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

最適化およびプログラミング・ガイド

バージョン 13.1

SA88-5402-00
(英文原典：SC27-4251-00)

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

最適化およびプログラミング・ガイド

バージョン 13.1

SA88-5402-00
(英文原典：SC27-4251-00)

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、 147 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM XL C/C++ for Linux, V13.1 (プログラム 5765-J08; 5725-C73)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。正しいレベルの製品をご使用になるようお確かめください。

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

原典： SC27-4251-00
IBM XL C/C++ for Linux, V13.1
Optimization and Programming Guide
Version 13.1

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

© Copyright IBM Corporation 1996, 2014.

目次

© Copyright IBM Corp. 1996, 2014

本書について

このガイドは、IBM® XL C/C++ for Linux, V13.1 コンパイラーの使用法に関する上級者向けのトピックを、特にプログラムの移植性と最適化に重点を置いて説明したものです。このガイドは、お勧めするプログラミング手法とコンパイル・プロシージャーを用いて、コンパイラーの能力を最大に引き出すための、参照情報と実用的ヒントの両方を提供しています。

本書の対象読者

本書は複雑なアプリケーションを構築するプログラマーにご利用頂くことを意図したもので、すでに XL C/C++ を使用したコンパイルの経験をお持ちであり、プログラムの最適化やチューニング、拡張プログラミング言語フィーチャー、およびアドオン・ツールやユーティリティーなどのサポートを含むコンパイラー機能のより高度な利点を活用する方々のためのものです。

本書の読み方

本書では各トピックの説明のために「タスク指向」アプローチを採用して、各章ごとに特定のプログラミングやコンパイルの問題に焦点を集中しています。各トピックにはまた IBM XL C/C++ for Linux, V13.1 文書セット内の解説書にある関連セクションに対する豊富な相互参照が含まれていて、コンパイラー・オプションやプラグマ、そして特定の言語拡張についての説明が提供されています。

本書の構成

このガイドは下記のトピックが記載されています。

- 1 ページの『第 1 章 32 ビット・モードおよび 64 ビット・モードの使用』 は、既存の 32 ビットのアプリケーションを 64 ビット・モードに移植するときに発生する共通問題について説明し、その問題を避けるための勧告を提供しています。
- 7 ページの『第 2 章 XL C/C++ の Fortran での使用』 では XL C/C++ プログラムから FORTRAN コードを呼び出す際の考慮事項について説明しています。
- 13 ページの『第 3 章 データの位置合わせ』 は、すべてのプラットフォーム上の構造体 およびクラスのような、集合体内におけるデータの位置合わせのコントロールに使用できる別のコンパイラー・オプションについて説明しています。
- 21 ページの『第 4 章 浮動小数点演算の処理』 は、コンパイラーが取り扱う浮動小数点操作の方法のコントロールに使用可能なオプションを説明します。
- 27 ページの『第 5 章 C++ コンストラクターの使用』 は、共通の初期化を 1 つのコンストラクターに集中させることが可能な委任コンストラクターについて説明します。
- 29 ページの『第 6 章 C++ テンプレートの使用』 は、C++ テンプレートを組み込むコンパイル・プログラムの別のオプションについて説明します。

© Copyright IBM Corp. 1996, 2014

- 37 ページの『第 7 章 ライブラリーの構成』 は、静的および共用ライブラリーのコンパイルおよびリンクの方法、そして C++ プログラム内で静的オブジェクトの初期化順序を指定する方法について説明します。
- 45 ページの『第 8 章 アプリケーションの最適化』 は、ユーザーのプログラムの最適化、そして別のオプションを使用するための推奨として、コンパイラーが提供する種々のオプションについて説明します。
- 83 ページの『第 9 章 最適化コードのデバッグ』 は、最適化されたプログラムの潜在的なユーザビリティーの問題と、最適化されたコードをデバッグする際に使用できるオプションについて説明します。
- 89 ページの『第 10 章 パフォーマンスを向上させるためのアプリケーションのコーディング』 は、プログラム・パフォーマンスとコンパイラーの最適化能力の互換性を向上させるために、推奨するプログラミング手法およびコーディング・テクニックについて説明します。
- 119 ページの『第 11 章 ハイパフォーマンス・ライブラリーの使用』 は、XL C/C++ と一緒に出荷される 2 つのパフォーマンス・ライブラリーについて説明します。1 つは Mathematical Acceleration Subsystem (MASS) で、これには標準数学ライブラリー関数の調整済みバージョンが含まれています。もう 1 つは Basic Linear Algebra Subprograms (BLAS) で、これには行列乗算用の基本関数が含まれています。
- 137 ページの『第 12 章 プログラムの並列処理』 は、XL C/C++ が提供する、マルチスレッド・プログラム (OpenMP 言語構造体を含む) 作成用の各種オプションについて概説します。

規則

活字の規則

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

表 1. 活字の規則

書体	意味	例
太字	小文字のコマンド、実行可能ファイル名、コンパイラー・オプション、およびディレクティブ。	コンパイラーには、さまざまな C/C++ 言語レベルおよびコンパイル環境をサポートするために、**xlc** と **xlC** (**xlc++**) という基本呼び出しコマンドとその他のいくつかのコンパイラー呼び出しコマンドが備わっています。
イタリック	パラメーターまたは変数。実際の名前と値はユーザーによって提供されます。イタリックは新規用語の導入にも使用されます。	要求された *size* よりも大きいものを戻す場合には、 *size* パラメーターの更新を確認してください。
下線	コンパイラー・オプションまたはディレクティブのパラメーターのデフォルト設定。	nomaf | maf

表 1. 活字の規則 (続き)

書体	意味	例
モノスペース	プログラミング・キーワードおよびライブラリー関数、コンパイラー・ビルトイン、プログラム・コードの例、コマンド・ストリング、またはユーザー定義の名前。	myprogram.c をコンパイルおよび最適化するには、`xlc myprogram.c -O3` と入力します。

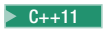
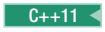
限定を示すエレメント (アイコン)

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

表 2. 限定を示すエレメント

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

表 2. 限定を示すエレメント (続き)

修飾子/アイコン	意味
C++11、または C++11 の始まり   C++11 の終わり	このテキストは、C++11 の一部として標準 C++ に導入されるフィーチャーを記述しています。

構文図

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

- 構文図は線のパスに沿って、左から右、上から下へと読んでいきます。

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

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

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

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

  完結したコマンド、ディレクティブ、またはステートメント以外の構文単位の図であるフラグメントは、|── 記号で始まり ──| 記号で終わります。
- 必須項目は、次のように横線 (メインパス) 上に表示されます。

```
▸▸─keyword─required_argument──────────────────────────────────────▸◂
```

- オプショナル項目は、次のようにメインパスの下側に表示されます。

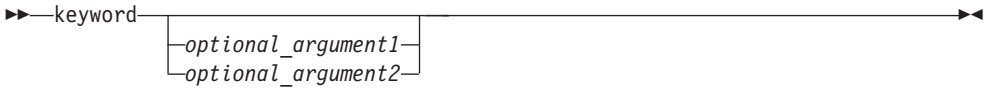
```
▸▸─keyword─┬──────────────────┬──────────────────────────────────▸◂
           └─optional_argument─┘
```

- 2 つ以上の項目から選択できる場合は、縦に重ねて表示されます。

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

```
▸▸─keyword─┬─required_argument1─┬──────────────────────────────────▸◂
           └─required_argument2─┘
```

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



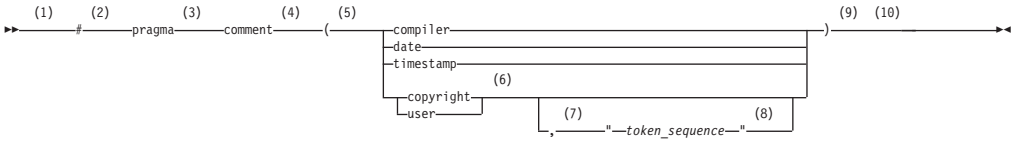
- 主線の上にある左に戻る矢印 (反復矢印) は、スタックされた項目から複数個選択できること、あるいは単一の項目を繰り返すことができることを示します。区切り文字も示されます (それがブランク以外の場合)。

