IBM XL C for AIX, V10.1

ランゲージ・リファレンス

SC88-4956-00
(英文原典：SC23-8884-00)

IBM XL C for AIX, V10.1

ランゲージ・リファレンス

SC88-4956-00
(英文原典：SC23-8884-00)

お願い

本書および本書で紹介する製品をご使用になる前に、 229 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM XL C for AIX, V10.1 (プログラム番号 5724-U80)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。製品のレベルに合った正しい版を使用していることを確認してください。

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

原典： SC23-8884-00
IBM XL C for AIX, V10.1
Language Reference
Version 10.1

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 2008.5

目次

本書について

本書では、C プログラミング言語の構文、セマンティクス、および IBM® XL C Enterprise Edition for AIX® インプリメンテーションについて説明します。XL C コンパイラーは、C プログラミング言語の ISO 標準により維持されている仕様に準拠していますが、同時にコア言語に多くの拡張機能を組み込んでいます。これらの拡張機能は、その他のコンパイラーとの互換性を高め、新しいハードウェア機能をサポートする目的で、インプリメントされました。例えば、2 つの開発環境間の移植性を最大にするために、GNU C コンパイラーとの互換性のための多くの言語構造体が追加されています。

本書の対象読者

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

本書の読み方

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

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

本書の構成

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

1 章から 8 章は、言語エレメント、すなわち、字句エレメント、データ型、宣言、宣言子、型変換、式、演算子、ステートメント、関数などについて説明しています。これらの章全体で、標準機能と拡張機能の両方が説明されています。 9 章は、プリプロセッサーのディレクティブを説明しています。

最後の章では、サポートされているすべての拡張機能の要約をリストしています。

表記規則

書体の規則

次の表では、IBM XL C for AIX, V10.1 の資料で使用される書体の規則について説明します。

表 1. 書体の規則

書体	対象	例
太字	小文字コマンド、実行可能ファイル名、コンパイラー・オプション、およびディレクティブ。	コンパイラーには、基本的な呼び出しコマンドである **xlc** と、さまざまな C 言語レベルとコンパイル環境をサポートするためのその他のコンパイラー呼び出しコマンドが併せて用意されています。
イタリック	実際の名前や値をユーザーが指定するパラメーターまたは変数。新規用語を紹介する際にも、イタリックが使用されます。	要求された *size* より大きな値を返す場合は、必ず *size* パラメーターを更新してください。
下線	コンパイラー・オプションまたはディレクティブのパラメーターのデフォルト設定。	nomaf | maf
モノスペース	プログラミング・キーワードおよびライブラリー関数、コンパイラーの組み込み関数、プログラム・コードの例、コマンド・ストリング、またはユーザー定義名。	myprogram.cをコンパイルして、最適化するには、`xlc myprogram.c -O3` と入力します。

構文図

本書では、構文図は XL C の構文を表します。このセクションでは、これらの図を解釈し、使用するための説明をします。

- 構文図は、線の経路に沿って左から右、上から下に読みます。

  ►►── 記号は、コマンド、ディレクティブ、またはステートメントの始まりを示します。

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

  ►── 記号は、コマンド、ディレクティブ、またはステートメントが前の行からの続きであることを示します。

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

  完全なコマンド、ディレクティブ、またはステートメントではない構文単位の断片図は、|── 記号で始まり、──| 記号で終わります。
- 必須項目は水平線 (メインパス) 上に示されます。

- オプション項目はメインパスの下側に示されます。

- 複数の項目から選択できる場合、それらの項目は縦に並べて表示されます。

  複数の項目から 1 つを選択しなければならない 場合、縦の並びの中の 1 つの項目がメインパス上に表示されます。

  複数の項目から 1 つを選択することがオプションの場合は、縦の並び全体がメインパスの下側に表示されます。

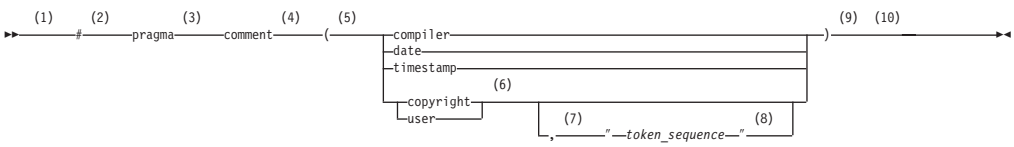
- メインライン上の左に戻る矢印 (繰り返し矢印) は、縦に並んだ項目から複数の項目を選択できること、または同じ項目を繰り返すことができることを示します。ブランク以外の分離文字も、次のように示されます。

- デフォルトの項目は、メインパスの上側に表示されます。

- キーワードは非イタリック体の文字で示され、ここに示されたとおり正確に入力する必要があります。
- 変数はイタリック体の小文字で示されます。変数はユーザー指定の名前や値を表します。
- 句読記号、括弧、算術演算子、またはその他の記号が示されている場合は、それらを構文の一部として入力する必要があります。

構文図の例

次の構文図の例に、**#pragma comment** ディレクティブの構文を示します。



注:

1 これが構文図の始まりです。

2 記号 # で始まります。

3 キーワード pragma は # 記号の後に続けます。

4 pragma comment の名前は、キーワード pragma の後に続けます。

5 ここに左括弧が必要です。

6 コメント・タイプとして、compiler、date、timestamp、copyright、または user のいずれかのタイプを入力します。

7 コメント・タイプ copyright または user とオプションの文字ストリングの間にはコンマを挿入します。

8 文字ストリングをコンマの後に続けます。この文字ストリングは二重引用符で囲みます。

9 ここに右括弧が必要です。

10 これが構文図の終わりです。

次の #pragma comment ディレクティブの例は、上記の図に従った構文上正しいものです。

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

本書での例

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

インストール情報の例は、「*例*」または「*基本例*」というラベルが付いています。*基本例* は、基本的な、あるいはデフォルトのインストール中に実行される手順の文書化を意図したものであり、これらの基本例を変更する必要はほとんど、あるいはまったくありません。

関連情報

以下のセクションには、XL Cの関連情報が記載されています。

IBM XL C の資料

XL C には、次の形式の製品情報が用意されています。

- README ファイル

  README ファイルには、製品情報への変更と修正を含む最新の情報が収められています。 README ファイルは、デフォルトでは XL C ディレクトリーにあり、インストール CD のルート・ディレクトリーにもあります。
- インストール可能なマニュアル・ページ

マニュアル・ページは、製品と共に提供されるコンパイラー呼び出しおよびすべてのコマンド行ユーティリティーに対して提供されています。マニュアル・ページのインストールおよびアクセス手順は、「*IBM XL C for AIX, V10.1 インストール・ガイド*」に記載されています。

- インフォメーション・センター

  検索可能な HTML ファイルのインフォメーション・センターをネットワーク上に立ち上げて、リモート側またはローカル側でアクセスすることができます。オンラインのインフォメーション・センターのインストールおよびアクセス手順は、「*IBM XL C for AIX, V10.1 インストール・ガイド*」に記載されています。

  インフォメーション・センターは、Web サイト (http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp) で表示できます。

- PDF 文書

  PDF 文書は、デフォルトでは /usr/vac/doc/*LANG*/pdf/ ディレクトリー (ここで、*LANG* は en_US、zh_CN または ja_JP のいずれかです) にあります。 PDF ファイルは、Web (http://www.ibm.com/software/awdtools/caix/library) から入手することもできます。

  以下が、XL C のすべての製品情報を構成するファイルです。

表 2. XL C PDF ファイル

文書タイトル	**PDF ファイル名**	**説明**
IBM XL C for AIX, V10.1 インストール・ガイド, GC88-4921-00	install.pdf	XL C のインストール、および基本コンパイルとプログラム実行用の環境の構成に関する情報が含まれます。
IBM XL C for AIX, V10.1 はじめに, GC88-4919-00	getstart.pdf	XL C 製品の紹介、および環境のセットアップと構成、プログラムのコンパイルとリンク、コンパイル・エラーのトラブルシューティングに関する情報が含まれます。
IBM XL C for AIX, V10.1 コンパイラー・リファレンス, SC88-4917-00	compiler.pdf	並列処理に使用されるものを含めた、さまざまなコンパイラー・オプション、プラグマ、マクロ、環境変数、および組み込み関数に関する情報が含まれます。
IBM XL C for AIX, V10.1 ランゲージ・リファレンス, SC88-4956-00	langref.pdf	ポータビリティーおよび非著作権標準への準拠に対応した言語拡張機能などの、IBM がサポートする C プログラミング言語に関する情報が含まれます。
IBM XL C for AIX, V10.1 最適化およびプログラミング・ガイド, SC88-4920-00	proguide.pdf	拡張プログラミングのトピック (アプリケーション・ポーティング、Fortran コードによる言語間呼び出し、ライブラリー開発、アプリケーションの最適化と並列化、および XL C ハイパフォーマンス・ライブラリーなど) の情報が含まれます。

PDF ファイルを読むには、Adobe® Reader を使用してください。Adobe Reader がない場合は、Adobe Web サイト (http://www.adobe.com) からダウンロードすることができます (ライセンス条項の対象となります)。

Redbooks、ホワイト・ペーパー、チュートリアル、およびその他の論文など、XL C に関する詳しい情報は、次の Web サイトから入手できます。

http://www.ibm.com/software/awdtools/caix/library

標準および仕様

XL C は、以下の標準および仕様をサポートするように設計されています。本書に記載された機能の一部について、正確な定義を確認するには、これらの標準を参照することができます。

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*、*C89* とも呼ばれています。
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*、*C99* とも呼ばれています。
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. このドラフト技術レポートは、C 標準委員会によって受諾されており、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf から入手できます。
- *AltiVec Technology Programming Interface Manual*、Motorola Inc。ベクトル処理テクノロジーをサポートするためのベクトル・データ型に関するこの仕様は、http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手できます。
- *Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*. このドラフト技術レポートは C 標準委員会に提出済みであり、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf から入手できます。
- *Decimal Types for C++: Draft 4* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1977.html

その他の IBM 資料

- *AIX コマンド・リファレンス 第 1 巻 から 第 6 巻*、*SC88-6875*
- *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2*、*SC23-4913*
- *AIX ナショナル・ランゲージ・サポート ガイドおよびリファレンス*、*SC88-6933*
- *AIX プログラミングの一般概念: プログラムの作成とデバッグ*、*SC88-6931*
- *AIX Assembler Language Reference*、*SC23-4923*

  AIX の全資料は http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp から入手できます。
- *Parallel Environment for AIX: Operation and Use*
- *ESSL for AIX V4.2 Guide and Reference*、*SA22-7904*、http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html から入手できます。

その他の資料

- *Using the GNU Compiler Collection*、http://gcc.gnu.org/onlinedocs から入手可能

技術サポート

追加の技術サポートは、XL C のサポート・ページ (http://www.ibm.com/software/awdtools/caix/support) から利用することができます。このページには、膨大な技術情報やその他のサポート情報を対象に検索が行える機能を備えたポータルが用意されています。

必要な情報が見つからない場合は、compinfo@ca.ibm.com に E メールを送信してください。

XL C の最新情報については、http://www.ibm.com/software/awdtools/caix の製品情報サイトにアクセスしてください。

第 1 章 言語レベルと言語拡張

本書で説明する C 言語は、 xii ページの『標準および仕様』 にリストされている標準に基づきます。

基本と拡張という考え方を取り入れるために、以下の言語仕様を「基本言語レベル」と呼ぶことにします。このコンテキストでは、基本言語レベルとは、以下の仕様を指します。

- C99
- C89

本書で用語 *K&R C* を使用した場合、それは、C 言語と、C の ISO 標準化の前に使用されていた、Brian Kernighan および Dennis Ritchie が作成し、広く受け入れられていた拡張機能を合わせたものを指します。

基本レベルでサポートされる機能に加えて、XL C は、使用可能度を向上させ、異なるプラットフォームにプログラムを移植しやくする、次のような言語拡張機能が組み込まれています。

- 2 ページの『GNU C に関連する拡張機能』
- 2 ページの『AltiVec プログラミング・インターフェースをサポートする拡張機能』

コンパイルに使用する言語レベルは、次のようないくつかの仕組みによってユーザーが制御することができます。

- 「*XL C コンパイラー・リファレンス*」の『各種の呼び出しコマンド』
- 「*XL C コンパイラー・リファレンス*」の**-qlanglvl** オプション

基本呼び出しコマンド **xlc** を使用してコンパイルする場合、いくつかの例外はありますが、言語拡張機能のほとんどすべてがサポートされます。

xlc 呼び出しコマンドのデフォルトの言語レベルは **extc99** です。これには、C99 標準により導入されたフィーチャーのすべて、および本書で説明するほとんどの IBM 拡張が組み込まれています。C 拡張機能およびそれらを使用可能にするさまざまな方法の完全なリストについては、 223 ページの『第 11 章 IBM XL C 言語拡張機能』を参照してください。

コンパイルの言語レベルを制御するさまざまな方法については、「*XL C コンパイラー・リファレンス*」の『コンパイラーの呼び出し』および**-qlanglvl**を参照してください。

10 進浮動小数点ハードウェアをサポートするための拡張機能

XL C は、「*Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*」で推奨されている、C 言語に組み込まれている 10 進浮動小数点型をサポー

トします。このサポートは、「*XL C コンパイラー・リファレンス*」内の **-qdfp** オプションで使用可能になります。

GNU C に関連する拡張機能

GNU C 機能に対応する特定の言語拡張機能は、移植性を向上させるためにインプリメントされています。このような拡張には、C89 および C99 への拡張機能が含まれます。本書全体にわたって、GNU C との互換性を確保するためにインプリメントされた IBM 拡張機能はテキストで明記されます。このような機能の全リストも 225 ページの『GNU C 互換性の拡張機能』 に掲載されています。

AltiVec プログラミング・インターフェースをサポートする拡張機能

XL C は、ベクトルのサポートが使用可能になっている場合に、AltiVec ベクトル型をサポートします。これらの言語拡張機能は、PowerPC® プロセッサーの SIMD および並列処理機能を活用し、関連する最適化手法を容易にします。IBM インプリメンテーションの AltiVec プログラミング・インターフェース仕様は、その大部分が GNU C インプリメンテーションの構文とセマンティクスに一致する拡張インプリメンテーションです。本書に記載されたテキストでベクトル拡張機能の振る舞いについて説明していますが、そのほかにも、AltiVec プログラミング・インターフェースに対する IBM 拡張のリストが、227 ページの『ベクトル処理の拡張機能サポート』に記載されています。

第 2 章 スコープとリンケージ

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

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

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

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

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

異なるタイプのスコープの区別については、4 ページの『スコープ』で説明しています。リンケージのタイプの違いについては、8 ページの『プログラム・リンケージ』で説明しています。

関連資料

43 ページの『ストレージ・クラス指定子』

スコープ

ID のスコープ は、 ID が該当のオブジェクトを参照するために使われる可能性があるプログラム・テキストの最大の領域です。 ID の意味は、その ID が使用されるコンテキストに依存します。スコープとは、名前の意味を区分するための一般的なコンテキストです。

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

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

表 3. C と C++ の用語の違い

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

すべての宣言において、ID は、初期化指定子の前にスコープに入れられます。次の例は、このことを示しています。

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

関数 `f()` の中で宣言された `x` は、ローカル・スコープをもっています (グローバル・スコープではなく)。

ブロック・スコープ

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

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

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

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

関連資料

155 ページの『ブロック・ステートメント』

46 ページの『register ストレージ・クラス指定子』

関数スコープ

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

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

関連資料

153 ページの『ラベル付きステートメント』

関数プロトタイプ・スコープ

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

関連資料

178 ページの『関数宣言』

ファイル・スコープ

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

関連資料

8 ページの『内部リンケージ』

C におけるスコープの例

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

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

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

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

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

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。同じコンテキストおよび同じスコープ内では、ID は、エンティティーを一意的に識別しなければなりません。 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・スペース内の同一の ID は、同じスコープ内にあっても互いに干渉しません。

各 ID がそのネーム・スペース内で固有である限り、同じ ID で異なるオブジェクトを宣言することができます。プログラム内の ID の構文上のコンテキストによって、コンパイラーは、そのネーム・スペースをあいまいさなしに解決します。

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

ID は、内側のプログラム・ブロックを使用して、同じネーム・スペース内で再定義できます。

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

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

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

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

プログラム・リンケージ

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

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

リンケージはスコープの扱いに影響を与えず、通常の名前検索の考慮事項が適用されます。

関連資料

44 ページの『static ストレージ・クラス指定子』

45 ページの『extern ストレージ・クラス指定子』

181 ページの『関数のストレージ・クラス指定子』

71 ページの『型修飾子』

内部リンケージ

次の種類の ID は、内部リンケージを持っています。

- `static` と明示的に宣言されたオブジェクト、参照、または関数。
- 指定子 `const` を使用してグローバル・スコープ内で宣言されていて、 `extern` と明示的に宣言されておらず、以前に外部リンケージをもっているとも宣言されていない、オブジェクトまたは参照。
- 無名共用体のデータ・メンバー。

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

ID の宣言にキーワード `extern` があり、 ID の直前の宣言がネーム・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。

関連資料

5 ページの『ファイル・スコープ』

44 ページの『static ストレージ・クラス指定子』

181 ページの『static ストレージ・クラス指定子』

外部リンケージ

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

- オブジェクト
- 関数

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

関連資料

45 ページの『extern ストレージ・クラス指定子』

90 ページの『配列の互換性』

182 ページの『extern ストレージ・クラス指定子』

リンケージなし

次の種類の ID には、リンケージがありません。

- 外部リンケージも内部リンケージももたない名前
- 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、リンケージをもたないエンティティーを指す typedef 名、型名、関数仮パラメーター、およびテンプレート名

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

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

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

第 3 章 字句エレメント

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

トークン

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

- キーワード
- ID
- リテラル
- 区切り子と演算子

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

キーワード

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

表 4. C のキーワード

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

C C99 レベルの標準 C 言語では、次のキーワードも予約されています。

表 5. C99 キーワード

_Bool	_Imaginary[1]
_Complex	inline
	restrict

注:

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

C

言語拡張のキーワード (IBM 拡張)

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

表 6. C 言語 言語拡張のためのキーワード

__alignof	_Decimal32[4]	__restrict
__alignof__	_Decimal64[4]	__restrict__
__asm (C のみ)	_Decimal128[4]	__signed__
__asm__ (C のみ)	__extension__	__signed
__attribute__	__label__	__volatile__
__attribute	__imag__	__thread[5]
bool	__inline__[2]	typeof[3]
__complex__	pixel[1]	__typeof__
__const__	__pixel[1]	vector[1]
	__real__	__vector[1]

注:

1. これらのキーワードは、ベクトル・サポートが使用可能 となっている場合に、ベクトル宣言コンテキスト内でのみ認識されます。
2. `__inline__` キーワードは、インライン関数用に GNU C のセマンティクスを使用します。詳しくは、 183 ページの『インライン関数のリンケージ』 を参照してください。
3. `typeof` は、**-qkeyword=typeof** が有効になっている場合にのみ認識されます。
4. これらのキーワードは、 **-qdfp** が使用可能となっている場合のみ認識されます。
5. `__thread` は、**-qtls** が使用可能となっている場合にのみ認識されます。

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

関連資料

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

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

55 ページの『ベクトル型 (IBM 拡張)』

13 ページの『ID』

ID

ID は、次の言語エレメントに名前を付けます。

- 関数
- オブジェクト
- ラベル
- 関数仮パラメーター
- マクロおよびマクロ・パラメーター
- 型定義
- 列挙型および列挙子
- 構造体および共用体の名前

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

ID の文字

ID の先頭文字は、文字または _ (下線) 文字でなければなりません。ただし、下線で始まる ID は、プログラミング・スタイルとしては良い方法ではありません。

コンパイラーでは、ID 内の大文字と小文字が区別されます。例えば、`PROFIT` と `profit` は、別の ID を表します。 ID の名前の一部に小文字の `a` が指定された場合、それを大文字の `A` に置き換えることはできません。小文字を使用しなければなりません。

基本ソース文字セット以外の文字および数字のユニバーサル文字名は、 C99 言語レベルで許可されています。

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

予約 ID

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

単一の下線で始まる ID は、通常のネーム・スペース内およびタグ・ネーム・スペース内のファイル・スコープに関する ID として予約されています。

システム呼び出しおよびライブラリー関数の名前は、適切なヘッダーをインクルードしない場合は予約語ではありませんが、これらの名前を ID としては、使用しないようにしてください。事前定義の名前を重複使用すると、コードの保守を行う人

が混乱したり、リンク時または実行時のエラーの原因になります。プログラムにライブラリーをインクルードする場合は、名前が重複しないように、そのライブラリーの関数名に注意を払ってください。標準のライブラリー関数を使用するときには常に、適切なヘッダーをインクルードする必要があります。

__func__ 事前定義 ID

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

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

ここで、*function-name* はこの ID を字句として含んでいる関数の名前です。

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

```
#include <stdio.h>

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

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

このプログラムの出力は、次のようになります。

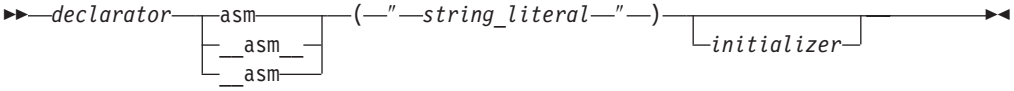
```
myfunc
size of __func__=7
```

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

アセンブリー・ラベル (IBM 拡張)

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

アセンブリー・ラベルの構文



string_literal は、指定されたオブジェクトまたは関数にバインドされる有効なアセンブリー名です。関数宣言に適用されるラベルの場合、この名前はコンパイル単位で

定義された既存の関数を指定している必要があります。定義されていない場合、リンク時にエラーが発生します。変数宣言に適用されるラベルの場合は、その他の定義は必要ありません。

次に、アセンブリー・ラベル指定の例を示します。

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

以下は、アセンブリー・ラベルの使用に関する制限です。

- ローカル変数や静的変数にアセンブリー・ラベルを指定することはできません。
- 同じコンパイル単位で、同じアセンブリー・ラベル名を複数の ID に適用することはできません。
- ラベル名と ID 名が同じ変数宣言または関数宣言で使用されている場合を除き、同じコンパイル単位で、アセンブリー・ラベル名は、その他のグローバル ID 名と同じ名前にすることはできません。
- アセンブリー・ラベルを `typedef` 宣言の中で指定することはできません。
- アセンブリー・ラベルは、直前の **#pragma map** ディレクティブが異なる変数や関数に指定した名前と同じ名前にすることはできません。同様に、 **#pragma map** ディレクティブで指定されたマップ名は、直前のアセンブリー・ラベルが異なる変数や関数に指定した名前と同じ名前にすることはできません。
- **#pragma map** ディレクティブが直前の変数宣言や関数宣言で異なる名前にマップした ID に、アセンブリー・ラベルを適用することはできません。同様に、アセンブリー・ラベルが直前に再マップした ID に、**#pragma map** ディレクティブを指定することはできません。
- 同じ変数または関数の複数の宣言で異なるラベルを適用すると、最初の指定は受け入れられますが、後続のアセンブリー・ラベルはすべて無視され、警告が出されます。

関連資料

32 ページの『ユニコード規格』

11 ページの『キーワード』

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

177 ページの『関数の宣言と定義』

「XL C コンパイラー・リファレンス」の #pragma map を参照

189 ページの『alias 関数属性』

指定したレジスターの変数 (IBM 拡張)

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

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

リテラル

リテラル定数 という語、すなわち、*リテラル* とは、プログラム内で使われる、変更できない値のことです。C 言語は、名詞*リテラル* の代わりに用語*定数* を使用し

ます。形容詞リテラル は、定数の概念に、その定数に関しては値だけを扱うという考え方を追加します。リテラル定数はアドレス指定不可です。つまり、その値はメモリー内のどこかに保管されており、そのアドレスにアクセスする手段はないことを意味します。

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

- 『整数リテラル』
- 18 ページの『ブール・リテラル』
- 18 ページの『浮動小数点リテラル』
- IBM 23 ページの『ベクトル・リテラル (IBM 拡張)』
- 25 ページの『文字リテラル』
- 26 ページの『ストリング・リテラル』

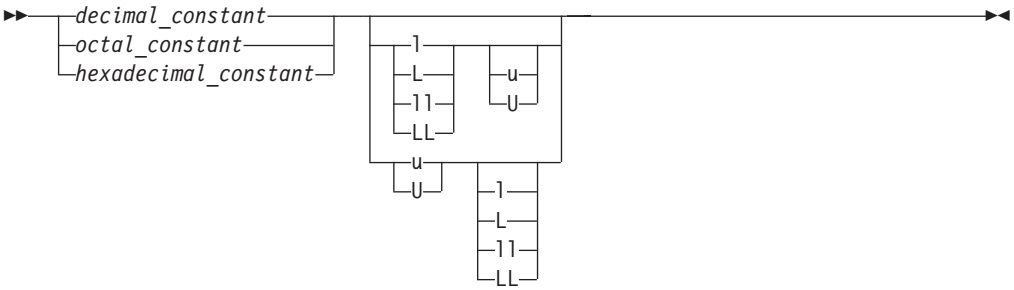
整数リテラル

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

- 10 進整数リテラル
- 16 進整数リテラル
- 8 進整数リテラル

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

整数リテラルの構文



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

整数リテラル	可能なデータ型
接尾部がない 10 進数	int、long int、 long long int[1]
接尾部がない 8 進数または 16 進数	int、unsigned int、long int、unsigned long int、 long long int[1]、 unsigned long long int[1]
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[1]

整数リテラル	可能なデータ型
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[1]
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[1]、unsigned long long int[1]
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[1]
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
注: 1. **xlc** または **c99** または **-qlanglvl=stdc99** または **-qlanglvl=extc99**。	

10 進整数リテラル

10 進整数リテラル は 0 から 9 までの数字を使用できます。最初の数字を 0 にすることはできません。数字 0 で始まる整数リテラルは、10 進整数リテラルとしてではなく、8 進整数リテラルとして解釈されます。

10 進整数リテラルの構文

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

10 進リテラルの例を次に示します。

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

16 進整数リテラル

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

16 進整数リテラルの構文

16 進整数リテラルの例を次に示します。

```
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 までの数字が続きます。

8 進整数リテラルの構文

8 進整数リテラルの例を次に示します。

```
0
0125
034673
03245
```

ブール・リテラル

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

浮動小数点リテラル

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

- 実リテラル
  - 2 進浮動小数点リテラル
  - 16 進浮動小数点リテラル
  - IBM 10 進浮動小数点リテラル (IBM 拡張)
- 複素数リテラル

2 進浮動小数点リテラル

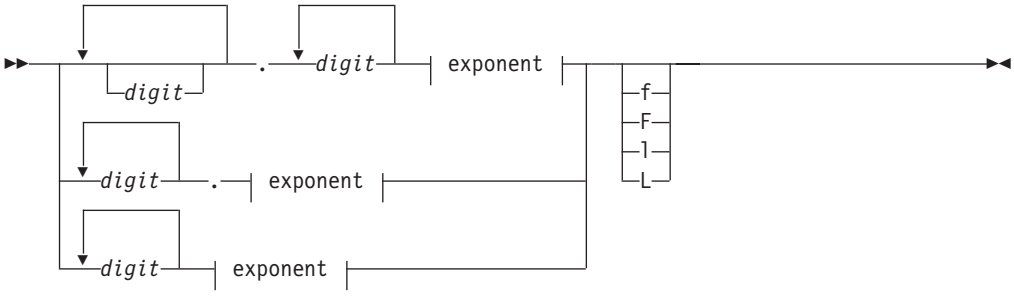
実 2 進浮動小数点定数は、次のもので構成されます。

- 整数部分
- 小数点
- 小数部分

- 指数部
- オプションの接尾部

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

2 進浮動小数点リテラルの構文



Exponent:

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

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

浮動小数点リテラルの例を次に示します。

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

16 進浮動小数点リテラル

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

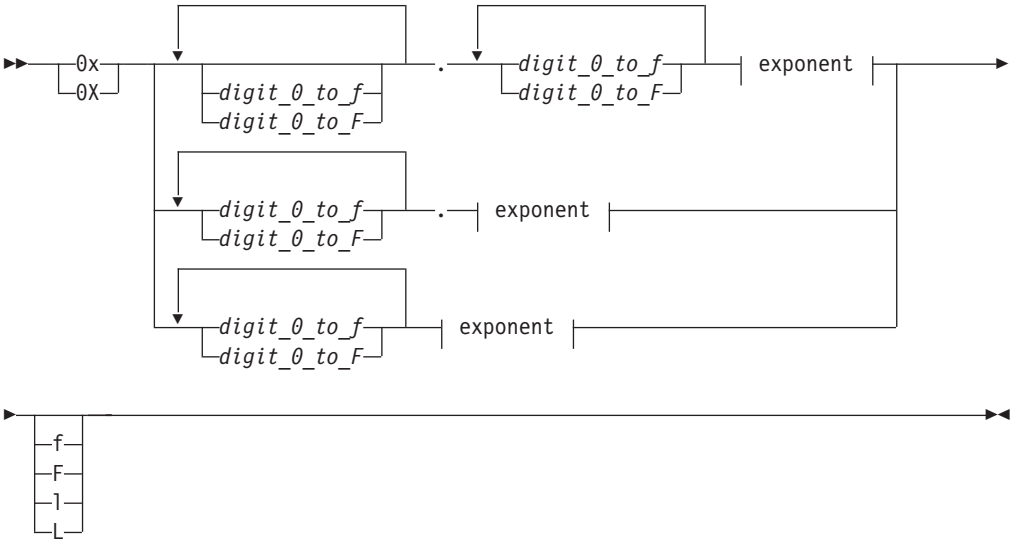
- 16 進数接頭部
- 有効部分
- 2 進指数部
- オプションの接尾部

有効部分は有理数を表し、以下のもので構成されます。
- 16 進数字のシーケンス (整数部分)
- オプションの小数部分

オプションの小数部は、ピリオドとその後の 16 進数字のシーケンスです。

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

16 進浮動小数点リテラルの構文



Exponent:

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

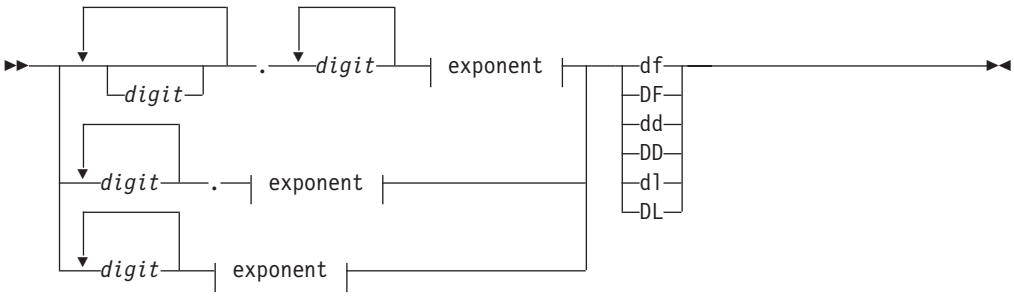
10 進浮動小数点リテラル (IBM 拡張)

実 10 進浮動小数点定数は、次のもので構成されます。

- 整数部分
- 小数点
- 小数部分
- 指数部
- オプションの接尾部

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

10 進浮動小数点リテラルの構文



Exponent:

接尾部 df または DF は _Decimal32の型を示し、接尾部 dd または DD は _Decimal64 の型を示します。また、接尾部 dl または DL は _Decimal128 の型を示します。 接尾部を指定しないと、浮動小数点定数には、double 型が指定されます。

リテラル接尾部には、大文字と小文字を混合することができません。

10 進浮動小数点リテラル宣言の例を次に示します。

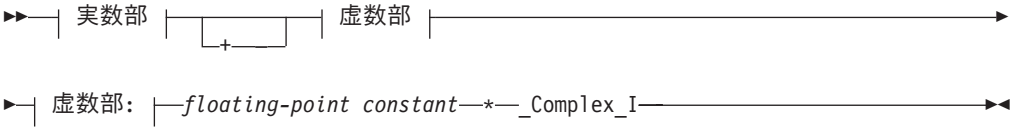
```
_Decimal32 a = 22.2df;
_Decimal64 b = 33.3dd;
```

注: 10 進浮動小数点リテラル接尾部は、**-qdfp** オプションが使用可能となっている場合に限り認識されます。

複素数リテラル

複素数リテラル (C99 の機能) は、実数部と虚数部という 2 つの部分で構成されています。

複素数リテラルの構文



実数部:

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

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

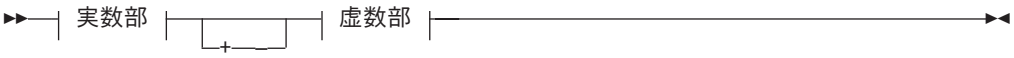
例えば、次の宣言

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

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

IBM また、GNU C で開発されたアプリケーションの移植を容易にするために、XL Cを使用すると、複素数の型 (float、double、または long double) を示す標準の接尾部のほかに、接尾部を付けて複素数リテラルの虚数部を示すこともできます。

GNU 接尾部を使用する複素数リテラルの単純化した構文は、次のとおりです。



実数部:

虚数部:

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

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

例えば、次の宣言

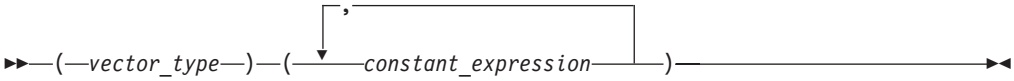
```
varComplex = 3.0f + 4.0fi;
```

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

ベクトル・リテラル (IBM 拡張)

ベクトル・リテラルは、値がベクトル型として解釈される定数式です。ベクトル・リテラルのデータ型は、ベクトル型を括弧で囲んだ形で表され、値は、ベクトル・エレメントを表す一組の定数式を括弧で囲んだ形で表されます。すべてのベクトル・エレメントの値が同じである場合は、リテラルは、括弧で囲んだ単一の定数式で表すことができます。ベクトル・リテラルにより、ベクトル型を初期化することができます。

ベクトル・リテラルの構文



vector_type は、サポートされているベクトル型です。 55 ページの『ベクトル型 (IBM 拡張)』に、これらがリストされていますので参照してください。

constant_expression は、次のどちらかです。

- 単一の式。これは、ベクトルのすべてのエレメントを同じ値に初期化します。
- コンマで区切られた式のリスト。式の数は、ベクトルの型によって決まっています。定数式の数は正確に以下のとおりでなければなりません。

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

  8 vector short および vector pixel 型の場合。

  16 vector char 型の場合。

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

表 7. ベクトル・リテラル

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

表 7. ベクトル・リテラル (続き)

構文	コンパイラーによる解釈
(vector unsigned short)(*unsigned int*)	それぞれが単一整数値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector unsigned short)(*unsigned int*, ...)	各整数 (8 つ) によってそれぞれ指定された値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector signed short)(*int*)	それぞれが単一整数値を有する、8 つの符号付き 16 ビット数量値のセット。
(vector signed short)(*int*, ...)	各整数 (8 つ) によってそれぞれ指定された値を有する、6 つの符号付き 16 ビット数量値のセット。
(vector bool short)(*unsigned int*)	それぞれが単一整数値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector bool short)(*unsigned int*, ...)	各整数 (8 つ) によってそれぞれ指定された値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector unsigned int)(*unsigned int*)	それぞれが単一整数値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector unsigned int)(*unsigned int*, ...)	各整数 (4 つ) によってそれぞれ指定された値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector signed int)(*int*)	それぞれが単一整数値を有する、4 つの符号付き 32 ビット数量値のセット。
(vector signed int)(*int*, ...)	各整数 (4 つ) によってそれぞれ指定された値を有する、4 つの符号付き 32 ビット数量値のセット。
(vector bool int)(*unsigned int*)	それぞれが単一整数値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector bool int)(*unsigned int*, ...)	各整数 (4 つ) によってそれぞれ指定された値を有する、4 つの符号なし 32 ビット数量値のセット。
(vector float)(*float*)	それぞれが単一の浮動小数値を有する、4 つの 32 ビット単精度浮動小数点数量値のセット。
(vector float)(*float*, ...)	各浮動小数点 (4 つ) によってそれぞれ指定された値を有する、4 つの 32 ビット単精度浮動小数点数量値のセット。
(vector pixel)(*unsigned int*)	それぞれが単一整数値を有する、8 つの符号なし 16 ビット数量値のセット。
(vector pixel)(*unsigned int*, ...)	各整数 (8 つ) によってそれぞれ指定された値を有する、8 つの符号なし 16 ビット数量値のセット。

例えば、符号なし整数ベクトル型の場合、リテラルは次のどちらでも可能です。

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

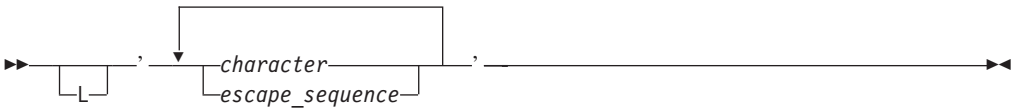
ベクトル・リテラルは、他のベクトル・タイプにキャストすることができます。ベクトル・リテラルのキャストによって、オペランドのビット・パターンが変わることはありません。つまり、値を表す 128 バイトは、キャストの前も後も同じです。

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'` です。文字リテラルには、例えば `L'c'` のように、文字 L を接頭部として付けることができます。L 接頭部のない文字リテラルは、*通常の文字リテラル*、すなわち*ナロー文字リテラル* です。 L 接頭部のある文字リテラルは、*ワイド文字リテラル* です。複数の文字またはエスケープ・シーケンス (単一引用符 (')、バックスラッシュ (\) 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。

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

文字リテラルの構文



文字リテラルには、少なくとも 1 つの文字またはエスケープ・シーケンスがなければなりません。また、文字リテラルは、単一の論理ソース行に記述されなければなりません。

この文字には、ソース・プログラムの文字セットのどの文字でも使用できます。二重引用符記号は、それ自体で二重引用符記号を表すことができます。しかし、単一引用符記号を表すには、円記号の後に単一引用符記号 (¥' エスケープ・シーケンス) を使用する必要があります。(エスケープ文字で表される他の文字のリストについては、 31 ページの『エスケープ・シーケンス』を参照してください。)

基本ソース文字セット外では、文字および数字のユニバーサル文字名は、 C99 言語レベルで使用できます。

文字リテラルの例を次に示します。

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

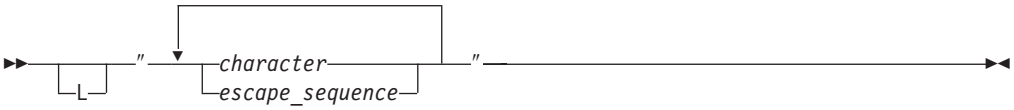
ストリング・リテラル

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*ナロー文字リテラル* です。

ナロー文字リテラルの型は、char の配列です。ワイド文字リテラルの型は wchar_t です。

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

ストリング・リテラルの構文



ストリング・リテラル内に含まれている複数のスペースは保存されます。

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

基本ソース文字セット外では、文字および数字のユニバーサル文字名は、 C99 言語レベルで使用できます。

IBM ストリング・リテラルの Pascal ストリング形式も、**-qmacpstr** オプションを使用してコンパイルする場合は受け入れられます。

ストリング・リテラルの例を、次に示します。

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

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

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

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

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

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

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

ストリング連結

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

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

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

ワイド・ストリング・リテラルと狭幅のストリング・リテラルが次のように隣接すると、

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

その結果は、ワイド・ストリング・リテラルになります。

連結の後で、各ストリングの終わりに、char 型の '\0' が付加されます。ワイド・ストリング・リテラルに対して、型 wchar_t の '\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"  */
```

関連資料

50 ページの『整数型』

108 ページの『整数の変換』

51 ページの『ブール型』

108 ページの『ブール変換』

52 ページの『浮動小数点型』

109 ページの『浮動小数点変換』

121 ページの『単項式』

複素数浮動小数点型

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

54 ページの『文字型』

29 ページの『ソース・プログラムの文字セット』
32 ページの『ユニコード規格』
u リテラルのストリング連結


「XL C コンパイラー・リファレンス」の 中の『 -qro』 を参照


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


「XL C コンパイラー・リファレンス」の #pragma map を参照


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


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

118 ページの『リテラル』

区切り子と演算子

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

C99 では、以下のトークンが区切り子、オペレーター、またはプリプロセッシング・トークンとして定義されています。

表 8. C の区切り子

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

代替トークン

下表に、一部の演算子および区切り子の代替表記をリストします。

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

上記にリストした演算子および区切り子のほかに、 C99 言語レベルは以下の代替表記をヘッダー・ファイル `iso646.h` の中でマクロとして定義して提供します。

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

関連資料

34 ページの『2 文字表記文字』

51 ページの『ブール型』

108 ページの『ブール変換』

52 ページの『浮動小数点型』

109 ページの『浮動小数点変換』

121 ページの『単項式』

55 ページの『ベクトル型 (IBM 拡張)』

94 ページの『ベクトルの初期化 (IBM 拡張)』

『ソース・プログラムの文字セット』

32 ページの『ユニコード規格』

54 ページの『文字型』

115 ページの『第 7 章 式と演算子』

ソース・プログラムの文字セット

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

- アルファベットの大文字および小文字

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

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

  0 1 2 3 4 5 6 7 8 9
- 以下の図形文字

  ! ″ # % & ’ ( ) * + , - . / : ; < = > ? [\ ] _ { } ~
  – ASCII の脱字記号 (^) 文字 (ビット単位の排他 OR 記号)

– ASCII の分割垂直バー (¦) 文字

- 空白文字
- 改行、水平タブ、垂直タブ、改ページ、ストリングの終り (ヌル文字)、アラート、バックスペース、および復帰を表す制御文字

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

関連資料

15 ページの『リテラル』

28 ページの『区切り子と演算子』

ID の文字

マルチバイト文字

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

マルチバイト文字は、以下のいずれのコンテキストにも含めることができます。

- ストリング・リテラルおよび文字定数。マルチバイト・リテラルを宣言するためには、 L の接頭部を付けたワイド文字表現を使用します。例えば、以下のようになります。

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

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

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

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

- # および ## 演算子
- プログラムのコメント

以下は、マルチバイト文字の使用に関する制限です。

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

関連資料

文字リテラル

32 ページの『ユニコード規格』

54 ページの『文字型』

「XL C コンパイラー・リファレンス」の 中の『 -qmbcs』 を参照

35 ページの『コメント』

エスケープ・シーケンス

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

エスケープ文字の構文

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

注: 行継続シーケンス (\ の後に改行文字が続く) は、エスケープ・シーケンスではありません。行継続シーケンスは、文字ストリングの中で使われ、ソース・コードの現在行が次の行に続くことを表します。

エスケープ・シーケンスと表される文字は、次のとおりです。

エスケープ・シーケンス	表される文字
\a	アラート (ベル、アラーム)
\b	バックスペース
\f	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としてではなく) は、¥¥ のように円記号エスケープ・シーケンスを使用する必要があります。次に例を示します。

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

このステートメントでは、次のように出力されます。

