void*** でなければなりません。
- 戻りの型は、型 **void** でなければなりません。
- 割り振り解除関数はテンプレート関数でもかまいません。最初のパラメーターも、戻りの型もテンプレート・パラメーターに依存しません。

次の例では、グローバル・ネーム・スペース **new** および **delete** の置換関数を定義しています。

```
#include <cstdio>
#include <cstdlib>

using namespace std;

void* operator new(size_t sz) {
  printf("operator new with %d bytes¥n", sz);
  void* p = malloc(sz);
  if (p == 0) printf("Memory error¥n");
  return p;
}

void operator delete(void* p) {
  if (p == 0) printf ("Deleting a null pointer¥n");
  else {
    printf("delete object¥n");
    free(p);
  }
}

struct A {
  const char* data;
  A() : data("Text String") { printf("Constructor of S¥n"); }
  ~A() { printf("Destructor of S¥n"); }
};

int main() {
  A* ap1 = new A;
  delete ap1;

  printf("Array of size 2:¥n");
  A* ap2 = new A[2];
  delete[] ap2;
}
```

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
```

関連参照

- 283 ページの『フリー・ストレージ』

関数へのポインター

関数へのポインターは、その関数の実行可能コードのアドレスを示します。ポインターを使用して関数を呼び出し、関数を引数として他の関数に渡すことができます。関数へのポインターに対してポインター演算を行うことはできません。

関数の戻りの型とパラメーター型の両方によって、関数へのポインターの型が決まります。

関数へのポインターの宣言では、ポインター名を小括弧に入れることが必要です。関数呼び出し演算子 () は、間接参照演算子 * よりも高い優先順位を持っています。括弧がないと、コンパイラーは、指定された戻りの型へのポインターを戻す関数として、そのステートメントを解釈します。次に例を示します。

```
int *f(int a);       /* function f returning an int*                        */
int (*g)(int a);     /* pointer g to a function returning an int            */
char (*h)(int, int)  /* h is a function
                        that takes two integer parameters and returns char */
```

最初の宣言では、f は、**int** を引数として取り、 **int** へのポインターを戻す関数として解釈します。 2 番目の宣言では、g を、**int** 引数を受け取り、**int** を戻す関数へのポインターと解釈します。

関連参照

- 128 ページの『ポインター型変換』

インライン関数

インライン関数とは、コンパイラーが、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 関数指定子を用いるか、あるいはクラスまたは構造体定義の中でメンバー関数を定義することによって、関数をインラインとして宣言します。 **inline** 指定子は、インライン展開が実行可能であることをコンパイラーに提案するものにすぎません。コンパイラーはこの提案を無視してもかまいません。

以下のコードは、インライン関数の定義を示しています。

```
inline int add(int i, int j) { return i + j; }
```

inline 指定子を使用しても、関数の意味は変わりません。ただし、関数のインライン展開を行うと、実引数の評価の順序が保たれなくなる場合があります。 また、インライン展開によって関数のリンケージが変更されることもありません。リンケージはデフォルトでは外部になっています。

C++ C++ では、メンバー関数と非メンバー関数は、両方ともインラインにすることができます。クラス宣言の本体内部にインプリメントされたメンバー関数は、暗黙で inline を宣言されます。コンパイラーによって作成されるコンストラクター、コピー・コンストラクター、代入演算子、およびデストラクターも、暗黙で inline を宣言されます。コンパイラーがインラインにしない inline 関数は、通常の関数と同じように処理されます。つまり、その関数が定義されている変換単位の数に関係なく、関数のコピーは 1 つしか存在しません。

コンパイラーは **inline** 関数指定子があっても、extern inline 関数 **two** をインラインにしない場合もあります。

関連参照

インライン関数

- 224 ページの『メンバー関数』
- 31 ページの『extern ストレージ・クラス指定子』

第 8 章 ステートメント

ステートメントは最も小さな独立した計算単位で、実行される処理を指定します。ほとんどの場合、ステートメントは順序どおりに実行されます。 以下に、C および C++ で使用可能なステートメントの要約を示します。

- ラベル付きステートメント
  - ID ラベル
  - **case** ラベル
  - **default** ラベル
- 式ステートメント
- ブロック・ステートメントまたは複合ステートメント
- 選択ステートメント
  - **if** ステートメント
  - **switch** ステートメント
- 繰り返しステートメント
  - **while** ステートメント
  - **do** ステートメント
  - **for** ステートメント
- 分岐ステートメント
  - **break** ステートメント
  - **continue** ステートメント
  - **return** ステートメント
  - **goto** ステートメント
- 宣言ステートメント
- C++ **try** ブロック

ラベル

ラベルには identifier、case、および default の 3 つの種類があります。

identifier ラベル・ステートメントの形式は、次のとおりです。

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することができます。 **goto** ステートメントでは、ラベルをその定義よりも前に使用することができます。 identifier ラベルは、それ自体のネーム・スペースを持っています。identifier ラベルが、他の ID と競合することについて心配する必要はありません。 ただし、関数内ではラベルを再宣言することはできません。

case および default ラベル・ステートメントは、**switch** ステートメントでのみ使用されます。 これらのラベルは、最も近い **switch** ステートメント内でのみアクセス可能です。

ラベル

case ステートメントの形式は、次のとおりです。

default ステートメントの形式は、次のとおりです。

ラベルの例

```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```

関連参照

- 172 ページの『goto ステートメント』
- 162 ページの『switch ステートメント』

式ステートメント

式ステートメント は、式を含んでいます。式は、ヌルでもかまいません。

式ステートメントの形式は、次のとおりです。

式ステートメントは式 を評価し、次に式の値を破棄します。式のない式ステートメントは、ヌル・ステートメントです。

式の例

```
printf("Account Number: ¥n");          /* call to the printf    */
marks = dollars * exch_rate;               /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

関連参照

- 79 ページの『第 5 章 式と演算子』

C++ でのあいまいなステートメントの解決

C++ C++ 構文は、式ステートメントと宣言ステートメントの間のあいまいさを明確化していません。式ステートメントの左端の副次式に関数スタイル・キャストがあると、あいまいさが生じます。 (C では、関数スタイルのキャストをサポートしないため、C プログラムではこのようなあいまいさは起きません。) ステートメントを宣言または式のいずれにも解釈できる場合、ステートメントは宣言ステートメントとして解釈されます。

注: あいまいさは、構文レベルでのみ解決されます。明確化に際して、名前の意味が型名であるかどうかを評価する場合を除いて、名前の意味を使用しません。

以下の式は、あいまいな副次式のあとに、代入または演算子が続いているため、式ステートメントとして解決されます。これらの式の type_spec は、任意の型指定子にすることができます。

```
type_spec(i)++;             // expression statement
type_spec(i,3)<<d;          // expression statement
type_spec(i)->l=24;         // expression statement
```

以下の例では、あいまいさを構文的に解決できません。コンパイラーは、ステートメントを宣言として解釈します。 type_spec は、任意の型指定子です。

```
type_spec(*i)(int);         // declaration
type_spec(j)[5];            // declaration
type_spec(m) = { 1, 2 };    // declaration
type_spec(*k) (float(3));   // declaration
```

上記の最後のステートメントは、浮動値を用いてポインターを初期化することができないため、コンパイル時エラーとなります。

上記の規則で解析されないあいまいなステートメントは、デフォルトにより宣言ステートメントであると見なされます。以下のステートメントはすべて宣言ステートメントです。

```
type_spec(a);               // declaration
type_spec(*b)();            // declaration
type_spec(c)=23;            // declaration
type_spec(d),e,f,g=0;       // declaration
type_spec(h)(e,3);          // declaration
```

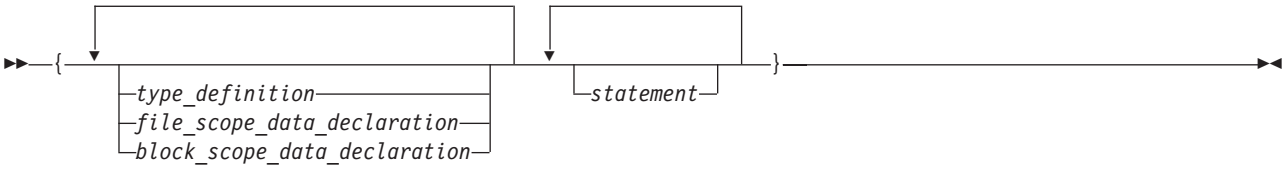
関連参照

- 27 ページの『第 3 章 宣言』
- 79 ページの『第 5 章 式と演算子』
- 88 ページの『関数呼び出し演算子 ( )』

ブロック・ステートメント

ブロック・ステートメント または*複合ステートメント* を使用すると、任意の数のデータ定義、宣言、およびステートメントを 1 つのステートメントにグループ化します。 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・ステートメントの形式は、次のとおりです。



C C89 言語レベルでは、定義および宣言はステートメントの前に配置しなければなりません。

C++ 宣言および定義を任意の位置に配置し、他のコードと混在させることができます。

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

ブロック・ステートメント

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。

if ステートメント

if ステートメントは、複数の制御フローが可能な選択ステートメントです。

C++ *if* ステートメント を使用すると、指定されたテスト式 (暗黙的に **bool** に変換されている) が **真** に評価されたときに、ステートメントを条件付きで処理することができます。 **bool** への暗黙的な変換が失敗した場合は、プログラムが不適格です。

C C では、**if** ステートメントを使用すると、指定されたテスト式の評価が非ゼロ値になった場合に、ステートメントを条件付きで処理することができます。テスト式は、算術型またはポインター型でなければなりません。

オプションで、**if** ステートメントで **else** 節を指定することができます。テスト式の評価が **false** (または C では、ゼロ値) になり、**else** 文節がある場合、**else** 文節に関連付けられているステートメントが実行されます。テスト式の評価が **true** になった場合、その式に続くステートメントが実行され、**else** 文節は無視されます。

if ステートメントの形式は、次のとおりです。

if ステートメントがネストされていて、**else** 節が存在する場合、指定された **else** は、同じブロック内の直前の **if** ステートメントに関連付けられます。

任意の選択ステートメント (**if**、**switch**) に続く単一のステートメントは、オリジナルのステートメントを含んでいる複合ステートメントとして扱われます。結果として、そのステートメントで宣言されたすべての変数は、**if** ステートメントの後、スコープの外にあります。次に例を示します。

```
if (x)
int i;
```

は、以下と同等です。

```
if (x)
{  int i; }
```

変数 i は、**if** ステートメント内でのみ可視です。同じ規則が **if** ステートメントの **else** 部分にも適用されます。

if ステートメントの例

以下の例では、score の値が 90 以上である場合に、grade が A という値を受け取るようにします。

```
if (score >= 90)
   grade = 'A';
```

以下の例では、number の値が 0 またはそれ以上である場合に Number is positive と表示します。number の値が 0 より小さい場合には、 Number is negative と表示します。

```
if (number >= 0)
   printf("Number is positive¥n");
else
   printf("Number is negative¥n");
```

以下の例は、ネストされた **if** ステートメントを示しています。

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
cout << "salary is " << salary << endl;
```

以下の例は、**else** 節を持たない、ネストされた **if** ステートメントを示しています。 **else** 節は、常に、最も近い **if** ステートメントに関連付けられるため、特定の **else** 節を強制的に正しい **if** ステートメントに関連付けるには、中括弧が必要なことがあります。この例では、中括弧を省略すると、**else** 節は、ネストされた **if** ステートメントに関連付けられることになります。

```
if (kegs > 0) {
   if (furlongs > kegs)
      fpk = furlongs/kegs;
}
else
   fpk = 0;
```

以下の例は、**else** 節の中にネストされた **if** ステートメントを示しています。この例では、複数の条件がテストされます。テストは、それらの条件が書かれている順序で行われます。 1 つのテストが非ゼロ値に評価されると、ステートメントが実行され、**if** ステートメント全体が終了します。

if ステートメント

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

switch ステートメント

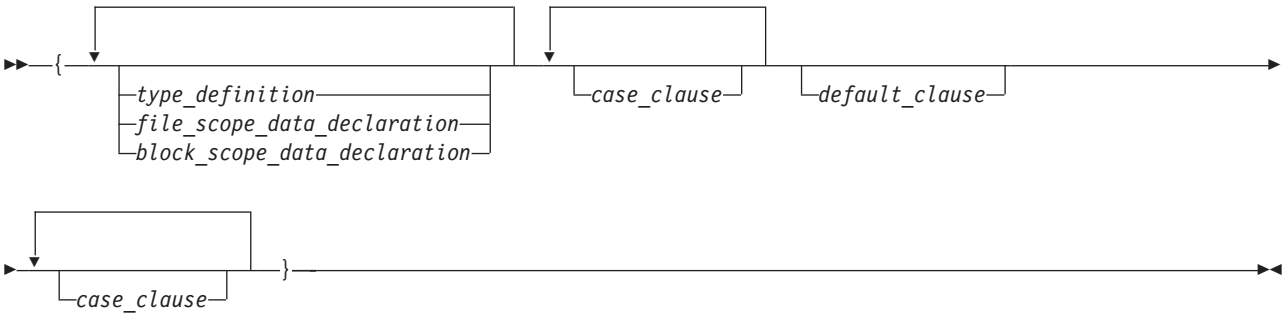
switch ステートメント は、switch 式の値に応じて、**switch** 本体内の別のステートメントに制御を移す選択ステートメントです。 **switch** 式の評価は、整数値または列挙値にならなければなりません。 **switch** ステートメントの本体には、以下で構成される *case 文節* が含まれています。

- **case** ラベル
- オプションの **default** ラベル
- **case** 式
- ステートメントのリスト。

switch 式の値が case 式の 1 つの値と同じ場合、その case 式に続くステートメントが処理されます。そうでない場合、default ラベル・ステートメント (あれば) が処理されます。

switch ステートメントの形式は、次のとおりです。

switch 本体 は、中括弧で囲まれ、定義、宣言、*case 文節*、および *default 文節* を含むことができます。case 文節と default 文節のそれぞれにステートメントを含めることができます。

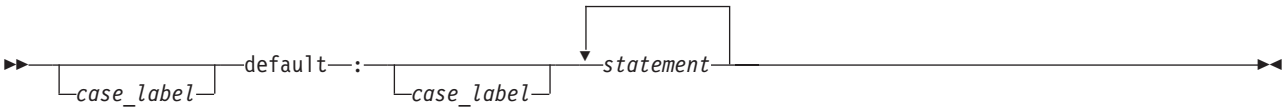


注: *type_definition*、*file_scope_data_declaration* または *block_scope_data_declaration* 内の初期化指定子は、無視されます。

case 文節 には、任意の数のステートメントが続く *case label* が含まれます。 case 文節の形式は、次のとおりです。

case ラベル には、**case** というワードと、それに続く整数定数式とコロンが入っています。 各整数定数式の値は、別々の値を表している必要があります。重複した **case** ラベルを持つことはできません。1 つの **case** ラベルを置けるところであればどこでも、複数の **case** ラベルを置くことができます。 case ラベルの形式は、次のとおりです。

default 文節 には、**default** ラベルと、それに続く 1 つまたは複数のステートメントが入っています。 **case** ラベルは、**default** ラベルのどちらの側にも置くことができます。 **switch** ステートメントに入れることができる **default** ラベルは、1 つだけです。 *default_clause* の形式は、次のとおりです。



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。

制約および制限事項

データ定義を **switch** 本体の先頭に置くことができますが、コンパイラーは、**switch** 本体の先頭にある **auto** および **register** 変数は、初期化しません。 **switch** 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。

C++ では、暗黙的または明示的な初期化指定子を含んでいる宣言を超えて、制御権を移動することはできません。ただし、宣言が、制御権の移動によって完全にう回される内部ブロックに入っている場合は、制御権を移動することができます。初期化指定子が入っている **switch** ステートメント本体内の宣言はすべて、内部ブロックに入れる必要があります。

switch ステートメントの例

switch ステートメント

以下の **switch** ステートメントには、複数の **case** 文節と 1 つの **default** 文節が入っています。各文節には、関数呼び出しと **break** ステートメントが入っています。**break** ステートメントは、**switch** 本体内の各ステートメントに制御が移されていくのを防止します。

switch 式の評価が '/' となった場合、その switch ステートメントは関数 divide を呼び出すことになります。その後、**switch** 本体に続くステートメントに制御が渡されます。

```
char key;

printf("Enter an arithmetic operator¥n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;

     default:
      printf("invalid key¥n");
      break;
}
```

switch 式と case 式が一致した場合には、**break** ステートメントが検出されるまで、または **switch** 本体の終わりに達するまで、case 式に続くステートメントが処理されます。以下の例には、**break** ステートメントはありません。 text[i] の値が 'A' である場合、コンパイラーは、3 つのカウンターすべてを増やします。 text[i] の値が 'a' と等しい場合には、lettera と total が増やされます。 text[i] が 'A' または 'a' と等しくない場合には、total のみが増やされます。

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

次の **switch** ステートメントは、複数の **case** ラベルに対して同じステートメントを実行します。

```
/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
```

```
 **/

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
     case 2:
        printf("month %d is a winter month¥n", month);
      break;

     case 3:
     case 4:
     case 5:
        printf("month %d is a spring month¥n", month);
      break;

     case 6:
     case 7:
     case 8:
        printf("month %d is a summer month¥n", month);
      break;

     case 9:
     case 10:
     case 11:
        printf("month %d is a fall month¥n", month);
      break;

     case 66:
     case 99:
     default:
        printf("month %d is not a valid month¥n", month);
  }

   return(0);
}
```

式 month の値が 3 の場合には、次のステートメントに制御が渡されます。

```
printf("month %d is a spring month¥n",
month);
```

break ステートメントは、**switch** 本体に続くステートメントに制御を渡します。

while ステートメント

while ステートメント は、制御式の評価が **false** (または C では 0) になるまで、ループの本体を繰り返し実行します。

while ステートメントの形式は、次のとおりです。

while ステートメント

C *expression* は、算術型またはポインター型でなければなりません。

式は評価されて、ループの本体を処理するかどうかを判別します。

C++ expression は、**bool** に変換可能なものでなければなりません。

式の評価が **false** の場合、ループの本体は実行されません。式が **false** に評価されないと、ループ本体は処理されます。本体が実行された後、制御は式に戻されます。それ以降の処理は、条件の値によって決まります。

break、**return**、または **goto** 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。

while ステートメントの例

次のプログラムでは、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 ステートメントの形式は、次のとおりです。

C++ 制御する *expression* は、型 **bool** に変換可能でなければなりません。

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。

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。

do ステートメントの例

次の例では、i が 5 より小さい間は、i を増分し続けます。

```
#include <stdio.h>

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

次に、上記の例の出力を示します。

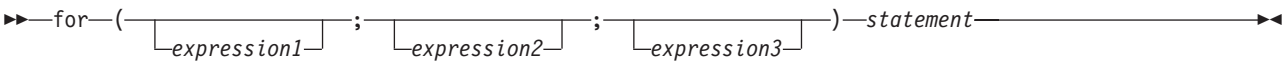
```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

for ステートメント

for ステートメント を使用すると、以下のことを行うことができます。

- ステートメントの最初の反復の前に式を評価する。(*初期化*)
- 式を指定して、ステートメントを処理するかどうかを判別する (*条件*)
- ステートメントが反復されるたびに、その後で式を評価する (反復のための増分によく使用される)
- 制御部分の評価が **false** (C では、0) にならない場合に、ステートメントを繰り返し処理する。

for ステートメントの形式は、次のとおりです。



expression1　*初期化式* です。これは、ステートメント が始めて処理される前においてのみ評価されます。この式を使用すると、変数を初期化することができます。ステートメントの最初の反復の前に式の評価を行いたくない場合には、この式を省略することができます。

expression2　*条件式* です。ステートメント の各反復の前に評価されます。

C これは、算術型またはポインター型でなければなりません。

評価が **false** (または C では 0) であった場合、そのステートメントは処理されず、制御は **for** ステートメントの次のステートメントに移ります。 *expression2* の評価が **false** でない場合、ステートメントは処理されます。*expression2* を省略すると、この式が **true** によって、置き換えられたのと同様になり、この条件の不備により **for** ステートメントが終了しないことになります。

expression3 ステートメント が反復された後で、毎回この式を評価します。この式は、変数に対する増分、減分、または代入のために頻繁に使用されます。この式はオプションです。

break、**return**、または **goto** ステートメントを使用すると、 2 番目の式の評価が **false**でなくても **for** ステートメントが終了させられます。*expression2* を省略する場合には、**break**、**return**、または **goto** ステートメントを使用して **for** ステートメントを終了することが必要になります。

C++ C++ プログラムの中で、*expression1* を使用すると、変数の初期化も、変数の宣言も行うことができます。この式で、またはステートメント の中のどこででも、変数を宣言すると、その変数は、**for** ループの終わりでスコープの外に出ます。

for ステートメントの例

次の **for** ステートメントは、count の値を 20 回印刷します。 **for** ステートメントは、count の値を 1 に初期設定します。ステートメントが反復されるたびに count が増やされます。

```
int count;
for (count = 1; count <= 20; count++)
   printf("count = %d¥n", count);
```

次の一連のステートメントも同じタスクを実行します。**for** ステートメントの代わりに **while** ステートメントを使用していることに注意してください。

```
int count = 1;
while (count <= 20)
{
   printf("count = %d¥n", count);
   count++;
}
```

以下の **for** ステートメントには、初期化の式が含まれていません。

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d¥n", index,
   list[index]);
}
```

以下の **for** ステートメントは、 scanf が e という文字を受け取るまで実行し続けます。

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '¥n')
            continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c¥n", letter);
}
```

以下の **for** ステートメントには、複数の初期化と増分が含まれています。コンマ演算子によってこの構造が可能となります。 **for** 式内の最初のコンマは、宣言の区切り子です。これは、i と j の 2 つの整数を

宣言して、初期化します。 2 番目のコンマ (コンマ演算子) によって、ループ内の各ステップを通るたびに、i と j を両方とも増加することが可能になります。

```
for (int i = 0,
j = 50; i < 10; ++i, j += 50)
{
   cout << "i = " << i << "and j = " << j
   << endl;
}
```

以下の例は、ネストされた **for** ステートメントを示しています。このステートメントは、[5][3] という次元の配列の値を印刷します。

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d¥n",
      table[row][column]);
```

row の値が 5 より小さい間、外部ステートメントが処理されます。外部の **for** ステートメントが実行されるたびに、内部の **for** ステートメントが column の初期値をゼロに設定します。そして、内部の **for** ステートメントのステートメントが 3 回実行されます。 column の値が 3 より小さい間は、内部ステートメントが実行されます。

break ステートメント

break ステートメント を使用すると、*反復* (**do**、**for**、**while**) ステートメントまたは **switch** 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 ステートメントの形式は、次のとおりです。

反復するステートメントにおいては、**break** ステートメントはループを終了して、ループの外側にある次のステートメントに制御を移します。ネストされたステートメントの中では、**break** ステートメントは、囲んでいる最小の **do**、**for**、**switch**、または **while** ステートメントのみを終了します。

switch ステートメントにおいては、**break** は、制御を **switch** 本体から **switch** 本体の外側にある、次のステートメントに渡します。

continue ステートメント

continue ステートメント を使用すると、進行中のループの反復を終了することができます。プログラム制御は、**continue** ステートメントからループ本体の終わりに渡されます。

continue ステートメントの形式は、次のとおりです。

continue ステートメントは、反復するステートメントの本体の中にしか存在できません。

continue ステートメントは、反復する (**do**、**for**、または **while**) ステートメントのアクション部分の処理を終了します。そして、制御を、ステートメントのループ連結部分へ移します。例えば、反復するステートメ

ントが **for** ステートメントである場合、制御は、ステートメントの条件部分の 3 番目の式に移動します。次に、ステートメントの条件部分の 2 番目の式 (テスト) に移動します。

ネストされたステートメントの中では、**continue** ステートメントは、直接に continue ステートメントを囲んでいる **do**、**for**、または **while** ステートメントの現行の反復だけを終了します。

continue ステートメントの例

以下の例は、**for** ステートメントにおける **continue** ステートメントを示しています。**continue** ステートメントを使用すると、値が 1 以下の配列 rates のエレメントに対する処理がスキップされます。

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00¥n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
            continue;
      printf("rate = %.2f¥n", rates[i]);
   }

   return(0);
}
```

プログラムは、以下の出力を作成します。

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

以下の例は、ネストされたループにおける **continue** ステートメントを示しています。内部ループが配列 strings の中である数に遭遇すると、そのループの反復は終了します。処理は、内部ループの 3 番目の式から続けられます。内部ループは、'¥0' エスケープ・シーケンスを検出した時点で終了します。

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '¥0';
      ++pointer)
```

```
      {                                 /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d¥n", letter_count);

   return(0);
}
```

プログラムは、以下の出力を作成します。

```
letter count = 5
```

return ステートメント

return ステートメント は、現行の関数の処理を終了して、関数の呼び出し元に制御を戻します。

return ステートメントの形式は、次の 2 つのうちいずれかです。

値を戻す関数には、**return** ステートメントに式 が含まれている必要があります。戻りの型が void である関数は、その return ステートメントに式 を含めることはできません。

戻りの型 void の関数の場合、return ステートメントは必ず必要というわけではありません。 **return** ステートメントを検出することなく関数の終わりに達すると、あたかも式のない **return** ステートメントが検出されたかのように、制御は呼び出し元に渡されます。つまり、最終ステートメントの完了時に暗黙的な戻りが実行され、制御は自動的に呼び出し元関数に戻ります。 1 つの関数に、複数の **return** ステートメントを含めることができます。 次に例を示します。

```
void copy( int *a, int *b, int c)
{
   /* Copy array a into b, assuming both arrays are the same size */

   if (!a || !b)       /* if either pointer is 0, return */
      return;

   if (a == b)         /* if both parameters refer */
      return;          /*    to same array, return */

   if (c == 0)         /* nothing to copy */
      return;

   for (int i = 0; i < c; ++i;) /* do the copying */
      b[i] = a[1];
                       /* implicit return */
}
```

この例では、**return** ステートメントは、**break** ステートメントのように、関数を途中で終了するために使用されています。

return ステートメントに現れる式は、このステートメントが現れる関数の戻りの型に変換されます。暗黙の変換ができない場合、**return** ステートメントは無効になります。

戻り式の値および関数値

return ステートメントに式がある場合、その式の値を呼び出し元に戻します。式のデータ型が関数の戻りの型と異なる場合には、式の値が、あたかも、関数の戻りの型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。

戻りの型が `void` の関数の **return** ステートメントの値は、この関数が値を戻さないことを意味しています。非 `void` の戻りの型を宣言されている関数内の **return** ステートメントに式が指定されていない場合は、コンパイラーによってエラー・メッセージが出力されます。

void 型を戻すものとして関数が宣言されている場合、式を持つ **return** ステートメントを使用することはできません。

return ステートメントの例

```
return;            /* Returns no value            */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1         */
return (x * x);    /* Returns the value of x * x  */
```

以下の関数は、整数の配列を検索して、変数 `number` と一致するものが存在するかどうか判別します。一致するものが存在すると、関数 `match` は `i` の値を戻します。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);
}
```

---

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 ステートメントの形式は、次のとおりです。

goto ステートメントは、通常の処理シーケンスを妨げる可能性があるため、これを使用すると、プログラムの読み取りおよび保守が難しくなります。多くの場合、**break** ステートメント、**continue** ステートメント、または関数呼び出しを使えば、**goto** ステートメントを使用しなくてもすみます。

goto ステートメントを使用してアクティブ・ブロックを終了する場合、そのブロックから制御が移動するときに、すべてのローカル変数は破棄されます。

goto ステートメントを使用して、初期化を飛び越えることはできません。

goto ステートメントの例

以下の例は、ネストされたループからジャンプするために使用される **goto** 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");
}
```

ヌル・ステートメント

ヌル・ステートメント はオペレーションを実行しません。形式は次のとおりです。

ヌル・ステートメントは、ラベル付きステートメントのラベルを保持したり、あるいは反復するステートメントの構文を完了したりすることができます。

ヌル・ステートメントの例

以下の例は、配列 price のエレメントを初期化します。初期化は **for** 式の中で起きるため、**for** 構文を終了するのに必要なものはステートメントのみで、オペレーションは必要ありません。

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

ブロック・ステートメントの終わりの前にラベルを必要とするときに、ヌル・ステートメントを使用できます。次に例を示します。

```
void func(void) {
  if (error_detected)
    goto depart;
  /* further processing */
  depart: ;  /* null statement required */
}
```

第 9 章 プリプロセッサー・ディレクティブ

プリプロセッサー・ディレクティブ は、ソース・プログラムのテキスト行と区別する # 文字で始まり、改行文字で終わるソース・ファイル内の行です。プリプロセッサー・ディレクティブの効力は、他の変換を行う前のソース・コードのテキストの変更であり、結果はディレクティブを含まない新しいソース・コードになります。プリプロセスされたソース・コードは、コンパイラーへの入力になるので、有効な C または C++ プログラムでなければなりません。

プリプロセッサー・ディレクティブの構文は、それ以外の言語の構文からは独立してはいますが類似しています。また、プリプロセス・フェーズの字句規則はコンパイラーの字句規則とは異なっています。標準 C および C++ トークン、およびそれ以外の、ファイル名、空白の有無、および行の終わりの位置を示すマーカーとして認識するための文字がプリプロセスされます。

本節では、プリプロセッサー・ディレクティブ、およびそれに関連するマクロ展開について説明します。プリプロセッサー・ディレクティブの概要に続いて、テキスト・マクロ、ファイルのインクルード、ISO 規格および事前定義されたマクロ名、条件付きコンパイル・ディレクティブ、およびプラグマを含むトピックについて説明します。

プリプロセッサーの概要

プリプロセス は、C および C++ ファイルがコンパイラーに渡される前に、それらのファイルに対して行われる予備操作です。 これによって、以下のことを実行できます。

- 現在のファイル内のトークンを指定された置換トークンと置き換える。
- 現在のファイル内にファイルを組み込む。
- 現在のファイルのセクションを条件によりコンパイルする。
- 診断メッセージを生成する。
- ソースの次の行の行番号を変更し、現在のファイルのファイル名を変更する。
- マシン特有の規則を、コードの指定されたセクションに適用する。

トークン は、空白で区切られた一連の文字です。プリプロセッサー・ディレクティブで認められている空白は、スペース、水平タブ、垂直タブ、改ページ、およびコメントだけです。改行文字も、プリプロセッサー・トークンを分離することができます。

プリプロセスされるソース・プログラム・ファイルは、有効な C または C++ プログラムでなければなりません。

プリプロセッサーは、以下のディレクティブによって制御されます。

#define	マクロを定義します。
#undef	プリプロセッサー・マクロ定義を除去します。
#error	コンパイル時エラー・メッセージ用のテキストを定義します。
#include	別のソース・ファイルからテキストを挿入します。
#if	定数式の結果に基づいて、ソース・コードの部分を条件により抑止します。
#ifdef	マクロ名が定義されている場合に、ソース・テキストを条件によりインクルードします。
#ifndef	マクロ名が定義されない場合に、ソース・テキストを条件によりインクルードします。
#else	直前の **#if**、**#ifdef**、**#ifndef**、または **#elif** テストが失敗した場合に、条件によりソース・テキストをインクルードします。

#elif	直前の **#if**、**#ifdef**、**#ifndef**、または **#elif** テストが失敗した場合に、定数式の値を基にして、条件によりソース・テキストをインクルードします。
#endif	条件テキストを終了します。
#line	コンパイラー・メッセージの行番号を提供します。
#pragma	コンパイラーに対してインプリメンテーション定義の命令を指定します。

プリプロセッサー・ディレクティブの形式

プリプロセッサー・ディレクティブは、# トークンで始まり、その後にプリプロセッサー・キーワードが続きます。# トークンは、空白でない行の先頭文字として存在しなければなりません。# はディレクティブ名の一部ではなく、空白で名前から分離することができます。

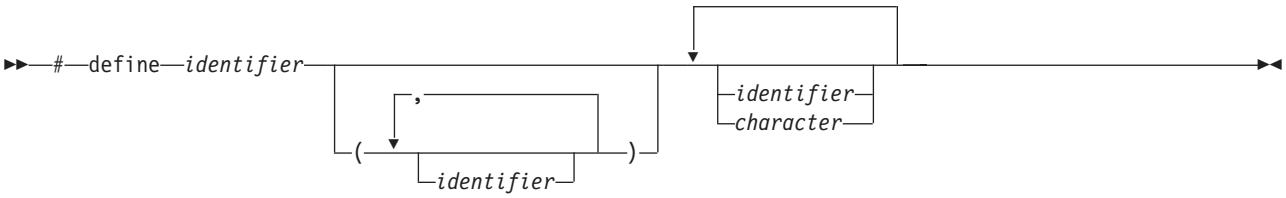
行の最後の文字が ¥ (円記号) 文字でない限り、プリプロセッサー・ディレクティブは改行文字で終了します。¥ 文字がプリプロセッサー行の最後の文字として現れると、プリプロセッサーは ¥ と改行文字を継続マーク文字として解釈します。プリプロセッサーは、¥ (およびそれに続く改行文字) を削除して、物理ソース行を継続する論理行に継ぎます。円記号と、行末文字またはレコードの物理的な終わりとの間に、空白文字があっても構いません。ただし、通常、この空白文字は編集の際には見えません。

一部の **#pragma** ディレクティブを除いて、プリプロセッサー・ディレクティブはプログラム内の任意の場所に入れることができます。

マクロの定義および展開 (#define)

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。

プリプロセッサーの **#define** ディレクティブの形式は、次のとおりです。



#define ディレクティブには、オブジェクト類似の定義または関数類似の定義を含めることができます。

#define と const の比較

- **#define** ディレクティブは、数値、文字、あるいはストリング定数の名前の作成に使用できるのに対し、const オブジェクトの場合は、任意の型を宣言することができます。
- const オブジェクトには変数に対するスコープ規則が適用されるのに対し、**#define** を使用して作成された定数には適用されません。
- const オブジェクトと違って、マクロはインラインで展開されるので、マクロの値は、コンパイラーが使用する中間ソース・コードには含まれません。インライン展開をすると、デバッガーはマクロ値を使用できなくなります。
- マクロは、バインドされた配列などの定数式の中で使用できますが、 const オブジェクトはできません。
- C++ コンパイラーは、マクロ引数を含めて、マクロの型検査は行いません。

関連参照
- 『オブジェクト類似マクロ』
- 『関数類似マクロ』
- 61 ページの『const 型修飾子』

オブジェクト類似マクロ

オブジェクト類似マクロ定義 は、単一の ID を指定された置換トークンに置き換えます。以下のオブジェクト類似の定義を使用すると、プリプロセッサーは、ID COUNT のこれ以降のすべてのインスタンスを、定数 1000 に置き換えます。

```
#define COUNT 1000
```

次のステートメント

```
int arry[COUNT];
```

が、この定義の後、かつ定義と同じファイル内に現れると、プリプロセッサーは、このステートメントをプリプロセッサーの出力で、以下のステートメントのように変更します。

```
int arry[1000];
```

他の定義が ID COUNT を参照することができます。

```
#define MAX_COUNT COUNT + 100
```

プリプロセッサーは、MAX_COUNT のこれ以降の出現を COUNT + 100 に置き換えます。これを、プリプロセッサーは、さらに 1000 + 100 に置き換えます。

マクロ展開によって部分的に構築された番号が作成された場合、プリプロセッサーは、その結果を単一の値であるとは見なしません。例えば、以下の結果は 10.2 という値にはならず、構文エラーになります。

```
#define a 10
a.2
```

マクロ展開によって部分的に構築される ID は、作成されない場合があります。したがって、以下の例は、2 つの ID を含んでいて、結果は構文エラーとなります。

```
#define d efg
abcd
```

関数類似マクロ

関数類似マクロは、オブジェクト類似マクロより複雑であって、その定義は、仮パラメーターの名前をコンマで区切って、括弧で囲んで宣言します。空の仮パラメーター・リストは有効です。そのようなマクロを使って、引数を取らない関数をシミュレートすることができます。

関数類似マクロの定義

小括弧に囲まれたパラメーター・リストおよび置換トークンが後ろに続く ID。パラメーターを置換コード内に組み込みます。空白文字で、ID (マクロの名前) とパラメーター・リストの左括弧とを分離することはできません。コンマで各パラメーターを区切る必要があります。

移植性のため、1 つのマクロには、パラメーターが 31 を超えないようにする必要があります。パラメーター・リストは、省略符号 (...) で終了することができます。 この場合、ID __VA_ARGS__ を置換リストに挿入することができます。

関数類似マクロの呼び出し

小括弧に入れられた、コンマで区切られた引数のリストが後に続く ID。引数の数は、定義内のパラメーター・リストが省略記号で終了していない限り、マクロ定義内のパラメーターの数に一致し

ている必要があります。定義内のパラメーター・リストが省略符号で終了する場合、マクロ呼び出しでの引数の数は、定義内のパラメーターの数を超えるはずです。 この過剰分の引数は*後続引数*と呼ばれます。 プリプロセッサーは、関数類似マクロの呼び出しを確認すると、引数の置換を行います。置換コード内のパラメーターは、対応する引数に置き換えられます。後続の引数がマクロ定義によって許可されている場合は、それらの引数は、その間にコンマを挟んでマージされ、それら後続引数があたかも 1 つの引数であるかのように、__VA_ARGS__ で置き換えられます。 引数自体に含まれるマクロの呼び出しはすべて、引数が置換コード内の対応するパラメーターと置き換わる前に、完全に置き換えられます。 この言語フィーチャーは、C++ の直交拡張です。

ID リストが省略符号で終了していない場合、マクロ呼び出しの中の引数の数は、対応するマクロ定義の中のパラメーターの数と同じでなければなりません。パラメーターの置換時、指定されたすべての引数 (区切り文字のコンマも含む) が置換された後に残っている引数は、変数引数と呼ばれる 1 つの引数にまとめられます。変数引数は、置換リストの中の ID __VA_ARGS__ のすべてのオカレンスを置き換えます。次の例は、このことを示しています。

```
#define debug(...)   fprintf(stderr, __VA_ARGS__)

debug("flag");     /*   Becomes fprintf(stderr, "flag");   */
```

マクロ呼び出し引数リストにおけるコンマは、以下の場合には、分離文字として作用しません。
- 文字定数内にある。
- ストリング・リテラル内にある。
- 小括弧で囲まれている。

以下の行は、a と b という 2 つのパラメーターと置換トークン (a + b) を持つものとしてマクロ SUM を定義します。

```
#define SUM(a,b) (a + b)
```

この定義により、プリプロセッサーは以下のステートメントを変更することになります (そのステートメントが前の定義の後に現れる場合)。

```
c = SUM(x,y);
c = d * SUM(x,y);
```

プリプロセッサーの出力においては、これらのステートメントは次のように表示されます。

```
c = (x + y);
c = d * (x + y);
```

置換テキストが正しく評価されるようにするためには、小括弧を使用してください。例えば、

```
#define SQR(c)  ((c) * (c))
```

上記の定義では、定義内の各パラメーター c の周りに小括弧を必要とします。以下のような表現も正しく評価することができます。

```
y = SQR(a + b);
```

プリプロセッサーはこのステートメントを次のように展開します。

```
y = ((a + b) * (a + b));
```

定義内に小括弧がないと、プリプロセッサーは評価の正しい順序を保てず、プリプロセッサーの出力は次のようになります。

```
y = (a + b * a + b);
```

および ## 演算子の引数は、関数類似マクロのパラメーターの置換の*前に* 変換されます。

プリプロセッサー ID は、いったん定義されると定義されたままとなり、言語のスコープ決定規則とは関係なく、有効となります。マクロ定義のスコープは定義から始まり、対応する **#undef** ディレクティブに遭遇するまで終了しません。対応する **#undef** ディレクティブがない場合、そのマクロ定義のスコープは、変換単位の終わりまで続きます。

再帰マクロは、完全には展開されません。例えば、以下の定義