- デフォルトの項目はメインパスの上に表示されます。

- キーワードは、イタリックでない文字で示され、示されているとおりに入力する必要があります。
- 変数は、イタリック体の小文字で示されます。変数は、ユーザー指定の名前や値を表します。
- 句読記号、括弧、算術演算子、または他のそのような記号が表示されている場合は、構文の一部として入力する必要があります。

構文図の例

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



注:

1 これが構文図の始まりです。

2 記号 # が最初に示される必要があります。

3 キーワード pragma が # 記号に続いて示される必要があります。

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

5 左括弧を指定する必要があります。

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

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

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

9 右括弧が必要です。

10 これが、構文図の終わりです。
以下の **#pragma comment** ディレクティブの例は、上記の図に従って、構文的に正しいものです。

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

本書の例

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

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

関連情報

以下のセクションでは、XL C/C++ に関連した情報を説明します。

IBM XL C/C++ 情報

XL C/C++ は、以下の形式で製品資料を提供しています。

- README ファイル

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

- インストール可能な man ページ

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

- インフォメーション・センター

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

- PDF 文書

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

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

表 3. XL C/C++ PDF ファイル

文書タイトル	PDF ファイル名	説明
IBM XL C/C++ for Linux, V13.1 インストール・ガイド, SA88-5404-00	install.pdf	XL C/C++ のインストール方法と基本的なコンパイルおよびプログラム実行のための環境の構成方法に関する情報が含まれています。
IBM XL C/C++ for Linux, V13.1 はじめに, SA88-5392-00	getstart.pdf	XL C/C++ 製品の概要と、環境のセットアップと構成、プログラムのコンパイルとリンク、およびコンパイル・エラーのトラブルシューティングに関する情報が含まれています。
IBM XL C/C++ for Linux, V13.1 コンパイラー・リファレンス, SA88-5388-00	compiler.pdf	さまざまなコンパイラー・オプション、プラグマ、マクロ、環境変数、および組み込み関数 (並列処理に使用されるものを含む) についての情報が含まれます。
IBM XL C/C++ for Linux, V13.1 ランゲージ・リファレンス, SA88-5396-00	langref.pdf	移植性および一般的規格への準拠についての言語拡張機能も含め、 IBM によってサポートされる C および C++ プログラミング言語に関する情報が記載されています。
IBM XL C/C++ for Linux, V13.1 最適化およびプログラミング・ガイド, SA88-5402-00	proguide.pdf	アプリケーションの移植、 Fortran コードによる言語間呼び出し、ライブラリー開発、アプリケーションの最適化および並列処理、および XL C/C++ 高性能ライブラリーなどの高度なプログラミング上のトピックに関する情報が記載されています。

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

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

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

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

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

標準および仕様

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

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*、別名 *C89*。
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*、別名 *C99*。

- *Information Technology - Programming languages - C, ISO/IEC 9899:2011*、別名 *C11*。(部分サポート)
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*、別名 *C++98*。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*、別名 *標準 C++*。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*、別名 *C++11*。(部分サポート)
- *Information Technology - Programming languages - Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. このドラフトの技術レポートは、C 標準委員会によって承認されており、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf で入手可能です。
- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. このドラフトの技術レポートは、C 標準委員会に提出されてあり、 http://www.open-std.org/JTC1/SC22/WG21/www/docs/papers/2005/n1836.pdf で入手可能です。
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. ベクトル処理テクノロジーをサポートするための、このベクトル・データ型の仕様はサイト http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf で使用可能です。
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- http://www.openmp.org で使用可能な「*OpenMP Application Program Interface Version 4.0 (部分サポート)*」。

その他の IBM 情報

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

その他の情報

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

テクニカル・サポート

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

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

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

第 1 章 32 ビット・モードおよび 64 ビット・モードの使用

XL C/C++ コンパイラーを使用して、32 ビット・アプリケーションと 64 ビット・アプリケーションのいずれも開発することができます。そのためには、コンパイル時に **-q32** (デフォルト) または **-q64** をそれぞれ指定します。

ただし、既存のアプリケーションを 32 ビット・モードから 64 ビット・モードに移植すると、さまざまな問題が生じる可能性があります。そのほとんどは、C/C++ の long データ型および pointer データ型のサイズと位置合わせが、この 2 つのモード間で異なることに関連します。次の表は、その違いをまとめたものです。

表 4. 32 ビット・モードおよび 64 ビット・モードにおけるデータ型のサイズと位置合わせ

データ型	**32 ビット・モード**		**64 ビット・モード**	
	サイズ	**位置合わせ**	**サイズ**	**位置合わせ**
long、signed long、unsigned long	4 バイト	4 バイト境界	8 バイト	8 バイト境界
pointer	4 バイト	4 バイト境界	8 バイト	8 バイト境界
size_t (ヘッダー・ファイル <cstddef> で定義)	4 バイト	4 バイト境界	8 バイト	8 バイト境界
ptrdiff_t (ヘッダー・ファイル <cstddef> で定義)	4 バイト	4 バイト境界	8 バイト	8 バイト境界

以下の各節では、上記のような違いが原因で陥りやすい落とし穴について説明するとともに、そのような問題の回避に役立つ、推奨されるプログラミングの実例をご紹介します。

- 2 ページの『long 値の割り当て』
- 4 ページの『ポインターの割り当て』
- 4 ページの『集合体データ位置合わせ』
- 5 ページの『Fortran コードの呼び出し』

32 ビットまたは 64 ビット・モードでコンパイルする場合は、アプリケーションの移植に関連する一部の問題を診断するのに役立つ、**-qwarn64** オプションを使用することができます。いずれのモードでも、プログラムの実行時にデータの切り捨てや損失などの問題が発生した場合、コンパイラーによって直ちに警告が出力されます。

64 ビット・モードでパフォーマンスを向上させるための提案事項については、『64 ビット・モードでの演算の最適化』を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-q32, -q64

-qwarn64

© Copyright IBM Corp. 1996, 2014

long 値の割り当て

limits.h 標準ライブラリー・ヘッダー・ファイルで定義される long 型整数の限界は、次の表で示すように、32 ビット・モードと 64 ビット・モードでは異なります。

表 5. 32 ビット・モードおよび 64 ビット・モードにおける長整数の定数限界

シンボリック定数	モード	値	16 進数	10 進数
LONG_MIN (signed long の最小値)	32 ビット	$-(2^{31})$	0x80000000L	–2,147,483,648
	64 ビット	$-(2^{63})$	0x8000000000000000L	–9,223,372,036,854,775,808
LONG_MAX (signed long の最大値)	32 ビット	$2^{31}-1$	0x7FFFFFFFL	+2,147,483,647
	64 ビット	$2^{63}-1$	0x7FFFFFFFFFFFFFFFL	+9,223,372,036,854,775,807
ULONG_MAX (unsigned long の最大値)	32 ビット	$2^{32}-1$	0xFFFFFFFFUL	+4,294,967,295
	64 ビット	$2^{64}-1$	0xFFFFFFFFFFFFFFFFUL	+18,446,744,073,709,551,615

この違いにより、次のような現象が生じます。

- double 変数に long 値を割り当てると、正確性が失われることがあります。
- long 変数に定数値を割り当てると、予期しない結果が生じることがあります。この問題については、 3 ページの『long 変数への定数値の割り当て』でさらに詳しく説明します。
- long 値をビット・シフトすると、 3 ページの『long 値のビット・シフト』で述べるように、それぞれ別の結果になります。
- 式で int 型と long 型を区別せずに使用すると、上位変換、下位変換、代入、引数引き渡しなどの方法で暗黙のうちに型変換が行われ、警告が発せられることなく、有効数字の切り捨て、符号のシフト、またはその他の予期しない結果を招くことがあります。これらの操作は、パフォーマンスに影響を与える場合があります。

他の変数に割り当てたり、関数に渡されるときに long 値がオーバーフローする場合は、以下を行う必要があります。

- 明示的な型キャストによって型を変更し、暗黙のうちに型変換されることのないようにする。
- long 型を受け付けるか戻す関数がすべて適切にプロトタイプ化されていることを確認する。
- long 型パラメーターが渡される先の関数が、この型のパラメーターを受け付け可能であることを確認する。

long 変数への定数値の割り当て

C および C++ では、定数の型識別は明示的規則に従って行われますが、多くのプログラムでは、16 進定数またはサフィックスなしの定数を「型のない」変数として使用し、2 の補数表示によって、32 ビット・システムで許可されている上限を超える値を途中で切り捨てます。このような大きな値は 64 ビット・モードでは 64 ビット long 型に拡張されることが多いため、たいていの場合次のような境界領域で、予期しない結果が生じることがあります。