```
The escape sequence ¥n.
```

ユニコード規格

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

ユニコード規格は、各文字に固有の数値と名前を指定しており、数値のビット表現に 3 つのエンコード方式を定義しています。名前と値のペアで文字の識別を行います。文字を表す 16 進値はコード・ポイント と呼ばれます。仕様には、全体の文字特性、すなわち、各文字の大/小文字、方向性、英字特性、その他のセマンティクス情報も記述されています。 ASCII に基づくと、ユニコード規格は英字、表意文字、およびシンボルを扱い、予約済みコード・ポイント範囲でインプリメンテーション

別の文字コードを定義することができます。したがって、ユニコード規格に準拠するエンコード方式は、世界中のすべての歴史的スクリプトへの対応を含め、あらゆる既知の文字エンコード要件を扱える十分な柔軟性があります。

C99 では、ISO/IEC 10646 で定義されているユニバーサル文字名構成を使用して、基本ソース文字セット外の文字を表すことができます。ID、文字定数、およびストリング・リテラルでユニバーサル文字名を使用することができます。

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

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

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

以下の文字も禁止されています。

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

UTF リテラル (IBM 拡張)

C 標準委員会は、ユニコード UTF-16 および UTF-32 文字リテラルをサポートするために、それぞれ *u-literals* および *U-literals* のインプリメンテーションを承認しました。ソース・コードで UTF リテラルのサポートを使用可能にするには、オプション **-qutf** を使用可能にしてコンパイルする必要があります。以下の表に、UTF リテラルの構文を示します。

表 9. UTF リテラル

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

XL C は、マクロ `uint_least16_t` および `uint_least32_t` をインプリメントします。これらのマクロは、ヘッダー・ファイル `stdint.h` の中に、UTF-16 および UTF-32 文字用のデータ型として定義されます。以下の例では、`uint_least16_t` の配列を定義し、そこにはコード・ポイント 1234 および 8180 で表される文字が含まれます。

```
#include <stdint.h>

uint_least16_t  msg[] = u"ucs characters \u1234 and \u8180 ";
```

u リテラルのストリング連結

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

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

複数の連結を使用することもできますが、上記の規則が再帰的に適用されます。

関連資料

13 ページの『ID』

15 ページの『リテラル』

28 ページの『区切り子と演算子』

30 ページの『マルチバイト文字』

「XL C コンパイラー・リファレンス」の 中の『 -qutf』 を参照

ストリング連結

2 文字表記文字

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

2 文字表記文字は、次の通りです。

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

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

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

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

関連資料

28 ページの『区切り子と演算子』


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

3 文字表記

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 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。次に例を示します。

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

これは、次のものを表します。

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

関連資料

124 ページの『ビット単位否定演算子 ~』

138 ページの『ビット単位排他 OR 演算子 ^』

139 ページの『ビット単位包含 OR 演算子 |』

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。
- // (2 つのスラッシュ) があって、その後に文字の任意のシーケンスが続きます。直前に円記号 (¥) がない状態で改行すれば、この形式のコメントは終了します。この種類のコメントは、通常、*単一行コメント* または *C++ コメント* と呼ばれています。 C++ コメントは、行結合 (\) 文字を使用して 1 行の論理ソース行に

結合することで、複数行の物理ソース行にまたがることができます。円記号の文字は 3 文字表記で表すこともできます。

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

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

注: 文字定数またはストリング・リテラルで検出される /* 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・コードで既にコメントにされている個所をコメントにしないようにするには、条件付きコンパイルのプリプロセッサー・ディレクティブを使用して、コンパイラーにプログラムのセクションを迂回させる必要があります。例えば、上記のステートメントをコメントにする代わりに、ソース・コードを次のように変更します。

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

#define TEST_FUNCTION 0
#include <stdio.h>

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

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

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

```
#include <stdio.h>

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

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

関連資料

「XL C コンパイラー・リファレンス」の 中の『 -qmbcs』 を参照

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


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

30 ページの『マルチバイト文字』

第 4 章 データ・オブジェクトとデータ宣言

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

トピックは、エレメントが宣言の中に現れる順序と概ね同じ順で記載されています。 81 ページの『第 5 章 宣言子』でも、引き続いてデータ宣言の追加エレメントについて説明します。

関連資料

153 ページの『第 8 章 ステートメント』

データ・オブジェクトとデータ宣言の概要

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

データ・オブジェクトの概要

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

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

「基本型」対「派生型」
: 基本 データ型は、言語に対して「基本」または「組み込み」とも呼ばれます。基本データ型としては、整数型、浮動小数点数型、および文字型があります。派生 型は基本型セットから作成され、配列、ポインター、構造体、共用体、および列挙型があります。

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

「スカラー型」対「集合体型」
: スカラー 型は単一のデータ値を表し、一方、集合体 型は、同じ型の複数の値、または異なる型の複数の値を表します。スカラーには、算術型とポインターがあります。集合体型には、配列と構造体があります。

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

表 10. C データ型

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

注:

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

不完全型

以下は、不完全型です。

- void 型
- 不明サイズの配列
- 不完全型エレメントの配列
- 定義のない構造体、共用体、または列挙型

配列サイズが、[*]で指定された場合、これは可変長配列であることを表し、サイズは指定されたものと見なされ、そのため、配列型は完全型と見なされます。詳しくは、89 ページの『可変長配列』を参照してください。

以下の例は、不完全型を示しています。

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

互換型および複合型

C では、互換性のある型とは、次のように定義されています。

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

2 つの互換性のある型が結合されると、結果は*複合型* になります。2 つの互換型から生ずる複合型の決定は、整数型が算術演算子で結合されたときの整数型の通常の 2 項変換の結果と似ています。

明らかに、2 つの同じ型は互換性があります。その複合型は同じ型です。同一でない型、ユーザー定義型、および型修飾された型の型互換性などを統制する規則は、分かりにくくなります。 50 ページの『型指定子』で、C での基本型とユーザー定義型の互換性を説明しています。

C

関連資料

54 ページの『void 型』

90 ページの『配列の互換性』

87 ページの『ポインターの互換性 (C のみ)』

180 ページの『互換性のある関数 (C のみ)』

データ宣言とデータ定義の概要

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

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

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

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

注: C99 標準では、最初のステートメントの前の関数の先頭にすべての宣言が現れなければならないという必要はなくなりました。C++ の場合のように、ユーザー・コードの中で宣言を他のステートメントと混在させることができます。

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

- スコープ。これは、ID がそのオブジェクトにアクセスするために使われるプログラム・テキストの領域を定義します。
- 可視性。これは、ID のオブジェクトに正しくアクセスできるプログラム・テキストの領域を定義します。
- 期間。これは、ID の物理オブジェクトが実際にメモリー内に割り振られている期間を定義します。

- リンケージ。ある 1 つの ID と 1 つの特定のオブジェクトの正しい関連付けを定義します。
- 型。これにより、オブジェクトに割り振るメモリーの大きさと、そのオブジェクトのストレージ割り振りで検出されたビット・パターンをプログラムはどのように解釈するかが決まります。

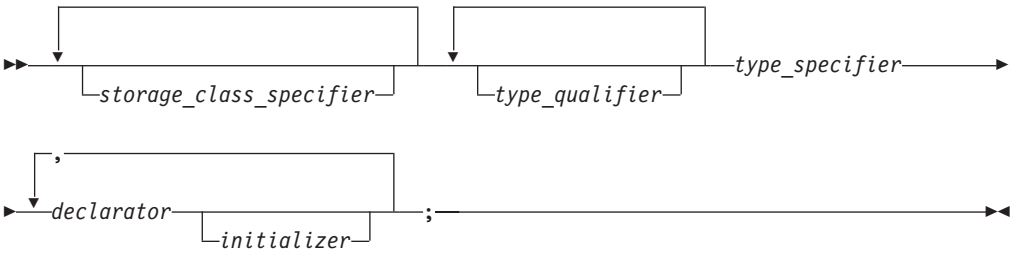
データ・オブジェクトを宣言する際のエレメントは、次のとおりです。

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

IBM さらに、XL C では、GCC との互換性のために、ユーザーが*属性* を使用してデータ・オブジェクトのプロパティーを変更できるようになりました。*型* 属性 (これを使用してユーザー定義型の定義を変更できます) については、 77 ページの『型属性 (IBM 拡張)』で説明しています。変数属性 (これを使用して変数の宣言を変更できます) については、 101 ページの『変数属性 (IBM 拡張)』で説明しています。

すべての宣言の形式は、次のとおりです。

データ宣言の構文



暫定定義

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

C 以下のステートメントは、通常の定義と暫定定義を示しています。

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

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

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

関連資料

177 ページの『関数の宣言と定義』

ストレージ・クラス指定子

ストレージ・クラス指定子は、変数、関数、およびパラメーターの宣言を改良するために使用します。ストレージ・クラスは次のことを決定します。

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

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

- auto
- static
- extern
- register
- IBM __thread

関連資料

3 ページの『第 2 章 スコープとリンケージ』

181 ページの『関数のストレージ・クラス指定子』

90 ページの『初期化指定子』

59 ページの『構造体および共用体』

auto ストレージ・クラス指定子

`auto` ストレージ・クラス指定子により、*自動ストレージ* を取る変数を明示的に宣言できます。 `auto` ストレージ・クラスは、ブロック内で宣言される変数の場合はデフォルトです。自動ストレージをもつ変数 `x` は、`x` が宣言されたブロックが終了するときに削除されます。

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

自動変数のストレージ期間

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

再帰的に呼び出す関数内で auto オブジェクトが定義される場合は、ブロックの各呼び出しごとに、メモリーがオブジェクトに割り振られます。

自動変数のリンケージ

auto 変数はブロック・スコープを持ち、リンケージはありません。

関連資料

91 ページの『初期化とストレージ・クラス』
155 ページの『ブロック・ステートメント』
170 ページの『goto ステートメント』
185 ページの『関数からの戻り値』

static ストレージ・クラス指定子

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

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

static ストレージ・クラス指定子は、以下の名前に適用できます。

- データ・オブジェクト
- 無名共用体

以下で static ストレージ・クラス指定子を使用することはできません。

- 型宣言
- 関数仮パラメーター

C C99 言語レベルでは、static キーワードは、関数への配列パラメーターの宣言で使用できます。 static キーワードは、関数に渡される引数は、少なくとも指定されたサイズの配列へのポインターであることを示します。このようにし

て、コンパイラーは、ポインター引数がヌルでないことを知らされます。詳しくは、 187 ページの『関数仮パラメーター宣言の中の静的配列指標 (C のみ)』を参照してください。

静的変数のリンケージ

オブジェクトの、static ストレージ・クラス指定子を含みファイル・スコープを持つ宣言は、その ID の内部リンケージを生じさせます。従って、そのような特定の ID の各インスタンスは 1 つのファイル内でのみ同じオブジェクトを表します。例えば、静的変数 x が関数 f で宣言されている場合、プログラムが f のスコープから出るとき、x は破棄されません。

```
#include <stdio.h>

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

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

上記の例の出力は、以下のとおりです。

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

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

関連資料

8 ページの『プログラム・リンケージ』

181 ページの『static ストレージ・クラス指定子』

91 ページの『初期化とストレージ・クラス』

8 ページの『内部リンケージ』

59 ページの『構造体および共用体』

186 ページの『パラメーター宣言』

194 ページの『main() 関数』

extern ストレージ・クラス指定子

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

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

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

外部変数のストレージ期間

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

外部変数のリンケージ

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

関連資料

8 ページの『プログラム・リンケージ』

9 ページの『外部リンケージ』

91 ページの『初期化とストレージ・クラス』

182 ページの『extern ストレージ・クラス指定子』

194 ページの『main() 関数』

register ストレージ・クラス指定子

register ストレージ・クラス指定子は、コンパイラーに、そのオブジェクトの値はマシン・レジスターに入れることを示します。register ストレージ・クラス指定子は、一般的には、アクセス時間を最小にすることによりパフォーマンスを上げたいときに、使用頻度の高い変数 (ループ制御変数など) に対して指定します。ただし、コンパイラーはこの要求を受け入れる必要はありません。多くのシステムで使用できるレジスターのサイズと数は制限されているので、実際にレジスターに書き込まれる変数は少なくなります。コンパイラーが、マシン・レジスターを register オブジェクトに割り振らない場合、そのオブジェクトはストレージ・クラス指定子 auto が設定されているものとして処理されます。

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

次の制約事項は、register ストレージ・クラス指定子に適用されます。

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

レジスター変数のストレージ期間

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

再帰的に呼び出す関数内で register オブジェクトが定義される場合は、ブロックの各呼び出しごとに、メモリーが変数に割り振られます。

レジスター変数のリンケージ

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

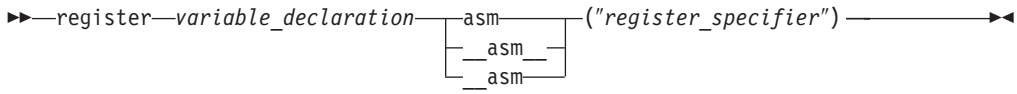
指定したレジスターの変数 (IBM 拡張)

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

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

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

レジスター変数宣言の構文規則



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

r0 から r31
汎用レジスター

f0 から f31
浮動小数点レジスター

v0 から v31
ベクトル・レジスター (選択されたプロセッサー上の)

以下に、レジスター変数の使用に関する規則を示します。

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

関連資料

91 ページの『初期化とストレージ・クラス』
4 ページの『ブロック・スコープ』
アセンブリー・ラベル (IBM 拡張)
172 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』


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

__thread ストレージ・クラス指定子 (IBM 拡張)

__thread ストレージ・クラスは、*スレッド局在*のストレージ期間を持つことで、静的変数を表します。つまり、マルチスレッド・アプリケーションでは、変数の固有のインスタンスがそれを使用する各スレッドごとに作成され、スレッドが終了する

と破棄されます。 __thread ストレージ・クラス指定子は、スレッド・セーフティーを保証する便利な方法を提供します。スレッドごとにオブジェクトを宣言すると、マルチスレッドが競合状態の心配をすることなく、オブジェクトにアクセスすることができます。一方、スレッドの同期化を行う低水準のプログラミングや、大規模なプログラム再構築の必要はありません。

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

```
__thread int i__attribute__((tls_model("local-exec")));
```

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

表 11. スレッド・アクセス・モデルのリンク時/実行時の振る舞い

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

注: __thread キーワードを認識するには、**-qtls** オプションを使用してコンパイルする必要があります。詳細については、「*XL C コンパイラー・リファレンス*」内の **-qtls** を参照してください。

この指定子は、以下のものに適用できます。

- グローバル変数
- ファイル・スコープの静的変数
- 関数スコープの静的変数

関数スコープの自動変数には適用できません。

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

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

__thread 指定子でマークされた変数は、初期化も未初期化も可能です。 C++ __thread 変数は、定数式を使用して初期化する必要があります。静的コンストラクターを持つことはできません。

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

関連資料


「XL C コンパイラー・リファレンス」の 中の『 -qtls』 を参照

型指定子

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

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

関連資料

184 ページの『関数の戻りの型指定子』
70 ページの『typedef 定義』
186 ページの『パラメーター宣言』

整数型

整数型は、次のカテゴリーに分かれます。

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

- unsigned char
- unsigned short int
- unsigned int
- unsigned long int
- unsigned long long int

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

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

関連資料

15 ページの『リテラル』

整数リテラル

108 ページの『整数の変換』
107 ページの『算術変換およびプロモーション』
66 ページの『列挙型』
110 ページの『整数および浮動小数点数のプロモーション』
141 ページの『配列添え字演算子 [ ]』

ブール型

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

C ブール型は C99 のフィーチャーです。ブール変数を宣言するには、_Bool 型指定子を使用してください。IBM トークン bool は、ベクトル宣言コンテキストで使用される場合、およびベクトルのサポートが使用可能になっている場合にのみ、C のキーワードとして認識されます。 IBM C

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

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

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

関連資料

15 ページの『リテラル』

28 ページの『区切り子と演算子』

ブール・リテラル

108 ページの『ブール変換』

55 ページの『ベクトル型 (IBM 拡張)』

110 ページの『整数および浮動小数点数のプロモーション』

123 ページの『論理否定演算子 !』

157 ページの『if ステートメント』

163 ページの『反復ステートメント』

浮動小数点型

浮動小数点型指定子は、次のカテゴリーに分かれます。

- 『実浮動小数点型』
- 53 ページの『複素数浮動小数点型 』

実浮動小数点型

一般型、バイナリー型、浮動小数点型は次の通りです。

- float
- double
- long double

IBM 10 進浮動小数点型は次のとおりです。

- _Decimal32
- _Decimal64
- _Decimal128

注: _Decimal32、Decimal64、および _Decimal128 キーワードを認識するためには、**-qdfp** オプションを使用してコンパイルする必要があります。詳細については、「*XL C コンパイラー・リファレンス*」の **-qdfp** を参照してください。

IBM

実浮動小数点型の大きさの範囲は次のテーブルの通りです。

表 12. 実浮動小数点型の大きさの範囲

Type	範囲
float	約 1.2^{-38} から 3.4^{38}
double、long double	約 2.2^{-308} から 1.8^{308}
_Decimal32	0.000001^{-95} から 9.999999^{96}
_Decimal64	0.000000000000001^{-383} から 9.999999999999999^{384}
_Decimal128	$0.00000000000000000000000000000001^{-6143}$ から $9.999999999999999999999999999999999^{6144}$

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

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

IBM 2 進浮動小数点型をサポートするすべての演算子は 10 進浮動小数点型でも使用できます。また、暗黙的または明示的な変換が、10 進浮動小数点型と、その他のすべての整数型または一般型の小数点型の間で可能です。ただし、10 進浮動小数点型とその他の算術型と同時に使用する場合には次のような制約事項があります。

- 算術式で、10 進浮動小数点型を一般の浮動小数点型または複素数浮動小数点型と混用しないでください。ただし明示的な型変換を使用する場合はこのかぎりではありません。
- 10 進浮動小数点型と実数の 2 進浮動小数点型との間の暗黙的な変換は、単純代入演算子 (=) で割り当てられた場合のみ許可されます。単純代入演算子の割り当てにより実行される暗黙的な変換には、関数の引数割り当ておよび関数からの戻り値も含まれます。

IBM

複素数浮動小数点型

複素数型指定子は、次のとおりです。

- float _Complex
- double _Complex
- long double _Complex

複素数型の表記要件および位置合わせ要件は、対応する実数型の 2 つのエレメントを含む配列型と同じです。実数部は、最初のエレメントと等しく、虚数部は 2 番目のエレメントと一致しています。

等価演算子および比較演算子は、実数型の場合と同じように振る舞います。関係演算子には、オペランドとして複素数型を指定できません。

IBM C99 に対する拡張機能として、複素数を単項演算子 ++ (増分)、--(減分)、および ~ (ビット単位否定) のオペランドとすることもできます。

関連資料

15 ページの『リテラル』

28 ページの『区切り子と演算子』

浮動小数点リテラル

109 ページの『浮動小数点変換』

107 ページの『算術変換およびプロモーション』

複素数リテラル (C のみ)

129 ページの『__real__ および __imag__ 演算子 (IBM 拡張)』

110 ページの『整数および浮動小数点数のプロモーション』

文字型

文字型は、次のカテゴリーに分かれます。

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

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

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

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

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

関連資料

15 ページの『リテラル』
28 ページの『区切り子と演算子』
30 ページの『マルチバイト文字』
文字リテラル
ストリング・リテラル
107 ページの『算術変換およびプロモーション』
110 ページの『整数および浮動小数点数のプロモーション』

void 型

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

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

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

関連資料

39 ページの『データ・オブジェクトの概要』
84 ページの『ポインター』
142 ページの『コンマ演算子 ,』
143 ページの『条件式』
177 ページの『関数の宣言と定義』
112 ページの『void* への変換』
186 ページの『パラメーター宣言』

算術型の互換性 (C のみ)

2 つの算術型は、それらが同じ型の場合のみ、互換性があります。

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

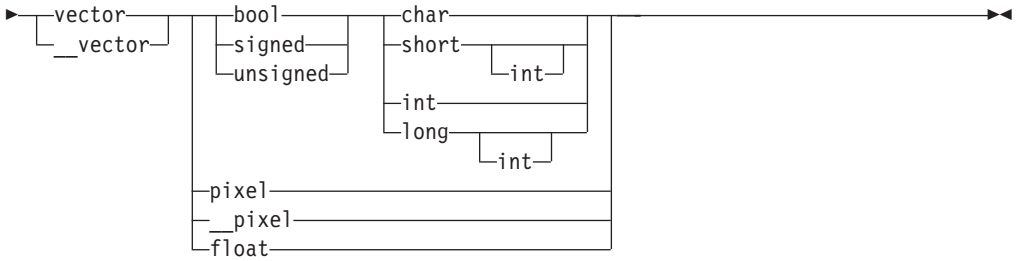
型修飾子があると、型は変更されます。すなわち、const int は int と同じ型ではなく、そのため、この 2 つの型は互換性はありません。

ベクトル型 (IBM 拡張)

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

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

ベクトル宣言の構文



注:

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

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

表 13. ベクトル・データ型

型	内容の解釈	値の範囲
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0, 255
vector unsigned short	8 unsigned short	0..65535
vector unsigned short int		
vector signed short	8 signed short	-32768..32767
vector signed short int		
vector bool short	8 unsigned short	0, 65535
vector bool short int		
vector unsigned int	4 unsigned int	$0..2^{32}-1$
vector unsigned long		
vector unsigned long int		
vector signed int	4 signed int	$-2^{31}..2^{31}-1$
vector signed long		
vector signed long int		
vector bool int	4 unsigned int	0, $2^{32}-1$
vector bool long		
vector bool long int		
vector float	4 float	IEEE-754 値6.80564694 * 10^{+38}。

表 13. ベクトル・データ型 (続き)

型	内容の解釈	値の範囲
vector pixel	8 unsigned short	1/5/5/5 pixel

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

ベクトル・データ型演算子

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

単項演算子の場合、ベクトルに含まれる各エレメントに演算が適用されます。

表 14. 単項演算子

演算子	整数ベクトル型	浮動小数点ベクトル型
++	可	可
––	可	可
+	可	可
–	可	可
~	可	否

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

表 15.

演算子	整数ベクトル型	浮動小数点ベクトル型
+	可	可
–	可	可
*	可	可
/	可	可
%	可	否
&	可	否
|	可	否
^	可	否
<<	可	否
>>	可	否
[]	可	可

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

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

表 16. 関係演算子

演算子	整数ベクトル型	浮動小数点ベクトル型
==	可	可
!=	可	可
<	可	可
>	可	可
<=	可	可
>=	可	可

次のコードの場合

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

c の値は (3,6,9,12)、d の値は (2,3,4,5)、e の値は 1、そして f の値は 3 になります。

ベクトル型キャスト

ベクトル型は他のベクトル型にキャストすることができます。キャストでは変換は行われません。つまり、128 ビット・パターンはそのまま維持されますが、値は必ずしも維持されません。ベクトル型とスカラー型との間でキャストすることはできません。

ベクトル・ポインターと非ベクトル型を指すポインターの間では、相互にキャストすることができます。非ベクトル型を指すポインターをベクトル・ポインターにキャストするときは、アドレスは 16 バイトで位置合わせすることが必要です。非ベクトル型を指すポインターが参照するオブジェクトを 16 バイト境界で位置合わせするには、__align 指定子または __attribute__((aligned(16))) を使用できます。

関連資料

11 ページの『キーワード』
28 ページの『区切り子と演算子』
51 ページの『ブール型』
ベクトル・リテラル
94 ページの『ベクトルの初期化 (IBM 拡張)』
73 ページの『__align 型修飾子 (IBM 拡張)』
102 ページの『aligned 変数属性』
128 ページの『typeof 演算子 (IBM 拡張)』

ユーザー定義の型

以下は、ユーザー定義型です。

- 構造体および共用体
- 列挙型
- typedef 定義

関連資料

77 ページの『型属性 (IBM 拡張)』

構造体および共用体

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

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

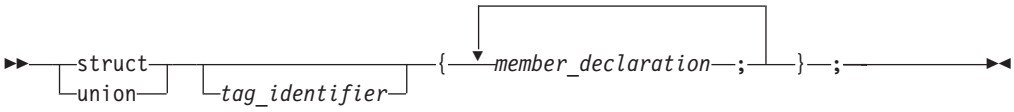
構造体または共用体の型は、『構造体および共用体の型定義』および 64 ページの『構造体および共用体の変数宣言』で説明しているように、その型の変数の定義とは別に宣言することができます。あるいは、構造体または共用体のデータ型とその型のすべての変数を、64 ページの『1 つのステートメントでの構造体および共用体の型定義と変数定義』で説明しているように、1 つのステートメントで定義することができます。

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

構造体および共用体の型定義

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

構造体または共用体の型定義の構文



tag_identifier は、型に名前を付けます。タグ名を指定しない場合は、64 ページの『1 つのステートメントでの構造体および共用体の型定義と変数定義』で説明しているように、その型を参照するすべての変数定義を型の宣言内に入れる必要があります。同様に、構造体または共用体の定義で型修飾子を使用することはできませ

ん。struct または union キーワードの前に置かれた型修飾子は、型定義内で宣言された変数にのみ適用されます。

メンバーの宣言

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

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

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

不完全型はメンバーとして許可されていないので、構造体型または共用体型は、それ自身のインスタンスをメンバーとして持つことはできませんが、それ自身のインスタンスを指すポインターを持つことができます。 特殊なケースとして、複数のメンバーを持つ構造体の最後のエレメントは、不完全な配列型を持つことができます。この型を、柔軟な配列メンバーの説明にあるように*柔軟な配列*メンバー と呼びます。

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

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

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

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

柔軟な配列メンバー

*柔軟な配列*メンバーは、不完全型が含まれている場合でも、構造体が複数の名前付きメンバーで構成されている場合は構造体の最後の構成要素として認められます。

柔軟な配列メンバーは C99 の機能の一つであり、可変長オブジェクトのアクセスにも使用できます。次のように、空の索引を宣言することができます。

array_identifier[ ];

例えば、b は Foo の柔軟な配列メンバーです。

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

柔軟な配列メンバーは不完全型なので、柔軟な配列に sizeof 演算子を適用することはできません。

柔軟な配列メンバーを含む構造体は、別の構造体または配列のメンバーになることはできません。

IBM GNU C との互換性を確保するために、XL C は、標準 C を拡張して柔軟な配列に関する制限を緩和し、以下を可能にしています。

- 柔軟な配列メンバーを、構造体の最後のメンバーとしてだけではなく、構造体のどの部分ででも宣言できます。 柔軟な配列メンバーの後に続くメンバーの型は、その柔軟な配列メンバーの型との互換性は必要ありません。しかし、この場合は警告が出されます。
- 柔軟な配列メンバーを含む構造体を他の構造体のメンバーにすることができます。
- 柔軟な配列メンバーを静的に初期化できます。

次の例では、

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

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

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

柔軟な配列メンバーは、ネストされた構造体の最外部に含まれる場合のみ、初期化できます。内側の構造体のメンバーは初期化できません。 IBM

ゼロ・エクステント配列メンバー (IBM 拡張)

ゼロ・エクステント配列とは、次元のない配列のことです。柔軟な配列メンバーと同様に、ゼロ・エクステント配列は、可変長オブジェクトのアクセスに使用できます。柔軟な配列メンバーと異なり、ゼロ・エクステント配列は C99 機能ではなく、GNU C との互換性の目的で提供されます。

ゼロ・エクステント配列は、次の例のように、その次元にゼロを明示的に宣言する必要があります。

array_identifier[0]

ゼロ・エクステント配列は、構造体の最後のメンバーとしてだけではなく、柔軟な配列メンバーのように構造体のどの部分ででも宣言できます。 ゼロ・エクステント配列の後に続くメンバーの型は、そのゼロ・エクステント配列の型との互換性は必要ありません。しかし、この場合は警告が出されます。

ゼロ・エクステント配列を含む構造体は、柔軟な配列メンバーと異なり、別の配列メンバーになることができます。また、sizeof 演算子はゼロ・エクステント配列に適用できます。戻される値は 0 です。

ゼロ・エクステント配列は、空集合で静的に初期化できるだけです。次に例を示します。

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

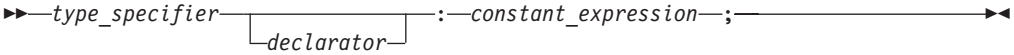
このようにしない場合、動的に割り振られる配列として初期化しなければなりません。

ゼロ・エクステント配列メンバーは、ネストされた構造体の最外部に含まれる場合のみ、初期化できます。内側の構造体のメンバーは初期化できません。

ビット・フィールド・メンバー

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

ビット・フィールド・メンバー宣言の構文



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

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

最大のビット・フィールド長は 64 ビットです。移植性のために、32 ビットを超えるサイズのビット・フィールドを使用しないでください。

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

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

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

次の制約事項は、ビット・フィールドに適用されます。次のことはできません。

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

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

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

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

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

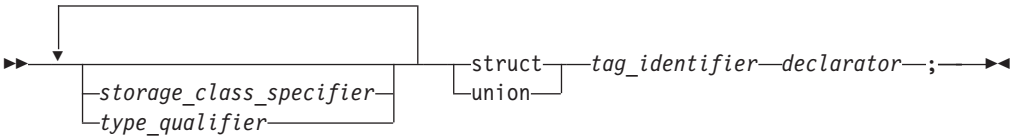
構造体 kitchen には、合計 16 バイトの 8 つのメンバーが含まれます。次の表に、各メンバーが占有するストレージを記述します。

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

構造体および共用体の変数宣言

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

構造体または共用体の変数宣言の構文



tag_identifier は構造体または共用体の、前に定義されたデータ型を示します。

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

次の例は、構造体型 `address` を定義しています。

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

次の例は、型 `address` の 2 つの構造体変数を宣言しています。

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

1 つのステートメントでの構造体および共用体の型定義と変数定義

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

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

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

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

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

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

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

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

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

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

構造体メンバーおよび共用体メンバーへのアクセス

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

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

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

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

無名共用体

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

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

例えば、次のコードでは、データ・メンバー i および cptr に直接アクセスできます。これは、これらのデータ・メンバーが、無名共用体を含むスコープにあるから

です。i と cptr は、共用体メンバーで、同じアドレスが指定されているので、一度にいずれか一方のみしか使用できません。メンバー cptr への代入は、メンバー i の値を変更します。

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

関連資料

78 ページの『aligned 型属性』

79 ページの『packed 型属性』

89 ページの『可変長配列』

「XL C 最適化およびプログラミング・ガイド」のビット・フィールドの位置合わせを参照

102 ページの『aligned 変数属性』

73 ページの『__align 型修飾子 (IBM 拡張)』

104 ページの『packed 変数属性』

95 ページの『構造体および共用体の初期化』

69 ページの『構造体、共用体、および列挙型の互換性 (C のみ)』

120 ページの『ドット演算子 .』

121 ページの『矢印演算子 ->』

43 ページの『ストレージ・クラス指定子』

71 ページの『型修飾子』

44 ページの『static ストレージ・クラス指定子』

70 ページの『typedef 定義』

146 ページの『Cast 演算子 ()』

列挙型

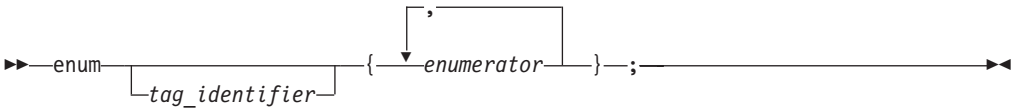
列挙型 とは、整数定数 (*列挙定数* と呼ばれます) を表す、一連の名前付き値で構成されるデータ型です。列挙定数のそれぞれに名前を作成するときに、値のそれぞれをリスト (列挙) しなければならないので、*列挙型* と呼ばれます。整数定数のセットを定義してグループ化する方法を提供することに加えて、列挙型は、起こりうる値の数が少ない変数に対しても有用です。

列挙型は、67 ページの『列挙型の定義』 および 68 ページの『列挙型変数の宣言』で説明しているように、その型の変数の定義とは別に宣言することができます。あるいは、列挙型データ型とその型のすべての変数を、68 ページの『1 つのステートメントでの列挙型と変数の定義』で説明しているように、1 つのステートメントで定義することができます。

列挙型の定義

列挙型の定義には、enum キーワードと、その後に続くオプションの ID (列挙型タグ) および中括弧で囲まれた列挙子のリストが含まれます。コンマは、列挙子のリスト内で各列挙子を分離します。 C99 では、最後の列挙子と右中括弧の間に後書きコンマを使用することができます。

列挙型定義の構文

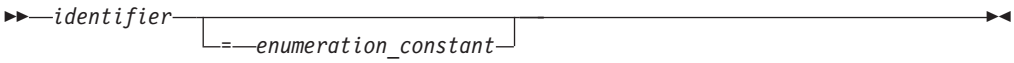


tag_identifier は、列挙型に名前を付けます。タグ名を指定しない場合は、 68 ページの『1 つのステートメントでの列挙型と変数の定義』で説明しているように、その列挙型を参照するすべての変数定義をその型の宣言内に入れる必要があります。同様に、列挙型の定義で型修飾子を使用することはできません。enum キーワードの前に置かれた型修飾子は、型定義内で宣言された変数に適用できるだけです。

列挙型メンバー

列挙型メンバーのリスト、すなわち、*列挙型定数* のリストでは、データ型と値のセットが提供されます。

列挙型メンバー宣言の構文



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

列挙型定数の値は、次の方法で判別されます。

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

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

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

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

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

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

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

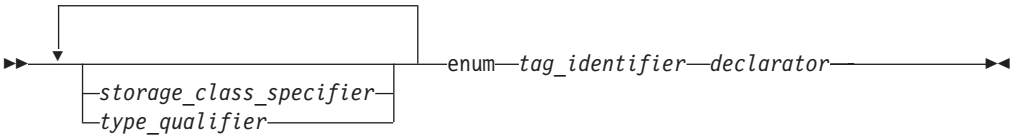
各列挙型定数は、列挙が定義されるスコープ内で固有にする必要があります。次の例では、average と poor の 2 番目の宣言が、コンパイラー・エラーの原因になります。

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

列挙型変数の宣言

列挙データ型を宣言してから、列挙型データ型の変数を定義できます。

列挙型変数宣言の構文



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

1 つのステートメントでの列挙型と変数の定義

変数定義の後に、宣言子とオプションの初期化指定子を使用することによって、型と変数を 1 つのステートメント内で定義できます。変数のストレージ・クラス指定子を指定するには、ストレージ・クラス指定子を宣言の先頭に入れる必要があります。次に例を示します。

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

この例は、次の 2 つの宣言と同等です。

```
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 で、ストレージ・クラス指定子 register、データ型 enum score、および初期値 good を定義します。

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

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

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

関連資料

107 ページの『算術変換およびプロモーション』

50 ページの『整数型』

97 ページの『列挙型の初期化』

『構造体、共用体、および列挙型の互換性 (C のみ)』

110 ページの『整数および浮動小数点数のプロモーション』

構造体、共用体、および列挙型の互換性 (C のみ)

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

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

互換性のある構造体は、互いに割り当てることができます。

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

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

別々のソース・ファイル間の互換性

2 つの構造体、共用体、または列挙型の定義が別々のソース・ファイルで定義されている場合、各ファイルは、同じ名前で同じ型のオブジェクトに対して、理論上は別の定義を有することになります。しかし、この 2 つの宣言には互換性がなければなりません。そうでなければ、プログラムの実行時の動作は未定義になります。そのため、この互換性規則は、同じソース・ファイル内の互換性に関する互換性規則よりも制限がより厳しく、より具体的になります。別々にコンパイルされたファイルで定義された構造型、共用体型、および列挙型につては、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したら、他方も同じタグ付きで宣言すること。
- 両方が完全型の場合、そのメンバーは、数が正確に一致し、互換型で宣言していて、名前が一致すること。

列挙型の場合、対応するメンバーも同じ値を持つこと。

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

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

関連資料

59 ページの『構造体および共用体』

66 ページの『列挙型』

107 ページの『算術変換およびプロモーション』

構造体または共用体の型定義

不完全型

typedef 定義

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

typedef 名のネーム・スペースは、他の ID と同じです。この規則の例外は、typedef 名が可変変更型の場合のみです。この場合、宣言にはブロック・スコープが入ります。

オブジェクトが typedef ID を使用して定義する場合は、定義したオブジェクトの属性は、ID で関連付けられたデータ型を明示的にリストすることによってオブジェクトを定義した場合とまったく同じです。

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

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

IBM

typedef 定義の例

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

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

次の宣言は、上記の宣言と同じです。

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

同様に、typedef は、構造体または共用体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、「パラメーターの指定なしで、int を戻す関数へのポインター」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 であり、ex2 の型は struct b です。

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

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

関連資料

82 ページの『型名』

50 ページの『型指定子』

59 ページの『構造体および共用体』

128 ページの『typeof 演算子 (IBM 拡張)』

型修飾子

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

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

XL C は、次の型修飾子を認識します。

- IBM __align
- const
- restrict
- volatile

const および volatile キーワードをポインターで使用する場合、修飾子の配置は非常に重要です。それが修飾されるポインター自身か、あるいはポインターが指す

オブジェクトかを決定するからです。volatile または const として修飾したいポインターの場合、* と ID の間にキーワードを入れる必要があります。次に例を示します。

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

volatile または const データ・オブジェクトを指すポインターで、型修飾子が * 演算子の後に続いていない場合、型指定子および修飾子を任意の順序で使用できます。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、これら宣言の意味を対比します。

宣言	説明
const int * ptr1;	定数整数を指すポインターを定義します。指される値は変更できません。
int * const ptr2;	整数を指す定数ポインターを定義します。この整数は変更できますが、ptr2 はそれ以外のものを指すことはできません。
const int * const ptr3;	定数整数を指す定数ポインターを定義します。指される値も、ポインター自身も変更できません。

宣言に複数の修飾子を入れることができます。コンパイラーは、重複した型修飾子を無視します。

型修飾子はユーザー定義型には適用できませんが、ユーザー定義型から作成されたオブジェクトにのみ適用できます。したがって、次の宣言は無効です。

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

ただし、変数 (1 つ以上) が型の定義内で宣言される場合、型修飾子をステートメントの先頭に置くか、または変数宣言子の前に置くと、型修飾子はその変数に適用できます。したがって、

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

上記の例では、次の例のストレージと同じストレージを提供します。

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

どちらの例でも、volatile 修飾子は、構造体変数 group に適用されます。

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

関連資料

8 ページの『プログラム・リンケージ』

59 ページの『構造体および共用体』

84 ページの『ポインター』

81 ページの『宣言子の概要』

82 ページの『宣言子の例』

131 ページの『代入演算子』

186 ページの『パラメーター宣言』

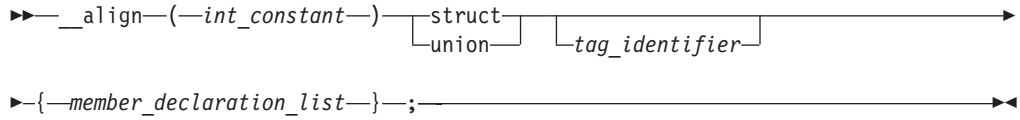
__align 型修飾子 (IBM 拡張)