```
#define x(a,b) x(a+1,b+1) + 4
```

は、

```
x(20,10)
```

を、以下のように展開します。

```
x(20+1,10+1) + 4
```

マクロ x を、それ自体の中で繰り返し展開しようとするよりも、上述の展開を行います。マクロ x が展開された後で、そのマクロは、関数 x() の呼び出しとなります。

置換トークンを指定するのに、定義は必須ではありません。以下の定義は、現在のファイル内のこれ以降の行から、トークン debug のすべてのインスタンスを除去します。

```
#define debug
```

2 番目のプリプロセッサー **#define** ディレクティブを用いて、定義済みの ID またはマクロの定義を変更することができます。ただし、2 番目のプリプロセッサー **#define** ディレクティブの前に、プリプロセッサー **#undef** ディレクティブがある場合に限ります。 **#undef** ディレクティブは、最初の定義を無効にして、同じ ID を再定義で使用できるようにします。

プログラムのテキストの中については、プリプロセッサーはマクロ呼び出しのための文字定数またはストリング定数のスキャンを行いません。

#define ディレクティブの例

以下のプログラムには、2 つのマクロ定義と、その定義されている両方のマクロを参照するマクロ起動が含まれています。

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d¥n", a); ¥
  printf("value 2 = %d¥n", b) ;

int main(void)
{
  int x = 2;
  int y = 3;

     PRNT(SQR(x),y);

   return(0);
}
```

プリプロセッサーによって解釈された後、このプログラムは、以下のものに等価のコードによって置き換えられます。

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

     printf("value 1 = %d¥n", ( (x) * (x) ) );
     printf("value 2 = %d¥n", y);

   return(0);
}
```

このプログラムの出力は次のようになります。

```
value 1 = 4
value 2 = 3
```

マクロ名のスコープ (#undef)

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。

プリプロセッサーの **#undef** ディレクティブの形式は、次のとおりです。

identifier が現在マクロとして定義されていなければ、**#undef** は無視されます。

#undef の例

以下のディレクティブは BUFFER および SQR を定義します。

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

以下のディレクティブはその定義を無効にします。

```
#undef BUFFER
#undef SQR
```

これらの **#undef** ディレクティブの後に続いて ID BUFFER および SQR が現れても、それらは、いかなる置換トークンにも置き換えられません。 **#undef** ディレクティブによってマクロの定義が除去されてしまえば、新しい **#define** ディレクティブでその ID を使用することができます。

演算子

(単一番号記号) 演算子は、関数類似マクロのパラメーターを文字ストリング・リテラルに変換します。例えば、以下のディレクティブを使用してマクロ ABC が定義される場合、

```
   #define ABC(x)   #x
```

これ以降のマクロ ABC の呼び出しはすべて、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"

呼び出し	マクロ展開の結果
ABC(Hello there)	"Hello there"

演算子を、ヌル・ディレクティブと混同してはなりません。

演算子は、下記の規則に従って、関数類似マクロ定義で使用してください。
- 関数類似マクロの中の # 演算子に続くパラメーターは、マクロに渡された引数を含む文字ストリング・リテラルに変換されます。
- プリプロセッサーは、マクロに渡された引数の前または後ろにある空白文字を削除します。
- マクロに渡された引数内に組み込まれた複数の空白文字は、単一のスペース文字に置き換えられます。
- マクロに渡された引数にストリング・リテラルがある場合、およびそのリテラル内に ¥ (円記号) 文字がある場合には、マクロ展開時に、元の ¥ の前に追加の ¥ 文字が挿入されます。
- マクロに渡された引数に " (二重引用符) 文字がある場合、マクロ展開時に、" の前に ¥ 文字が挿入されます。
- 引数のストリング・リテラルへの変換は、その引数でマクロが展開される前に行われます。
- マクロ定義の置換リスト内に複数の ## 演算子または # 演算子がある場合、その演算子の評価の順序は定義されていません。
- マクロ展開の結果が有効な文字ストリング・リテラルでない場合、その振る舞いは予期できません。

 # 演算子の例

以下の例は、# 演算子の使用法を示したものです。

```
#define STR(x)        #x
#define XSTR(x)       STR(x)
#define ONE           1
```

呼び出し	マクロ展開の結果
STR(¥n "¥n" '¥n')	"¥n ¥"¥¥n¥" '¥¥n'"
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"¥"hello¥""

演算子とのマクロ連結

(二重番号記号) 演算子は、マクロ定義に含まれるマクロの呼び出し (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。

演算子

- マクロ定義の置換リスト内に複数の ## 演算子、または # 演算子、またはその両方がある場合、その演算子の評価の順序は定義されていません。

演算子の例

以下の例は、## 演算子の使用法を示したものです。

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

呼び出し	マクロ展開の結果
ArgArg(lady, bug)	"ladybug"
ArgText(con)	"conTEXT"
TextArg(book)	"TEXTbook"
TextText	"TEXTtext"
ArgArg(Jitter, bug)	3

プリプロセッサーの Error ディレクティブ (#error)

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 ディレクティブの形式は、次のとおりです。

#error ディレクティブは、コンパイル時の安全チェックとして、**#if-#elif-#else** 構造の **#else** 部でよく使用されます。例えば、**#error** ディレクティブをソース・ファイルで使用すると、プログラムのバイパスすべき部分に到達したら、コードが生成されないようにすることできます。

例えば、以下のディレクティブ

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

は、次のエラー・メッセージを生成します。

```
BUFFER_SIZE is too small.
```

ファイルのインクルード (#include)

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。

プリプロセッサーの **#include** ディレクティブの形式は、次のとおりです。

C および C++ のすべてのインプリメンテーションにおいて、プリプロセッサーは **#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
```

例えば次のように、ファイル名が二重引用符で囲まれている場合、

```
#include "payroll.h"
```

プリプロセッサーは、これをユーザー定義のファイルとして扱い、プリプロセッサーが定義した方法でファイルを検索します。

例えば次のように、ファイル名が不等号括弧で囲まれている場合、

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

プリプロセッサーは、これをシステム定義のファイルとして扱い、プリプロセッサーが定義した方法でファイルを検索します。

改行および > の文字は、< と > で区切られたファイル名の中に入れることができません。改行および " (二重引用符) の文字は、 " と " で区切られたファイル名の中に入れることはできません。 > は入れることができます。

いくつかのファイルによって使用される宣言を 1 つのファイルの中に入れ、**#include** を用いて、それらを使用する各ファイルに含めることができます。例えば、以下のファイル defs.h には、いくつかの定義と、宣言の追加ファイルのインクルードが 1 つ入っています。

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

以下のディレクティブを用いて、defs.h 内にある定義を組み込むことができます。

```
#include "defs.h"
```

以下の例では、**#define** は、いくつかのプリプロセッサー・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
```

ISO 規格事前定義マクロの名前

C および C++ は両方とも、ISO C 言語規格で指定されている次の事前定義マクロの名前を提供しています。 __FILE__ および __LINE__ の場合以外は、事前定義マクロの値が変換単位全体を通して変化することがありません。

マクロ名 **説明**

__DATE__ ソース・ファイルがコンパイルされた日付が入っている文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__DATE__ の値が変わります。日付は次の形式です。

```
"Mmm dd yyyy"
```

各値は、次のとおりです。

Mmm 月を省略形式 (Jan、Feb、Mar、Apr、May 、Jun、Jul、Aug、Sep、Oct、Nov、または Dec) で表します。

dd 日を表します。日が 10 より小さい場合、最初の d はブランク文字になります。

yyyy 年を表します。

__FILE__ ソース・ファイルの名前が入った文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__FILE__ の値が変わります。これは、**#line** ディレクティブを使用して設定できます。

__LINE__ 現行のソース行番号を表す整数

コンパイラーがソース・プログラムの後続の行を処理すると、コンパイル中に __LINE__ の値が変わります。これは、**#line** ディレクティブを使用して設定できます。

__STDC__ C の場合、整数 1 であると、C コンパイラーが ISO 規格をサポートすることを示します。言語レベルを ANSI 以外にセットすると、このマクロは定義されません。(マクロが未定義の場合、#if ステートメントで使用されると、あたかもマクロが整数値 0 を持っているかのように振る舞います。)

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_VERSION__

long int 型の整数定数: C89 言語レベルでは 199409L。

__TIME__ ソース・ファイルがコンパイルされた時刻が入っている文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__TIME__ の値が変わります。時刻は次の形式です。

"hh:mm:ss"

各値は、次のとおりです。
hh　時間を表します。
mm　分を表します。
ss　秒を表します。

__cplusplus　C++ プログラムの場合、このマクロは、長整数リテラル 199711L に展開し、コンパイラーが C++ コンパイラーであることを示します。 C プログラムの場合、このマクロは定義されていません。このマクロ名には、末尾に下線がないことに注意してください。

言語標準で必要とされる事前定義マクロのほか、事前定義マクロ __IBMC__ は、C コンパイラーのレベルを示しています。事前定義マクロ __IBMCPP__ は、C++ コンパイラーのレベルを示しています。

関連参照

- 189 ページの『行制御 (#line)』
- 177 ページの『オブジェクト類似マクロ』

条件付きコンパイル・ディレクティブ

プリプロセッサーの条件付きコンパイル・ディレクティブ を使用すると、プリプロセッサーは、ソース・コードのコンパイルの一部を条件付きで抑止します。これらのディレクティブは、定数式または ID をテストして、プリプロセッサーがコンパイラーに渡すべきトークン、およびプリプロセス時にう回すべきトークンを判別します。この条件付きディレクティブには、次のものがあります。
- **#if**
- **#ifdef**
- **#else**
- **#ifndef**
- **#elif**
- **#endif**

プリプロセッサーの条件付きコンパイル・ディレクティブは、下記のいくつかの行に及びます。
- 条件指定行 (**#if**、**#ifdef**、または **#ifndef** で始まる)
- 条件の評価が非ゼロ値になった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- **#elif** 行 (オプション)
- 条件の評価が非ゼロ値になった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- **#else** 行 (オプション)
- 条件の評価がゼロになった場合にプリプロセッサーがコンパイラーに渡すコードが入っている行 (オプション)
- プリプロセッサーの **#endif** ディレクティブ

#if、**#ifdef**、および **#ifndef** の各ディレクティブのそれぞれに対して、ゼロまたは複数の **#elif** ディレクティブ、ゼロまたは 1 つの **#else** ディレクティブ、および一致する 1 つの **#endif** ディレクティブがあります。一致するディレクティブは、すべて同じネスト・レベルにあるものと見なします。

条件付きコンパイル・ディレクティブをネストすることができます。以下のディレクティブでは、最初の **#else** は、**#if** ディレクティブに突き合わせられます。

```
#ifdef MACNAME
                /*  tokens added if MACNAME is defined */
#   if TEST <=10
                /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                /* tokens added if MACNAME is defined and TEST >  10 */
#   endif
#else
                /*  tokens added if MACNAME is not defined */
#endif
```

各ディレクティブは、その直後のブロックを制御します。ブロックは、ディレクティブの後の行から始まって、同じネスト・レベルにある次の条件付きコンパイル・ディレクティブで終了する、すべてのトークンで構成されます。

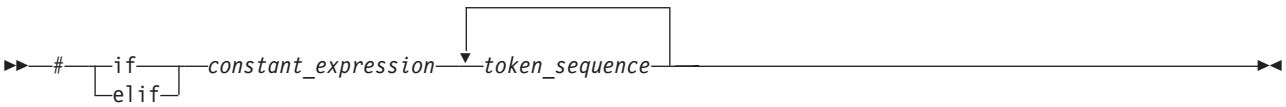
各ディレクティブは、検出された順序で処理されます。式の評価がゼロの場合、ディレクティブの後に続くブロックは無視されます。

プリプロセッサー・ディレクティブの後に続くブロックを無視することになっているとき、条件付きネスト・レベルが判別できるように、そのブロック内のプリプロセッサー・ディレクティブを識別するだけのために、トークンが検査されます。ディレクティブの名前以外のトークンは、すべて無視されます。

式が非ゼロとなる最初のブロックのみを処理します。そのネスト・レベルにある残りのブロックは無視します。そのネスト・レベルにあるブロックのどれも処理されていなくて、**#else** ディレクティブがある場合、**#else** ディレクティブに続くブロックが処理されます。そのネスト・レベルにあるブロックのどれも処理されていなくて、**#else** ディレクティブがない場合、ネスト・レベル全体が無視されます。

#if、#elif

#if および **#elif** ディレクティブは、 *constant_expression* の値をゼロと比較します。



定数式が非ゼロ値に評価される場合、条件の直後にあるコードの行をコンパイラーに渡します。

式がゼロに評価され、条件付きコンパイル・ディレクティブが、プリプロセッサー **#elif** ディレクティブを含んでいる場合、**#elif** および次の **#elif** またはプリプロセッサー **#else** ディレクティブとの間にあるソース・テキストが、プリプロセッサーによって選択され、コンパイラーに渡されます。 **#elif** ディレクティブをプリプロセッサーの **#else** ディレクティブの後ろに入れることはできません。

すべてのマクロが展開され、`defined()` の式はすべて処理され、残りのすべての ID は、トークン `0` に置き換えられます。

テストされる *constant_expression* は、以下の属性を持つ整定数式でなければなりません。

- キャストは実行されません。
- **long int** 値を使用して演算を実行します。
- 定義済みマクロを *constant_expression* に入れることはできます。それ以外の ID は式の中には入れられません。
- *constant_expression* には、単項演算子 **defined** を入れることができます。この演算子は、プリプロセッサー・キーワードの **#if** または **#elif** を用いた場合にのみ使用できます。以下の式の評価は、プリプロセッサーに *identifier (ID)* が定義されている場合は、`1` に、それ以外の場合は、`0` になります。

```
defined identifier
defined(identifier)
```

次に例を示します。

```
#if defined(TEST1) || defined(TEST2)
```

注: マクロが定義されていない場合、0 (ゼロ) の値がそれに代入されます。以下の例では、TEST がマクロ ID であることが必要です。

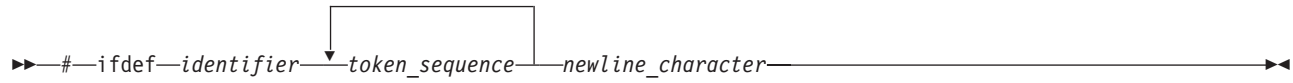
```
#if TEST >= 1
   printf("i = %d¥n", i);
   printf("array[i] = %d¥n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds ¥n");
#endif
```

#ifdef

#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。

プリプロセッサーの **#ifdef** ディレクティブの形式は、次のとおりです。



以下の例は、プリプロセッサーに対して 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。

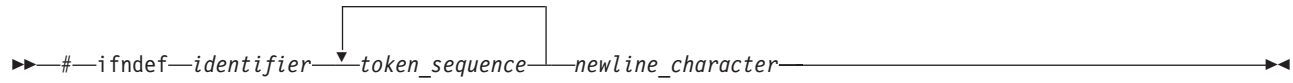
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef

#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。

プリプロセッサーの **#ifndef** ディレクティブの形式は、次のとおりです。



ID は、**#ifndef** キーワードの後に続いていなければなりません。以下の例は、プリプロセッサーに対して 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

#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。

プリプロセッサーの **#else** ディレクティブの形式は、次のとおりです。

#endif

プリプロセッサー・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。

形式は次のとおりです。

条件付きコンパイル・ディレクティブの例

以下の例は、プリプロセッサ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。

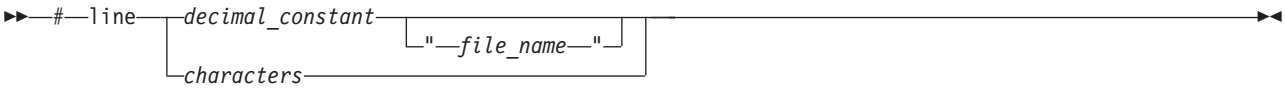
```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#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);
}
```

行制御 (#line)

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。

プリプロセッサーの **#line** ディレクティブの形式は、次のとおりです。



コンパイラーがプリプロセスされたソース内の行番号への参照をわかりやすく行えるようにするため、プリプロセッサーは必要な個所に (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 ディレクティブの例

#line 制御ディレクティブを使用して、コンパイラーにもっとわかりやすいエラー・メッセージを提供させることができます。以下のプログラムは、**#line** 制御ディレクティブを使用して、認識しやすい行番号を各関数に提供します。

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
```

```
{
   printf("Func_1 - the current line number is %d¥n",_ _LINE_ _);
}

#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d¥n",_ _LINE_ _);
}
```

このプログラムの出力は次のようになります。

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

ヌル・ディレクティブ (#)

ヌル・ディレクティブ ではアクションは行われません。このディレクティブは、ディレクティブ自体の行上の単一の # で構成されます。

ヌル・ディレクティブを、# 演算子や、プリプロセッサー・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
```

関連参照

- 180 ページの『# 演算子』

プラグマ・ディレクティブ (#pragma)

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。

ここで、*character_sequence* は、特定のコンパイラー・ディレクティブおよび引数 (あれば) を指定する一連の文字です。 プラグマ・ディレクティブは*改行* 文字で終了する必要があります。

プラグマの *character_sequence* は、マクロ置換を受けることがあります。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
```

1 つの **#pragma** ディレクティブで、複数のプラグマ構成を指定することができます。コンパイラーは、認識されないプラグマを無視します。

第 10 章 ネーム・スペース

C++ ネーム・スペース は、オプションとして命名されたスコープです。クラスまたは列挙に対してするように、ネーム・スペース内で名前を宣言します。ネストされたクラス名にアクセスするのと同じように、スコープ・レゾリューション (**::**) 演算子を使用することによって、ネーム・スペース内で宣言された名前にアクセスできます。ただし、ネーム・スペースは、クラスや列挙型が持つ追加のフィーチャーを持ちません。ネーム・スペースの主要な目的は、追加の ID (ネーム・スペースの名前) を名前に追加することです。

関連参照
- 87 ページの『C++ スコープ・レゾリューション演算子 ::』

ネーム・スペースの定義

C++ ネーム・スペースを一意的に識別するためには、**namespace** キーワードを使用します。

構文 - ネーム・スペース

オリジナルのネーム・スペース定義の中の *identifier* (ID) は、ネーム・スペースの名前です。オリジナルのネーム・スペース定義が現れる宣言領域では、ネーム・スペースを拡張するケースを除いて、ID は前に定義されていないことがあります。 ID が使用されない場合、そのネーム・スペースは、*名前なしネーム・スペース* です。

関連参照
- 194 ページの『名前なしネーム・スペース』

ネーム・スペースの宣言

C++ ネーム・スペース名に使用される ID は、固有でなければなりません。前にグローバル ID として使用されていてはなりません。

```
namespace Raymond {
  // namespace body here...
  }
```

この例では、Raymond は、ネーム・スペースの ID です。ネーム・スペースのエレメントにアクセスする意図がある場合、ネーム・スペースの ID は、すべての変換単位で既知である必要があります。

関連参照
- 3 ページの『グローバル・スコープ』

ネーム・スペース別名の作成

C++ 特定のネーム・スペース ID を参照するために、代替名を使用できます。

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  void f();
}

namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

この例では、IBM ID は INTERNATIONAL_BUSINESS_MACHINES の別名です。これは、長いネーム・スペース ID を参照するのに有用です。

ネーム・スペース名または別名が、同じ宣言領域内の他のエンティティーの名前として宣言されると、コンパイラー・エラーの結果を生じます。また、グローバル・スコープで定義されたネーム・スペース名が、プログラムのいずれかのグローバル・スコープ内の他のエンティティーの名前として宣言されると、コンパイラー・エラーの結果を生じます。

関連参照

- 3 ページの『グローバル・スコープ』

ネストされたネーム・スペースの別名の作成

C++ ネーム・スペース定義は、宣言を持っています。ネーム・スペース定義は宣言そのものであるので、ネーム・スペース定義をネストすることができます。

ネストされたネーム・スペースに、別名を適用することもできます。

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  int j;
    namespace NESTED_IBM_PRODUCT {
      void a() { j++; }
      int j;
      void b() { j++; }
  }
}
namespace NIBM = INTERNATIONAL_BUSINESS_MACHINES::NESTED_IBM_PRODUCT
```

この例では、NIBM ID は、ネーム・スペース NESTED_IBM_PRODUCT の別名です。このネーム・スペースは、INTERNATIONAL_BUSINESS_MACHINES ネーム・スペース内でネストされます。

ネーム・スペースの拡張

C++ ネーム・スペースは拡張可能です。前に定義されたネーム・スペースに、後続の宣言を追加できます。拡張部分は、オリジナルのネーム・スペース定義から分離されたファイル、またはオリジナルのネーム・スペース定義に付加されたファイルに現れます。次に例を示します。

```
namespace X { // namespace definition
  int a;
  int b;
  }

namespace X { // namespace extension
  int c;
  int d;
  }

namespace Y { // equivalent to namespace X
  int a;
```

```
  int b;
  int c;
  int d;
  }
```

この例では、namespace X は、a および b を使用して定義され、後で c および d を使用して拡張されます。その結果、 namespace X は 4 つのメンバーを含むようになっています。すべての必要なメンバーを 1 つのネーム・スペース内に宣言することもできます。この方式は namespace Y によって表されています。このネーム・スペースには a、b、c、および d が含まれています。

ネーム・スペースおよび多重定義

C++ 複数のネーム・スペースにまたがって関数を多重定義することができます。次に例を示します。

```
// Original X.h:
  f(int);

// Original Y.h:
  f(char);

// Original program.c:
  #include "X.h"
  #include "Y.h"

void z()
{
  f('a'); // calls f(char) from Y.h
}
```

ソース・コードを大幅に変更することなく、ネーム・スペースを上記の例に導入できます。

```
// New X.h:
namespace X {
  f(int);
  }

// New Y.h:
namespace Y {
  f(char);
  }

// New program.c:
  #include "X.h"
  #include "Y.h"

  using namespace X;
  using namespace Y;

void z()
{
  f('a'); // calls f() from Y.h
}
```

program.c で、関数 void z() は、ネーム・スペース Y のメンバーである関数 f() を呼び出します。using ディレクティブをヘッダー・ファイルに入れると、program.c のソース・コードは、変更されないままです。

関連参照

- 199 ページの『第 11 章 多重定義』

名前なしネーム・スペース

C++ 左中括弧の前に ID のないネーム・スペースは、*名前なし*ネーム・スペースになります。各変換単位は、それ自体の固有の名前なしネーム・スペースを含むことができます。次の例は、名前なしネーム・スペースがいかに有用であるかを示しています。

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   int variable;
   }

int main()
{
   cout << i << endl;
   variable = 100;
   return 0;
}
```

上記の例で、名前なしネーム・スペースは、スコープ・レゾリューション演算子を使用しないで `i` および `variable` にアクセスすることを許可しています。

名前なしネーム・スペースを、不適切に使用している例を以下に示します。

```
#include <iostream>

using namespace std;

namespace {
   const int i = 4;
   }

int i = 2;

int main()
{
   cout << i << endl; // error
   return 0;
}
```

コンパイラーは、グローバル名と、同じ名前を持つ名前なしネーム・スペースのメンバーとを区別できないので、`main` の内部では `i` はエラーの原因となります。 上記の例が作用するためには、ネーム・スペースは ID によって一意的に識別される必要があります。そして、`i` は、使用しているネーム・スペースを指定する必要があります。

名前なしネーム・スペースは、同じ変換単位内で拡張できます。次に例を示します。

```
#include <iostream>

using namespace std;

namespace {
   int variable;
   void funct (int);
   }

namespace {
   void funct (int i) { cout << i << endl; }
   }
```

```
int main()
{
   funct(variable);
   return 0;
}
```

funct のプロトタイプおよび定義の両方共、同じ名前なしネーム・スペースのメンバーです。

注: 名前なしネーム・スペースで定義された項目は、内部結合を持っています。キーワード **static** を使用して、内部結合を持つ項目を定義するよりも、代わりに名前なしネーム・スペースでそれらの項目を定義します。

関連参照
- 6 ページの『プログラム・リンケージ』
- 6 ページの『内部結合』

ネーム・スペース・メンバー定義

C++ ネーム・スペースは、それ自体の内部、または明示的修飾を使用して外部で、自身のメンバーを定義できます。以下に、ネーム・スペースが、内部的にメンバーを定義する例を示します。

```
namespace A {
  void b() { /* definition */ }
}
```

ネーム・スペース A 内で、メンバー void b() が内部的に定義されます。

ネーム・スペースは、定義されようとしている名前に明示的修飾を使用して、外部的にそのメンバーを定義することもできます。定義されようとしているエンティティーは、ネーム・スペース内ですでに宣言されている必要があります。定義は、宣言のネーム・スペースを囲むネーム・スペース内の、宣言のポイントの後に現れる必要があります。

ネーム・スペースが、外部的にメンバーを指定する例を以下に示します。

```
namespace A {
  namespace B {
    void f();
  }
  void B::f() { /* defined outside of B */ }
}
```

この例では、関数 f() は、ネーム・スペース B 内で宣言され、A 内で (B の外で) 定義されています。

ネーム・スペースおよびフレンド

C++ 最初にネーム・スペース内で宣言された名前はすべて、そのネーム・スペースのメンバーです。非ローカルのクラス内のフレンド宣言が、最初にクラスまたは関数を宣言する場合、そのフレンド・クラスまたは関数は、最内部の囲みネーム・スペースのメンバーです。

この構成の例を以下に示します。

```
// f has not yet been defined
void z(int);
namespace A {
  class X {
    friend void f(X);  // A::f is a friend
    };
```

```
  // A::f is not visible here
  X x;
  void f(X) { /* definition */}  // f() is defined and known to be a friend
}

using A::x;

void z()
{
  A::f(x);    // OK
  A::X::f(x); // error: f is not a member of A::X
}
```

この例では、関数 f() は、呼び出し A::f(s); を使用して、ネーム・スペース A によってのみ、呼び出すことができます。 A::X::f(x); 呼び出しを使用して class X によって、関数 f() を呼び出そうとする試みは、コンパイラー・エラーの結果になります。フレンド宣言は、最初に非ローカルのクラス内で行われるので、フレンド関数は、最内部の囲みネーム・スペースのメンバーです。このフレンド関数は、そのネーム・スペースによってのみアクセスすることができます。

関連参照
- 238 ページの『フレンド』

---

using ディレクティブ

C++ *using ディレクティブ* は、すべてのネーム・スペース修飾子およびスコープ演算子へのアクセスを提供します。これは、using キーワードをネーム・スペース ID に適用することによって行われます。

構文 - Using ディレクティブ

name は、前に定義されたネーム・スペースでなければなりません。 using ディレクティブは、グローバルおよびローカルのスコープで適用できますが、クラス・スコープでは適用できません。ローカル・スコープは、類似の宣言を隠蔽することによって、グローバル・スコープに優先します。

スコープが、2 番目のネーム・スペースを指名する using ディレクティブを含んでいる場合、そしてその 2 番目のネーム・スペースが別の using ディレクティブを含んでいる場合、 2 番目のネーム・スペースの using ディレクティブは、あたかも 1 番目のスコープ内に常駐しているかのように振る舞います。

```
namespace A {
  int i;
}
namespace B {
  int i;
  using namespace A;
}
void f()
{
  using namespace B;
  i = 7; // error
}
```

初期設定この例で、関数 f() 内で i を初期化しようとすると、コンパイラー・エラーを生じます。なぜなら、関数 f() は、どの i を呼び出すのか、つまりネーム・スペース A から i を呼び出すのか、またはネーム・スペース B から i を呼び出すのか、を知ることができないからです。

関連参照
- 252 ページの『using 宣言およびクラス・メンバー』

using 宣言およびネーム・スペース

C++ *using 宣言* は、特定のネーム・スペース・メンバーへのアクセスを提供します。これは、対応するネーム・スペース・メンバーを持っているネーム・スペース名に using キーワードを適用することによって行われます。

構文 - Using 宣言

この構文図で、修飾子名が using 宣言の後にきます。そして、*member* が修飾子名の後にきます。宣言が機能するには、メンバーは与えられたネーム・スペース内で宣言される必要があります。次に例を示します。

```
namespace A {
  int i;
  int k;
  void f;
  void g;
  }

using A::k
```

この例では、using 宣言の後にネーム・スペース A の名前である A がきます。次にその後にスコープ演算子 (::) および k がきます。この形式は、using 宣言によって、ネーム・スペース A の外で k にアクセスすることを可能にします。 using 宣言を出した後、その特定のネーム・スペースに対して行われたすべての拡張は、using 宣言が行われた時点では、認識されません。

特定の関数の多重定義されたバージョンは、その特定の関数の宣言の前に、ネーム・スペースに含める必要があります。 using 宣言は、ネーム・スペース、ブロックおよびクラス・スコープの中に置くことができます。

関連参照
- 252 ページの『using 宣言およびクラス・メンバー』

明示的アクセス

C++ ネーム・スペースのメンバーを明示的に修飾するには、ネーム・スペース ID を :: スコープ・レゾリューション演算子と一緒に使用します。

構文 - 明示的アクセス修飾

次に例を示します。

```
namespace VENDITTI {
  void j()
};

VENDITTI::j();
```

この例では、スコープ・レゾリューション演算子は、ネーム・スペース VENDITTI 内に保持されている関数 j へのアクセスを提供します。スコープ・レゾリューション演算子 :: は、グローバルおよびローカルの両方のネーム・スペース内の ID にアクセスするために使用されます。アプリケーションの中の ID はすべて、十分な修飾によってアクセスできます。明示的なアクセスは、名前なしネーム・スペースには適用できません。

関連参照

- 87 ページの『C++ スコープ・レゾリューション演算子 ::』

第 11 章 多重定義

C++ 関数名または演算子に対して同じスコープ内で複数の定義を指定すると、その関数名または演算子を*多重定義した* ことになります。

多重定義された宣言 は、同じスコープで前に宣言された宣言と同じ名前を使用して宣言された宣言です。ただし、両方の宣言は、異なる型を持っています。

多重定義された関数名または演算子を呼び出す場合、コンパイラーは、関数または演算子を呼び出すのに使用した引数型を、定義に指定されているパラメーター型と比較することによって、使用するのに最も適切な定義を判別します。最も適切な多重定義された関数または演算子を選択するプロセスは、*多重定義解決* と呼ばれています。

関数の多重定義

C++ 同じスコープ内で名前 f を持つ関数を複数個宣言すると、関数名 f を多重定義することになります。 f の宣言は、型または引数リスト中の引数の数、またはその両方で、互いに異なるものでなければなりません。 f という名前の多重定義された関数を呼び出すとき、関数呼び出しの引数リストを、名前 f を持つ多重定義された候補関数のそれぞれのパラメーター・リストと比較することによって、正しい関数が選択されます。*候補関数* とは、多重定義された関数名の呼び出しのコンテキストに基づいて呼び出すことのできる関数のことです。

関数 print (**int** を表示する) について考えてみましょう。次の例に示すように、関数 print を多重定義して、他の型 (例えば、**double** および **char***) を表示することができます。異なるデータ型に対して、それぞれ同様のオペレーションを実行する、同じ名前の関数を 3 つ持つことができます。

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}

void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
}
```

次に、上記の例の出力を示します。

```
 Here is int 10
 Here is float 10.1
 Here is char* ten
```

多重定義された関数の制約事項

C++ 以下の関数宣言は、同じスコープ内にあっても、それらを多重定義することはできません。このリストは、明示的に宣言された関数および **using** 宣言によって導入された関数にのみ適用されることに注意してください。

- 戻りの型が異なるだけの関数宣言。例えば、以下の宣言を宣言することはできません。

```
int f();
float f();
```

- 同じ名前および同じパラメーター型を持つメンバー関数宣言。ただし、これらの宣言のうちの 1 つは、静的メンバー関数宣言です。例えば、以下の f() の 2 つのメンバー関数宣言を宣言することはできません。

```
struct A {
  static int f();
  int f();
};
```

- 同じ名前、同じパラメーター型、および同じテンプレート・パラメーター・リストを持つメンバー関数テンプレート宣言。ただし、これらの宣言のうちの 1 つは、静的テンプレート・メンバー関数宣言です。
- 等価のパラメーター宣言を持つ関数宣言。これらの宣言は、同じ関数を宣言することになるので、許可されません。
- 同じ型を表す **typedef** 名の使用のみが異なるパラメーターを持つ関数宣言。 **typedef** は、別の型名に同義語で、独立した型を表すのではないことに注意してください。例えば、次の 2 つの f の宣言は、同じ関数の宣言です。

```
typedef int I;
void f(float, int);
void f(float I);
```

- 一方はポインター、もう一方は配列であることのみが異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

```
f(char*);
f(char[10]);
```

  パラメーターを差別化するときに、最初の配列次元が無意味で、他のすべての配列次元が有効であるもの。例えば、次の宣言は、同じ関数の宣言です。

```
g(char(*)[20]);
g(char[5][20]);
```

  次の 2 つの宣言は、等価ではありません。

```
g(char(*)[20]);
g(char(*)[40]);
```

- 一方は関数型、もう一方は同じ型の関数を指すポインターであることによってのみ異なるパラメーターを持つ関数宣言。例えば、次の宣言は、同じ関数の宣言です。

```
void f(int(float));
void f(int (*)(float));
```

- **const** 修飾子および **volatile** 修飾子ということのみが異なるパラメーターを持つ関数宣言。これは、これらの修飾子のどれかが、パラメーター型指定の最外部レベルに現れる場合にのみ適用されます。例えば、次の宣言は、同じ関数の宣言です。

```
int f(int);
int f(const int);
int f(volatile int);
```

  const 修飾子および **volatile** 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、次の宣言は、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)。

演算子の多重定義

C++ C++ のほとんどの組み込み演算子の関数を、再定義または多重定義することができます。これらの演算子は、グローバルに、またはクラス単位で多重定義できます。多重定義された演算子は、関数としてインプリメントされ、メンバー関数またはグローバル関数になることができます。

多重定義された演算子は、*演算子関数* と呼ばれます。演算子の前にキーワード **operator** を置いて、演算子関数を宣言します。多重定義された演算子は、多重定義された関数とは別個のものです。ただし、多重定義された関数と同様、演算子で使用されるオペランドの数と型によって区別されます。

標準の + (プラス) 演算子について考えます。この演算子が異なる標準型のオペランドで使用される場合は、演算子の意味が多少変わります。例えば、2 個の整数の加算は、2 個の浮動小数点数の加算と同様にはインプリメントされていません。 C++ では、標準の C++ 演算子をクラス型に適用するときに、それらにユーザー独自の意味を定義することができます。次の例では、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 + (プラス) 演算子は、2 つの複素数を加算するようにこのクラスで再定義されます。

```
// This example illustrates overloading the plus (+) operator.