- constant > UINT_MAX
- constant < INT_MIN
- constant > INT_MAX

境界での予期しない副次作用の例をいくつか、次の表に示します。

表 6. long 型に割り当てられる定数の、境界での予期しない結果

long に割り当てられる定数	等価の値	32 ビット・モード	64 ビット・モード
–2,147,483,649	INT_MIN–1	+2,147,483,647	–2,147,483,649
+2,147,483,648	INT_MAX+1	–2,147,483,648	+2,147,483,648
+4,294,967,726	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFF	UINT_MAX	–1	+4,294,967,295
0x100000000	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFFFFFFFFFF	ULONG_MAX	–1	–1

サフィックスのない定数では、型があいまいになることがあります。その場合、sizeof 演算の結果を変数に割り当てる場合など、プログラムの他の部分に影響が出ることが考えられます。例えば、32 ビット・モードでは、コンパイラーが 4294967295 (UINT_MAX) のような数値を unsigned long として入力すると、sizeof は 4 バイトを戻します。64 ビット・モードでは、この同じ数値が signed long となり、sizeof は 8 バイトを戻します。定数を関数に直接渡すと、同様の問題が起こります。

このような問題を回避するには、サフィックス L (long 型の定数の場合)、UL (unsigned long 型の定数の場合)、LL (long long 型の定数の場合)、または ULL (unsigned long long 型の定数の場合) を使用して、プログラムの他の部分における割り当てや式の計算に影響を与えると思われる定数をすべて明示的に入力します。上記の例では、4294967295U のように数値にサフィックスを付けると、コンパイラーは、32 ビット・モードまたは 64 ビット・モードで、この定数を常に unsigned int と認識するようになります。これらのサフィックスは、16 進定数に適用することもできます。

long 値のビット・シフト

long 値を左にビット・シフトした場合、32 ビット・モードと 64 ビット・モードでは結果が異なります。 4 ページの表 7 の例は、以下のコード・セグメントを使用して long 型の定数でビット・シフトを実行した場合の結果を示したものです。

```
long l=valueL<<1;
```

表 7. long 値のビット・シフトの結果

初期値	シンボリック定数	1 ビット単位のビット・シフト後の値	
		32 ビット・モード	64 ビット・モード
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	0x00000001FFFFFFFE

32 ビット・モードでは、0xFFFFFFFE は負になります。64 ビット・モードでは、0x00000000FFFFFFFE および 0x00000001FFFFFFFE は両方とも正になります。

ポインターの割り当て

64 ビット・モードでは、ポインターと int 型のサイズが同じではなくなりました。これによって、次のような影響が出ます。

- ポインターと int 型を交換すると、セグメンテーションに障害が発生します。
- int 型が予想される関数にポインターを渡すと、切り捨てが行われます。
- ポインターを戻す関数がそのように明示的にプロトタイプ化されていない場合は、ポインターではなく int が戻され、以下の例に示すように、結果として生じるポインターは切り捨てられます。

上記のような問題を回避するためには、次のようにします。

- 可能な場合は適切なヘッダー・ファイルを使用して、ポインターを返すすべての関数をプロトタイプ化する。
- 関数 (ポインターまたは int) 呼び出しで渡すパラメーターの型が、呼び出される関数で予想される型と一致するようにする。
- ポインターを整数型として処理するアプリケーションでは、32 ビット・モードでも、64 ビット・モードでも、long 型または unsigned long 型を使用する。
- 潜在的な問題関する警告メッセージをリスト・ファイルで取得するために、**-qwarn64** オプションを使用する。

集合体データ位置合わせ

通常、構造体は、32 ビット・モードと 64 ビット・モードの両方で、最も厳密に位置合わせされているメンバーに合わせて位置合わせされます。ただし、64 ビットでは long 型とポインターのサイズおよび位置合わせが変わるため、構造体の最も厳密なメンバーの位置合わせも変わる可能性があり、その場合は、構造体そのものの位置合わせにも変化が生じます。

ポインターまたは long 型を含む構造体は、32 ビット・アプリケーションと 64 ビット・アプリケーションで共用することはできません。long 型と int 型を共用するか、あるいはポインターを int 型にオーバーレイする共用体は、位置合わせを変更することができます。通常は、最も単純なものを除くすべての構造体について、位置合わせとサイズの依存関係を調べる必要があります。

あるモードでファイルに書き込まれた集合データは、他のモードで正しく読み取ることはできません。他の言語とのデータ交換にも同様の問題が伴います。

データ構造体 (ビット・フィールドを含む構造体など) の位置合わせについて詳しくは、 13 ページの『第 3 章 データの位置合わせ』を参照してください。

Fortran コードの呼び出し

非常に多くのアプリケーションが、C と C++ と Fortran を併用して、お互いを呼び出したり、ファイルを共用したりしています。そのようなアプリケーションでデータのサイズや型を変更する場合、現時点では、Fortran サイドで行うより C や C++ サイドで行う方が簡単です。次の表は、 C および C++ の型とそれに相当する Fortran の型を、モード別に示したものです。

表 8. C/C++ と Fortran の同等のデータ型

C/C++ の型	**Fortran の型**	
	32 ビット	**64 ビット**
signed int	INTEGER	INTEGER
signed long	INTEGER	INTEGER*8
unsigned long	LOGICAL	LOGICAL*8
pointer	INTEGER	INTEGER*8
		integer POINTER (8 バイト)

関連情報:

7 ページの『第 2 章 XL C/C++ の Fortran での使用』

第 2 章 XL C/C++ の Fortran での使用

XL C/C++ を使って FORTRAN で書かれた関数を、ユーザーの C および C++ プログラムから呼び出すことができます。このセクションでは、下記の領域内で FORTRAN コードを呼び出すためのプログラミング上の考慮事項をいくつか説明しています。

- 『ID』
- 8 ページの『対応するデータ型』
- 9 ページの『文字および集合データ』
- 10 ページの『関数呼び出し、およびパラメーター引き渡し』
- 11 ページの『関数ポインター』
- 11 ページの『サンプル・プログラム: Fortran を呼び出す C/C++』 は、FORTRAN サブルーチンを呼び出す C プログラムの例を提供しています。

言語の相互運用性について詳しくは、「*XL Fortran ランゲージ・リファレンス*」の『BIND』(Fortran 2003) および『プロシージャーの相互運用』を参照してください。

関連情報:

5 ページの『Fortran コードの呼び出し』

ID

Fortran から呼び出し可能な C++ 関数は、名前のマングリングを避けるため、`extern "C"` を使用して宣言する必要があります。詳しくは、Fortran 最適化およびプログラミング・ガイドで、Fortran と C/C++ コードの混用に関するオプションと規則を参照してください。

Fortran で書かれた関数を呼び出す C コードおよび C++ コードを書くときは、これらの勧告に従うことが必要です。

- ID として大文字を使用することは避ける。 デフォルトで XL Fortran は外部 ID を小文字にしますが、FORTRAN コンパイラーは外部の名前を大/小文字に分けて識別するように設定することができます。
- ID として長 ID の使用を避ける。 XL Fortran の ID での有効文字の最大数は 250[1] です。

注:

1. Fortran 90 および 95 言語標準では、ID を 31 文字以下にする必要があります。Fortran 2003 および 2008 標準では、ID を 63 文字以下にする必要があります。

© Copyright IBM Corp. 1996, 2014

対応するデータ型

下記のテーブルは、C/C+ および Fortran で使用可能なデータ型の対応を示しています。 C にあるいくつかのデータ型には、Fortran に等価な表記がないものがあります。言語間呼び出しを伴うプログラミングを行うときは、これらを使わないでください。