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

宣言は、次のいずれかの形式をとります。

単純変数のための __align 修飾子の構文

構造体または共用体のための __align 修飾子の構文



ここで、*int_constant* は、バイト位置合わせ境界を示す正の整数値です。有効な値は、1、2、4、8、または 16 です。

以下の制限が適用されます。

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

– 他の型の宣言または定義 (例えば typedef)、関数、または列挙型。

__align 修飾子の使用例

静的変数またはグローバル変数への **__align** の適用:

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

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

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

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

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

集合体メンバーに影響を与えない、集合体タグの位置合わせと埋め込みのための **__align** の適用:

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

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

構造体または共用体への **__align** の適用 (この構造体または共用体を使用する集合体のサイズと位置合わせが影響を受ける場合):

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

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

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

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

配列への **__align** の適用:

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

変数位置合わせのサイズが型位置合わせのサイズと異なる場合の **__align** の適用:

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

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

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

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

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

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

関連資料

55 ページの『ベクトル型 (IBM 拡張)』

59 ページの『構造体および共用体』

102 ページの『aligned 変数属性』

126 ページの『__alignof__ 演算子 (IBM 拡張)』

「XL C 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

78 ページの『aligned 型属性』

79 ページの『packed 型属性』

104 ページの『packed 変数属性』

const 型修飾子

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

const オブジェクトを定数式で使用することはできません。明示ストレージ・クラスがないグローバル const オブジェクトは、デフォルトで extern と見なされます。

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

関連資料

200 ページの『#define ディレクティブ』

restrict 型修飾子

ポインターは、メモリー内のロケーションのアドレスです。複数のポインターがメモリーの同じチャンクにアクセスし、プログラムの途中でそのチャンクを変更することができます。 restrict (または __restrict または __restrict__)[1] 型修飾子は、restrict で修飾されたポインターによってアドレス指定されたメモリーが変更された場合、他のポインターはそのメモリーをアクセスしないことをコンパイラーに伝える指示です。コンパイラーは、restrict で修飾されたポインターに関係のあるコードを、誤った振る舞いをさせないようにする方法で最適化することを選択することができます。 restrict で修飾されたポインターが意図されたとおりに使用されるようにするのは、プログラマーの責任です。そうでないと、振る舞いは未定義になる可能性があります。

メモリーの特定のチャンクが変更されない場合、そのチャンクには、複数の制限付きポインターで別名を割り当てることができます。以下の例は、制限付きポインターを foo() のパラメーターとして示し、無変更のオブジェクトに 2 つの制限付きポインターによって別名をどのように割り当てるかを示しています。

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

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

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

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

注:

1. restrict 修飾子は、次のキーワード (すべて、同じ意味です) によって表されます。
   - restrict キーワードは、**xlc** or **c99** または **-qlanglvl=stdc99** または **-qlanglvl=extc99** オプション、または **-qkeyword=restrict**を使用したコンパイルで認識されます。 __restrict および __restrict__ キーワードは、すべての言語レベルで認識されます。

   関連資料

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

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

volatile 型修飾子

volatile 修飾子は、データ・オブジェクトへのメモリー・アクセスの整合性を保持します。Volatile オブジェクトは、その値が必要になるたびに、メモリーから読み取られ、値が変更されるたびにメモリーに書き戻されます。volatile 修飾子は、その値がコンパイラーが制御できない、または検出できない様々な方法で変更されるデータ・オブジェクト (システム・クロックや別のプログラムによって更新される変数など) を宣言します。これにより、コンパイラーは、オブジェクトの値をメモリーから読み取るのではなく、レジスターに保管し、レジスターから読み取り直すので、オブジェクトを参照するコードを最適化できなくなります。レジスターの中では、オブジェクトの値は変更される可能性があります。

volatile で修飾された左辺値式にアクセスすると、副次作用が発生します。副次作用とは、実行環境の状態が変わることを意味します。

オブジェクト型 ″pointer to volatile″ への参照は最適化されますが、それが指す先のオブジェクトへの参照を最適化することはできません。型「volatile T へのポインター」の値を型「T へのポインター」のオブジェクトに割り当てるには、明示キャストを使用しなければなりません。以下は、volatile オブジェクトの有効な例を示しています。

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

シグナル処理関数は、sig_atomic_t 型変数に volatile が宣言されている場合、その変数に値を保管できます。これは、シグナル処理関数は、静的ストレージ期間の変数にアクセスできないという規則の例外です。

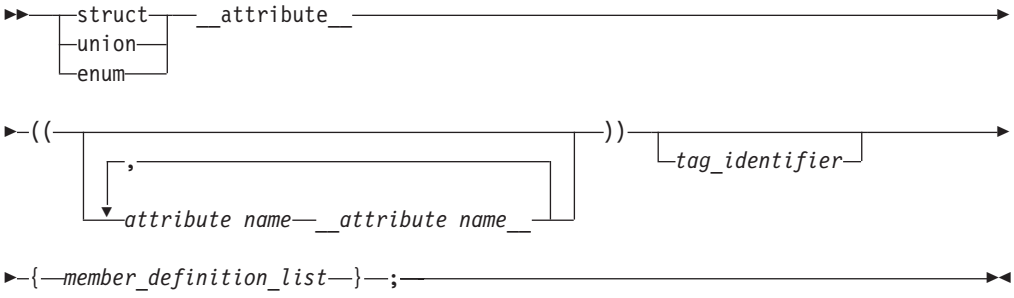
1 つの項目が、const および volatile の両方になり得ます。この場合、その項目はそれ自体のプログラムによって正当に変更することはできないが、ある種の非同期処理によって変更することはできます。

型属性 (IBM 拡張)

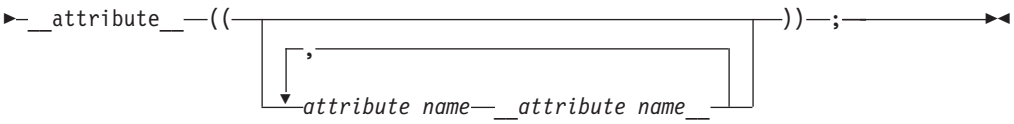
型属性は、GNU C コンパイラーで開発されたプログラムのコンパイルを容易にするために提供された言語拡張機能です。この言語機能により、名前付き属性を使用して、データ・オブジェクトの特殊なプロパティーを指定することができます。*型*属性は、構造体、共用体、列挙型、クラスなどのユーザー定義型の定義、および typedef 定義に適用されます。その型を持つと宣言された変数は、それらに適用される属性を持ちます。

型属性は、キーワード __attribute__、その後に続く属性名、さらに、その属性名が必要とする追加引数によって指定されます。変形はありますが、型属性の一般的な構文形式は次のとおりです。

型属性の構文 - aggregate 型



型属性の構文 - typedef 宣言



attribute name は、前後に 2 つの下線文字を付けても付けなくても指定できます。ただし、2 つの下線を使用すると、同じ名前のマクロと名前が競合する可能性が低くなります。 サポートされない属性名については、XL C コンパイラーは診断メッセージを出して、その属性の指定を無視します。同じ属性指定に複数の属性名を指定することができます。

次の型属性がサポートされています。

- 『aligned 型属性』
- 79 ページの『packed 型属性』
- 79 ページの『transparent_union 型属性 (C のみ)』

関連資料

59 ページの『ユーザー定義の型』

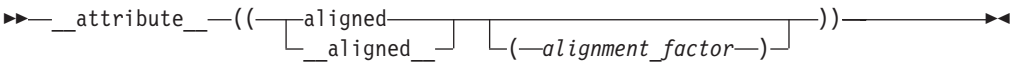
101 ページの『変数属性 (IBM 拡張)』

188 ページの『関数属性 (IBM 拡張)』

aligned 型属性

aligned 型属性は、typedef 宣言で作成された構造体、共用体、列挙型、または他のユーザー定義型について、最小の位置合わせ値をバイト数で指定するデフォルトの位置合わせモードをオーバーライドすることができます。 aligned 属性は、一般的には、その属性が適用される型を宣言された変数の位置合わせを大きくするために使用します。

aligned 型属性の構文



alignment_factor はバイト数ですが、定数式として指定され、その式が評価されて 2 の正の累乗になります。最大 1048576 バイトまでの値を指定できます。位置合わせ係数 (およびそれを囲む括弧) を省略すると、コンパイラーは自動的に 16 バイトを使用します。最大より大きい位置合わせ係数を指定すると、属性指定は無視され、コンパイラーは、単純に、有効になっているデフォルトの位置合わせを使用します。

ここで指定された位置合わせ値は、その型の全インスタンスに適用されます。また、この位置合わせ値は、全体としての変数に適用されます。変数が集合体の場合、位置合わせ値は全体としての集合体に適用されます。集合体の個々のメンバーに適用されるのではありません。

以下の例のすべてにおいて、aligned 属性は構造体型 A に適用されます。a は型 A の変数として宣言されているので、型 A を宣言された他のインスタンスと同様に、a も、この位置合わせ指定を受け取ります。

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

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

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

関連資料

59 ページの『構造体および共用体』

73 ページの『__align 型修飾子 (IBM 拡張)』

102 ページの『aligned 変数属性』

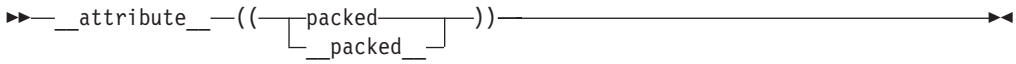
126 ページの『__alignof__ 演算子 (IBM 拡張)』

「XL C 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

packed 型属性

packed 型属性は、構造体、共用体、または列挙型のメンバーに対して最小の位置合わせを使用することを指定します。構造体型または共用体型の場合、位置合わせは、メンバーに対しては 1 バイト、ビット・フィールド・メンバーに対しては 1 ビットです。列挙型の場合、位置合わせは、列挙型の値の範囲を収納する最小のサイズです。 その型の全インスタンスの全メンバーが最小の位置合わせを使用します。

packed 型属性の構文



packed 型属性は、aligned 型属性と違って typedef 宣言では使用できません。

関連資料

59 ページの『構造体および共用体』

73 ページの『__align 型修飾子 (IBM 拡張)』

104 ページの『packed 変数属性』

126 ページの『__alignof__ 演算子 (IBM 拡張)』

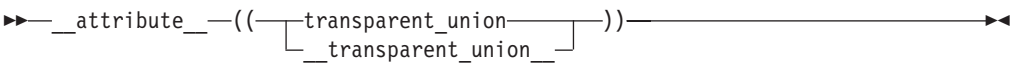
「XL C 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

transparent_union 型属性 (C のみ)

transparent_union 属性を共用体定義または共用体 typedef に適用した場合は、その共用体を*透過共用体* として使用できることを示します。ある関数仮パラメーターの型が透過共用体であるときに、その関数が呼び出された場合は、その透過共用体は、その共用体のいずれかのメンバーの型に一致するすべての型の引数を、明示キャストなしで受け入れることができます。この関数仮パラメーターへの引数は、該当の共用体型の最初のメンバーの呼び出し規則に従って、透過共用体に渡されます。したがって、この共用体のすべてのメンバーについて、マシン表現が同じでな

ければなりません。透過共用体は、互換性の問題を解決するために複数のインターフェースを使用するライブラリー関数の中で使用すると便利です。

transparent_union 型属性の構文



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

transparent_union 型属性が、ネストされた共用体の外部共用体に適用される場合は、その内部共用体のサイズ (つまりその最大メンバー) のサイズを使用して、外部共用体の他のメンバーと同じマシン表現を持っているかどうかが判別されます。次に例を示します。

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

この例では、属性は無視されます。なぜなら、共用体 u_t の最初のメンバー (このメンバー自体も 1 つの共用体です) のマシン表現が 2 バイトであるのに対し、共用体 u_t のその他のメンバーの型は int であり、したがってマシン表現は 4 バイトだからです。

共用体の中の構造体であるメンバーにも、同じ原理が適用されます。型属性 transparent_union が適用された共用体メンバーが struct である場合は、メンバーではなくその struct 全体のマシン表現が考慮に入れられます。

共用体のすべてのメンバーは、共用体の最初のメンバーとマシン表現が同じでなければなりません。つまり、すべてのメンバーが、共用体の最初のメンバーと同じ量のメモリーで表現できることが必要です。最初のメンバーのマシン表現は、共用体の残りのメンバーすべてについて、個々のメンバー最大メモリー・サイズを表します。例えば、transparent_union 属性が適用された共用体の最初のメンバーの型が int であるとすれば、以降のメンバーはすべて最大 4 バイトで表現できることが必要です。この透過共用体では、1、2、または 4 バイトで表現できるメンバーは有効と見なされます。

浮動小数点型 (float、double、float_Complex 、または double _Complex) またはベクトル型は、透過共用体のメンバーでかまいませんが、最初のメンバーであってはなりません。この場合も、透過共用体のすべてのメンバーのマシン表現が、最初のメンバーと同じでなければならないという制約条件は変わりません。

関連資料

113 ページの『関数引数の変換』

146 ページの『Cast 演算子 ()』

第 5 章 宣言子

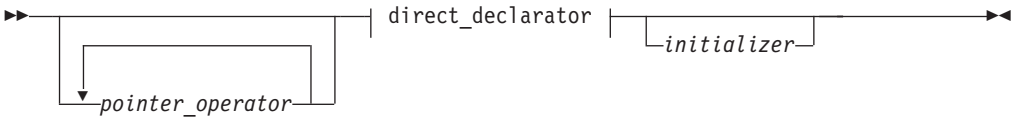
このセクションでは、引き続いてデータ宣言について説明します。ここには、型名、ポインター、配列、、初期化指定子、および変数属性についての情報が記載されています。

宣言子の概要

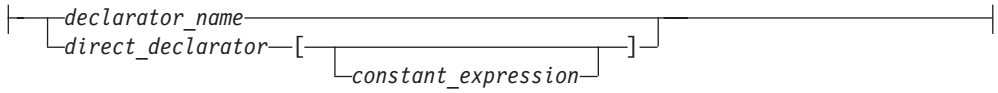
宣言子 は、データ・オブジェクトまたは関数を指定します。宣言子は、初期化も含みます。宣言子は、多くのデータ定義と宣言および一部の型定義で使用します。

データ宣言の場合、宣言子の形式は次のとおりです。

宣言子の構文



direct declarator:



ポインター演算子 (C のみ):

宣言子名 (C のみ):

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

初期化指定子 は 90 ページの『初期化指定子』で説明しています。

次のものは *派生宣言子* 型と呼ばれ、したがって、これらについては、このセクションで説明します。

- 84 ページの『ポインター』
- 87 ページの『配列』

IBM さらに、GNU C の互換性を保つために、XL C では、*変数属性*を使用してデータ・オブジェクトのプロパティーを変更することができます。変数属性は、通常、宣言の中で宣言子の一部として指定されるので、これについては、このセクションの 101 ページの『変数属性 (IBM 拡張)』で説明しています。

関連資料

71 ページの『型修飾子』

宣言子の例

次の表に、宣言で使われる宣言子を示します。

宣言	宣言子	説明
`int owner;`	`owner`	`owner` は、整数データ・オブジェクトです。
`int *node;`	`*node`	`node` は、整数データ・オブジェクトを指すポインターです。
`int names[126];`	`names[126]`	`names` は、126 個の整数エレメントの配列です。
`volatile int min;`	`min`	`min` は、揮発性整数です。
`int * volatile volume;`	`* volatile volume`	`volume` は、整数を指す揮発性ポインターです。
`volatile int * next;`	`*next`	`next` は、揮発性整数を指すポインターです。
`volatile int * sequence[5];`	`*sequence[5]`	`sequence` は、揮発性整数データ・オブジェクトを指す 5 つのポインターの配列です。
`extern const volatile int clock;`	`clock`	`clock` は、静的ストレージ期間および外部リンケージが指定された定数および volatile 整数です。

関連資料

71 ページの『型修飾子』

141 ページの『配列添え字演算子 [ ]』

186 ページの『関数宣言子』

型名

型名 は、オブジェクトを宣言せずに何かを指定する必要がある場合のコンテキストに必要です。例えば、明示的にキャスト式を書く場合、または型に `sizeof` 演算子を適用する場合などです。構文的には、データ型の名前は、その型の関数またはオブジェクトの宣言と同じですが、ID はありません。

型名を正しく読んだり、書いたりするには、構文内に「仮想の」ID を入れ、型名をより単純なコンポーネントに分割してください。例えば、`int` は型指定子ですが、

これは、宣言内で常に ID の左に現れます。この単純な場合には、仮想 ID は不要です。ただし、int *[5] (int への 5 つのポインターの配列) も型の名前です。型指定子 int * は、常に ID の左に現れ、配列添え字演算子は常に右に現れます。この場合には、仮想 ID があると、型指定子を見分けやすくなります。

一般的な規則として、宣言内の ID は、常に、添え字演算子および関数呼び出し演算子の左に現れ、型指定子、型修飾子、または間接演算子の右側に現れます。型名宣言では、添え字演算子、関数呼び出し演算子、および間接演算子のみ使用できます。これらは、通常の演算子優先順位に従ってバインドされます。つまり、間接演算子は添え字演算子または関数呼び出し演算子より優先順位は低く、添え字演算子と関数呼び出し演算子の優先順位は同じです。括弧を使用して間接演算子のバインドを制御することができます。

型名の中に型名を持つことができます。例えば、関数型において、パラメーター型の構文は関数型名内にネストします。同じ経験法則が再帰的に適用されます。

以下の構造体は、型命名規則の適用を示しています。

表 17. 型名

構文	説明
int *[5]	int を指す 5 つのポインターの配列
int (*)[5]	5 つの整数の配列を指すポインター
int (*)[*]	整数の数が指定されていない可変長配列を指すポインター
int *()	int を指すポインターを戻す、パラメーター指定がない関数
int (*)(void)	int を戻す、パラメーターがない関数
int (*const [])(unsigned int, ...)	int を戻す関数を指す定数ポインターの数が指定されていない配列。各関数は、型 unsigned int を指定する 1 つのパラメーターと、数が指定されていない他のパラメーターを取ります。

コンパイラーは関数指定子を関数へのポインターに変えます。この振る舞いは関数呼び出しの構文を単純化します。

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

関連資料

70 ページの『typedef 定義』

149 ページの『演算子優先順位と結合順序』

151 ページの『式と優先順位の例』

118 ページの『括弧で囲んだ式 ( )』

127 ページの『sizeof 演算子』

128 ページの『typeof 演算子 (IBM 拡張)』

146 ページの『Cast 演算子 ()』

ポインター

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

ポインターに共通な用法を次に示します。

- リンクされたリスト、ツリー、キューなどの動的データ構造にアクセスする。
- 配列のエレメント、構造体のメンバーにアクセスする。
- 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。 そのアドレスを通して変数を参照することによって、関数はその変数の内容を変更できます。

型修飾子 `volatile` および `const` の配置は、ポインター宣言のセマンティクスに影響することに注意してください。 `*` の前にどちらの修飾子が現れても、その宣言子は、型で修飾されたオブジェクトを指すポインターを記述します。`*` と ID の間にどちらの修飾子が現れても、その宣言子は、型で修飾されたポインターを記述します。

次の表に、いくつかのポインター宣言の例を示します。

表 18. ポインター宣言

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

関連資料

54 ページの『void 型』

71 ページの『型修飾子』

98 ページの『ポインターの初期化』

87 ページの『ポインターの互換性 (C のみ)』

111 ページの『ポインター型変換』

124 ページの『アドレス演算子 &』

125 ページの『間接演算子 *』

197 ページの『関数を指すポインター』

121 ページの『矢印演算子 ->』

131 ページの『代入演算子』
141 ページの『配列添え字演算子 [ ]』

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 NULL ポインターを割り当てることができます。

IBM ポインター演算は、ベクトル型に対するポインターに定義します。次の場合、

```
vector unsigned int *v;
```

式 v + 1 は、v の次のベクトルへのポインターを表します。

関連資料

122 ページの『増分演算子 ++』
87 ページの『配列』
122 ページの『減分演算子 --』
115 ページの『第 7 章 式と演算子』
121 ページの『単項式』
134 ページの『加法演算子 +』
134 ページの『減法演算子 -』
141 ページの『配列添え字演算子 [ ]』

型ベースの別名割り当て

-qalias=ansi オプションが有効な場合 (デフォルトで有効になります)、コンパイラーは、C 標準による、型ベースの別名割り当て規則に従います。この規則では、ANSI 別名割り当て規則としても知られているように、ポインターは、同じ型のオブジェクトまたは互換性のある型に対してのみ間接参照が可能であると定めています。[1] 互換性のない型にポインターをキャストしてから、それを間接参照するという、一般的なコーディング方法はこのルールに違反しています。 (ただし、char ポインターは、この規則に対し例外であることに注意してください。)

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

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

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

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

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

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

1. C 標準では、オブジェクトの保管値は、次のいずれかの型を持つ左辺値からのみアクセスできると明記されています。
   - 宣言されたオブジェクト型、
   - 宣言されたオブジェクト型の修飾バージョン、
   - 宣言されたオブジェクト型と対応する、符号付きまたは符号なしの型、
   - 宣言されたオブジェクト型の修飾バージョンと対応する、符号付きまたは符号なしの型、
   - メンバー (さらに、副集合体、または包含されている共用体のメンバーを含む) の中に前述の型のいずれかが含まれる構造体または共用体の型または
   - 文字型

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

ポインターの互換性 (C のみ)

同じ型修飾子を指定された 2 つのポインター型は、それらが互換型のオブジェクトを指している場合、互換性があります。 2 つの互換ポインター型に関する複合型は、複合型への、同じように修飾されたポインターです。

次の例では、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 */
```

関連資料

39 ページの『データ・オブジェクトの概要』

84 ページの『ポインター』

配列

配列 とは、メモリー内で連続して割り振られた、同じデータ型のオブジェクトの集合です。配列内の個々のオブジェクトは *エレメント* と呼ばれ、それらは、配列内のそれぞれの位置によってアクセスされます。サブスクリプト演算子 ([]) は、配列エレメントへの指標を作成する方法を提供します。このアクセス方式は、*指標方式* または *添え字方式* と呼ばれます。配列では、各エレメントに実行されるステートメント群をループに入れて、そのループを配列内の各エレメントに繰り返すようにできるので、反復タスクのコーディングが容易になります。

C 言語では、配列型に対して、限定された組み込みサポート (すなわち個々のエレメントの読み取りおよび書き込みのサポート) が提供されます。ある配列を別の配列に割り当てたり、2 つの配列を同等のものかどうか比較したり、自己認識サイズを戻したりする操作はサポートされていません。

配列の型はエレメントの型から派生し、このことは、*配列型派生* と呼ばれます。配列オブジェクトが不完全型の場合、配列型も不完全と見なされます。配列エレメントは、void 型にも関数型にもすることができません。ただし、関数へのポインター配列は許可されます。

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

配列添え字宣言子の構文

constant_expression は定数整数式であって、配列のサイズを示し、正でなければなりません。

宣言がブロック・スコープまたは関数スコープで行われる場合、配列添え字宣言子に対して定数式以外の式を指定することができます。その配列は、89 ページの『可変長配列』で説明しているように、*variable-length array* と見なされます。

添え字宣言子は、配列の次元の数と各次元のエレメントの数を記述します。大括弧で囲まれた式、または添え字は、別の次元を表します。これらは、定数式でなければなりません。

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

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

各次元の 1 番目の添え字は、0 です。配列 list には以下のエレメントが含まれます。

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

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

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

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

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

ストレージ内では、roster のエレメントは、以下のように保管されます。

```
 |               |               |

 └───────────────┴───────────────┘
 ________________
 ↑               ↑               ↑
 |               |               |
 roster[0][0]    roster[0][1]
 roster[1][0]
```

以下の場合には、最初の (最初のみ) 添え字の大括弧のセットを空にしたままにすることができます。

- 初期化を含む配列定義
- extern 宣言
- パラメーター宣言

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

関連資料

85 ページの『ポインター演算』

141 ページの『配列添え字演算子 [ ]』

98 ページの『配列の初期化』

111 ページの『ポインター型変換』

115 ページの『左辺値と右辺値』

125 ページの『間接演算子 *』

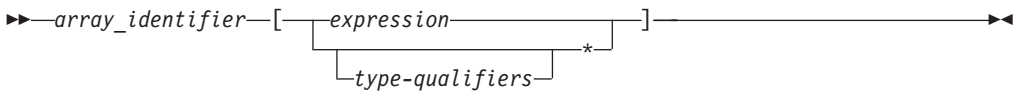
127 ページの『sizeof 演算子』

186 ページの『パラメーター宣言』

可変長配列

可変長配列 (C99 の機能) は自動ストレージ期間の配列であって、その長さは実行時に決定されます。

可変長配列宣言子の構文



可変長配列のサイズが式ではなく * で示されている場合は、その配列のサイズは指定されていないものと見なされます。このような配列は完全型と見なされますが、関数プロトタイプ・スコープの宣言の中でしか使用できません。

可変長配列および可変長配列へのポインターは、*可変変更型* と見なされます。可変変更型の宣言は、ブロック・スコープまたは関数プロトタイプ・スコープになければなりません。 extern ストレージ・クラス指定子を宣言される配列オブジェクトは、可変長配列型であってはなりません。 static ストレージ・クラス指定子で宣言された配列オブジェクトは、可変長配列へのポインターになりますが、実際の可変長配列であってはなりません。可変長配列は初期化することはできません。

可変長配列は、sizeof 式のオペランドとして使用できます。この場合は、オペランドは実行時に評価されます。したがって、可変長配列の各インスタンスのサイズがその存続期間中に変化しなかったとしても、そのサイズは整数定数でも定数式でもありません。

可変長配列は typedef ステートメントの中で使用できます。typedef 名は、ブロック・スコープだけを持ちます。配列の長さは、typedef 名が定義されたときに固定されるのであって、それが使用されるたびに固定されるのではありません。

関数仮パラメーターは可変長配列となることができます。必要なサイズ式は、関数定義の中で指定する必要があります。コンパイラーは、関数に入るときに、変更されたパラメーターのサイズ式を評価します。以下のように、関数が可変長配列をパラメーターとして宣言されているとします。

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

この場合、可変長配列引数のサイズは、関数定義のサイズと一致していなければなりません。

関連資料

59 ページの『構造体および共用体』

柔軟な配列メンバー (C のみ)

配列の互換性

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

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

これらには、互換性はありません。

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

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

この場合、ex3 および ex4 の複合型は char[42] です。元の型の 1 つが可変長配列の場合、複合型はその型になります。

関連資料

39 ページの『データ・オブジェクトの概要』

9 ページの『外部リンケージ』

初期化指定子

初期化指定子 は、データ・オブジェクトの初期値を指定する、データ宣言のオプションの部分です。特定の宣言で有効な初期化指定子は、初期化されるオブジェクトの型とストレージ・クラスに依存します。

初期化指定子は、= シンボルと、その後に続く、初期式 (*expression*)、またはコンマで区切られた初期式の中括弧で囲まれたリストで構成されます。個々の式は、コンマで区切られる必要があります。式のグループは中括弧で囲み、コンマで区切ることができます。文字ストリングの初期化指定子がストリング・リテラルの場合、中

括弧 ({ }) はオプションです。初期化指定子の数は、初期化されるエレメントの数よりも多くてはいけません。初期式は、データ・オブジェクトの最初の値に評価されます。

算術型またはポインター型に値を代入するには、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 にセットします。

```
int group = 3;
```

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

『初期化とストレージ・クラス』で、変数のストレージ・クラスに従った初期化の規則を説明しています。

92 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』 で、指定された初期化指定子を説明しています。これは、配列、構造体、および共用体を初期化するために使用できる C99 の機能です。

以下のセクションで、派生型に対する初期化を説明しています。

- 94 ページの『ベクトルの初期化 (IBM 拡張)』
- 95 ページの『構造体および共用体の初期化』
- 98 ページの『ポインターの初期化』
- 98 ページの『配列の初期化』

関連資料

43 ページの『ストレージ・クラス指定子』

初期化とストレージ・クラス

自動変数の初期化

auto 変数 (関数仮パラメーターを除く) は、初期化できます。自動オブジェクトを明示的に初期化しない場合は、その値は確定できません。初期値を提供する場合は、初期値を表す式を C 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。

静的変数の初期化

静的オブジェクトの初期化を、定数式、または既に extern または static と宣言されているオブジェクトのアドレスに変換する式 (多く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、それがポインターでなければ、その初期値は、該当する型の値ゼロになります。その変数がポインターの場合は、NULL に初期化されます。

ブロック内の static 変数は、プログラム実行の前に 1 回初期化されるだけですが、初期化指定子を持つ auto 変数は、それが存在するようになるたびに初期化されます。

外部変数の初期化

extern ストレージ・クラス指定子を持つオブジェクトは、で初期化できます。extern オブジェクトの初期化指定子は次のどちらかでなければなりません。

- 定義の一部として現れ、定数式によって初期値が記述されていなければならない。または
- 静的ストレージ期間を持つ、既に宣言済みのオブジェクトのアドレスに変換されなければならない。このオブジェクトは、ポインター演算によって変更することができます。 (言い換えると、オブジェクトは、整数定数式の加算または減算によって変更することができます。)

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

レジスター変数の初期化

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

関連資料

43 ページの『auto ストレージ・クラス指定子』

44 ページの『static ストレージ・クラス指定子』

45 ページの『extern ストレージ・クラス指定子』

46 ページの『register ストレージ・クラス指定子』

集合体型に対する、指定された初期化指定子 (C のみ)

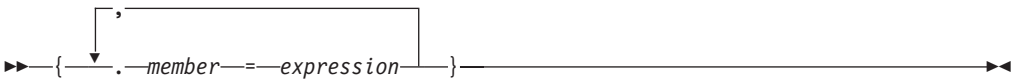
集合体型 (配列、構造体、共用体など) 用に、*指定された初期化指定子* (C99 の機能) がサポートされています。指定された初期化指定子、すなわち、*指定子* は、初期化する特定のエレメントを示します。*指定子リスト* は、1 つ以上の指定子をコンマで区切ったリストです。直後に等号が記入された指定子リストによって、*指定* が構成されます。

指定された初期化指定子は次の柔軟性を考慮しています。

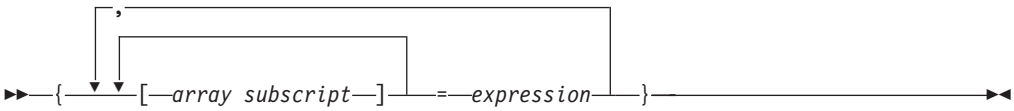
- 集合体内のエレメントは、任意の順に初期化できます。
- 初期化指定子リストは、集合体の終わりに宣言されるエレメントのみでなく、集合体の任意の場所に宣言されるエレメントも省略できます。省略されたエレメントは、それが静的オブジェクトであるかのように初期化されます。すなわち、算術型は 0 に初期化され、ポインターは NULL に初期化されます。

- 多次元配列またはネストされた集合体に対する初期化指定子の大括弧の使用法が矛盾しているか、または不完全であるかを理解することが難しそうな場所では、指定子は、初期化するエレメントまたはメンバーをより明確に識別することができます。

構造体または共用体に対する指定子リストの構文



配列に対する指定子リストの構文



以下の例において、指定子は .any_member であり、指定された初期化指定子は .any_member = 13 です。

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

次の例は、構造体変数 klm の 2 番目と 3 番目のメンバー b および c は、指定された初期化指定子で初期化されることを示しています。

```
struct xyz {
       int a;
       int b;
       int c;
       } klm = { .a = 99, .c = 100 };
```

次の例では、1 次元の配列 aa の 3 番目と 2 番目のエレメントは、それぞれ、 3 および 6 に初期化されます。

```
int aa[4] = { [2] = 3, [1] = 6 };
```

次の例は、最初の 4 つのエレメントと最後の 4 つのエレメントを初期化し、中間の 4 つのエレメントを省略しています。

```
      static short grid[3] [4] = { [0][0]=8, [0][1]=6,
                                   [0][2]=4, [0][3]=1,
                                   [2][0]=9, [2][1]=3,
                                   [2][2]=1, [2][3]=1 };
```

grid の、省略された 4 つのエレメントはゼロに初期化されます。

エレメント	値	エレメント	値
grid[0] [0]	8	grid[1] [2]	0
grid[0] [1]	6	grid[1] [3]	0
grid[0] [2]	4	grid[2] [0]	9
grid[0] [3]	1	grid[2] [1]	3
grid[1] [0]	0	grid[2] [2]	1

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

次の例のように、指定された初期化指定子は通常の初期化指定子と結合することができます。

```
int a[10] = {2, 4, [8]=9, 10}
```

この例では、a[0] は 2 に初期化され、a[1] は 4 に初期化されます。a[2] から a[7] は 0 に初期化され、 a[9] は 10 に初期化されます。

この例では、単一の指定子を使用して、配列の両端からスペースを「割り振って」います。

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

指定された初期化指定子 [MAX-5] = 8 は、添え字 MAX-5 の配列エレメントを値 8 に初期化することを意味します。MAX が 15 の場合、a[5] から a[9] は、ゼロに初期化されます。MAX が 7 の場合、a[2] から a[4] は、最初は、それぞれ、値 5、7、および 9 になり、それらは、値 8、6、および 4 でオーバーライドされます。つまり、MAX が 7 の場合、初期化は、宣言が次のように書かれるのと同じです。

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

指定子を使用して、ネストされた構造体のメンバーを表すこともできます。次に例を示します。

```
struct a {
         struct b {
              int c;
              int d;
          } e;
         float f;
} g = {.e.c = 3 };
```

この例は、構造体変数 g のメンバーである構造体変数 e のメンバー c を値 3 に初期化します。

関連資料

95 ページの『構造体および共用体の初期化』

98 ページの『配列の初期化』

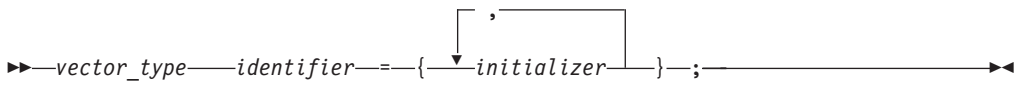
ベクトルの初期化 (IBM 拡張)

ベクトル型は、ベクトル・リテラル、または同じベクトル型を持つ任意の式により初期化されます。次に例を示します。

```
vector unsigned int v1;
vector unsigned int v2 = (vector unsigned int)(10);
v1 = v2;
```

XL C は AltiVec 指定を、初期化指定子リストによってベクトル型を初期化できるように拡張しました。この機能は、GNU C との互換性のための拡張機能です。

ベクトル初期化指定子リストの構文



中括弧に入れた初期化指定子リスト内の値の数は、該当のベクトル型のエレメントの数以下でなければなりません。未初期化エレメントはゼロに初期化されます。

以下は、初期化指定子リストを使用したベクトル初期化の例です。

```
vector unsigned int v1 = {1};  // initialize the first 4 bytes of v1 with 1
                               //  and the remaining 12 bytes with zeros

vector unsigned int v2 = {1,2};  // initialize the first 8 bytes of v2 with 1 and 2
                                 // and the remaining 8 bytes with zeros

vector unsigned int v3 = {1,2,3,4};  // equivalent to the vector literal
                                     // (vector unsigned int) (1,2,3,4)
```

初期化指定子リスト内の値は、ベクトル・リテラルとは異なり、初期化されるベクトル変数が静的期間を持っているのでなければ、定数式である必要はありません。したがって、以下の指定は有効です。

```
int i=1;
int foo() { return 2; }
int main()
{
   vector unsigned int v1 = {i, foo()};
   return 0;
}
```

関連資料

28 ページの『区切り子と演算子』

55 ページの『ベクトル型 (IBM 拡張)』

構造体および共用体の初期化

構造体の初期化指定子は、中括弧で囲まれた、コンマで区切られた値のリストです。共用体の場合、中括弧に入れられた単一の値です。初期化指定子の前には等号 (=) を付けます。

C99 では、共用体型または構造体型の自動メンバー変数の初期化指定子は定数式または非定数式にすることができます。

構造体および共用体の初期化指定子は、次の 2 とおりの方法で指定できます。

- C89 スタイルの初期化指定子については、構造体メンバーは、宣言された順に初期化されなければならず、共用体の場合、最初のメンバーのみ初期化できます。
- *指定された* 初期化指定子、つまり初期化されるメンバーに*名前* を付けることを可能にする C99 フィーチャーを使用すると、構造体メンバーを任意の順序で初期化することができ、かつ共用体の任意の (単一) メンバーを初期化することができます。指定された初期化指定子については、92 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』 で詳しく説明しています。

次の例では、C89 スタイルの初期化を使用して、共用体変数 people の最初の共用体メンバーである birthday を初期化する方法を示しています。

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

C 指定された初期化指定子を同じ例に使用すると、以下は 2 番目の共用体メンバー age を初期化します。

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = { .age = 14 };
```

次の定義では、完全に初期化される構造体を示します。

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

perm_address の値は、次のとおりです。

メンバー	値
perm_address.street_no	3
perm_address.street_name	ストリング "Savona Dr." のアドレス
perm_address.city	ストリング "Dundas" のアドレス
perm_address.prov	ストリング "Ontario" のアドレス
perm_address.postal_code	ストリング "L4B 2A1" のアドレス

名前なし構造体メンバーまたは共用体メンバーは初期化には関与せず、初期化後に中間値を持ちます。従って、次の例では、ビット・フィールドは初期化されず、初期化指定子 3 は、メンバー b に適用されます。

```
struct {
     int a;
     int :10;
     int b;
     } w = { 2, 3 };
```

構造体または共用体の全メンバーを初期化する必要はありません。初期化されていない構造体メンバーの初期値は、その構造体変数または共用体変数に関連付けられたストレージ・クラスによって異なります。静的と宣言された構造体では、初期化されないメンバーはすべて、該当の型のゼロに暗黙的に初期化されます。自動ストレージが指定された構造体のメンバーは、デフォルトの初期化は行われません。静的ストレージが指定された共用体のデフォルトの初期化指定子は、最初のコンポーネントのデフォルトです。自動ストレージが指定された共用体は、デフォルトの初期化は行われません。

次の定義では、一部のみが初期化される構造体を示します。

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

temp_address の値は、次のとおりです。

メンバー	値
temp_address.street_no	44
temp_address.street_name	ストリング "Knyvet Ave." のアドレス
temp_address.city	ストリング "Hamilton" のアドレス
temp_address.prov	ストリング "Ontario" のアドレス
temp_address.postal_code	temp_address 変数のストレージ・クラスによって異なります。それが静的の場合、値は NULL になります。

C temp_address 変数の 3 番目と 4 番目のメンバーのみを初期化するには、指定された初期化指定子リストを次のように使用します。

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

関連資料

59 ページの『構造体および共用体』

92 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』

構造体および共用体の変数宣言

131 ページの『代入演算子』

列挙型の初期化

列挙変数の初期化指定子には、= シンボルと、その後に続く式 *enumeration_constant* が含まれます。

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

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

関連資料

66 ページの『列挙型』

ポインターの初期化

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

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

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

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

は、以下と同等です。

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

初期化指定子のストリング定数を指定することによって、ストリング定数の 1 番目の文字のアドレスをポインターに割り当てることができます。次の例では、ポインター変数 string とストリング定数 "abcd" を定義します。ポインター string は、ストリング "abcd" の文字 a を指すように初期化されます。

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

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

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

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

関連資料

84 ページの『ポインター』

配列の初期化

配列の初期化指定子は、中括弧 ({ }) で囲まれ、コンマで区切られた定数式のリストです。初期化指定子の前には等号 (=) を付けます。配列内のすべてのエレメントを初期化する必要はありません。配列が部分的に初期化されている場合は、初期化されていないエレメントの値は、該当の型の値 0 となります。静的ストレージ期間

を持つ配列のエレメントについても、同じことが言えます。 (静的ストレージ期間を持つのは、static キーワードを指定して宣言されているすべてのファイル・スコープ変数および関数スコープ変数です。)

配列の初期化指定子は、次の 2 とおりの方法で指定できます。

- C89 スタイルの初期化指定子の場合、配列エレメントは添え字順に初期化しなければなりません。
- *指定された* 初期化指定子を使用する場合、初期化する添え字エレメントの値を指定することができ、配列エレメントを任意の順序で初期化することができます。指定された初期化指定子については、92 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』 で詳しく説明しています。

次の定義では、C89 スタイルの初期化指定子を使用して、完全に初期化される 1 次元の配列を示しています。

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

配列 number には、次の値が含まれます。すなわち、number[0] は 5、number[1] は 7、number[2] は 2 です。エレメントの数 (この例では、3) を定義する添え字宣言子の中で式を使用する場合、配列のエレメントの数より多い初期化指定子を使用することはできません。