#include <iostream>
using namespace std;

class 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、本節で説明する一般規則には従いません。

= 演算子を除く全演算子は継承されます。

単項演算子の多重定義

C++ 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、以下のとおりです。

```
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、!oz を許可しません。なぜなら、! 演算子は、クラス Z に対して定義されていないからです。)

関連参照

- 98 ページの『単項式』

増分と減分の多重定義

C++ 1 個のクラス型の引数かクラス型への参照を持つ非メンバー関数演算子を使用して、あるいは引数のないメンバー関数演算子を使用して、前置増分演算子 (++) を多重宣言します。

次の例では、増分演算子は以下に示す両方の方法で多重定義されます。

```
class X {
public:

  // member prefix ++x
  void operator++() { }
};

class Y { };

// non-member prefix ++y
void operator++(Y&) { }

int main() {
  X x;
```

```
  Y y;

  // calls x.operator++()
  ++x;

  // explicit call, like ++x
  x.operator++();

  // calls operator++(y)
  ++y;

  // explicit call, like ++y
  operator++(y);
}
```

2 つの引数 (1 番目はクラス型、2 番目は型 **int** を持っている) を持つ非メンバー関数演算子 `operator++()` を宣言することによって、あるクラス型に対して、後置増分演算子 (++) を多重定義することができます。あるいは、型 `int` の 1 個の引数を持つ、メンバー関数演算子 **operator++()** を宣言することができます。コンパイラーは **int** 引数を使用して、前置増分演算子と後置増分演算子を区別します。暗黙の呼び出しの場合、デフォルト値はゼロです。

次に例を示します。

```
class X {
public:

  // member postfix x++
  void operator++(int) { };
};

class Y { };

// nonmember postfix y++
void operator++(Y&, int) { };

int main() {
  X x;
  Y y;

  // calls x.operator++(0)
  // default argument of zero is supplied by compiler
  x++;
  // explicit call to member postfix x++
  x.operator++(0);

  // calls operator++(y, 0)
  y++;

  // explicit call to non-member postfix y++
  operator++(y, 0);
}
```

前置減分演算子と後置減分演算子は、対応する増分演算子と同じ規則に従います。

関連参照

- 99 ページの『増分 ++』
- 99 ページの『減分 --』

2 項演算子の多重定義

C++ パラメーターを 1 つ持っている非静的メンバー関数、またはパラメーターを 2 つ持っている非メンバー関数のいずれかを使用して、2 項演算子を多重定義します。 2 項演算子 @ は、ステートメント t @ u の形で呼び出されるものと想定します。ここで、t は、型 T のオブジェクト、u は、型 U のオブジェクトです。この演算子を多重定義する非静的メンバー関数は、以下の形式になっています。

```
return_type operator@(T)
```

同じ演算子を多重定義する非メンバー関数は、以下の形式になっています。

```
return_type operator@(T, U)
```

多重定義された 2 項演算子は、どのような型でも戻すことができます。

次の例では、* 演算子を多重定義します。

```
struct X {

  // member binary operator
  void operator*(int) { }
};

// non-member binary operator
void operator*(X, float) { }

int main() {
  X x;
  int y = 10;
  float z = 10;

  x * y;
  x * z;
}
```

呼び出し x * y は、x.operator*(y) と解釈されます。呼び出し x * z は、operator*(x, z) と解釈されます。

関連参照

- 109 ページの『2 項式』

代入の多重定義

C++ パラメーターを 1 つだけ持っている非静的メンバー関数を使用して、代入演算子 **operator=** を多重定義します。非メンバー関数である、多重定義された代入演算子を宣言することはできません。次の例では、特定のクラスについて、代入演算子を多重定義する方法を示します。

```
struct X {
  int data;
  X& operator=(X& a) { return a; }
  X& operator=(int a) {
    data = a;
    return *this;
  }
};

int main() {
  X x1, x2;
  x1 = x2;      // call x1.operator=(x2)
  x1 = 5;       // call x1.operator=(5)
}
```

代入 x1 = x2 は、コピー代入演算子 X& X::operator=(X&) を呼び出します。代入 x1 = 5 は、コピー代入演算子 X& X::operator=(int) を呼び出します。ユーザー自身があるクラスのコピー代入演算子を定義しない場合、コンパイラーがそれを暗黙的に宣言します。 したがって、派生クラスのコピー代入演算子 (**operator=**) が、その基底クラスのコピー代入演算子を隠蔽します。

ただし、コピー代入演算子はいずれも、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&)
```

代入 *ap1 = 'z' は、A& A::operator=(char) を呼び出します。この演算子は **virtual** と宣言されていないので、コンパイラーは、ポインター ap1 の型に基づいて関数を選択します。代入 *ap2 = b2 は、B& B::operator=(const &A) を呼び出します。この演算子は **virtual** と宣言されているので、コンパイラーは、ポインター ap1 が指すオブジェクトの型に基づいて関数を選択します。クラス C で宣言された、暗黙的に宣言されたコピー代入演算子は、 B& B::operator=(char) を隠蔽するので、コンパイラーは、代入 c1 = 'z' を許可しません。

関連参照

- 293 ページの『コピー代入演算子』
- 120 ページの『代入式』

関数呼び出しの多重定義

C++ 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。

関連参照

- 88 ページの『関数呼び出し演算子 ( )』

サブスクリプトの多重定義

C++ パラメーターを 1 つだけ持っている非静的メンバー関数を使用して、**operator[]** を多重定義します。次の例は、多重定義された添え字演算子を持っている単純配列クラスです。多重定義された添え字演算子は、ユーザーが指定された境界の外で配列にアクセスしようとすると、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、以下のとおりです。

```
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。

関連参照

- 90 ページの『配列サブスクリプト演算子 [ ]』

クラス・メンバー・アクセスの多重定義

C++ パラメーターを持っていない非静的メンバー関数を使用して、**operator->** を多重定義します。次の例は、コンパイラーがどのように、多重定義されたクラス・メンバー・アクセス演算子を解釈するかを示しています。

```
struct Y {
  void f() { };
};

struct X {
 Y* ptr;
 Y* operator->() {
   return ptr;
 };
};

int main() {
  X x;
  x->f();
}
```

ステートメント `x->f()` は、`(x.operator->())->f()` と解釈されます。

operator-> は、「スマート・ポインター」をインプリメントするために使用されます (しばしばポインター参照解除演算子と組にして)。これらのポインターは、普通のポインターのように振る舞うオブジェクトですが、次の点で異なります。すなわち、それらのポインターによってユーザーがオブジェクトにアクセスするときに、自動オブジェクト削除 (ポインターが破棄されるとき、または別のオブジェクトを指すためにポインターが使用されるとき)、あるいは参照カウント (同じオブジェクトを指すスマート・ポインターの数をカウントし、そして、そのカウントがゼロになったときに、オブジェクトを自動的に削除する) などの別の作業を実行します。

`auto_ptr` と呼ばれるスマート・ポインターの 1 つの例が、C++ 標準ライブラリーに含まれています。`<memory>` ヘッダーの中に、それを見出すことができます。 `auto_ptr` クラスは、自動オブジェクト削除をインプリメントします。

関連参照
- 91 ページの『矢印演算子 ->』

多重定義解決

C++ 最も適切な多重定義された関数または演算子を選択するプロセスは、*多重定義解決* と呼ばれています。

`f` が、多重定義された関数名であると想定します。多重定義された関数 `f()` を呼び出す場合、コンパイラーは*候補関数* のセットを作成します。この関数のセットには、`f()` を呼び出したポイントからアクセスできる、`f` という名前のすべての関数が含まれています。コンパイラーは、多重定義解決を促進するために、`f` という名前のこれらのアクセス可能な関数の中の 1 つの関数の代替表記を、候補関数として含めることができます。

候補関数のセットを作成した後、コンパイラーは、*実行可能関数* のセットを作成します。この関数のセットは、候補関数のサブセットです。各実行可能関数のパラメーター数は、`f()` を呼び出すのに使用した引数の数に一致します。

コンパイラーは、実行可能関数のセットから*最良の実行可能関数* を選択します。これは、f() を呼び出すときに C++ ランタイムが使用する関数宣言です。コンパイラーは、*暗黙的変換シーケンス* によって、これを行います。暗黙的変換シーケンスは、関数呼び出しの中の引数を、関数宣言の中の対応するパラメーターの型へ変換するのに必要な変換のシーケンスです。暗黙的変換シーケンスはランク付けされます。いくつかの暗黙的変換シーケンスは、他のものより良いシーケンスです。コンパイラーは、そのすべてのパラメーターが、他のすべての実行可能関数よりも良い、または等しいランク付けの暗黙的変換シーケンスを持つ、1 つの実行可能関数を検出しようと試みます。コンパイラーが検出する実行可能関数が、最良の実行可能関数です。コンパイラーは、コンパイラーが複数の最良実行可能関数を検出できたプログラムは、許可しません。

明示的キャストを使用することによって、正確な一致をオーバーライドすることができます。次の例では、f() に対する 2 回目の呼び出しが f(void*) と一致します。

```
void f(int) { };
void f(void*) { };

int main() {
   f(0xaabb);            // matches f(int);
   f((void*) 0xaabb);    // matches f(void*)
}
```

関連参照
- 『暗黙的変換シーケンス』

暗黙的変換シーケンス

C++ *暗黙的変換シーケンス*は、関数呼び出しの中の引数を、関数宣言の中の対応するパラメーターの型へ変換するのに必要な変換のシーケンスです。

コンパイラーは、各引数に対する暗黙的変換シーケンスを決定しようとします。コンパイラーは次に、各暗黙的変換シーケンスを 3 つのカテゴリーの中の 1 つにカテゴリー化し、カテゴリーに応じてそれらをランク付けします。コンパイラーは、引数に対する暗黙的変換シーケンスを検出できないようなプログラムは、許可しません。

以下に、変換シーケンスの 3 つのカテゴリーを、最良から最悪への順序で示します。
- 標準変換シーケンス
- ユーザー定義の変換シーケンス
- 省略符号変換シーケンス

注 : 2 つの標準変換シーケンスまたは 2 つのユーザー定義の変換シーケンスが、異なるランクを持つことがあります。

標準変換シーケンス

標準変換シーケンスは、3 つのランクの中の 1 つにカテゴリー化されます。ランクは、最良から最悪への順序でリストされています。
- 完全一致 : このランクには、以下の変換が含まれます。
  - 識別変換
  - 左辺値から右辺値への変換
  - 配列からポインターへの変換
  - 修飾変換
- 拡張: このランクには整数および浮動小数点拡張が含まれます。
- 変換 : このランクには、以下の変換が含まれます。

– 整数および浮動小数点変換
– 浮動 - 整数変換
– ポインター型変換
– ポインターからメンバーへの変換
– ブール変換

コンパイラーは、標準変換シーケンスを、その最悪ランクの標準変換によってランク付けします。例えば、標準変換シーケンスが浮動小数点変換を持っている場合、そのシーケンスは、変換ランクを持っていることになります。

ユーザー定義の変換シーケンス

ユーザー定義の変換シーケンス は、以下のもので構成されています。

- 標準変換シーケンス
- ユーザー定義の変換
- 2 番目の標準変換シーケンス

ユーザー定義の変換シーケンス A およびユーザー定義の変換シーケンス B の両者が、同じユーザー定義の変換関数またはコンストラクターを持っていて、A の 2 番目の標準変換シーケンスが、B の 2 番目の標準変換シーケンスよりも良ければ、前者の方が良いランクの変換シーケンスです。

省略符号変換シーケンス

省略符号変換シーケンス は、コンパイラーが関数呼び出しの中の引数を、対応する省略符号パラメーターに突き合わせるときに生じます。

関連参照

- 126 ページの『左辺値から右辺値への変換』
- 128 ページの『ポインター型変換』
- 130 ページの『修飾変換』
- 127 ページの『整数変換』
- 128 ページの『浮動小数点の型変換』
- 127 ページの『ブール変換』

多重定義された関数のアドレスの解決

C++ 引数が指定されていない多重定義された関数名 f を使用する場合、その名前は、関数、関数を指すポインター、メンバー関数を指すポインター、または関数テンプレートの特殊化を参照することができます。引数が指定されなかったので、コンパイラーは、関数呼び出しまたは演算子の使用に対して実行するのと同じ方法では、多重定義解決を実行することができません。その代わりに、コンパイラーは、f を使用した場所に応じて、以下の式のうちの 1 つの式の型に一致する、最良の実行可能関数を選択しようと試みます。

- 初期化しようとしているオブジェクトまたは参照
- 代入の左辺
- 関数またはユーザー定義の演算子のパラメーター
- 関数、演算子、または変換の戻り値
- 明示的型変換

ユーザーが f を使用したときに、コンパイラーが非メンバー関数または静的メンバー関数の宣言を選択した場合、コンパイラーは、その宣言を型「ポインターから関数へ」または「参照から関数へ」の式に突き合

わせました。コンパイラーが非静的メンバー関数の宣言を選択した場合、コンパイラーは、その宣言を型「ポインターからメンバー関数へ」の式に突き合わせました。次の例は、このことを示しています。

```
struct X {
  int f(int) { return 0; }
  static int f(char) { return 0; };
}

int main() {
  int (X::*a)(int) = &X::f;
//  int (*b)(int) = &X::f;
  int (*c)(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ます。 このセットにテンプレート関数だけがある場合、コンパイラーは、最も特殊化されたテンプレート関数を選択します。次の例は、このことを示しています。

```
template<class T> int f(T) { return 0; }
template<> int f(int) { return 0; }
int f(int) { return 0; }

int main() {
  int (*a)(int) = f;
  a(1);
}
```

関数呼び出し `a(1)` は、`int f(int)` を呼び出します。

関連参照

- 155 ページの『関数へのポインター』
- 228 ページの『メンバーへのポインター』
- 307 ページの『関数テンプレート』
- 318 ページの『明示的特殊化』

第 12 章 クラス

C++ クラス は、ユーザー定義のデータ型を作成するためのメカニズムの 1 つです。これは、C 言語の構造体のデータ型と似ています。 C では、構造体は、データ・メンバーのセットで構成されます。 C++ では、クラス型は C 構造体と似ていますが、クラスは、データ・メンバーのセット、およびそのクラスで実行できるオペレーションのセットで構成される点が異なります。

C++ において、クラスの型は **union**、**struct**、または **class** というキーワードを用いて宣言することができます。共用体オブジェクトは、名前付きメンバーのセットの 1 つを保持できます。構造体およびクラス・オブジェクトは、全メンバーのセットを保持します。各クラス型はそれぞれ、データ・メンバー、メンバー関数、および他の型名を含む、クラス・メンバーの固有のセットを表します。メンバーに対するデフォルトのアクセスは、クラス・キーによって決まります。

- クラス・キー **class** を用いて宣言されたクラスのメンバーは、デフォルトにより private になります。クラスは、デフォルトで private で継承されます。
- キーワード **struct** を用いて宣言されたクラスのメンバーは、デフォルトでは public になります。構造体は、デフォルトで public で継承されます。
- (キーワード **union** を用いて宣言された) 共用体のメンバーは、デフォルトにより public になります。共用体は、派生における基底クラスとして使用することはできません。

クラス型を作成すると、1 つまたは複数のそのクラス型のオブジェクトを宣言することができます。次に例を示します。

```
class X
{
     /* define class members here */
};
int main()
{
      X xobject1;       // create an object of class type X
      X xobject2;       // create another object of class type X
}
```

C++ には、*ポリモアフィック・クラス*があります。ポリモアフィズムにより、コンパイル時において関数が所属するクラスを正確に把握しなくても、別々のクラス (継承に関連した) に現れる関数名を使用することができます。

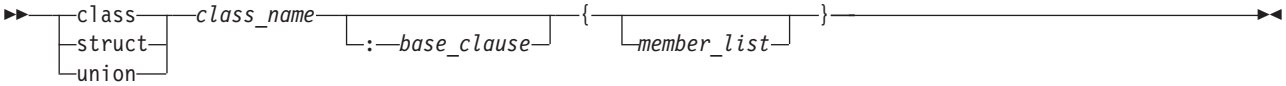
C++ では、多重定義の概念から、標準の演算子および関数を再定義することができます。演算子の多重定義により、組み込み (標準装備の) 型と同じように簡単にクラスを使用できるので、データ抽出が容易になります。

クラス・タイプの宣言

C++ クラス宣言は、固有の型のクラス名を作成します。

クラス指定子 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。クラス指定子の形式は、次のとおりです。

構文 - クラス指定子



class_name は、スコープ内で予約語となる固有の ID です。クラス名が宣言されると、そのクラス名は、囲みスコープ内にある同じ名前の他の宣言を隠蔽します。

member_list は、クラス *class_name* のクラス・メンバー (データと関数の両方) を指定します。クラスの *member_list* が空の場合、そのクラスのオブジェクトのサイズはゼロではありません。クラスのサイズが必要でなければ、クラス指定子自体の *member_list* 内で *class_name* を使用することができます。

これは、クラス *class_name* が継承するメンバーの元の *base_clause* (複数の場合もある) を指定します。*base_clause* が空でない場合、クラス *class_name* は、*派生クラス* と呼ばれます。

構造体 は、*class_key* **struct** を使用して宣言されたクラスです。構造体のメンバーおよび基底クラスは、デフォルトにより public になります。*共用体* は、 *class_key* **union** を使用して宣言されたクラスです。共用体のメンバーは、デフォルトにより public になります。共用体は、一時に 1 つのデータ・メンバーのみ保持します。

集合体クラス は、ユーザー定義のコンストラクター、private またはプロテクトされた非静的データ・メンバー、基底クラス、および仮想関数のないクラスです。

クラス・オブジェクトの使用

C++ クラス型を使用して、そのクラス型のインスタンスつまりオブジェクト を作成することができます。例えば、クラス名の X、Y、および Z を指定して、それぞれクラス、構造体、および共用体を宣言することができます。

```
class X {
  // members of class X
};

struct Y {
  // members of struct Y
};

union Z {
  // members of union Z
};
```

これで、各クラス型のオブジェクトを宣言することができます。クラス、構造体、および共用体は、すべて C++ クラス型であることを忘れないでください。

```
int main()
{
      X xobj;      // declare a class object of class type X
      Y yobj;      // declare a struct object of class type Y
      Z zobj;      // declare a union object of class type Z
}
```

C++ では、C とは異なり、クラスの名前が隠れていない限り、クラス・オブジェクトの宣言の前に、キーワード **union**、**struct**、および **class** を入れる必要はありません。次に例を示します。

```
struct Y { /* ... */ };
class X { /* ... */ };
int main ()
```

```
{
      int X;              // hides the class name X
      Y yobj;             // valid
      X xobj;             // error, class name X is hidden
      class X xobj;       // valid
}
```

1 つの宣言で複数のクラス・オブジェクトを宣言すると、宣言子は個々に宣言されたように扱われます。例えば、以下のように、クラス S の 2 つのオブジェクトを単一の宣言で宣言する場合、

```
class S { /* ... */ };
int main()
{
      S S,T; // declare two objects of class type S
}
```

この宣言は、以下と同等です。

```
class S { /* ... */ };
int main()
{
      S S;
      class S T;       // keyword class is required
                       // since variable S hides class type S
}
```

しかし、以下と同等ではありません。

```
class S { /* ... */ };
int main()
{
      S S;
      S T;             // error, S class type is hidden
}
```

また、クラスへの参照、クラスへのポインター、およびクラスの配列を宣言することもできます。次に例を示します。

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
int main()
{
      X xobj;
      X &xref = xobj;          // reference to class object of type X
      Y *yptr;                 // pointer to struct object of type Y
      Z zarray[10];            // array of 10 union objects of type Z
}
```

コピー制限のないクラス型のオブジェクトは、関数への引数として、代入または引き渡しを行うことができ、また関数により戻すことができます。

クラスと構造体

C++ C++ のクラスは、C 言語の構造体の拡張機能です。構造体とクラスの唯一の相違点は、デフォルトによるアクセスが、構造体メンバーは public アクセスで、クラス・メンバーは private アクセスであることです。したがって、キーワードの **class** または **struct** を使用して、同等のクラスを定義できます。

例えば、以下のコードにおいて、クラス X は、構造体 Y と同等です。

```
class X {

  // private by default
```

```
  int a;

public:

  // public member function
  int f() { return a = 5; };
};

struct Y {

  // public by default
  int f() { return a = 5; };

private:

  // private data member
  int a;
};
```

構造体を定義してから、キーワード **class** を使用して、その構造体のオブジェクトを宣言すると、デフォルトによりそのオブジェクトのメンバーは、public のままです。以下の例において、obj_X が、クラス・キー **class** を使用する詳述型指定子の使用を宣言していますが、main() は、obj_X のメンバーへのアクセスを行います。

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

struct X {
 int a;
 int b;
};

class X obj_X;

int main() {
  obj_X.a = 0;
  obj_X.b = 1;
  cout << "Here are a and b: " << obj_X.a << " " << obj_X.b << endl;
}
```

上記の例の出力は、以下のとおりです。

```
Here are a and b: 0 1
```

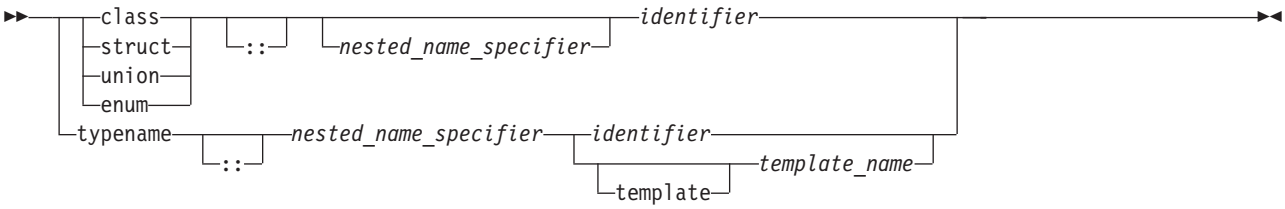
関連参照
- 46 ページの『構造体』

クラス名のスコープ

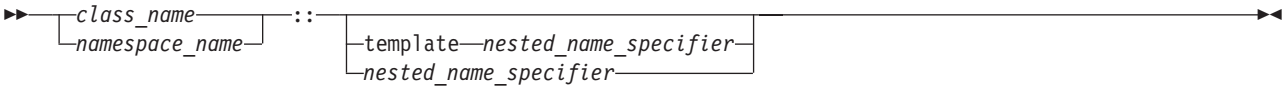
C++ クラス宣言は、クラスが宣言されたスコープ内にクラス名を導入します。囲みスコープ内にある、その名前のクラス、オブジェクト、関数、または他の宣言はすべて隠蔽されます。

クラス名が、同じ名前を持つ関数、列挙子、またはオブジェクトと同じスコープ内で宣言される場合は、*詳述型指定子* を使用してそのクラスを参照してください。

構文 - 詳述型指定子



構文 - ネストされた名前指定子



以下の例では、関数 A() の定義がクラス A を隠しているため、このクラスを参照するには、詳述型指定子を使用する必要があります。

```
class A { };

void A (class A*) { };

int main()
{
      class A* x;
      A(x);
}
```

宣言 class A* x が、詳述型指定子です。上記で示したような、別の関数、列挙子、またはオブジェクトと同じ名前でクラスを宣言することは、お勧めしません。

また、詳述型指定子をクラス型の不完全な宣言で使用して、現行スコープ内のクラス型のための名前を予約することもできます。

不完全なクラス宣言

C++ *不完全なクラス宣言* とは、クラス・メンバーを定義していないクラス宣言のことです。宣言が完全なものになるまでは、そのクラス型のオブジェクトを宣言したり、クラスのメンバーを参照することはできません。ただし、クラスのサイズが必要でなければ、不完全な宣言を使用して、クラスの定義の前にそのクラスに特定の参照を行うことはできます。

例えば、以下に示すように、構造体 second の定義で、構造体 first へのポインターを定義することができます。 second の定義の前に、不完全なクラス宣言で、構造体 first が宣言されています。構造体 second 内の oneptr の定義では、first のサイズは必要ありません。

```
struct first;           // incomplete declaration of struct first

struct second           // complete declaration of struct second
{
      first* oneptr;    // pointer to struct first refers to
                        // struct first prior to its complete
                        // declaration

      first one;        // error, you cannot declare an object of
                        // an incompletely declared class type
      int x, y;
};

struct first            // complete declaration of struct first
```

```
{
      second two;       // define an object of class type second
      int z;
};
```

しかし、空のメンバー・リストを指定してクラスを宣言すると、それは完全なクラス宣言となります。次に例を示します。

```
class X;                // incomplete class declaration
class Z {};             // empty member list
class Y
{
public:
      X yobj;           // error, cannot create an object of an
                        // incomplete class type
      Z zobj;           // valid
};
```

関連参照

- 223 ページの『クラス・メンバー・リスト』

ネスト・クラス

C++ ネスト・クラス は、別のクラスのスコープ内で宣言されるものです。ネスト・クラスの名前は、その囲みクラスに対してローカルです。ポインター、参照、またはオブジェクト名を明示的に使用しない限り、ネスト・クラスでの宣言で使用できるのは可視構成だけで、これには、囲みクラスからの型名、静的メンバー、および列挙子と、グローバル変数が含まれます。

ネスト・クラスのメンバー関数は、正規のアクセス規則に従い、囲みクラスのメンバーへの特別なアクセス権を持ちません。囲みクラスのメンバー関数は、ネスト・クラスのメンバーへの特別なアクセスは行いません。次の例は、このことを示しています。

```
class A {
  int x;

  class B { };

  class C {

    // The compiler cannot allow the following
    // declaration because A::B is private:
    //   B b;

    int y;
    void f(A* p, int i) {

    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;

    }
  };

  void g(C* p) {

    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
  }
};

int main() { }
```

コンパイラーは、クラス A::B が private なので、オブジェクト b の宣言を許可しません。コンパイラーは、A::x が private なので、ステートメント p->x = i を許可しません。コンパイラーは、C::y が private なので、ステートメント int i = p->y を許可しません。

ネーム・スペース・スコープで、ネスト・クラスのメンバー関数および静的データ・メンバーを定義することができます。例えば、以下のコードでは、修飾された型名を使用して、静的メンバー x と y およびネスト・クラス nested のメンバー関数 f() と g() にアクセスすることができます。修飾された型名を使用すると、**typedef** を定義して、修飾されたクラス名を表すことができます。その後、:: (スコープ・レゾリューション) 演算子を用いた **typedef** を使用して、ネスト・クラスまたはクラス・メンバーを参照することができます。

```
class outside
{
public:
      class nested
      {
public:
            static int x;
            static int y;
            int f();
            int g();
      };
};
int outside::nested::x = 5;
int outside::nested::f() { return 0; };

typedef outside::nested outnest;       // define a typedef
int outnest::y = 10;                   // use typedef with ::
int outnest::g() { return 0; };
```

しかし、ネスト・クラス名を示す typedef を使用すると、情報を隠蔽し、理解が困難なコードが作成されます。

詳述型指定子では、typedef 名を使用できません。例えば、上記の例で次の宣言は、使用できません。

```
  class outnest obj;
```

ネスト・クラスは、その囲みクラスの private メンバーから継承します。次の例は、このことを示しています。

```
class A {
  private:
  class B { };
  B *z;

  class C : private B {
  private:
      B y;
//      A::B y2;
      C *x;
//      A::C *x2;
    };
};
```

ネスト・クラス A::C は A::B から継承します。コンパイラーは、A::B も A::C も private なので、宣言 A::B y2 および A::C *x2 を許可しません。

ローカル・クラス

C++ *ローカル・クラス* は、関数定義の中で宣言されます。ローカル・クラスでの宣言が使用できるのは、外部変数および関数に加えて、囲みスコープからの型名、列挙、静的変数だけです。

次に例を示します。

```
int x;                          // global variable
void f()                        // function definition
{
      static int y;             // static variable y can be used by
                                // local class
      int x;                    // auto variable x cannot be used by
                                // local class
      extern int g();           // extern function g can be used by
                                // local class

      class local               // local class
      {
            int g() { return x; }      // error, local variable x
                                       // cannot be used by g
            int h() { return y; }      // valid,static variable y
            int k() { return ::x; }    // valid, global x
            int l() { return g(); }    // valid, extern function g
      };
}

int main()
{
      local* z;                 // error: the class local is not visible
      // ...}
```

ローカル・クラスのメンバー関数は、メンバー関数が定義される場合、そのクラス定義の中で定義してください。結果として、ローカル・クラスのメンバー関数は、インライン関数です。ローカル・クラスのスコープの中で定義されているメンバー関数は、すべてのメンバー関数と同様に、キーワード **inline** は必要ありません。

ローカル・クラスが静的データ・メンバーを持つことはできません。以下の例において、ローカル・クラスの静的メンバーを定義しようとするとエラーになります。

```
void f()
{
    class local
    {
       int f();              // error, local class has noninline
                             // member function
       int g() {return 0;}   // valid, inline member function
       static int a;         // error, static is not allowed for
                             // local class
       int b;                // valid, nonstatic variable
    };
}
//      . . .
```

囲み関数は、ローカル・クラスのメンバーに特別なアクセスを行いません。

関連参照
- 224 ページの『メンバー関数』
- 155 ページの『インライン関数』

ローカル型名

C++ ローカル型名は、他の名前と同じスコープ規則に従います。クラス宣言の中で定義される型名にはクラス・スコープがあり、修飾をしなければ、それらのクラスの外側で使用することはできません。

型名で使用されているクラス名、**typedef** 名、または定数名をクラス宣言で使用すると、その名前をクラス宣言で再定義することはできません。

次に例を示します。

```
int main ()
{
      typedef double db;
      struct st
      {
            db x;
            typedef int db; // error
            db y;
      };
}
```

次の宣言は有効です。

```
typedef float T;
class s {
      typedef int T;
      void f(const T);
};
```

ここで、関数 `f()` は型 `s::T` の引数を取ります。しかし、`s` のメンバーの順序が逆になっている以下の宣言は、エラーとなります。

```
typedef float T;
class s {
      void f(const T);
      typedef int T;
};
```

クラス宣言で一度その名前を使用したクラス名または **typedef** 名に対して、クラス名または **typedef** 名でない名前をクラス宣言で再定義することはできません。

関連参照

- 1 ページの『スコープ』
- 36 ページの『typedef』

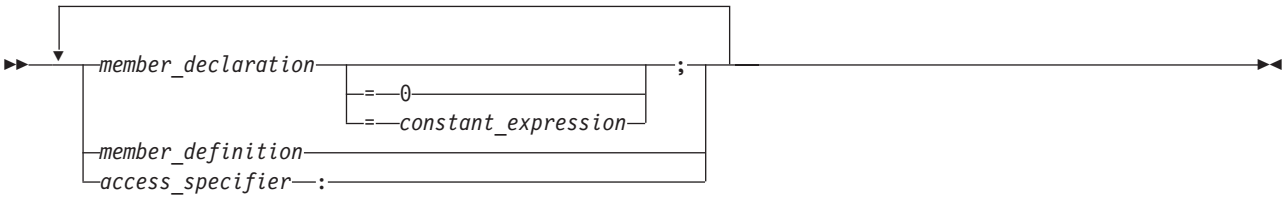
第 13 章 クラス・メンバーとフレンド

C++ 本節では、情報隠蔽メカニズムに関するクラス・メンバーの宣言について説明するとともに、クラスがフレンド・メカニズムを使用して、クラスの非 public メンバーへの関数およびクラス・アクセスをどのように認可するかについても述べます。 C++ では、情報隠蔽の概念を拡大して、private インプリメンテーションを除くパブリック・クラス・インターフェースを持つという概念を追加しています。これは、プログラム内の関数による、クラス型の内部表記への直接アクセスを制限するためのメカニズムです。

クラス・メンバー・リスト

C++ オプションのメンバー・リスト は、クラス・メンバー と呼ばれるサブオブジェクトを宣言します。クラス・メンバーとしては、データ、関数、ネストされた型、および列挙子が可能です。

構文 - クラス・メンバー・リスト



メンバー・リストは、クラス名の後に続き、中括弧の中に入れられます。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・メンバーの割り振り順は、インプリメンテーションに依存します。

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

データ・メンバー

C++ データ・メンバーには、基本型だけでなく、ポインター、参照、配列の型、ビット・フィールド、およびユーザー定義の型などの、他の型を用いて宣言されるメンバーが含まれます。データ・メンバーは、変数と同じ方法で宣言できますが、明示的初期化指定子をクラス定義の内部に設定することはできません。しかし、整数型または列挙型の const 静的データ・メンバーは、明示的初期化指定子を持つこともあります。

配列を非静的クラス・メンバーとして宣言する場合には、その配列のすべての次元を指定する必要があります。

クラスには、クラス型のメンバー、またはクラス型へのポインターまたは参照であるメンバーを入れることができます。クラス型のメンバーは、事前に宣言されているクラス型のメンバーでなければなりません。クラスのサイズが必要でなければ、メンバー宣言で、不完全なクラス型を使用することができます。例えば、不完全なクラス型へのポインターであるメンバーを宣言することができます。

クラス X には、型 X のメンバーを入れることはできません。ただし、X へのポインター、X への参照、および X の静的オブジェクトは入れることができます。 X のメンバー関数は、型 X の引数を取り、X 型を戻します。例えば、次のようになります。

```
class X
{
      X();
      X *xptr;
      X &xref;
      static X xcount;
      X xfunc(X);
};
```

メンバー関数

C++ *メンバー関数* とは、クラスのメンバーとして宣言される演算子および関数のことです。メンバー関数には、**friend** 指定子を用いて宣言される演算子および関数は含まれません。これらは、クラスの*フレンド* と呼ばれます。メンバー関数を **static** として宣言できます。これは、*静的メンバー関数* と呼ばれます。 **static** として宣言されていないメンバー関数は、*非静的メンバー関数* と呼ばれます。

クラス A の x という名前のオブジェクトを作成し、クラス A には、非静的メンバー関数 f() があるとします。関数 x.f() を呼び出す場合、f() の本体にあるキーワード **this** は、x のアドレスとなります。

メンバー関数の定義は、それを囲むクラスのスコープ内にあります。メンバー関数の定義が、クラス・メンバー・リストのそのメンバーの宣言の前にある場合であっても、メンバー関数の本体は、クラス宣言の後で分析され、そのクラスのメンバーを、メンバー関数の本体で使用できるようにします。以下の例では、関数 add() が呼び出されるとき、データ変数の a、b、および c を add() の本体で使用することができます。

```
class x
{
public:
      int add()             // inline member function add
      {return a+b+c;};
private:
      int a,b,c;
};
```

インライン・メンバー関数

クラス定義の内部にメンバー関数を定義するか、またはクラス定義にメンバー関数をすでに宣言している (しかし、定義はされていない) 場合は、メンバー関数をクラス定義の外側で定義できます。

そのクラス・メンバー・リスト内で定義されるメンバー関数は、*インライン・メンバー関数* と呼ばれます。コードを数行含んでいるメンバー関数は、通常インラインで宣言されます。上記の例では、add() がインライン・メンバー関数です。クラス定義の外側にメンバー関数を定義する場合、メンバー関数は、クラス定義を囲むネーム・スペース・スコープ内に入れる必要があります。スコープ・レゾリューション (::) 演算子を使用して、メンバー関数名を修飾することもできます。

インライン・メンバー関数を宣言するのと同等な方法は、**inline** キーワードを指定してクラス内でその関数を宣言するか (そしてクラスの外側で関数を定義する)、または **inline** キーワードを使用して、クラス宣言の外側でその関数を定義することです。

次の例では、メンバー関数 Y::f() は、インライン・メンバー関数です。

```
struct Y {
private:
  char a*;
public:
  char* f() { return a; }
};
```

次の例は、直前の例と同等なものです。Y::f() が、インライン・メンバー関数です。

```
struct Y {
private:
  char a*;
public:
  char* f();
};

inline char* Z::f() { return a; }
```

リンケージはデフォルトでは外部結合なので、**inline** 指定子はメンバー関数または非メンバー関数のリンケージには影響を与えません。

ローカル・クラスのメンバー関数

メンバー関数

ローカル・クラスのメンバー関数は、そのクラス定義の中で定義する必要があります。結果として、ローカル・クラスのメンバー関数は、暗黙的にインライン関数です。これらのインライン・メンバー関数には、リンケージはありません。

関連参照
- 238 ページの『フレンド』
- 235 ページの『静的メンバー関数』
- 135 ページの『第 7 章 関数』
- 155 ページの『インライン関数』
- 219 ページの『ローカル・クラス』

const および volatile メンバー関数

C++ **const** 修飾子を指定して宣言されたメンバー関数は、定数オブジェクトおよび非定数オブジェクトに対して呼び出すことができます。非定数メンバー関数は、非定数オブジェクトに対してのみ呼び出すことができます。同様に、**volatile** 修飾子を指定して宣言されたメンバー関数は、volatile オブジェクトおよび非 volatile オブジェクトに対して呼び出すことができます。非 volatile メンバー関数は、非 volatile オブジェクトに対してのみ呼び出すことができます。

関連参照
- 60 ページの『型修飾子』
- 61 ページの『const 型修飾子』

仮想メンバー関数

C++ 仮想メンバー関数は、キーワード **virtual** によって宣言されます。この関数を使用すると、メンバー関数の動的バインディングが可能となります。仮想関数は、すべてメンバー関数でなければならないため、仮想メンバー関数は、単に*仮想関数* と呼ばれます。

関数の宣言内の純粋指定子によって仮想関数の定義が置き換えられた場合、その関数は純粋を宣言されたと言います。純粋仮想関数を 1 つでも持つクラスは、*抽象クラス* と呼ばれます。

関連参照
- 262 ページの『仮想関数』
- 268 ページの『抽象クラス』

特殊なメンバー関数

C++ *特殊なメンバー関数* は、クラス・オブジェクトの作成、破棄、初期化、変換、およびコピーを行う場合に使用されます。そのような関数には、以下のものが含まれます。
- コンストラクター
- デストラクター
- 変換コンストラクター
- 変換関数
- コピー・コンストラクター

関連参照
- 271 ページの『第 15 章 特殊なメンバー関数』

メンバー・スコープ

C++ クラス・メンバー・リスト内ですでに宣言してあるが、定義はしていないメンバー関数と静的メンバーは、それらのクラス宣言の外側で定義することができます。非静的データ・メンバーは、それらのクラスのオブジェクトが作成されたときに定義されます。静的データ・メンバーの宣言は、定義ではありません。メンバー関数の宣言は、関数の本体も指定されている場合には、定義になります。

クラス・メンバーの定義がクラス宣言の外側にある場合、メンバー名は、:: (スコープ・レゾリューション) 演算子を使用して、クラス名によって修飾する必要があります。

以下の例では、クラス宣言の外側でメンバー関数を定義しています。

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

struct X {
  int a, b ;

  // member function declaration only
  int add();
};

// global variable
int a  = 10;

// define member function outside its class declaration
int X::add() { return a + b; }

int main() {
  int answer;
  X xobject;
  xobject.a = 1;
  xobject.b = 2;
  answer = xobject.add();
  cout << xobject.a << " + " << xobject.b << " = " << answer << endl;
}
```

この例の出力は 1 + 2 = 3 となります。

すべてのメンバー関数は、それらがクラス宣言の外側で定義されている場合であっても、クラス・スコープ内にあります。上記の例では、メンバー関数 add() はデータ・メンバー a を戻しますが、グローバル変数 a は戻しません。

クラス・メンバーの名前は、そのクラスに対してローカルなものです。 . (ドット)、-> (矢印)、または :: (スコープ・レゾリューション) 演算子の、いずれのクラス・アクセス演算子も使用しない場合、使用できるクラス・メンバーは、そのクラスとネスト・クラス内のメンバー関数のクラス・メンバーのみです。ネスト・クラス内で、:: 演算子で修飾されていないものは、型、列挙、および静的メンバーしか使用できません。

メンバー関数本体で名前を検索する順序は、次のとおりです。
1. メンバー関数本体自体の中
2. すべての囲みクラスの中 (それらのクラスの継承メンバーを含めて)
3. 本体宣言の字句範囲の中

継承メンバーを含めた、囲みクラスの検索の例を、以下に示します。

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
```

```
class Z : A {
      class Y : B {
            class X : C { int f(); /* ... */ };
      };
};
int Z::Y::X f()
{
      char j;
      return 0;
}
```

この例で、関数 f の定義内での名前 j の検索は、以下の順序に従います。
1. 関数 f の本体の中
2. X およびその基底クラス C の中
3. Y およびその基底クラス B の中
4. Z およびその基底クラス A の中
5. f の本体の字句範囲の中。この場合はグローバル・スコープ

収容クラスが検索されるときは、収容クラスの定義とそれらの基底クラスだけが検索されるということに留意してください。基底クラスの定義を含むスコープ (この例ではグローバル・スコープ) は、検索しません。

関連参照

- 4 ページの『クラス・スコープ』

メンバーへのポインター

C++ メンバーへのポインターを使用すると、クラス・オブジェクトの非静的メンバーを参照することができます。メンバーへのポインターを使用して静的クラス・メンバーを指すことはできません。静的メンバーのアドレスは、特定のオブジェクトに関連付けられていないからです。静的クラス・メンバーを指すためには、標準のポインターを使用する必要があります。

メンバー関数へのポインターは、関数へのポインターと同じ方法で使用することができます。メンバー関数へのポインターを比較し、値を割り当て、さらにそれらを使用してメンバー関数を呼び出すことができます。メンバー関数の型は、番号、引数の型、および戻りの型が同じ非メンバー関数と同じではないことに注意してください。

メンバーへのポインターは、以下の例に示すように宣言し、使用することができます。

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

class X {
public:
  int a;
  void f(int b) {
    cout << "The value of b is "<< b << endl;
  }
};

int main() {

  // declare pointer to data member
  int X::*ptiptr = &X::a;

  // declare a pointer to member function
  void (X::* ptfptr) (int) = &X::f;
```

```
  // create an object of class type X
  X xobject;

  // initialize data member
  xobject.*ptiptr = 10;

  cout << "The value of a is " << xobject.*ptiptr << endl;

  // call member function
  (xobject.*ptfptr) (20);
}
```

この例の出力は次のようになります。

```
The value of a is 10
The value of b is 20
```

複雑な構文を簡単にするために、**typedef** がメンバーへのポインターであると宣言することができます。メンバーへのポインターは、以下のコード・フラグメントに示すように宣言し、使用することができます。

```
typedef int X::*my_pointer_to_member;
typedef void (X::*my_pointer_to_function) (int);

int main() {
  my_pointer_to_member ptiptr = &X::a;
  my_pointer_to_function ptfptr = &X::f;
  X xobject;
  xobject.*ptiptr = 10;
  cout << "The value of a is " << xobject.*ptiptr << endl;
  (xobject.*ptfptr) (20);
}
```

メンバーへのポインター演算子 `.*` および `->*` は、特定のクラス・オブジェクトのメンバーへのポインターをバインドする際に用いられます。 `()` (関数呼び出し演算子) の優先順位の方が `.*` および `->*` よりも高いため、`ptf` によって指示される関数を呼び出す際は小括弧を使用することが必要です。

関連参照

- 117 ページの『メンバーを指す C++ ポインター演算子 (.* ->*)』
- 29 ページの『オブジェクト』

this ポインター

C++ キーワード **this** は、特定の型のポインターを識別します。クラス `A` の `x` という名前のオブジェクトを作成し、クラス `A` には、非静的メンバー関数 `f()` があるとします。関数 `x.f()` を呼び出す場合、`f()` の本体にあるキーワード **this** は、`x` のアドレスです。 **this** ポインターを宣言したり、またはそれに割り当てたりすることはできません。

静的メンバー関数は、**this** ポインターを持ちません。

クラス型 **X** のメンバー関数に対する `this` ポインターの型は、`X* const` です。メンバー関数が **const** 修飾子を用いて宣言されている場合、クラス **X** のそのメンバー関数に対する `this` ポインターの型は、`const X* const` です。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++; }
};
```

this ポインター

コンパイラーは、関数 f() の本体で、ステートメント a++ を許可しません。関数 f() では、**this** ポインターは、A* const 型です。関数 f() は、**this** が指すオブジェクトの一部を変更しようとしています。

this ポインターは、すべての非静的メンバー関数呼び出しに隠れた引数として渡され、すべての非静的関数本体の中のローカル変数として使用することができます。

例えば、メンバー関数本体内で **this** ポインターを使用することによって、メンバー関数が呼び出される特定のクラス・オブジェクトを参照することができます。以下の例で示すコードによって作成される出力は、a = 5 です。

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

struct X {
private:
  int a;
public:
  void Set_a(int a) {

    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

メンバー関数 Set_a() では、ステートメント this->a = a は、**this** ポインターを使用して、自動変数 a によって隠された xobj.a を検索します。

クラス・メンバー名が隠蔽されていない限り、クラス・メンバー名の使用は、**this** ポインターとクラス・メンバー・アクセス演算子 (->) を用いたクラス・メンバー名の使用と同じです。

以下のテーブルの最初の列は、**this** ポインターを指定しないで、クラス・メンバーを使用するコードの例を示しています。 2 番目の列にあるコードは、変数 THIS を使用して、最初の列で、その使用が隠蔽されている **this** ポインターをシミュレートします。