表 9. C, C++ および Fortran 間のデータ型の対応

C および C++ データ型	**Fortran データ型**	
	型	**ISO_C_BINDING モジュールの種類の型付きパラメーターを持つ型**
bool (C++)_Bool (C)	LOGICAL(1)	LOGICAL(C_BOOL)
char	CHARACTER	CHARACTER(C_CHAR)
signed char	INTEGER*1	INTEGER(C_SIGNED_CHAR)
unsigned char	LOGICAL*1	適用外
signed short int	INTEGER*2	INTEGER(C_SHORT)
unsigned short int	LOGICAL*2	適用外
int	INTEGER*4	INTEGER(C_INT)
unsigned int	LOGICAL*4	適用外
signed long int	INTEGER*4 (with **-q32**) INTEGER*8 (with **-q64**)	INTEGER(C_LONG)
unsigned long int	LOGICAL*4 (with **-q32**) INTEGER*8 (with **-q64**)	適用外
signed long long int	INTEGER*8	INTEGER(C_LONG_LONG)
unsigned long long int	LOGICAL*8	適用外
size_t	INTEGER*4 (with **-q32**) INTEGER*8 (with **-q64**)	INTEGER(C_SIZE_T)
intptr_t	INTEGER*4 (with **-q32**) INTEGER*8 (with **-q64**)	INTEGER(C_INTPTR_T)
intmax_t	INTEGER*8	INTEGER(C_INTMAX_T)
int8_t	INTEGER*1	INTEGER(C_INT8_T)
int16_t	INTEGER*2	INTEGER(C_INT16_T)
int32_t	INTEGER*4	INTEGER(C_INT32_T)
int64_t	INTEGER*8	INTEGER(C_INT64_T)
int_least8_t	INTEGER*1	INTEGER(C_INT_LEAST8_T)
int_least16_t	INTEGER*2	INTEGER(C_INT_LEAST16_T)
int_least32_t	INTEGER*4	INTEGER(C_INT_LEAST32_T)
int_least64_t	INTEGER*8	INTEGER(C_INT_LEAST64_T)
int_fast8_t	INTEGER または INTEGER*4	INTEGER(C_INT_FAST8_T)
int_fast16_t	INTEGER*4	INTEGER(C_INT_FAST16_T)
int_fast32_t	INTEGER*4	INTEGER(C_INT_FAST32_T)

表 9. C, C++ および Fortran 間のデータ型の対応 (続き)

C および C++ データ型	Fortran データ型	
	型	ISO_C_BINDING モジュールの種類の型付きパラメーターを持つ型
int_fast64_t	INTEGER*8	INTEGER(C_INT_FAST64_T)
float	REAL または REAL*4	REAL(C_FLOAT)
double	REAL*8 または DOUBLE PRECISION	REAL(C_DOUBLE)
long double	REAL*8 または DOUBLE PRECISION	REAL(C_LONG_DOUBLE)
float _Complex	COMPLEX*8 または COMPLEX(4)	COMPLEX(C_FLOAT_COMPLEX)
double _Complex	COMPLEX*16 または COMPLEX(8)	COMPLEX(C_DOUBLE_COMPLEX)
long double _Complex	COMPLEX*32 または COMPLEX(16)	COMPLEX(C_LONG_DOUBLE_COMPLEX)
構造体または共用体	派生型	適用外
enumeration	INTEGER*4	適用外
char[n]	CHARACTER*n	適用外
タイプへの配列ポインター、または type []	次元つき変数 (転置)	適用外
関数へのポインター	関数パラメーター	適用外
構造 (-qalign=packed つき)	シーケンス派生タイプ	適用外

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qldbl128、-qlongdouble

-qalign

文字および集合データ

大部分の数値データ型は、C/C++ および Fortran 間に対応があります。しかし、文字および集合データ型には特別な処理が必要です。

- C 文字ストリングは '**¥0**' 文字で区切られています。 Fortran 内では、すべての文字変数と式にはコンパイル時に決定された長さがあります。いつでも Fortran がストリング引数を別のルーチンに渡すときには、ストリング引数の長さを提供する非表示の引数を付加します。この長さ引数は C に明示的に宣言されなければなりません。C コードは NULL 終了文字を想定していません。供給された、または宣言された長さをいつも使う必要があります。

- n エレメントの C/C++ 配列の索引は 0...n-1 という形式になりますが、n エレメントの Fortran 配列の索引は、通常 1...n という形式になります。また、Fortran はユーザー指定の境界をサポートしますが、C/C++ はサポートしません。
- C は配列エレメントを行優先順位 (同じ行内の配列エレメントは隣接のメモリー・ロケーションを占める) で保管します。 Fortran は配列エレメントを昇順の記憶装置に列優先順位 (同じ列内の配列エレメントは隣接のメモリー・ロケーションを占める) で保管します。 表 10 には、「C 内の A[3][2] および Fortran 内の A(3,2) によって宣言された 2 次元配列の保管方法」が示されています。

表 10. 2 次元配列のストレージ

ストレージ・ユニット	C および C++ エレメント名	Fortran エレメント名
最低位	A[0][0]	A(1,1)
	A[0][1]	A(2,1)
	A[1][0]	A(3,1)
	A[1][1]	A(1,2)
	A[2][0]	A(2,2)
最高位	A[2][1]	A(3,2)

- 一般的に、多次元の配列について、ユーザーが配列のエレメントをメモリー内に並べられた順序でリストすると、行優先 (row-major) では右端の指標は最も高速に変化し、列優先 (column-major) では左端の指標がもっとも高速に変化するような形になります。

関数呼び出し、およびパラメーター引き渡し

関数は、C/C++ および Fortran の両方で等しい形でプロトタイプ化されなければなりません。

C では、デフォルトで、すべての関数引数は値による受け渡しであり、呼び出された関数は引き渡された値のコピーを受け取ります。 Fortran では、デフォルトで、引数は参照による受け渡しであり、呼び出された関数は引き渡された値のアドレスを受け取ります。 Fortran %VAL 組み込み関数、または VALUE 属性を使って値による受け渡しをすることができます。詳しくは、「*XL Fortran ランゲージ・リファレンス*」を参照してください。

参照による呼び出し (Fortran の場合) の場合はパラメーターのアドレスがレジスター内で渡されます。参照によってパラメーターを渡す場合、ユーザーが Fortran で書かれたプログラムを呼び出す、 C または C++ 関数を書くと、すべての引数はポインター、またはアドレス演算子を持つスカラーでなければなりません。

関数またはルーチンに対する言語間呼び出しについて詳しくは、「*XL Fortran 最適化およびプログラミング・ガイド*」の『言語間呼び出し』を参照してください。

関数ポインター

関数ポインターは、その値が関数のアドレスであるデータ型です。 Fortran では、EXTERNAL ステートメント内に現れる仮引数が関数ポインターです。 Fortran 2003 標準以降では、C_FUNPTR 型の Fortran 変数は関数ポインターと相互動作可能です。 関数ポインターは、呼び出し文のターゲット、または文の実引数のように、コンテキストでサポートされています。

サンプル・プログラム: Fortran を呼び出す C/C++

以下の例は、異なった言語で書かれたプログラム単位が、どのように結合されて 1 つのプログラムが作成されるか説明しています。また、引数として異なったデータ型を含むパラメーターが、C/C++ および Fortran サブルーチン間で受け渡しされることをデモします。例には、次のソース・ファイルが含まれています。

- メインプログラムのソース・ファイル: `example.c`
- Fortran の add 関数のソース・ファイル: `add.f`

メインプログラムのソース・ファイル: example.c

```
#include <stdio.h>
extern double add(int *, double [], int *, double []);

double ar1[4]={1.0, 2.0, 3.0, 4.0};
double ar2[4]={5.0, 6.0, 7.0, 8.0};

main()
{
int x, y;
double z;

x = 3;
y = 3;


z = add(&x, ar1, &y, ar2); /* Call Fortran add routine */
/* Note: Fortran indexes arrays 1..n */
/* C indexes arrays 0..(n-1) */

printf("The sum of %1.0f and %1.0f is %2.0f ¥n",
ar1[x-1], ar2[y-1], z);
}
```

Fortran の add 関数のソース・ファイル: add.h

```
REAL*8 FUNCTION ADD (A, B, C, D)
REAL*8 B,D
INTEGER*4 A,C
DIMENSION B(4), D(4)
ADD = B(A) + D(C)
RETURN
END
```

次のように、メインプログラムおよび Fortran の add 関数のソース・ファイルをコンパイルします。

```
xlc -c example.c
xlf -c add.f
```

コンパイル・ステップからのオブジェクト・ファイルをリンクして、実行可能ファイル `add` を作成します。

```
xlc -o add example.o add.o
```