次の定義は、一部のみが初期化される 1 次元配列を示しています。

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

number1[0] および number1[1] の値は前の定義のときと同じですが、number1[2] は 0 です。

C 次の定義では、明示的に初期化しない配列のエレメントをスキップオーバーするための、指定された初期化指定子の使用法を示しています。

```
static int number[3] = { [0] = 5, [2] = 7 };
```

配列 number には、次のような値が入れられます。すなわち、 number[0] は 5、number[1] は暗黙的に 0 に初期化され、number[2] は 7 です。 C

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

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

サイズが指定されておらず、5 つの初期化指定子があるので、コンパイラーは item に、初期化された 5 つのエレメントを与えます。

文字配列の初期化

次のものを指定して、1 次元の文字配列を初期化できます。

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

ストリング定数を初期化すると、空いている場所がある場合または配列の次元が指定されていない場合は、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";
```

多次元配列の初期化

次の方法のいずれかにより、多次元配列を初期化できます。

- 初期化するすべてのエレメントの値を、コンパイラーが値を割り当てる順序でリストする。コンパイラーは、最後の次元の添え字を最も速く増やして、値を割り当てます。この形式の多次元配列の初期化は、1 次元配列の初期化と似ています。次の定義では、配列 month_days を完全に初期化します。

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

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

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

- ネストされた中括弧を使用して、次元および次元のエレメントを選択的に初期化する。次の例では、配列 grid の最初の 8 つのエレメントのみが、明示的に初期化されます。明示的に初期化されない残りの 4 つのエレメントは、自動的にゼロに初期化されます。

  ```
  static short grid[3] [4] = {8, 6, 4, 1, 9, 3, 1, 1};
  ```

  grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	8	grid[1] [2]	1

エレメント	値	エレメント	値
grid[0] [1]	6	grid[1] [3]	1
grid[0] [2]	4	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	9	grid[2] [2]	0
grid[1] [1]	3	grid[2] [3]	0

- C *指定された* 初期化指定子の使用。以下の例は、指定された初期化指定子を使用して、配列の最後の 4 つのエレメントだけを明示的に初期化します。明示的に初期化されない最初の 8 個のエレメントは、自動的にゼロに初期化されます。

```
static short grid[3] [4] = { [2][0] = 8, [2][1] = 6,
                             [2][2] = 4, [2][3] = 1 };
```

grid の初期値は、次のとおりです。

エレメント	値	エレメント	値
grid[0] [0]	0	grid[1] [2]	0
grid[0] [1]	0	grid[1] [3]	0
grid[0] [2]	0	grid[2] [0]	8
grid[0] [3]	0	grid[2] [1]	6
grid[1] [0]	0	grid[2] [2]	4
grid[1] [1]	0	grid[2] [3]	1

関連資料

87 ページの『配列』

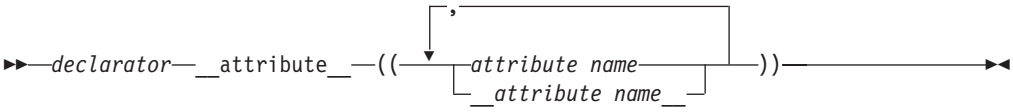
92 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』

変数属性 (IBM 拡張)

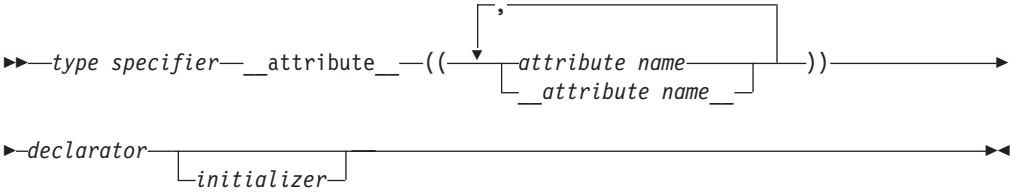
変数属性は、GNU C コンパイラーで開発されたプログラムのコンパイルを容易にするために提供された言語拡張機能です。この言語機能により、名前付き属性を使用して、データ・オブジェクトの特殊なプロパティーを指定することができます。変数 属性は、単純な変数、集合体、および集合体のメンバー変数の宣言に適用されます。

変数属性は、キーワード __attribute__、その後に続く属性名、さらに、その属性名が必要とする追加引数によって指定されます。変数の __attribute__ 指定は、変数の宣言の中に組み込み、宣言子の前または後に置くことができます。変形はありますが、構文形式は一般的に次のどちらかです。

変数属性の構文: 宣言子の後



変数属性の構文: 宣言子の前



attribute name は、前後に 2 つの下線文字を付けても付けなくても指定できます。ただし、2 つの下線文字を使用すると、同じ名前のマクロと名前が競合する可能性が低くなります。サポートされない属性名については、XL C コンパイラーは診断メッセージを出して、その属性の指定を無視します。同じ属性指定に複数の属性名を指定することができます。

単一の宣言行の、コンマで区切った宣言子のリストにおいて、変数属性がすべての宣言子の前にある場合、その変数属性は、宣言内のすべての宣言子に適用されます。属性が宣言子の後にある場合、その属性は、直前の宣言子にのみ適用されます。次に例を示します。

```
struct A  {

 int b __attribute__((aligned));          /* typical placement of variable */
                                                 /* attribute */

 int __attribute__((aligned))__ c = 10;   /* variable attribute can also be */
                                            /* placed here */

 int d, e, f __attribute__((aligned));    /* attribute applies to f only */

 int g __attribute__((aligned)), h, i;    /* attribute applies to g only */

 int __attribute__((aligned)) j, k, l;    /* attribute applies to j, k, and l */

 };
```

次の変数属性がサポートされています。

関連資料

77 ページの『型属性 (IBM 拡張)』

188 ページの『関数属性 (IBM 拡張)』

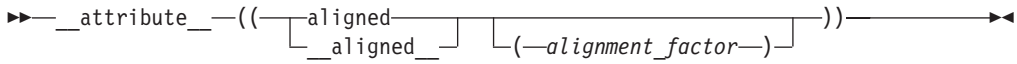
aligned 変数属性

aligned 変数属性は、次のいずれかについて、最小の位置合わせ値をバイト数で指定するデフォルトの位置合わせモードをオーバーライドすることができます。

- 集合体以外の変数
- 集合体変数 (構造体、共用体など)
- 選択されたメンバー変数

この属性は、一般的には、特定の変数の位置合わせを大きくするために使用します。

aligned 変数属性の構文



alignment_factor はバイト数ですが、定数式として指定し、その式が評価されて 2 の正の累乗になります。最大 1048576 バイトまでの値を指定できます。位置合わせ係数 (およびそれを囲む括弧) を省略すると、コンパイラーは自動的に 16 バイトを使用します。最大より大きい位置合わせ係数を指定すると、属性指定は無視され、コンパイラーは、単純に、有効になっているデフォルトの位置合わせを使用します。

aligned 属性をビット・フィールド構造体メンバーの変数に適用すると、属性指定は、ビット・フィールドのコンテナー に適用されます。コンテナーのデフォルトの位置合わせが位置合わせ係数より大きい場合、デフォルトの位置合わせが使用されます。

次の例では、構造体 first_address および second_address は 16 バイトの位置合わせに設定されています。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov;
                  char *postal_code;
                } first_address __attribute__((__aligned__(16))) ;

struct address second_address __attribute__((__aligned__(16))) ;
```

次の例では、メンバー first_address.prov および first_address.postal_code のみが 16 バイトの位置合わせに設定されています。

```
struct address {
                  int street_no;
                  char *street_name;
                  char *city;
                  char *prov __attribute__((__aligned__(16))) ;
                  char *postal_code __attribute__((__aligned__(16))) ;
                } first_address  ;
```

関連資料

55 ページの『ベクトル型 (IBM 拡張)』

59 ページの『構造体および共用体』

73 ページの『__align 型修飾子 (IBM 拡張)』

78 ページの『aligned 型属性』

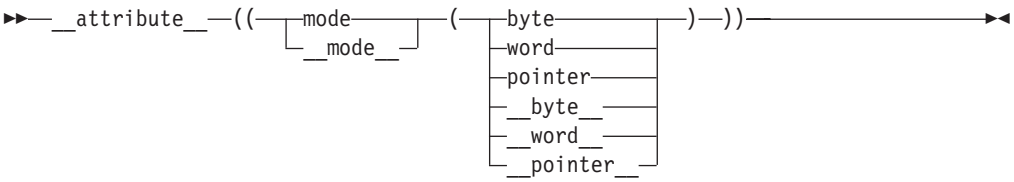
「XL C 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

126 ページの『__alignof__ 演算子 (IBM 拡張)』

mode 変数属性

変数属性 mode は変数宣言内の型指定子をオーバーライドして、特定の整数型のサイズを指定することができます。

mode 変数属性の構文



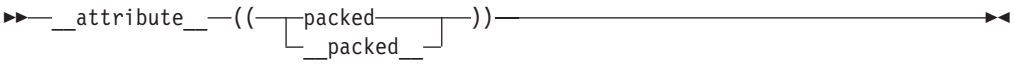
モードの有効な引数は、特定の幅を示す以下の型指定子のいずれかです。

- byte は 1 バイトの整数型を意味します。
- word は 4 バイトの整数型を意味します。
- pointer は、32 ビット・モードでは 4 バイトの整数型を表し、 64 ビット・モードでは 8 バイトの整数型を表します。

packed 変数属性

変数属性 packed はデフォルトの位置合わせをオーバーライドして、集合体の全メンバーまたは集合体の選択されたメンバーの位置合わせを可能な最小の位置合わせに縮小することができます。すなわち、メンバーの場合は 1 バイトに、ビット・フィールド・メンバーの場合は 1 ビットに縮小します。

packed 変数属性の構文



関連資料

59 ページの『構造体および共用体』

79 ページの『packed 型属性』

73 ページの『__align 型修飾子 (IBM 拡張)』

「XL C 最適化およびプログラミング・ガイド」の『位置合わせデータ』を参照

126 ページの『__alignof__ 演算子 (IBM 拡張)』

tls_model 属性

tls_model 属性を使用すると、特定の変数に対して使用されるスレッド・ローカル・ストレージ・モデルをソース・レベルで制御することができます。**tls_model** 属性には、local-exec、initial-exec、local-dynamic、または global-dynamic アクセス

方式のいずれか 1 つを指定する必要があります。これは、その変数の **-qtls** オプションをオーバーライドします。次に例を示します。

```
__thread int i__attribute__((tls_model("local-exec")));
```

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

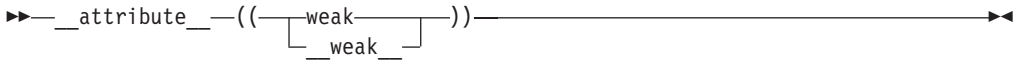
表 19. スレッド・アクセス・モデルのリンク時/実行時の振る舞い

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

weak 変数属性

weak 変数属性があると、変数宣言の結果によるシンボルは、オブジェクト・ファイルの中にグローバル・シンボルとしてではなく、弱いシンボルとして現れます。この言語フィーチャーは、ライブラリー関数を書くプログラマーが、ユーザー・コード内の変数定義でライブラリー宣言をオーバーライドした場合に、重複した名前エラーを回避するための機能です。

weak 変数属性の構文



関連資料

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

193 ページの『weak 関数属性』
189 ページの『alias 関数属性』

第 6 章 変換

所定の型の式は、以下の状況にあるとき、*暗黙的に変換* されます。

- 式が算術演算または論理演算のオペランドとして使用される場合。
- 式が `if` ステートメントまたは反復ステートメント (`for` ループなど) の中で条件として使用される場合。この式はブール (C89 では、整数) に変換されます。
- 式が `switch` ステートメントの中で使用される場合。この式は整数型に変換されます。
- 式が初期化として使用される場合。これには、次の場合が含まれます。
  - 代入される値とは型が異なる左辺値に対して、代入が行われる。
  - 関数に、パラメーターとは異なる型を持つ引数値が与えられる。
  - 関数の `return` ステートメントに指定された値が、その関数用に定義されている戻りの型と異なった型になっている。

145 ページの『キャスト式』で説明しているように、*cast* 式を使用して*明示的* 型変換を行うことができます。

関連資料

159 ページの『switch ステートメント』
157 ページの『if ステートメント』
169 ページの『return ステートメント』

算術変換およびプロモーション

以下のセクションでは、算術型に関する標準の変換の規則を説明します。

- 108 ページの『整数の変換』
- 109 ページの『浮動小数点変換』
- 108 ページの『ブール変換』

式の中に 2 つの異なる型のオペランドが含まれている場合、110 ページの『整数および浮動小数点数のプロモーション』で説明されているように、*通常の算術変換* の規則に従います。

関連資料

50 ページの『整数型』
52 ページの『浮動小数点型』
54 ページの『文字型』
66 ページの『列挙型』
69 ページの『構造体、共用体、および列挙型の互換性 (C のみ)』
121 ページの『単項式』
130 ページの『2 項式』

整数の変換

符号なし整数から符号なし整数へ、または符号付き整数から符号付き整数へ

2 つの型が同一の場合、変更はありません。2 つの型のサイズが異なる場合、その値が新規の型で表現できるならば、値は変更されません。値が新規の型で表現できない場合、切り捨てが行われるか、または符号シフトが行われます。

符号付き整数から符号なし整数へ

結果の値は、元の整数に適合する、最小の符号なし整数型です。その値が新規の型で表現できない場合、切り捨てが行われるか、または符号シフトが行われます。

符号なし整数から符号付き整数へ

符号付き型が元の値を入れられる十分な大きさである場合、変更はありません。値が新規の型で表現できる場合、値は変更されません。値が新規の型で表現できない場合、切り捨てが行われるか、または符号シフトが行われます。

符号付きおよび符号なし文字型から整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、`unsigned int` にプロモートされます。

ワイド文字型 wchar_t から整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、それを収容できる最小の型、すなわち、`unsigned int`、`long`、または `unsigned long` にプロモートされます。

符号付きおよび符号なし整数ビット・フィールドから整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、`unsigned int` にプロモートされます。

列挙型から整数へ

元の値が `int` で表現できる場合、`int` として表現されます。元の値が `int` で表現できない場合、それを収容できる最小の型、すなわち、`unsigned int`、`long`、または `unsigned long` にプロモートされます。ただし、列挙型は整数型に変換できますが、整数型は列挙型に変換できないことに注意してください。

関連資料

15 ページの『リテラル』

50 ページの『整数型』

ブール変換

ブールから整数へ

C ブール値が 0 の場合、結果は値 0 の `int` です。ブール値が 1 の場合、結果は値 1 の `int` です。

スカラーからブールへ

C スカラー値が 0 に等しい場合、ブール値は 0 です。そうでない場合、ブール値は 1 です。

関連資料

15 ページの『リテラル』
28 ページの『区切り子と演算子』
51 ページの『ブール型』

浮動小数点変換

実浮動小数点型 (2 進数から 2 進数、10 進数から 10 進数、10 進数から 2 進数) 間の変換の標準規則は、次のとおりです。

変換される値が新しい型で厳密に表示できる場合、その型は変更されません。変換される値が表示できる値の範囲内にあるが、厳密に表示できない場合、コンパイル時またはランタイム時の、現在有効な丸めの方式に従がって結果が丸められます。変換する値が表示可能な値の範囲外の場合、結果は丸めの方式に左右されます。

整数から浮動小数点 (2 進数または10 進数) へ
変換される値が新しい型で厳密に表示できる場合、その型は変更されません。変換される値が表示できる値の範囲内にあるが、厳密に表示できない場合、結果は正確に丸められます。変換する値が表示可能な値の範囲外の場合、結果は静止 NaN です。

浮動小数点 (2 進数または 10 進数) から整数へ
小数部分は破棄されます (つまり、値はゼロに切り捨てられます)。整数部分の値が整数型で表示できない場合、結果は次のいずれかです。

- 整数型が符号なしで、かつ浮動小数点数が正の場合、結果は表示可能な最大数であり、それ以外の場合は 0 です。
- 整数型がサイン付きの場合、浮動小数点数の符号に応じて、結果は負の最小値または正の最大値です。

複素数変換

複素数から複素数へ
2 つの型が同一の場合、変更はありません。2 つの型のサイズが異なり、かつその値が新規の型で表現できる場合は、値は変更されません。値が新規の型で表現できない場合、実数部と虚数部は上記の標準変換規則に従って変換されます。

複素数から実数 (2 進数) へ
複素数値の虚数部は廃棄されます。必要に応じて、実数部の値は上記の標準変換規則に従って変換されます。

複素数から実数 (10 進数) へ
複素数値の虚数部は廃棄されます。実数部の値は、上記の標準変換規則に従って、2 進から 10 進浮動小数点数へ変換されます。

実数 (2 進数) から複素数へ
ソース値が複素数値の実数部として使用され、必要に応じて上記の標準変換規則に従って、変換されます。虚数部の値はゼロです。

実数 (10 進数) から複素数へ
上記の標準変換規則に従って、ソース値は 10 進から 2 進浮動小数点数へ変換され、複素数値の実数部として使用されます。虚数部の値はゼロです。

関連資料

15 ページの『リテラル』
28 ページの『区切り子と演算子』
52 ページの『浮動小数点型』

整数および浮動小数点数のプロモーション

所定の型の式の中でオペランドとして異なる算術型が使用されている場合、*通常の算術変換* と呼ばれる標準の変換が適用されます。このような変換は、算術型のランクに従って適用されます。すなわち、下位ランクの型を持つオペランドが上位ランクのオペランドの型に変換されます。これを、整数または浮動小数点数の*プロモーション* といいます。

例えば、2 つの異なる整数型の値を加算する場合、最初に両方の値が同じ型に変換されます。すなわち、`short int` 値と `int` 値を加算する場合、`short int` 値が `int` 型に変換されます。 115 ページの『第 7 章 式と演算子』 に、通常の算術変換に関係のある演算子と式のリストを掲載しています。

算術型のランキングは、最高から最低への順にリストすると、次のようになります。

表 20. 浮動小数点型に対する変換ランキング

オペランドの型
`long double` または `long double _Complex`
`double` または `double _Complex`
`float` または `float _Complex`

表 21. 10 進浮動小数点型の変換ランキング

オペランドの型
IBM _Decimal128
IBM _Decimal64
IBM _Decimal32

表 22. 整数型に対する変換ランキング

オペランドの型
`unsigned long long` または `unsigned long long int`
`long long` または `long long int`[1]
`unsigned long int`[1]
`long int`[2]
`unsigned int`[2]
`int` および列挙型
`short int`
`char`、`signed char` および `unsigned char`
ブール

注:

1. 一方のオペランドは unsigned long int 型で、他方のオペランドは long long int (または long long) 型であるが、unsigned long int の値が、long long int (または long long) では表現できない場合、両方のオペランドが unsigned long long int (または unsigned long long) に変換されます。
2. 一方のオペランドは unsigned int 型で、他方のオペランドは long int 型であるが、unsigned int の値が long int で表現できない場合、両方のオペランドが unsigned long int に変換されます。

関連資料

50 ページの『整数型』
51 ページの『ブール型』
52 ページの『浮動小数点型』
54 ページの『文字型』
66 ページの『列挙型』
130 ページの『2 項式』
113 ページの『関数引数の変換』

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。次の表に、これの例外をリストします。

変換前の状況	結果の振る舞い
左辺値は関数型です。	
左辺値は配列です。	
左辺値の型は不完全型です。	コンパイル時エラー
左辺値は未初期化オブジェクトを参照しています。	未定義の振る舞い
左辺値は、右辺値の型でもなく、右辺値の型から派生する型でもないオブジェクトを参照しています。	未定義の振る舞い

関連資料

115 ページの『左辺値と右辺値』

ポインター型変換

ポインター型変換は、ポインターが使用されるときに実行されます。この型変換には、ポインターの割り当て、初期化、および比較が含まれます。

C ポインターを含む変換は、明示的型キャストを使用しなければなりません。この規則の例外は、C ポインターに関する許容代入変換です。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 23. 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 (void*) へのポインター
(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。オブジェクト・ポインターは、他方のポインターが型 void* の場合、不完全型でもかまいません。 C

ゼロ定数からヌル・ポインターへ

評価がゼロになる定数式は、*ヌル・ポインター定数* です。この式をポインターに変換することができます。このポインターは、ヌル・ポインター (ゼロ値を持つポインター) となり、どのオブジェクトをも指さないようになります。

配列からポインターへ

型「*N* の配列」(*N* は、配列の単一エレメントの型) の左辺値または右辺値は *N** へ。結果は、配列の初期エレメントを指すポインターです。式が & (アドレス) 演算子または sizeof 演算子のオペランドとして使用される場合は、この変換は実行できません。

関数からポインターへ

関数である左辺値は、同じ型の関数を指すポインターである右辺値に変換できます。ただし、その式が、& (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。

関連資料

84 ページの『ポインター』

118 ページの『整数定数式』

87 ページの『配列』

197 ページの『関数を指すポインター』

134 ページの『加法演算子 +』

134 ページの『減法演算子 -』

void* への変換

C ポインターは、必ずしも、型 int と同じサイズではありません。関数に渡されるポインター引数は、その関数によって期待される正しい型が必ず渡されるように、明示的にキャストでなければなりません。 C での総称オブジェクト・ポインターは void* ですが、総称関数ポインターはありません。

オブジェクトを指すポインター (型で修飾されることがあります) は、同じ const または volatile 修飾を保持しながら、void* に変換することができます。

C 以下の表に、左方オペランドとして void* を持つ許容代入変換を示します。

表 24. C での void に関する有効な代入変換*

左方オペランドの型	許可される右方オペランドの型
(void*)	• 定数 0 • オブジェクトを指すポインター。このオブジェクトは、不完全型でもかまいません。 • (void*)

関連資料

54 ページの『void 型』

関数引数の変換

関数が呼び出されたとき、関数宣言が存在していて、宣言された引数の型が含まれている場合、コンパイラーは型検査を行います。コンパイラーは、呼び出し側の関数によって提供されるデータ型を、呼び出し先の関数が予想するデータ型と比較し、必要な型変換を行います。例えば、関数 funct が呼び出されると、引数 f は double に変換され、引数 c は int に変換されます。

```
char * funct (double d, int i);
     /* ... */
int main(void)
{
   float f;
   char c;
   funct(f, c) /* f is converted to a double, c is converted to an int */
   return 0;
}
```

関数が呼び出されたときに可視の関数宣言がない場合、またはプロトタイプ引数リストの可変部分に式が引数として現れると、コンパイラーは、関数に引数を渡す前にデフォルトの引数プロモーションを行うか、または式の値を変換します。自動変換では、次のことが行われます。

- 整数値および浮動小数点値はプロモートされます。
- 配列または関数はポインターに変換されます。

関連資料

110 ページの『整数および浮動小数点数のプロモーション』

79 ページの『transparent_union 型属性 (C のみ)』

120 ページの『関数呼び出し式』

195 ページの『関数呼び出し』

第 7 章 式と演算子

式は、計算を指定する演算子、オペランド、および区切り子のシーケンスです。式に含まれている演算子と、それらの演算子が使われるコンテキストに基づいて式の評価を行います。式は結果として値になりますが、*副次作用*が発生します。副次作用とは、実行環境の状態の変化のことです。

149 ページの『演算子優先順位と結合順序』に、先にリストされた各セクションで説明されるすべての演算子の優先順位をリストした表を掲載してあります。

関連資料

28 ページの『区切り子と演算子』

85 ページの『ポインター演算』

155 ページの『式ステートメント』

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、変更できない左辺値です。*変更可能な左辺値* という用語は、その左辺値は、指定されたオブジェクトを検査するだけでなく、変更できることを強調するために使用されます。次のオブジェクトの型は左辺値ですが、変更可能な左辺値ではありません。

- 配列型
- 不完全型
- `const` 修飾された型
- メンバーの 1 つが `const` 型として修飾されている構造体または共用体の型

これらの左辺値は変更可能ではないので、代入ステートメントの左側に使用することはできません。

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も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 型をもっています。

C では、*関数指定子* が関数型を持つ式として定義されます。関数指定子は、オブジェクト型または左辺値とは区別されます。関数指定子は関数の名前、または関数ポインターを間接参照した結果を使用できます。 C 言語は、関数ポインターの処理とオブジェクト・ポインターの処理を変えています。

演算子の中には、自分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、その使用法に対する追加制限を掲載していま

す。

演算子	要件
& (単項)	オペランドは、左辺値でなければなりません。
++ --	オペランドは、左辺値でなければなりません。これは、接頭部と接尾部の両方の形式に適用されます。
= += -= *= %= <<= >>= &= ^= |=	左方オペランドは、左辺値でなければなりません。

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なければなりません。

アドレス演算子 (&) には、オペランドとして左辺値が必要です。一方、増分演算子 (++) と減分演算子 (--) には、修正可能な左辺値がオペランドとして必要です。以下の例に、式と、その対応する左辺値を示します。

式	左辺値
x = 42	x
*ptr = newvalue	*ptr
a++	a

IBM GNU C 言語拡張機能を使用可能にしてコンパイルした場合は、複合式、条件式、およびキャストは、それぞれのオペランドが左辺値であれば、左辺値として使用できます。

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

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

シーケンス内の最後の式が左辺値であれば、複合式にアドレス演算子を適用することができます。次の式は、同等です。

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

条件式は、その型が void でなく、真と偽に関する、その分岐が両方とも有効な左辺値である場合、有効な左辺値になりえます。キャストは、オペランドが左辺値である場合、有効な左辺値です。基本的な制限は、左辺値キャストのアドレスを使用できないことです。IBM

関連資料

111 ページの『左辺値から右辺値への変換』

87 ページの『配列』

121 ページの『単項式』

130 ページの『2 項式』

131 ページの『代入演算子』

141 ページの『配列添え字演算子 [ ]』
146 ページの『Cast 演算子 ()』

1 次式

1 次式 は、次のカテゴリーに分類できます。

- 名前 (ID)
- リテラル (定数)
- 整数定数式
- 括弧で囲んだ式 ( )

名前

名前の値は、その型によって異なります。型は、その名前がどのように宣言されるかによって決まります。次の表に、名前が左辺値式であるかどうかを示します。

表 25. 1 次式: 名前

宣言された名前は	評価された結果は	左辺値か
算術型、ポインター型、列挙型、構造体型、または共用体型の変数	その型のオブジェクト	はい
列挙型定数	関連付けられた整数値	いいえ
配列	その配列。変換が必要なコンテキストでは、名前が sizeof 演算子の引数として使用される場合を除いて、配列内の最初のオブジェクトへのポインター。	いいえ
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。	いいえ

式としては、名前は、ラベル、typedef 名、構造体メンバー、共用体メンバー、構造体タグ、共用体タグ、または列挙型タグを参照できません。このような目的に使用される名前は、式で使用される名前のネーム・スペースとは別のネーム・スペースに置かれています。ただし、このような名前のいくつかは、特殊な構成によって式の中で参照することができます。例えば、ドット演算子または矢印演算子は、構造体または共用体のメンバーを参照するために使用できます。typedef 名は、キャストの中で、または sizeof 演算子への引数として使用できます。

リテラル

リテラルは、数値定数またはストリング・リテラルです。リテラルが式として評価されると、その値は定数です。字句定数は左辺値にはなりません。ただし、ストリング・リテラルは左辺値です。

関連資料

15 ページの『リテラル』

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。ただし、オペランドは可変長配列であってはならない。

可変長配列型に適用される `sizeof` 演算子は、実行時に評価され、そのため、定数式ではありません。

以下のような状況においては、整数定数式を使用する必要があります。

- 添え字宣言子の中 (バインドされた配列の記述として)。
- `switch` ステートメントのキーワード `case` の後。
- 列挙型定数の中で、列挙型定数の数値として。
- ビット・フィールド幅の指定子の中。
- プリプロセッサーの `#if` ステートメントの中。(プリプロセッサーの `#if` ステートメントに列挙型定数、アドレス定数、および `sizeof` は指定できません。)

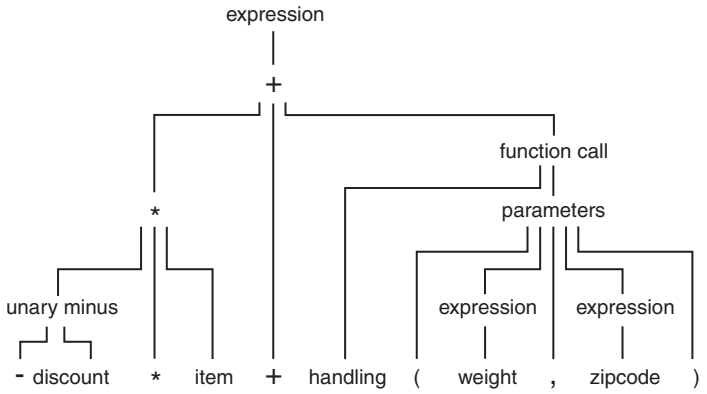
関連資料

111 ページの『ポインター型変換』

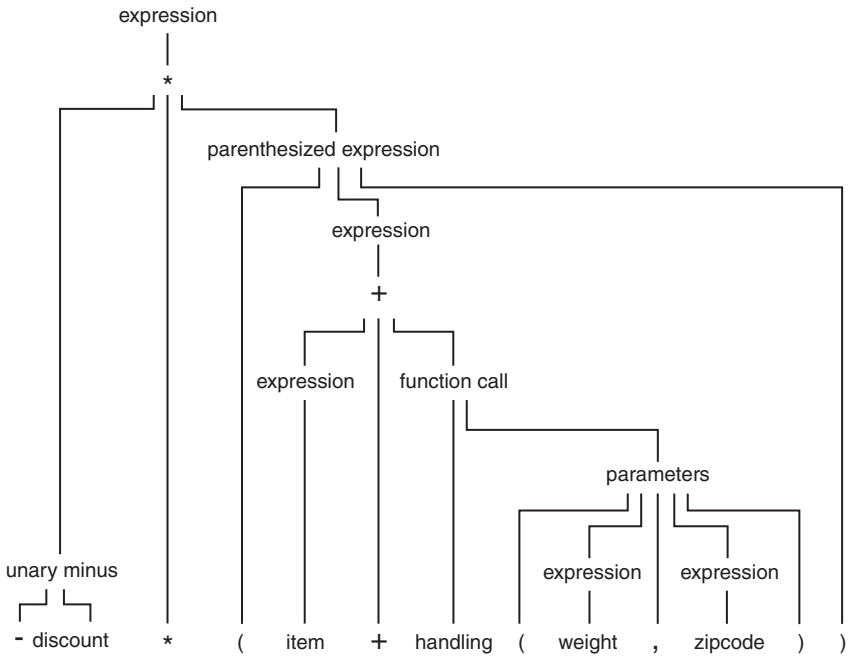
127 ページの『sizeof 演算子』

括弧で囲んだ式 ( )

小括弧を使用して、式の評価順序を明示的に強制します。次の式は、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);
```

関連資料

82 ページの『型名』

149 ページの『演算子優先順位と結合順序』

200 ページの『#define ディレクティブ』

関数呼び出し式

関数呼び出し は、関数名と、その後に続く関数呼び出し演算子 () とで構成される式です。関数がパラメーターを受け取るように定義されている場合、関数に送られる値は、関数呼び出し演算子の括弧内にリストされています。引数リストには、コンマで区切った式をいくつでも入れることができます。引数リストは、空にすることもできます。

関数呼び出し式の型は、その関数の戻りの型です。この型は、完全型または型 `void` のいずれかです。関数呼び出し式は常に右辺値です。

次は、関数呼び出し演算子の例です。

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

関数呼び出しの引数の評価順序は指定されていません。次の例では、

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

引数 `batch.process--` が最後に評価される可能性があり、その場合、最後の 2 つの引数は同じ値で渡されることになります。

関連資料

113 ページの『関数引数の変換』

195 ページの『関数呼び出し』

メンバー式

メンバー式は、構造体、または共用体のメンバーを示します。メンバー演算子は次のとおりです。

- ドット演算子 .
- 矢印演算子 ->

ドット演算子 .

. (ドット) 演算子は、構造体、または共用体メンバーにアクセスするために使用されます。メンバーは、後置式によって指定され、その後に . (ドット) 演算子、さらにその後に、場合によっては、修飾された ID が続きます。後置式は、`struct`、または `union` 型のオブジェクトでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。後置式と名前が左辺値の場合は、その式の値も左辺値です。後置式が型修飾されている場合、結果の式の中の指定されたメンバーには同じ型修飾子が適用されます。

関連資料

59 ページの『構造体および共用体』

構造体メンバーおよび共用体メンバーへのアクセス

矢印演算子 ->

-> (矢印) 演算子は、ポインターを使用する、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 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。後置式は、struct、または union 型のオブジェクトを指すポインターでなければなりません。名前は、そのオブジェクトのメンバーでなければなりません。

式の値は、選択されたメンバーの値です。名前が左辺値の場合は、式の値も左辺値です。式が修飾型へのポインターである場合、同じ型修飾子が結果の式の中の指定されたメンバーに適用されます。

関連資料

59 ページの『構造体および共用体』

84 ページの『ポインター』

構造体メンバーおよび共用体メンバーへのアクセス

単項式

単項式 には、1 つのオペランドと単項演算子が含まれています。

サポートされる単項演算子は次のとおりです。

- 122 ページの『増分演算子 ++』
- 122 ページの『減分演算子 --』
- 123 ページの『単項プラス演算子 +』
- 123 ページの『単項減法演算子 -』
- 123 ページの『論理否定演算子 !』
- 124 ページの『ビット単位否定演算子 ~』
- 124 ページの『アドレス演算子 &』
- 125 ページの『間接演算子 *』

- IBM alignof
- sizeof
- IBM typeof
- IBM __real__ および __imag__
- IBM vec_step

すべての単項演算子には、150 ページの表 28 に示されているように、同じ優先順位が付けられ、右から左の結合順序が適用されます。

演算子の説明で示されているように、ほとんどの単項式のオペランドで、通常の算術変換が行われます。

関連資料

15 ページの『リテラル』

28 ページの『区切り子と演算子』
85 ページの『ポインター演算』
115 ページの『左辺値と右辺値』
107 ページの『算術変換およびプロモーション』

増分演算子 ++

++ (増分) 演算子は、スカラー・オペランドの値に 1 を加えます。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ増分します。オペランドは、増分演算の結果を受け取ります。オペランドは、算術型またはポインター型の変更可能な左辺値でなければなりません。

++ は、オペランドの前にも後にも置くことができます。それがオペランドの前にくると、オペランドは増分されます。増分された値が、式の中で使用されます。オペランドの後に ++ を置くと、オペランドを増分する*前*に、そのオペランドの値が使用されます。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、整数拡張後にオペランドと同じ型になります。

オペランドには、通常の算術変換が実行されます。

IBM 増分演算子は、複素数型を扱うように拡張されています。この演算子の働きは、オペランドの実数部のみが増分され、虚数部は変更されないことを除いて、実数型に対する場合と同じです。

関連資料

85 ページの『ポインター演算』

減分演算子 --

-- (減分) 演算子は、スカラー・オペランドの値から 1 を減算します。または、オペランドがポインターの場合、オペランドを、それが指しているオブジェクトのサイズの分だけ、減じます。オペランドは、減分演算の結果を受け取ります。オペランドは、変更可能な左辺値でなければなりません。

減分演算子は、オペランドの前後に -- を入れることができます。この演算子がオペランドの前にあると、オペランドを減分し、減らした値が式で使われます。 -- がオペランドの後にある場合は、オペランドの現行値が式で使われ、その後でオペランドが減らされます。

次に例を示します。

```
play = --play1 + play2--;
```

これは、以下の式に似ています。play2 は play の前で変更されます。

```
int temp, temp1, temp2;

temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;
```

結果は、左辺値ではなく、整数拡張後にオペランドと同じ型になります。

オペランドには、通常の算術変換が実行されます。

IBM 減分演算子は、GNU C との互換性のために、複素数型を扱うように拡張されています。この演算子の働きは、オペランドの実数部のみが減分され、虚数部は変更されないことを除いて、実数型に対する場合と同じです。

関連資料

85 ページの『ポインター演算』

単項プラス演算子 +

+ (単項プラス) 演算子は、オペランドの値を保持します。オペランドには、任意の算術型またはポインター型を指定できます。結果は、左辺値ではありません。

結果は、整数拡張後にオペランドと同じ型になります。

注: 定数の前の正符号は、定数の一部ではありません。

単項減法演算子 -

- (単項減算) 演算子は、オペランドの値を否定します。オペランドには、任意の算術型を指定できます。結果は、左辺値ではありません。

例えば、quality に値 100 が指定された場合は、-quality は値 -100 になります。

結果は、整数拡張後にオペランドと同じ型になります。

注: 定数の前の負符号は、定数の一部ではありません。

論理否定演算子 !

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

オペランドの評価の結果が 0 になる場合は、式の値は 1 (true) になり、オペランドの評価の結果がゼロ以外の値になる場合は、式の値は 0 (false) になります。

次の 2 つの式は、同じです。

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

関連資料

51 ページの『ブール型』

ビット単位否定演算子 ~

~ (ビット単位否定) 演算子は、オペランドのビット単位の補数を生成します。結果の 2 進表示では、すべてのビットは、オペランドの 2 進表示の同じビットの値と反対の値を保持します。オペランドには、整数型が指定されている必要があります。結果には、オペランドと同じ型が指定され、左辺値ではありません。

x が、10 進数の値 5 を表すとします。x の 16 ビットの 2 進表示は次のとおりです。

```
0000000000000101
```

式 ~x の結果は、次のようになります (ここでは、16 ビットの 2 進数で表されます)。

```
1111111111111010
```

~ 文字は、3 文字表記文字の ??- によって表されることに注意してください。

~0 の 16 ビットの 2 進表示は、次のとおりです。

```
1111111111111111
```

IBM ビット単位否定演算子は、複素数型を扱うように拡張されています。複素数型では、この演算子は、虚数部の符号を反転することにより、オペランドの複素数共役を計算します。

関連資料

35 ページの『3 文字表記』

アドレス演算子 &

& (アドレス) 演算子は、そのオペランドを指すポインターを生成します。オペランドは、左辺値、関数指定機能、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;
```

IBM アドレス演算子は、ベクトルのサポートが使用可能になっている場合に、ベクトル型を処理するために拡張されました。ベクトル型にアドレス演算子を適用した結果は、互換性のあるベクトル型を指すポインターに保管することができます。ベクトル型のアドレスを使用して、ベクトル型を指すポインターを初期化することができます。ただし、初期化の両辺が互換性のある型であることが必要です。`void` を指すポインターも、ベクトル型のアドレスで初期化することができます。

IBM ラベルのアドレスは、GNU C アドレス演算子 `&&` を使用して取ることができます。したがって、そのラベルは値として使用できます。

関連資料

84 ページの『ポインター』

『間接演算子 *』

154 ページの『値としてのラベル (IBM 拡張)』

間接演算子 *

`*` (間接) 演算子は、ポインター型オペランドによって参照される値を決めます。オペランドは、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 。 配列と関数はポインターに変換されます。

オペランドの型は、結果の型を決定します。例えば、オペランドが `int` を指すポインターの場合は、結果は、`int` 型になります。

無効なアドレス (`NULL` など) を含むポインターに間接演算子を適用しないでください。結果は、予測できません。

`p_to_y` が `int` を指すポインターとして定義され、 `y` が `int` として定義されている場合、式は次のようになります。

```
p_to_y = &y;
*p_to_y = 3;
```

変数 `y` は値 `3` を受け取ります。

IBM 間接演算子 `*` は、ベクトルのサポートが使用可能になっている場合に、ベクトル型を指すポインターを処理するために拡張されました。ベクトル・ポインターは、16 バイトで位置合わせされたメモリー位置を指し示している必要があります。ただし、コンパイラーは、この制約を強制適用しません。ベクトル・ポインターから逆参照したときに、ベクトル型とその 16 バイト位置合わせが維持されます。プログラムが、16 バイトで位置合わせされたアドレスを含まないベクトル・ポインターから逆参照した場合は、どのような振る舞いが生じるかは未定義です。

関連資料

84 ページの『ポインター』

124 ページの『アドレス演算子 &』

87 ページの『配列』

__alignof__ 演算子 (IBM 拡張)

__alignof__ 演算子は、そのオペランドの位置合わせに使われるバイト数を戻す、C99 の言語拡張機能です。オペランドは式であってもよいし、括弧で囲んだ型 ID であってもかまいません。オペランドが左辺値を表す式である場合、__alignof__ によって戻される数値は、その左辺値が持つことが判明している位置合わせを表します。式の型はコンパイル時に判別されますが、式そのものは評価されません。オペランドが型である場合、その数値は、ターゲット・プラットフォーム上でその型に対して通常必要な位置合わせを表します。

__alignof__ 演算子を次の項目に適用することはできません。

- ビット・フィールドを表す左辺値
- 関数型
- 未定義の構造体またはクラス
- 不完全型 (void など)