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
```

関連参照

- 205 ページの『代入の多重定義』

- 292 ページの『コピー・コンストラクター』

静的メンバー

C++ クラス・メンバーは、クラス・メンバー・リストで、ストレージ・クラス指定子 **static** を使用して宣言することができます。プログラム中の 1 つのクラスのすべてのオブジェクトが、静的メンバーの 1 つのコピーのみを共用します。静的メンバーを持つクラスのオブジェクトを宣言すると、その静的メンバーはそのクラス・オブジェクトの一部にはなりません。

静的メンバーの一般的な使用法は、クラスの全オブジェクトに共通なデータを記録する際に使用することです。例えば、静的データ・メンバーをカウンターとして使用して、作成された特定のクラス型のオブジェクト数を保管することができます。新しいオブジェクトが作成されるたびに、この静的データ・メンバーを増やして、オブジェクトの総数を記録することができます。

:: (スコープ・レゾリューション) 演算子を使用してクラス名を修飾して、静的メンバーにアクセスすることができます。次の例では、型 X のオブジェクトが宣言されていなくても、クラス型 X の静的メンバー f() を、X::f(): として参照することができます。

```
struct X {
  static int f();
};

int main() {
  X::f();
}
```

関連参照
- 33 ページの『static ストレージ・クラス指定子』
- 223 ページの『クラス・メンバー・リスト』

静的メンバーでのクラス・アクセス演算子の使用

C++ 静的メンバーを参照するのに、クラス・メンバー・アクセス構文を使用する必要はありません。つまり、クラス X の静的メンバー s にアクセスするために、式 X::s が使用できます。以下の例は、静的メンバーへのアクセスを説明しています。

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

struct A {
  static void f() { cout << "In static function A::f()" << endl; }
};

int main() {

  // no object required for static member
  A::f();

  A a;
  A* ap = &a;
  a.f();
  ap->f();
}
```

ステートメント A::f()、a.f()、および ap->f() の 3 つはすべて、同じ静的メンバー関数 A::f() を呼び出します。

そのクラスの同じスコープ内、または静的メンバーのクラスから派生したクラスのスコープ内にある、静的メンバーを直接参照することができます。次の例は、後者のケースを説明しています (静的メンバーのクラスから派生したクラスのスコープ内にある静的メンバーを直接参照する)。

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

int g() {
   cout << "In function g()" << endl;
   return 0;
}

class X {
   public:
      static int g() {
         cout << "In static member function X::g()" << endl;
         return 1;
      }
};

class Y: public X {
   public:
      static int i;
};

int Y::i = g();

int main() { }
```

次に、上記のコード出力を示します。

```
In static member function X::g()
```

初期設定 `int Y::i = g()` は、`X::g()` を呼び出しますが、グローバル・ネーム・スペースに宣言されている関数、`g()` は呼び出しません。

1 つのクラスの全オブジェクトによって共用される静的メンバーは 1 つだけなので、クラス・オブジェクトとのいかなる関連付けとも関係なく、静的メンバーを参照することができます。静的メンバーは、たとえクラスのオブジェクトが宣言されていなくても存在します。

関連参照
- 91 ページの『ドット演算子 .』
- 91 ページの『矢印演算子 ->』

静的データ・メンバー

C++ クラスの静的データ・メンバーのコピーが 1 つだけ存在し、そのクラスのすべてのオブジェクトで共用されます。

ネーム・スペース・スコープのクラスの静的データ・メンバーには、外部リンケージがあります。静的データ・メンバーは、通常のクラス・アクセス規則に従いますが、ファイル・スコープで初期化できる点が異なります。静的データ・メンバーとその初期化指定子は、そのクラスの、他の静的の private メンバーおよび保護メンバーにアクセスすることができます。静的データ・メンバー用の初期化指定子は、メンバーを宣言するクラスのスコープ内にあります。

静的データ・メンバーは、**void**、あるいは **const** または **volatile** で修飾された **void** を除く、あらゆる型に指定できます。

静的メンバー

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。静的データ・メンバーの定義は、外部変数の定義と同じです。静的メンバーは、ネーム・スペース・スコープのクラス宣言の外側で定義することが必要です。

次に例を示します。

```
class X
{
public:
      static int i;
};
int X::i = 0; // definition outside class declaration
```

静的データ・メンバーを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。

以下の例は、他の静的メンバー (そのメンバーが private であっても) を使用して、どのように静的メンバーを初期化できるかを示しています。

```
class C {
      static int i;
      static int j;
      static int k;
      static int l;
      static int m;
      static int n;
      static int p;
      static int q;
      static int r;
      static int s;
      static int f() { return 0; }
      int a;
public:
      C() { a = 0; }
      };

C c;
int C::i = C::f();    // initialize with static member function
int C::j = C::i;      // initialize with another static data member
int C::k = c.f();     // initialize with member function from an object
int C::l = c.j;       // initialize with data member from an object
int C::s = c.a;       // initialize with nonstatic data member
int C::r = 1;         // initialize with a constant value

class Y : private C {} y;

int C::m = Y::f();
int C::n = Y::r;
int C::p = y.r;       // error
int C::q = y.f();     // error
```

`y` は、`C` から private に派生したクラスのオブジェクトであるため、`C::p` および `C::x` の初期化は、エラーとなり、このオブジェクトのメンバーは、`C` のメンバーからはアクセスできません。

静的データ・メンバーが **const** 整数型、または **const** 列挙型のメンバーである場合、*定数初期化指定子* を静的データ・メンバーの宣言で指定できます。この定数初期化指定子は、整数定数式でなければなりません。定数初期化指定子は、定義ではないことに注意してください。まだ、囲みネーム・スペースに静的メンバーを定義する必要があります。次の例は、このことを示しています。

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

struct X {
```

```
  static const int a = 76;
};

const int X::a;

int main() {
  cout << X::a << endl;
}
```

静的データ・メンバー a の最後に宣言されているトークン = 76 が、定数初期化指定子です。

注:

1. プログラム内には、静的メンバーの定義は 1 つしか入れられません。名前のないクラスや、名前のないクラスに含まれるクラスは、静的データ・メンバーを持つことができません。
2. 静的データ・メンバーを **mutable** として宣言できません。
3. 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。
4. C++ クラス・オブジェクトで 'static __thread' メンバー・データを宣言できます。

静的メンバー関数

C++ 同じ名前、および引数の数と型が同じである、静的メンバー関数と非静的メンバー関数を持つことはできません。

静的データ・メンバーのように、クラス A のオブジェクトを使用しないで、クラス A の静的メンバー関数 f() にアクセスできます。

静的メンバー関数は、**this** ポインターを持ちません。次の例は、このことを示しています。

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

struct X {
private:
  int i;
  static int si;
public:
  void set_i(int arg) { i = arg; }
  static void set_si(int arg) { si = arg; }

  void print_i() {
    cout << "Value of i = " << i << endl;
    cout << "Again, value of i = " << this->i << endl;
  }

  static void print_si() {
    cout << "Value of si = " << si << endl;
//    cout << "Again, value of si = " << this->si << endl;
  }

};

int X::si = 77;       // Initialize static data member

int main() {
  X xobj;
  xobj.set_i(11);
  xobj.print_i();

  // static data members and functions belong to the class and
  // can be accessed without using an object of class X
```

```
  X::print_si();
  X::set_si(22);
  X::print_si();
}
```

上記の例の出力は、以下のとおりです。

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

コンパイラーは、このメンバー関数が静的として宣言されていて、そのためメンバー関数が **this** ポインターを持っていないので、関数 `A::print_si()` で、メンバー・アクセス操作 `this->si` を認めません。

非静的メンバー関数の **this** ポインターを使用して、静的メンバー関数を呼び出すことができます。以下の例では、非静的メンバー関数の `printall()` が、`this` ポインターを使用して静的メンバー関数の **f()** を呼び出します。

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

class C {
  static void f() {
    cout << "Here is i: " << i << endl;
  }
  static int i;
  int j;
public:
  C(int firstj): j(firstj) { }
  void printall();
};

void C::printall() {
  cout << "Here is j: " << this->j << endl;
  this->f();
}

int C::i = 3;

int main() {
  C obj_C(0);
  obj_C.printall();
}
```

上記の例の出力は、以下のとおりです。

```
Here is j: 0
Here is i: 3
```

キーワード **virtual**、**const**、**volatile**、または **const volatile** を使用した静的メンバー関数の宣言はできません。

静的メンバー関数がアクセスできるのは、その関数が宣言されているクラスの静的メンバー、列挙子、およびネストされた型の名前だけです。静的メンバー関数 `f()` が、クラス `X` のメンバーであるとします。静的メンバー関数 `f()` は、非静的メンバー `X` または基底クラス `X` の非静的メンバーにアクセスできません。

関連参照

• 229 ページの『this ポインター』

メンバー・アクセス

C++ メンバー・アクセスは、式または宣言内で、クラス・メンバーがアクセス可能かどうかを判別します。 x がクラス A のメンバーであるとします。このクラス・メンバー x は、以下のいずれかのレベルのアクセス可能性を持つものとして宣言することができます。

- **public**: x は、private や protected で定義したアクセス制限以外の場所なら、どこででも使用できます。
- **private**: x は、クラス A のメンバーとフレンドだけが使用できます。
- **protected**: x は、クラス A のメンバーとフレンド、およびクラス A から派生したクラスのメンバーとフレンドだけが使用できます。

キーワード **class** を指定して宣言されたクラスのメンバーのデフォルトは、private です。キーワード **struct** または **union** を指定して宣言されたクラスのメンバーのデフォルトは、public です。

クラス・メンバーのアクセスを制御するには、*アクセス指定子* **public**、**private**、または **protected** をクラス・メンバー・リストのラベルとして使用します。次の例はこれらのアクセス指定子を示しています。

```
struct A {
  friend class C;
private:
  int a;
public:
  int b;
protected:
  int c;
};

struct B : A {
  void f() {
    // a = 1;
    b = 2;
    c = 3;
  }
};

struct C {
  void f(A x) {
    x.a = 4;
    x.b = 5;
    x.c = 6;
  }
};

int main() {
  A y;
//  y.a = 7;
  y.b = 8;
//  y.c = 9;

  B z;
//  z.a = 10;
  z.b = 11;
//  z.c = 12;
}
```

次の表は、上記の例のようなさまざまなスコープ内のデータ・メンバー A::a、A::b、および A::c へのアクセスについて示しています。

スコープ	A::a	A::b	A::c
関数 B::f()	アクセス不可。メンバー A::a は、private です。	アクセス可能。メンバー A::b は、public です。	アクセス可能。クラス B は、A から継承します。

スコープ	A::a	A::b	A::c
関数 C::f()	アクセス可能。クラス C は、A のフレンドです。	アクセス可能。メンバー A::b は、public です。	アクセス可能。クラス C は、A のフレンドです。
main() のオブジェクト y	アクセス不可。 メンバー y.a は、private です。	アクセス可能。メンバー y.a は、public です。	アクセス不可。メンバー y.c は、protected です。
main() のオブジェクト z	アクセス不可。メンバー z.a は、private です。	アクセス可能。メンバー z.a は、public です。	アクセス不可。メンバー z.c は、protected です。

アクセス指定子は、次のアクセス指定子まで、またはクラス定義の終わりまでその後に続くメンバーのアクセス可能性を指定します。任意の数のアクセス指定子を、任意の順序で使用することができます。クラス定義内に後からクラス・メンバーを定義する場合、そのアクセス指定は、その宣言と同一にする必要があります。次の例は、このことを示しています。

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

コンパイラーは、クラス B がすでに private として宣言されているため、このクラスの定義を許可しません。

クラス・メンバーには、それがそのクラス内、またはクラスの外側に定義されたかどうかには関係なく、同じアクセス制御があります。

アクセス制御は、名前に対応しています。特に、アクセス制御を typedef 名に追加する場合、それは typedef 名だけに影響します。次の例は、このことを示しています。

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

int main() {
  A::C x;
//  A::B y;
}
```

コンパイラーは、typedef 名 A::C が public なので、宣言 A::C x を許可します。コンパイラーは、A::B は、private なので、宣言 A::B y を認めません。

アクセス可能性と可視性は、別々のものであることに注意してください。可視性は、C++ のスコープ規則に基づきます。クラス・メンバーが、可視であり、同時にアクセス不能ということはあり得ます。

フレンド

C++ クラス X のフレンドとは、X のメンバーではないが、X のメンバーと同じ X へのアクセスを認可されている関数またはクラスです。クラス・メンバー・リスト内で、**friend** 指定子を使用して宣言された関数は、そのクラスの*フレンド関数* と呼ばれます。別のクラスのメンバー・リスト内で **friend** 指定子を用いて宣言されたクラスは、そのクラスの*フレンド・クラス* と呼ばれます。

クラス Y を定義してからでなければ、Y の任意のメンバーを、別のクラスのフレンドとして宣言することはできません。

以下の例では、フレンド関数 print は、クラス Y のメンバーであり、クラス X の private データ・メンバー a および b にアクセスします。

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

class X;

class Y {
public:
  void print(X& x);
};

class X {
  int a, b;
  friend void Y::print(X& x);
public:
  X() : a(1), b(2) { }
};

void Y::print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
}

int main() {
  X xobj;
  Y yobj;
  yobj.print(xobj);
}
```

上記の例の出力は、以下のとおりです。

```
a is 1
b is 2
```

クラス全体をフレンドとして宣言することができます。クラス F が、クラス A のフレンドだとします。メンバー関数、およびクラス F の静的データ・メンバー定義はいずれも、クラス A へのアクセスを行えます。

次の例では、フレンド・クラス F には、クラス X の private データ・メンバー a および b にアクセスする、メンバー関数 print があります。これは、前述の例のフレンド関数 print と同じタスクを実行します。また、クラス F に宣言されたその他のメンバーも、クラス X のメンバーすべてにアクセスできます。

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

class X {
  int a, b;
  friend class F;
public:
  X() : a(1), b(2) { }
};

class F {
public:
  void print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
  }
};
```

```
int main() {
  X xobj;
  F fobj;
  fobj.print(xobj);
}
```

上記の例の出力は、以下のとおりです。

```
a is 1
b is 2
```

クラスをフレンドとして宣言する場合は、詳述型指定子を使用する必要があります。次の例は、このことを示しています。

```
class F;
class G;
class X {
  friend class F;
  friend G;
};
```

フレンド宣言では、クラスを定義できません。例えば、コンパイラーは次のコードを許可しません。

```
class F;
class X {
  friend class F { };
};
```

しかし、フレンド宣言で関数を定義できます。クラスは、非ローカル・クラスの関数で、関数名が非修飾であり、ネーム・スペース・スコープを持っている必要があります。次の例は、このことを示しています。

```
class A {
  void g();
};

void z() {
  class B {
//    friend void f() { };
  };
}

class C {
//  friend void A::g() { }
  friend void h() { }
};
```

コンパイラーは、`f()` または `g()` の関数定義を許可しません。コンパイラーは、`h()` の定義は認めます。

ストレージ・クラス指定子を使用して、フレンドを宣言することはできません。

関連参照
- 237 ページの『メンバー・アクセス』
- 250 ページの『継承されたメンバー・アクセス』

フレンドのスコープ

C++ フレンド宣言で最初に導入されるフレンド関数またはフレンド・クラスの名前は、フレンド関係を認可するクラス (*囲みクラスとも呼ばれます*) のスコープ内にはなく、フレンド関係を認可するクラスのメンバーでもありません。

フレンド宣言で最初に導入された関数の名前は、囲みクラスが含まれている最初の非クラス・スコープのスコープ内にあります。フレンド宣言で与えられる関数の本体は、クラス内で定義されるメンバー関数と同じ方法で処理されます。定義の処理は、最外部の囲みクラスの終わりまで開始されません。さらに、関数定義の本体内の修飾されない名前による検索は、その関数定義が入っているクラスから始められます。

フレンド宣言で最初に宣言されるクラスは、**extern** 宣言と同等です。次に例を示します。

```
class B {};
class A
{
      friend class B; // global class B is a friend of A
};
```

フレンド・クラスの名前が、フレンド宣言よりも前に導入されている場合、コンパイラーは、そのフレンド・クラスの名前と一致するクラス名の検索を、フレンド宣言のスコープの先頭から開始します。ネスト・クラスの宣言の後に同じ名前のフレンド・クラスの宣言が続いている場合、そのネスト・クラスは、囲みクラスのフレンドです。

フレンド・クラス名のスコープは、最初の非クラスの囲みスコープです。次に例を示します。

```
class A {
   class B { // arbitrary nested class definitions
      friend class C;
   };
};
```

は、以下と同等です。

```
class C;
class A {
   class B { // arbitrary nested class definitions
      friend class C;
   };
};
```

フレンド関数が別のクラスのメンバーである場合には、スコープ・レゾリューション演算子 (**::**) を使用することが必要です。次に例を示します。

```
class A {
public:
  int f() { }
};

class B {
  friend int A::f();
};
```

基底クラスのフレンドは、その基底クラスから派生したクラスには、継承されません。次の例は、このことを示しています。

```
class A {
  friend class B;
  int a;
};

class B { };

class C : public B {
  void f(A* p) {
//    p->a = 2;
  }
};
```

C は A のフレンドから継承していますが、クラス C がクラス A のフレンドではないので、コンパイラーは、ステートメント p->a = 2 を許可しません。

フレンド関係は、移行できません。次の例は、このことを示しています。

```
class A {
  friend class B;
  int a;
};

class B {
  friend class C;
};

class C {
  void f(A* p) {
//    p->a = 2;
  }
};
```

C は A のフレンドのフレンドですが、クラス C がクラス A のフレンドではないので、コンパイラーは、ステートメント p->a = 2 を許可しません。

ローカル・クラスにフレンドを宣言し、フレンドの名前が非修飾である場合、コンパイラーは、最も内側にある囲みの非クラス・スコープ内でのみ名前を検索します。関数を宣言してから、ローカル・スコープのフレンドとして関数を宣言する必要があります。クラスでこれを実行する必要はありません。しかし、フレンド・クラスの宣言は、囲みスコープ内の、同じ名前のクラスを隠します。次の例は、このことを示しています。

```
class X { };
void a();

void f() {
  class Y { };
  void b();
  class A {
    friend class X;
    friend class Y;
    friend class Z;
//    friend void a();
    friend void b();
//    friend void c();
  };
  ::X moocow;
//  X moocow2;
}
```

上記の例では、コンパイラーは、次のステートメントを認めます。

- friend class X: このステートメントは、このクラスが別の方法で宣言されていなくても、::X を A のフレンドとしてではなく、ローカル・クラス X をフレンドとして宣言しています。
- friend class Y: ローカル・クラス Y は、 f() のスコープに宣言されました。
- friend class Z: このステートメントは、Z が別の方法で宣言されていなくても、ローカル・クラス Z を A のフレンドとして宣言しています。
- friend void b(): 関数 b() は、f() のスコープに宣言されました。
- ::X moocow: この宣言は、非ローカル・クラス ::X のオブジェクトを作成します。

コンパイラーは、次のステートメントを許可しません。

- `friend void a()`: このステートメントは、関数 `a()` を、ネーム・スペース・スコープに宣言されたと認めません。関数 `a()` が、`f()` に宣言されていないので、コンパイラーはこのステートメントを認めません。
- `friend void c()`: 関数 `c()` が、`f()` のスコープに宣言されていないので、コンパイラーはこのステートメントを認めません。
- `X moocow2`: この宣言は、非ローカル・クラス `::X` ではなく、ローカル・クラス `X` のオブジェクトを作成しようとします。ローカル・クラス `X` が定義されていないので、コンパイラーは、このステートメントを認めません。

関連参照

- 219 ページの『ローカル・クラス』

フレンドのアクセス

C++ クラスのフレンドは、そのクラスの private メンバーおよび保護メンバーにアクセスすることができます。通常、クラスの private メンバーには、そのクラスのメンバー関数を介してのみアクセスでき、クラスの保護メンバーにはクラスのメンバー関数、またはクラスから派生したクラスを介してのみ、アクセスすることができます。

フレンド宣言は、アクセス指定子には影響されません。

関連参照

- 237 ページの『メンバー・アクセス』

第 14 章 継承

C++ 継承 とは、既存のクラスを変更しないで、再利用したり拡張したりするメカニズムのことです。

継承は、オブジェクトをクラスに組み込むのとほぼ同じです。クラス A のオブジェクト x を、B のクラス定義に宣言するとします。その結果、クラス B は、クラス A の public データ・メンバーおよびメンバー関数のすべてにアクセスできます。しかし、クラス B では、クラス A のデータ・メンバーおよびメンバー関数にアクセスするのに、オブジェクト x を介してアクセスする必要があります。次の例は、このことを示しています。

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

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};

class B {
public:
   A x;
};

int main() {
   B obj;
   obj.x.f(20);
   cout << obj.x.g() << endl;
//   cout << obj.g() << endl;
}
```

main 関数のオブジェクト obj は、ステートメント obj.x.f(20) を使用したデータ・メンバー B::x を介して、関数 A::f() にアクセスします。オブジェクト obj は、同様な方法で、ステートメント obj.x.g() で A::g() にアクセスします。コンパイラーは、g() がクラス A のメンバー関数であり、クラス B のメンバー関数ではないので、ステートメント obj.g() を許可しません。

継承メカニズムにより、上記の例で示した obj.g() のようなステートメントを使用できます。ステートメントを有効にするには、 g() が、クラス B のメンバー関数である必要があります。

継承により、別のクラスのメンバーの名前および定義を、新規クラスの一部としてインクルードできます。新規クラスにインクルードしたいメンバーを持つ元のクラスは、基底クラス と呼ばれます。新規クラスは、基底クラスから派生します。新規クラスは、基底クラスの型のサブオブジェクト を含みます。次の例は、継承メカニズムを使用してクラス B にクラス A のメンバーへのアクセスを与える点を除いては、直前の例と同じです。

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

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};
```

```
class B : public A { };

int main() {
   B obj;
   obj.f(20);
   cout << obj.g() << endl;
}
```

クラス A は、クラス B の基底クラスです。クラス A のメンバーの名前、および定義は、クラス B の定義にインクルードされます。つまり、クラス B は、クラス A のメンバーを継承します。クラス B は、クラス A から派生します。クラス B には、型 A のサブオブジェクトが含まれます。

派生クラスに新たにデータ・メンバーやメンバー関数を追加することもできます。新たに派生させたクラスで基底クラスのメンバー関数やデータをオーバーライドすることによって、既存メンバー関数やデータのインプリメンテーションを変更することができます。

別の派生クラスからもクラスを派生できます。その結果、別のレベルの継承を作成します。次の例は、このことを示しています。

```
struct A { };
struct B : A { };
struct C : B { };
```

クラス B は、A の派生クラスでもあり、同時に、C の基底クラスでもあります。継承のレベル数は、リソースによってのみ限定されます。

多重継承 を使用すると、複数の基底クラスの属性を継承する派生クラスを作成することができます。派生クラスは、その全基底クラスからメンバーを継承するので、その結果あいまいさが生じる可能性があります。例えば、2 つの基底クラスに同じ名前のメンバーがある場合、派生クラスでは 2 つのメンバーを暗黙的に区別することができません。多重継承を使用するときは、基底クラスの名前へのアクセスがあいまいにならないように注意してください。

直接基底クラス とは、その派生クラスの宣言の中に、基底指定子として直接現れる基底クラスのことです。

間接基底クラス とは、派生クラスの宣言の中には直接出てこないが、その基底クラスの 1 つを介して派生クラスで使用できる基底クラスのことです。あるクラスについて、直接基底クラスでない基底クラスは、すべて間接基底クラスです。次の例は、直接基底クラスおよび間接基底クラスを示しています。

```
class A {
  public:
    int x;
};
class B : public A {
  public:
    int y;
};
class C : public B { };
```

クラス B は、C の直接基底クラスです。クラス A は、B の直接基底クラスです。クラス A は、C の間接基底クラスです。 (x および y は、クラス C のデータ・メンバーになります。)

ポリモアフィック関数 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。

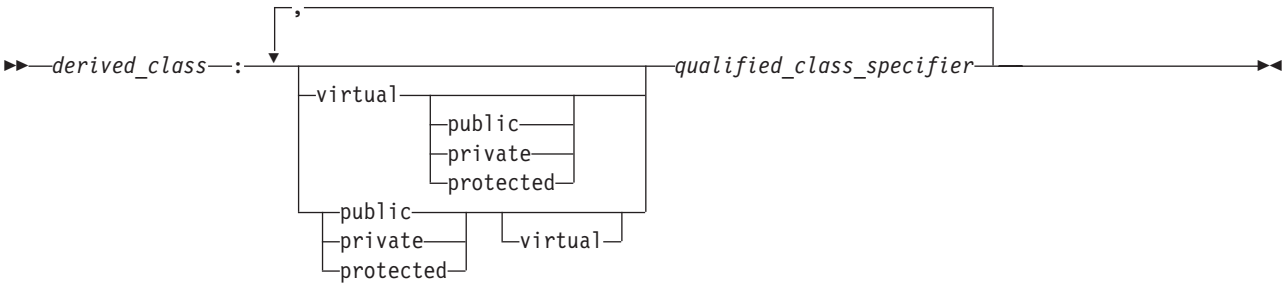
- C++ が、仮想関数を提供します。*仮想関数* は、派生を介して関連付けられている、いくつかの様々なユーザー定義の型について呼び出すことができる関数です。仮想関数は、実行時に動的にバインドされます。

関連参照
- 258 ページの『マルチアクセス』
- 256 ページの『多重継承』
- 262 ページの『仮想関数』

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。

構文 - 派生クラスの派生



派生クラスの宣言の中で、派生クラスの基底クラスをリストします。派生クラスは、これらの基底クラスからそのメンバーを継承します。

qualified_class_specifier は、クラス宣言で事前に宣言されているクラスである必要があります。

アクセス指定子 は、**public**、**private**、または **protected** のいずれかです。

virtual キーワードは、仮想基底クラスの宣言に使用できます。

次の例は、派生クラス D と、基底クラス V、B1、および B2 の宣言を示しています。クラス B1 は、クラス V から派生し、D の基底クラスなので、基底クラスでもあり派生クラスでもあります。

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

宣言されているが定義されていないクラスは、基底リストに入れることができません。

次に例を示します。

```
class X;

// error
class Y: public X { };
```

コンパイラーは、X が定義されていないので、クラス Y の宣言を許可しません。

派生

クラスを派生させると、派生クラスは、基底クラスのクラス・メンバーを継承します。継承されたメンバー (基底クラスのメンバー) は、派生クラスのメンバーと同様に参照することができます。次に例を示します。

```
class Base {
public:
  int a,b;
};

class Derived : public Base {
public:
  int c;
};

int main() {
  Derived d;
  d.a = 1;    // Base::a
  d.b = 2;    // Base::b
  d.c = 3;    // Derived::c
}
```

派生クラスには新しいクラス・メンバーを追加したり、既存の基底クラス・メンバーを再定義することもできます。上記の例では、派生クラスのメンバー `c` に加えて、派生クラス `d` の 2 つの継承されたメンバー `a` および `b` に値が代入されます。派生クラスの中で基底クラスのメンバーを再定義しても、`::` (スコープ・レゾリューション) 演算子を使用すれば、依然として基底クラスのメンバーを参照することができます。次に例を示します。

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

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  // call Derived::display()
  d.display();

  // call Base::display()
  d.Base::display();
}
```

上記の例の出力は、以下のとおりです。

```
Derived Class, Base Class
Base Class
```

派生クラスのオブジェクトは、基底クラスのオブジェクトと同様に操作できます。派生クラスのオブジェクトに対するポインターや参照を、その基底クラスに対するポインターや参照の代わりに使用することができます。例えば、派生クラスのオブジェクト D に対するポインターまたは参照を、D の基底クラスに対するポインターまたは参照を予期している関数に渡すことができます。これを実行するために明示的キャストを使用する必要はありません。標準型変換が実行されます。派生クラスに対するポインターを、明らかにアクセス可能な基底クラスを指すように、暗黙的に変換することができます。また派生クラスに対する参照を、基底クラスに対する参照に暗黙的に変換することもできます。

次の例では、派生クラスを指すポインターを基底クラスを指すポインターに変換する、標準型変換を示します。

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

class Base {
public:
  char* name;
  void display() {
    cout << name << endl;
  }
};

class Derived: public Base {
public:
  char* name;
  void display() {
    cout << name << ", " << Base::name << endl;
   }
};

int main() {
  Derived d;
  d.name = "Derived Class";
  d.Base::name = "Base Class";

  Derived* dptr = &d;

  // standard conversion from Derived* to Base*
  Base* bptr = dptr;

  // call Base::display()
  bptr->display();
}
```

上記の例の出力は、以下のとおりです。

```
Base Class
```

ステートメント Base* bptr = dptr は、Derived 型のポインターを Base 型のポインターに変換します。

この逆は認められていません。基底クラスのオブジェクトへのポインターや参照を、派生クラスへのポインターや参照に暗黙的に変換することはできません。例えば、クラス Base および Class が上記の例のように定義されている場合、コンパイラーは、次のコードを許可しません。

```
int main() {
  Base b;
  b.name = "Base class";

  Derived* dptr = &b;
}
```

コンパイラーは、ステートメントが暗黙的に Base 型のポインターを Derived 型のポインターに変換するので、ステートメント Derived* dptr = &b を許可しません。

派生クラスのメンバーと基底クラスのメンバーが同じ名前を持っている場合、基底クラス・メンバーは、派生クラスの中で隠されます。派生クラスのメンバーが基底クラスと同じ名前を持っている場合、基底クラス名は、派生クラスの中で隠されます。

関連参照

- 257 ページの『仮想基底クラス』
- 217 ページの『不完全なクラス宣言』
- 87 ページの『C++ スコープ・レゾリューション演算子 ::』

継承されたメンバー・アクセス

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

C++ 基底クラスから派生したどのクラスのメンバーおよびフレンドも、次のいずれかの方法を使用して、protected 非静的基底クラス・メンバーにアクセスすることができます。
- 直接または間接の派生クラスへのポインター
- 直接または間接の派生クラスへの参照
- 直接または間接の派生クラスのオブジェクト

基底クラスから private にクラスを派生させた場合、基底クラスの全 protected メンバーは、派生クラスの private メンバーになります。

派生クラス B のフレンドまたはメンバー関数で、基底クラスの A の protected 非静的メンバー x を参照する場合、A から派生したクラスに対するポインター、参照、またはオブジェクトを介して x にアクセスする必要があります。しかし、x にアクセスし、メンバーに対するポインターを作成している場合は、派生クラス B の名前を付けるネスト名前指定子で x を修飾する必要があります。

```
class A {
public:
protected:
  int i;
};


class B : public A {
  friend void f(A*, B*);
  void g(A*);
};

void f(A* pa, B* pb) {
//  pa->i = 1;
  pb->i = 2;

//  int A::* point_i = &A::i;
  int A::* point_i2 = &B::i;
}

void B::g(A* pa) {
//  pa->i = 1;
  i = 2;

//  int A::* point_i = &A::i;
```

```
  int A::* point_i2 = &B::i;
}

void h(A* pa, B* pb) {
//  pa->i = 1;
//  pb->i = 2;
}

int main() { }
```

クラス A には、protected データ・メンバーである、整数 i が入っています。 B は A から派生するので、B のメンバーは、 A の protected メンバーへのアクセスが可能です。関数 f() は、クラス B のフレンドです。

- コンパイラーは、pa が派生クラス B に対するポインターでないので、pa->i = 1 を許可しません。
- コンパイラーは、i が派生クラス B の名前で修飾されていないので、int A::* point_i = &A::i を許可しません。

関数 g() は、クラス B のメンバー関数です。コンパイラーが許可するあるいは許可しないステートメントについての直前の注釈のリストは、次の点を除き、g() にも適用できます。

- コンパイラーは、i = 2 は、this->i = 2 と同等なので、認めます。

関数 h() は、A の派生クラスのフレンドでもメンバーでもないので、h() は、A のどの protected メンバーにもアクセスすることはできません。

関連参照

- 77 ページの『参照』
- 29 ページの『オブジェクト』

規定クラス・メンバーのアクセス制御

C++ 派生クラスの宣言においては、派生クラスの基底リストの中の各基底クラスの前に、アクセス指定子を置くことができます。これによって、基底クラスから見たときの基底クラスの各メンバーのアクセス属性は、変更されませんが、派生クラスが、基底クラスのメンバーへのアクセス制御を制限できるようになります。

3 つのアクセス指定子のいずれかを使用して、クラスを派生させることができます。

- **public** 基底クラスでは、基底クラスの public および protected メンバーは、派生クラスにおいても public および protected メンバーです。
- **protected** 基底クラスにおいては、基底クラスの public および protected メンバーは、派生クラスの protected メンバーになります。
- **private** 基底クラスにおいては、基底クラスの public および protected メンバーは、派生クラスでは private メンバーになります。

すべての場合において、基底クラスの private メンバーは private のままです。基底クラスの private メンバーは、基底クラス内のフレンド宣言において、明示的にアクセスを認可されている場合でなければ、派生クラスから使用することはできません。

次の例では、クラス d は、クラス b から public に派生します。クラス b は、この宣言により、public 基底クラスに宣言されます。

```
class b { };
class d : public b // public derivation
{ };
```

構造体とクラスの両方を、派生クラス宣言の基底リストの中の基底クラスとして使用することができます。

- 派生クラスがキーワード **class** で宣言される場合、その基本リスト指定子にあるデフォルトのアクセス指定子は、**private** です。
- 派生クラスがキーワード **struct** で宣言される場合、その基本リスト指定子にあるデフォルトのアクセス指定子は、**public** です。

次の例では、基本リストで使用されるアクセス指定子がなく、派生クラスがキーワード **class** で宣言されているので、デフォルトで private の派生が使用されます。

```
struct B
{ };
class D : B // private derivation
{ };
```

クラスのメンバーおよびフレンドは、そのクラスのオブジェクトに対するポインターを暗黙的に次のいずれかに対するポインターに変換することができます。
- 直接 private 基底クラス
- protected 基底クラス (直接または間接)

関連参照

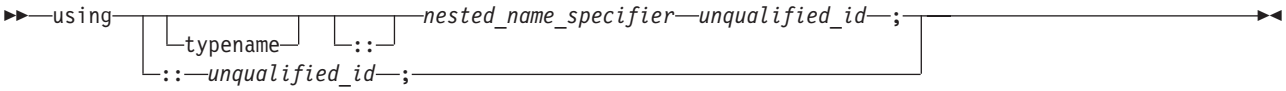
- 237 ページの『メンバー・アクセス』
- 227 ページの『メンバー・スコープ』

using 宣言およびクラス・メンバー

C++ クラス A の定義での using 宣言により、データ・メンバーまたはメンバー関数の*名前* を、A の基底クラスから A のスコープに導入できます。

規定および派生クラスからメンバー関数の多重定義セットを作成したい場合、またはクラス・メンバーのアクセスを変更したい場合は、クラス定義で using 宣言が必要です。

構文 - using 宣言



クラス A の using 宣言は、次のいずれかに名前を付けます。
- A の基底クラスのメンバー
- A の基底クラスのメンバーである無名共用体のメンバー
- A の基底クラスのメンバーである列挙型の列挙子

次の例は、このことを示しています。

```
struct Z {
  int g();
};

struct A {
  void f();
  enum E { e };
  union { int u; };
};

struct B : A {
  using A::f;
```

```
  using A::e;
  using A::u;
//  using Z::g;
};
```

コンパイラーは、Z が A の基底クラスでないので、using 宣言 using Z::g を許可しません。

using 宣言は、テンプレートに名前を付けることはできません。例えば、コンパイラーは次のコードを許可しません。

```
struct A {
  template<class T> void f(T);
};

struct B : A {
  using A::f<int>;
};
```

using 宣言で示されている名前のインスタンスは、いずれもアクセス可能でなければなりません。次の例は、このことを示しています。

```
struct A {
private:
  void f(int);
public:
  int f();
protected:
  void g();
};

struct B : A {
//  using A::f;
  using A::g;
};
```

コンパイラーは、int A::f() はアクセス可能ですが、void A::f(int) が B からアクセス不可能なので、using 宣言 using A::f を許可しません。

規定クラスおよび派生クラスからのメンバー関数の多重定義

C++ クラス A で f と名付けられたメンバー関数は、戻りの型や引数に関係なく、A の基底クラスで、他の f という名前のメンバーをすべて隠します。次の例は、このことを示しています。

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

struct B : A {
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
//  obj_B.f();
}
```

コンパイラーは、void B::f(int) の宣言が A::f() を隠しているので、関数呼び出し obj_B.f() を許可しません。

基底クラス A の関数を、派生クラス B で、隠蔽ではなく多重定義するには、using 宣言を使用して、関数の名前を B のスコープに導入します。以下の例は、using 宣言 using A::f を除いては、直前の例と同じです。

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

struct B : A {
  using A::f;
  void f(int) { }
};

int main() {
  B obj_B;
  obj_B.f(3);
  obj_B.f();
}
```

クラス B に using 宣言があるので、名前 f は 2 つの関数で多重定義されます。これで、コンパイラーは、関数呼び出し obj_B.f() を許可します。

同じ方法で仮想関数を多重定義できます。次の例は、このことを示しています。

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

struct A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
  B obj_B;
  B* pb = &obj_B;
  pb->f(3);
  pb->f();
}
```

上記の例の出力は、以下のとおりです。

```
void B::f(int)
void A::f()
```

関数 f を using 宣言を指定して、基底クラス A から派生クラス B に導入し、さらに A::f として同じパラメーター型を持つ B::f という名前の関数が存在するとします。関数 B::f は、関数 A::f と競合するというより、むしろそれを隠蔽します。次の例は、このことを示しています。

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

struct A {
  void f() { }
  void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
  using A::f;
  void f(int) { cout << "void B::f(int)" << endl; }
};
```

```
int main() {
  B obj_B;
  obj_B.f(3);
}
```

上記の例の出力は、以下のとおりです。

```
void B::f(int)
```

クラス・メンバーのアクセスの変更

C++ クラス B が、クラス A の直接基底クラスであるとします。クラス B が、クラス A のメンバーにアクセスすることを制限するには、アクセス指定子 **protected** または **private** のどちらかを使用して、B を A から派生させてください。

クラス B から継承された、クラス A のメンバー x のアクセスを増やすには、using 宣言を使用してください。 using 宣言を指定して x へのアクセスを制限することはできません。次のメンバーのアクセスを増やすことができます。

- **private** として継承されたメンバー。 (using 宣言は、メンバーの名前にアクセスするはずなので、**private** として宣言されたメンバーのアクセスを増やすことはできません。)
- **protected** として継承、または宣言されたメンバー。

次の例は、このことを示しています。

```
struct A {
protected:
   int y;
public:
  int z;
};

struct B : private A { };

struct C : private A {
public:
  using A::y;
  using A::z;
};

struct D : private A {
protected:
  using A::y;
  using A::z;
};

struct E : D {
   void f() {
    y = 1;
    z = 2;
  }
};

struct F : A {
public:
  using A::y;
private:
  using A::z;
};

int main() {
  B obj_B;
//  obj_B.y = 3;
//  obj_B.z = 4;
```

```
  C obj_C;
  obj_C.y = 5;
  obj_C.z = 6;

  D obj_D;
//  obj_D.y = 7;
//  obj_D.z = 8;