バイナリーを実行します。

```
./add
```

出力は次のようになります。

```
The sum of 3 and 7 is 10
```

第 3 章 データの位置合わせ

XL C/C++ では、個々の変数、集合体のメンバー、集合体全体、およびコンパイル単位全体の各レベルでデータ位置合わせを指定するためのさまざまなメカニズムを用意しています。異なるプラットフォーム間で、または 32 ビット・モードと 64 ビット・モード間でアプリケーションの移植を行う場合は、それぞれの環境で使用できる位置合わせの設定の違いを考慮して、データの破損やパフォーマンスの低下を防ぐようにしてください。特にベクトル型の場合は特別な位置合わせ要件があり、それに従っていないと誤った結果を生ずることがあります。詳しくは、*AltiVec Technology Programming Interface Manual* を参照してください

XL C/C++ は、データの位置合わせを指定するための位置合わせモードと位置合わせ修飾子を提供します。 事前定義されたサブオプションを指定することにより、位置合わせモードを使って、コンパイル単位 (またはコンパイル単位のサブセクション) のデータ型のすべてにデフォルトで位置合わせを設定することができます。

位置合わせに使用すべきバイト数を正確に指定することにより、位置合わせ修飾子を使って、コンパイル単位内の特定の変数またはデータ型の位置合わせを設定することができます。

『位置合わせモードの使用』では、各種プラットフォームおよびアドレッシング・モデルでのすべてのデータ型に対するデフォルトの位置合わせモード、デフォルト設定を変更したりオーバーライドするために使うことが可能なサブオプションとプラグマ、そして単純変数、集合体、およびビット・フィールドの位置合わせモードの規則などについて説明します。

17 ページの『位置合わせ修飾子の使用法』では、特定の変数宣言のために現在有効になっている位置合わせモードをオーバーライドするための、ソース・コード内で使用可能な各種の指定子、プラグマ、および属性について説明します。さらに、コンパイル・プロセスで位置合わせモードと修飾子の優先順位を決定する規則も提供しています。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qaltivec

関連外部情報

「AltiVec Technology Programming Interface Manual」、http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手可能

位置合わせモードの使用

XL C/C++ がサポートする各々のデータ型は、プラットフォーム固有のデフォルト位置合わせモードに従って、バイト境界に沿って位置合わせされます。Linux では、デフォルトの位置合わせモードは **linuxppc** です。

© Copyright IBM Corp. 1996, 2014

ユーザーは下記のいずれかの仕組みを使用して、デフォルトの位置合わせモードを変更することができます。

- コンパイル・プロセスで、単一または複数ファイル内のすべての変数の位置合わせモードを設定する。

  この方法を使用するには、コンパイル時に、**linuxppc** サブオプション (デフォルト) または **bit_packed** サブオプションとともに、**-qalign** コンパイラー・オプションを指定します。
- ソース・コードのセクション内で、すべての変数の位置合わせモードを設定する。

  この方法を使用するには、**linuxppc** (デフォルト)、**bit_packed**、**reset** のいずれかのサブオプションとともに、ソース・ファイル内で **#pragma align** ディレクティブまたは **#pragma options align** ディレクティブを指定します。各ディレクティブは、別のディレクティブが出現するまで、またはコンパイル単位の終わりまで、そのディレクティブ以降のすべての変数で有効な位置合わせモードを変更します。

有効な位置合わせモードの各々は 表 11 に定義済みで、すべてのデータ型のスカラー変数について位置合わせ値としてバイト数を提供します。 32 ビット・モードと 64 ビット・モードに違いがある場合は、それらも示されます。また、集合体の最初 (スカラー) のメンバーと後続のメンバーの間に違いがあるときは、それらも示されます。

表 11. 位置合わせの設定 (値はバイト数で示される)

データ型	ストレージ	位置合わせの設定	
		linuxppc	**bit_packed**
_Bool (C)、bool (C++)	1	1	1
char、signed char、unsigned char	1	1	1
wchar_t (32 ビット・モード)	2	2	1
wchar_t (64 ビット・モード)	4	4	1
int、unsigned int	4	4	1
short int、unsigned short int	2	2	1
long int、unsigned long int (32 ビット・モード)	4	4	1
long int, unsigned long int (64 ビット・モード)	8	8	1
long long	8	8	1
float	4	4	1
double	8	8	1
long double	8	8	1
long double with **-qldbl128**	16	16	1
pointer (32 ビット・モード)	4	4	1
pointer (64 ビット・モード)	8	8	1
vector types	16	16	1

あるプラットフォーム上のアプリケーションでデータを生成し、そのデータを別のプラットフォーム上のアプリケーションで読み取る場合は、結果としてすべてのプラットフォーム上で等価データ位置合わせとなる、**bit_packed** モードの使用を推奨します。

注:

- ビット・パック構造体内のベクトルは、その確実な位置合わせのために追加の処置をしない限り正しく位置合わせされない可能性があります。
- ヒープ割り当てストレージまたはポインター演算を経由してベクトルにアクセスすると、ベクトルの位置合わせで問題が発生する可能性があります。例えば、`double __align(16) my_array[1000]` は 16 バイトで位置合わせされますが、`my_array[1]` はされません。`my_array[i]` の位置合わせ方法は、`i` の値によって決まります。

『集合体の位置合わせ』では、集合体全体の位置合わせの規則を説明して、集合体レイアウトの実例を提供しています。 16 ページの『ビット・フィールドの位置合わせ』では、その他の規則と、ビット・フィールドの使用と位置合わせに関する考慮事項について説明し、ビット・パック位置合わせの例を示します。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qalign

-qldbl128、-qlongdouble

#pragma options

集合体の位置合わせ

14 ページの表 11 (13 ページの『位置合わせモードの使用』) に含まれるデータは、スカラー変数と、構造体、共用体、クラスなどの集合体のメンバーの変数に適用されます。下記の規則が集合体変数、すなわち構造体、共用体、またはクラスの全体に対して適用されます (修飾子が何も存在しない場合)。

- すべての位置合わせモードで、集合体のサイズ は、その位置合わせ値の倍数のうち、集合体のすべてのメンバーを包含することのできる最小の倍数となる。
- C 空の集合体はサイズ 0 バイトが割り当てられている。そのため、2 つの異なる変数が同じアドレスを持つ場合があります。
- C++ 空の集合体はサイズ 1 バイトが割り当てられている。静的データ・メンバーは、位置合わせ、または集合体のサイズには加わらないことに注意してください。したがって、単一の静的データ・メンバーのみを含む構造体またはクラスはサイズ 1 バイトになります。
- すべての位置合わせモードで、集合体の位置合わせは、そのいずれかのメンバーの最大の位置合わせ値と等しい。パック位置合わせモードを例外として、natural 位置合わせがその集合体の位置合わせより小さいメンバーの場合は、空のバイトで埋め込まれます。
- 位置合わせされる集合体はネストすることができ、ネストされた個々の集合体に適用できる位置合わせ規則は、ネストされた集合体の宣言時に有効になっている位置合わせモードによって決まる。

注:

- C++ C++ コンパイラーは、基本クラスまたは仮想関数を含むクラスに対して、余分なフィールドを生成することがあります。これらの型のオブジェクトは、集合体に対する通常のマッピングに準拠しない可能性があります。
- 集合体の位置合わせは、すべてのコンパイル単位で同じになっている必要があります。例えば、集合体の宣言がヘッダー・ファイルにあり、このヘッダー・ファイルを 2 つの別々のコンパイル単位にインクルードする場合、両方のコンパイル単位で同じ位置合わせモードを選択します。

ビット・フィールドを含む集合体の位置合わせ規則については、『ビット・フィールドの位置合わせ』を参照してください。

ビット・フィールドの位置合わせ

ビット・フィールドを宣言する際に使用できるデータ型は、_Bool (C)、

C++ bool (C++)、char、signed char、unsigned char、short、unsigned short C++、int、unsigned int、long、unsigned long、C++ long long、または unsigned long long C++ です。 ビット・フィールドの位置合わせは、その基本タイプおよびコンパイル・モード (32 ビットまたは 64 ビット) によって異なります。

C ビット・フィールドの長さは、その基本タイプの長さを超えることはできません。拡張モードでは、ビット・フィールドに対して sizeof 演算子を使用することができます。ビット・フィールド上の sizeof 演算子は基本タイプのサイズを戻します。 C

C++ ビット・フィールドの長さは、その基本タイプの長さを超えてもかまいませんが、残りのビットはフィールドの埋め込みに使用され、値は実際には保管されません。 C++