__alignof__ 演算子の構文

type-id が参照、または参照された型である場合、結果は、参照された型の位置合わせです。 *type-id* が配列である場合、結果は、配列エレメント型の位置合わせです。*type-id* が基本型である場合、結果は、インプリメンテーションで定義されたとおりです。

例えば、AIX では、__alignof__(wchar_t) は、32 ビット・ターゲットに対して 2 を戻し、64 ビット・ターゲットに対して 4 を戻します。

ベクトルのサポートが使用可能になっている場合には、__alignof__ のオペランドをベクトル型にすることができます。次に例を示します。

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int *pv1 = &v1;
__alignof__(v1); // vector type alignment: 16.
__alignof__(&v1); // address of vector alignment: 4.
__alignof__(*pv1); // dereferenced pointer to vector alignment: 16.
__alignof__(pv1); // pointer to vector alignment: 4.
__alignof__(vector signed char); // vector type alignment: 16.
```

__attribute__((aligned)) を使用して、ベクトル型の変数の位置合わせ値を増加させた場合は、__alignof__ 演算子から戻される値は、__attribute__((aligned)) に指定されている位置合わせ計数です。

関連資料

73 ページの『__align 型修飾子 (IBM 拡張)』

78 ページの『aligned 型属性』

79 ページの『packed 型属性』

102 ページの『aligned 変数属性』

104 ページの『packed 変数属性』

sizeof 演算子

sizeof 演算子は、オペランドのサイズをバイト数で生成します。オペランドは、式であってもよいし、括弧で囲んだ型の名前であってもかまいません。

sizeof 演算子の構文

どちらの種類のオペランドであっても、結果は左辺値ではなく、定数整数値です。結果の型は、ヘッダー・ファイル stddef.h で定義された符号なし整数型 size_t です。

プリプロセッサー・ディレクティブの中以外では、整数定数が必要なときは、sizeof 式を使用できます。sizeof 演算子がよく使われる 1 つの例は、ストレージ割り振り時、入力関数、および出力関数で参照されるオブジェクトのサイズを決める場合です。

もう 1 つの sizeof の使用法は、プラットフォームをまたがってコードを移植する場合です。sizeof 演算子を使用して、データ型が表すサイズを判別することができます。次に例を示します。

```
sizeof(int);
```

型名に適用された sizeof 演算子は、その型のオブジェクトに使用されるメモリー量を、内部埋め込みまたは末尾埋め込みを含めて計算します。

IBM ベクトルのサポートが使用可能になっている場合には、sizeof 演算子のオペランドをベクトル型、またはベクトル型へのポインターの間接参照の結果とすることができます。その場合は、sizeof の戻り値は常に 16 です。

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1); // vector type: 16.
sizeof(&v1); // address of vector: 4.
sizeof(*pv1); // dereferenced pointer to vector: 16.
sizeof(pv1); // pointer to vector: 4.
sizeof(vector bool int); // vector type: 16.
```

複合型の場合、結果は次のとおりです。

オペランド	結果
配列	結果は、配列内のバイトの合計数になります。例えば、10 のエレメントがある配列では、サイズは、単一のエレメントのサイズの 10 倍になります。コンパイラーは、式を評価する前には、配列をポインターに変換しません。

sizeof 演算子は、次のものに適用することはできません。

- ビット・フィールド
- 関数型
- 未定義の構造体またはクラス
- 不完全型 (void など)

式に適用された sizeof 演算子は、その式の型の名前だけに適用された場合と同じ結果になります。コンパイル時、コンパイラーは式を分析してその型を判別します。式の型分析で行われる通常の型変換はいずれも、直接的には sizeof 演算子に起因しません。オペランドに型変換を行う演算子が含まれている場合は、コンパイラーは、型は判別するときに、これらの型変換を考慮します。例えば、次の例の 2 行目では、通常の算術変換が行われます。 short が 2 バイトのストレージを使用し、int が 4 バイトを使用するものとすると、次のようになります。

```
short x; ... sizeof (x)         /* the value of sizeof operator is 2 */
short x; ... sizeof (x + 1)     /* value is 4, result of addition is type int */
```

式 x + 1 の結果は int 型になり、 sizeof(int) と同等です。 x が char、short、または int 型、または任意の列挙型の場合にも、値は 4 です。

関連資料

118 ページの『整数定数式』

82 ページの『型名』

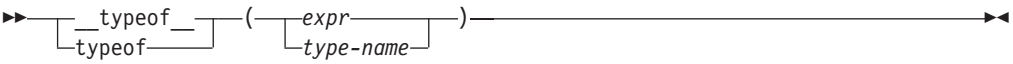
87 ページの『配列』

typeof 演算子 (IBM 拡張)

typeof 演算子は、その引数の型を戻します。引数は、式または型です。この言語フィーチャーは、式から型を引き出す手段を提供していることになります。式 e を例にとると、__typeof__(e) は、型名が必要などの場所ででも、例えば、宣言の中ででも、キャストの中ででも使用できます。キーワードの代替スペル、__typeof__ が推奨されています。

typeof 演算子は、ベクトルのサポートが使用可能になっている場合に、ベクトル型をオペランドとして受け入れるために拡張されました。

typeof 演算子の構文



typeof 構成そのものは式ではなく、型の名前です。 typeof 構成は、typedef を使用して定義された型名のように振る舞いますが、構文は sizeof の構文に似ています。

以下の例は、その基本的な構文を示しています。式 e の場合:

```
int e;
__typeof__(e + 1) j;   /* the same as declaring int j;     */
e = (__typeof__(e)) f; /* the same as casting e = (int) f; */
```

typeof 構成を使用することは、typedef 名を宣言することと同等です。例えば、次の場合、

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

次のように書くことができます。

```
__typeof__(i) a;         /* all three constructs have the same meaning */
__typeof__(int[2]) a;
__typeof__(T) a;
```

このコードの振る舞いは、int a[2]; を宣言した場合と同じです。

ビット・フィールドの場合、typeof は、そのビット・フィールドの基礎となる型を表します。例えば、int m:2;、typeof(m) は int です。ビット・フィールドのプロパティーは予約されないので、typeof(m) n; 内の n は int n と同じですが、int n:2 と同じではありません。

typeof 演算子は、sizeof の中およびそれ自身の中にネストすることができます。int へのポインターの配列としての arr の以下の 2 つの宣言は同等です。

```
int *arr[10];                       /* traditional C declaration          */
__typeof__(__typeof__ (int *)[10]) a;  /* equivalent declaration  */
```

typeof 演算子は、式 e がパラメーターであるマクロ定義の中で有用です。次に例を示します。

```
#define SWAP(a,b) { __typeof__(a) temp; temp = a; a = b; b = temp; }
```

注:

1. typeof および __typeof__ キーワードは、次のようにサポートされています。
   - __typeof__ キーワードは、**xlc** 呼び出しコマンドまたは **-qlanglvl=extc89**, **-qlanglvl=extc99**, または **-qlanglvl=extended** オプションでコンパイルしたとき認識されます。 typeof キーワードは、**-qkeyword=typeof** でコンパイルしたときのみ認識されます。

関連資料

82 ページの『型名』

70 ページの『typedef 定義』

55 ページの『ベクトル型 (IBM 拡張)』

__real__ および __imag__ 演算子 (IBM 拡張)

XL C では、C99 規格が拡張され、単項演算子 __real__ および __imag__ がサポートされるようになっています。これらの演算子を使用することにより、複素数型の実数部と虚数部を抽出することができます。これらの拡張機能は、GNU C を使用して開発されたアプリケーションの移植を容易にすることを目的としてインプリメントされています。

__real__ および __imag__ 演算子の構文

var_identifier は、以前に宣言された複素数変数の名前です。__real__ 演算子は複素数変数の実数部を戻し、__imag__ 演算子はその変数の虚数部を戻します。これらの演算子のオペランドが左辺値の場合、結果の式は、左辺値が許可されているどのコンテキストででも使用できます。 これらの演算子は、複素数変数の初期化のときと、複素数型を表すフォーマット指定子を持たない、printf および scanf などのライブラリー関数呼び出しへの引数として特に有用です。次に例を示します。

```
float _Complex myvar;
__imag__(myvar) = 2.0f;
__real__(myvar) = 3.0f;
```

上記の例は、複素数変数 myvar の虚数部を 2.0*i* に、実数部を 3.0 に初期化し、さらに次の関数は、

```
printf("myvar = %f + %f * i\n", __real__(myvar), __imag__(myvar));
```

以下を印刷します。

```
myvar = 2.000000 + 3.000000 * i
```

関連資料

52 ページの『浮動小数点型』

複素数リテラル (C のみ)

複素数浮動小数点型 (C のみ)

vec_step 演算子 (IBM 拡張)

vec_step 演算子はベクトル型引数を取り、ベクトル・エレメントを指すポインターが 16 バイト単位で移動して増分するべき量を表す整数値を戻します。この演算子に関する詳しい説明については、「*AltiVec Technology Programming Interface Manual*」(http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手可能) を参照してください。

2 項式

2 項式 には、1 つの演算子によって分離される 2 つのオペランドが含まれます。サポートされる 2 項演算子は次のとおりです。

- 131 ページの『代入演算子』
- 133 ページの『乗算演算子 *』
- 133 ページの『除算演算子 /』
- 133 ページの『剰余演算子 %』
- 134 ページの『加法演算子 +』
- 134 ページの『減法演算子 -』
- 135 ページの『ビット単位の左および右シフト演算子 << >>』
- 135 ページの『関係演算子 < > <= >=』
- 136 ページの『等価および非等価演算子 == !=』
- 138 ページの『ビット単位 AND 演算子 &』
- 138 ページの『ビット単位排他 OR 演算子 ^』
- 139 ページの『ビット単位包含 OR 演算子 |』
- 139 ページの『論理 AND 演算子 &&』
- 140 ページの『論理 OR 演算子 ||』
- 141 ページの『配列添え字演算子 [ ]』
- 142 ページの『コンマ演算子 ,』

2 項演算子は、すべてに左から右への結合順序が指定されますが、すべてに同じ優先順位が付けられるわけではありません。2 項演算子のランキングおよび優先順位規則を 150 ページの表 29 で要約しています。

ほとんどの 2 項演算子のオペランドが評価される順序は、指定されていません。正しい結果を得るためには、コンパイラーがオペランドを評価する順序に依存する 2 項式を作成しないようにします。

演算子の説明で示されているように、ほとんどの 2 項式のオペランドで、通常の算術変換が行われます。

関連資料

110 ページの『整数および浮動小数点数のプロモーション』

115 ページの『左辺値と右辺値』

107 ページの『算術変換およびプロモーション』

代入演算子

代入式 は、左方オペランドによって指定されたオブジェクトに値を保管します。代入演算子には、次の 2 つの型があります。

- 『単純代入演算子 =』
- 132 ページの『複合代入演算子』

すべての代入式の左方オペランドは、変更可能な左辺値でなければなりません。式の型は、左方オペランドの型です。式の値は、代入が完了した後の左方オペランドの値です。

代入式の結果は左辺値ではありません。

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適用されます。

単純代入演算子 =

単純代入演算子の形式は、次のとおりです。

lvalue = *expr*

演算子は、右方オペランド *expr* の値を、左方オペランド *lvalue* によって指定されたオブジェクトに保管します。

左方オペランドは、変更可能な左辺値でなければなりません。割り当て演算の型は、左方オペランドの型です。

左方オペランドがクラス型またはベクトル型ではない場合は、右方オペランドは暗黙的に左方オペランドの型に変換されます。この変換された型は、`const` または `volatile` によって修飾されることはありません。

左方オペランドがクラス型である場合、その型は完全でなければなりません。左方オペランドのコピー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。

IBM 代入演算子が拡張され、ベクトル型のオペランドが使用できるようになりました。代入式の両辺は同じベクトル型でなければなりません。

複合代入演算子

複合代入演算子は、2 項演算子と単純代入演算子で構成されます。複合代入演算子は、両方のオペランドに 2 項演算子の演算を実行し、その演算の結果を左方オペランドに保管します。左方オペランドは変更可能な左辺値でなければなりません。

次の表では、複合代入式のオペランドの型を示します。

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。

IBM GNU C 言語フィーチャーが使用可能になっているとき、複合式および条件式は、それらのオペランドが左辺値である場合に左辺値として認められます。複

合式 (a, b) の以下の複合代入は、式 b が左辺値であれば、すなわち、より一般的に言えば、シーケンス中の最後の式が左辺値であれば、GNU C では有効です。

```
(a,b) += 5  /* Under GNU C, this is equivalent to
a, (b += 5)    */
```

関連資料

95 ページの『構造体および共用体の初期化』

115 ページの『左辺値と右辺値』

84 ページの『ポインター』

71 ページの『型修飾子』

乗算演算子 *

* (乗算) 演算子は、そのオペランドの積を生成します。オペランドは、算術型または列挙型でなければなりません。結果は、左辺値ではありません。オペランドには、通常の算術変換が実行されます。

乗法演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数の乗法演算子を含む式の中でオペランドの再配置を行うことができます。例えば、次の式

```
sites * number * cost
```

は、次のいずれかの方法に解釈できます。

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

除算演算子 /

/ (除法) 演算子はそのオペランドの商を生成します。オペランドが両方とも整数である場合は、小数部分 (剰余) は廃棄されます。小数部分の廃棄は、しばしば、*ゼロに向かった切り捨て* と呼ばれます。オペランドは、算術型または列挙型でなければなりません。右方オペランドをゼロにすることはできません。右方オペランドの評価結果が 0 の場合、結果は未定義です。例えば、式 7 / 4 は、値 1 を生成します (1.75 または 2 ではありません)。結果は、左辺値ではありません。

オペランドには、通常の算術変換が実行されます。

剰余演算子 %

% (剰余) 演算子は、左方オペランドを右方オペランドで割り算した剰余を生成します。例えば、式 5 % 3 は 2 を生成します。結果は左辺値にはなりません。

オペランドは両方とも、整数型または列挙型でなければなりません。右方オペランドが 0 になる場合は、結果は未定義です。いずれかのオペランドに負の値がある場合は、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。

加法演算子 +

+ (加法) 演算子は、そのオペランドの合計を生成します。オペランドは両方とも算術型を保持するか、一方のオペランドがオブジェクト型を指すポインターで、もう一方のオペランドが整数型または列挙型を保持する必要があります。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

配列内のオブジェクトを指すポインターは、整数型を持つ値に加算できます。結果は、ポインター・オペランドと同じ型のポインターになります。結果は、オリジナルのエレメントから、添え字として扱われる整数値の量だけオフセットされた、配列の中の別のエレメントを参照します。結果のポインターが、配列の外側のストレージ (配列の外側の最初のロケーション以外) を指す場合、結果は未定義です。配列の終わりより 1 つ後のエレメントを指すポインターを使用して、そのアドレスにあるメモリーの内容にアクセスすることはできません。コンパイラーは、ポインターの境界検査は行いません。例えば、以下の例で、加算の後で、ptr は配列の 3 番目のエレメントを指します。

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

関連資料

85 ページの『ポインター演算』

111 ページの『ポインター型変換』

減法演算子 -

- (減法) 演算子は、そのオペランドの差を生成します。オペランドは両方とも算術型または列挙型を保持するか、左方オペランドがポインター型で、右方オペランドがポインターの型と同じ型か整数型または列挙型を保持する必要があります。整数値からポインターを減算することはできません。

オペランドが両方とも算術型のときは、オペランドに通常の算術変換を実行します。結果には、オペランドの型変換によって生成される型が保持されます。結果は、左辺値ではありません。

左方オペランドがポインターで、右方オペランドが整数型を保持するときは、コンパイラーは、右方の値を相対位置のアドレスに変換します。結果は、ポインター・オペランドと同じ型のポインターになります。

オペランドが両方とも同じ配列内のエレメントを指すポインターである場合は、結果は、2 つのアドレスを分離するオブジェクトの数です。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。

関連資料

85 ページの『ポインター演算』

111 ページの『ポインター型変換』

ビット単位の左および右シフト演算子 << >>

ビット単位シフト演算子は、2 進数オブジェクトのビット値を移動します。左方オペランドには、シフトされる値を指定します。右方オペランドには、値のビットがシフトされる桁数を指定します。結果は、左辺値ではありません。両方のオペランドに同じ優先順位が付けられ、左から右の結合順序が指定されます。

演算子	使用法
<<	ビットが左方にシフトされることを指示します。
>>	ビットが右方にシフトされることを指示します。

各オペランドは、整数型または列挙型でなければなりません。コンパイラーは、オペランドの整数拡張を実行し、その後、右方オペランドが `int` 型に変換されます。結果には、左方オペランドと同じ型が指定されます (算術変換の後)。

右方オペランドが負の値、またはシフトされる式のビット幅より大きいかまたは等しい値を保持しないようにする必要があります。このような値でビット単位シフトを行うと、結果は予測不能な値になります。

右方オペランドに `0` がある場合は、結果は左方オペランドの値になります (通常の算術変換が実行された後)。

`<<` 演算子は、空になったビットをゼロで埋めます。例えば、`left_op` が値 `4019` を持っている場合、`left_op` のビット・パターン (16 ビットの形式) は次のようになります。

```
0000111110110011
```

式 `left_op << 3` は、次のビット・パターンを生成します。

```
0111110110011000
```

式 `left_op >> 3` は、次のビット・パターンを生成します。

```
0000000111110110
```

関係演算子 < > <= >=

関係演算子は、2 つのオペランドを比較して、リレーションシップの妥当性を判別します。次の表では、4 つの関係演算子を説明します。

演算子	使用法
<	左方オペランドの値が、右方オペランドの値より小さいかどうかを示します。
>	左方オペランドの値が、右方オペランドの値より大きいかどうかを示します。
<=	左方オペランドの値が、右方オペランドの値より小さいまたは等しいかどうかを示します。

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。

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。

結果は、左辺値ではありません。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。ポインターが同じ配列のオブジェクトを参照しない場合は、結果は未定義になります。

ポインターは、0 に評価される定数式と比較することができます。また、ポインターを void* 型のポインターと比較することもできます。ポインターを void* 型のポインターに変換します。

2 つのポインターが同じオブジェクトを参照する場合は、この 2 つのポインターは等しいと見なされます。 2 つのポインターが同一オブジェクトの非静的メンバーを参照する場合、これらのポインターがアクセス指定子によって分離されていなければ、後で宣言されるオブジェクトを指すポインターの方がより大きいです。分離されていれば、比較は未定義です。 2 つのポインターが同じ共用体のデータ・メンバーを参照する場合は、この 2 つのポインターは同じアドレス値を保持します。

2 つのポインターが同じ配列のエレメント、または配列の最後のエレメントを超えて最初のエレメントを参照する場合、より大きい添え字値をもっているエレメントを指すポインターの方が、より大きいです。

関係演算子では、同じオブジェクトのメンバーだけを比較できます。

関係演算子には、左から右の結合順序が指定されます。例えば、次の式

```
a < b <= c
```

は、次のように解釈されます。

```
(a < b) <= c
```

a の値が b の値よりも小さい場合は、最初の関係演算は、C では値 1その後で、コンパイラーは、値 true (または 1) を c の値と比較します (必要であれば、整数拡張が実行されます)。

等価および非等価演算子 == !=

等価演算子は、関係演算子と同様に、リレーションシップの妥当性について 2 つのオペランドを比較します。ただし、等価演算子には、関係演算子よりも低い優先順

位が付けられます。次の表で、2 つの等価演算子を説明します。

演算子	使用法
==	左方オペランドの値が、右方オペランドの値と等価かどうかを示します。
!=	左方オペランドの値が、右方オペランドの値と等価でないかどうかを示します。

オペランドは両方とも算術型または列挙型を保持するか、同じ型を指すポインターである必要があります。あるいは、一方のオペランドがポインター型を保持し、もう一方のオペランドが void を指すポインターまたはヌル・ポインターである必要があります。

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。

オペランドが算術型のときは、オペランドに通常の算術変換を実行します。

オペランドがポインターの場合は、ポインターが参照するオブジェクトのロケーションによって結果が決まります。

一方のオペランドがポインターで、もう一方のオペランドが値 0 の整数の場合、== 式は、ポインターのオペランドが NULL に評価される場合にだけ真になります。!= 演算子は、ポインター・オペランドが NULL に評価されない 場合に、真になります。

等価演算子を使用して、型が同じでも、同じオブジェクトに属さないメンバーを指すポインターを比較することもできます。次の式には、等価演算子と関係演算子の例が含まれています。

```
time < max_time == status < complete
letter != EOF
```

注: 等価演算子 (==) を、代入 (=) 演算子と混同しないでください。

次に例を示します。

if (x == 3)

x が 3 の場合**真** (または 1) になります。このような等価テストでは、意図しない代入を防ぐために、演算子とオペランドの間にスペースを入れてコーディングする必要があります。

一方

if (x = 3)

(x = 3) がゼロ以外の値 (3) になるので、真になります。また、この式では、3 が x に代入されます。

関連資料

単純代入演算子 =

ビット単位 AND 演算子 &

& (ビット単位 AND) 演算子は、第 1 オペランドの各ビットと第 2 オペランドの対応するビットを比較します。ビットが両方とも 1 であれば、対応する結果のビットを 1 にセットします。 1 でなければ、対応する結果のビットを 0 にセットします。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。

ビット単位 AND 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位 AND 演算子を含む式の中でオペランドの再配置を行うことができます。

次の例では、16 ビットの 2 進数で表された、a と b の値と、a & b の結果を示します。

a のビット・パターン	0000000001011100
b のビット・パターン	0000000000101110
a & b のビット・パターン	0000000000001100

注: ビット単位 AND (&) を、論理 AND (&&) 演算子と混同しないでください。例えば、

1 & 4 は 0 になります。
一方
1 && 4 は true になります。

ビット単位排他 OR 演算子 ^

ビット単位排他 OR 演算子 (EBCDIC では ^ シンボルは ¬ シンボルで表します) 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 ??'.

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

関連資料

35 ページの『3 文字表記』

ビット単位包含 OR 演算子 |

| (ビット単位包含 OR) 演算子は、各オペランドの値 (2 進形式) を比較し、いずれかのオペランドのどのビットの値が 1 であるかを示すビット・パターンの値を生成します。両方のビットが 0であると、その結果のビットは 0 になり、それ以外の結果は 1 になります。

オペランドは両方とも、整数型または列挙型でなければなりません。各オペランドには、通常の算術変換が実行されます。結果には、変換されたオペランドと同じ型が保持されます。結果は、左辺値ではありません。

ビット単位包含 OR 演算子には、結合属性と可換属性の両方があるので、コンパイラーは、複数のビット単位包含 OR 演算子を含む式の中でオペランドの再配置を行うことができます。 | 文字は、3 文字表記文字の ??! によって表されることに注意してください。

次の例では、16 ビットの 2 進数で表された、a と b の値と、a | b の結果を示します。

a のビット・パターン	0000000001011100
b のビット・パターン	0000000000101110
a?|?b のビット・パターン	0000000001111110

注: ビット単位 OR (|) を、論理 OR (||) 演算子と混同しないでください。例えば、

1 | 4 は 5 になります。一方
1 || 4 は true になります。

関連資料

35 ページの『3 文字表記』

論理 AND 演算子 &&

&& (論理 AND) 演算子は、両方のオペランドが真であるかどうかを示します。

両方のオペランドにゼロ以外の値がある場合、結果の値は 1 になります。そうでない場合、結果の値は 0 になります。その結果の型は、int です。オペランドは両方とも、算術型またはポインター型でなければなりません。各オペランドには、通常の算術変換が実行されます。

& (ビット単位 AND) 演算子と異なり、&& 演算子では、オペランドは必ず左から右に評価されます。左方オペランドが 0 (または偽) になる場合、右方オペランドは評価されません。

次の例では、論理 AND 演算子を含む式が評価される方法を示します。

式	結果
1 && 0	false または 0
1 && 4	true または 1
0 && 0	false または 0

次の例では、論理 AND 演算子を使用して、ゼロによる割り算を行わないようにします。

```
(y != 0) && (x / y)
```

y != 0 が 0 (つまり false) と評価される場合、式 x / y は評価されません。

注: 論理 AND 演算子 (&&) を、ビット単位 AND 演算子 (&) と混同しないでください。次に例を示します。

1 && 4 は 1 (または true) になります。
一方
1 & 4 は 0 になります。

論理 OR 演算子 ||

|| (論理 OR) 演算子は、いずれかのオペランドが真であるかどうかを示します。

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。

| (ビット単位包含 OR) 演算子と異なり、|| 演算子では、オペランドは必ず左から右に評価されます。左方オペランドがゼロ以外の値 (または真) である場合は、右方オペランドは評価されません。

次の例では、論理 OR 演算子を含む式が評価される方法を示します。

式	結果
1 || 0	true または 1
1 || 4	true または 1
0 || 0	false または 0

次の例では、論理 OR 演算子を使用して、y を条件付きで増分します。

```
++x || ++y;
```

式 ++x が、ゼロ以外 (または真) の値になる場合、式 ++y は、評価されません。

注: 論理 OR 演算子 (||) を、ビット単位 OR 演算子 (|) と混同しないでください。次に例を示します。

1 || 4 は 1 (または真) になります。一方
1 | 4 は 5 になります。

配列添え字演算子 [ ]

後置式とその後に続く [ ] (大括弧) 内の式が、配列のエレメントを指定します。大括弧内の式は、*添え字* と呼ばれます。配列の最初のエレメントの添え字はゼロです。

定義により、式 a[b] は、式 *((a) + (b)) と等価です (また、加算は結合であるので、b[a] とも等価です)。式 a と b の間で、一方は、型 T に対するポインターでなければなりません。そして他方は、整数型または列挙型をもっていなければなりません。配列添え字の結果は、左辺値になります。次の例は、このことを示しています。

```
#include <stdio.h>

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

上記の例の出力は、以下のとおりです。

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

各配列の最初のエレメントに、添え字 0 が付けられます。式 contract[35] は、配列 contract の 36 番目のエレメントを参照します。

多次元配列では、最も頻繁に右端添え字を増分することによって (保管場所の昇順で)、各エレメントを参照できます。

例えば、次のステートメントは、配列 code[4][3][6] の各エレメントに 100 を与えます。

```
for (first = 0; first < 4; ++first)
   {
   for (second = 0; second < 3; ++second)
      {
      for (third = 0; third < 6; ++third)
         {
         code[first][second][third] =
         100;
         }
      }
   }
```

C C99 aは、左辺値でない配列に配列添え字を許可します。ただし、左辺値でないもののアドレスは、配列添え字として使用することは、まだ許可されていません。次の例は、C99 で有効です。

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

関連資料

82 ページの『宣言子の例』

87 ページの『配列』

84 ページの『ポインター』

50 ページの『整数型』

115 ページの『左辺値と右辺値』

85 ページの『ポインター演算』

186 ページの『パラメーター宣言』

コンマ演算子 ,

コンマ式 には、任意の型の 2 つのオペランドがコンマで区切られて含まれており、左から右の結合順序が適用されます。左方オペランドは評価されますが、副次作用が生じる可能性があり、値がある場合、その値は廃棄されます。次に、右方オペランドが評価されます。コンマ式の結果の型および値は、通常の単項変換後の右方オペランドの型および値です。

C コンマ式の結果は左辺値ではありません。

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);
```

最初のオペランドの評価後に評価順序点が生じます。 delta の値は廃棄されます。同様に、以下の例では、次の式

```
intensity++, shade * increment, rotate(direction);
```

の値は、次の式の値になります。

```
rotate(direction)
```

コンマ文字が使われているコンテキストによっては、あいまいさを避けるために括弧が必要な場合があります。例えば、次の関数

```
f(a, (t = 3, t + 2), c);
```

の引数は値 a、値 5、および値 c の 3 個だけです。括弧が必要な他のコンテキストは、構造体宣言子リストおよび共用体宣言子リスト内のフィールド長の式の中、列挙型宣言子リスト内の列挙型の値の式の中、および宣言と初期化指定子内の初期化の式の中です。

上の例では、関数呼び出しの中の引数の式を区切るためにコンマが使用されています。このコンテキストでは、コンマを使用しても、関数の引数の評価順序 (左から右) は保証されません。

コンマ演算子の基本的な使用目的は、次のような状況で、副次作用をもたらすことです。

- 関数の呼び出し
- 反復ループへの入力または繰り返し
- 条件のテスト
- 副次作用は必要であるが、式の結果は今すぐに必要ではないその他の状況

次の表では、いくつかのコンマ演算子の使用例を示します。

ステートメント	効果
`for (i=0; i<2; ++i, f() );`	`for` ステートメントでは、`i` が増分され、反復のたびに `f()` が呼び出されます。
`if ( f(), ++i, i>1 ) { /* ... */ }`	`if` ステートメントでは、関数 `f()` が呼び出され、変数 `i` が増分され、変数 `i` が値に対してテストされます。このコンマ式内の最初の 2 つの式は、式 `i>1` の前に評価されます。最初の 2 つの式の結果に関係なく、3 番目の式が評価され、その結果が `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。

関連資料

54 ページの『void 型』

条件式

条件式 は、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 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))
```

関連資料

54 ページの『void 型』

C の条件式での型 (C のみ)

C では、条件式は左辺値ではなく、その結果も左辺値ではありません。

表 26. C の条件式のオペランドと結果の型

一方のオペランドの型	もう一方のオペランドの型	結果の型
算術	算術	通常の算術変換後の算術型
構造体または共用体型	互換構造体または共用体型	両方のオペランドにすべての修飾子が付く構造体または共用体型
void	void	void
互換型を指すポインター	互換型を指すポインター	型に指定されたすべての修飾子が付く型を指すポインター
型を指すポインター	NULL ポインター (定数 0)	型を指すポインター
オブジェクトまたは不完全型を指すポインター	void を指すポインター	型に指定されたすべての修飾子が付く void を指すポインター

IBM GNU C では、条件式は、その型が void でなく、その分岐の両方が有効な左辺値であれば、有効な左辺値です。次の条件式 (a ? b : c) は、GNU C では有効です。

```
(a ? b : c) = 5
/*  Under GNU C, equivalent to (a ? b = 5 : (c = 5))  */
```

この拡張機能は、拡張言語レベルの 1 つでコンパイルされるとき、使用可能です。

条件式の例

次の式では、値が大きい方の変数が y か z かを判別し、大きい方の値を変数 x に代入します。

```
x = (y > z) ? y : z;
```

次に等価なステートメントを示します。

```
if (y > z)
   x = y;
else
   x = z;
```

次の式では、関数 printf を呼び出し、c が数字に評価される場合に、この関数が、変数 c の値を受け取ります。そうでない場合は、printf は文字定数 'x' を受け取ります。

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

条件式の最後のオペランドに代入演算子が含まれる場合は、小括弧を使用して、式が正しい評価を行うようにします。例えば、次の式では、= 演算子には ?: 演算子より高い優先順位が付けられます。

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

このコンパイラーは、次のように括弧で囲まれているように解釈されるので、エラーになります。

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

つまり、k が代入式 k = j 全体としてではなく、第 3 オペランドとして扱われます。

j の値を k に代入するのは、i == 7 が偽のときで、最後のオペランドを小括弧で囲みます。

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

キャスト式

キャスト演算子は*明示的型変換* に使用されます。この演算子は式の値を、指定された型に変換します。

Cast 演算子 ()

Cast 式の構文

この演算の結果は左辺値ではありません。

以下は、サイズ 10 の整数配列を動的に作成するためのキャスト演算子の使用法を示したものです。

```
#include <stdlib.h>

int main(void) {
   int* myArray = (int*) malloc(10 * sizeof(int));
   free(myArray);
   return 0;
}
```

malloc ライブラリー関数は、その引数のサイズのオブジェクトを保持するメモリーを指す void ポインターを戻します。ステートメント int* myArray = (int*) malloc(10 * sizeof(int)) 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へ変換します。
- 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。

共用体型へのキャスト (C のみ) (IBM 拡張)

共用体型へのキャストにより、共用体のメンバーを、そのメンバーが属している共用体と同じ型にキャストすることができます。この種のキャストでは、他のキャストと異なり、左辺値は生じません。このフィーチャーは、GNU C を使用して開発されたプログラムの移植を容易にするためにインプリメントされたものであり、C99 の拡張機能としてサポートされています。

共用体型にキャストできるのは、その共用型のメンバーとして明示的に存在している型のみです。キャストには、共用体型のタグか、または typedef 式で宣言されている共用体型名を使用できます。指定する型は完全共用体型でなければなりません。無名共用体型は、タグまたは型名を持っている場合、共用体型へのキャストで使用できます。ビット・フィールドを共用体型にキャストできるのは、共用体に同じ型のビット・フィールド・メンバーが含まれている場合であり、その長さは同じでなくても構いません。次に共用体へのキャストの簡単な例を示します。

```
#include <stdio.h>

union foo {
   char t;
   short u;
   int v;
   long w;
   long long x;
   float y;
   double z;
```

```
};

int main() {
  union foo u;
  char a = 1;
  u = (union foo)a;
  printf("u = %i\n", u.t);
}
```

この例の出力は次のとおりです。

```
u = 1
```

ネストされた共用体へのキャストも可能です。以下の例では、double 型 dd をネストされた共用体 u2_t にキャストすることができます。

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

この例の出力は次のとおりです。

```
U.u2 is 1.234
```

共用体キャストは、関数引数として、さらに静的または非静的なデータ・オブジェクト の初期化のための定数式の一部として有効であり、かつ複合リテラル・ステートメントの中でも有効です。次の例は、静的データ・オブジェクトの初期化のための式の一部として使用される共用体へのキャストです。

```
struct S
{
   int a;
};

union U {
   struct S *s;
};

struct T {
   union U u;
} t[] = {
            {(union U)&s}
  };
```

関連資料

59 ページの『構造体および共用体』

79 ページの『transparent_union 型属性 (C のみ)』

82 ページの『型名』

115 ページの『左辺値と右辺値』

複合リテラル式

複合リテラル とは、値が初期化指定子によって与えられる名前なしオブジェクトを提供する後置式です。 C99 言語フィーチャーでは、一時変数を必要とせずに、パラメーターを関数に受け渡すことができます。これは、集合体型 (配列、構造体、および共用体) のインスタンスが 1 つだけ必要な場合に、その型の定数を指定するのに便利です。

複合リテラルの構文は cast 式の構文に似ています。ただし、複合リテラルは左辺値ですが、cast 式の結果はそうではありません。さらに、キャストはスカラー型または `void` にしか変換できませんが、複合リテラルは指定された型のオブジェクトになります。

複合リテラルの構文

type_name は、ベクトル型およびユーザー定義の型など任意のデータ型にすることができます。サイズが不明な配列にすることができますが、可変長配列にすることはできません。型が不明サイズの配列の場合、サイズは、初期化指定子リストによって決まります。

次の例では、2 つの整数メンバーを含む `point` 型の定数構造体変数を、関数 `drawline` に受け渡します。

```
drawline((struct point){6,7});
```

複合リテラルが関数の本体外で発生する場合、初期化指定子リストは定数式で構成される必要があり、名前なしのオブジェクトには静的ストレージ期間が指定されます。複合リテラルが関数の本体内で発生する場合、初期化指定子リストは定数式で構成される必要はなく、名前なしのオブジェクトには自動ストレージ期間が指定されます。

IBM GNU C との互換性を確保するために、初期化指定子リストのすべての初期化指定子が定数式であれば、同じ型の複合リテラルを使用して静的変数を初期化することができます。

関連資料

ストリング・リテラル

ラベル値演算子 (IBM 拡張)

ラベル値演算子 `&&` は、オペランドのアドレスを戻します。オペランドは、現行関数またはそれを包含する関数で定義されているラベルでなければなりません。値は `void*` 型の定数であり、computed goto (計算型 goto) ステートメントの中でのみ使用すべきものです。この言語フィーチャーは C の拡張機能の 1 つであり、GNU C を使用して開発されたプログラムの移植を容易にすることを目的としてインプリメントされています。

関連資料

154 ページの『値としてのラベル (IBM 拡張)』

計算型 goto ステートメント

170 ページの『goto ステートメント』

演算子優先順位と結合順序

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

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

```
b = 9;
c = 5;
a = b = c;
```

副次式の評価順序が指定されていないので、小括弧を使用して、演算子付きのオペランドのグループ化を明示的に強制することができます。

次の式では、

```
a + b * c / d
```

優先順位により、+ 演算の前に、* および / 演算が実行されます。結合順序により、b は d によって除算される前に、c と乗算されます。

次の表に、 言語の演算子を優先順位の順序でリストし、各演算子の結合順序の方向を示します。同位にある演算子には、同じ優先順位が付けられています。

表 27. 後置演算子の優先順位と結合順序

ランク	右結合か	演算子関数	使用法
1		メンバー選択	*object* . *member*
1		メンバー選択	*pointer* -> *member*
1		添え字	*pointer* [*expr*]
1		関数呼び出し	*expr* (*expr_list*)
1		値生成	*type* (*expr_list*)
1		後置増分	*lvalue* ++
1		後置減分	*lvalue* --

表 28. 単項演算子の優先順位と結合順序

ランク	右結合か	演算子関数	使用法
2	はい	オブジェクトのサイズ (バイト)	sizeof *expr*
2	はい	型のサイズ (バイト)	sizeof (*type*)
2	はい	前置増分	++ *lvalue*
2	はい	前置減分	-- *lvalue*
2	はい	ビット単位否定	~ *expr*
2	はい	not	! *expr*
2	はい	単項減算	- *expr*
2	はい	単項プラス	+ *expr*
2	はい	のアドレス	& *lvalue*
2	はい	間接指示または間接参照	* *expr*
2	はい	型変換 (キャスト)	(*type*) *expr*

表 29. 2 項演算子の優先順位と結合順序

ランク	右結合か	演算子関数	使用法
3		乗算	*expr* * *expr*
3		除法	*expr* / *expr*
3		モジュロ (剰余)	*expr* % *expr*
4		2 項加算	*expr* + *expr*
4		2 項減算	*expr* - *expr*
5		左へのビット単位シフト	*expr* << *expr*
5		右へのビット単位シフト	*expr* >> *expr*
6		より小さい	*expr* < *expr*
6		より小さいまたは等しい	*expr* <= *expr*
6		より大きい	*expr* > *expr*
6		より大きいまたは等しい	*expr* >= *expr*
7		等しい	*expr* == *expr*
7		等しくない	*expr* != *expr*
8		ビット単位 AND	*expr* & *expr*
9		ビット単位排他 OR	*expr* ^ *expr*
10		ビット単位包含 OR	*expr* | *expr*
11		論理 AND	*expr* && *expr*
12		論理包含 OR	*expr* || *expr*
13		条件式	*expr* ? *expr* : *expr*
14	はい	単純代入	*lvalue=expr*
14	はい	乗算および代入	*lvalue* *= *expr*
14	はい	除算および代入	*lvalue* /= *expr*
14	はい	モジュロおよび代入	*lvalue* %= *expr*
14	はい	加算および代入	*lvalue* += *expr*
14	はい	減算および代入	*lvalue-=expr*

表 29. 2 項演算子の優先順位と結合順序 (続き)

ランク	右結合か	演算子関数	使用法
14	はい	左へのシフトおよび代入	*lvalue* <<=*expr*
14	はい	右へのシフトおよび代入	*lvalue* >>=*expr*
14	はい	ビット単位 AND および代入	*lvalue* &=*expr*
14	はい	ビット単位排他 OR および代入	*lvalue* ^= *expr*
14	はい	ビット単位包含 OR および代入	*lvalue*|=*expr*
15		コンマ (順序付け)	*expr* , *expr*

関連資料

82 ページの『型名』

118 ページの『括弧で囲んだ式 ( )』

200 ページの『#define ディレクティブ』

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();
```

関数呼び出しの引数または 2 項演算子のオペランドの評価順序は、指定されていません。したがって、次の式はあいまいです。

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

最初のステートメントの前に、y に値 1 が指定されている場合は、値 1 または値 2 が func1() に渡されるかどうかは不明です。2 番目のステートメントでは、式が評価される前に i の値が 1 である場合は、x[1] または x[2] が 2 番目の引数として func2() に渡されるかどうかは不明です。

関連資料

82 ページの『型名』

第 8 章 ステートメント

ステートメントは最小の独立した計算単位であって、実行するアクションを指定します。ほとんどの場合、ステートメントは順々に実行されます。以下に、C で使用可能なステートメントの要約を示します。

- ラベル付きステートメント
- 式ステートメント
- ブロック・ステートメント
- 選択ステートメント
- 反復ステートメント
- 分岐ステートメント
- 宣言ステートメント
- ヌル・ステートメント
- IBM インライン・アセンブリー・ステートメント (C のみ) (IBM 拡張)