  F obj_F;
  obj_F.y = 9;
  obj_F.z = 10;
}
```

コンパイラーは、上記の例から、次の割り当てを許可しません。

- `obj_B.y = 3` および `obj_B.z = 4`: メンバー `y` および `z` は、**private** として継承されました。
- `obj_D.y = 7` および `obj_D.z = 8`: メンバー `y` および `z` は、**private** として継承されましたが、それらのアクセスは **protected** に変更されました。

コンパイラーは、上記の例から、次のステートメントを許可します。

- `D::f()` の `y = 1` および `z = 2`: メンバー `y` および `z` は、**private** として継承されましたが、それらのアクセスは **protected** に変更されました。
- `obj_C.y = 5` および `obj_C.z = 6`: メンバー `y` および `z` は **private** として継承されましたが、それらのアクセスは **public** に変更されました。
- `obj_F.y = 9`: メンバー `y` のアクセスは、**protected** から **public** に変更されました。
- `obj_F.z = 10`: メンバー `z` のアクセスは、**public** のままです。 **private** の using 宣言 `using A::z` は、`z` のアクセスに影響を与えません。

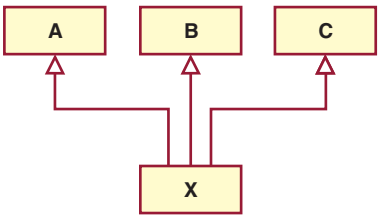
多重継承

C++ 1 つのクラスを複数の基底クラスから派生できます。複数の直接基底クラスから一つのクラスが派生することを、*多重継承* と呼びます。

次の例では、クラス `A`、`B`、および `C` は、派生クラス `X` の直接の基底クラスです。

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

次の*継承グラフ* は、上記の例で示した継承の関係を説明しています。矢印は、矢印の尾部の地点にあるクラスの直接基底クラスを指し示します。

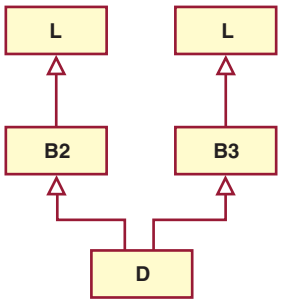


派生の順序は、コンストラクターによるデフォルト初期化およびデストラクターによる終結処理の順序の決定にのみ関係します。

直接の基底クラスは、派生クラスの基底リストに 2 回以上現れることはできません。

```
class B1 { /* ... */ };                    // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

ただし、次の例に示すように、派生クラスは間接の基底クラスを複数回継承することができます。




```
class L { /* ... */ }; // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

上記の例では、クラス D が、クラス B2 を介して、間接基底クラス L を 1 回継承し、クラス B3 を介して 1 回継承します。ただし、クラス L の 2 つのサブオブジェクトが存在し、両方ともクラス D を介してアクセスできるので、あいまいになる可能性があります。これは、修飾されたクラス名を使用して、クラス L を参照することによって避けることができます。次に例を示します。

```
B2::L
```

または、

```
B3::L
```

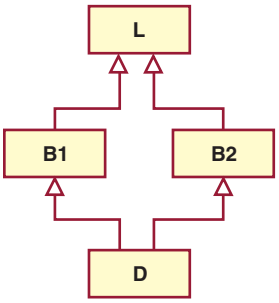
また基底クラスの宣言に基底指定子 **virtual** を使用しても、このようなあいまいさを避けることができます。

仮想基底クラス

C++ 共通の基底クラス A を持つ 2 つの派生クラス B および C があり、さらに B および C から継承した別のクラス D があるとします。基底クラス A を*仮想* として宣言することで、B および C が、同じ A のサブオブジェクトを共用していることを保証できます。

次の例では、クラス D のオブジェクトには、クラス L の 2 つの別個のサブオブジェクトがあり、一方はクラス B1 を介し、もう一方はクラス B2 を介しています。クラス B1 および B2 の*基底リスト* の基底クラス指定子に、キーワード **virtual** を使用することで、クラス B1 およびクラス B2 が共用している、型 L のただ一つのサブオブジェクトが存在することを指示できます。

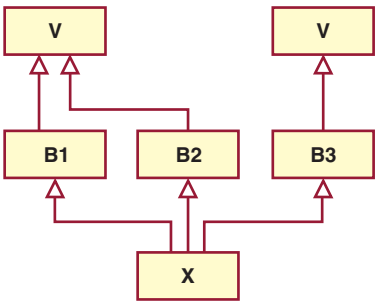
次に例を示します。




```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

この例で、キーワード **virtual** を使用すると、クラス D のオブジェクトが、クラス L のサブオブジェクトを 1 つだけ継承するようにすることができます。

派生クラスが仮想基底クラスと非仮想基底クラスを両方とも持つ場合があります。次に例を示します。




```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class X : public B1, public B2, public B3 { /* ... */
};
```

上記の例では、クラス x にクラス V のサブオブジェクトが 2 個あり、一つはクラス B1 とクラス B2 で共用され、もう一方はクラス B3 を介して共用されます。

マルチアクセス

C++ 仮想基底クラスを含む継承グラフでは、複数のパスを経由して到達できる名前は、最大広範囲のアクセスを提供するパスを介してアクセスされます。

次に例を示します。

```
class L {
public:
  void f();
};

class B1 : private virtual L { };

class B2 : public virtual L { };

class D : public B1, public B2 {
```

```
public:
   void f() {
    // L::f() is accessed through B2
    // and is public
    L::f();
  }
};
```

上記の例では、関数 f() はクラス B2 を介してアクセスされます。クラス B2 は公的に継承され、クラス B1 は私的に継承されているので、クラス B2 の方がより多くのアクセスを提供します。

あいまいな基底クラス

C++ クラスを派生させるとき、基底クラスと派生クラスとに同じ名前のメンバーがあると、あいまいさが生じる可能性があります。固有な関数、またはオブジェクトを参照しない名前または修飾名を使用すると、基底クラス・メンバーへのアクセスがあいまいになります。派生クラス内であいまいな名前のメンバーを宣言してもエラーではありません。あいまいなメンバー名を使用すると、そのあいまいさにエラーとしてフラグを付けます。

例えば、A と B という名前の 2 つのクラスが、両方とも x という名前のメンバーを持ち、C という名前のクラスは、A と B の両方から継承するとします。クラス C から x にアクセスする試みは、あいまいになります。スコープ・レゾリューション (::) 演算子を使用して、そのクラス名でメンバーを修飾することによって、あいまいさを解消することができます。

```
class B1 {
public:
  int i;
  int j;
  void g(int) { }
};

class B2 {
public:
  int j;
  void g() { }
};

class D : public B1, public B2 {
public:
  int i;
};

int main() {
  D dobj;
  D *dptr = &dobj;
  dptr->i = 5;
//  dptr->j = 10;
  dptr->B1::j = 10;
//  dobj.g();
  dobj.B2::g();
}
```

ステートメント dptr->j = 10 は、B1 と B2 の両方に名前 j が現れるので、あいまいになります。B1::g(int) および B2::g() が異なるパラメーターを持っていますが、名前 g が B1 および B2 の両方に現れるので、ステートメント dobj.g() は、あいまいになります。

コンパイラーはコンパイル時にあいまいさを検査します。あいまいさの検査は、アクセス制御や型検査の前に行われるので、同じ名前の複数のメンバーの中の 1 つだけが派生クラスからアクセス可能である場合でも、あいまいさが検出される可能性があります。

名前の隠蔽

A と B という名前の 2 つのサブオブジェクトには、両方とも x という名前のメンバーがあるとします。A が B の基底クラスである場合、サブオブジェクト B のメンバー名 x は、サブオブジェクト A のメンバー名 x を隠します。次の例は、このことを示しています。

```
struct A {
   int x;
};

struct B: A {
   int x;
};

struct C: A, B {
   void f() { x = 0; }
};

int main() {
   C i;
   i.f();
}
```

関数 C::f() の割り当て x = 0 は、宣言 B::x が A::x を隠しているので、あいまいになりません。しかし、コンパイラーは、B を介してサブオブジェクト A にすでにアクセスしているので、コンパイラーは、A からの C の派生が冗長になっていることを警告します。

基底クラス宣言は、継承グラフの 1 つのパスに従っては、隠すことができますが、別のパスでは隠されません。次の例は、このことを示しています。

```
struct A { int x; };
struct B { int y; };
struct C: A, virtual B { };
struct D: A, virtual B {
   int x;
   int y;
};
struct E: C, D { };

int main() {
   E e;
//   e.x = 1;
   e.y = 2;
}
```

割り当て e.x = 1 は、あいまいです。宣言 D::x は、パス D::A::x に従って A::x を隠しますが、パス D::A::x では、A::x を隠しません。したがって、変数 x は、D::x または A::x のいずれかを参照できます。割り当て e.y = 2 は、あいまいではありません。宣言 D::y は、B が仮想基底クラスなので、パス D::B::y および C::B::y の両方で B::y を隠します。

あいまいさと using 宣言

A という名前のクラスから継承している C という名前のクラスがあり、さらに x が A のメンバー名だとします。using 宣言を使用して A::x を C に宣言し、x も C のメンバーであれば、C::x は、A::x を隠しません。したがって、using 宣言は、継承メンバーによるあいまいさを解決することはできません。次の例は、このことを示しています。

```
struct A {
   int x;
};
```

```
struct B: A { };

struct C: A {
   using A::x;
};

struct D: B, C {
   void f() { x = 0; }
};

int main() {
   D i;
   i.f();
}
```

コンパイラーは、割り当てがあいまいなので、関数 D::f() の割り当て x = 0 を許可しません。コンパイラーは、x を 2 つの方法で検索でき、B::x として、または C::x として見つけ出します。

あいまいなクラス・メンバー

コンパイラーは、オブジェクトが持っている型 A のサブオブジェクトの数に関係なく、基底クラス A に定義された、静的メンバー、ネスト型、および列挙子をあいまいさなく検出することができます。次の例は、このことを示しています。

```
struct A {
   int x;
   static int s;
   typedef A* Pointer_A;
   enum { e };
};

int A::s;

struct B: A { };

struct C: A { };

struct D: B, C {
   void f() {
      s = 1;
      Pointer_A pa;
      int i = e;
//      x = 1;
   }
};

int main() {
   D i;
   i.f();
}
```

コンパイラーは、割り当て s = 1、宣言 Pointer_A pa、およびステートメント int i = e を許可します。静的変数 s、typedef Pointer_A、および列挙子 e は、それぞれ 1 つだけあります。コンパイラーは、x がクラス B またはクラス C からアクセスされるので、割り当て x = 1 を許可しません。

ポインター型変換

派生クラスのポインターまたは参照から基底クラスのポインターまたは参照への変換 (明示的または暗黙の) は、同一のアクセス可能基底クラス・オブジェクトを一義的に参照しなければなりません。 (アクセス*可能基底クラス* とは、継承の階層の中で隠蔽されておらず、またあいまいでもない、public に派生された基底クラスです。) 次に例を示します。

多重継承

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
      Z z;
      X* xptr = &z;      // valid
      Y* yptr = &z;      // valid
      W* wptr = &z;       // error, ambiguous reference to class W
                          // X's W or Y's W ?
}
```

仮想基底クラスを使用すれば、あいまいな参照を避けることができます。次に例を示します。

```
class W { /* ... */ };
class X : public virtual W { /* ... */ };
class Y : public virtual W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
      Z z;
      X* xptr = &z;      // valid
      Y* yptr = &z;      // valid
      W* wptr = &z;      // valid, W is virtual therefore only one
                         // W subobject exists
}
```

多重定義解決

多重定義解決は、コンパイラーが任意の関数名をあいまいさなく検出した後で 行われます。次の例は、このことを示しています。

```
struct A {
   int f() { return 1; }
};

struct B {
   int f(int arg) { return arg; }
};

struct C: A, B {
   int g() { return f(); }
};
```

コンパイラーは、名前 `f` が `A` および `B` の両方で宣言されているので、`C::g()` の `f()` の関数呼び出しを許可しません。コンパイラーは、多重定義解決が基底一致 `A::f()` を選択する前に、あいまいさエラーを検出します。

関連参照
- 87 ページの『C++ スコープ・レゾリューション演算子 ::』
- 257 ページの『仮想基底クラス』

仮想関数

C++ 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知できるのは、関数 g() の引数が、A から派生したオブジェクトの参照である可能性があるということです。引数が、型 A のオブジェクトへの参照なのか、または型 B のオブジェクトへのものなのかどうかを判別することはできません。しかし、これは実行時に判別されます。次の例は、A::f() が **virtual** キーワードで宣言されている点を除いては、直前の例と同じです。

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

struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
   arg.f();
}

int main() {
   B x;
   g(x);
}
```

上記の例の出力は、以下のとおりです。

```
Class B
```

virtual キーワードは、参照の型ではなく、参照先のオブジェクト型を使用して、f() のための適切な定義を選択する必要があることをコンパイラーに指示します。

したがって、*仮想関数* とは、別の派生クラスのために再定義できるメンバー関数です。また、たとえオブジェクトの基底クラスに対するポインター、または参照を使用して関数を呼び出したとしても、対応する派生クラスのオブジェクト向けに再定義された仮想関数を、コンパイラーが呼び出せることも保証できます。

仮想関数

仮想関数を宣言するクラス、または継承するクラスは、*ポリモアフィック* と呼ばれます。

仮想メンバー関数は、任意の派生クラスにおいて、どのメンバー関数とも同じように、再定義することができます。クラス A で f という名前の仮想関数を宣言し、A から直接的、または間接的に B という名前のクラスを派生させたとします。 A::f と同じ名前、および同じパラメーター・リストを指定して、クラス B で f という名前の関数を宣言する場合、B::f もまた仮想であり (**virtual** キーワードを使用して B::f を宣言しているかどうかには関係なく)、それは A::f をオーバーライド します。しかし、A::f と B::f のパラメーター・リストが異なり、A::f と B::f は違うものであると見なされる場合、B::f は A::f をオーバーライドしませんし、B::f は、仮想ではありません (**virtual** キーワードを使用してこれを宣言していない場合)。代わりに、B::f は、A::f を隠します。次の例は、このことを示しています。

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

struct A {
   virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
   void f(int) { cout << "Class B" << endl; }
};

struct C: B {
   void f() { cout << "Class C" << endl; }
};

int main() {
   B b; C c;
   A* pa1 = &b;
   A* pa2 = &c;
//   b.f();
   pa1->f();
   pa2->f();
}
```

上記の例の出力は、以下のとおりです。

```
Class A
Class C
```

関数 B::f は、仮想ではありません。これは A::f を隠します。つまり、コンパイラーは、関数呼び出し b.f() を許可しません。関数 C::f は、仮想です。A::f は C で可視ではありませんが、これは A::f をオーバーライドします。

基底クラス・デストラクターを仮想として宣言する場合、派生クラス・デストラクターは、デストラクターが継承されていなくても、基底クラス・デストラクターをオーバーライドします。

オーバーライドする仮想関数の戻りの型は、オーバーライドされる仮想関数の戻りの型とは異なる場合があります。このオーバーライド関数は、*共変仮想関数* と呼ばれます。 B::f が、仮想関数 A::f をオーバーライドするとします。 A::f と B::f の戻りの型は、次の条件のすべてが満たされるかどうかで変わります。

- 関数 B::f は、型 T のクラスに対する参照、またはポインターを戻し、A::f は、T の明確な直接、あるいは間接基底クラスに対するポインター、または参照を戻す。
- B::f が戻すポインター、あるいは参照における const または volatile 修飾は、A::f が戻すポインター、または参照と同等な、あるいはより低い const または volatile 修飾を持っている。
- B::f の戻りの型は、B::f の宣言ポイントで完了する必要がある。または、型 B となる。

次の例は、このことを示しています。

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

struct A { };

class B : private A {
   friend class D;
   friend class F;
};

A global_A;
B global_B;

struct C {
   virtual A* f() {
      cout << "A* C::f()" << endl;
      return &global_A;
   }
};

struct D : C {
   B* f() {
      cout << "B* D::f()" << endl;
      return &global_B;
   }
};

struct E;

struct F : C {

// Error:
//   E is incomplete
//   E* f();
};

struct G : C {

// Error:
//   A is an inaccessible base class of B
//   B* f();
};

int main() {
   D d;
   C* cp = &d;
   D* dp = &d;

   A* ap = cp->f();
   B* bp = dp->f();
};
```

上記の例の出力は、以下のとおりです。

```
B* D::f()
B* D::f()
```

ステートメント `A* ap = cp->f()` は、`D::f()` を呼び出して、戻されるポインターを型 `A*` に変換します。ステートメント `B* bp = dp->f()` は、`D::f()` も呼び出しますが、戻されるポインターを変換しません。戻りの型は `B*` となります。コンパイラーは、`E` が完全なクラスではないので、仮想関数 `F::f()` の宣言を許可しません。コンパイラーは、クラス `A` が `B` にアクセス可能な基底クラスではないので、仮想関数 `G::f()` の宣言を許可しません (フレンド・クラス `D` および `F` と異なり、`B` の定義は、クラス `G` のメンバーにアクセス権を与えません)。

定義によると、仮想関数は、基底クラスのメンバー関数であり、特定のオブジェクトに従って、関数のどのインプリメンテーションを呼び出すかを決めるので、仮想関数をグローバルにも静的にもすることはできません。仮想関数を別のクラスのフレンドとして宣言することができます。

基底クラスにおいて仮想と宣言した関数の場合でも、スコープ・レゾリューション (::) 演算子を使用すれば、それを直接にアクセスすることができます。この場合、仮想関数呼び出しのメカニズムを抑止し、基底クラスで定義された関数インプリメンテーションが使用されます。さらに、派生クラスで仮想メンバー関数を再オーバーライドしなければ、その関数に対する呼び出しでは、基底クラスで定義された関数インプリメンテーションが使用されます。

仮想関数は次のいずれかでなければなりません。
- 定義された
- 宣言された純粋
- 定義され宣言された純粋

1 つまたは複数の純粋仮想メンバー関数を含む基底クラスは、*抽象クラス* と呼ばれます。

あいまいな仮想関数呼び出し

C++ 1 つの仮想関数を、2 つ以上のあいまいな仮想関数でオーバーライドすることはできません。これは、仮想基底クラスから派生した 2 つの非仮想基底から継承する派生クラスで発生する可能性があります。

次に例を示します。

```
class V {
public:
 virtual void f() { }
};

class A : virtual public V {
  void f() { }
};

class B : virtual public V {
 void f() { }
};

// Error:
// Both A::f() and B::f() try to override V::f()
class D : public A, public B { };

int main() {
  D d;
  V* vptr = &d;

  // which f(), A::f() or B::f()?
 vptr->f();
}
```

コンパイラーは、クラス D の定義を許可しません。クラス A では、A::f() のみが V::f() をオーバーライドします。同様にクラス B でも、B::f() のみが V::f() をオーバーライドします。ただし、クラス D では、A::f() と B::f() の両方が、V::f() をオーバーライドしようとします。上記の例で示すように、D オブジェクトが、クラス V を指すポインターを使用して参照される場合、コンパイラーは、どちらの関数を呼び出すべきかを決めることができないので、このような試みは許されません。 1 つの関数のみが仮想関数をオーバーライドできます。

同じクラス型の別個のインスタンスがあることによって、仮想関数のあいまいなオーバーライドが起きた場合、特殊なケースになります。次の例では、クラス D には、クラス A の 2 つの別々のサブオブジェクトがあります。

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

struct A {
   virtual void f() { cout << "A::f()" << endl; };
};

struct B : A {
   void f() { cout << "B::f()" << endl;};
};

struct C : A {
   void f() { cout << "C::f()" << endl;};
};

struct D : B, C { };

int main() {
   D d;

   B* bp = &d;
   A* ap = bp;
   D* dp = &d;

   ap->f();
//   dp->f();
}
```

クラス D には、クラス A のオカレンスが 2 つあり、一つは B から継承されたもので、もう一つは C から継承されたものです。したがって、仮想関数 A::f にも 2 つのオカレンスがあります。ステートメント ap->f() は、D::B::f を呼び出します。しかし、コンパイラーは、D::B::f または D::C::f のどちらも呼び出すことができるので、ステートメント dp->f() を許可しません。

仮想関数のアクセス

C++ 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

C++ *抽象クラス* とは、特に基底クラスとして使用するように設計されたクラスです。抽象クラスには、少なくとも 1 つの*純粋仮想関数* が含まれています。クラス宣言の中の仮想メンバー関数の宣言で、*純粋指定子* (= 0) を使用することによって、純粋仮想関数を宣言することができます。

次に抽象クラスの例を示します。

```
class AB {
public:
  virtual void f() = 0;
};
```

関数 AB::f は、純粋仮想関数です。関数宣言に、純粋指定子と定義の両方を入れることはできません。例えば、コンパイラーは、次の表記を許可しません。

```
struct A {
  virtual void g() { } = 0;
};
```

抽象クラスをパラメーター型、関数からの戻りの型、または明示型変換の型として使用することはできませんし、抽象クラスのオブジェクトも宣言することはできません。ただし、抽象クラスを指すポインター、および参照を宣言することは可能です。次の例は、このことを示しています。

```
struct A {
  virtual void f() = 0;
};

struct B : A {
  virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

// Error:
// Class A is an abstract class
//   A a;
```

```
   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

クラス A は、抽象クラスです。コンパイラーは、関数宣言 A g() または void h(A)、オブジェクト a の宣言、そして b の型 A への静的キャストも許可しません。

仮想メンバー関数は、継承されます。派生クラスにある各純粋仮想関数をオーバーライドしない限り、抽象基底クラスから派生するクラスも抽象になります。

次に例を示します。

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
  D2 d;
}
```

コンパイラーは、オブジェクト d の宣言を許可しません。 D2 が、AB から純粋仮想関数 f() を継承した抽象クラスだからです。関数 D2::g() を定義すれば、コンパイラーは、オブジェクト d の宣言を行うことができます。

非抽象クラスから抽象クラスを派生させたり、非純粋仮想関数を純粋仮想関数でオーバーライドできることに注意してください。

抽象クラスのコンストラクターまたはデストラクターから、メンバー関数を呼び出すことができます。ただし、コンストラクターから純粋仮想関数を呼び出した (直接または間接) 結果は、未定義です。次の例は、このことを示しています。

```
struct A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

A のデフォルトのコンストラクターは、直接的、および間接的 (indirect() を介して) の両方で、純粋仮想関数 direct() を呼び出します。

第 15 章 特殊なメンバー関数

C++ デフォルトのコンストラクター、デストラクター、コピー・コンストラクター、およびコピー代入演算子は、*特殊なメンバー関数* です。これらの関数は、クラス・オブジェクトを、作成、破棄、変換、初期化およびコピーします。

コンストラクターとデストラクターの概要

C++ データや関数を含むクラスの内部構造は複雑なので、オブジェクトの初期化やクラス終結処理は、単純なデータ構造の場合より格段に複雑です。コンストラクターやデストラクターは、クラス・オブジェクトの構成や破棄に使用されるクラスの特殊なメンバー関数です。構成では、オブジェクトのストレージ割り当てや初期化もあわせて行われる場合があります。破棄に際しては、オブジェクトのストレージの終結処理や割り当て解除も行われる場合があります。

他のメンバー関数と同様、コンストラクターとデストラクターは、クラス宣言の中で宣言されます。これらは、インラインまたはクラス宣言の外で定義することができます。コンストラクターは、デフォルトの引数を持つことができます。コンストラクターは、他のメンバー関数と異なり、メンバー初期化リストを持つことができます。コンストラクターとデストラクターには、次の制約事項が適用されます。

- コンストラクターとデストラクターには戻りの型がなく、また値を戻すこともできません。
- 参照とポインターは、コンストラクターとデストラクターには、そのアドレスを取得できないので、使用することはできません。
- コンストラクターは、**virtual** というキーワードでは、宣言できません。
- コンストラクターおよびデストラクターは **static**、**const**、または **volatile** として宣言することができません。
- 共用体には、コンストラクターやデストラクターのあるクラス・オブジェクトを入れることができません。

コンストラクターとデストラクターは、メンバー関数と同じアクセス規則に従います。例えば、protected を使用してコンストラクターを宣言した場合、それを使用してクラス・オブジェクトを使用できるのは、派生クラスとフレンドだけです。

コンパイラーは、クラス・オブジェクトを定義するときはコンストラクターを、クラス・オブジェクトがスコープ外に出るときはデストラクターを、それぞれ自動的に呼び出します。コンストラクターは、その **this** ポインターが参照するクラス・オブジェクトに、ストレージを割り振ることはありませんが、そのクラス・オブジェクトが参照するオブジェクトより多くのオブジェクトに、ストレージを割り振る場合があります。オブジェクトのためにストレージ割り当てが必要な場合、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ませんが、基底クラスのコンストラクターやデストラクターを呼び出します。デストラクターは、**virtual** というキーワードで宣言できます。

コンストラクターは、ローカルまたは一時クラス・オブジェクトが作成されるときにも呼び出され、デストラクターは、ローカルまたは一時オブジェクトがスコープを超えたときに呼び出されます。

コンストラクターまたはデストラクターから、メンバー関数を呼び出すことができます。クラス A のコンストラクター、またはデストラクターから、直接的に、あるいは間接的に仮想関数を呼び出すことができます。この場合、呼び出される関数は A で定義されているものか、A の基底クラスであり、A から派生したクラスでオーバーライドされた関数ではありません。これにより、コンストラクター、またはデストラクターから、非構成オブジェクトにアクセスする可能性を回避します。次の例は、このことを示しています。

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

struct A {
  virtual void f() { cout << "void A::f()" << endl; }
  virtual void g() { cout << "void A::g()" << endl; }
  virtual void h() { cout << "void A::h()" << endl; }
};

struct B : A {
  virtual void f() { cout << "void B::f()" << endl; }
  B() {
    f();
    g();
    h();
  }
};

struct C : B {
  virtual void f() { cout << "void C::f()" << endl; }
  virtual void g() { cout << "void C::g()" << endl; }
  virtual void h() { cout << "void C::h()" << endl; }
};

int main() {
  C obj;
}
```

次に、上記の例の出力を示します。

```
void B::f()
void A::g()
void A::h()
```

例では obj いう名前で型 C のオブジェクトを作成しますが、B のコンストラクターは、C が B から派生しているので、C でオーバーライドされた関数は、どれも呼び出しません。

コンストラクター、またはデストラクターで **typeid** または **dynamic_cast** 演算子を使用でき、同様に、コンストラクターのメンバー初期化指定子も使用できます。

関連参照
- 103 ページの『C++ の new 演算子』
- 107 ページの『C++ の delete 演算子』
- 283 ページの『フリー・ストレージ』

コンストラクター

C++ コンストラクター は、そのクラスと同じ名前を持つメンバー関数です。次に例を示します。

```
class X {
public:
  X();      // constructor for class X
};
```

コンストラクターは、そのクラス型のオブジェクトの作成に使用され、オブジェクトを初期化できます。

コンストラクターは、**virtual** または **static** として宣言できませんし、**const**、**volatile**、または **const volatile** として宣言することもできません。

コンストラクターの戻りの型は、指定しません。コンストラクター本体にあるリターン・ステートメントは、戻り値を持てません。

関連参照

- 283 ページの『フリー・ストレージ』

デフォルト・コンストラクター

C++ デフォルト・コンストラクター とは、パラメーターがないか、ある場合でも、すべての パラメーターにデフォルト値があるコンストラクターです。

クラス A にユーザー定義のコンストラクターが必要であるが、それが存在しない場合、コンパイラーは、コンストラクター A::A() を暗黙的に宣言します。このコンストラクターは、そのクラスのインライン・パブリック・メンバーです。コンパイラーが、コンストラクターを使用して、型 A のオブジェクトを作成する時に、コンパイラーは、A::A() を暗黙的に定義 します。コンストラクターは、コンストラクター初期化指定子もヌル・ボディも持つようにはなりません。

コンパイラーは、まず最初に暗黙的に宣言された基底クラスのコンストラクターと、クラス A の非静的データ・メンバーを暗黙的に定義してから、暗黙的に宣言された A のコンストラクターを定義します。定数や参照型メンバーを持つクラスに対して、デフォルトのコンストラクターは作成されません。

クラス A のコンストラクターは、次のことがすべて真であれば、*単純* です。
- 暗黙的に定義される
- A に仮想関数がなく、仮想基底クラスもない
- A の直接基底クラスが、すべて単純コンストラクターを持っている
- A のすべての非静的データ・メンバーに関するクラスが、単純コンストラクターを持っている

上記のいずれかが偽であれば、コンストラクターは、*非単純* です。

共用体メンバーは、非単純コンストラクターを持つクラス型にはできません。

すべての関数と同様、コンストラクターは、デフォルト引数を持つことができます。これらは、メンバー・オブジェクトの初期化に使用されます。デフォルト値が提供される場合、末尾引数は、コンストラクターの式リストで省略できます。コンストラクターにデフォルト値を持たない引数がある場合、それはデフォルト・コンストラクターではないことに注意してください。

クラス A の*コピー・コンストラクター* とは、その第 1 パラメーターが、型 A&、const A&、volatile A&、または const volatile A& のいずれかであるコンストラクターです。コピー・コンストラクターは、あるクラス・オブジェクトを、同じクラス型の別のクラス・オブジェクトからコピーするために使用されます。そのクラスと同じタイプの引数でコピー・コンストラクターを使用することはできません。参照を使用する必要があります。すべてがデフォルト引数である限り、追加デフォルト引数を持つコピー・コンストラクターを提供することはできます。あるクラスにユーザー定義のコンストラクターが必要であるにもかかわらず、それが存在しない場合、コンパイラーは、そのクラス用に、パブリック・アクセスを持つコピー・コンストラクターを作成します。コンパイラーは、メンバーまたは基底クラスにアクセス不能なコピー・コンストラクターがあるクラスに対しては、コピー・コンストラクターを作成しません。

次のコードは、デフォルト・コンストラクターとコピー・コンストラクターがある 2 つのクラスを示しています。

```
class X {
public:

  // default constructor, no arguments
  X();

  // constructor
  X(int, int , int = 0);

  // copy constructor
  X(const X&);

  // error, incorrect argument type
  X(X);
};

class Y {
public:

  // default constructor with one
  // default argument
  Y( int = 0);

  // default argument
  // copy constructor
  Y(const Y&, int = 0);
};
```

関連参照
- 292 ページの『コピー・コンストラクター』

コンストラクターでの明示的初期化

C++ 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。

コンストラクター、仮想関数、private メンバーまたは protected メンバー、および基底クラスのいずれも持たないクラス・オブジェクトは、*集合体* と呼ばれます。 集合体の例としては、C 形式の構造体および共用体があります。

クラス・オブジェクトを作成する場合、そのオブジェクトを明示的に初期化します。クラス・オブジェクトを初期化するには、次の 2 つの方法があります。
- 括弧で囲んだ式のリストの使用。コンパイラーは、このリストをコンストラクターの引数リストとして使用し、クラスのコンストラクターを呼び出します。
- 単一初期化値、および = 演算子の使用。このような型の式は、代入でなく初期化なので、代入演算子関数 (これが存在する場合でも) は、呼び出されません。単一引数の型は、コンストラクターに対する最初の引数の型と一致していなければなりません。コンストラクターに残りの引数がある場合、これらの引数はデフォルト値を持っている必要があります。
-

コンストラクターを用いてクラス・オブジェクトを明示的に初期化する初期化指定子の構文を次に示します。

次の例は、クラス・オブジェクトを明示的に初期化するいくつかのコンストラクターの宣言および使用の方法を示します。

```
// This example illustrates explicit initialization
// by constructor.
#include <iostream>
using namespace std;

class complx {
  double re, im;
public:

  // default constructor
  complx() : re(0), im(0) { }

  // copy constructor
  complx(const complx& c) { re = c.re; im = c.im; }

  // constructor with default trailing argument
  complx( double r, double i = 0.0) { re = r; im = i; }

  void display() {
    cout << "re = "<< re << " im = " << im << endl;
  }
};

int main() {

  // initialize with complx(double, double)
  complx one(1);

  // initialize with a copy of one
  // using complx::complx(const complx&)
  complx two = one;

  // construct complx(3,4)
  // directly into three
  complx three = complx(3,4);

  // initialize with default constructor
  complx four;

  // complx(double, double) and construct
  // directly into five
  complx five = 5;

  one.display();
  two.display();
  three.display();
  four.display();
  five.display();
}
```

上記の例で作成される出力は次のようになります。

```
re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0
```

関連参照
- 66 ページの『初期化指定子』

規定クラスおよびメンバーの初期化

C++ コンストラクターは、次に示す 2 とおりの異なった方法でメンバーを初期化できます。コンストラクターは渡された引数を使用して、コンストラクター定義内のメンバー変数を初期化することができます。

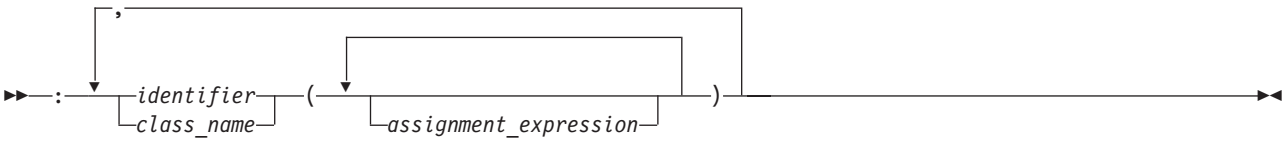
```
complx(double r, double i = 0.0) { re = r; im = i; }
```

またはコンストラクターは、定義の中に*初期化指定子リスト* を含めることができますが、それらは、関数本体の前に置く必要があります。

```
complx(double r, double i = 0) : re(r), im(i) { /* ... */ }
```

どちらの方法でも、引数値は適切なクラスのデータ・メンバーに代入されます。

コンストラクター初期化指定子リストの構文を次に示します。



コンストラクター宣言の一部ではなく、関数定義の一部として初期化リストをインクルードします。次に例を示します。

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

class B1 {
  int b;
public:
  B1() { cout << "B1::B1()" << endl; };

  // inline constructor
  B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};
class B2 {
  int b;
protected:
  B2() { cout << "B1::B1()" << endl; }

  // noninline constructor
  B2(int i);
};

// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {
  int d1, d2;
public:
```

```
  D(int i, int j) : B1(i+1), B2(), d1(i) {
    cout << "D1::D1(int, int)" << endl;
    d2 = j;}
};

int main() {
  D obj(1, 2);
}
```

次に、上記の例の出力を示します。

```
B1::B1(int)
B1::B1()
D1::D1(int, int)
```

コンストラクターのある基底クラスまたはメンバーをコンストラクターを呼び出すことによって、明示的に初期化するのでない場合、コンパイラーは、自動的にデフォルト・コンストラクターのある基底クラスまたはメンバーを初期化します。上記の例では、クラス D のコンストラクター内の呼び出し B2() を除外すると (後に示すように)、空の式リストのあるコンストラクター初期化指定子が自動的に作成されて、B2 を初期化します。クラス D のコンストラクター (上記と下記に示されています) は、クラス D のオブジェクトと同じ構造体になります。

```
class D : public B1, public B2 {
  int d1, d2;
public:

  // call B2() generated by compiler
  D(int i, int j) : B1(i+1), d1(i) {
    cout << "D1::D1(int, int)" << endl;
    d2 = j;}
};
```

上記の例では、コンパイラーは、B2() のデフォルトのコンストラクターを自動的に呼び出します。

派生クラスがコンストラクターを呼び出すことができるようにするには、コンストラクターを public または protected 付きで宣言する必要があります。次に例を示します。

```
class B {
  B() { }
};

class D : public B {

  // error: implicit call to private B() not allowed
  D() { }
};
```

コンパイラーは、コンストラクターが private コンストラクター B::B() にアクセスできないので、D::D() の定義を許可しません。

初期化指定子リストを指定して、次のことを初期化する必要があります。それらは、デフォルト・コンストラクターのない基底クラス、参照データ・メンバー、非静的 const データ・メンバー、または定数データ・メンバーを含むクラス・タイプです。次の例は、このことを示しています。

```
class A {
public:
  A(int) { }
};

class B : public A {
  static const int i;
  const int j;
  int &k;
```

```
public:
  B(int& arg) : A(0), j(1), k(arg) { }
};

int main() {
  int x = 0;
  B obj(x);
};
```

データ・メンバー j および k、さらに基底クラス A は、B のコンストラクターの初期化指定子リストで初期化される必要があります。

クラスのメンバーを初期化する際、データ・メンバーを使用できます。次の例は、このことを示しています。

```
struct A {
  int k;
  A(int i) : k(i) { }
};
struct B: A {
  int x;
  int i;
  int j;
  int& r;
  B(int i): r(x), A(i), j(this->i), i(i) { }
};
```

コンストラクター B(int i) は、次のことを初期化します。

- B::x を参照するための B::r
- B(int i) への引数の値を指定したクラス A
- B::i の値を指定した B::j
- B(int i) への引数の値を指定した B::i

クラスのメンバーを初期化する場合、メンバー関数 (仮想メンバー関数を含む) を呼び出したり、あるいは演算子 **typeid** または **dynamic_cast** を使用することもできます。しかし、すべての基底クラスが初期化される前に、メンバー初期化リストにあるこれらの演算を実行する場合、その振る舞いは予期できません。次の例は、このことを示しています。

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

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(f()), j(1234) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

上記の例の出力は、次の出力に類似しています。

```
Value of i: 8
Value of j: 1234
```

B のコンストラクターの初期化指定子 A(f()) の振る舞いは、未定義です。ランタイムは、B::f() を呼び出し、基底 A が初期化されていなくても、A::i にアクセスしようとします。

次の例は、B::B() の初期化指定子が異なる引数を持つ点を除いては、直前の例と同じです。

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

struct A {
  int i;
  A(int arg) : i(arg) {
    cout << "Value of i: " << i << endl;
  }
};

struct B : A {
  int j;
  int f() { return i; }
  B();
};

B::B() : A(5678), j(f()) {
   cout << "Value of j: " << j << endl;
}

int main() {
  B obj;
}
```

次に、上記の例の出力を示します。

```
Value of i: 5678
Value of j: 5678
```

B のコンストラクターでは、初期化指定子 j(f()) の振る舞いは、明確に定義されています。 B::j が初期化される時、基底クラス A も初期化済みです。

関連参照

- 273 ページの『デフォルト・コンストラクター』
- 92 ページの『typeid 演算子』
- 96 ページの『dynamic_cast 演算子』

派生クラス・オブジェクトの構築順序

C++ コンストラクターを使用して派生クラス・オブジェクトを作成する場合、そのオブジェクトは、次の順番で作成されます。

1. 基底リストに示されている順序にしたがって、仮想基底クラスが初期化される。
2. 宣言に示されている順序にしたがって、非仮想基底クラスが初期化される。
3. クラス・メンバーは、宣言の順番に (初期化リストでの順番には関係なく) 初期化される。
4. コンストラクターの本体が、実行される。

次の例は、このことを示しています。

```
#include <iostream>
using namespace std;
struct V {
  V() { cout << "V()" << endl; }
};
struct V2 {
  V2() { cout << "V2()" << endl; }
};
struct A {
  A() { cout << "A()" << endl; }
};
struct B : virtual V {
  B() { cout << "B()" << endl; }
};
struct C : B, virtual V2 {
  C() { cout << "C()" << endl; }
};
struct D : C, virtual V {
  A obj_A;
  D() { cout << "D()" << endl; }
};
int main() {
  D c;
}
```

次に、上記の例の出力を示します。

```
V()
V2()
B()
C()
A()
D()
```

上記の出力は、型 D のオブジェクトを作成するために、C++ ランタイムがコンストラクターを呼び出す順序をリストしています。

関連参照

- 257 ページの『仮想基底クラス』

デストラクター

C++ デストラクター は、オブジェクトを破棄するときに、ストレージの割り当て解除、およびクラス・オブジェクトとそのクラス・メンバーに関するその他の終結処理を行うために使用されます。オブジェクトがスコープを超えるか、明示的にオブジェクトを削除するときに、そのクラス・オブジェクトのデストラクターを呼び出します。

デストラクターは、そのクラスと同じ名前を持つメンバー関数で、接頭部として ~ (波形記号) が付きます。次に例を示します。

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

デストラクターは、引数を使用せず、戻りの型もありません。そのアドレスを取得することはできません。デストラクターを **const**、**volatile**、**const volatile**、または **static** で宣言できません。デストラクターは、**virtual**、または純粋 **virtual** で宣言できます。

あるクラスにユーザー定義のデストラクターが必要であるが、それが存在しない場合、コンパイラーは、デストラクターを暗黙的に宣言します。この暗黙的に宣言されたデストラクターは、そのクラスのインライン・パブリック・メンバーです。

コンパイラーがデストラクターを使用して、デストラクターのクラス型のオブジェクトを破棄する場合、コンパイラーは、暗黙的に宣言されたデストラクターを暗黙的に定義します。クラス A が、暗黙的に宣言されたデストラクターを持っているとします。次は、コンパイラーが A に対して暗黙的に定義を行う関数と同等です。

```
~A::A() { }
```

コンパイラーは、まず最初に暗黙的に宣言された基底クラスのデストラクターと、クラス A の非静的データ・メンバーを暗黙的に定義してから、暗黙的に宣言された A のデストラクターを定義します。

クラス A のデストラクターは、次のことがすべて真であれば*単純* です。
- 暗黙的に定義される
- A の直接基底クラスは、すべて単純デストラクターを持っている
- A のすべての非静的データ・メンバーに関するクラスが、単純デストラクターを持っている

上記のいずれかが偽であれば、デストラクターは*非単純* です。

共用体メンバーは、非単純デストラクターを持つクラス型にはできません。

クラス型であるクラス・メンバーは、独自のデストラクターを持つことができます。基底クラスと派生クラスは、両方ともデストラクターを持つことができますが、デストラクターは継承されません。基底クラス A または A のメンバーがデストラクターを持っており、A から派生したクラスがデストラクターを宣言しない場合、デフォルトのデストラクターが生成されます。

デフォルト・デストラクターは、基底クラスおよび派生クラスのメンバーのデストラクターを呼び出します。

基底クラスおよびメンバーのデストラクターは、それらのコンストラクターが完了する順番の逆順で呼び出されます。

1. クラス・オブジェクト用のデストラクターは、メンバーおよび基底用のデストラクターより前に呼び出されます。
2. 非静的基底クラスのデストラクターは、仮想基底クラスのデストラクターより前に、呼び出されます。
3. 非仮想基底クラスのデストラクターは、仮想基底クラスのデストラクターより前に、呼び出されます。

デストラクターを持つクラス・オブジェクトの例外をスロー (throw) すると、プログラムが catch ブロックの外に制御を渡すまで、スローする一時オブジェクトのデストラクターを呼び出しません。

自動オブジェクト (**auto** または **register** を宣言されたローカル・オブジェクト、あるいは **static** または **extern** として宣言されていないローカル・オブジェクト) または一時オブジェクトがスコープ外に渡されると、デストラクターが暗黙的に呼び出されます。構築された外部オブジェクトや静的オブジェクトのプログラム終了処理時にも、暗黙的にデストラクターを呼び出します。デストラクターは、**new** 演算子によって作成されたオブジェクトに対してユーザーが **delete** 演算子を使用すると呼び出されます。

次に例を示します。

```
#include <string>

class Y {
private:
```

```
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
  // Destructor
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // ...