ただし、ビット・フィールドを含む集合体の位置合わせ規則は、有効な位置合わせモードによって異なります。この規則については、以下で説明します。

Linux PowerPC® 位置合わせの規則

- ビット・フィールドは、ビット・フィールド・コンテナーから割り当てられます。このコンテナーのサイズは、宣言されたビット・フィールドの型によって決定されます。例えば、 char ビット・フィールドは 8 ビット・コンテナーを使用し、int ビット・フィールドは 32 ビット・コンテナーを使用するといったようになります。コンテナーは、そのビット・フィールドを収容できる十分の大きさがなければなりません。ビット・フィールドを複数のコンテナー間で分割して使用することはできません。
- コンテナーは、そのコンテナーの型の自然境界上で開始されるものとして、集合体内で位置合わせされます。ビット・フィールドは、コンテナーの開始点から割り当てられるとは限りません。
- 長さ 0 のビット・フィールドが集合体の最初のメンバーである場合は、その集合体の位置合わせに影響を与えることはなく、次のデータ・メンバーでオーバーラップされます。長さ 0 のビット・フィールドが集合体の最初のメンバーでない場合は、その基底宣言型によって決まる次の位置合わせ境界まで埋め込みを行いますが、その集合体の位置合わせには影響を与えません。

- 名前のないビット・フィールドは、集合体の位置合わせには影響を及ぼしません。

ビット・パック位置合わせの規則

- ビット・フィールドは 1 バイトで位置合わせされ、デフォルトではビット・フィールド間の埋め込みなしでパックされます。
- 長さ 0 のビット・フィールドがあると、次のメンバーは、次のバイト境界から開始します。長さ 0 のビット・フィールドがすでにバイト境界にある場合は、次のメンバーはこの境界から開始します。ビット・フィールドに続く非ビット・フィールド・メンバーは、次のバイト境界に位置合わせします。

ビット・パック位置合わせの例

```
#pragma options align=bit_packed
struct {
   int a : 8;
   int b : 10;
   int c : 12;
   int d : 4;
   int e : 3;
   int : 0;
   int f : 1;
   char g;
   } A;

pragma options align=reset
```

A のサイズは 7 バイトです。A の位置合わせは 1 バイトです。 A のレイアウトは次のようになります。

メンバー名	バイト・オフセット	ビット・オフセット
a	0	0
b	1	0
c	2	2
d	3	6
e	4	2
f	5	0
g	6	0

位置合わせ修飾子の使用法

XL C/C++ はまた、位置合わせ修飾子を提供します。これによりユーザーは、個々の変数または集合体のメンバーの宣言や定義のレベルで、位置合わせをさらに細かくコントロールして実行することができるようになります。提供される修飾子には次のものがあります。

#pragma pack(...)

有効なアプリケーション。
ディレクティブの直後に続く集合体の全体 (全体として)。

効果。 適用対象の集合体のメンバーに対して、最大の位置合わせとして特定のバイ

ト数を設定します。また、1 つのビット・フィールドがコンテナー境界をクロスすることを許容します。選択された集合体の有効な位置合わせ値の低減に使用しています。

有効な値。

-qpack_semantic=ibm が有効な場合は (XL C/C++ ではデフォルト)、**1**、**2**、**4**、**8**、**16**、**nopack**、**pop**、および空の小括弧です。空の小括弧を使用することには、**nopack** と同一の機能があります。 **-qpack_semantic=gnu** が有効な場合は (gxlc および gxlc++ ユーティリティーを使用する場合にはデフォルト)、**[push,]1**、**[push,]2**、**[push,]4**、**[push,]8**、**[push,]16**、**pop**、および空の括弧です。

__attribute__((aligned(n)))

有効なアプリケーション。

1 つの変数属性として単一の集合体 (全体として)、すなわち構造体、共用体、またはクラスに適用されます。または集合体の個々のメンバーに適用されます。 [1] 1 つの 型属性として、その型として宣言されているすべての集合体に適用されます。 [2] これが `typedef` 宣言に適用された場合は、その型のすべてのインスタンスに適用されます。

効果。 指定された変数 (単数または複数可) の最小の位置合わせ値として特定のバイト数を設定します。一般的に選択された変数の有効な位置合わせ値の増加に使用しています。

有効な値。

n は、2 の正のべき数、または NIL であることが必須。 NIL は、`__attribute__((aligned()))` または `__attribute__((aligned))` として指定できます。これは、最大システム位置合わせ (すべての UNIX プラットフォームで 16 バイト) を指定するのと同じです。

__attribute__((packed))

有効なアプリケーション。

1 つの変数属性として単純な変数、または集合体の個々のメンバー、すなわち構造体、またはクラス [1] に適用されます。1 つの型属性として、その型として宣言されているすべての集合体のすべてのメンバーに適用されます。

効果。 選択された変数 (単数、複数可) の最大位置合わせ値を、適用対象である可能な限度の最小位置合わせ値、すなわち 1 バイトの変数と 1 ビットのビット・フィールドに適用します。

__align(n)

効果。 特定のバイト数に適用する、変数または集合体の最小の位置合わせ値を設定します。またこの変数が使用しているストレージ容量を効果的に増やすこともあります。選択された変数の有効な位置合わせ値の増加に使用しています。

有効なアプリケーション。

集合体の個々のメンバーではなく、集合体でもない限り、単純な静的 (または全体的な) 変数、または全体的な集合体に適用します。

有効な値。

n は、正の 2 のべき数でなければなりません。XL C/C++ では、システムの最大値を超える値を指定することもできます。

注:

1. 宣言のコンマで区切られた変数リスト内で修飾子が宣言の先頭に置かれている場合、それは宣言内のすべての変数に適用されます。さもなければ、その直前の変数のみに適用されます。
2. `struct` の宣言内の修飾子の置き方によっては型の定義に適用することが可能であり、したがってその型の すべて のインスタンスに適用されます。またはその型の単一インスタンスのみに適用されることもあります。詳細については、「*XL C/C++ ランゲージ・リファレンス*」の*型属性*を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

#pragma pack

-qpack_semantic

「*XL C/C++ ランゲージ・リファレンス*」の関連情報

aligned 型属性 (IBM 拡張)

packed 型属性 (IBM 拡張)

__align 型修飾子 (IBM 拡張)

型属性 (IBM 拡張)

aligned 変数属性 (IBM 拡張)

packed 変数属性 (IBM 拡張)

第 4 章 浮動小数点演算の処理

以下の節では、参照情報、移植の際の考慮事項、およびコンパイラー・オプションを使用して浮動小数点演算を管理する場合に推奨される手順について述べます。

- 『浮動小数点フォーマット』
- 『乗加法演算の処理』
- 22 ページの『厳密な IEEE 準拠のためのコンパイル』
- 22 ページの『浮動小数点定数の折り畳みと丸めの処理』
- 24 ページの『浮動小数点例外の処理』

浮動小数点フォーマット

XL C/C++ は以下の 2 進浮動小数点フォーマットをサポートします。

- 0 および 10^{-38} から 10^{+38} の近似絶対正規化範囲で、10 進法で約 7 桁の精度を持つ 32 ビット単精度の浮動小数点数
- 0 および 10^{-308} から 10^{+308} の近似絶対正規化範囲で、10 進法で約 16 桁の精度を持つ 64 ビット倍精度の浮動小数点数
- 倍精度値より範囲が広く、10 進法で約 32 桁の精度を持つ 128 ビット拡張精度の浮動小数点数

`long double` 型は、**-qldbl128** コンパイラー・オプションの設定によっては倍精度値または拡張精度値を表すことがあるので、注意してください。デフォルトは 128 ビットです。以前のコンパイルとの互換性のために、`long double` が 64 ビットである必要がある場合には、**-qnoldbl128** を使用できます。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qldbl128、-qlongdouble

乗加法演算の処理

コンパイラーはデフォルトで、*a+b*c* のような 2 進浮動小数点式の、単一の IEEE 754 非互換の乗加法命令を生成します。これは 2 命令よりも 1 命令が高速であることが 1 つの理由です。乗算と加算の間の演算には丸めがないことと、より高い精度の結果が得られる可能性があるからです。しかし、精度が高くなった結果、他の環境では異なった結果が得られることになる可能性があり、 *x*y-x*y* がゼロにならない場合があります。この問題を避けるため、**-qfloat=nomaf** オプションを使用して乗加法命令の生成を抑止することができます。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qfloat