関連資料

39 ページの『第 4 章 データ・オブジェクトとデータ宣言』

178 ページの『関数宣言』

ラベル付きステートメント

ラベルには identifier、case、および default の 3 つの種類があります。

ラベル付きステートメントの構文

ラベルは、*identifier* およびコロン (**:**) 文字から構成されます。

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。

case および default ラベル・ステートメントは、`switch` ステートメントでのみ使用されます。これらのラベルは、最も近くの、括弧で囲まれた `switch` ステートメント内でのみアクセス可能です。

case ステートメントの構文

default ステートメントの構文

ラベルの例を次に示します。

```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```

関連資料

5 ページの『関数スコープ』

170 ページの『goto ステートメント』

159 ページの『switch ステートメント』

ローカルに宣言されたラベル (IBM 拡張)

ローカルに宣言されたラベル、すなわち、*ローカル・ラベル* は、ステートメント式の先頭で宣言される identifier ラベルであって、そのためのスコープは、そのラベルが宣言されて定義されたステートメント式です。この言語フィーチャーは C の拡張機能であって、GNU C で開発されたプログラムの処理を容易にするためのものです。

ローカル・ラベルは、goto ステートメントのターゲットとして使用することができ、ラベルが宣言されたブロック内から、そのラベルにジャンプすることができます。この言語拡張機能は、ネストされたループを含むマクロを書く場合に特に有用で、ローカル・ラベルのステートメント・スコープと通常のラベルの関数スコープとの違いを利用することができます。

ローカルに宣言されるラベルの構文

ステートメント式の中で、ローカル・ラベルの宣言は、左小括弧および左中括弧の直後に記述しなければならず、通常の宣言およびステートメントの前でなければなりません。ラベルは、ステートメント式のステートメント内に、名前とコロンを使用して通常の方法で定義します。

関連資料

157 ページの『ステートメント式 (IBM 拡張)』

197 ページの『ネストされた関数 (IBM 拡張)』

値としてのラベル (IBM 拡張)

現行関数またはそれを包含する関数の中で定義されているラベルのアドレスを取得して、void* 型の定数が有効な場所であればどこでも、そのアドレスを値として使用することができます。ラベルが単項演算子 && のオペランドのときは、「アドレス」がその戻り値です。ラベルのアドレスを値として使用できる機能は、C99 の拡張機能の 1 つで、GNU C を使用して開発されたプログラムの移植を容易にするためにインプリメントされています。

以下の例では、computed goto (計算型 goto) ステートメントにより、label1 および label2 の値が、関数内のこれらのラベル位置にジャンプするために使用されています。

```
int main()
{
   void * ptr1, *ptr2;
   ...
   label1: ...
   ...
   label2: ...
   ...
   ptr1 = &&label1;
   ptr2 = &&label2;
   if (...) {
      goto *ptr1;
   } else {
      goto *ptr2;
   }
   ...
}
```

関連資料

124 ページの『アドレス演算子 &』

148 ページの『ラベル値演算子 (IBM 拡張)』

計算型 goto ステートメント

170 ページの『goto ステートメント』

式ステートメント

式ステートメント は、式を含んでいます。式は、ヌルでもかまいません。

式ステートメントの構文

式ステートメントは式 を評価し、次に式の値を破棄します。式のない式ステートメントは、ヌル・ステートメントです。

ステートメントの例を次に示します。

```
printf("Account Number: ¥n");          /* call to the printf    */
marks = dollars * exch_rate;                /* assignment to marks     */
(difference < 0) ? ++losses : ++gain;  /* conditional increment */
```

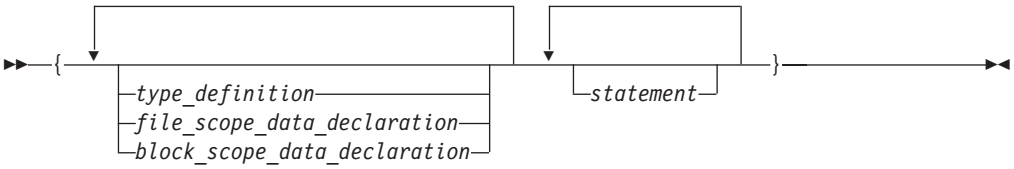
関連資料

115 ページの『第 7 章 式と演算子』

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・ステートメントの構文



ブロックは、ローカル・スコープを定義します。データ・オブジェクトがブロック内で使用可能であり、その ID が再定義されていない場合には、ネストされたすべてのブロックが、そのデータ・オブジェクトを使用できます。

関連資料

4 ページの『ブロック・スコープ』

43 ページの『auto ストレージ・クラス指定子』

ブロックの例

以下のプログラムは、ネストされたブロック内でデータ・オブジェクトの値がどのように変更されるかを示しています。

```
/**
 ** This example shows how data objects change in nested blocks.
 **/
  #include <stdio.h>

  int main(void)
  {
     int x = 1;                    /* Initialize x to 1  */
     int y = 3;

     if (y > 0)
     {
        int x = 2;                 /* Initialize x to 2  */
        printf("second x = %4d\n", x);
     }
     printf("first  x = %4d\n", x);

     return(0);
  }
```

プログラムは、以下の出力を作成します。

```
second x =    2
first  x =    1
```

x という名前の 2 つの変数が main の中で定義されています。 x の最初の定義は、main が実行されている間、ストレージを保存します。しかし、x の 2 番目の定義 (再定義) がネストされたブロック内で発生するため、 printf("second x = %4d¥n", x); は、x を前の行で定義された変数であると認識します。
printf("first x = %4d¥n", x); は、ネストされたブロックの一部ではないので、x は、x の最初の定義として認識されます。

ステートメント式 (IBM 拡張)

複合ステートメントは、中括弧で囲まれた複数のステートメントのシーケンスです。 GNU C では、小括弧内の複合ステートメントは、いわゆる、ステートメント式 の中に式として使用することができます。

ステートメント式の構文

ステートメント式の値は、構成全体に現れる最後の単純式の値です。最後のステートメントが式でない場合、その構成は void 型であって、値はありません。

このステートメント式を typeof 演算子と組み合わせると、各オペランドが一度だけ評価される関数に似た複雑なマクロを作成することができます。次に例を示します。

```
#define SWAP(a,b) ( {__typeof__(a) temp; temp=a; a=b; b=temp;} )
```

関連資料

154 ページの『ローカルに宣言されたラベル (IBM 拡張)』

選択ステートメント

選択ステートメントは、以下のステートメントで構成されます。

- if ステートメント
- switch ステートメント

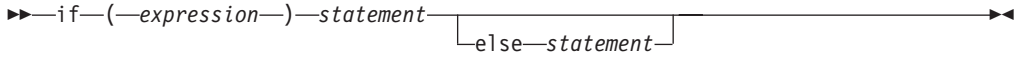
if ステートメント

if ステートメントは、複数の制御のフローを可能にする選択ステートメントです。

C では、if ステートメントを使用すると、指定されたテスト式の評価が非ゼロ値になった場合に、ステートメントを条件付きで処理することができます。テスト式は、算術型またはポインター型でなければなりません。

オプションで、if ステートメントで else 文節を指定することができます。テスト式の評価が 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 1 になった場合、その式に続くステートメントが実行され、else 文節は無視されます。

if ステートメントの構文



if ステートメントがネストされていて、else 文節が存在する場合、指定された else は、同じブロック内の直前の if ステートメントに関連付けられます。

任意の選択ステートメント (if、switch) に続く単一のステートメントは、オリジナルのステートメントを含んでいる複合ステートメントとして扱われます。結果として、そのステートメントで宣言されたすべての変数は、if ステートメントの後、スコープの外にあります。次に例を示します。

```
if (x)
int i;
```

は、以下と同等です。

```
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 文節を持たない、ネストされた if ステートメントを示しています。 else 文節は、常に、最も近い if ステートメントに関連付けられるため、特定の else 文節を強制的に正しい if ステートメントに関連付けるには、中括弧が必要なことがあります。この例では、中括弧を省略すると、else 文節は、ネストされた if ステートメントに関連付けられることになります。

```
if (kegs > 0) {
   if (furlongs > kegs)
      fxph = furlongs/kegs;
}
else
   fxph = 0;
```

以下の例は、else 文節の中にネストされた if ステートメントを示しています。この例では、複数の条件がテストされます。テストは、それらの条件が書かれている順序で行われます。 1 つのテストが非ゼロ値に評価されると、ステートメントが実行され、if ステートメント全体が終了します。

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

関連資料

107 ページの『第 6 章 変換』

51 ページの『ブール型』

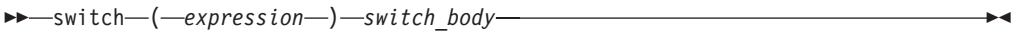
switch ステートメント

switch ステートメントは、 *switch* 式の値に基づいて、*switch* 本体の中の別のステートメントに制御を移すことができる選択ステートメントです。 switch 式の評価は、整数値または列挙値にならなければなりません。 switch ステートメントの本体には、以下のもので構成される *case* 文節 が含まれています。

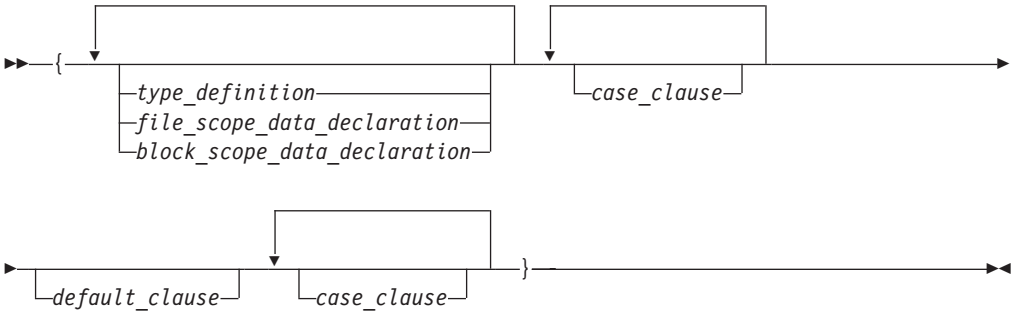
- case ラベル
- オプションの default ラベル
- case 式
- ステートメントのリスト

switch 式の値が case 式の 1 つの値と同じ場合、その case 式に続くステートメントが処理されます。そうでない場合、default ラベル・ステートメント (あれば) が処理されます。

switch ステートメントの構文



switch 本体 は、中括弧で囲まれ、定義、宣言、*case* 文節、および *default* 文節 を含むことができます。 case 文節と default 文節のそれぞれにステートメントを含めることができます。



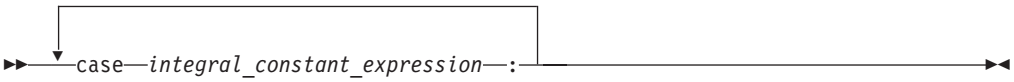
注: *type_definition*、*file_scope_data_declaration* または *block_scope_data_declaration* 内の初期化指定子は、無視されます。

case 文節 には、任意の数のステートメントが続く *case label* が含まれます。 CASE 節の形式は、次のとおりです。

Case 文節の構文

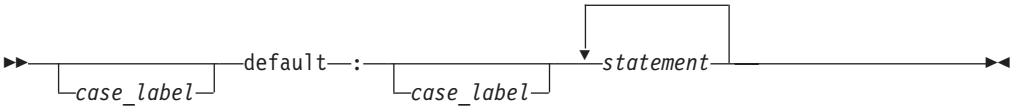
case ラベル には、case というワードと、それに続く整数定数式とコロンが入っています。各整数定数式の値は、別々の値を表していることが必要です。重複した case ラベルを持つことはできません。 1 つの case ラベルを置けるところであればどこでも、複数の case ラベルを置くことができます。 case ラベルの形式は、次のとおりです。

case ラベルの構文



default 文節 には、default ラベルと、それに続く 1 つ以上のステートメントが入っています。 case ラベルは、default ラベルのどちらの側にも置くことができます。 switch ステートメントに入れることができる default ラベルは、1 つだけです。 *default_clause* の形式は、次のとおりです。

default 文節のステートメント



switch ステートメントは、いずれか 1 つのラベルに続くステートメント、または switch 本体に続くステートメントに制御を渡します。switch 本体の前にある式の値によって、制御を受け取るステートメントが決まります。この式は *switch 式* と呼ばれます。

switch 式の値は、各 case ラベルの式の値と比較されます。一致している値が検出されれば、一致したその値が入っている case ラベルに続くステートメントに、制御が渡されます。一致する値はないが、switch 本体の中に default ラベルがある場合には、default ラベルの付いたステートメントに制御が渡されます。一致する値が見つからず、switch 本体の中のどこにも default ラベルがない場合には、switch 本体のどの部分も処理されません。

switch 本体の中のステートメントに制御が渡されると、 break ステートメントが検出されたとき、または switch 本体の中の最後のステートメントが処理されたときにのみ、制御は switch 本体を離れます。

必要であれば、整数拡張が、制御式上で実行されます。また、case ステートメント内のすべての式が、制御式と同じ型に変換されます。 switch 式は、整数型または列挙型への単一変換があれば、クラス型の式にもなります。

オプション **-qinfo=gen** を使用してコンパイルすると、失敗すべきでないときに失敗する case ラベルが検出されます。

switch ステートメントの制約事項

データ定義を switch 本体の先頭に置くことができますが、コンパイラーは、switch 本体の先頭にある auto および register 変数は、初期化しません。switch ステートメントの本体の中に、宣言を入れることができます。

switch ステートメントを使用して、初期化を飛び越えることはできません。

可変変更型の ID のスコープに switch ステートメントの case ラベルまたは default ラベルが含まれるときは、switch ステートメント全体がその ID のスコープ内にあると見なされます。すなわち、この ID の宣言は switch ステートメントの前になければなりません。

switch ステートメントの例

以下の switch ステートメントには、複数の case 文節と 1 つの default 文節が入っています。各文節には、関数呼び出しと break ステートメントが入っています。break ステートメントは、switch 本体内の各ステートメントに制御が移されていくのを防止します。

switch 式の評価が '/' となった場合、その switch ステートメントは関数 divide を呼び出すことになります。その後、switch 本体に続くステートメントに制御が渡されます。

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;

   default:
      printf("invalid key\n");
      break;
}
```

switch 式と case 式が一致した場合には、break ステートメントが検出されるまで、または switch 本体の終わりに達するまで、case 式に続くステートメントが処理されます。以下の例には、break ステートメントはありません。 text[i] の値が 'A' と等しい場合、 3 つのカウンターすべてが増分されます。 text[i] の値が

'a' と等しい場合には、lettera と total が増分されます。 text[i] が 'A' または 'a' と等しくない場合には、total のみが増分されます。

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

次の switch ステートメントは、複数の case ラベルに対して同じステートメントを実行します。

```
/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
 **/

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
     case 2:
        printf("month %d is a winter month\n", month);
        break;

     case 3:
     case 4:
     case 5:
        printf("month %d is a spring month\n", month);
        break;

     case 6:
     case 7:
     case 8:
        printf("month %d is a summer month\n", month);
        break;

     case 9:
     case 10:
     case 11:
        printf("month %d is a fall month\n", month);
        break;

     case 66:
```

```
    case 99:
    default:
       printf("month %d is not a valid month\n", month);
  }

  return(0);
}
```

式 month の値が 3 の場合には、次のステートメントに制御が渡されます。

```
printf("month %d is a spring month\n", month);
```

break ステートメントは、switch 本体に続くステートメントに制御を渡します。

関連資料

107 ページの『第 6 章 変換』

153 ページの『ラベル付きステートメント』

「XL C コンパイラー・リファレンス」の 中の『 -qinfo=gen』を参照

case ラベルと default ラベル

167 ページの『break ステートメント』

反復ステートメント

反復ステートメントは、以下のステートメントで構成されます。

- while ステートメント
- do ステートメント
- for ステートメント

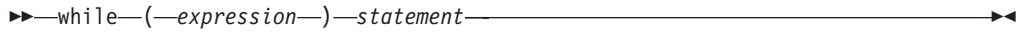
関連資料

51 ページの『ブール型』

while ステートメント

while ステートメント は、制御式の評価が 0 になるまで、ループの本体を繰り返し実行します。

while ステートメントの構文



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。式の評価が 0 の場合、ループの本体は実行されません。式が 0 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 0 でない場合でも、while ステートメントを終了できます。

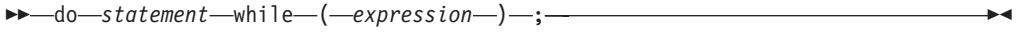
次の例では、式 ++index の値が MAX_INDEX より小さい間は、 item[index] は 3 倍され、印刷されます。 ++index の評価が MAX_INDEX になると、while ステートメントは終了します。

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d¥n", index, item[index]);
      ++index;
   }

   return(0);
}
```

do ステートメント

do ステートメント は、テスト式の評価が 0 になるまで、ステートメントを繰り返し実行します。処理の順序のために、そのステートメントは少なくとも 1 回は実行されます。

do ステートメントの構文



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 0 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 0 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 0 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。

次の例では、i が 5 より小さい間は、i を増分し続けます。

```
#include <stdio.h>

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

上記の例の出力は、以下のとおりです。

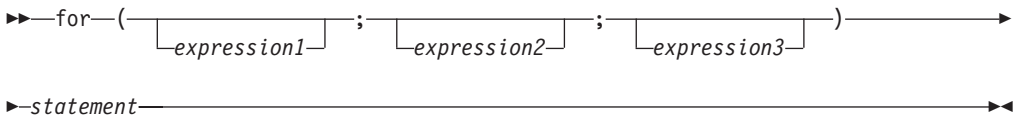
```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

for ステートメント

for 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 0 にならない場合に、ステートメントを繰り返し処理する。

for ステートメントの構文



expression1 は*初期化式* です。これは、ステートメント が始めて処理される前においてのみ評価されます。この式を使用すると、変数を初期化することができます。この式を使用して変数を宣言することもできますが、その変数は、静的として宣言されていてはなりません (その変数は自動でなければならず、また、register としても宣言されていてかまいません)。この式で、またはステートメント の中のどこででも、変数を宣言すると、その変数は、for ループの終わりでスコープの外に出ます。ステートメントの最初の反復の前に式の評価を行いたくない場合には、この式を省略することができます。

expression2 は*条件式* です。ステートメント の各反復の前に評価されます。*expression2* 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 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 0 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 1 によって、置き換えられたのと同様になり、この条件の不備により for ステートメントが終了しないことになります。

expression3 は、*statement* の毎回の反復後に評価されます。この式は、変数に対する増分、減分、または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 ステートメントがあると、 2 番目の式の評価が 0 でない場合でも、for ステートメントを終了できます。 *expression2* を省略する場合は、break、return、または goto ステートメントを使用して for ステートメントを終了する必要があります。

for ステートメントの例

次の for ステートメントは、count の値を 20 回印刷します。for ステートメントは、count の値を 1 に初期設定します。ステートメントが反復されるたびに count が増やされます。

```
int count;
for (count = 1; count <= 20; count++)
   printf("count = %d¥n", count);
```

次の一連のステートメントも同じタスクを実行します。for ステートメントの代わりに while ステートメントを使用していることに注意してください。

```
int count = 1;
while (count <= 20)
{
   printf("count = %d\n", count);
   count++;
}
```

以下の for ステートメントには、初期化の式が含まれていません。

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d\n", index,
   list[index]);
}
```

以下の for ステートメントは、scanf が e という文字を受け取るまで実行し続けます。

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '\n')
      continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c\n", letter);
}
```

以下の for ステートメントには、複数の初期化と増分が含まれています。コンマ演算子によってこの構造が可能となります。 for 式内の最初のコンマは、宣言の区切り子です。これは、i と j の 2 つの整数を宣言して、初期化します。 2 番目のコンマ (コンマ演算子) によって、ループ内の各ステップを通るたびに、i と j を両方とも増加することが可能になります。

```
for (int i = 0,
j = 50; i < 10; ++i, j += 50)
{
   cout << "i = " << i << "and j = " << j
   << endl;
}
```

以下の例は、ネストされた for ステートメントを示しています。このステートメントは、[5][3] という次元の配列の値を印刷します。

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d\n",
      table[row][column]);
```

row の値が 5 より小さい間、外部ステートメントが処理されます。外部の for ステートメントが実行されるたびに、内部の for ステートメントが column の初期値をゼロに設定します。そして、内部の for ステートメントのステートメントが 3 回実行されます。 column の値が 3 より小さい間は、内部ステートメントが実行されます。

分岐ステートメント

分岐ステートメントは、以下のステートメントで構成されます。

- break ステートメント
- continue ステートメント
- return ステートメント
- goto ステートメント
- IBM 計算後の goto ステートメント (IBM 拡張)

break ステートメント

break ステートメント を使用すると、反復 (do、for、while) ステートメントまたは switch ステートメントを終了して、論理終了以外の任意のポイントで、ステートメントから出ることができます。 break は、これらのステートメントのいずれかだけに使用できます。

break ステートメントの構文

反復するステートメントにおいては、break ステートメントはループを終了して、ループの外側にある次のステートメントに制御を移します。ネストされたステートメントの中では、break ステートメントは、囲んでいる最小の do、for、switch、または while ステートメントのみを終了します。

switch ステートメントにおいては、break は、制御を switch 本体から switch 本体の外側にある、次のステートメントに渡します。

関連資料

159 ページの『switch ステートメント』

continue ステートメント

continue ステートメント を使用すると、進行中のループの反復を終了することができます。プログラム制御は、continue ステートメントからループ本体の終わりに渡されます。

continue 文の形式は、次のとおりです。

continue ステートメントは、do、for、while などの反復ステートメントの本体の中にしか存在できません。

continue ステートメントは、反復するステートメントのアクション部分の処理を終了します。そして、制御を、ステートメントのループ連結部分へ移します。例えば、反復するステートメントが for ステートメントである場合、制御は、ステートメントの条件部分の 3 番目の式に移動します。次に、ステートメントの条件部分の 2 番目の式 (テスト) に移動します。

ネストされたステートメントの中では、continue ステートメントは、直接に continue ステートメントを囲んでいる do、for、または while ステートメントの現行の反復だけを終了します。

continue ステートメントの例

以下の例は、for ステートメントにおける continue ステートメントを示しています。continue ステートメントを使用すると、値が 1 以下の配列 rates のエレメントに対する処理がスキップされます。

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00\n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f\n", rates[i]);
   }

   return(0);
}
```

プログラムは、以下の出力を作成します。

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

以下の例は、ネストされたループにおける continue ステートメントを示しています。内部ループが配列 strings の中である数に遭遇すると、そのループの反復は終了します。処理は、内部ループの 3 番目の式から続けられます。内部ループは、'\0' エスケープ・シーケンスを検出した時点で終了します。

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
```

```
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '¥0';
      ++pointer)
      {                                   /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);

   return(0);
}
```

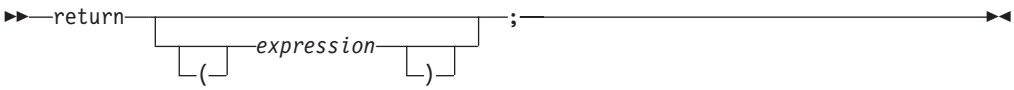
プログラムは、以下の出力を作成します。

```
letter count = 5
```

return ステートメント

return ステートメント は、現行の関数の処理を終了して、関数の呼び出し元に制御を戻します。

return ステートメントの構文



値を戻す関数は、return ステートメントが組み込まれていなければならず、その中には、*expression* が含まれていなければなりません。 C 非 void の戻りの型が宣言されている関数内の return ステートメントに式が指定されていない場合、コンパイラーは警告メッセージを発行します。

式のデータ型が関数の戻りの型と異なる場合には、式の値が、あたかも、関数の戻りの型が同じであるオブジェクトに割り当てられたかのように、戻り値の変換が行われます。

戻りの型 void の関数の場合、return ステートメントは必ず必要というわけではありません。 return ステートメントを検出することなくそのような関数の終わりに達すると、制御は、式のない return ステートメントが検出された場合と同じように呼び出し元へ渡されます。つまり、最終ステートメントの完了時に暗黙的な戻りが実行され、制御は自動的に呼び出し元関数に戻ります。

return ステートメントの例

return ステートメントの例を次に示します。

```
return;            /* Returns no value            */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1         */
return (x * x);    /* Returns the value of x * x  */
```

以下の関数は、整数の配列を検索して、変数 number と一致するものが存在するかどうか判別します。一致するものが存在すると、関数 match は i の値を戻します。一致するものが存在しない場合、関数 match は -1 (マイナス 1) という値を戻します。

```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
      if (number == array[i])
         return (i);
   return(-1);
}
```

1 つの関数に、複数の return ステートメントを含めることができます。次に例を示します。

```
void copy( int *a, int *b, int c)
{
   /* Copy array a into b, assuming both arrays are the same size */

   if (!a || !b)       /* if either pointer is 0, return */
      return;

   if (a == b)         /* if both parameters refer */
      return;          /*    to same array, return */

   if (c == 0)         /* nothing to copy */
      return;

   for (int i = 0; i < c; ++i;) /* do the copying */
      b[i] = a[1];
                       /* implicit return */
}
```

この例では、return ステートメントを使用して関数の早期終了が行われます。break ステートメントに似ています。

return ステートメントにある式は、そのステートメントが属している関数の戻りの型に変換されます。暗黙的な変換が不可能な場合、return ステートメントは無効です。

関連資料

107 ページの『第 6 章 変換』

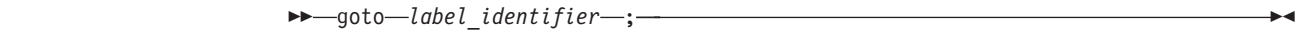
184 ページの『関数の戻りの型指定子』

185 ページの『関数からの戻り値』

goto ステートメント

goto ステートメント を使用すると、プログラムは、goto ステートメントで指定されたラベルに関連付けられたステートメントに無条件に制御を移します。

goto ステートメントの構文



goto ステートメントは、通常の処理シーケンスを妨げる可能性があるため、これを使用すると、プログラムの読み取りおよび保守が難しくなります。多くの場合、break ステートメント、continue ステートメント、または関数呼び出しを使用すれば、goto ステートメントを使用しなくてもすみます。

goto ステートメントを使用してアクティブ・ブロックを終了する場合、そのブロックから制御が移動するときに、すべてのローカル変数は破棄されます。

goto ステートメントを使用して、初期化を飛び越えることはできません。

goto ステートメントは、可変長配列のスコープ内にジャンプすることができますが、可変的に変更される型を持つオブジェクトの宣言を飛び越えてジャンプすることはできません。

以下の例は、ネストされたループからジャンプするために使用される goto ステートメントを示しています。この関数は、goto ステートメントを使用しなくても作成することができます。

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
   int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6\n");
}
```

計算後の goto ステートメント (IBM 拡張)

computed goto (計算型 goto) は、ターゲットが同一関数からのラベルである goto ステートメントです。ラベルのアドレスは void* 型の定数であり、これは、単項ラベル値演算子 && をラベルに適用することにより取得されます。computed goto のターゲットは実行時に判明し、同じ関数からの computed goto ステートメントのター

ゲットはすべて同じです。この言語フィーチャーは C99 の拡張機能であり、 GNU C を使用して開発されたプログラムの移植を容易にすることを目的としてインプリメントされています。

計算型 goto ステートメントの構文

**expression* は、型 `void*` の式です。

関連資料

43 ページの『auto ストレージ・クラス指定子』
153 ページの『ラベル付きステートメント』
154 ページの『値としてのラベル (IBM 拡張)』
148 ページの『ラベル値演算子 (IBM 拡張)』

ヌル・ステートメント

ヌル・ステートメント はオペレーションを実行しません。形式は次のとおりです。

ヌル・ステートメントは、ラベル付きステートメントのラベルを保持したり、あるいは反復するステートメントの構文を完了したりすることができます。

以下の例は、配列 `price` のエレメントを初期化します。初期化は `for` 式の中で起きるため、`for` 構文を終了するのに必要なものはステートメントのみで、オペレーションは必要ありません。

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

ブロック・ステートメントの終わりの前にラベルを必要とするときに、ヌル・ステートメントを使用できます。次に例を示します。

```
void func(void) {
  if (error_detected)
    goto depart;
  /* further processing */
  depart: ;  /* null statement required */
}
```

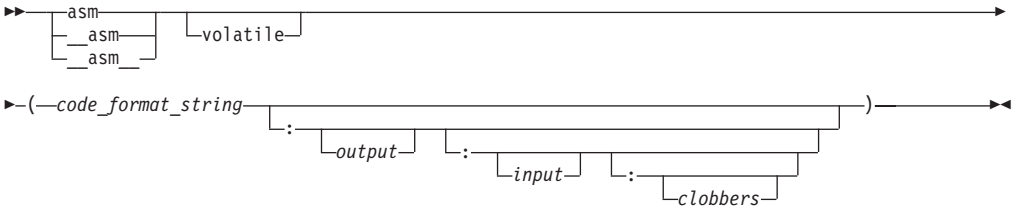
インライン・アセンブリー・ステートメント (IBM 拡張)

拡張言語レベルの下では、コンパイラーは、C ソース・ステートメント間での組み込みアセンブリー・コード・フラグメントを完全にサポートします。この拡張機能は、元々は GNU C を使用して開発された汎用システム・プログラミング・コード、およびオペレーティング・システム・カーネルおよびデバイス・ドライバーで使用するために、インプリメントされています。

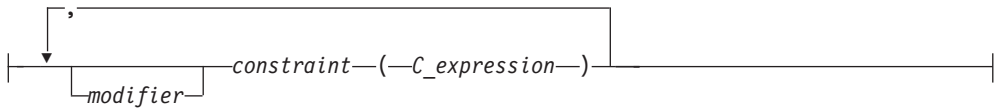
キーワード asm は、アセンブリー・コードを表します。コンパイルで厳格な言語レベルが使用されると、C コンパイラーは宣言内のキーワード asm を認識して、それを無視します。

構文は次のとおりです。

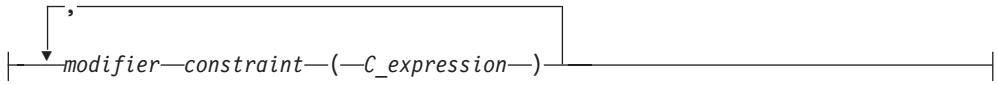
asm ステートメントの構文 - ローカル・スコープのステートメント



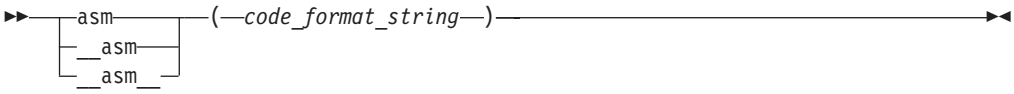
input:



output:



asm ステートメントの構文 - グローバル・スコープのステートメント



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

code_format_string は、asm 命令のソース・テキストであり、 printf フォーマット指定子に似たストリング・リテラルです。アセンブリー命令で使用するオペランドによりますが、ストリングにはゼロまたは 1 つ以上の修飾子が含まれ、それぞれはコンマで区切られ、入力または出力オペランドに対応しています。

% 修飾子は、次のいずれかの形式になります。

- *%integer*。ここで、*integer* は入力または出力オペランドのシーケンス番号を参照します。
- *%*[*symbolic_name*]。ここで *symbolic_name* は、オペランド・リストで参照されます。シンボリック・オペランド名は、C の ID とは何の関係もありません。任意

の名前を使用でき、既存の C のシンボル名さえも使用できます。ただし、同一のアセンブリー・ステートメントの中では、2 つのオペランドが同一のシンボル名は使用できません。

input は、コンマで区切られた、ゼロまたは 1 つ以上のオペランドから成り立ちます。各オペランドは *constraint*(*C_expression*) の対から成り立ちます。

output は、コンマで区切られた、ゼロまたは 1 つ以上のオペランドから成り立ちます。各オペランドは *constraint*(*C_expression*) の対から成り立ちます。出力オペランドは、= または + 修飾子 (下で説明) 、および追加の % または & 修飾子 (オプション) により制約する必要があります。

修飾子 は、次のいずれかです。

= この命令では、オペランドが書き込み専用であることを示します。前の値は廃棄され、出力データで置き換えられます。

+ オペランドが、命令によって読み込みと書き込みの両方に使用されることを示します。

& 入力オペランドを使用して、命令が終了する前にオペランドが変更される場合があることを示します。入力として使用されるレジスターは、ここでは再使用できません。

% 命令が、このオペランドと次のオペランドに対して可換性があることを宣言します。 このことは、このオペランドと次のオペランドの順序は、命令の生成時に交換できることを意味します。この修飾子は、入力または出力オペランドで使用できますが、最後のオペランドでは使用できません。

constraint は、許可されるオペランドの種類を記述するストリング・リテラルで、1 つの制約につき 1 文字です。指定できる制約は以下のとおりです。

b ゼロ以外の汎用レジスターを使用する。

c CTR レジスターを使用する。

f 浮動小数点レジスターを使用する。

g 汎用レジスター、メモリー、または即値オペランドを使用する。

h CTR または LINK レジスターを使用する。

i オペランドに即値整数またはストリング・リテラルを使用する。

l CTR レジスターを使用する。

m マシンがサポートしているメモリー・オペランドを使用する。

n 即値整数を使用する。

o オフセット可能なメモリー・オペランドを使用する。

r 汎用レジスターを使用する。

s オペランドにストリング・リテラルを使用する。

v ベクトル・レジスターを使用する。

0, 1, 2, ...
マッチング制約。出力では、対応する入力と同じレジスターを割り当てる。

I, J, K, L, M, N, O, P
定数値。オペランド内の式を折りたたんで、値を %i 指定子に置き換える。これらの制約には、オペランドに対する最大値を、次のように指定します。

- I - 符号付き 16 ビット
- J - 符号なし 16 ビット 左シフト 16 ビット
- K - 符号なし 16 ビット定数
- L - 符号付き 16 ビット 左シフト 16 ビット
- M - 符号なし 31 より大きい定数
- N - 符号なし 2 の累乗の定数
- O - ゼロ
- P - 符号付で、その否定形が符号付 16 ビット定数

C_expression は C 式です。この式の値は、asm 命令のオペランドとして使用されます。出力オペランドは変更可能な左辺値でなければなりません。 *C_expression* は、それに指定された制約と整合している必要があります。例えば、i を指定した場合には、オペランドは整数定数であることが必要です。

注: ポインター式が *input* または *output* で使用される場合、アセンブリー命令は ANSI 別名割り当て規則を順守する必要があります (詳しくは、 86 ページの『型ベースの別名割り当て』を参照)。つまり、ポインター式のオペランドの値を使用する間接アドレッシングは、ポインター型と整合している必要があります。そうでない場合、コンパイル時に **-qalias=ansi** オプションを使用不可にする必要があります。

clobbers は、二重引用符で囲み、コンマで区切ったレジスター名のリストです。これらのレジスターは、asm 命令により更新できるレジスターです。以下に、有効なレジスター名を示します。

r0 から r31
汎用レジスター

f0 から f31
浮動小数点レジスター

lr リンク・レジスター

ctr ループのカウント、減分およびブランチ用レジスター

fpscr 浮動小数点の状況および制御レジスター

xer 固定小数点例外レジスター

cr0 から cr7
条件レジスター

v0 から v31
ベクトル・レジスター (選択されたプロセッサー上のみ)

関連資料

13 ページの『ID』

46 ページの『register ストレージ・クラス指定子』

「XL C コンパイラー・リファレンス」の 中の『 -qalias=ansi』を参照

インライン・アセンブリー・ステートメントの例

次の例では、

```
int a ;
int b = 100 ;
int c = 200 ;
 asm("add %0, %1, %2" : "=r"(a) : "r"(b) , "r"(c));
```

add は、アセンブラーによって理解される命令の命令コードです。%0、%1、および %2 は、出力/入力オペランド欄で C 式で置換されるオペランドです。出力オペランドは、= 制約を使用して、変更可能オペランドが必要であることを示します。r 制約は、汎用レジスターが必要であることを示すために使用されます。同様に、入力オペランドの r は、汎用レジスターが必要であることを示します。これらの制約事項の範囲内で、コンパイラーは %0、%1、および %2 を置換するレジスターを自由に選択できます。

次の例では、% 制約修飾子はコンパイラーに対して、交換によってより良いコードを生成できると判断できる場合は、オペランド a と b を交換できることを伝えています。

```
asm("add %0, %1, %2" : "=r"(c) : "%r"(a), "r"(b));
```

次の例は、入力および出力オペランドのシンボル名の使用法を示しています 。

```
int a ;
int b = 1, c = 2, d = 3 ;
__asm("addc %[result],%[first],%[second]" : [result]"=r"(a) : [first]"r"(b), [second]"r"(d));
```

インライン・アセンブリー・ステートメントの制約事項

次は、インライン・アセンブリー・ステートメントの使用に関する制限です。

- アセンブラー命令は、asm ステートメント内で自己完結していることが必要です。 asm ステートメントは、命令を生成するために使用できるだけです。プログラムのその他の部分への関連付けは、すべて、出力および入力オペランド・リストを使用して設定する必要があります。
- オペランド・リストを通さずに外部シンボルを直接参照することは、サポートされていません。
- 対のレジスターを必要とするアセンブラー命令は、制約によって指定できないので、サポートされません。例えば、_Decimal128 オペランドでは %f 制約を使用できません。

関連資料

アセンブリー・ラベル (IBM 拡張)

指定したレジスターの変数 (IBM 拡張)


「XL C コンパイラー・リファレンス」の 中の『 -qasm』を参照

第 9 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 プログラムにおける関数は、すべての入力に対して一貫性のある出力を生成するとは限らず、出力をまったく生成しないこともあり、副次作用を持つこともあります。関数は、パラメーター・リストのパラメーター (存在する場合) をオペランドとする、ユーザー定義の演算と考えることができます。

関数については、以下の情報があります。

- 関数の宣言と定義
- 関数のストレージ・クラス指定子
- 関数指定子
- 関数の戻りの型指定子
- 関数宣言子
- 関数属性 (IBM 拡張)
- main() 関数
- 関数呼び出し
- 関数を指すポインター
- ネストされた関数 (IBM 拡張)

関数の宣言と定義

関数*宣言* と関数*定義* の違いは、データ宣言とデータ定義の違いに似ています。宣言では関数の名前と特性を設定しますが、関数にストレージを割り振りません。一方、*定義* では、関数の本体を指定し、関数に ID を関連付け、関数にストレージを割り振ります。したがって、次の例で宣言された ID は、

```
float square(float x);
```

ストレージを割り振りません。

関数定義 には、関数宣言と関数本体が含まれます。本体とは、関数の処理を実行するステートメントのブロックです。この例で宣言された ID はストレージを割り振ります。これらは、宣言と定義の両方になります。

```
float square(float x)
{ return x*x; }
```

1 つの関数をプログラム内で複数回宣言することができますが、特定の関数のすべての宣言は互換性がなければなりません。つまり、戻りの型が同じで、パラメーターの型が同じでなければなりません。ただし、1 つの関数には 1 つの定義しか認められません。通常、宣言はヘッダー・ファイルに入れますが、定義はソース・ファイルに入れます。

関連資料

13 ページの『ID』

41 ページの『データ宣言とデータ定義の概要』

54 ページの『void 型』

関数宣言

前に戻りの型が付き、後にパラメーター・リストが続く関数 ID は、*関数宣言* または*関数プロトタイプ* と呼ばれます。プロトタイプは、コンパイラーに、その関数を使用する前に関数の形式と存在を知らせます。コンパイラーは、関数呼び出しのパラメーターと関数宣言におけるパラメーターとの不一致を検査することができます。コンパイラーは、引数の型の検査および引数の変換のためにも、この宣言を使用します。

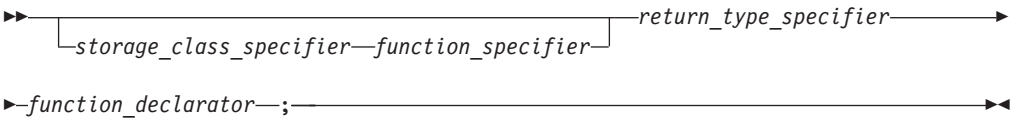
関数の呼び出しが行われた地点で関数宣言が可視でない場合、コンパイラーは、`extern int func();` の暗黙宣言を想定します。ただし、C99 に準拠するためには、ユーザーは、関数を呼び出す前に、その関数をプロトタイプ化しておくべきです。

関数を宣言する際のエレメントは、次のとおりです。

- 181 ページの『関数のストレージ・クラス指定子』。これは、リンケージを指定します。
- 184 ページの『関数の戻りの型指定子』。これは、戻される値のデータ型を指定します。
- 182 ページの『関数指定子』。これは、関数の追加プロパティーを指定します。
- 186 ページの『関数宣言子』。これは、パラメーターのリストだけでなく、関数の ID を含みます。

関数宣言の形式はすべて、次のとおりです。

関数宣言の構文



IBM さらに、XL C では、GNU C との互換性のために、ユーザーが *属性* を使用して関数のプロパティーを変更できるようになりました。これについては、188 ページの『関数属性 (IBM 拡張)』で説明しています。