  // Destructor ~Y is called before
  // control returns from main()
}
```

デストラクターを明示的に使用してオブジェクトを破棄することもできますが、この方法はお勧めできません。しかし、配置 **new** 演算子を使用して作成されたオブジェクトを破棄するために、オブジェクトのデストラクターを明示的に呼び出すことができます。次の例は、このことを示しています。

```
#include <new>
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
  char* p = new char[sizeof(A)];
  A* ap = new (p) A;
  ap->A::~A();
  delete [] p;
}
```

ステートメント `A* ap = new (p) A` は、型 `A` の 新規のオブジェクトを、フリー・ストレージにではなく、`p` が割り振ったメモリーに動的に作成します。ステートメント `delete [] p` は、`p` が割り振ったストレージを削除します。しかし、ランタイムは、`A` のデストラクターを明示的に呼び出すまでは (ステートメント `ap->A::~A()` と指定して)、`ap` が指すオブジェクトはまだ存在していると判断しています。

非クラス型は、*疑似デストラクター* を持っています。次の例は、整数型の疑似デストラクターを呼び出します。

```
typedef int I;
int main() {
  I x = 10;
  x.I::~I();
  x = 20;
}
```

疑似デストラクターの呼び出し `x.I::~I()` は、まったく効果はありません。オブジェクト `x` は、破棄されておらず、代入 `x = 20` はまだ有効です。疑似デストラクターは、非クラス型を有効にするためにデストラ

クターを明示的に呼び出すための構文を必要とするので、任意の型に対するデストラクターが存在するかどうかを把握しなくても、コードの書き込みができます。

関連参照
- 287 ページの『一時オブジェクト』

フリー・ストレージ

C++ フリー・ストレージ は、プログラムの実行中に、オブジェクトのストレージを割り振り (および割り振り解除) できるメモリーのプールです。 **new** 演算子および **delete** 演算子は、それぞれ、フリー・ストレージの割り当てと割り当て解除に使用されます。

ユーザー専用のクラス用の **new** と **delete** を多重定義して、独自バージョンを定義することができます。 **new** 演算子と **delete** 演算子に追加オペレーターを宣言することができます。 **new** 演算子と **delete** 演算子がクラス・オブジェクトに使用されると、クラス・メンバー演算子関数の **new** と **delete** が、宣言してあれば、呼び出されます。

クラス・オブジェクトを **new** 演算子で作成する場合、オブジェクト作成のために、演算子関数の **operator new()** または **operator new[]()** (宣言済みの場合) のいずれかが呼び出されます。クラスの **operator new()** や **operator new[]()** は、キーワード **static** なしで宣言されている場合でも、常に静的クラス・メンバーです。この戻りの型は **void*** です。その最初のパラメーターは、オブジェクト型のサイズで、**std::size_t** 型でなければなりません。これを virtual にすることは、できません。

型 **std::size_t** は、インプリメンテーション依存の符号なし整数型で、標準ライブラリー・ヘッダー `<cstddef>` に定義されています。 **new** 演算子を多重定義するときは、戻り型が **void*** で、最初のパラメーターが **std::size_t** のクラス・メンバーとして宣言する必要があります。 **operator new()** または **operator new[]()** の宣言で、追加パラメーターを宣言できます。割り当て式の中で、これらのパラメーターの値を指定する配置構文を使用します。

次の例は、2 つの operator **new** 関数を多重定義します。
- `X::operator new(size_t sz)`: これは、`malloc()` が失敗した場合に、C 関数 `malloc()` でメモリーを割り振り、ストリングをスロー (`std::bad_alloc` の代わりに) することによって、デフォルトの **new** 演算子を多重定義します。
- `X::operator new(size_t sz, int location)`: この関数は、追加の整数パラメーター、`location` を取ります。この関数は、`X` オブジェクト 3 つまでに対してストレージを管理する、「メモリー・マネージャー」を最も単純化したものをインプリメントします。

  静的配列 `X::buffer` は、3 つの `Node` オブジェクトを保持します。各 `Node` オブジェクトは、`data` という名前の `X` オブジェクトを指すポインター、および `filled` という名前のブール変数を含んでいます。各 `X` オブジェクトは、number と呼ぶ整数を保管します。

  この **new** 演算子を使用する場合、引数 `location` を渡して、新規の `X` オブジェクトを「作成」したい `buffer` の配列ロケーションを示します。配列ロケーションが「filled」(`filled` のデータ・メンバーは、配列ロケーションにおいては `false` と同じ) でなければ、**new** 演算子は、`buffer[location]` に配置された `X` オブジェクトを指すポインターを戻します。

```
#include <new>
#include <iostream>

using namespace std;

class X;
```

```
struct Node {
  X* data;
  bool filled;
  Node() : filled(false) { }
};

class X {
  static Node buffer[];

public:

  int number;

  enum { size = 3};

  void* operator new(size_t sz) throw (const char*) {
    void* p = malloc(sz);
    if (sz == 0) throw "Error: malloc() failed";
    cout << "X::operator new(size_t)" << endl;
    return p;
  }

  void *operator new(size_t sz, int location) throw (const char*) {
    cout << "X::operator new(size_t, " << location << ")" << endl;
    void* p = 0;
    if (location < 0 || location >= size || buffer[location].filled == true) {
      throw "Error: buffer location occupied";
    }
    else {
    p = malloc(sizeof(X));
   if (p == 0) throw "Error: Creating X object failed";
   buffer[location].filled = true;
      buffer[location].data = (X*) p;
    }
    return p;
  }

  static void printbuffer() {
    for (int i = 0; i < size; i++) {
      cout << buffer[i].data->number << endl;
    }
  }

};

Node X::buffer[size];

int main() {
  try {
    X* ptr1 = new X;
    X* ptr2 = new(0) X;
    X* ptr3 = new(1) X;
    X* ptr4 = new(2) X;
    ptr2->number = 10000;
    ptr3->number = 10001;
    ptr4->number = 10002;
    X::printbuffer();
    X* ptr5 = new(0) X;
  }
  catch (const char* message) {
    cout << message << endl;
  }
}
```

次に、上記の例の出力を示します。

```
X::operator new(size_t)
X::operator new(size_t, 0)
X::operator new(size_t, 1)
X::operator new(size_t, 2)
10000
10001
10002
X::operator new(size_t, 0)
Error: buffer location occupied
```

ステートメント X* ptr1 = new X は、X::operator new(sizeof(X)) を呼び出します。ステートメント X* ptr2 = new(0) X は、X::operator new(sizeof(X),0) を呼び出します。

delete 演算子は、**new** 演算子によって作成されたオブジェクトを破棄します。**delete** のオペランドは **new** によって戻されたポインターでなければなりません。 デストラクターを持つオブジェクトに対して **delete** を呼び出すと、オブジェクトの割り当て解除を行う前に、デストラクターが呼び出されます。

delete 演算子によってクラス・オブジェクトを破棄する場合、演算子関数の **operator delete()** または **operator delete[]()** (これが定義されている場合) が呼び出され、これによってオブジェクトが破棄されます。クラスの **operator delete()** や **operator delete[]()** は、キーワード **static** なしで宣言されている場合でも、常に静的メンバーです。この最初のパラメーターの型は **void*** でなければなりません。 **operator delete()** および **operator delete[]()** の戻りの型は **void** なので、これらは値を戻すことはできません。

次の例では、演算子関数 **operator new()** および **operator delete()** の宣言と使用方法について説明します。

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

class X {
public:
  void* operator new(size_t sz) throw (const char*) {
    void* p = malloc(sz);
    if (p == 0) throw "malloc() failed";
    return p;
  }

  // single argument
  void operator delete(void* p) {
 cout << "X::operator delete(void*)" << endl;
    free(p);
  }

};

class Y {
  int filler[100];
public:

  // two arguments
  void operator delete(void* p, size_t sz) throw (const char*) {
    cout << "Freeing " << sz << " byte(s)" << endl;
    free(p);
  };

};

int main() {
  X* ptr = new X;

  // call X::operator delete(void*)
  delete ptr;
```

```
  Y* yptr = new Y;

  // call Y::operator delete(void*, size_t)
  // with size of Y as second argument
  delete yptr;
}
```

上記の例は、次の式と同じ出力を生成します。

```
X::operator delete(void*)
Freeing 400 byte(s)
```

ステートメント `delete ptr` は、`X::operator delete(void*)` を呼び出します。ステートメント delete yptr は、`Y::operator delete(void*, size_t)` を呼び出します。

削除されたオブジェクトにアクセスしようとした場合の結果は、削除後にオブジェクトの値が変化する可能性があるので、未定義です。

演算子関数 **new** および **delete** を宣言していないクラス・オブジェクト、または非クラス・オブジェクトに対して、**new** および **delete** を呼び出す場合、グローバル演算子 **new** および **delete** が使用されます。グローバル演算子 **new** および **delete** は、C++ ライブラリーに用意されています。

クラス・オブジェクトの配列の割り当ておよび割り当て解除用の C++ 演算子は、**operator new[ ]()** および **operator delete[ ]()** です。

delete 演算子を仮想として宣言できません。ただし、基底クラスのデストラクターを仮想として宣言することで、**delete** 演算子に、ポリモアフィックの振る舞いを追加できます。次の例は、このことを示しています。

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

struct A {
  virtual ~A() { cout << "~A()" << endl; };
  void operator delete(void* p) {
    cout << "A::operator delete" << endl;
    free(p);
  }
};

struct B : A {
  void operator delete(void* p) {
    cout << "B::operator delete" << endl;
    free(p);
  }
};

int main() {
  A* ap = new B;
  delete ap;
}
```

次に、上記の例の出力を示します。

```
~A()
B::operator delete
```

ステートメント `delete ap` は、`A` のデストラクターが仮想として宣言されているので、クラス `A` の代わりに、クラス `B` から **delete** 演算子を使用します。

delete 演算子からポリモアフィックの振る舞いを獲得できますが、静的に可視である **delete** 演算子は、別の **delete** 演算子が呼び出されることがあっても、アクセス可能にしておく必要があります。例えば、上記の例で、代わりに `B::operator delete(void*)` が呼び出されても、関数 `A::operator delete(void*)` をアクセス可能にしておく必要があります。

仮想デストラクターは、配列 (**operator delete[]()**) に対するどの割り振り解除演算子にも影響を与えません。次の例は、このことを示しています。

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

struct A {
  virtual ~A() { cout << "~A()" << endl; }
  void operator delete[](void* p, size_t) {
    cout << "A::operator delete[]" << endl;
    ::delete [] p;
  }
};

struct B : A {
  void operator delete[](void* p, size_t) {
    cout << "B::operator delete[]" << endl;
    ::delete [] p;
  }
};

int main() {
  A* bp = new B[3];
  delete[] bp;
};
```

ステートメント `delete[] bp` の振る舞いは、未定義です。

delete 演算子を多重定義するときは、前述のように、戻り型が **void** で、最初のパラメーターの型が **void*** のクラス・メンバーとして宣言する必要があります。宣言に型 **size_t** の 2 番目のパラメーターを追加することができます。単一クラスには、**operator delete()** または **operator delete[]()** を 1 つしか持つことができません。

関連参照
- 103 ページの『C++ の new 演算子』
- 107 ページの『C++ の delete 演算子』
- 104 ページの配置構文
- 153 ページの『割り当ておよび割り当て解除関数』

一時オブジェクト

C++ コンパイラーによる一時オブジェクトの作成が必要になる場合があります。コンパイラーがこれらのオブジェクトを使用するのは、参照を初期化するときや、標準型変換、引数の受け渡し、関数からの戻り、および **throw** 式の評価が含まれている式を評価するときです。

コンパイラーが参照変数の初期化のために一時オブジェクトを作成する場合、一時オブジェクトの名前は、参照変数の名前と同じスコープを持ちます。一時オブジェクトが、完全式 (別の式の副次式ではない式) の評価中に作成される場合、一時オブジェクトは、それが作成されたポイントを字句として含んでいる評価の最終ステップとして破棄されます。

完全式の破棄には、例外が 2 つあります。

- 式は、オブジェクトを定義する宣言のための初期化指定子として現れます。一時オブジェクトは、初期化が完了すると破棄されます。
- 参照は、一時オブジェクトに拘束されます。一時オブジェクトは、参照が存続期間を終了する時に破棄されます。

コンパイラーは、コンストラクターを用いてクラスの一時オブジェクトを作成する場合、適切な (一致する) コンストラクターを呼び出して、一時オブジェクトを作成します。

一時オブジェクトが破棄され、デストラクターが存在すると、コンパイラーは、そのデストラクターを呼び出して、一時オブジェクトを破棄します。ユーザーが、一時オブジェクトが作成されたスコープから出るときに、その一時オブジェクトが破棄されます。参照が一時オブジェクトにバインドされている場合、一時オブジェクトが破棄されるのは、完了前に制御の流れが中断されたために一時オブジェクトが破棄された場合以外では、参照がスコープ外に出た場合です。例えば、参照メンバー用にコンストラクター初期化指定子によって作成された一時オブジェクトは、コンストラクターを離れるときに破棄されます。

そのような一時オブジェクトが冗長な場合は、コンパイラーは、最適化されたより効率的なコードを作成するために、それらの一時オブジェクトを構成しません。プログラムをデバッグする場合、特にメモリー問題のデバッグにおいては、この振る舞いを考慮に入れてください。

関連参照

- 335 ページの『catch ブロックの引数』
- 78 ページの『参照の初期化』
- 108 ページの『キャスト式』
- 152 ページの『関数からの戻り値』

ユーザー定義の型変換

C++ ユーザー定義の型変換 を使用して、コンストラクターまたは変換関数を使用したオブジェクト変換を指定することができます。初期化指定子、関数引数、関数からの戻り値、式のオペランド、式の反復制御、選択文の変換、および明示的型変換の標準型変換の他に、ユーザー定義の型変換が暗黙的に使用されます。

ユーザー定義の型変換には次の 2 つのタイプがあります。
- コンストラクターによる変換
- 変換関数

単一の値を暗黙的に変換する場合、コンパイラーは、ユーザー定義の型変換を 1 つだけ (型変換コンストラクターまたは型変換関数のどちらか) を使用することができます。次の例は、このことを示しています。

```
class A {
  int x;
public:
  operator int() { return x; };
};

class B {
  A y;
public:
  operator A() { return y; };
};

int main () {
```

```
  B b_obj;
//  int i = b_obj;
  int j = A(b_obj);
}
```

コンパイラーは、ステートメント int i = b_obj を許可しません。コンパイラーは、暗黙的に b_obj を型 A のオブジェクトに変換してから (B::operator A() を使用して)、暗黙的にそのオブジェクトを整数に変換しなければなりません (A::operator int() を使用して)。ステートメント int j = A(b_obj) は、暗黙的に b_obj を型 A のオブジェクトに変換してから、暗黙的にそのオブジェクトを整数に変換します。

ユーザー定義の型変換は、明確でなければなりません。さもないとそれらは呼び出されません。派生クラスの型変換関数は、両方の型変換関数が同じ型に変換されない限り、基底クラスにある別の型変換関数を隠しません。関数の多重定義解決は、最適な型変換関数を選択します。次の例は、このことを示しています。

```
class A {
  int a_int;
  char* a_carp;
public:
  operator int() { return a_int; }
  operator char*() { return a_carp; }
};

class B : public A {
  float b_float;
  char* b_carp;
public:
  operator float() { return b_float; }
  operator char*() { return b_carp; }
};

int main () {
  B b_obj;
//  long a = b_obj;
  char* c_p = b_obj;
}
```

コンパイラーは、ステートメント long a = b_obj を許可しません。コンパイラーは、A::operator int() または B::operator float() のどちらかを使用して、b_obj を **long** に変換できます。 B::operator char*() が A::operator char*() を隠すので、ステートメント char* c_p = b_obj は、B::operator char*() を使用して、b_obj を **char*** に変換します。

引数を持つコンストラクターを呼び出し、その引数型を受け入れるコンストラクターが定義されていない場合、標準型変換だけを使用して、その引数を、そのクラスのコンストラクターによって受け入れ可能な別の引数型に変換します。そのクラス用に定義されたコンストラクターに受け入れられる型に引数を変換するために、他のコンストラクターや変換関数を呼び出すことはありません。次の例は、このことを示しています。

```
class A {
public:
  A() { }
  A(int) { }
};

int main() {
   A a1 = 1.234;
//   A moocow = "text string";
}
```

コンパイラーは、ステートメント `A a1 = 1.234` を認めます。コンパイラーは、標準型変換を使用して、`1.234` を **int** に変換してから、暗黙的に変換コンストラクター `A(int)` を呼び出します。コンパイラーは、ステートメント `A moocow = "text string"` を許可しません。テキスト・ストリングを整数に変換するのは、標準の型変換ではありません。

関連参照
- 126 ページの『標準の型変換』

コンストラクターによる変換

C++ 変換コンストラクター は、関数指定子 **explicit** を指定せずに宣言される単一パラメーター・コンストラクターです。コンパイラーは、変換コンストラクターを使用して、オブジェクトを第 1 パラメーターの型から、変換コンストラクターのクラスの型に変換します。次の例は、このことを示しています。

```
class Y {
  int a, b;
  char* name;
public:
  Y(int i) { };
  Y(const char* n, int j = 0) { };
};

void add(Y) { };

int main() {

  // equivalent to
  // obj1 = Y(2)
  Y obj1 = 2;

  // equivalent to
  // obj2 = Y("somestring",0)
  Y obj2 = "somestring";

  // equivalent to
  // obj1 = Y(10)
  obj1 = 10;

  // equivalent to
  // add(Y(5))
  add(5);
}
```

上記の例には、次の 2 つの変換コンストラクターがあります。
- `Y(int i)` は、整数をクラス Y のオブジェクトに変換するために使用。
- `Y(const char* n, int j = 0)` は、ストリングを指すポインターを、クラス Y のオブジェクトに変換するために使用。

コンパイラーは、上記で説明したような型を、**explicit** キーワードを使用して宣言されたコンストラクターで暗黙的に変換しません。コンパイラーは、**new** 式、**static_cast** 式と明示キャスト、および基底とメンバーの初期化で明示的に宣言されたコンストラクターだけを使用します。次の例は、このことを示しています。

```
class A {
public:
  explicit A() { };
  explicit A(int) { };
};

int main() {
  A z;
```

```
//  A y = 1;
  A x = A(1);
  A w(1);
  A* v = new A(1);
  A u = (A)1;
  A t = static_cast<A>(1);
}
```

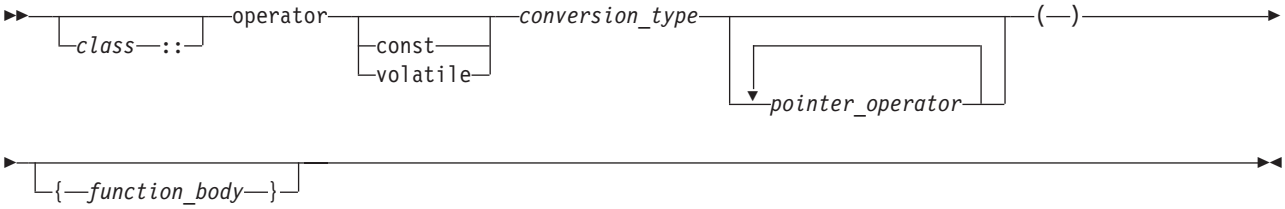
コンパイラーは、これが暗黙の型変換なので、ステートメント `A y = 1` を許可しません。クラス `A` は、型変換コンストラクターを持っていません。

コピー・コンストラクターは、変換コンストラクターです。

関連参照
- 133 ページの『明示的キーワード』
- 103 ページの『C++ の new 演算子』
- 93 ページの『static_cast 演算子』

変換関数

C++ *変換関数* と呼ばれる、クラスのメンバー関数を定義することができ、これは、そのクラスの型を指定された別の型に変換するものです。



クラス `X` に所属する型変換関数は、クラス型 `X` から *conversion_type* で指定する型に変換を指定します。次のコードは `operator int()` という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。

関連参照
- 126 ページの『標準の型変換』
- 288 ページの『ユーザー定義の型変換』
- 290 ページの『コンストラクターによる変換』
- 125 ページの『第 6 章 暗黙の型変換』

コピー・コンストラクター

C++ コピー・コンストラクター により、初期化をすることで、既存オブジェクトから新規オブジェクトを作成できます。クラス A のコピー・コンストラクターは、第 1 パラメーターが、型 A&、const A&、volatile A&、または const volatile A& である非テンプレート・コンストラクターであり、そのパラメーターの残りは (他にあれば)、デフォルト値を持っています。

クラス A に対してコピー・コンストラクターを宣言しない場合、コンパイラーは、コピー・コンストラクターを暗黙的に宣言し、それはインライン・パブリック・メンバーとなります。

次の例は、暗黙的に定義されたコピー・コンストラクターと、暗黙的なユーザー定義のコピー・コンストラクターを示しています。

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

struct A {
  int i;
  A() : i(10) { }
};

struct B {
  int j;
  B() : j(20) {
    cout << "Constructor B(), j = " << j << endl;
  }

  B(B& arg) : j(arg.j) {
    cout << "Copy constructor B(B&), j = " << j << endl;
  }

  B(const B&, int val = 30) : j(val) {
    cout << "Copy constructor B(const B&, int), j = " << j << endl;
  }
};

struct C {
  C() { }
  C(C&) { }
};

int main() {
  A a;
  A a1(a);
  B b;
  const B b_const;
  B b1(b);
  B b2(b_const);
  const C c_const;
//  C c1(c_const);
}
```

上記の例の出力は、以下のとおりです。

```
Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30
```

ステートメント A a1(a) は、暗黙的定義のコピー・コンストラクターを使用して、a から新規オブジェクトを作成します。ステートメント B b1(b) は、ユーザー定義のコピー・コンストラクター B::B(B&) を使用して、b から新規オブジェクトを作成します。ステートメント B b2(b_const) は、コピー・コンストラクター B::B(const B&, int) を使用して、新規オブジェクトを作成します。コンパイラーは、第 1 パラメーターとして型 const C& のオブジェクトを取得するコピー・コンストラクターが定義されていないので、ステートメント C c1(c_const) を許可しません。

次のことが真の場合、暗黙的に宣言されたクラス A のコピー・コンストラクターは、A::A(const A&) の書式を持ちます。

- A の直接基底および仮想基底は、第 1 パラメーターが、**const** または **const volatile** で修飾されたコピー・コンストラクターを持っている。
- A の非静的クラス型、またはクラス型データ・メンバーの配列は、第 1 パラメーターが、**const** または **const volatile** で修飾されたコピー・コンストラクターを持っている。

クラス A に対して上記のことが真でない場合、コンパイラーは、A::A(A&) の書式のコピー・コンストラクターを暗黙的に宣言します。

コンパイラーは、コンパイラーがクラス A のコピー・コンストラクターを暗黙的に定義する必要があり、次の中で 1 つまたは複数が真になるプログラムは、許可しません。

- クラス A が、非静的データ・メンバーを持ち、その型が、アクセス不能またはあいまいなコピー・コンストラクターを持っている。
- クラス A は、アクセス不能またはあいまいなコピー・コンストラクターを持つクラスから派生している。

型 A のオブジェクト、またはクラス A から派生したオブジェクトを初期化する場合、コンパイラーは、暗黙的に宣言されたクラス A のコンストラクターを暗黙的に定義します。

暗黙的に定義されたコピー・コンストラクターは、コンストラクターがオブジェクトの基底およびメンバーを初期化する順序と同じ順序で、オブジェクトの基底およびメンバーをコピーします。

関連参照

- 271 ページの『コンストラクターとデストラクターの概要』

コピー代入演算子

C++ コピー代入演算子 により、初期化をすることで、既存オブジェクトから新規オブジェクトを作成できます。クラス A のコピー代入演算子は、次の書式のいずれかを持つ非静的、非テンプレートのメンバー関数です。

- A::operator=(A)
- A::operator=(A&)
- A::operator=(const A&)
- A::operator=(volatile A&)
- A::operator=(const volatile A&)

クラス A に対してコピー代入演算子を宣言しない場合、コンパイラーは、コピー代入演算子を暗黙的に宣言し、それはインライン・パブリックとなります。

次の例は、暗黙的に定義された代入演算子と、暗黙的なユーザー定義の代入演算子を示しています。

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

struct A {
  A& operator=(const A&) {
    cout << "A::operator=(const A&)" << endl;
    return *this;
  }

  A& operator=(A&) {
    cout << "A::operator=(A&)" << endl;
    return *this;
  }

};
class B {
  A a;
};

struct C {
  C& operator=(C&) {
    cout << "C::operator=(C&)" << endl;
    return *this;
  }
  C() { }
};

int main() {
  B x, y;
  x = y;

  A w, z;
  w = z;

  C i;
  const C j();
//  i = j;
}
```

上記の例の出力は、以下のとおりです。

```
A::operator=(const A&)
A::operator=(A&)
```

代入 x = y は、暗黙的に定義された B のコピー代入演算子を呼び出します。つまり、ユーザー定義のコピー代入演算子 A::operator=(const A&) を呼び出します。代入 w = z は、ユーザー定義の演算子 A::operator=(A&) を呼び出します。コンパイラーは、演算子 C::operator=(const C&) が定義されていないので、代入 i = j を許可しません。

次のことが真の場合、暗黙的に宣言されたクラス A のコピー代入演算子は、 A& A::operator=(const A&) の書式を持ちます。

- クラス A の直接または仮想基底 B が、パラメーターが、型 const B&、const volatile B&、または B であるコピー代入演算子を持っている。
- クラス A に属する、型 X の非静的クラス型データ・メンバーが、パラメーターが、型 const X&、const volatile X&、または X であるコピー・コンストラクターを持っている。

クラス A に対して上記のことが真でない場合、コンパイラーは、A& A::operator=(A&) の書式を使用したコピー代入演算子を暗黙的に宣言します。

暗黙的に宣言されたコピー代入演算子は、演算子の引数への参照を戻します。

派生クラスのコピー代入演算子は、その基底クラスのコピー代入演算子を隠します。

コンパイラーは、クラス A に対してコピー代入演算子を暗黙的に定義しなければならず、次のなかで 1 つまたは複数が真になっているようなプログラムは、許可しません。
- クラス A は、**const** 型、または参照型の非静的データ・メンバーを持つ。
- クラス A は、型が非静的データ・メンバーで、アクセス不能のコピー代入演算子を持つ。
- クラス A は、アクセス不能なコピー代入演算子を使用した基底クラスから派生する。

暗黙的に定義されたクラス A のコピー代入演算子は、まず最初に、A の定義にそれらが現れる順序で、A の直接基底クラスを割り当てます。次に、暗黙的に定義されるコピー代入演算子は、A の定義でそれらが宣言されている順序で、A の非静的データ・メンバーを割り当てます。

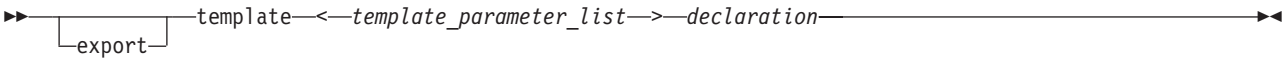
関連参照
- 120 ページの『代入式』

第 16 章 テンプレート

C++ テンプレート では、関連するクラスのセット、または関連する関数のセットについて記述し、その宣言のパラメーターのリストでは、そのセットのメンバーが、どのように異なるかを記述します。これらのパラメーターに引数を提供すると、コンパイラーは、新規のクラスまたは関数を生成します。このプロセスは、テンプレートのインスタンス化 と呼ばれます。テンプレート、およびテンプレート・パラメーターのセットから生成されたこのクラスや関数定義は、特殊化 と呼ばれます。

400 i5/OS 固有の使用の詳細については、「*ILE C/C++ Programmer's Guide*」の第 28 章『Using Templates in C++ Programs』を参照してください。

構文 - テンプレート宣言



コンパイラーは、テンプレートで export キーワードを受諾し、暗黙に無視します。

template_parameter_list は、次に示す種類のテンプレート・パラメーターの、コンマで分離したリストです。
- 非型
- 型
- template

declaration は、次のいずれかです。
- 関数またはクラスの宣言または定義
- メンバー関数またはクラス・テンプレートのメンバー・クラスの定義
- クラス・テンプレートの静的データ・メンバーの定義
- クラス・テンプレート内のネスト・クラスの静的データ・メンバーの定義
- クラスまたはクラス・テンプレートのメンバー・テンプレートの定義

型 の *ID* が、テンプレート宣言のスコープ内の *type_name* であると定義されます。テンプレート宣言は、ネーム・スペース・スコープ宣言またはクラス・スコープ宣言として現れます。

次の例は、クラス・テンプレートの使用法を示しています。

```
template<class L> class Key
{
    L k;
    L* kptr;
    int length;
public:
    Key(L);
    // ...
};
```

その後、次の宣言が現れるとします。

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

コンパイラーは、3 つのオブジェクトを作成します。次の表は、3 つのオブジェクトが、ソース・コードの書式で、テンプレートとしてではなく通常のクラスとして書き出された場合の、それらオブジェクトの定義を示しています。

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

C++ テンプレート・パラメーターには、下記の 3 種類があります。

- 型
- 非型
- template

テンプレート・パラメーター宣言で、キーワード **class** と **typename** は交換できます。テンプレート・パラメーター宣言内では、ストレージ・クラス指定子 (**static** および **auto**) は使用できません。

「型」テンプレート・パラメーター

C++ 以下は、「型」テンプレート・パラメーター宣言の構文です。

構文 -「型」テンプレート・パラメーター宣言

identifier は、型の名前です。

「非型」テンプレート・パラメーター

C++ 「非型」テンプレート・パラメーターの構文は、次のいずれかの型の宣言と同じです。

- 整数または列挙型
- オブジェクトへのポインターまたは関数を指すポインター
- オブジェクトへの参照または関数への参照
- メンバーへのポインター

配列、または関数として宣言された「非型」テンプレート・パラメーターは、ポインター、または関数を指すポインターにそれぞれ変換されます。次の例は、このことを示しています。

```
template<int a[4]> struct A { };
template<int f(int)> struct B { };

int i;
int g(int) { return 0;}

A<&i> x;
B<&g> y;
```

&i の型は、**int** * で、 &g の型は、**int (*)(int)** です。

「非型」テンプレート・パラメーターを **const** または **volatile** で修飾できます。

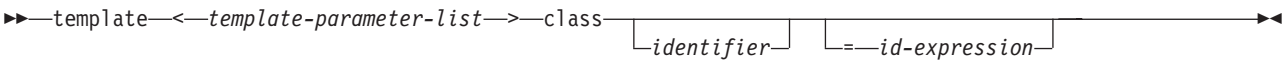
「非型」テンプレート・パラメーターを、浮動小数点、クラス、または void 型として宣言することはできません。

「非型」テンプレート・パラメーターは、左辺値ではありません。

「テンプレート」テンプレート・パラメーター

C++ 以下は、「テンプレート」テンプレート・パラメーター宣言の構文です。

構文 -「テンプレート」テンプレート・パラメーター宣言



次の例は、「テンプレート」テンプレート・パラメーターの宣言と使い方を示しています。

```
template<template <class T> class X> class A { };
template<class T> class B { };

A<B> a;
```

テンプレート・パラメーターのデフォルトの引数

C++ テンプレート・パラメーターは、デフォルトの引数を持つことができます。デフォルトのテンプレート引数のセットは、任意のテンプレートの宣言すべてに累積していきます。次の例は、このことを示しています。

```
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
   public:
      T x;
      U y;
};

A<> a;
```

メンバー a.x の型は **float** で、a.y の型は **int** です。

デフォルトの引数を、同じスコープ内の異なる宣言にある、同じテンプレート・パラメーターに与えることはできません。例えば、コンパイラーは次のことを許可しません。

```
template<class T = char> class X;
template<class T = char> class X { };
```

あるテンプレート・パラメーターが、デフォルトの引数を持つ場合、それに続くテンプレート・パラメーターも、すべてデフォルトの引数を持つはずです。例えば、コンパイラーは次のことを許可しません。

```
template<class T = char, class U, class V = int> class X { };
```

テンプレート・パラメーター U は、デフォルトの引数が必要です。あるいは T のデフォルトを除去する必要があります。

テンプレート・パラメーターのスコープは、その宣言のポイントからそのテンプレート定義の終了までです。つまり、他のテンプレート・パラメーター宣言内のテンプレート・パラメーターの名前、およびそれらのデフォルトの引数を使用できるということです。次の例は、このことを示しています。

```
template<class T = int> class A;
template<class T = float> class B;
template<class V, V obj> class A;
// a template parameter (T) used as the default argument
// to another template parameter (U)
template<class T, class U = T> class C { };
```

テンプレート引数

C++ 下記の 3 つのテンプレート・パラメーターの型に対応する、3 種類のテンプレート引数があります。

- type
- 非型
- template

テンプレート引数は、テンプレートに宣言された対応パラメーターが指定する型、およびフォームと一致しなければなりません。

テンプレート・パラメーターのデフォルト値を使用するには、対応するテンプレート引数を省略します。しかし、たとえすべてのテンプレート・パラメーターがデフォルトを持っていても、<> 大括弧を使用する必要があります。例えば、次は構文エラーが発生します。

```
template<class T = int> class X { };
X<> a;
X b;
```

最後の宣言 X b は、エラーになります。

テンプレート型引数

C++ 次のいずれかを、「型」テンプレート・パラメーターのテンプレート引数として使用することはできません。

- ローカル型
- リンケージなしの型
- 無名型
- 上記の型のいずれかを複合した型

テンプレート引数が、型なのか、式なのかあいまいな場合は、テンプレート引数は、型であると見なされます。次の例は、このことを示しています。

```
template<class T> void f() { };
template<int i> void f() { };

int main() {
  f<int()>();
}
```

関数呼び出し f<int()>() は、T をテンプレート引数として、関数を呼び出します。このときコンパイラーは、int() を型として扱い、したがって暗黙的にインスタンスを作成し、最初の f() を呼び出します。

テンプレート非型引数

C++ テンプレート引数リストに指定されている「非型」テンプレート引数は、コンパイル時に値が決められる式です。このような引数は、定数式、関数のアドレス、外部リンケージのあるオブジェクト、または静的クラス・メンバーのアドレスでなければなりません。通常は、「非型」テンプレート引数を使用して、クラスの初期化またはクラス・メンバーのサイズを指定します。

非型整数引数の場合、インスタンス引数は、その引数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引数と一致します。

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引数と一致していなければなりません。

テンプレート引数リスト内に「非型」テンプレート引数がある場合、結果として得られる値は、そのテンプレート・クラス型の一部を形成します。 2 つのテンプレート・クラス名が同じテンプレート名を持っており、それらの引数の値が同じ場合、それらは同じクラスであるといいます。

次の例では、クラス・テンプレートが、型引数だけでなく、「非型」テンプレート **int** 引数も必要であると定義されています。

```
template<class T, int size> class myfilebuf
{
      T* filepos;
      static int array[size];
public:
      myfilebuf() { /* ... */ }
      ~myfilebuf();
      advance(); // function defined elsewhere in program
};
```

この例では、テンプレート引数 size が、テンプレート・クラス名の一部になります。このようなテンプレート・クラスのオブジェクトは、クラス型引数 T と「非型」テンプレート引数 size の値の両方を指定して作成されます。

オブジェクト x および引数 **double** と size=200 を持つ、その対応するテンプレート・クラスは、その 2 番目のテンプレート引数として値を持ったこのテンプレートから作成することができます。

```
myfilebuf<double,200> x;
```

x も、演算式を使用して作成できます。

```
myfilebuf<double,10*20> x;
```

これらの式によって作成されるオブジェクトは、テンプレート引数の評価が同じになるので、同一になります。最初の式の中の値 200 は、2 番目の構成に示すように、コンパイル時の結果が 200 に等しいということがわかっている式によって表すこともできます。

注: < 記号、または > 記号を含む引数は、関係演算子またはネストされたテンプレート区切り文字として使用される場合は、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、それを括弧で囲む必要があります。例えば、次の定義の中の引数は有効です。

```
myfilebuf<double, (75>25)> x;       // valid
```

ただし次の定義では、より大の演算子 (>) がテンプレート引数リストの終了区切り文字と解釈されるので、有効ではありません。

```
myfilebuf<double, 75>25> x;         // error
```

テンプレート引数の評価結果が同一でなければ、作成されたオブジェクトは異なる型になります。

```
myfilebuf<double,200> x;  // create object x of class
                                  // myfilebuf<double,200>
myfilebuf<double,200.0> y;        // error, 200.0 is a double,
                                  // not an int
```

y のインスタンス化は、値 200.0 の型が **double** で、テンプレート引数の型が **int** であるために失敗します。

次の 2 つのオブジェクトは、

```
myfilebuf<double, 128> x
myfilebuf<double, 512> y
```

分離テンプレート特殊化のオブジェクトです。後でこれらのオブジェクトのいずれかを myfilebuf<double> で参照するとエラーになります。

クラス・テンプレートは、非型引数を持つ場合は、型引数を持つ必要がありません。例えば、次のテンプレートは有効なクラス・テンプレートです。

```
template<int i> class C
{
      public:
            int k;
            C() { k = i; }
};
```

このクラス・テンプレートは、次のような宣言でインスタンスを生成できます。

```
class C<100>;
class C<200>;
```

繰り返しですが、これら 2 つの宣言は、これらの非型引数の値が異なるので、別個のクラスを参照しています。

「テンプレート」テンプレート引数

C++ 「テンプレート」テンプレート・パラメーターのテンプレート引数は、クラス・テンプレートの名前です。

コンパイラーが「テンプレート」テンプレート引数と一致するテンプレートの検索を試みる場合、それは主クラス・テンプレートだけを検索します。 (主テンプレート は、特殊化しようとしているテンプレートのことです。) コンパイラーは、たとえそれらのパラメーター・リストが、「テンプレート」テンプレート・パラメーターのリストと一致していても、部分的な特殊化は考慮に入れません。例えば、コンパイラーは次のコードを許可しません。

```
template<class T, int i> class A {
   int x;
};

template<class T> class A<T, 5> {
   short x;
};

template<template<class T> class U> class B1 { };

B1<A> c;
```