© Copyright IBM Corp. 1996, 2014

厳密な IEEE 準拠のためのコンパイル

XL C/C++ はデフォルトで、IEEE 標準について、そのすべてではないが大部分の規則に従っています。 **-qnostrict** オプションでコンパイルする場合、これがデフォルトで最適化レベル **-O3** 以上であると、いくつかの IEEE 浮動小数点規則に違反することになり、パフォーマンスは向上しますがプログラムの正確さに影響します。この問題を避けると同時に IEEE 標準に厳密に準拠したコンパイルを実行するには、以下のオプション使用します。

- **-qfloat=nomaf** コンパイラー・オプションを使用します。
- 実行時にプログラムが丸めモードを変更する場合は、**-qfloat=rrm** オプションを使用します。
- データまたはプログラム・コードにシグナル方式 NaN 値 (NaNS) が含まれている場合は、以下のオプションの組み合わせのいずれかを使用します。 (シグナル方式 NaN は静止 NaN とは異なります。プログラムまたはデータに明示的にコーディングするか、または **-qinitauto** コンパイラー・オプションを使ってそれらを作成する必要があります。)
  - **-qfloat=nans** オプションおよび **-qstrict=nans** オプション
  - **-qfloat=nans** オプションおよび **-qstrict** オプション
- **-O3**、**-O4**、または **-O5** を使ってコンパイルする場合、その後にオプション **-qstrict** を組み込んでください。**-qstrict** サブオプションを使用して、最適化プログラムで実行されるトランスフォーメーションの制御レベルを改善することができます。

関連情報:

49 ページの『拡張最適化』

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

-qfloat

-qstrict

-qinitauto

浮動小数点定数の折り畳みと丸めの処理

コンパイラーはデフォルトで、定数オペランドに関与する大部分の操作をそのコンパイル時の結果で置き換えます。この処理は、定数折り畳みと呼ばれます。最適化、または **-qnostrict** オプションを使う場合に、追加の折り畳みの機会が発生する可能性があります。コンパイル時の浮動小数点操作で折り畳みが行われた結果は、通常は実行時に得られたものと同じ結果ですが、以下の場合は異なります。

- コンパイル時の丸めモードが実行時の丸めモードと異なるとき。デフォルトでは、両方の場合とも最も近い値への丸めですが、プログラムが実行時に異なった結果を避けるために丸めモードを変更する場合は、以下のいずれかの操作を行ってください。

  - 適切な **-y** オプションを使用して、コンパイル時の丸めモードを実行時のモードと一致するように変更してください。詳細な情報と例示については、『コンパイル時と実行時の丸めモードのマッチング』を参照してください。
  - **-qfloat=nofold** オプションを使ってコンパイルすることにより、折り畳みを抑止します。
- *a+b*c* のような式については、コンパイル時に部分的または全面的に評価を行います。実行時の乗加法命令は中間の丸めを全く使用しないにもかかわらず、*b*c* は *a* に加えられる前に丸められる場合があるため、実行時に得られた結果とは異なる可能性があります。異なった結果を避けるには下記のどちらかの操作を行います。
  - **-qfloat=nomaf** オプションを使用してコンパイルすることにより、乗加法命令の使用を抑止します。
  - **-qfloat=nofold** オプションを使ってコンパイルすることにより、折り畳みを抑止します。
- 演算の結果は、無限大、NaN、またはゼロにアンダーフローのいずれかとなります。たとえ、**-qflttrap** オプションを指定してコンパイルしても、コンパイル時折り畳みによって例外の実行時検出が不可能となります。このような例外の脱落を避けるには、**-qfloat=nofold** オプションを使用して折り畳みを抑止します。

関連情報:

24 ページの『浮動小数点例外の処理』

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

-qfloat

-qstrict

-qflttrap

コンパイル時と実行時の丸めモードのマッチング

コンパイル時と実行時に使用されるデフォルトの丸めモードは、最も近い偶数への丸めです。プログラムが実行時に丸めモードを変更すると、浮動小数点計算はコンパイル時に得られる結果と少し違う場合があります。以下の例は を説明しています。

```
#include <float.h>
#include <fenv.h>
#include <stdio.h>

int main ( )
{
 volatile double one = 1.f, three = 3.f;  /* volatiles are not folded */
 double one_third;

 one_third = 1. / 3.;  /* folded */
 printf ("1/3 with compile-time rounding = %.17f¥n", one_third);

 fesetround (FE_TOWARDZERO);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to zero = %.17f¥n", one_third);

 fesetround (FE_TONEAREST);
```

```
one_third = one / three;  /* not folded */

printf ("1/3 with execution-time rounding to nearest = %.17f¥n", one_third);

fesetround (FE_UPWARD);
one_third = one / three;  /* not folded */

printf ("1/3 with execution-time rounding to +infinity = %.17f¥n", one_third);

fesetround (FE_DOWNWARD);
one_third = one / three;  /* not folded */

printf ("1/3 with execution-time rounding to -infinity = %.17f¥n", one_third);

return 0;
}
```

デフォルト・オプションでコンパイルすると、このコードは以下の結果を示します。

```
1/3 with compile-time rounding = 0.33333333333333331
1/3 with execution-time rounding to zero = 0.33333333333333331
1/3 with execution-time rounding to nearest   = 0.33333333333333331
1/3 with execution-time rounding to +infinity = 0.33333333333333337
1/3 with execution-time rounding to -infinity = 0.33333333333333331
```

4 番目の計算は丸めモードを「無限に丸め」で行ったもので、コンパイル時に「最近値へ丸め」を使用して行った最初の計算とは少し異なっています。 浮動小数点計算のコンパイル時折り畳みを抑止するための **-qfloat=nofold** オプションを使用しない場合は、 **-y** コンパイラー・オプションを、コンパイル時と実行時の丸めモードに一致するサブオプション付きで使用することを推奨します。上記の例で、**-yp** (round-to-infinity) を使用したコンパイルの計算結果は、最初の計算で以下のようになっています。

```
1/3 with compile-time rounding = 0.33333333333333337
```

一般的に、丸めモードを +infinity または -infinity に変更する場合は、**-qfloat=rrm** オプションも使用することをお勧めします。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qfloat

-y

浮動小数点例外の処理

デフォルトでは、ゼロによる除法、無限大による除法、オーバーフロー、アンダーフローなどの無効な演算は、実行時には無視されます。ただし、**-qflttrap** オプションを使用するか、C またはオペレーティング・システムの関数を呼び出すことで、これらのタイプの例外を検出できます。

さらに、適切なサポート・コードをプログラムに追加すると、例外が発生してもプログラムの実行を続けて、例外の原因となった演算の結果を修正することができます。

しかし、定数を含む浮動小数点計算は、コンパイル時に折り畳みがされるのが普通であるため、実行時に発生する可能性のある例外は生じません。**-qflttrap** オプショ

ンが、実行時の浮動小数点例外をすべてトラップできるようにするには、**-qfloat=nofold** オプションを使用して、コンパイル時の折り畳みをすべて抑止することを検討してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

 -qfloat

 -qflttrap

第 5 章 C++ コンストラクターの使用

C++11

C++11 以前は、同じクラスの複数コンストラクターにおける共通の初期化を、堅固で保守可能な方式で 1 個所に集中させることができませんでした。この問題は、以下のような基本的な方法で解決することができます。

委任コンストラクターの使用:

委任コンストラクター機能を使用すると、共通の初期化を 1 つのコンストラクターに集中させることができるため、プログラムの可読性および保守可能性を高めることができます。委任コンストラクターは、オブジェクト・ファイルのコード・サイズと総体サイズを小さくするのに役立ちます。詳しくは、『委任コンストラクター (C++11) の使用』を参照してください。

C++11

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qlanglvl

委任コンストラクター (C++11) の使用

注: IBM は、C++11 (標準化前の呼称は C++0x) の一部の機能をサポートしています。IBM は引き続き、この標準の機能を開発および実装する予定です。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含めて、C++11 のすべての機能の実装を IBM が完了するまでは、リリースごとに実装が変更される可能性があります。IBM は、ソース、バイナリー、リストなどのコンパイラー・インターフェースにおいて、新しい C++11 機能の IBM 実装の旧リリースとの互換性を維持することを試みません。

構文上は、*委任コンストラクター* と*ターゲット・コンストラクター* は他のコンストラクターと同じインターフェースを提示します。「*XL C/C++ ランゲージ・リファレンス*」の『委任コンストラクター (C++11)』を参照してください。