関連資料

5 ページの『関数プロトタイプ・スコープ』

153 ページの『第 8 章 ステートメント』

関数定義

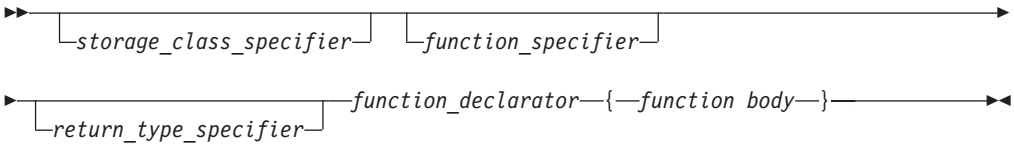
関数定義のエレメントは、次のとおりです。

- 181 ページの『関数のストレージ・クラス指定子』。これは、リンケージを指定します。
- 184 ページの『関数の戻りの型指定子』。これは、戻される値のデータ型を指定します。
- 182 ページの『関数指定子』。これは、関数の追加プロパティーを指定します。
- 186 ページの『関数宣言子』。これは、パラメーターのリストだけでなく、関数の ID を含みます。
- *関数本体*。これは、その関数が実行するアクションを表す、中括弧で囲まれた一連のステートメントです。

IBM さらに、XL C では、GNU C との互換性のために、ユーザーが *属性* を使用して関数のプロパティーを変更できるようになりました。これについては、188 ページの『関数属性 (IBM 拡張)』で説明しています。

関数定義の形式は、次のとおりです。

関数定義の構文 (C のみ)



関数宣言の例

以下のコード・フラグメントは、それぞれ、関数宣言 (または *プロトタイプ*) を示しています。最初の部分は、2 つの整数の引数を採用し、戻りの型が `void` である関数 `f` を宣言しています。

```
void f(int, int);
```

このフラグメントは、固定文字を指すポインターを取り、整数を戻す関数を指すポインター `p1` を宣言しています。

```
int (*p1) (const char*);
```

次のコードは、関数 `f1` を宣言し、関数 f1 は、1 つの整数の引数を取り、整数の引数を取って整数を戻す関数へのポインターを戻します。

```
int (*f1(int)) (int);
```

関数 `f1` の複雑な戻りの型に対して、上記の関数の代わりに、`typedef` を使用することができます。

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

次の宣言は、最初の引数として定数整数を取る外部関数 `f2()` の宣言です。この宣言は、複数の型がある、可変数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, ...);
```

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 の戻りの型をもっています。

```
class X
{
public:
      int f6() const;
};
```

関数定義の例

次の例は、関数 sum の定義です。

```
int sum(int x,int y)
{
   return(x + y);
}
```

関数 sum には、外部リンケージがあり、int 型のオブジェクトを戻し、 x および y と宣言された int 型の 2 つのパラメーターがあります。この関数本体には、x と y の合計を戻す 1 個のステートメントが入っています。

次の関数 set_date は、date 型の構造体へのポインターをパラメーターとして宣言します。 date_ptr は、ストレージ・クラス指定子 register を持っています。

```
void set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

互換性のある関数 (C のみ)

互換性のある 2 つの関数型の場合、次の要件を満たす必要があります。

- パラメーターの数 (および省略符号の使用) を一致させなければなりません。
- 互換性のある戻りの型を持たなければなりません。
- 対応するパラメーターは、デフォルト引数プロモーションのアプリケーションの結果による型と、互換性がなければなりません。

2 つの関数型の複合型は、次のように決定されます。

- 関数の型の 1 つにパラメーター型リストがある場合、複合型は、同じパラメーター型リストを持つ関数プロトタイプであること。
- 両方の関数型がパラメーター型リストを持つ場合、各パラメーターの複合型は次のように決定されます。
  - 異なるランクのパラメーターの複合は、デフォルト引数プロモーションのアプリケーションの結果による型です。
  - 配列または関数型を持つパラメーターの複合は、調整された型です。
  - 修飾された型を持つパラメーターの複合は、宣言された型の非修飾版です。

例えば、2 つの関数宣言は以下のようになります。

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

結果の複合型は次のようになります。

```
int f(int (*)(char *), double (*)[3]);
```

関数宣言子が関数宣言の一部でない場合、パラメーターは不完全型にすることができます。このパラメーターは、宣言子指定子のシーケンスで [*] 表記を使用して、可変長の配列の型を指定することもできます。以下は、互換性のある関数プロトタイプ宣言子の例です。

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

関連資料

39 ページの『データ・オブジェクトの概要』

互換型および複合型

関数のストレージ・クラス指定子

関数の場合は、ストレージ・クラス指定子は、関数のリンケージを決めます。デフォルトで、関数定義は、外部リンケージを持ち、他のファイルで定義された関数で呼び出すことができます。 1 つの例外はインライン関数です。インライン関数は、デフォルトで内部結合を持っているものとして処理されます。詳しくは、183 ページの『インライン関数のリンケージ』を参照してください。

ストレージ・クラス指定子は、関数宣言と関数定義の両方で使用できます。関数のストレージ・クラス・オプションは、次の 2 つだけです。

- `static`
- `extern`

関連資料

8 ページの『プログラム・リンケージ』

43 ページの『ストレージ・クラス指定子』

static ストレージ・クラス指定子

`static` ストレージ・クラス指定子を使って宣言された関数は、内部結合を持ちます。これは、この関数が、関数が定義されている変換単位内でしか呼び出しできないことを意味します。

`static` ストレージ・クラス指定子は、関数宣言がファイル・スコープにある場合のみ、関数宣言で使用できます。ブロック内の関数を `static` と宣言することはできません。

関連資料

44 ページの『static ストレージ・クラス指定子』

8 ページの『内部リンケージ』

182 ページの『inline 関数指定子』

extern ストレージ・クラス指定子

extern ストレージ・クラス指定子を宣言された関数は外部リンケージを持ちます。つまり、その関数は、他の変換単位から呼び出すことができます。キーワード extern はオプションです。ストレージ・クラス指定子を指定しない場合は、関数に外部結合を持つと想定されています。

関連資料

45 ページの『extern ストレージ・クラス指定子』

9 ページの『外部リンケージ』

『inline 関数指定子』

197 ページの『関数を指すポインター』

関数指定子

関数宣言および関数定義に使用できる関数指定子は、次に説明する inline のみです。

inline 関数指定子

インライン関数とは、コンパイラーが、個別の命令セットをメモリー内に作成するのではなく、関数定義からのコードを呼び出し元関数のコードに直接コピーする関数です。関数コード・セグメントとの間で制御権を移動するのではなく、変更された関数本体のコピーを関数呼び出しの代わりに直接使用することができます。このようにすると、関数呼び出しのパフォーマンス・オーバーヘッドを回避することができます。 inline 指定子の使用は、インライン展開を実行できるという提案をコンパイラーに行うだけです。コンパイラーはその提案を自由に無視できます。

main を除くすべての関数は、inline 関数指定子を使用してインラインとして宣言または定義することができます。静的ローカル変数を、インライン関数の本体で定義することは許可されていません。

次のコード・フラグメントはインライン関数定義を示しています。

```
inline int add(int i, int j) { return i + j; }
```

inline 指定子を使用しても、関数の意味は変わりません。ただし、関数のインライン展開を行うと、実引数の評価の順序が保たれなくなる場合があります。

インライン関数の最も効果的なコーディング方法は、インライン関数の定義をヘッダー・ファイルに配置し、次に、そのヘッダーを、インライン化したい関数への呼び出しを含む任意のファイルに組み込むことです。

注: inline 指定子は次のキーワードで表されます。

- inline キーワードは、**xlc** または **c99**、または **-qlanglvl=stdc99** または **-qlanglvl=extc99** オプション、あるいは **-qkeyword=inline**を使用したコンパイルで認識されます。 __inline__ キーワードは、すべての言語レベルで認識されますが、このキーワードのセマンティクスについては後述の 183 ページの『インライン関数のリンケージ』を参照してください。

インライン関数のリンケージ

C インライン関数は、デフォルトで静的結合を持っているものとして処理されます。すなわち、インライン関数は単一の変換単位内でのみ可視です。 よって以下の例では、関数 foo がまったく同じ方法で定義されていますが、ファイル a.c の foo とファイル b.c の foo は別の関数として処理されます。2 つの関数の本体が生成され、メモリー内の異なる 2 つのアドレスが割り当てられます。

```
// a.c

#include <stdio.h>

inline int foo(){
return 3;
}

void g() {
printf("foo called from g: return value = %d, address = %p¥n", foo(), &foo);
}


// b.c

#include <stdio.h>

inline int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d, address = %p¥n", foo(), &foo);
g();
}
```

コンパイル済みプログラムの出力は、次のようになります。

```
foo called from main: return value = 3, address = 0x10000580
foo called from g: return value = 3, address = 0x10000500
```

インライン関数は内部結合を持っているものとして処理されますので、インライン関数の定義は、別の変換単位内で同じ名前を持つ関数の、通常の外部定義と共存できます。ただし、インライン定義を含むファイルから関数を呼び出す場合、コンパイラーはその呼び出しに対して、同じファイルに定義されたインライン・バージョンまたは 別のファイルに定義された外部バージョンのいずれか を選択できます。プログラムは、呼び出されるインライン・バージョンに依存しません。以下の例では、関数 g からの foo の呼び出しで、6 または 3 のいずれかを戻します。

```
// a.c

#include <stdio.h>

inline int foo(){
return 6;
}

void g() {
printf("foo called from g: return value = %d¥n", foo());
}


// b.c
```

```
#include <stdio.h>

int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d¥n", foo());
g();
}
```

同様に、関数を extern inline として定義する場合、または inline 関数を extern として再宣言する場合は、その関数は単に通常の外部関数となるだけで、インライン化されません。

IBM 末尾に下線を付けて __inline__ キーワードを指定すると、コンパイラーはインライン関数に GNU C のセマンティクスを使用します。 C99 のセマンティクスに比べると、__inline__ として定義された関数は外部定義のみを提供し、static __inline__ と定義された関数は内部結合 (C99 と同様) を持つインライン定義を提供します。さらに、extern __inline__ と定義された関数は、最適化を使用可能にしてコンパイルすると、同じ関数のインライン定義と外部定義の共存を許可します。インライン関数の GNU C インプリメンテーションについて詳しくは、GCC 資料 (http://gcc.gnu.org/onlinedocs/ で入手可能) を参照してください。 IBM

関連資料

190 ページの『always_inline 関数属性 (IBM 拡張)』

192 ページの『noinline 関数属性』

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

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

181 ページの『static ストレージ・クラス指定子』

182 ページの『extern ストレージ・クラス指定子』

194 ページの『main() 関数』

関数の戻りの型指定子

関数の結果は*戻り値* と呼ばれ、戻り値のデータ型は*戻りの型* と呼ばれます。

関数宣言に戻りの型が指定されていない場合、コンパイラーは、暗黙の戻りの型 int を想定します。ただし、C99 に準拠するために、その関数が int を戻すかどうかに関係なく、すべての関数宣言および関数定義に戻りの型をユーザーが指定するべきです。

関数は、配列型および関数型を除いて、値の型を戻すように定義できます。これら 2 つの例外は、配列または関数にポインターを戻すことにより、処理しなければなりません。関数が値を戻さない場合、void が、関数宣言および定義での型指定子になります。

関数は、volatile または const 型のデータ・オブジェクトを戻すものとして宣言することはできません。しかし、volatile または const オブジェクトへのポインターを戻すことはできます。

関数は、ユーザー定義型である戻りの型を持つことができます。次に例を示します。

```
enum count {one, two, three};
enum count counter();
```

ユーザー定義型は、関数宣言内でも定義できます。

```
enum count{one, two, three} counter();   // legal
```

関連資料

50 ページの『型指定子』

169 ページの『return ステートメント』

関数からの戻り値

関数が戻りの型 void を持つように定義された場合、その関数は値を戻さないはずです。

関数が void 以外の戻りの型を持つように定義された場合、その関数は値を戻すはずです。C99 に厳密に準拠するコンパイルでは、戻りの型を定義された関数は、戻される値を入れる式が含まれていなければなりません。

関数が値を戻す場合、その値は、それが定義されている関数の戻りの型に暗黙的に変換された後、return ステートメントを介して関数の呼び出し元に戻されます。以下のコードは、return ステートメントを含む関数定義を示しています。

```
int add(int i, int j)
{
  return i + j; // return statement
}
```

以下のコードで示すように、関数 add() を呼び出すことができます。

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

この例では、return ステートメントは、戻された型の変数を初期化します。変数 answer を int の値 30 で初期化しています。戻された式の型を、戻りの型と照らし合わせて検査します。必要に応じて、すべての標準型変換およびユーザー定義の型変換が適用されます。

関数が呼び出されるたびに、自動ストレージを持つその変数の新しいコピーが作成されます。関数が終了した後に、これらの自動変数のストレージは再利用されることがあるので、自動変数を指すポインターは戻してはなりません。

関連資料

169 ページの『return ステートメント』

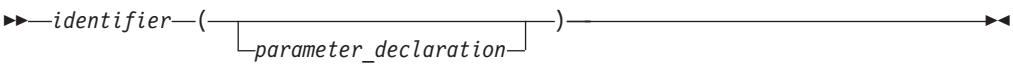
43 ページの『auto ストレージ・クラス指定子』

関数宣言子

関数宣言子は以下のエレメントで構成されます。

- *ID*、すなわち、名前
- 『パラメーター宣言』。これは、関数呼び出しで関数に渡すことができるパラメーターを指定します。

関数宣言子の構文 (C のみ)



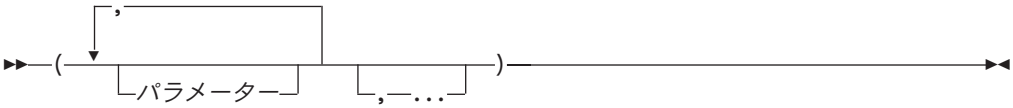
関連資料

82 ページの『宣言子の例』

パラメーター宣言

関数宣言子は、関数が別の関数から呼び出されたとき、また自分自身で呼び出したとき、その関数に渡すことができるパラメーターのリストを含みます。

関数仮パラメーター宣言の構文



パラメーター

C 関数*定義* の中に空の引数リストがある場合、その関数が引数を取らないことを表します。関数*宣言* の中の空の引数リストは、その関数は任意の数の引数、または任意の型の引数を取ることができることを表します。したがって、次の例は、

```
int f()
{
...
}
```

関数 `f` は引数を取らないことを表します。しかし、次の例は、

```
int f();
```

単に、パラメーターの数および型は不明であることを表しているだけです。関数が引数を取らないことを明示的に示すには、キーワード `void` を使用して関数を定義してください。

パラメーター指定の終わりにある省略符号は、関数が、可変数のパラメーターを持っていることを指定するために使用します。パラメーターの数は、パラメーター指定の数に等しいか、またはそれより多くなります。

```
int f(int, ...);
```

少なくとも 1 つのパラメーター宣言と省略符号の前のコンマは、C では両方とも必須です。

パラメーターの型

関数宣言、すなわち、プロトタイプでは、各パラメーターの型を指定しなければなりません。 関数定義 では、パラメーターの型が指定されていない場合、 int と想定されます。

ユーザー定義型の変数を、次の例に示したように、パラメーター宣言で宣言することができます。この例では、x は、初めて宣言されています。

```
struct X { int i; };
void print(struct X x);
```

ユーザー定義型は、パラメーター宣言内でも定義できます。

```
void print(struct X { int i; } x);   // legal
```

パラメーターの名前

関数定義 では、パラメーターの ID は必須です。関数宣言、すなわち、プロトタイプでは、ID の指定はオプショナルです。したがって、次の例は、関数宣言では有効です。

```
int func(int,long);
```

関数仮パラメーター宣言の中の静的配列指標 (C のみ)

一部のコンテキストの場合を除いて、添え字なし配列名 (例えば、 region であって、region[4] ではない) は、配列が既に宣言されている場合、値が配列の最初のエレメントのアドレスであるポインターを表します。関数のパラメーター・リスト内の配列型も、対応するポインター型に変換されます。配列が関数本体内からアクセスされると、引数配列のサイズに関する情報は失われます。

この情報は最適化に有用なので、この情報を保持しておくには、static キーワードを使用して引数配列の指標を宣言するとよいでしょう。定数式は、最適化の前提事項として使用できる最小のポインター・サイズを指定します。 static キーワードの、この特殊な使用法は、厳しい規定があります。このキーワードは、最外部の配列型派生の中でのみ、そして関数仮パラメーター宣言でのみ使用できます。関数の呼び出し側がこのような制限に従っていない場合、振る舞いは未定義です。

次の例は、このフィーチャーの使われ方を示しています。

```
void foo(int arr [static 10]);       /* arr points to the first of at least
                                           10 ints                          */
void foo(int arr [const 10]);        /* arr is a const pointer              */
void foo(int arr [static const i]);  /* arr points to at least i ints;
                                           i is computed at run time.       */
void foo(int arr [const static i]);  /* alternate syntax to previous example */
void foo(int arr [const]);           /* const pointer to int                */
```

関連資料

44 ページの『static ストレージ・クラス指定子』
87 ページの『配列』
141 ページの『配列添え字演算子 [ ]』
54 ページの『void 型』
50 ページの『型指定子』
71 ページの『型修飾子』

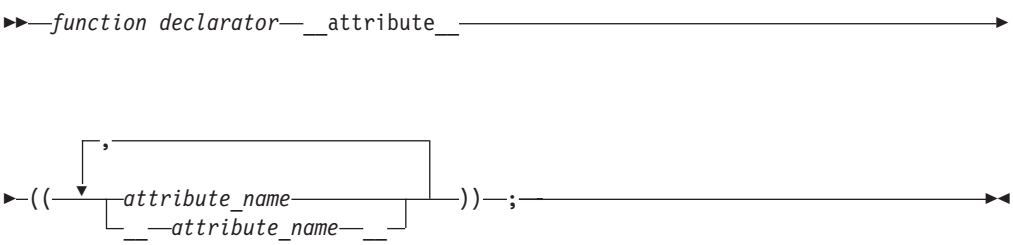
関数属性 (IBM 拡張)

関数の属性は、GNU C で開発されたプログラムの移植性を高めるためにインプリメントされた拡張機能です。関数に属性を指定することは、コンパイラーが関数呼び出しを最適化するのを支援し、コードをより多面的に検査するようにコンパイラーに指示する明示的な方法です。追加機能を提供する属性もあります。

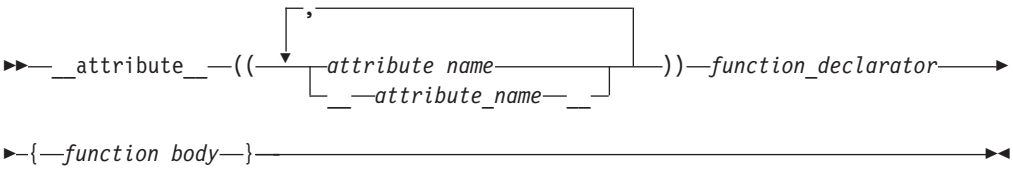
IBM C は GNU C 関数属性のサブセットを実装しています。 特定の関数属性がインプリメントされていない場合、その指定は受け入れられますが、セマンティクスは無視されます。これらの言語フィーチャーは、拡張言語レベルの 1 つでコンパイルされるとき、一括で使用可能になります。

関数属性は、キーワード __attribute__、その後に続く属性名、さらに、その属性名が必要とする追加引数によって指定されます。関数の __attribute__ 指定は、関数の宣言または定義に含まれます。構文の形式は次のとおりです。

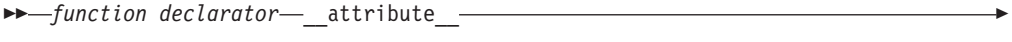
関数属性の構文: 関数宣言

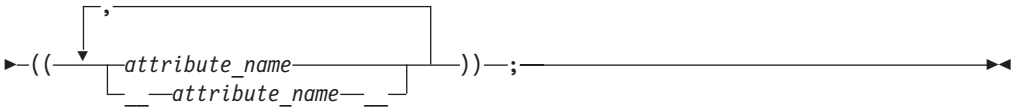


関数属性の構文： 関数定義 (C のみ)



関数属性の構文： 関数定義 (C++ のみ)





関数宣言内の関数属性は、括弧に入れたパラメーター宣言を含めて、常に宣言子の後に書きます。

```
/* Specify the attribute on a function prototype declaration */
void f(int i, int j) __attribute__((individual_attribute_name));
void f(int i, int j) { }
```

C++ C++ では、属性指定も、関数に対して存在することがある例外宣言の後に書かなければなりません。

C 古いスタイルのパラメーター宣言の解析にはあいまいさがあるので、関数定義での属性指定は宣言子の*前*で 行わなければなりません。

```
int __attribute__((individual_attribute_name)) foo(int i) { }
```

形式 *__attribute_name__* (すなわち、前後に下線文字が 2 つずつ付いた属性名) を使用する関数属性の指定は、同じ名前のマクロと名前が競合する可能性を小さくします。

次の関数属性がサポートされています。

- alias 関数属性 (IBM 拡張)
- always_inline 関数属性 (IBM 拡張)
- format 関数属性 (IBM 拡張)
- format_arg 関数属性 (IBM 拡張)
- noinline 関数属性 (IBM 拡張)
- noreturn 関数属性 (IBM 拡張)
- pure 関数属性 (IBM 拡張)
- weak 関数属性 (IBM 拡張)

関連資料

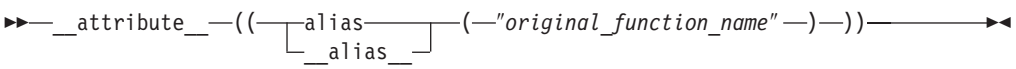
77 ページの『型属性 (IBM 拡張)』

101 ページの『変数属性 (IBM 拡張)』

alias 関数属性

alias 関数属性があると、関数宣言は、オブジェクト・ファイルの中で他のシンボルに対する別名として現れます。この言語フィーチャーは、重複した名前または扱いにくい名前に対処する手法として利用できます。

alias 関数属性の構文



別名の関数は、その別名の指定の後に、この関数属性で定義することができます。C は、同じコンパイル単位内に別名関数の定義がなくても、別名の指定を許可します。

次の例は、bar を __foo の別名と宣言しています。

```
/* C only */
void __foo(){   /*  function body  */   }
void bar() __attribute__((alias("__foo")));
```

コンパイラーは、bar の宣言と __foo の定義の整合性の検査は行いません。そのような整合性は、プログラマーに任されています。

関連資料

13 ページの『ID』

193 ページの『weak 関数属性』

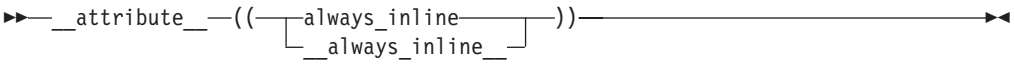
「XL C コンパイラー・リファレンス」の #pragma weak を参照

105 ページの『weak 変数属性』

always_inline 関数属性 (IBM 拡張)

always_inline 関数属性は、コンパイル時に最適化が指定されたかどうかに関係なく、inline 関数をインライン化するようにコンパイラーに指示します。ただし、プログラムが no-opt レベルでコンパイルされる場合、この属性の効果はありません。inline 指定のない関数にこの属性を指定した場合も、効果はありません。この属性は、インライン化コンパイラー・オプションより優先されます。

always_inline 関数属性の構文



関連資料

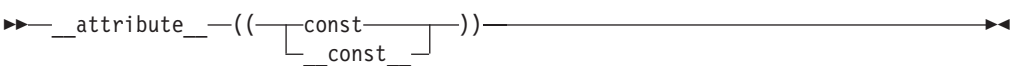
182 ページの『inline 関数指定子』

192 ページの『noinline 関数属性』

const 関数属性

const 関数属性は、その関数の呼び出し回数が、ソース・コードで指定された回数より少なくても問題ないことをコンパイラーに知らせます。この言語フィーチャーは、その関数は引数以外の値を検査しないこと、およびその戻り値のため以外には効力を持たないことを明示的に指定できる機能であり、コンパイラーは、それによって、コードを最適化しやすくなります。

const 関数属性の構文



次の種類の関数には、const を宣言しないでください。

- 指し示されたデータを調べるポインター引数を持つ関数。
- const 関数以外の関数を呼び出す関数。

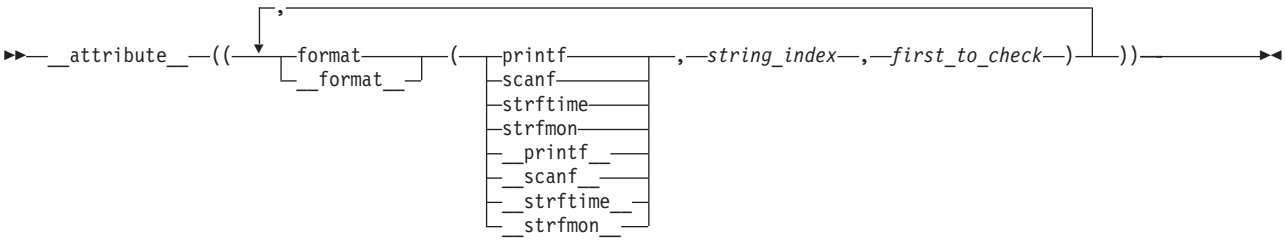
関連資料

「XL C コンパイラー・リファレンス」の『#pragma isolated call』を参照

format 関数属性

format 関数属性を使用することにより、書式制御ストリングを引数として取るユーザー定義関数を識別することができます。これにより、これらの関数に対する呼び出しは、書式制御ストリングと対照して型検査されます。これは、コンパイラーが、関数 printf、scanf、strftime、および strfmon に対する呼び出しをエラーがないか検査する方法に似ています。

format 関数属性の構文



ここで、

string_index
ユーザー関数の宣言内のどの引数が書式制御ストリングであるかを指定する定数整数式です。

first_to_check
これは、書式制御ストリングに照らして検査する最初の引数を指定する定数整数式です。書式制御ストリングに照らして検査すべき引数がない場合 (つまり、書式制御ストリングの構文およびセマンティクスについてのみ診断を行う場合) は、*first_to_check* の値には 0 を指定します。strftime スタイルの書式の場合は、*first_to_check* は 0 でなければなりません。

同じ関数について複数の format 属性を指定できます。その場合はそれらの属性がすべて適用されます。

```
void my_fn(const char* a, const char* b, ...)
       __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

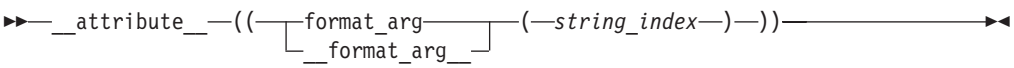
同じストリングを、いくつかの異なる書式スタイルについて診断することもできます。その場合は、すべてのスタイルが診断されます。

```
void my_fn(const char* a, const char* b, ...)
       __attribute__((__format__(__printf__,2,3),
                      __format__(__strftime__,2,0),
                      __format__(__scanf__,2,3)));
```

format_arg 関数属性

format_arg 関数属性は、書式制御ストリングを変更するユーザー定義関数を識別するための手段として使用できます。該当するユーザー定義関数が識別されると、その関数に対する呼び出しをオペランドとして持つ printf、scanf、strftime、または strfmon などの関数について、エラーの有無を検査することができます。

format_arg 関数属性の構文



ここで、*string_index* は、どの引数が書式制御ストリング引数であるかを指定する定数整数式であり、値は 1 から始まります。

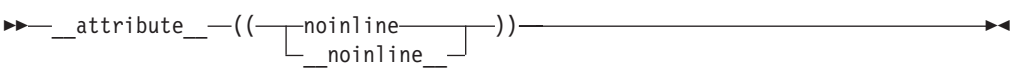
同じ関数について複数の format_arg 属性を指定でき、その場合はそのすべてが適用されます。

```
extern char* my_dgettext(const char* my_format, const char* my_format2)
          __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%","%"));
//printf-style format diagnostics are performed on both "%" strings
```

noinline 関数属性

noinline 関数属性を使用することにより、対象の関数について inline または non-inline のどちらが宣言されているかに関係なく、その関数がインライン化されないようにすることができます。この属性は、インライン化コンパイラー・ オプション、inline キーワード、および always_inline 関数属性より優先されます。

noinline 関数属性の構文



この属性はインライン化を防止するだけのものであり、インライン関数のセマンティクスは除去されません。

関連資料

182 ページの『inline 関数指定子』

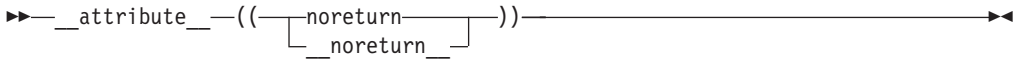
190 ページの『always_inline 関数属性 (IBM 拡張)』

noreturn 関数属性

noreturn 関数属性は、その関数は戻らないことをコンパイラーに示します。この言語フィーチャーは、コンパイラーがコードを最適化しやすくなるように、また初期化されていない変数に対する誤った警告を減らすように、プログラマーが明示的に指定できる機能です。

この関数の戻りの型は void でなければなりません。

noreturn 関数属性の構文



呼び出し元の関数によって保管されたレジスターは、戻らない関数を呼び出す前に必ず復元されるとは限りません。

関連資料

「XL C コンパイラー・リファレンス」の 中の『#pragma_leaves 』 を参照

pure 関数属性

pure 関数属性は、その関数の呼び出し回数が、ソース・コード明記された回数より少なくてもよい関数を宣言します。pure 属性を持つ関数を宣言することは、その関数は、パラメーター、グローバル変数、またはその両方に依存する戻り値以外は効力を持たないことを意味します。

pure 関数属性の構文

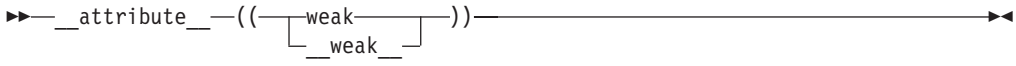
関連資料

「XL C コンパイラー・リファレンス」の『#pragma isolated call』を参照

weak 関数属性

weak 関数属性があると、関数宣言の結果によるシンボルは、オブジェクト・ファイルの中に、グローバル・シンボルとしてではなく、弱いシンボルとして現れます。この言語フィーチャーは、ライブラリー関数を書くプログラマーが、ユーザー ・コード内の変数定義で、重複した名前エラーを起こすことなく、ライブラリー関数宣言をオーバーライドするための機能です。

weak 関数属性の構文



関連資料

105 ページの『weak 変数属性』

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

189 ページの『alias 関数属性』

main() 関数

プログラムが実行を開始すると、システムは main 関数を呼び出します。この関数は、プログラムのエントリー・ポイントを表します。デフォルトでは、main は、ストレージ・クラス extern を持ちます。どのプログラムにも、main という名前の関数が 1 つなければなりません。そして、次の制約が適用されます。

- プログラムにあるそれ以外の関数を main と呼ぶことはできません。
- main は、inline または static と定義することはできません。
- main は、プログラム内から呼び出すことはできません。
- main のアドレスを取得することはできません。
- main 関数は、多重定義できません。

関数 main は、次のいずれかの形式で、パラメーターを指定せずに、またはパラメーターを指定して定義することができます。

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

どのような名前もこれらのパラメーターに付けることはできますが、それらは通常、argc および argv と呼ばれています。最初のパラメーターの argc (引数カウント) は、プログラムが開始された時、コマンド行にいくつの引数が入力されたかを示す整数です。 2 番目のパラメーターの argv (引数ベクトル) は、文字オブジェクトの配列を指すポインターの配列です。配列オブジェクトはヌル終了ストリングであって、プログラムが開始された時、コマンド行に入力された引数を表します。

この配列の最初のエレメント argv[0] は、コマンド行から実行されているプログラムのプログラム名または起動名が入っている文字配列を指すポインターです。argv[1] は、プログラムに渡される最初の引数を示し、argv[2] は 2 番目の引数などとなります。

次のプログラム例 backward は、コマンド行に入力された引数を、最後の引数が最初に印刷されるように印刷します。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
  printf("\n");
}
```

以下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" を指すポインター
argv[3]	NULL

関連資料

45 ページの『extern ストレージ・クラス指定子』

182 ページの『inline 関数指定子』

44 ページの『static ストレージ・クラス指定子』

『関数呼び出し』

関数呼び出し

関数は、宣言され定義されたら、プログラムのどこからでも*呼び出す* ことができます。すなわち、main 関数内や別の関数から、または自分自身からでも呼び出すことができます。関数を呼び出すには、関数名、その後に関数呼び出し演算子、および関数が受け取ることを期待するデータ値を指定します。これらの値は、関数に定義されたパラメーターの*引数* であり、前述のプロセスは、関数に*引数を渡す* と呼ばれています。

引数の受け渡しは、次の 2 とおりの方法で行うことができます。

- 『値による受け渡し』。これは、引数の*値* を、呼び出された関数の対応するパラメーターにコピーします。
- 196 ページの『参照による受け渡し』。これは、引数の*アドレス* を、呼び出された関数の対応するパラメーターに渡します。

関連資料

113 ページの『関数引数の変換』

120 ページの『関数呼び出し式』

194 ページの『main() 関数』

値による受け渡し

値 による受け渡しを使用する場合、コンパイラーは、呼び出し側の関数の引数の値を、呼び出された関数定義の中のポインター以外パラメーターにコピーします。呼び出された関数のパラメーターは、渡された引数の値で初期化されます。そのパラメーターが定数と宣言されていなければ、パラメーターの値は変更できますが、変更は、呼び出された関数のスコープ内でのみ行われます。すなわち、この変更は、呼び出し側の関数の引数の値には、影響しません。

次の例では、main から func に、5 および 7 の 2 つの値が渡されます。関数 func は、これらの値のコピーを受け取り、ID a と b によって、それらにアクセスします。関数 func は、値 a を変更します。 main に制御が戻るときには、x と y の実際の値は、変わっていません。

```
/**
 ** This example illustrates calling a function by value
 **/

#include <stdio.h>

void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d\n", a, b);
}

int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d\n", x, y);
   return 0;
}
```

このプログラムの出力は、次のようになります。

```
In func, a = 12 b = 7
In main, x = 5  y = 7
```

参照による受け渡し

*参照による 受け渡し*とは、呼び出し側の関数の引数のアドレスを、呼び出された関数の対応するパラメーターに渡す方式を指しています。C では、呼び出された関数の対応するパラメーターは、ポインター型と宣言されている必要があります。

このようにして、呼び出す側の関数内の引数の値は、呼び出された関数によって変更することができます。

次の例は、参照によって引数がどのように渡されるかを示しています。関数が呼び出されると、ポインター・パラメーターはポインター値によって初期化されることに注意してください。

C

```
#include <stdio.h>

void swapnum(int *i, int *j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;

  swapnum(&a, &b);
  printf("A is %d and B is %d¥n", a, b);
  return 0;
}
```

C

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
```

関数を指すポインター

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、引数として int を取り、int を指すポインターを戻す関数として解釈されます。2 番目の宣言では、g は、int 引数を取り、int を戻す関数へのポインターと解釈されます。

関連資料

84 ページの『ポインター』

111 ページの『ポインター型変換』

182 ページの『extern ストレージ・クラス指定子』

ネストされた関数 (IBM 拡張)

C ネストされた関数とは、他の関数の定義の中で定義されている関数です。ネストされた関数は、変数宣言を使用できる場所ならどこでも定義することができ、したがって、ネストされた関数の中にさらに関数をネストすることもできます。包含する関数の中で、ネストされた関数は、auto キーワードを使用して定義する前に宣言することができます。そうでない場合は、ネストされた関数は内部リンケージを持つことになります。この言語フィーチャーは C89 および C99 の拡張機能であって、GNU C を使用して開発されたプログラムの移植を容易にするためにインプリメントされています。

ネストされた関数は、その関数の定義より前にある、包含する関数のすべての ID にアクセスできます。

ネストされた関数は、包含する関数の終了後に呼び出すことはできません。

ネストされた関数は、goto ステートメントを使用して、包含する関数内のラベルへ、または包含する関数から継承された __label__ キーワードを宣言されたローカル・ラベルへジャンプすることはできません。 C

関連資料

154 ページの『ローカルに宣言されたラベル (IBM 拡張)』

第 10 章 プリプロセッサー・ディレクティブ

プリプロセッサーは、コンパイルの前にコードを処理するために、コンパイラーによって呼び出されるプログラムです。このプログラムのためのコマンドはディレクティブ と呼ばれ、ソース・ファイル内の、# 文字で始まる行がこれにあたります。この文字により、このようなコマンド行とソース・プログラムのテキストが区別されます。各プリプロセッサー・ディレクティブの効力は、ソース・コード・テキストの変更であり、結果は新しいソース・コード・ファイルになり、そこには、ディレクティブは含まれません。プリプロセスされたソース・コードは中間ファイルですが、これがコンパイラーへの入力になるので、このソース・コードは有効な C プログラムでなければなりません。

プリプロセッサー・ディレクティブには、次のようなものがあります。

- 200 ページの『マクロ定義ディレクティブ』。このディレクティブは、現在のファイル内のトークンを、指定された置換トークンに置き換えます。
- 208 ページの『ファイル・インクルード・ディレクティブ』。このディレクティブは、現在のファイル内にファイルを組み込みます。
- 211 ページの『条件付きコンパイル・ディレクティブ』。このディレクティブは、現在のファイルのセクションを条件付きでコンパイルします。
- 216 ページの『メッセージ生成ディレクティブ』。このディレクティブは、診断メッセージの生成を制御します。
- IBM 218 ページの『アサーション・ディレクティブ (IBM 拡張)』。このディレクティブは、プログラムを実行するシステムの属性を指定します。
- 219 ページの『ヌル・ディレクティブ (#)』。このディレクティブは、何もアクションを行いません。
- 219 ページの『プラグマ・ディレクティブ』。このディレクティブは、コンパイラー固有の規則を、指定されたコードのセクションに適用します。

プリプロセッサー・ディレクティブは、# トークンで始まり、そのあとにプリプロセッサー・キーワードが続きます。# トークンは、空白文字でない行の先頭文字として存在しなければなりません。# はディレクティブ名の一部ではなく、空白文字で名前から分離することができます。

行の最後の文字が ¥ (円記号) 文字でない限り、プリプロセッサー・ディレクティブは改行文字で終了します。¥ 文字がプリプロセッサー行の最後の文字として現れると、プリプロセッサーは ¥ と改行文字を継続マーク文字として解釈します。プリプロセッサーは、¥ (およびそれに続く改行文字) を削除して、物理ソース行を継続する論理行に継ぎます。円記号と、行末文字またはレコードの物理的な終わりとの間に、空白文字があっても構いません。ただし、通常、この空白文字は編集の際には見えません。

一部の `#pragma` ディレクティブを除いて、プリプロセッサー・ディレクティブはプログラム内の任意の場所に入れることができます。

マクロ定義ディレクティブ

マクロ定義ディレクティブには、次のディレクティブと演算子が含まれます。

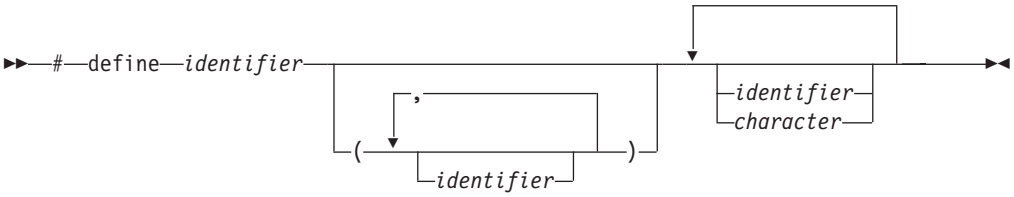
- 『#define ディレクティブ』。マクロを定義します。
- 205 ページの『#undef ディレクティブ』。マクロ定義を除去します。

207 ページの『標準の事前定義マクロ名』で、ISO C 標準によって事前定義されたマクロについて説明しています。 IBM XL C 用に事前定義されているマクロについては、「*XL C コンパイラー・リファレンス*」の『事前定義マクロ』に説明があります。

#define ディレクティブ

プリプロセッサー定義ディレクティブ は、これ以降のマクロの出現を、指定された置換トークンに置き換えるようプリプロセッサーに指示します。

#define ディレクティブの構文



#define ディレクティブは、次のものを指定することができます。

- 『オブジェクト類似マクロ』
- 201 ページの『関数類似マクロ』

以下に、#define と const 型修飾子との違いのいくつかを示します。

- #define ディレクティブは、数値定数、文字定数、およびストリング定数を作成するために使用できますが、const オブジェクトは任意の型を宣言できます。
- const オブジェクトは変数のスコープ規則に従いますが、 #define を使用して作成される定数はそうではありません。
- const オブジェクトと違って、マクロはインラインで展開されるので、マクロの値は、コンパイラーが使用する中間ソース・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。

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、正式なパラメーターの名前をコンマで区切って、括弧で囲んで宣言します。空の正式パラメーター・リストは有効です。そのようなマクロを使用して、引数を取らない関数をシミュレートすることができます。 C99 は、可変個の引数を持つ関数類似マクロのサポートを追加しました。

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。. この場合には、置換リストに ID __VA_ARGS__ を使用できます。

関数類似マクロの呼び出し

小括弧に入れられた、コンマで区切られた引数のリストが後に続く ID。引数の数は、マクロ定義内のパラメーター・リストが省略符号で終了するのでなければ、定義内のパラメーターの数と一致しなければなりません。定義内のパラメーター・リストが省略符号で終了する場合、マクロ呼び出しでの引数の数は、定義内のパラメーターの数を超えるはずです。この超過部分は、*後続引数* と呼ばれます。プリプロセッサーは、関数類似マクロの呼び出しを確認すると、引数の置換を行います。置換コード内のパラメーターは、対

応する引数に置き換えられます。マクロ定義で後続引数が許可されている場合、その引数は間にコンマが挿入されて、その引数全体が単一の引数であるかのように、 ID __VA_ARGS__ を置き換えます。引数自体に含まれるマクロの呼び出しはすべて、引数が置換コード内の対応するパラメーターと置き換わる前に、完全に置き換えられます。

マクロ引数は空でもかまいません (ゼロ個のプリプロセス・トークンで構成されます)。次に例を示します。