コンパイラーは、宣言 B1<A> c を許可しません。A の部分的な特殊化は、B1 の「テンプレート」テンプレート・パラメーター U と一致しているように見えますが、コンパイラーは、U とは異なるテンプレート・パラメーターを持つ、主テンプレート A だけを考慮します。

「テンプレート」テンプレート・パラメーターを基にした特殊化のインスタンスをいったん作成すると、コンパイラーは、それに対応する「テンプレート」テンプレート引数に基づく部分的な特殊化を考慮します。次の例は、このことを示しています。

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

template<class T, class U> class A {
   int x;
};

template<class U> class A<int, U> {
   short x;
};

template<template<class T, class U> class V> class B {
   V<int, char> i;
   V<char, char> j;
};

B<A> c;

int main() {
   cout << typeid(c.i.x).name() << endl;
   cout << typeid(c.j.x).name() << endl;
}
```

上記の例の出力は、以下のとおりです。

```
short
int
```

宣言 V<int, char> i は、部分的な特殊化を使用しますが、宣言 V<char, char> j は、主テンプレートを使用します。

クラス・テンプレート

C++ クラス・テンプレートと個々のクラスとの間の関係は、クラスと個々のオブジェクトとの間の関係に似ています。個々のクラスがオブジェクトのグループの構成方法を定義し、一方、クラス・テンプレートがクラスのグループの生成方法を定義します。

クラス・テンプレート とテンプレート・クラス という用語の間の区別に注意してください。

クラス・テンプレート
: これは、テンプレート・クラスの生成に使用されるテンプレートです。クラス・テンプレートのオブジェクトは、宣言できません。

テンプレート・クラス
: クラス・テンプレートのインスタンスです。

テンプレート定義は、下記の点を除いて、テンプレートが生成し得る有効なクラス定義のいずれとも同一です。

- クラス・テンプレート定義には、次の語が先行します。

  template
  < *template-parameter-list* **>**

  ここで、*template-parameter-list* は、次に示す種類のテンプレート・パラメーターの 1 つまたは複数を、コンマで分離したリストです。
  - 型
  - 非型
  - template
- クラス・テンプレート内の型、変数、定数およびオブジェクトを、テンプレート・パラメーターおよび明示型 (例えば、**int** や **char**) を使用して宣言することができます。

詳述型指定子を使用して定義しなくても、クラス・テンプレートを宣言することができます。次に例を示します。

```
template
<class L,class T> class key;
```

これにより、名前がクラス・テンプレート名として予約されます。クラス・テンプレートのテンプレート宣言は、すべてが、同じ型と同じ数のテンプレート引数を持っていなければなりません。クラス定義を含む 1 つのテンプレート宣言だけが許可されます。

注: テンプレート引数リストがネストされている場合は、内側のリストの終わりの > と外側のリストの終わりの > の間に分離スペースが必要です。これがなければ、出力演算子 >> と 2 つのテンプレート・リスト区切り文字 > との間があいまいになります。

```
template <class L,class T> class key { /* ... */
};
template <class L> class vector { /* ... */ };;

int main ()
{
   class key <int, vector<int> >;
   // implicitly instantiates template
}
```

通常のクラス・メンバーのオブジェクトや関数のアクセスに使用されるどの手法でも、個々のテンプレート・クラスのオブジェクトや関数メンバーをアクセスすることができます。次のクラス・テンプレートがあるとします。

```
template<class T> class vehicle
{
public:
    vehicle() { /* ... */ }    // constructor
    ~vehicle() {};             // destructor
    T kind[16];
```

```
    T* drive();
    static void roadmap();
    // ...
};
```

そして、次の宣言を行います。

```
vehicle<char> bicycle; // instantiates the template
```

コンストラクター、構成オブジェクト、およびメンバー関数 drive() は、次のいずれかを指定してアクセスできます (標準ヘッダー・ファイル <string.h> が、プログラム・ファイルに含まれているとします)。

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();`

クラス・テンプレートの宣言と定義

C++ クラス・テンプレートを宣言してから、対応するテンプレート・クラスの宣言を行う必要があります。クラス・テンプレートの定義は、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。

テンプレート・クラスの定義が必要になるまで、クラス・テンプレートの定義はコンパイルされません。その定義が必要になった時点で、テンプレート・クラスの引数リストを使用して、テンプレート引数のインスタンスを生成することによって、クラス・テンプレートの定義をコンパイルします。この時点で、クラス定義の中のエラーにフラグを付けます。

静的データ・メンバーとテンプレート

C++ どのクラス・テンプレートのインスタンス化も、静的データ・メンバーの専用コピーを所有します。静的宣言は、テンプレート引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<class T> T K<T>::x` は、クラス `T` の静的メンバーを定義しますが、`main()` 関数のステートメントは、`K` のデータ・メンバーを初期化します。

クラス・テンプレートのメンバー関数

C++ テンプレートのメンバー関数を、そのクラス・テンプレート定義の外側に定義できます。

クラス・テンプレート特殊化のメンバー関数を呼び出す場合、コンパイラーは、以前、クラス・テンプレートの作成に使用したテンプレート引数を使用します。次の例は、このことを示しています。

```
template<class T> class X {
   public:
      T operator+(T);
};

template<class T> T X<T>::operator+(T arg1) {
   return arg1;
};

int main() {
   X<char> a;
   X<int> b;
   a +'z';
   b + 4;
}
```

多重定義された加法演算子は、クラス `X` の外側で定義されています。ステートメント `a + 'z'` は、`a.operator+('z')` と同等です。ステートメント `b + 4` は、`b.operator+(4)` と同等です。

フレンドとテンプレート

C++ テンプレートを含める場合、クラスとそれらのフレンドとの間には 4 種類の関係があります。

- *1 対多*: 非テンプレート関数は、すべてのテンプレート・クラスのインスタンス生成へのフレンドです。
- *多対 1*: テンプレート関数のすべてのインスタンス生成は、通常の非テンプレート・クラスへのフレンドです。
- *1 対 1*: テンプレート引数の 1 セットを使用したテンプレート関数のインスタンス生成は、同じテンプレート引数のセットを使用してインスタンス生成された 1 つのテンプレート・クラスのフレンドです。これは、通常の非テンプレート・クラスと通常の非テンプレート・フレンド関数との間の関係でもあります。
- *多対多*: テンプレート関数のすべてのインスタンス生成は、テンプレート・クラスのすべてのインスタンス生成へのフレンドです。

次の例は、これらの関係を示しています。

```
class B{
   template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<class U> friend int h();
};
```

- 関数 e() は、クラス A と 1 対多の関係を持ちます。関数 e() は、クラス A のすべてのインスタンス生成のフレンドです。
- 関数 f() は、クラス A と 1 対 1 の関係を持ちます。コンパイラーは、この種の宣言に対して、以下に類似する警告を出します。

```
The friend function declaration "f" will cause an error when the enclosing
template class is instantiated with arguments that declare a friend function
that does not match an existing definition. The function declares only one
function because it is not a template but the function type depends on
one or more template parameters.
```

- 関数 g() は、クラス A と 1 対 1 の関係を持ちます。関数 g() は、関数テンプレートです。この前に宣言する必要があります。そうしないと、コンパイラーは g<T> をテンプレート名として認識しません。 A のインスタンス生成ごとに、g() にマッチングするインスタンス生成が 1 つあります。例えば、g<int> は、A<int> のフレンドです。
- 関数 h() は、クラス A と多対多の関係を持ちます。関数 h() は、関数テンプレートです。 A のすべてのインスタンス生成にとって、h() のインスタンス生成は、すべてフレンドです。
- 関数 j() は、クラス B と多対 1 の関係を持ちます。

これらの関係は、フレンド・クラスにも適用します。

関数テンプレート

C++ *関数テンプレート* は、関数のグループの生成方法を定義します。

非テンプレート関数は、テンプレートから生成された特殊化の関数と同じ名前、およびパラメーター・プロファイルを持っていたとしても、非テンプレート関数は、関数テンプレートとは関連がありません。非テンプレート関数は、関数テンプレートの特殊化と見なされることは、ありません。

次の例は、quicksort という名前の関数テンプレートを使用した、QuickSort アルゴリズムをインプリメントします。

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

template<class T> void quicksort(T a[], const int& leftarg, const int& rightarg)
{
  if (leftarg < rightarg) {

    T pivotvalue = a[leftarg];
    int left = leftarg - 1;
    int right = rightarg + 1;

  for(;;) {

    while (a[--right] > pivotvalue);
    while (a[++left] < pivotvalue);
```

```
    if (left >= right) break;

    T temp = a[right];
    a[right] = a[left];
    a[left] = temp;
  }

  int pivot = right;
  quicksort(a, leftarg, pivot);
  quicksort(a, pivot + 1, rightarg);
  }
}

int main(void) {
  int sortme[10];

  for (int i = 0; i < 10; i++) {
    sortme[i] = rand();
    cout << sortme[i] << " ";
  };
  cout << endl;

  quicksort<int>(sortme, 0, 10 - 1);

  for (int i = 0; i < 10; i++) cout << sortme[i] << "
  ";
  cout << endl;
  return 0;
}
```

上記の例は、次に類似する出力を行います。

```
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
4086 5627 5758 7419 10113 16212 16838 17515 23010 31051
```

QuickSort アルゴリズムは、型 T の配列 (これの関係演算子および代入演算子は、定義されている) をソートします。テンプレート関数は、1 つのテンプレート引数と 3 つの関数引数を取ります。

- ソートされる配列の型、T
- ソートされる配列の名前、a
- 配列の下限、leftarg
- 配列の上限、rightarg

上記の例では、次のステートメントを使用して、quicksort() テンプレート関数を呼び出すこともできます。

```
  quicksort(sortme, 0, 10 - 1);
```

コンパイラーが、テンプレート関数呼び出しの使い方とコンテキストにより、テンプレート引数を推定できる場合、テンプレート引数を省略できます。ここでは、コンパイラーは、sortme が、型 **int** の配列であると推定します。

テンプレート引数の推定

C++ テンプレート関数を呼び出す場合、テンプレート関数呼び出しの使い方とそのコンテキストによって、コンパイラーが決定または*推定* できるテンプレート引数は、どれでも省略できます。

コンパイラーは、対応するテンプレート・パラメーターの型と、関数呼び出しで使用される引数の型を比較することにより、テンプレート引数を推定しようとします。テンプレート引数の推定を行うためには、コン

パイラーが比較する 2 つの型 (テンプレート・パラメーターと関数呼び出しで使用される引数) は、ある特定の構造体でなければなりません。下記に、これらの型構造体をリストします。

```
T
const T
volatile T
T&
T*
T[10]
A<T>
C(*)(T)
T(*)()
T(*)(U)
T C::*
C T::*
T U::*
T (C::*)()
C (T::*)()
D (C::*)(T)
C (T::*)(U)
T (C::*)(U)
T (U::*)()
T (U::*)(V)
E[10][i]
B<i>
TT<T>
TT<i>
TT<C>
```

- T、U、および V は、テンプレート型引数を表す
- 10 は、整数定数を示す
- i は、テンプレート非型引数を示す
- [i] は、参照またはポインター型の配列境界、あるいは標準配列の非主配列の境界を示す
- TT は、「テンプレート」テンプレート引数を示す
- (T)、(U)、および (V) は、少なくとも 1 つのテンプレート型引数を持つ引数リストを示す
- () は、テンプレート引数を持っていない引数リストを示す
- <T> は、少なくとも 1 つのテンプレート型引数を持つテンプレート引数リストを示す
- <i> は、少なくとも 1 つのテンプレート非型引数を持つテンプレート引数リストを示す
- <C> は、テンプレート・パラメーターに従属するテンプレート引数を持っていない、テンプレート引数リストを示す

次の例は、これら型構造体のそれぞれの使用法を示しています。例では、引数として上記の各構造体を使用して、テンプレート関数を宣言しています。そして、これらの関数が、宣言順に (テンプレート引数を使用せずに) 呼び出されます。この例は、型構造体のリストと同様なものを出力します。

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

template<class T> class A { };
template<int i> class B { };

class C {
  public:
      int x;
};

class D {
  public:
      C y;
      int z;
};

template<class T> void f (T)          { cout << "T" << endl; };
```

```
template<class T> void f1(const T)    { cout << "const T" << endl; };
template<class T> void f2(volatile T) { cout << "volatile T" << endl;  };
template<class T> void g (T*)         { cout << "T*" << endl; };
template<class T> void g (T&)         { cout << "T&" << endl; };
template<class T> void g1(T[10])      { cout << "T[10]" << endl;};
template<class T> void h1(A<T>)       { cout << "A<T>" << endl; };

void test_1() {
   A<char> a;
   C c;

   f(c);   f1(c);   f2(c);
   g(c);   g(&c);   g1(&c);
   h1(a);
}

template<class T>          void j(C(*)(T)) { cout << "C(*) (T)" << endl; };
template<class T>          void j(T(*)())  { cout << "T(*) ()" << endl; }
template<class T, class U> void j(T(*)(U)) { cout << "T(*) (U)" << endl; };

void test_2() {
   C (*c_pfunct1)(int);
   C (*c_pfunct2)(void);
   int (*c_pfunct3)(int);
   j(c_pfunct1);
   j(c_pfunct2);
   j(c_pfunct3);
}

template<class T>          void k(T C::*) { cout << "T C::*" << endl; };
template<class T>          void k(C T::*) { cout << "C T::*" << endl; };
template<class T, class U> void k(T U::*) { cout << "T U::*" << endl; };

void test_3() {
   k(&C::x);
   k(&D::y);
   k(&D::z);
}

template<class T>     void m(T (C::*)() )
   { cout << "T (C::*)()" << endl; };
template<class T>     void m(C (T::*)() )
   { cout << "C (T::*)()" << endl; };
template<class T>     void m(D (C::*)(T))
   { cout << "D (C::*)(T)" << endl; };
template<class T, class U>  void m(C (T::*)(U))
   { cout << "C (T::*)(U)" << endl; };
template<class T, class U>  void m(T (C::*)(U))
   { cout << "T (C::*)(U)" << endl; };
template<class T, class U>  void m(T (U::*)() )
   { cout << "T (U::*)()" << endl; };
template<class T, class U, class V> void m(T (U::*)(V))
   { cout << "T (U::*)(V)" << endl; };

void test_4() {
   int (C::*f_membp1)(void);
   C (D::*f_membp2)(void);
   D (C::*f_membp3)(int);
   m(f_membp1);
   m(f_membp2);
   m(f_membp3);

   C (D::*f_membp4)(int);
   int (C::*f_membp5)(int);
   int (D::*f_membp6)(void);
   m(f_membp4);
   m(f_membp5);
```

```
   m(f_membp6);

   int (D::*f_membp7)(int);
   m(f_membp7);
}

template<int i> void n(C[10][i]) { cout << "E[10][i]" << endl; };
template<int i> void n(B<i>)     { cout << "B<i>" << endl; };

void test_5() {
   C array[10][20];
   n(array);
   B<20> b;
   n(b);
}

template<template<class> class TT, class T> void p1(TT<T>)
   { cout << "TT<T>" << endl; };
template<template<int> class TT, int i>     void p2(TT<i>)
   { cout << "TT<i>" << endl; };
template<template<class> class TT>          void p3(TT<C>)
   { cout << "TT<C>" << endl; };

void test_6() {
   A<char> a;
   B<20> b;
   A<C> c;
   p1(a);
   p2(b);
   p3(c);
}

int main() { test_1(); test_2(); test_3(); test_4(); test_5(); test_6(); }
```

「型」テンプレート引数の推定

C++ コンパイラーは、リストされたいくつかの型構造体で構成される型から、テンプレート引数を推定できます。次の例は、いくつかの型構造体で構成される型からのテンプレート引数の推定を示しています。

```
template<class T> class Y { };

template<class T, int i> class X {
  public:
      Y<T> f(char[20][i]) { return x; };
      Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
   void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
   Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
   g(p);
}
```

型 `Y<int> (X<int, 20>::*p)(char[20][20])` `T<U> (V::*)(W[20][i])` は、型構造体 `T (U::*)(V)` に基づいています。

- `T` は、`Y<int>` です
- `U` は、`X<int, 20>` です。
- `V` は、`char[20][20]` です

型が属するクラスを使用してその型を修飾し、そのクラス (*ネストされた名前指定子*) がテンプレート・パラメーターに依存する場合、コンパイラーは、そのパラメーターのテンプレート引数を推定できません。型

が、この理由により推定できないテンプレート引数を含んでいる場合、その型にあるすべてのテンプレート引数は、推定されません。次の例は、このことを示しています。

```
template<class T, class U, class V>
  void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
  Y<int>::Z<char> a;
  Y<int> b;
  Y<float> c;

  h<int, char, float>(a, b, c);
  h<int, char>(a, b, c);
  // h<int>(a, b, c);
}
```

コンパイラーは、`typename Y<T>::template Z<U>` のテンプレート引数 `T` および `U` を推定できません (しかし、`Y<T>` の `T` は推定します)。コンパイラーは、`U` がそのコンパイラーによって推定されないので、テンプレート関数呼び出し `h<int>(a, b, c)` を許可しません。

コンパイラーは、関数を指すポインターから、またはいくつかの多重定義関数名を与えられたメンバー関数引数を指すポインターから、関数テンプレート引数を推定できます。しかし、多重定義関数が、いずれも関数テンプレートではないこともあり、複数の多重定義関数が、要求される型と一致しないこともあります。次の例は、このことを示しています。

```
template<class T> void f(void(*) (T,int)) { };

template<class T> void g1(T, int) { };

void g2(int, int) { };
void g2(char, int) { };

void g3(int, int, int) { };
void g3(float, int) { };

int main() {
//   f(&g1);
//   f(&g2);
   f(&g3);
}
```

コンパイラーは、`g1()` が関数テンプレートなので、呼び出し `f(&g1)` を許可しません。コンパイラーは、`g2()` という名前の関数が、両方とも `f()` が必要とする型と一致するので、呼び出し `f(&g2)` を許可しません。

コンパイラーは、デフォルト引数の型からテンプレート引数を推定できません。次の例は、このことを示しています。

```
template<class T> void f(T = 2, T = 3) { };

int main() {
   f(6);
//   f();
   f<int>();
}
```

コンパイラーは、関数呼び出しの引数値からテンプレート引数 (**int**) を推定できるので、呼び出し `f(6)` を許可します。コンパイラーは、`f()` のデフォルトの引数からテンプレート引数を推定できないので、呼び出し `f()` を許可しません。

コンパイラーは、「非型」テンプレート引数の型からテンプレート型引数を推定できません。例えば、コンパイラーは次のコードを許可しません。

```
template<class T, T i> void f(int[20][i]) { };

int main() {
   int a[20][30];
   f(a);
}
```

コンパイラーは、テンプレート・パラメーター T の型を推定できません。

非型テンプレート引数の推定

C++ コンパイラーは、境界が参照またはポインター型を参照しない限り、主配列の境界の値を推定できません。主配列の境界は、関数仮パラメーター型の一部ではありません。次のコードは、このことを示しています。

```
template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
   int b[10][20];
   int c[10];
   f(b);
   // g(c);
   h(c);
}
```

コンパイラーは、呼び出し g(c) を許可しません。コンパイラーは、テンプレート引数 i を推定できません。

コンパイラーは、テンプレート関数のパラメーター・リストの式で使用されている、「非型」テンプレート引数の値を推定できません。次の例は、このことを示しています。

```
template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
  X<0> a;
  f<1>(a);
  // f(a);
}
```

関数 f() をオブジェクト a で呼び出すためには、関数が、型 X<0> の引数を受諾する必要があります。しかし、コンパイラーは、X<i - 1> が X<0> と同等であるためには、テンプレート引数 i が、1 と等しい必要があるということを推定できません。したがって、コンパイラーは、関数呼び出し f(a) を許可しません。

コンパイラーに「非型」テンプレート引数を推定させたい場合、パラメーターの型が、関数呼び出しで使用される値の型と正確に一致しなければなりません。例えば、コンパイラーは次のことを許可しません。

```
template<int i> class A { };
template<short d> void f(A<d>) { };

int main() {
   A<1> a;
   f(a);
}
```

例で **f()** を呼び出す場合、コンパイラーは、**int** を short に変換しません。

しかし、推定された配列の境界は、整数型になります。

関数テンプレートの多重定義

C++ 非テンプレート関数、または別の関数テンプレートのどちらかを使用して、関数テンプレートを多重定義できます。

多重定義関数テンプレートの名前を呼び出す場合、コンパイラーは、そのテンプレート引数の推定を試み、明示的に宣言されたテンプレート引数をチェックします。成功すれば、コンパイラーは、関数テンプレート特殊化のインスタンスを作成してから、この特殊化を多重定義解決で使用する*候補関数* のセットに追加します。コンパイラーは、多重定義解決を続け、候補関数のセットから最も適切な関数を選択します。非テンプレート関数は、テンプレート関数より優先順位があります。次の例は、このことを示しています。

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

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');
}
```

上記の例の出力は、以下のとおりです。

```
Non-template
Template
Non-template
```

関数呼び出し f(1, 2) は、テンプレート関数と非テンプレート関数の両方の引数型と一致します。多重定義の解決では非テンプレート関数の方が優先順位が高いと見なされるため、非テンプレート関数が呼び出されます。

関数呼び出し f('a', 'b') は、テンプレート関数の引数型とだけ一致します。テンプレート関数が呼び出されます。

関数呼び出し f(1, 'b') では、引数の推定は失敗します。コンパイラーは、テンプレート関数特殊化を生成せず、また、多重定義解決も生じません。非テンプレート関数は、関数引数 'b' に、標準型変換を使用して **char** から **int** に変換した後で、この関数呼び出しを解決します。

関数テンプレートの部分選択

C++ 関数テンプレートの特殊化は、テンプレート引数の推定で、特殊化が複数の多重定義と関連付けられるので、あいまいになります。そのため、コンパイラーは、最も特殊化された定義を選択します。関数テンプレート定義を選択するこの処理は、*部分選択* と呼ばれます。

X からの特殊化と一致する引数リストは、いずれも Y からの特殊化と一致するが、その逆では一致しないという場合は、テンプレート X は、テンプレート Y よりもさらに特殊化されています。次の例は、部分選択を示しています。

```
template<class T> void f(T)  { }
template<class T> void f(T*) { }
template<class T> void f(const T*) { }

template<class T> void g(T) { }
template<class T> void g(T&) { }

template<class T> void h(T) { }
template<class T> void h(T, ...) { }

int main() {
   const int *p;
   f(p);

   int q;
//  g(q);
//  h(q);
}
```

宣言 `template<class T> void f(const T*)` は、 `template<class T> void f(T*)` よりもさらに特殊化されています。したがって、関数呼び出し `f(p)` は、 `template<class T> void f(const T*)` を呼び出します。しかし、`void g(T)` および `void g(T&)` はどちらも、特殊化の程度には差はありません。したがって、関数呼び出し `g(q)` はあいまいになります。

省略符号は、部分選択に影響を与えません。したがって、関数呼び出し `h(q)` もあいまいです。

コンパイラーは、次の場合に、部分選択を使用します。
- 多重定義解決を必要とする関数テンプレート特殊化の呼び出し
- 関数テンプレート特殊化のアドレスの取得
- フレンド関数宣言、明示的インスタンス生成、または明示的特殊化が、関数テンプレート特殊化を参照するとき
- 任意の割り振り解除 new の関数テンプレートでもある適切な配置解除関数の決定

テンプレートのインスタンス化

C++ 関数、クラス、またはクラスのメンバーの新規定義を、テンプレート宣言と 1 つ以上のテンプレート引数から作成する処理は、*テンプレートのインスタンス化* と呼ばれます。テンプレートのインスタンス化から作成された定義は、*特殊化* と呼ばれます。

暗黙のインスタンス化

C++ テンプレートの特殊化が明示的にインスタンス化されない限り、または明示的に特殊化されない限り、コンパイラーは、定義が必要とされる場合にのみ、テンプレートの特殊化を生成します。これは、*暗黙のインスタンス化* と呼ばれます。

コンパイラーが、クラス・テンプレート特殊化のインスタンスを生成する必要があり、テンプレートが宣言される場合、テンプレートも定義する必要があります。

例えば、クラスを指すポインターを宣言する場合、そのクラスの定義は、必要とされず、そのクラスは、暗黙的にインスタンス作成されません。次は、コンパイラーが、テンプレート・クラスのインスタンスを作成する例を示しています。

```
template<class T> class X {
  public:
    X* p;
    void f();
```

```
    void g();
};

X<int>* q;
X<int> r;
X<float>* s;
r.f();
s->g();
```

コンパイラーは、次のクラスおよび関数のインスタンス生成を必要とします。
- オブジェクト r が宣言されたときの X<int>
- メンバー関数呼び出し r.f() での X<int>::f()
- クラス・メンバー・アクセス関数呼び出し s->g() での X<float> および X<float>::g()

したがって、上記の例をコンパイルするには、関数 X<T>::f() および X<T>::g() を定義する必要があります。 (コンパイラーは、オブジェクト r を作成する場合、クラス X のデフォルトのコンストラクターを使用します。) コンパイラーは、次の定義のインスタンス生成を必要としません。
- ポインター p が宣言されたときの クラス X
- ポインター q が宣言されたときの X<int>
- ポインター s が宣言されたときの X<float>

コンパイラーが、ポインター型変換、またはメンバー型変換を指すポインターに関係する場合、それはクラス・テンプレート特殊化のインスタンスを暗黙的に生成します。次の例は、このことを示しています。

```
template<class T> class B { };
template<class T> class D : public B<T> { };

void g(D<double>* p, D<int>* q)
{
  B<double>* r = p;
  delete q;
}
```

割り当て B<double>* r = p は、型 D<double>* の p を B<double>* の型に変換します。コンパイラーは、D<double> のインスタンスを生成する必要があります。コンパイラーは、q の削除を試みる際に、D<int> のインスタンスを生成しなければなりません。

コンパイラーが、静的メンバーを含むクラス・テンプレートのインスタンスを暗黙的に生成する場合、それらの静的メンバーのインスタンスは、暗黙的には生成されません。コンパイラーは、静的メンバーの定義を必要とする場合のみ、静的メンバーのインスタンスを生成します。インスタンスを生成されたクラス・テンプレートは、いずれも静的メンバーのそれ自身のコピーを所有しています。次の例は、このことを示しています。

```
template<class T> class X {
  public:
   static T v;
};

template<class T> T X<T>::v = 0;

X<char*> a;
X<float> b;
X<float> c;
```

オブジェクト a には、型 char* の静的メンバー変数 v があります。オブジェクト b には、型 float の静的変数 v があります。オブジェクト b と c は、単一静的データ・メンバー v を共用します。

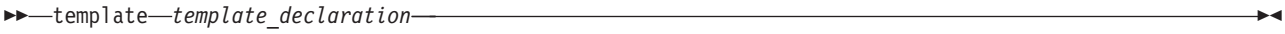
暗黙的にインスタンスを生成されたテンプレートは、テンプレートを定義した場所と同じネーム・スペースにあります。

関数テンプレート、またはメンバー関数テンプレート特殊化が、多重定義解決にかかわってくる場合、コンパイラーは、特殊化の宣言のインスタンスを暗黙的に生成します。

明示的テンプレート

C++ コンパイラーに、テンプレートから定義をいつ生成するのかを明示的に指示できます。これは、*明示的インスタンス化* と呼ばれます。

構文 - 明示的インスタンス化宣言



下記は、明示的インスタンス生成の例です。

```
template<class T> class Array { void mf(); };
template class Array<char>;      // explicit instantiation
template void Array<int>::mf();  // explicit instantiation

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); // explicit instantiation

namespace N {
   template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);                  // explicit instantiation

template <class T> class X {
   private:
      T v(T arg) { return arg; };
};

template int X<int>::v(int);    // explicit instantiation

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

テンプレート宣言は、テンプレートの明示的インスタンス生成のインスタンス化のポイントのスコープ内に存在する必要があります。テンプレート特殊化の明示的インスタンス生成は、テンプレートを定義した場所と同じネーム・スペースにあります。

アクセス検査規則は、明示的インスタンス生成には適用されません。明示的インスタンス生成の宣言にあるテンプレート引数および名前は、private 型、またはオブジェクトになります。上記の例では、コンパイラーは、メンバー関数が private で宣言されていても、`template int X<int>::v(int)` を許可します。

テンプレートのインスタンスを明示的に生成する場合、コンパイラーは、デフォルトの引数を使用しません。上記の例では、コンパイラーは、デフォルトの引数が型 **int** のアドレスであっても、明示的インスタンス化 `template char g(char)` を許可します。

次の例では、**extern** を使用したテンプレートのインスタンス化を示します。

```
template<class T>class C {
   static int i;
   void f(T) { }
};
template<class U>int C<U>::i = 0;
```

```
extern template C<int>;  // extern explicit template instantiation
C<int>c;   // does not cause instantiation of C<int>::i
           // or C<int>::f(int) in this file,
           // but the class is instantiated for mapping
C<char>d;  // normal instantiations

template<class C> C foo(C c) { return c; }
extern template int foo<int>(int);  // extern explicit template instantiation
int i = foo(1);  // does not cause instantiation of the body of foo<int>
```

テンプレート特殊化

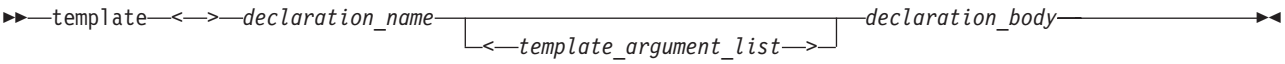
C++ 関数、クラス、またはクラスのメンバーの新規定義を、テンプレート宣言と 1 つ以上のテンプレート引数から作成する処理は、*テンプレートのインスタンス化* と呼ばれます。テンプレートのインスタンス化から作成された定義は、*特殊化* と呼ばれます。*主テンプレート* とは、特殊化しようとしているテンプレートのことです。

明示的特殊化

C++ テンプレート引数の特定のセットでテンプレートをインスタンス化する場合、コンパイラーは、それらのテンプレート引数に基づいて新規の定義を生成します。この定義生成の振る舞いをオーバーライドできます。その代わりに、コンパイラーがテンプレート引数の任意のセットで使用する定義を、指定することができます。これは、*明示的特殊化* と呼ばれます。次のものを明示的に特殊化できます。

- 関数またはクラス・テンプレート
- クラス・テンプレートのメンバー関数
- クラス・テンプレートの静的データ・メンバー
- クラス・テンプレートのメンバー・クラス
- クラス・テンプレートのメンバー関数テンプレート
- クラス・テンプレートのメンバー・クラス・テンプレート

構文 - 明示的特殊化宣言



template<> 接頭部は、次のテンプレート宣言が、テンプレート・パラメーターを取得しないということを示しています。 *declaration_name* は、以前宣言されたテンプレートの名前です。少なくとも特殊化が参照されているまでは、明示的特殊化を前もって宣言できること、その場合 *declaration_body* は、オプションであることに注意してください。

次の例は、明示的特殊化を示しています。

```
using namespace std;

template<class T = float, int i = 5> class A
{
  public:
      A();
      int value;
};

template<> class A<> { public: A(); };
template<> class A<double, 10> { public: A(); };

template<class T, int i> A<T, i>::A() : value(i) {
   cout << "Primary template, "
```

```
        << "non-type argument is " << value << endl;
}

A<>::A() {
   cout << "Explicit specialization "
        << "default arguments" << endl;
}

A<double, 10>::A() {
   cout << "Explicit specialization "
        << "<double, 10>" << endl;
}

int main() {
   A<int,6> x;
   A<> y;
   A<double, 10> z;
}
```

上記の例の出力は、以下のとおりです。

```
Primary template non-type argument is: 6
Explicit specialization default arguments
Explicit specialization <double, 10>
```

この例では、*主テンプレート* (特殊化しようとしているテンプレート) クラス `A` に対して、2 つの明示的特殊化を宣言しました。オブジェクト `x` は、主テンプレートのコンストラクターを使用します。オブジェクト `y` は、明示的特殊化 `A<>::A()` を使用します。オブジェクト `z` は、明示的特殊化 `A<double, 10>::A()` を使用します。

明示的特殊化の定義と宣言

C++ 明示的特殊化クラスの定義は、主テンプレートの定義とは無関係です。特殊化を定義するために、主テンプレートを定義する必要はありません (または、主テンプレートを定義するために、特殊化を定義する必要もありません)。例えば、コンパイラーは、次のコードを許可します。

```
template<class T> class A;
template<> class A<int>;

template<> class A<int> { /* ... */ };
```

主テンプレートは定義されませんが、明示的特殊化は定義されます。

宣言はされているが、不完全なクラスと同じように定義されていない、明示的特殊化の名前を使用できます。次の例は、このことを示しています。

```
template<class T> class X { };
template<>  class X<char>;
X<char>* p;
X<int> i;
// X<char> j;
```

コンパイラーは、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

C++ 主テンプレートの宣言は、明示的特殊化を*宣言するポイント* のあるスコープ内になければなりません。言い換えれば、明示的特殊化宣言は、主テンプレートの宣言の後に入れなければなりません。例えば、コンパイラーは次のことを許可しません。

```
template<> class A<int>;
template<class T> class A;
```

明示的特殊化は、主テンプレートの定義と同じネーム・スペースに存在します。

明示的特殊化のクラス・メンバー

C++ 明示的特殊化クラスのメンバーは、主テンプレートのメンバー宣言から暗黙的にインスタンス化されることはありません。クラス・テンプレート特殊化のメンバーを明示的に定義する必要があります。明示的特殊化テンプレート・クラスのメンバーを、**template<>** 接頭部を使用せずに、標準クラスのメンバーを定義するのと同じように定義します。さらに、明示的特殊化のメンバーをインラインに定義できます。ここでは、特別なテンプレート構文は使用されていません。次の例は、クラス・テンプレート特殊化を示しています。

```
template<class T> class A {
   public:
      void f(T);
};

template<> class A<int> {
   public:
      int g(int);
};

int A<int>::g(int arg) { return 0; }

int main() {
   A<int> a;
   a.g(1234);
}
```

明示的特殊化 `A<int>` は、メンバー関数 `g()` を含んでいますが、主テンプレートは、これを含んでいません。

テンプレート、メンバー・テンプレート、またはクラス・テンプレートのメンバーを明示的に特殊化する場合、この特殊化を宣言してから、特殊化のインスタンスを暗黙的に生成する必要があります。例えば、コンパイラーは次のコードを許可しません。

```
template<class T> class A { };

void f() { A<int> x; }
template<> class A<int> { };

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

コンパイラーは、関数 `f()` が、特殊化の前に、この特殊化 (`x` の構造体にある) を使用するので、明示的特殊化 `template<> class A<int> { };` を許可しません。

関数テンプレートの明示的特殊化

C++ 関数テンプレート特殊化では、コンパイラーが関数引数の型からテンプレート引数を推定できるのであれば、テンプレート引数はオプションです。次の例は、このことを示しています。

```
template<class T> class X { };
template<class T> void f(X<T>);
template<> void f(X<int>);
```

明示的特殊化 `template<> void f(X<int>)` は、 `template<> void f<int>(X<int>)` と同等です。

次に関する宣言および定義に対しては、デフォルトの関数引数は、指定できません。

- 関数テンプレートの明示的特殊化
- メンバー関数テンプレートの明示的特殊化

例えば、コンパイラーは次のコードを許可しません。

```
template<class T> void f(T a) { };
template<> void f<int>(int a = 5) { };

template<class T> class X {
  void f(T a) { }
};
template<> void X<int>::f(int a = 10) { };
```

クラス・テンプレートのメンバーの明示的特殊化

C++ インスタンス化された各クラス・テンプレート特殊化は、静的メンバーの専用コピーを所有します。静的メンバーを明示的に特殊化できます。次の例は、このことを示しています。

```
template<class T> class X {
public:
   static T v;
   static void f(T);
};

template<class T> T X<T>::v = 0;
template<class T> void X<T>::f(T arg) { v = arg; }

template<> char* X<char*>::v = "Hello";
template<> void X<float>::f(float arg) { v = arg * 2; }

int main() {
   X<char*> a, b;
   X<float> c;
   c.f(10);
}
```

このコードは、テンプレート引数 **char*** がストリング "Hello" を指すようにする、静的データ・メンバー X::v の初期化を明示的に特殊化します。関数 X::f() は、テンプレート引数 **float** に対して明示的に特殊化されます。オブジェクト a および b の静的データ・メンバー v は、同じストリング、つまり "Hello" を指します。 c.v の値は、関数呼び出し c.f(10) の後で 20 になります。

メンバー・テンプレートを、複数の囲みクラス・テンプレート内でネストできます。いくつかの囲みクラス・テンプレート内でネストされたテンプレートを明示的に特殊化する場合、特殊化するすべての囲みクラス・テンプレートに、その宣言の前に **template<>** を付ける必要があります。特殊化されていない囲みクラス・テンプレートも残すことはできますが、その囲みクラス・テンプレートを明示的に特殊化しない限り、ネスト・クラス・テンプレートを明示的に特殊化できません。次の例は、ネストされたメンバー・テンプレートの明示的特殊化を示しています。

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

template<class T> class X {
  public:
  template<class U> class Y {
  public:
    template<class V> void f(U,V);
    void g(U);
  };
};

template<class T> template<class U> template<class V>
  void X<T>::Y<U>::f(U, V) { cout << "Template 1" <<   endl; }

template<class T> template<class U>
  void X<T>::Y<U>::g(U) { cout << "Template 2" <<   endl; }
```

```
template<> template<>
  void X<int>::Y<int>::g(int) { cout << "Template 3"   << endl; }

template<> template<> template<class V>
  void X<int>::Y<int>::f(int, V) { cout << "Template 4" << endl; }

template<> template<> template<>
  void X<int>::Y<int>::f<int>(int, int) { cout << "Template 5" << endl; }

// template<> template<class U> template<class V>
//    void X<char>::Y<U>::f(U, V) { cout << "Template 6" << endl; }

// template<class T> template<>
//    void X<T>::Y<float>::g(float) { cout << "Template 7" << endl; }

int main() {
  X<int>::Y<int> a;
  X<char>::Y<char> b;
  a.f(1, 2);
  a.f(3, 'x');
  a.g(3);
  b.f('x', 'y');
  b.g('z');
}
```

下記は、上記のプログラムの出力です。

```
Template 5
Template 4
Template 3
Template 1
Template 2
```

- コンパイラーは、メンバー (関数 `f()`) を、それが含んでいるクラス (`Y`) を特殊化せずに特殊化しようとしているので、`"Template 6"` を出力するテンプレート特殊化定義を許可しません。
- コンパイラーは、クラス `Y` の囲みクラス (クラス `X`) が明示的に特殊化されていないので、`"Template 7"` を出力するテンプレート特殊化定義を許可しません。

フレンド宣言は、明示的特殊化を宣言できません。

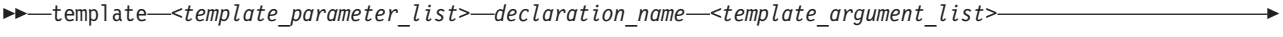
部分的特殊化

C++ クラス・テンプレートをインスタンス化する場合、コンパイラーは、受け渡したテンプレート引数に基づいて定義を作成します。代替として、それらすべてのテンプレート引数が、明示的特殊化のものと一致する場合、コンパイラーは、明示的特殊化が定義した定義を使用します。

部分的特殊化 は、明示的特殊化に汎用性を持たせたものです。明示的特殊化は、テンプレート引数リストだけを持っています。部分的特殊化は、テンプレート引数リストとテンプレート・パラメーター・リストの両方を持っています。コンパイラーは、テンプレート引数リストが、テンプレートのインスタンス生成のテンプレート引数のサブセットと一致する場合、部分的特殊化を使用します。そして、コンパイラーは、テンプレートのインスタンス生成の一致しない残りのテンプレート引数を使用して、部分的特殊化から新規定義を生成します。

関数テンプレートは、部分的に特殊化することはできません。

構文 - 部分的特殊化



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

C++ 主テンプレートは、テンプレート引数リストを持っていません。このリストは、テンプレート・パラメーター・リストに含まれています。

テンプレート・パラメーターを主テンプレートでは指定しているが、部分的特殊化で使用していなければ、それを部分的特殊化のテンプレート・パラメーター・リストから省略できます。部分的特殊化の引数リストの順序は、主テンプレートの暗黙の引数リストの順序と同じです。

部分的テンプレート・パラメーターのテンプレート引数リストでは、式が ID のみとなる場合を除いて、非型引数を含む式を持つことはできません。次の例では、コンパイラーは、最初の部分的特殊化を許可しませんが、2 番目のものは許可します。

```
template<int I, int J> class X { };

// Invalid partial specialization
template<int I> class <I * 4, I + 3> { };

// Valid partial specialization
template <int I> class <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・テンプレートの部分的特殊化のマッチング

C++ コンパイラーは、クラス・テンプレート特殊化のテンプレート引数と、主テンプレートおよび部分的特殊化のテンプレート引数リストを突き合わせて、主テンプレートを使用するのか、その部分的特殊化の 1 つを使用するのかを判別します。

- コンパイラーが特殊化を 1 つだけ検出する場合、コンパイラーは、その特殊化から定義を生成します。
- コンパイラーが複数の特殊化を検出する場合、コンパイラーは、どの特殊化が最も特殊化されているのかを判別します。 X からの特殊化と一致する引数リストは、いずれも Y からの特殊化と一致するが、その逆では一致しないという場合は、テンプレート X は、テンプレート Y よりもさらに特殊化されています。コンパイラーが最も特殊化された特殊化を検出できない場合は、クラス・テンプレートの使用は、あいまいになります。つまり、コンパイラーは、プログラムを許可しません。
- コンパイラーがどのような一致も検出しない場合、コンパイラーは、主テンプレートから定義を生成します。

名前のバインディングおよび従属名

C++ *名前のバインディング* は、テンプレートで明示的に、または暗黙的に使用されている名前ごとに宣言を検出する処理です。コンパイラーは、テンプレートの定義で名前をバインドしたり、またはテンプレートのインスタンス生成において名前をバインドします。

従属名 は、テンプレート・パラメーターの型、または値に依存する名前です。次に例を示します。

```
template<class T> class U : A<T>
{
  typename T::B x;
  void f(A<T>& y)
```

```
  {
    *y++;
  }
};
```

この例では、従属名は、基底クラス A<T>、型名 T::B、および変数 y です。

テンプレートのインスタンスが作成されると、コンパイラーは、従属名をバインドします。テンプレートが定義されると、コンパイラーは、非従属名をバインドします。次に例を示します。

```
void f(double) { cout << "Function f(double)" << endl; }

template<class T> void g(T a) {
  f(123);
  h(a);
}

void f(int) { cout << "Function f(int)" << endl; }
void h(double) { cout << "Function h(double)" << endl; }

void i() {
  extern void h(int);
  g<int>(234);
}

void h(int) { cout << "Function h(int)" << endl; }
```

関数 i() を呼び出す場合、以下が出力されます。

```
Function f(double)
Function h(double)
```

テンプレートの*定義のポイント* は、それの定義の直前に配置されます。この例では、関数テンプレート g(T) の定義のポイントは、キーワード **template** の直前に配置されます。関数呼び出し f(123) は、テンプレート引数に依存しないので、コンパイラーは、関数テンプレート g(T) の定義の前に宣言された名前を考慮します。したがって、呼び出し f(123) は、f(double) を呼び出します。 f(int) のほうが的確ですが、それは、g(T) の定義のポイントのあるスコープ内に存在していません。

テンプレートの*インスタンス生成のポイント* は、その使用を囲む宣言の直前に配置されます。この例では、g<int>(234) 呼び出しのインスタンス生成のポイントは、i() の直前に配置されます。関数呼び出し h(a) がテンプレート引数に依存するので、コンパイラーは、関数テンプレート g(T) のインスタンス生成の前に、宣言された名前を考慮します。したがって、h(a) の呼び出しは、h(double) を呼び出します。この関数が g<int>(234) のインスタンス生成のポイントのあるスコープ内に存在しないので、コンパイラーは、h(int) を考慮しません。

インスタンス生成バインディングのポイントは、次のことを意味しています。
- テンプレート・パラメーターは、ローカル名、またはクラス・メンバーに依存できない。
- テンプレートの修飾名は、ローカル名、またはクラス・メンバーに依存できない。

typename キーワード

C++ 型を参照する修飾名やテンプレート・パラメーターに依存する修飾名がある場合は、キーワード **typename** を使用してください。キーワード **typename** のみを、テンプレート宣言または定義で使用してください。次の例は、キーワード **typename** の使用法を示しています。

```
template<class T> class A
{
  T::x(y);
  typedef char C;
  A::C d;
}
```

ステートメント `T::x(y)` は、あいまいです。そのステートメントは、非ローカル引数 `y` を使用した関数 `x()` の呼び出しや、または、型 `T::x` を使用して変数 `y` の宣言にすることができます。 C++ は、このステートメントを関数呼び出しとして解釈します。コンパイラーにこのステートメントを宣言として解釈させるには、キーワード **typename** をそのステートメントの開始位置に追加します。ステートメント `A::C d;` は、不適格です。クラス `A` は、`A<T>` も参照するので、テンプレート・パラメーターに依存します。キーワード **typename** をこの宣言の開始位置に追加する必要があります。

```
  typename A::C d;
```

テンプレート・パラメーター宣言で、キーワード **class** の代わりに、キーワード **typename** も使用できます。

修飾子としてのキーワード・テンプレート

C++ メンバー・テンプレートと他の名前を区別するために、キーワード **template** を修飾子として使用してください。次の例は、**template** を修飾子として使用しなければならない状況を示しています。

```
class A
{
  public:
    template<class T> T function_m() { };
};

template<class U> void function_n(U argument)
{
  char object_x = argument.function_m<char>();
}
```

宣言 `char object_x = argument.function_m<char>;` は、不適格です。コンパイラーは、`<` が「より小演算子」であると見なします。コンパイラーに関数テンプレート呼び出しを認識させるには、**template** 修飾子を追加する必要があります。

```
char object_x = argument.template function_m<char>();
```

メンバー・テンプレート特殊化の名前が、`.`、`->`、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);
}
```

上記の例の出力は、以下のとおりです。

```
Primary: 100
Specialized, non-type argument = 20
member template's member function: 40
```

これらのケースでキーワード **template** を使用しない場合、コンパイラーは、< を「より小演算子」として解釈します。例えば、次のコード行は、不適格です。

```
p->f<100>();
```

コンパイラーは、f を非テンプレート・メンバーとして、 < を「より小演算子」として解釈します。

第 17 章 例外処理

C++ *例外処理* は、例外的な状況を検出したり、処理するコードをプログラムの他の部分から分離するメカニズムです。例外的な状況は、必ずしもエラーではないことに注意してください。

関数が例外的な状況を検出する場合、オブジェクトでこれを表します。このオブジェクトを*例外オブジェクト* と呼びます。 例外的な状況を処理するには、*例外をスロー*します。これによって、例外をスローした関数を直接的または間接的に呼び出した元のコードの指定ブロックに、制御だけでなく例外も渡されます。コードのこのブロックは、*ハンドラー* と呼ばれます。 処理させる例外のタイプを、ハンドラーに指定します。 C++ ランタイムは、生成コードとともに、スローされた例外を処理できる最初の適切なハンドラーに制御を渡します。これが起きる場合、例外はキャッチ されました。ハンドラーは、別のハンドラーが例外をキャッチできるように、それを*再スロー* します。

400 i5/OS 固有の使用の詳細については、「*ILE C/C++ Programmer's Guide*」の第 21 章『Handling Exceptions in a Program』を参照してください。

例外処理のメカニズムは、次の要素で構成されます。
- **try** ブロック: 特別なプロセスで処理したい例外をスローするコードのブロック
- **catch** ブロックまたはハンドラー: try ブロックが例外に遭遇するときに実行されるコードのブロック
- **throw** 式: プログラムが例外に遭遇するときを示す
- 例外指定: 関数が、どの例外 (例外があれば) をスローするのかを指定する
- `unexpected()` 関数: 例外指定に指定されていない例外を、関数がスローした場合に呼び出される
- `terminate()` 関数: キャッチされない例外に対して呼び出される

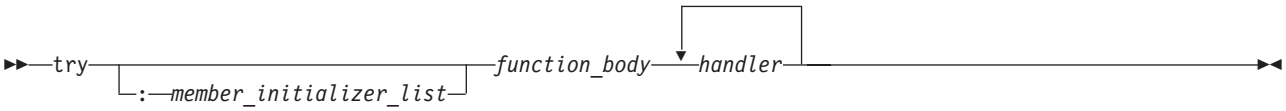
関連参照
- 『try キーワード』
- 331 ページの『catch ブロック』
- 337 ページの『throw 式』
- 340 ページの『例外の指定』
- 343 ページの『unexpected()』
- 344 ページの『terminate()』

try キーワード

C++ *try ブロック* を使用して、すぐに処理する例外をスローする可能性のあるプログラムのエリアを示します。*関数 try ブロック* を使用して、関数本体の全体で例外を検出することを指示します。

構文 - try ブロック

構文 — 関数 try ブロック



以下は、メンバー初期化指定子、関数 try ブロック、および try ブロックを持つ関数 try ブロックの例です。

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

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { }
};

class A {
   public:
      int i;

      // A function try block with a member
      // initializer
      A() try : i(0) {
         throw E("Exception thrown in A()");
      }
      catch (E& e) {
         cout << e.error << endl;
      }
};

// A function try block
void f() try {
   throw E("Exception thrown in f()");
}
catch (E& e) {
   cout << e.error << endl;
}

void g() {
   throw E("Exception thrown in g()");
}

int main() {
   f();

   // A try block
   try {
      g();
   }
   catch (E& e) {
      cout << e.error << endl;
   }
   A x;
}
```

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。

関連参照
- 276 ページの『規定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

C++ try ブロックがネストされており、内側の try ブロックによって呼び出された関数内で **throw** が生じる場合は、制御は、引数が throw 式の引数と一致する最初の catch ブロックが見つかるまで、ネストされた try ブロック間を外側に向けて渡されて行きます。

次に例を示します。

```
try
{
      func1();
      try
      {
            func2();
      }
      catch (spec_err) { /* ... */ }
      func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

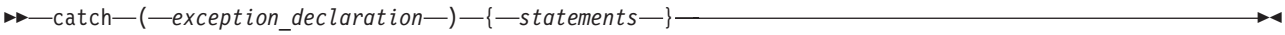
上記の例で、spec_err が内側の try ブロック (この場合は、func2() から) 内でスローされる場合、内側の catch ブロックがこの例外をキャッチします。この catch ブロックが制御権移動を行わない場合は、func3() が呼び出されます。内側の try ブロックの後で spec_err がスローされた場合 (例えば func3() によって)、この例外はキャッチされず、関数 terminate() が呼び出されます。内側の try ブロックの中の func2() からスローされた例外が、type_err である場合、プログラムは、func3() を呼び出さずに、両方の try ブロックから出て 2 番目の catch ブロックにスキップします。内側の try ブロックの後には、適切な catch ブロックがないためです。

catch ブロック内で try ブロックをネストすることもできます。

関連参照
- 344 ページの『terminate()』
- 343 ページの『unexpected()』
- 343 ページの『特殊な例外処理関数』

catch ブロック

C++ 以下は、例外ハンドラー、または catch ブロック の構文です。



ハンドラーが、多くの型の例外をキャッチできるように宣言することができます。関数がキャッチできる許容オブジェクトは、**catch** キーワードの後に続く括弧の中 (*exception_declaration*) に宣言します。 基本的な型のオブジェクト、基底および派生クラス・オブジェクト、参照、およびこれらすべての型を指すポインタ

ーをキャッチすることができます。 **const** 型と **volatile** 型もキャッチすることができます。 *exception_declaration* を、非完了型、または以下を除く非完了型を指す参照、またはポインターにすることはできません。

- **void***
- **const void***
- **volatile void***
- **const volatile void***

exception_declaration では、型を定義できません。

catch(...) 形式のハンドラーを使用して、前の catch ブロックでキャッチされなかった、スローされた例外をすべてキャッチすることができます。catch 引数の中の省略符号は、このハンドラーが、スローされたどの例外も処理できることを示しています。

catch(...) ブロックによって例外がキャッチされた場合には、スローされたオブジェクトにアクセスする直接的方法はありません。 **catch(...)** によって、キャッチされた例外に関する情報は、非常に限られています。

catch ブロック内にあるスローされたオブジェクトにアクセスしたい場合は、オプションの変数名を宣言することができます。

catch ブロックはアクセス可能オブジェクトしかキャッチできません。キャッチされたオブジェクトには、アクセス可能なコピー・コンストラクターがあるはずです。

関連参照

- 60 ページの『型修飾子』
- 237 ページの『メンバー・アクセス』

関数 try ブロック・ハンドラー

C++ 関数またはコンストラクターのパラメーターのスコープおよび存続期間が、関数 try ブロックのハンドラーにまで適用されます。次の例は、このことを示しています。

```
void f(int &x) try {
   throw 10;
}
catch (const int &i)
{
   x = i;
}

int main() {
   int v = 0;
   f(v);
}
```

`f()` が呼び出された後は、`v` の値は、`10` になります。

`main()` の関数 try ブロックは、静的ストレージ期間を持つオブジェクトのデストラクター、またはネーム・スペース・スコープ・オブジェクトのコンストラクターで、スローされる例外をキャッチしません。

次の例は、静的オブジェクトのデストラクターから例外をスローします。

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

class E {
```

```
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};

class A {
  public: ~A() { throw E("Exception in ~A()"); }
};

class B {
  public: ~B() { throw E("Exception in ~B()"); }
};

int main() try {
  cout << "In main" << endl;
  static A cow;
  B bull;
}
catch (E& e) {
  cout << e.error << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In main
Exception in ~B()
```

ランタイムは、オブジェクト cow が、プログラムの終わりに破棄される時にスローされる例外をキャッチできません。

次の例は、ネーム・スペース・スコープ・オブジェクトのコンストラクターから例外をスローします。

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

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { }
};

namespace N {
  class C {
  public:
    C() {
      cout << "In C()" << endl;
      throw E("Exception in C()");
    }
  };

  C calf;
};

int main() try {
  cout << "In main" << endl;
}
catch (E& e) {
  cout << e.error << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In C()
```

コンパイラーは、オブジェクト calf の作成時にスローされる例外をキャッチできません。

関数 try ブロックのハンドラーでは、コンストラクター本体、またはデストラクター本体にジャンプすることはできません。

リターン・ステートメントは、コンストラクターの関数 try ブロック・ハンドラー内に置くことはできません。

オブジェクトのコンストラクター、またはデストラクターの関数 try ブロック・ハンドラーに入ると、そのオブジェクトの完全な構成の基底クラスおよびメンバーは破棄されます。次の例は、このことを示しています。

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

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
};

class B {
   public:
      B() { };
      ~B() { cout << "~B() called" << endl; };
};

class D : public B {
   public:
      D();
      ~D() { cout << "~D() called" << endl; };
};

D::D() try : B() {
   throw E("Exception in D()");
}
catch(E& e) {
   cout << "Handler of function try block of D(): " << e.error << endl;
};

int main() {
   D val;
};
```

上記の例の出力は、以下のとおりです。

```
~B() called
Handler of function try block of D(): Exception in D()
```

D() の関数 try ブロックのハンドラーに入ると、ランタイムは、まず最初に D の基底クラスのデストラクター、つまり B を呼び出します。 val が完全に構成されていないので、D のデストラクターは呼び出されません。

ランタイムは、コンストラクターまたはデストラクターの関数 try ブロックのハンドラーの終了時に、例外を再スローします。その他のすべての関数は、関数 try ブロックのハンドラーの終了に到達した時点でリターンします。次の例は、このことを示しています。

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

class E {
   public:
      const char* error;
      E(const char* arg) : error(arg) { };
```

```
};

class A {
   public:
      A() try { throw E("Exception in A()"); }
      catch(E& e) { cout << "Handler in A(): " << e.error << endl; }
};

int f() try {
   throw E("Exception in f()");
   return 0;
}
catch(E& e) {
   cout << "Handler in f(): " << e.error << endl;
   return 1;
}

int main() {
   int i = 0;
   try { A cow; }
   catch(E& e) {
      cout << "Handler in main(): " << e.error << endl;
   }

   try { i = f(); }
   catch(E& e) {
      cout << "Another handler in main(): " << e.error << endl;
   }

   cout << "Returned value of f(): " << i << endl;
}
```

上記の例の出力は、以下のとおりです。

```
Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1
```

関連参照

- 145 ページの『main() 関数』
- 33 ページの『static ストレージ・クラス指定子』
- 191 ページの『第 10 章 ネーム・スペース』
- 280 ページの『デストラクター』

catch ブロックの引数

C++ catch ブロックの引数に対してクラス型を指定する場合 (*exception_declaration*)、コンパイラーはコピー・コンストラクターを使用して、その引数を初期化します。その引数に名前が入っていなければ、コンパイラーは、一時オブジェクトを初期化し、ハンドラーが終了するとき、それを破棄します。

冗長と思われる場合、ISO C++ 仕様に一時オブジェクトを作成するコンパイラーは不要です。

関連参照

- 287 ページの『一時オブジェクト』

スローされた例外とキャッチされた例外とのマッチング

C++ ハンドラーの catch 引数の中の引数は、次のいずれかの条件が満たされる場合、throw 式 (throw 引数) の *assignment_expression* の引数と一致します。

- catch 引数の型が、スローされたオブジェクトの型と一致する。
- catch 引数が、スローされたクラス・オブジェクトのパブリック基底クラスである。
- catch がポインターの型を指定し、スローされたオブジェクトが、標準ポインター型変換によって catch 引数のポインター型に変換できるポインター型である。

注: スローされたオブジェクトの型が **const** または **volatile** である場合、一致するには、catch 引数も **const** または **volatile** であることが必要です。ただし、**const**、**volatile**、または参照型の catch 引数が、非定数、非 volatile、または非参照オブジェクト型と一致することがあります。非参照 catch 引数型は、同じ型のオブジェクトへの参照と一致します。

関連参照

- 128 ページの『ポインター型変換』
- 60 ページの『型修飾子』
- 77 ページの『参照』
- 343 ページの『特殊な例外処理関数』

キャッチの順序

C++ コンパイラーが try ブロックで例外を検出すると、その出現の順に各ハンドラーを試行します

基底クラスのオブジェクト用の catch ブロックが、その基底クラスから派生するクラスのオブジェクト用の catch ブロックより前にある場合は、コンパイラーは警告を発行し、派生クラス・ハンドラー内に到達不能コードがあっても、プログラムのコンパイルを続行します。

catch(...) の書式の catch ブロックは、try ブロックの後に続く最後の catch ブロックでなければならず、そうでなければエラーが起こります。このように配置することによって、**catch(...)** ブロックによって、さらに特定の catch ブロックが、本来キャッチすることになっている例外をキャッチすることを防ぐことができなくなります。

ランタイムが、現行スコープ内に一致するハンドラーを検出できない場合、ランタイムは、動的な囲み try ブロック内で一致するハンドラーの検出を継続します。次の例は、このことを示しています。

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

class E {
public:
  const char* error;
  E(const char* arg) : error(arg) { };
};

class F : public E {
public:
  F(const char* arg) : E(arg) { };
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    throw E("Class E exception");
  }
  catch (F& e) {
    cout << "In handler of f()";
    cout << e.error << endl;
  }
};
```

```
int main() {
  try {
    cout << "In main" << endl;
    f();
  }
  catch (E& e) {
    cout << "In handler of main: ";
    cout << e.error << endl;
  };
  cout << "Resume execution in main" << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In main
In try block of f()
In handler of main: Class E exception
Resume execution in main
```

関数 f() では、ランタイムは、スローされた型 E の例外を処理するハンドラーを検出できません。ランタイムは、動的な囲み try ブロック内、つまり main() 関数の try ブロック内で、一致するハンドラーを検出します。

ランタイムが、プログラムで一致するハンドラーを検出できない場合、terminate() 関数を呼び出します。

関連参照
- 329 ページの『try キーワード』
- 344 ページの『terminate()』

---

throw 式

C++ *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。

構文 - throw 式

assignment_expression の型は、非完了型、または以下を除く非完了型を指すポインターにすることはできません。
- **void***
- **const void***
- **volatile void***
- **const volatile void***

assignment_expression は、呼び出しでの関数引数、またはリターン・ステートメントのオペランドと同じように扱われます。

assignment_expression が、クラス・オブジェクトの場合、そのオブジェクトのコピー・コンストラクターおよびデストラクターは、アクセス可能でなければなりません。例えば、プライベートとして宣言されたコピー・コンストラクターを持つクラス・オブジェクトをスローすることはできません。

関連参照
- 64 ページの『不完全型』

例外の rethrow

C++ catch ブロックが、キャッチした特定の例外を処理できない場合、その例外を再スローする (rethrow) ことができます。 rethrow 式 (*assignment_expression* がない **throw**) は、最初にスローされたオブジェクトを再びスローします。

例外が、rethrow 式が発生するスコープですでにキャッチされているので、その例外は、次の動的な囲み try ブロックへ再びスローされます。したがって、rethrow 式の発生したスコープの catch ブロックは、その例外を処理できなくなります。動的な囲み try ブロックの catch ブロックは、いずれも、例外をキャッチする機会があります。

次の例は、例外の再スローを示しています。

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

struct E {
  const char* message;
  E() : message("Class E") { }
};

struct E1 : E {
  const char* message;
  E1() : message("Class E1") { }
};

struct E2 : E {
  const char* message;
  E2() : message("Class E2") { }
};

void f() {
  try {
    cout << "In try block of f()" << endl;
    cout << "Throwing exception of type E1" << endl;
    E1 myException;
    throw myException;
  }
  catch (E2& e) {
    cout << "In handler of f(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E1& e) {
    cout << "In handler of f(), catch (E1& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
  catch (E& e) {
    cout << "In handler of f(), catch (E& e)" << endl;
    cout << "Exception: " << e.message << endl;
    throw;
  }
}

int main() {
  try {
    cout << "In try block of main()" << endl;
    f();
  }
  catch (E2& e) {
    cout << "In handler of main(), catch (E2& e)" << endl;
    cout << "Exception: " << e.message << endl;
  }
```

```
  catch (...) {
    cout << "In handler of main(), catch (...)" << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In try block of main()
In try block of f()
Throwing exception of type E1
In handler of f(), catch (E1& e)
Exception: Class E1
In handler of main(), catch (...)
```

main() 関数の try ブロックは、関数 f() を呼び出します。関数 f() の try ブロックは、myException という名前の、型 E1 のオブジェクトをスローします。ハンドラー catch (E1 &e) が、myException キャッチします。それからハンドラーは、ステートメント throw を使用して、myException を、次の動的な囲み try ブロック、つまり main() 関数の try ブロックに再びスローします。ハンドラー catch(...) が、myException をキャッチします。

関連参照

- 337 ページの『throw 式』

スタック・アンワインド

C++ 例外がスローされ、制御が try ブロックからハンドラーに渡されると、C++ ランタイムは、 try ブロックの開始以降に作成されたすべての自動オブジェクトに対して、デストラクターを呼び出します。この処理は、*スタック・アンワインド* と呼ばれます。 自動オブジェクトは、その作成の逆順で破棄されます。 (自動オブジェクトは、**auto** または **register** と宣言されているか、あるいは **static** または **extern** と宣言されていない、ローカル・オブジェクトです。自動オブジェクト x は、x が宣言されているブロックのプログラムの終了時に、必ず削除されます。)

例外が、サブオブジェクトまたは配列エレメントを含むオブジェクトの作成中にスローされる場合、デストラクターは、例外がスローされる前に正常に作成されたサブオブジェクトまたは配列エレメントに対してのみ、呼び出されます。ローカル静的オブジェクトに対するデストラクターは、オブジェクトが正常に作成された場合にのみ呼び出されます。

スタック・アンワインド中にデストラクターが例外をスローし、その例外が処理されない場合、terminate() 関数が呼び出されます。次の例は、このことを示しています。

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

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_terminate() {
  cout << "Call to my_terminate" << endl;
};

struct A {
  A() { cout << "In constructor of A" << endl; }
  ~A() {
    cout << "In destructor of A" << endl;
    throw E("Exception thrown in ~A()");
  }
```

```
};

struct B {
  B() { cout << "In constructor of B" << endl; }
  ~B() { cout << "In destructor of B" << endl; }
};

int main() {
  set_terminate(my_terminate);

  try {
    cout << "In try block" << endl;
    A a;
    B b;
    throw("Exception thrown in try block of main()");
  }
  catch (const char* e) {
    cout << "Exception: " << e << endl;
  }
  catch (...) {
    cout << "Some exception caught in main()" << endl;
  }

  cout << "Resume execution of main()" << endl;
}
```

上記の例の出力は、以下のとおりです。

```
In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate
```

try ブロックでは、2 つの自動オブジェクト、a と b が作成されます。 try ブロックは、型 **const char*** の例外をスローします。ハンドラー catch (const char* e) が、この例外をキャッチします。 C++ ランタイムは、スタックをアンワインドし、a および b のデストラクターを、それらが構築された逆順で呼び出します。 a のデストラクターが、例外をスローします。プログラムにこの例外を処理できるハンドラーが存在しないので、C++ ランタイムは terminate() を呼び出します。 (関数 terminate() は、set_terminate() に引数として指定した関数を呼び出します。この例では、terminate() は、my_terminate() を呼び出すよう指定されています。)

関連参照

- 344 ページの『terminate()』
- 345 ページの『set_unexpected() と set_terminate()』

例外の指定

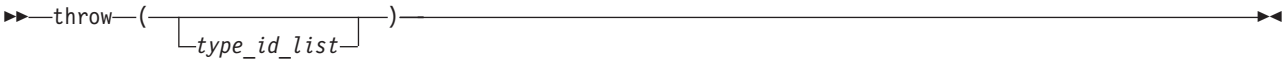
C++ C++ には、特定の関数が、指定されたリストの例外だけをスローするように限定するメカニズムがあります。任意の関数の先頭に例外を指定すると、関数の呼び出し元に対して、その関数が例外の指定に含まれていない例外を直接にも間接にもスローしないことを保証することができます。

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。

構文 - 例外指定



type_id_list は、コンマで区切られた型のリストです。このリストでは、オプションで **const** または **volatile**、あるいはその両方で修飾した、非完了型、および非完了型を指すポインターまたは参照 (**void** を指すポインターを除く) を指定することはできません。例外指定では、型を定義できません。

例外指定のない関数は、すべての例外のスローを認めます。空の *type_id_list* を持つ例外指定を使用する関数、**throw()** は、例外のスローを許可しません。

例外指定は関数の型の一部ではありません。

例外指定は、関数、関数を指すポインター、関数への参照、メンバー関数宣言を指すポインター、またはメンバー関数定義を指すポインターの関数宣言子の終了にのみ現れます。例外指定を typedef 宣言に入れることはできません。次の宣言は、このことを示しています。

```
void f() throw(int);
void (*g)() throw(int);
void h(void i() throw(int));
// typedef int (*j)() throw(int);  This is an error.
```

コンパイラーは、最後の宣言、`typedef int (*j)() throw(int)` を許可しません。

クラス `A` が、関数の例外指定の *type_id_list* に入っている型の一つだとします。その関数は、クラス `A` またはクラス `A` から public に派生したクラスの例外オブジェクトをスローします。次の例は、このことを示しています。

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

void f(int i) throw (A) {
   switch (i) {
      case 0: throw A();
      case 1: throw B();
      default: throw C();
   }
}

void g(int i) throw (A*) {
   A* a = new A();
   B* b = new B();
   C* c = new C();
   switch (i) {
      case 0: throw a;
      case 1: throw b;
      default: throw c;
   }
}
```

関数 `f()` は、型 `A` または `B` のオブジェクトをスローできます。関数が型 `C` のオブジェクトをスローしようとする場合、コンパイラーは、`C` が関数の例外指定に指定されてもいないし、それが `A` から public に派生したものでもないので、`unexpected()` を呼び出します。同様に、関数 `g()` は、型 `C` のオブジェクトを指すポインターをスローできません。関数は、型 `A` のポインター、または `A` から public に派生するオブジェクトのポインターをスローします。

例外の指定

仮想関数をオーバーライドする関数は、その仮想関数が指定する例外のみをスローすることができます。次の例は、このことを示しています。

```
class A {
   public:
      virtual void f() throw (int, char);
};

class B : public A{
   public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
   class C : public A {
      public: void f() { }
   };

   class D : public A {
      public: void f() throw (int, char, double) { }
   };
*/
```

コンパイラーは、メンバー関数が、型 **int** の例外のみをスローするので、`B::f()` を認めます。コンパイラーは、メンバー関数が、どの種類の例外もスローするので、`C::f()` を許可しません。コンパイラーは、メンバー関数が、`A::f()` よりも多くの型の例外 (**int**、**char**、および **double**) をスローするので、`D::f()` を許可しません。

`x` という名前の関数を指すポインター、または `y` という名前の関数を指すポインターの割り当て、または初期化を行うとします。関数 `x` を指すポインターは、`y` の例外指定が指定する例外のみをスローできます。次の例は、このことを示しています。

```
void (*f)();
void (*g)();
void (*h)() throw (int);

void i() {
   f = h;
//   h = g;  This is an error.
}
```

コンパイラーは、`f` がどのような種類の例外もスローできるので、割り当て `f = h` を認めます。 `g` は、どのような種類の例外もスローできますが、`h` が、型 **int** のオブジェクトしかスローできないので、コンパイラーは、割り当て `h = g` を許可しません。

暗黙的に宣言された特殊メンバー関数 (デフォルトのコンストラクター、コピー・コンストラクター、デストラクター、およびコピー代入演算子) には、例外指定があります。暗黙的に宣言された特殊メンバー関数の例外指定の中には、その特殊関数が起動する対象の関数の例外指定の中で宣言されている型が含められます。特別な関数を起動する関数が、例外をすべて許可する場合は、特別な関数も例外をすべて許可します。特別な関数が呼び出す関数のすべてが、例外を許可しない場合は、特別な関数も例外を許可しません。次の例は、このことを示しています。

```
class A {
   public:
      A() throw (int);
      A(const A&) throw (float);
      ~A() throw();
};

class B {
   public:
```

```
      B() throw (char);
      B(const A&);
      ~B() throw();
};

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

上記の例で示した次の特別な関数は、暗黙的に宣言されています。

```
C::C() throw (int, char);
C::C(const C&);   // Can throw any type of exception, including float
C::~C() throw();
```

デフォルトの C のコンストラクターは、型 **int** または **char** の例外をスローできます。 C のコピー・コンストラクターは、どのような種類の例外もスローできます。 C のデストラクターは、いかなる例外もスローできません。

関連参照

- 64 ページの『不完全型』
- 136 ページの『関数宣言』
- 155 ページの『関数へのポインター』
- 271 ページの『第 15 章 特殊なメンバー関数』
- 『unexpected()』

特殊な例外処理関数

C++ スローされたすべてのエラーが catch ブロックによってキャッチされ、正常に処理されるというわけではありません。ある状況においては、例外を処理する最良の方法は、プログラムを終了することです。 C++ には、catch ブロックによって正しく処理できない例外、または有効な try ブロックの外部にスローされる例外の処理のために、 2 つの特殊なライブラリー関数がインプリメントされています。これらの関数は、unexpected() および terminate() です。

unexpected()

C++ 例外指定を持つ関数が、例外指定にリストされていない例外をスローすると、C++ ランタイムは、以下を行います。

1. unexpected() 関数が呼び出されます。
2. unexpected() 関数は、unexpected_handler によって指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。

terminate()

C++ 例外処理メカニズムが正しく機能せず、void terminate() の呼び出しが起きるケースもあります。この terminate() の呼び出しは、次のいずれかの状況で行われます。

- 例外処理メカニズムが、スローされた例外のハンドラーを検出できません。以下に、上記のさらに詳しい事例を示します。
  - スタック・アンワインド中に、デストラクターが例外をスローし、その例外が処理されません。
  - スローされた例外が、また例外をスローし、その例外が処理されません。
  - 非ローカル静的オブジェクトのコンストラクターまたはデストラクターが例外をスローし、その例外が処理されません。
  - atexit() で登録された関数が例外をスローし、その例外が処理されません。以下に、このことを示します。

```
extern "C" printf(char* ...);
#include <exception>
#include <cstdlib>
using namespace std;

void f() {
  printf("Function f()¥n");
  throw "Exception thrown from f()";
```

```
}

void g() { printf("Function g()¥n"); }
void h() { printf("Function h()¥n"); }

void my_terminate() {
  printf("Call to my_terminate¥n");
  abort();
}

int main() {
  set_terminate(my_terminate);
  atexit(f);
  atexit(g);
  atexit(h);
  printf("In main¥n");
}
```

上記の例の出力は、以下のとおりです。

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

atexit() で関数を登録するには、登録したい関数を指すポインターに、atexit() へのパラメーターを渡します。標準プログラム終了処理で atexit() は、引数のない登録済み関数を逆順で呼び出します。 atexit() 関数は、<cstdlib> ライブラリーに入っています。

- オペランドを指定しない throw 式が、例外を再びスローしようとし、現在処理されている例外がありません。
- 関数 f() が、その例外指定に違反する例外をスローします。 unexpected() 関数が、f() の例外指定に違反する例外をスローし、f() の例外指定が、クラス std::bad_exception を含んでいませんでした。
- unexpected_handler のデフォルト値が呼び出されます。

terminate() 関数は、terminate_handler 関数によって指定された関数を呼び出します。デフォルトで terminate_handler は、プログラムから終了する関数 abort() を指します。 terminate_handler のデフォルト値を、関数 set_terminate() に置き換えることができます。

終了関数は、return を使用するか例外をスローすることによって呼び出し元に戻ることはできません。

set_unexpected() と set_terminate()

C++ 関数 unexpected() は、起動されたときに、set_unexpected() に、最後に引数として渡された関数を呼び出します。 set_unexpected() がまだ呼び出されていない場合、unexpected() は terminate() を呼び出します。

関数 terminate() は、起動されると、set_terminate() に、一番最近に引数として供給された関数を呼び出します。 set_terminate() がまだ呼び出されていない場合、terminate() は abort() を呼び出し、これによってプログラムが終了します。

set_unexpected() および set_terminate() を使用して、unexpected() および terminate() によって呼び出される、ユーザー定義関数を登録することができます。 set_unexpected() と set_terminate() は、標準ヘッダー・ファイルに入っています。これらの各関数は、その戻りの型およびその引数の型として、戻りの型が **void** で引数なしの関数を指すポインターを持っています。引数として提供する関数を指すポインタ

ーは、対応する特殊な関数によって呼び出される関数になります。 set_unexpected() への引数が unexpected() によって呼び出される関数になり、set_terminate() への引数が terminate() によって呼び出される関数になります。

set_unexpected() および set_terminate() は、前にそれぞれの特殊な関数 (unexpected() および terminate()) によって呼び出された関数を指すポインターを戻します。戻り値を保管することによって、後で元の特殊な関数を復元して、unexpected() と terminate() が、再び terminate() と abort() を呼び出すようにすることができます。

set_terminate() を使用して、ユーザー自身の関数を登録する場合、関数は、呼び出し元にはリターンせず、プログラムの実行を終了する必要があります。

例外処理関数の使用例

C++ 次の例は、制御の流れと、例外処理で使用する特殊な関数を示しています。

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

class X { };
class Y { };
class A { };

// pfv type is pointer to function returning void
typedef void (*pfv)();

void my_terminate() {
  cout << "Call to my terminate" << endl;
  abort();
}

void my_unexpected() {
  cout << "Call to my_unexpected()" << endl;
  throw;
}

void f() throw(X,Y, bad_exception) {
  throw A();
}

void g() throw(X,Y) {
  throw A();
}

int main()
{
  pfv old_term = set_terminate(my_terminate);
  pfv old_unex = set_unexpected(my_unexpected);
  try {
    cout << "In first try block" << endl;
    f();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e1) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
```

```
    cout << "Caught some exception" << endl;
  }

  cout << endl;

  try {
    cout << "In second try block" << endl;
    g();
  }
  catch(X) {
    cout << "Caught X" << endl;
  }
  catch(Y) {
    cout << "Caught Y" << endl;
  }
  catch (bad_exception& e2) {
    cout << "Caught bad_exception" << endl;
  }
  catch (...) {
    cout << "Caught some exception" << endl;
  }
}
```

上記の例の出力は、以下のとおりです。

```
In first try block
Call to my_unexpected()
Caught bad_exception

In second try block
Call to my_unexpected()
Call to my terminate
```

実行時に、このプログラムは次のように振る舞います。

1. set_terminate() を呼び出すと、old_term に、 set_terminate() が前に呼び出されたときに最後に set_terminate() に渡された関数のアドレスが割り当てられます。
2. set_unexpected() を呼び出すと、old_unex に、 set_unexpected() が前に呼び出されたときに、最後に set_unexpected() に渡された関数のアドレスが割り当てられます。
3. 最初の try ブロックの中では、関数 f() が呼び出されます。 f() が、予期しない例外をスローするので、unexpected() への呼び出しが行われます。次に、unexpected() は、my_unexpected() を呼び出し、標準出力に対してメッセージを印刷します。関数 my_unexpected() は、型 A の例外を再びスローしようとします。クラス A が、関数 f() の例外指定で指定されていないので、 my_unexpected() は、型 bad_exception の例外をスローします。
4. bad_exception が、関数 f() の例外指定で指定されているので、ハンドラー catch (bad_exception& e1) は、例外を処理できます。
5. 2 番目の try ブロックの中では、関数 g() が呼び出されます。 g() が、予期しない例外をスローするので、unexpected() への呼び出しが行われます。 unexpected() は、型 bad_exception の例外をスローします。 bad_exception は、g() の例外指定に指定されていないので、 unexpected() は、terminate() を呼び出し、これが関数 my_terminate() を呼び出します。
6. my_terminate() は、メッセージを表示してから、プログラムを終了する abort() を呼び出します。

例外が、my_unexpected() によって、有効な例外としてではなく、予期しないスロー (throw) として処理されたので、2 番目の try ブロックに続く catch ブロックには、入らないことに留意してください。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、米国以外の国においては本書で述べる製品、サービス、またはプログラムを提供しない場合があります。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711
東京都港区六本木 3-2-12
IBM World Trade Asia Corporation
Intellectual Property Law & Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901 U.S.A.

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。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

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。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

プログラミング・インターフェース情報は、プログラムを使用してアプリケーション・ソフトウェアを作成する際に役立ちます。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

以下は、International Business Machines Corporation の米国およびその他の国における商標です。

IBM	iSeries	System i
IBM (ロゴ)	i5/OS	WebSphere
Integrated Language Environment		

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

業界標準

次の規格がサポートされます。

- C 言語は、International Standard C (ANSI/ISO-IEC 9899-1990 [1992]) に準拠しています。この規格は、ANSI System-Programming Language C (X3.159-1989) を公式に置き換えるもので、 ANSI C 規格と技術的に同等です。コンパイラーは、ISO/IEC 9899:1990/Amendment 1:1994 によって C 標準に採用された変更をサポートしています。
- C++ 言語は、International Standard for Information System-Programming Language C++ (ISO/IEC 14882:1998) に準拠しています。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

[ワ行]

[数字]

A

B

C

D

V

W

[特殊文字]

プログラム番号: 5761–WDS

SC88-4026-01

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12