```
#define SUM(a,b,c) a + b + c
SUM(1,,3)  /* No error message.
              1 is substituted for a, 3 is substituted for c. */
```

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。

```
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。

以下のプログラム例には、2 つのマクロ定義と、その定義されている両方のマクロを参照するマクロ呼び出しが含まれています。

```
/**
 ** This example illustrates #define directives.
 **/

#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
```

可変数引数マクロ拡張機能 (IBM 拡張)

可変数引数マクロ (variadic macro) 拡張機能は、可変数の引数を持つマクロに関連した、C99 の 2 つの拡張機能です。これらの拡張の 1 つは、変数引数 ID を __VA_ARGS__ からユーザー定義の ID に名前変更するためのメカニズムです。もう 1 つの拡張機能は、変数引数が指定されていない場合に、可変数引数マクロ内のダングリング・コンマを除去するための手段を提供します。どちらの拡張機能も、GNU C を使用して開発されたプログラムの移植を容易にするためにインプリメントされています。

以下の例は、__VA_ARGS__ の代わりに ID を使用する方法を示しています。マクロ debug の最初の定義は、__VA_ARGS__ の通常の使用方法の例を示しています。 2 番目の定義は、__VA_ARGS__ の代わりの ID として args を使用する方法を示しています。

```
#define debug1(format, ...)  printf(format, ## __VA_ARGS__)
#define debug2(format, args ...)  printf(format, ## args)
```

呼び出し	マクロ展開の結果
`debug1("Hello %s/n"."World");`	`printf("Hello %s/n"."World");`
`debug2("Hello %s/n"."World");`	`printf("Hello %s/n"."World");`

プリプロセッサーは、関数マクロへの変数引数が省略されるかまたは空であり、関数マクロ定義内の変数引数 ID の前に ## を伴うコンマがある場合に、末尾のコンマを除去します。

関連資料

75 ページの『const 型修飾子』
149 ページの『演算子優先順位と結合順序』
118 ページの『括弧で囲んだ式 ( )』
206 ページの『## 演算子』

#undef ディレクティブ

プリプロセッサーの undef ディレクティブ により、プリプロセッサーはプリプロセッサー定義のスコープを終わらせます。

#undef ディレクティブの構文

identifier が現在マクロとして定義されていなければ、#undef は無視されます。

以下のディレクティブは BUFFER および SQR を定義します。

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

以下のディレクティブはその定義を無効にします。

```
#undef BUFFER
#undef SQR
```

これらの #undef ディレクティブの後に続いて ID BUFFER および SQR が現れても、それらは、いかなる置換トークンにも置き換えられません。 #undef ディレクティブによってマクロの定義が除去されてしまえば、新しい #define ディレクティブでその ID を使用することができます。

演算子

(単一番号記号) 演算子は、関数類似マクロのパラメーターを文字ストリング・リテラルに変換します。例えば、以下のディレクティブを使用してマクロ ABC が定義される場合、

```
#define ABC(x)   #x
```

これ以降のマクロ ABC の呼び出しはすべて、ABC に渡された引数を含む文字ストリング・リテラルに展開されます。次に例を示します。

呼び出し	マクロ展開の結果
ABC(1)	″1″
ABC(Hello there)	″Hello there″

演算子を、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 2 番目の ¥ 文字が挿入されます。
- マクロに渡された引数に " (二重引用符) 文字がある場合、マクロ展開時に、" の前に ¥ 文字が挿入されます。
- 引数のストリング・リテラルへの変換は、その引数でマクロが展開される前に行われます。
- マクロ定義の置換リスト内に複数の ## 演算子または # 演算子がある場合、その演算子の評価の順序は定義されていません。
- マクロ展開の結果が有効な文字ストリング・リテラルでない場合、その振る舞いは未定義です。

以下の例は、# 演算子の使用法を示したものです。

```
#define STR(x)        #x
#define XSTR(x)       STR(x)
#define ONE           1
```

呼び出し	マクロ展開の結果
STR(¥n "¥n" '¥n')	"¥n ¥"¥¥n¥" '¥¥n'"
STR(ONE)	″ONE″
XSTR(ONE)	″1″
XSTR("hello")	″¥″hello¥″″

関連資料

219 ページの『ヌル・ディレクティブ (#)』

演算子

(二重番号記号) 演算子は、マクロ定義に含まれるマクロの呼び出し (テキストまたは引数、あるいはその両方) における 2 つのトークンを連結します。

以下のディレクティブを使用して、マクロ XY が定義された場合、

```
#define XY(x,y)    x##y
```

x に対する引数の最後のトークンは、y に対する引数の最初のトークンと連結されます。

演算子は、以下の規則に従って使用します。

- ## 演算子を、マクロ定義の置換リスト内の最初の項目または最後の項目にすることはできません。
- ## 演算子の前にある項目の最後のトークンは、 ## 演算子の後ろにある項目の最初のトークンに連結されます。
- 連結は、引数の中のマクロのいずれかが展開される前に行われます。

- 連結の結果が有効なマクロ名になった場合には、たとえその名前が、通常はその中では使用できないコンテキストの中に現れたとしても、その名前を、その後の置換に使用することができます。
- マクロ定義の置換リスト内に複数の ## 演算子、または # 演算子、またはその両方がある場合、その演算子の評価の順序は定義されていません。

以下の例は、## 演算子の使用法を示したものです。

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

呼び出し	マクロ展開の結果
`ArgArg(lady, bug)`	`"ladybug"`
`ArgText(con)`	`"conTEXT"`
`TextArg(book)`	`"TEXTbook"`
`TextText`	`"TEXTtext"`
`ArgArg(Jitter, bug)`	`3`

関連資料

200 ページの『#define ディレクティブ』

標準の事前定義マクロ名

C は、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__ と __LINE__ を除いて、事前定義されたマクロの値は、変換単位全体にわたって定数のままです。

__DATE__

ソース・ファイルがコンパイルされた日付が入っている文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__DATE__ の値が変わります。日付は次の形式です。

```
"Mmm dd yyyy"
```

ここで、

Mmm 月を省略形式 (Jan、Feb、Mar、Apr、May 、Jun、Jul、Aug、Sep、Oct、Nov、または Dec) で表します。

dd 日を表します。日が 10 より小さい場合、最初の d はブランク文字になります。

yyyy 年を表します。

__FILE__

ソース・ファイルの名前が入った文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__FILE__ の値が変わります。これは、#line ディレクティブを使用して設定できます。

__LINE__
現行のソース行番号を表す整数

コンパイラーがソース・プログラムの後続の行を処理すると、コンパイル中に __LINE__ の値が変わります。これは、#line ディレクティブを使用して設定できます。

__STDC__
C の場合、整数 1 であると、C コンパイラーが ISO 規格をサポートすることを示します。言語レベルを **classic** に設定すると、このマクロは定義されません。 (未定義のマクロは、#if ステートメントで使用されると、あたかも整数値 0 を持っているかのように振る舞います。)

__STDC_HOSTED__ (C のみ)
この C99 マクロの値は 1 であって、C コンパイラーはホスト型インプリメンテーションであることを示します。このマクロは、__STDC__ も定義されている場合にのみ定義されることに注意してください。

__STDC_VERSION__ (C のみ)
long int 型の整数定数: C89 言語レベルでは 199409L、C99 言語レベルでは 199901L。このマクロは、__STDC__ も定義されている場合にのみ定義されることに注意してください。

__TIME__
ソース・ファイルがコンパイルされた時刻が入っている文字ストリング・リテラル。

ソース・プログラムの一部であるインクルード・ファイルをコンパイラーが処理すると、__TIME__ の値が変わります。時刻は次の形式です。

```
"hh:mm:ss"
```

ここで、

hh 時間を表します。

mm 分を表します。

ss 秒を表します。

関連資料

217 ページの『#line ディレクティブ』

オブジェクト類似マクロ

ファイル・インクルード・ディレクティブ

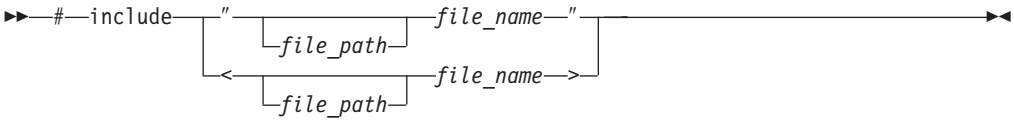
ファイル・インクルード・ディレクティブは次のもので構成されます。

- 209 ページの『#include ディレクティブ』。別のソース・ファイルからテキストを挿入します。
- IBM 210 ページの『#include_next ディレクティブ (IBM 拡張)』。コンパイラーがインクルード・ファイルの検索時に、検索パスからインクルード・ファイルのディレクトリーを除外するようにします。

#include ディレクティブ

プリプロセッサーの *include* ディレクティブ を使用すると、プリプロセッサーは、そのディレクティブを指定されたファイルの内容に置き換えます。

#include ディレクティブの構文



例えば次のように、*file_name* が二重引用符で囲まれている場合、

```
#include "payroll.h"
```

ユーザー定義ファイルとして扱われ、ヘッダー・ファイルまたはソース・ファイルを表す場合があります。

例えば次のように、*file_name* が不等号括弧で囲まれている場合、

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

システム定義のファイルとして扱われ、必ずヘッダー・ファイルを表します。

改行および > の文字は、 < と > で区切られたファイル名の中に入れることができません。改行および " (二重引用符) の文字は、 " と " で区切られたファイル名の中に入れることはできません。 > は入れることができます。

file_path は、絶対パスでも相対パスでもかまいません。二重引用符を使用する場合、*file_path* が相対パスであるかまたは指定されていなければ、プリプロセッサーはインクルードするファイルのディレクトリーを、インクルードされるファイルを検索するためのパスのリストに追加します。不等号括弧を使用する場合、*file_path* が相対パスであるか指定されていなければ、プリプロセッサーはインクルードするファイルのディレクトリーを、インクルードされるファイルを検索するためのパスのリストに追加しません。

プリプロセッサーは #include ディレクティブに含まれているマクロを解決します。マクロ置き換え後に結果として得られるトークン・シーケンスは、二重引用符または文字< および > で囲まれたファイル名で構成されます。次に例を示します。

```
#define MONTH <july.h>
#include MONTH
```

いくつかのファイルによって使用される宣言を 1 つのファイルの中に入れ、#include を用いて、それらを使用する各ファイルに含めることができます。例えば、以下のファイル defs.h には、いくつかの定義と、宣言の追加ファイルのインクルードが 1 つ入っています。

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
```

```
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"
```

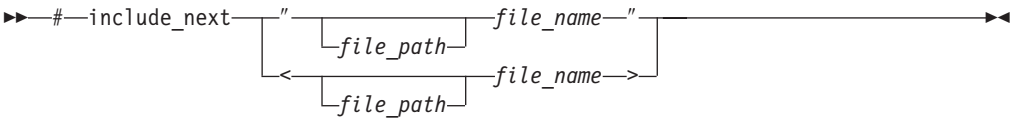
以下の例では、#define は、いくつかのプリプロセッサー・マクロを結合して C 標準入出力ヘッダー・ファイルを表すマクロを定義しています。#include により、ヘッダー・ファイルをプログラムで使用できるようになります。

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 *   #include <stdio.h>
 */

#include C_IO_HEADER
```

#include_next ディレクティブ (IBM 拡張)

プリプロセッサー・ディレクティブ #include_next は #include ディレクティブのように振る舞いますが、インクルードするファイルのディレクトリーを、指定されたファイルを検索するためのリストから、特に除外するところが異なります。インクルードするファイルのディレクトリー以下にあるすべての検索パスは、インクルードされるファイルを検索するためのパスのリストから除外されます。これにより、ファイルの複数バージョンを、同じ名前でアプリケーションの異なる部分に組み込むことができます。または、1 つのヘッダー・ファイルを別のヘッダー・ファイルに、同じ名前 (それ自身を再起的に組み込むヘッダーは付いていない) で組み込むことができます。異なるファイル・バージョンが異なるディレクトリーに保管される場合に、このディレクティブで、ファイル名を指定するために絶対パスを使用する必要なく、ファイルの各バージョンにアクセスすることができるようになります。

#include_next ディレクティブの構文



このディレクティブはヘッダー・ファイルのみで使用する必要があり、*file_name* によって指定されたファイルがヘッダー・ファイルでなければなりません。ファイル名を囲むために使用する、二重引用符と不等号括弧との間に違いはありません。

検索パスが #include_next ディレクティブでどのように解決されるかという例として、ファイル t.h の 2 つのバージョンがあると想定します。最初のファイルは、ソース・ファイル t.c に組み込まれ、サブディレクトリー path1 にあります。2 番目のファイルは最初のファイルに組み込まれ、サブディレクトリー path2 にあります。 t.c のコンパイル時に、両方のディレクトリーがインクルード・ファイル検索パスとして指定されます。

```
/* t.c  */

#include "t.h"

int main()
{
printf(", ret_val);
}

/* t.h in path1 */

#include_next "t.h"

int ret_val = RET;

/* t.h in path2 */

#define RET 55;
```

#include_next ディレクティブは、プリプロセッサーに、 path1 ディレクトリーをスキップし、path2 ディレクトリーからインクルード・ファイルの検索を開始するよう指示します。このディレクティブにより、t.h の異なる 2 つのバージョンを使用でき、この結果 t.h が繰り返しインクルードされることがなくなります。

条件付きコンパイル・ディレクティブ

プリプロセッサーの条件付きコンパイル・ディレクティブ を使用すると、プリプロセッサーは、ソース・コードのコンパイルの一部を条件付きで抑止します。これらのディレクティブは、定数式または ID をテストして、プリプロセッサーがコンパイラーに渡すべきトークン、およびプリプロセス時に迂回すべきトークンを判別します。この条件付きディレクティブには、次のものがあります。

- 212 ページの『#if および #elif ディレクティブ』。定数式の結果に基づいて、ソース・コードの部分を条件により組み込むか、抑止します。
- 213 ページの『#ifdef ディレクティブ』。マクロ名が定義されている場合に、ソース・テキストを条件により組み込みます。
- 214 ページの『#ifndef ディレクティブ』。マクロ名が定義されない場合に、ソース・テキストを条件により組み込みます。
- 214 ページの『#else ディレクティブ』。直前の #if、#ifdef、#ifndef、または #elif の検査が失敗した場合に、条件によりソース・テキストを組み込みます。
- 215 ページの『#endif ディレクティブ』。条件付きテキストを終了します。

プリプロセッサーの条件付きコンパイル・ディレクティブは、下記のいくつかの行に及びます。

- 条件指定行 (#if、#ifdef、または #ifndef で始まる)
- 条件の評価が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。

関連資料

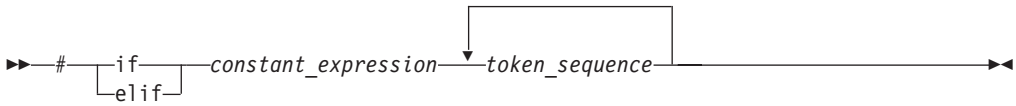
216 ページの『メッセージ生成ディレクティブ』

218 ページの『アサーション・ディレクティブ (IBM 拡張)』

#if および #elif ディレクティブ

#if および #elif ディレクティブは、 *constant_expression* の値をゼロと比較します。

#if および #elif ディレクティブの構文



定数式が非ゼロ値に評価される場合、条件の直後にあるコードの行をコンパイラーに渡します。

式がゼロに評価され、条件付きコンパイル・ディレクティブが、プリプロセッサー #elif ディレクティブを含んでいる場合、#elif および次の #elif またはプリプロセッサー #else ディレクティブとの間にあるソース・テキストが、プリプロセッサーによって選択され、コンパイラーに渡されます。#elif ディレクティブをプリプロセッサーの #else ディレクティブの後ろに入れることはできません。

すべてのマクロが展開され、defined() の式はすべて処理され、残りのすべての ID は、トークン 0 に置き換えられます。

テストされる *constant_expression* は、以下の属性を持つ整定数式でなければなりません。

- キャストは実行されません。
- long int 値を使用して演算を実行します。
- 定義済みマクロを *constant_expression* に入れることはできます。それ以外の ID は式の中には入れられません。
- *constant_expression* には、単項演算子 defined を入れることができます。この演算子は、プリプロセッサー・キーワードの #if または #elif を用いた場合にのみ使用できます。以下の式の評価は、プリプロセッサーに *identifier (ID)* が定義されている場合は、1 に、それ以外の場合は、0 になります。

  ```
  defined identifier
  defined(identifier)
  ```

  次に例を示します。

  ```
  #if defined(TEST1) || defined(TEST2)
  ```

注: マクロが定義されていない場合、0 (ゼロ) の値がそれに代入されます。以下の例では、TEST がマクロ ID であることが必要です。

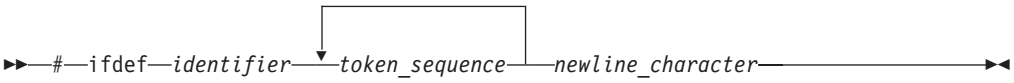
```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds ¥n");
#endif
```

#ifdef ディレクティブ

#ifdef ディレクティブは、マクロ定義の存在を検査します。

指定された ID がマクロとして定義されている場合、条件の直後にあるコードの行がコンパイラーに渡されます。

#ifdef ディレクティブの構文



以下の例は、プリプロセッサーに対して EXTENDED が定義されている場合に、MAX_LEN を 75 として定義します。定義されていない場合には、MAX_LEN を 50 として定義します。

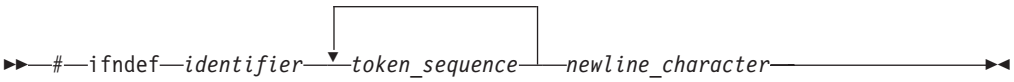
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef ディレクティブ

#ifndef ディレクティブは、マクロが定義されていないかどうかをチェックします。

指定された ID がマクロとして定義されていない場合、条件の直後にあるコードの行がコンパイラーに渡されます。

#ifndef ディレクティブの構文



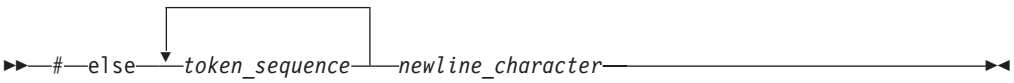
ID は、#ifndef キーワードの後に続いていなければなりません。以下の例は、プリプロセッサーに対して EXTENDED が定義されていない場合に、MAX_LEN を 50 として定義します。定義されていない場合、MAX_LEN を 75 として定義します。

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

#else ディレクティブ

#if、#ifdef、または #ifndef ディレクティブで指定された条件が 0 に評価され、条件付きコンパイル・ディレクティブが、プリプロセッサー #else ディレクティブを含んでいる場合、プリプロセッサー #else ディレクティブおよびプリプロセッサー #endif ディレクティブとの間にあるコードの行が、プリプロセッサーによって選択され、コンパイラーに渡されます。

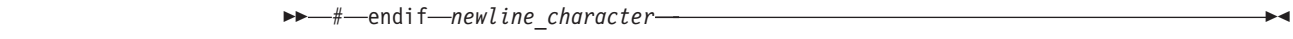
#else ディレクティブの構文



#endif ディレクティブ

プリプロセッサー・ディレクティブの #endif は、条件付きコンパイル・ディレクティブを終了します。

#endif ディレクティブの構文



条件付きコンパイル・ディレクティブの例

以下の例は、プリプロセッサーの条件付きコンパイル・ディレクティブをどのようにネストできるかを示しています。

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

以下のプログラムには、プリプロセッサーの条件付きコンパイル・ディレクティブが含まれています。

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n",
   array[i]);
#endif

   }
   return(0);
}
```

メッセージ生成ディレクティブ

メッセージ生成ディレクティブには、次のようなものがあります。

- 『#error ディレクティブ』。コンパイル時のエラー・メッセージ用のテキストを定義します。
- 『#warning ディレクティブ (IBM 拡張)』。コンパイル時の警告メッセージ用のテキストを定義します。
- 217 ページの『#line ディレクティブ』。コンパイラー・メッセージの行番号を提示します。

関連資料

211 ページの『条件付きコンパイル・ディレクティブ』

#error ディレクティブ

プリプロセッサーの error ディレクティブ を使用すると、プリプロセッサーはエラー・メッセージを生成して、コンパイルを失敗させます。

#error ディレクティブの構文

#error ディレクティブは、コンパイル時の安全チェックとして、#if-#elif-#else 構造の #else 部でよく使用されます。例えば、#error ディレクティブをソース・ファイルで使用すると、プログラムのバイパスすべき部分に到達したら、コードが生成されないようにすることできます。

例えば、以下のディレクティブ

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

は、次のエラー・メッセージを生成します。

```
BUFFER_SIZE is too small.
```

#warning ディレクティブ (IBM 拡張)

プリプロセッサーの warning ディレクティブ を使用すると、プリプロセッサーは警告メッセージを生成します。ただし、コンパイルは続行します。#warning に対する引数は、マクロ展開されません。

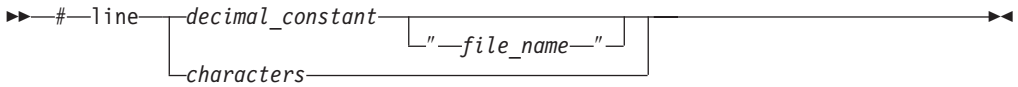
#warning ディレクティブの構文

プリプロセッサー #warning ディレクティブは、GNU C で開発されたプログラムの処理を容易にするために提供される言語拡張機能です。IBM インプリメンテーションでは、複数の空白文字が保持されています。

#line ディレクティブ

プリプロセッサーの行制御ディレクティブ は、コンパイラー・メッセージに対して行番号を提供します。このディレクティブにより、コンパイラーは、次のソース行の行番号を指定された番号として表示します。

#line ディレクティブの構文



コンパイラーがプリプロセスされたソース内の行番号への参照をわかりやすく行えるようにするため、プリプロセッサーは必要な個所に (例えば、含まれているテキストの始めまたはテキストの終わりの後に)、#line ディレクティブを挿入します。

二重引用符で囲まれたファイル名の指定を行番号の後に続けることができます。ファイル名を指定すると、コンパイラーは指定されたファイルの一部として次の行を表示します。ファイル名を指定しないと、コンパイラーは現行ソース・ファイルの一部として次の行を表示します。

C99 言語レベルでは、#line プリプロセッシング・ディレクティブの最大値は 2147483647 です。

すべての C インプリメンテーションにおいて、#line ディレクティブのトークン・シーケンスは、マクロ置き換えをすることがあります。マクロ置き換え後に結果として得られる文字シーケンスは、10 進定数 (オプションで、二重引用符で囲まれたファイル名が後に続く) で構成されます。

#line 制御ディレクティブを使用して、コンパイラーにもっと分かりやすいエラー・メッセージを提供させることができます。以下のプログラム例は、#line 制御ディレクティブを使用して、認識しやすい行番号を各関数に提供します。

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",_ _LINE_ _);
}
```

```
#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d\n",_ _LINE_ _);
}
```

このプログラムの出力は次のようになります。

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

関連資料


「XL C コンパイラー・リファレンス」の『__C99_MAX_LINE_NUMBER』を参照

207 ページの『標準の事前定義マクロ名』

アサーション・ディレクティブ (IBM 拡張)

アサーション・ディレクティブは、コンパイルされたプログラムが実行されるコンピューターまたはシステムを定義するために使用される、マクロ定義の代わりのものです。アサーションは、通常、事前定義されていますが、#assert プリプロセッサー・ディレクティブで定義することもできます。

#assert ディレクティブの構文

predicate は、定義しようとしているアサーション・エンティティーを表します。*answer* は、このアサーションに割り当てる値を表します。同じ述部 (predicate) と異なる応答 (answer) を使用して、複数のアサーションを作成することができます。特定の述部に対するすべての応答は、同時に真です。例えば、次のディレクティブは、フォントのプロパティーに関するアサーションを作成します。

```
#assert font(arial)
#assert font(blue)
```

アサーションが定義されると、現在のシステムをテストするために、そのアサーション述部を条件付きディレクティブの中で使用することができます。以下のディレクティブは、font について、arial が assert されるか、blue が assert されるかをテストします。

```
#if #font(arial) || #font(blue)
```

条件付きディレクティブの中で応答を省略することにより、述部について、いずれかの応答が assert されるかをテストすることができます。

```
#if #font
```

アサーションは、#unassert ディレクティブで取り消すことができます。#assert ディレクティブと同じ構文を使用すると、そのディレクティブは、指定された応答のみを取り消します。例えば、次のディレクティブは、font 述部に対して arial 応答を取り消します。

```
#unassert font(arial)
```

#unassert ディレクティブから応答を省略すると、述部全体が取り消されます。次のディレクティブは、 font 述部を完全に取り消します。

```
#unassert font
```

関連資料

211 ページの『条件付きコンパイル・ディレクティブ』

事前定義されたアサーション

AIX プラットフォームに対しては、以下のアサーションが事前定義されています。

表 30. AIX 用に事前定義されたアサーション

#machine	#system(unix) #system(aix)	#cpu

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
```

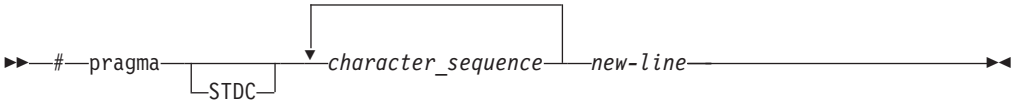
関連資料

205 ページの『# 演算子』

プラグマ・ディレクティブ

プラグマ は、コンパイラーに対するインプリメンテーション定義の命令です。一般的な形式は次のとおりです。

#pragma ディレクティブの構文



character_sequence は、文字が書かれている場合、特定のコンパイラー命令と引数を指定する一連の文字です。トークン STDC は、標準プラグマを示しています。したがって、このディレクティブに対してはマクロ置換は行われません。 *new-line* 文字は、プラグマ・ディレクティブを終了させなければなりません。

プラグマの *character_sequence* は、マクロ置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
```

注: _Pragma 演算子の構文を使用してプラグマ・ディレクティブを指定することもできます。詳しくは、『_Pragma プリプロセッシング演算子』を参照してください。

1 つのプラグマ・ディレクティブで、複数のプラグマ構成を指定することができます。コンパイラーは、認識されないプラグマを無視します。

標準 C プラグマを 『標準のプラグマ』で説明しています。 IBM XL C で使用可能なプラグマについては、「*XL C コンパイラー・リファレンス*」の『汎用プラグマ』に説明があります。

_Pragma プリプロセッシング演算子

単項演算子 _Pragma (C99 の機能) を使用すると、プラグマ・ディレクティブにプリプロセッサー・マクロを含むことができます。

_Pragma 演算子の構文

string_literal には接頭部 L を付けることができ、そうすると、それはワイド・ストリング・リテラルになります。

ストリング・リテラルは、ストリングが解除され、トークンになります。トークンの結果のシーケンスは、プラグマ・ディレクティブに現れたかのように処理されます。次に例を示します。

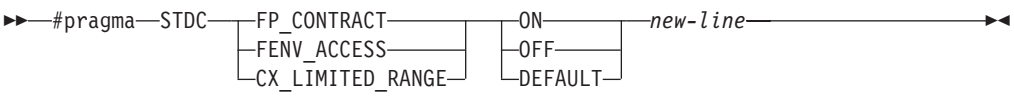
```
_Pragma ( "pack(full)" )
```

は、以下と同等です。

```
#pragma pack(full)
```

標準のプラグマ

標準のプラグマ は、C 標準が構文およびセマンティクスを定義し、マクロ置換が行われないプラグマ・プリプロセッサー・ディレクティブです。標準のプラグマは、以下のいずれかでなければなりません。



FP_CONTRACT および FENV_ACCESS プラグマは認識され、無視されます。

CX_LIMITED_RANGE については以下で説明します。

pragma STDC CX_LIMITED_RANGE

複素数の乗算、割り算、および絶対値用の通常の数学公式は、無限大の処理および不適切なオーバーフローやアンダーフローのため、問題があります。通常の公式は以下のとおりです。

$$(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$$

$$(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$$

$$| x + iy | = \mathrm{sqrt}(x^2 + y^2)$$

デフォルトで、コンパイラーはこれらの計算を実行するために、少々複雑ですが数学的にはより安全なアルゴリズムを使用します。通常の数学公式が安全だと判断する場合は、 STDC CX_LIMITED_RANGE プラグマを使用して、状態が「on」のときにこれらの公式が許容できるものであるとコンパイラーに対して知らせることができます。そうすることによって、コンパイラーは、これらの計算のための高速なコードを生成することができます。状態が「off」の場合は、コンパイラーは引き続き、より安全なアルゴリズムを使用します。このプラグマのインプリメンテーションについて詳しくは、「*XL C コンパイラー・リファレンス*」内の**#pragma STDC cx_limited_range**を参照してください。

第 11 章 IBM XL C 言語拡張機能

IBM XL C 拡張機能には、C89 に対する拡張機能としての C フィーチャー、Unicode サポート、GNU C 互換性、ベクトル処理サポート、および 10 進浮動小数点サポートがあります。

C89 の拡張機能としての C99 フィーチャー

次のいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stcd99 | extc89 | extended** オプション

これらのオプションについて詳しくは、「*XL C コンパイラー・リファレンス*」の『**-qlanglvl**』オプションを参照してください。

表 31. C89 への拡張機能としてのデフォルト C99 機能

言語の機能	参照先
16 進浮動小数点定数	16 進浮動小数点リテラル
__func__ 事前定義 ID	14 ページの『__func__ 事前定義 ID』
ワイド文字ストリングおよび非ワイド文字ストリングの連結	ストリング連結
混合宣言およびコード	41 ページの『データ宣言とデータ定義の概要』
long long データ型	50 ページの『整数型』
複素数データ型	53 ページの『複素数浮動小数点型 』
_Bool データ型	51 ページの『ブール型』
enum 宣言内で許可される末尾コンマ	67 ページの『列挙型の定義』
重複型修飾子	71 ページの『型修飾子』
可変長配列	89 ページの『可変長配列』
左辺値以外の配列添え字	141 ページの『配列添え字演算子 [ ]』
構造体または共用体の最後にある柔軟な配列メンバー	柔軟な配列メンバー
構造体または共用体の初期化指定子の非定数式	95 ページの『構造体および共用体の初期化』
指定された初期化指定子	92 ページの『集合体型に対する、指定された初期化指定子 (C のみ)』
暗黙的関数宣言の除去	178 ページの『関数宣言』
関数宣言内の暗黙的 int 戻りの型の除去	184 ページの『関数の戻りの型指定子』
関数仮パラメーターとしての静的配列	187 ページの『関数仮パラメーター宣言の中の静的配列指標 (C のみ)』
関数類似マクロの変数引数	201 ページの『関数類似マクロ』

表 31. C89 への拡張機能としてのデフォルト C99 機能 (続き)

言語の機能	参照先
関数類似マクロの空の引数	201 ページの『関数類似マクロ』
追加の事前定義マクロ名	207 ページの『標準の事前定義マクロ名』
複合リテラル	148 ページの『複合リテラル式』
_Pragma 演算子	220 ページの『_Pragma プリプロセッシング演算子』
標準プラグマ	220 ページの『標準のプラグマ』
#line ディレクティブの新規制限	217 ページの『#line ディレクティブ』

次のいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stcd99 | extc89 | extended** オプション

また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 32. C89 への拡張機能としてのデフォルト C99 機能 (個々のオプション制御付き)

言語の機能	参照先	個々のオプション制御
2 文字表記	34 ページの『2 文字表記文字』	**-q[no]digraph**
C++ スタイル・コメント	35 ページの『コメント』	**-q[no]cpluscmt**
inline 関数指定子	182 ページの『inline 関数指定子』	**-qkeyword=inline**

次のいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stcd99** オプション

言語の機能	参照先
サフィックスなしの long long 整数リテラル	10 進整数リテラル

次のいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **c99** 呼び出しコマンド
- **-qlanglvl=extc99 | stcd99** オプション

また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 33. C89 への拡張機能としての厳密な C99 機能 (個々のオプション制御付き)

言語の機能	参照先	個々のオプション制御
ユニバーサル文字名	32 ページの『ユニコード規格』	**-qlanglvl=[no]ucs**
`restrict` 型修飾子	75 ページの『restrict 型修飾子』	**-qkeyword=restrict**

関連資料

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

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

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

「XL C コンパイラー・リファレンス」の『コンパイラーの呼び出し』を参照

Unicode の拡張機能サポート

次の機能では、追加のオプションを使用してコンパイルする必要があります。

言語の機能	参照先	必須コンパイル・オプション
UTF-16、UTF-32 リテラル	33 ページの『UTF リテラル (IBM 拡張)』	**-qutf**

関連資料

「XL C コンパイラー・リファレンス」の 中の『 -qutf』 を参照

GNU C 互換性の拡張機能

以下の機能は、デフォルトですべての言語レベルで使用可能です。

表 34. GNU C との互換性のためのデフォルト IBMXL C 拡張機能

言語の機能	参照先
`#include_next` プリプロセッサー・ディレクティブ	210 ページの『#include_next ディレクティブ (IBM 拡張)』

次のいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **-qlanglvl=extc99 | extc89 | extended** オプション

表 35. GNU C との互換性のためのデフォルト IBMXL C 拡張機能

言語の機能	参照先
代替キーワード	12 ページの『言語拡張のキーワード (IBM 拡張)』

表 35. GNU C との互換性のためのデフォルト IBMXL C 拡張機能 (続き)

言語の機能	参照先
__extension__ キーワード	12 ページの『言語拡張のキーワード (IBM 拡張)』
asm ラベル	14 ページの『アセンブリー・ラベル (IBM 拡張)』
複素数リテラル接尾部	複素数リテラル
グローバル・レジスター変数	47 ページの『指定したレジスターの変数 (IBM 拡張)』
構造体または共用体の任意の場所への柔軟な配列メンバーの配置	柔軟な配列メンバー
集合体の柔軟な配列メンバーの静的初期化	柔軟な配列メンバー
ゼロ・エクステント配列	ゼロ・エクステント配列メンバー (IBM 拡張)
型属性	77 ページの『型属性 (IBM 拡張)』
変数属性	101 ページの『変数属性 (IBM 拡張)』
ローカルに宣言されたラベル	154 ページの『ローカルに宣言されたラベル (IBM 拡張)』
値としてのラベル	154 ページの『値としてのラベル (IBM 拡張)』
__alignof__ 演算子	126 ページの『__alignof__ 演算子 (IBM 拡張)』
__typeof__ 演算子	128 ページの『typeof 演算子 (IBM 拡張)』
生成された左辺値	115 ページの『左辺値と右辺値』
単項演算子への複素数型引数	121 ページの『単項式』
複合リテラルによる静的変数の初期化	148 ページの『複合リテラル式』
__imag__ および __real__ 複素数型演算子	129 ページの『__real__ および __imag__ 演算子 (IBM 拡張)』
共用体型へのキャスト	146 ページの『共用体型へのキャスト (C のみ) (IBM 拡張)』
計算後の goto ステートメント	171 ページの『計算後の goto ステートメント (IBM 拡張)』
式内のステートメントおよび宣言	157 ページの『ステートメント式 (IBM 拡張)』
関数の属性	188 ページの『関数属性 (IBM 拡張)』
__inline__ 関数指定子	182 ページの『inline 関数指定子』
ネストされた関数	197 ページの『ネストされた関数 (IBM 拡張)』
可変数引数マクロ拡張機能	可変数引数マクロ拡張機能 (IBM 拡張)
#warning プリプロセッサー・ディレクティブ	216 ページの『#warning ディレクティブ (IBM 拡張)』
#assert、#unassert プリプロセッサー・ディレクティブ	218 ページの『アサーション・ディレクティブ (IBM 拡張)』

次のいずれかを指定してコンパイルする場合、以下の機能がデフォルトで使用可能です。

- **xlc** 呼び出しコマンド
- **-qlanglvl=extc99 | extc89 | extended** オプション

また、下記の表にリストされている特定のコンパイラー・オプションによって使用可能または使用不可になります。

表 36. GNU C との互換性のための IBM XL C 拡張機能 (個々のオプション制御付き)

言語の機能	参照先	個々のオプション制御
asm、および __asm キーワード	14 ページの『アセンブリー・ラベル (IBM 拡張)』, 172 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』	**-qkeyword=asm**, **-qasm**
asm インライン・アセンブリー言語ステートメント	172 ページの『インライン・アセンブリー・ステートメント (IBM 拡張)』	**-qasm**

次の機能では、追加のオプションを使用してコンパイルする必要があります。

表 37. GNU C との互換性のための IBM XL C 拡張機能 (追加のコンパイラー・オプション要)

言語の機能	参照先	必須コンパイル・オプション
ID 内のドル記号	13 ページの『ID の文字』	**-qdollar**
typeof キーワード	128 ページの『typeof 演算子 (IBM 拡張)』	**-qkeyword=typeof**
__thread ストレージ・クラス指定子	48 ページの『__thread ストレージ・クラス指定子 (IBM 拡張)』	**-qtls**

関連資料

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

「XL C コンパイラー・リファレンス」の 中の『 -qasm』を参照

「XL C コンパイラー・リファレンス」の 中の『 -qtls』 を参照

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

「XL C コンパイラー・リファレンス」の『コンパイラーの呼び出し』を参照

ベクトル処理の拡張機能サポート

ベクトル拡張機能は、以下の条件のすべてが満たされたときのみ受け入れられます。

1. **-qarch** オプションが、ベクトル処理命令をサポートするターゲット・アーキテクチャーに設定される。
2. **-qenablevmx** オプションが有効なとき。
3. **-qaltivec** オプションが有効なとき。

これらのオプションについて詳しくは、「*XL C コンパイラー・リファレンス*」を参照してください。

表 38. AltiVec Application Programming Interface 仕様 をサポートするための IBM XL C 拡張機能

言語の機能	参照先
ベクトル・プログラミング言語の拡張機能	55 ページの『ベクトル型 (IBM 拡張)』, 23 ページの『ベクトル・リテラル (IBM 拡張)』

以下の機能は、AltiVec Application Programming Interface 仕様への IBM 拡張機能です。

表 39. AltiVec Application Programming Interface 仕様に対する IBM XL C 拡張機能

言語拡張	参照先
ベクトル定数の初期化指定子リスト	94 ページの『ベクトルの初期化 (IBM 拡張)』
ベクトル型の `typedef` 定義	70 ページの『typedef 定義』
静的ベクトル変数の初期化指定子としての複合リテラル	148 ページの『複合リテラル式』
`__alignof__` および `typeof` 演算子への引数としてのベクトル型	126 ページの『__alignof__ 演算子 (IBM 拡張)』, 128 ページの『typeof 演算子 (IBM 拡張)』

10 進浮動小数点サポートの拡張機能

次の機能では、追加のオプションを使用してコンパイルする必要があります。

言語の機能	参照先	必須コンパイル・オプション
10 進浮動小数点型	52 ページの『実浮動小数点型』、10 進浮動小数点リテラル (IBM 拡張)、 109 ページの『浮動小数点変換』	**-qdfp**

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711
東京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、

利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、 IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

商標

IBM、IBM ロゴ、および ibm.com は、International Business Machines Corporation の米国およびその他の国における商標です。この資料が公開された時点で、米国において IBM が所有する登録商標または商標には、初出時に商標記号 (® または ™) を付けて示しています。このような商標は、その他の国においても、登録商標または商標である可能性があります。現時点での IBM の商標については、http://www.ibm.com/legal/copytrade.shtml の「Copyright and trademark information」をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft および Windows は、Microsoft Corporation の米国およびその他の国における商標です。

Cell Broadband Engine, Cell/B.E は、米国およびその他の国における Sony Computer Entertainment, Inc. の商標であり、同社の許諾を受けて使用しています。

UNIX は The Open Group の米国およびその他の国における登録商標です。

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。

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

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

[特殊文字]

プログラム番号: 5724-U80

Printed in Japan

SC88-4956-00

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12