委任コンストラクター機能を使用する場合は、以下の点を考慮してください。

- 仮想基底、直接非仮想基底、およびクラス・メンバーがターゲット・コンストラクターによって適宜初期化されるような形で、ターゲット・コンストラクター・インプリメンテーションを呼び出してください。
- この機能がコンパイル時間と実行時パフォーマンスに与える影響はごくわずかです。しかし、可能な限り、委任コンストラクターの代わりに既存のコンストラクターでデフォルト引数を使用することを推奨します。インライン化およびプロシージャー間分析を行わないと、機能呼び出しのオーバーヘッドと不透明性の増大のため、実行時パフォーマンスが低下することがあります。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qlanglvl

© Copyright IBM Corp. 1996, 2014

「*XL C/C++ ランゲージ・リファレンス*」**の関連情報**


委任コンストラクター (C++11)

第 6 章 C++ テンプレートの使用

C++ では、テンプレートを使用して次の関連項目のセットを宣言することができます。

- クラス (構造体を含む)
- 関数
- テンプレート・クラスの静的データ・メンバー

冗長なテンプレート・インスタンス生成の削減

アプリケーション内では、同じテンプレートのインスタンスを複数回生成することができます。その場合の引数は、同じであっても異なっていてもかまいません。さまざまなコンパイル単位内のテンプレート・インスタンス化に対して同じ引数を使用する場合、繰り返されるインスタンスは冗長になります。これらの冗長なインスタンス生成は、コンパイル時間や実行可能プログラムのサイズの増大につながり、何のメリットもありません。

冗長なインスタンス生成の問題に対処するには、基本的に次のいくつかの方法があります。

リンク中の冗長度の処理

最終実行可能ファイルのサイズの増加量は、プログラムのコンパイル方法またはソース・ファイルの変更方法を変えるだけの正当な理由にならないほどに、小さい可能性があります。ほとんどのリンカーには、なんらかの形式のガーベッジ・コレクション機能があります。 **-qtemplateregistry** または **-qtempinc** を使用せずに **-qtmplinst=always** または **-qtmplinst=auto** を使用すると、重複インスタンス生成のコンパイル時間管理が行われません。

ソース・コードでの暗黙的インスタンス生成の制御

特殊化の暗黙的インスタンス生成の集中: オブジェクト・ファイルに含まれる、必要なインスタンス生成ごとのインスタンスの数および未使用のインスタンス生成の数がより少なくなるよう、ソース・コードを編成します。これは、使用するには最も適さない方法です。個々のテンプレートがどこで定義され、どのインスタンス生成が使用され、個々のインスタンス生成に対して明示的インスタンス生成をどこで宣言するべきかを知っている必要があるためです。

C++11 **明示的インスタンス生成宣言の使用:** 明示的インスタンス生成宣言機能を使用すると、テンプレート特殊化またはそのメンバーの暗黙的インスタンス生成を抑制することができます。この機能は、オブジェクト・ファイルの総体サイズを減少させるのに役立ちます。また、抑制されたシンボル定義が共有ライブラリー内で検出されるようになっている場合、またはシステム・リンカーがシンボルの追加定義を常に除去することができない場合、この機能によって最終的な実行可能ファイルのサイズが減少する場合があります。詳しくは、 35 ページの『明示的インスタンス生成宣言の使用 (C++11)』 C++11 を参照してください。

© Copyright IBM Corp. 1996, 2014

注: 暗黙のインスタンス生成をソース・コードで制御する、または明示的インスタンス生成宣言を使用するには、**-qtmplinst=none** または **-qtmplinst=noinlines** オプションを使用して、意図しない暗黙のインスタンス生成の発生を防ぐことができます。

コンパイラーに、インスタンス生成情報をレジストリーに保管するよう指示する

-qtemplateregistry コンパイラー・オプションを使用します。テンプレートによる個々のインスタンス生成に関する情報が、テンプレート・レジストリーに保管されます。コンパイラーは、同じテンプレートのインスタンスを同じ引数で再び生成するように要求されると、新たに生成する代わりに、最初のオブジェクト・ファイルにあるインスタンス生成をポイントします。この方法については、 33 ページの『-qtemplateregistry コンパイラー・オプションの使用』で説明します。

コンパイラーに、生成したインスタンスをテンプレート・インクルード・ディレクトリーに保管するよう指示する

-qtempinc コンパイラー・オプションを使用します。 テンプレート定義ファイルとテンプレート・インプリメンテーション・ファイルの構造が所定のものである場合は、テンプレートで生成された個々のインスタンスはテンプレート・インクルード・ディレクトリーに保管されます。コンパイラーは、同じテンプレートのインスタンスを同じ引数で再び生成するように要求されると、新たに生成する代わりに保管したバージョンを使用します。テンプレート・インクルード・ディレクトリーに作成されたソース・ファイルは、リンク・ステップで、すべてのインスタンス生成が行われるまで再帰的にコンパイルされます。 この方法については、 31 ページの『-qtempinc コンパイラー・オプションの使用』で説明します。

注: この方法は、レガシー・コードのみに使用してください。

注:

- **-qtempinc** コンパイラー・オプションと **-qtemplateregistry** コンパイラー・オプションは、同時には使用できません。
- **-qtemplateregistry** は、以下の理由により、**-qtempinc** よりも優れた方法になります。
  - **-qtemplateregistry** は、**-qtempinc** よりもメリットが大きい。
  - **-qtemplateregistry** では、ヘッダー・ファイルの変更が必要ない。

コンパイラーは、以下の条件のいずれかが真になるまで、暗黙のインスタンス生成用のコードを生成します。

- **-qtmplinst=none** または **-qtmplinst=noinlines** を使用する。
- **-qtmplinst** と **-qnotemplateregistry** を組み合わせたときのデフォルト・サブオプション **-qtmplinst=auto** を使用する。
- **-qtmplinst=auto** と **-qtempinc** を組み合わせ、**-qtempinc** を使用するように編成されたテンプレート・ソースを使用する。
- C++11 このインスタンス生成の明示的インスタンス生成宣言は、現行の変換単位にあります。 C++11

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

 -qtempinc (C++ のみ)

 -qtemplateregistry (C++ のみ)

 -qtmplinst (C++ のみ)

 -qlanglvl

-qtempinc コンパイラー・オプションの使用

-qtempinc を使用するには、アプリケーションを次のように構成する必要があります。

- テンプレート宣言ファイルで、クラス・テンプレートと関数テンプレートを拡張子 `.h` を付けて宣言する。
- テンプレート宣言ごとに、1 つのテンプレート定義ファイルを作成する。このファイルの名前は、テンプレート宣言ファイルの名前と同じで拡張子が `.c` または `.t`、であるか、あるいは **#pragma implementation** ディレクティブで指定する必要があります。クラス・テンプレートの場合は、インプリメンテーション・ファイルがメンバー関数と静的データ・メンバーを定義します。関数テンプレートの場合は、インプリメンテーション・ファイルはその関数を定義します。
- ソース・プログラムで、個々のテンプレート宣言ファイルに対して `#include` ディレクティブを指定する。
- (オプション) コードが **-qtempinc** コンパイルと **-qnotempinc** コンパイルの両方に適用できることを確認するためには、個々のテンプレート宣言ファイルに、__TEMPINC__ マクロが定義されて いない ことを条件に、対応するテンプレート・インプリメンテーション・ファイルを組み込む。(このマクロは、 **-qtempinc** コンパイル・オプションを使用すると、自動的に定義されます。) こうすると、次のような結果が得られます。
  - **-qnotempinc** を指定してコンパイルすると、必ず、テンプレート・インプリメンテーション・ファイルが組み込まれます。
  - **-qtempinc** を指定してコンパイルすると、コンパイラーは、テンプレート・インプリメンテーション・ファイルを組み込みません。その代わりに、コンパイラーは、特定のインスタンス生成が最初に必要になったときに、テンプレート・インプリメンテーション・ファイルと名前が同じで、拡張子が `.c` であるファイルを探します。これ以後は、同じインスタンス生成が必要になると、コンパイラーは、テンプレート・インクルード・ディレクトリーに保管されているコピーを使用します。

注:

- **-qtempinc** オプションは、レガシー・アプリケーションのみに使用します。
- **-qtemplateregistry** を使用します。これは、**-qtempinc** よりもメリットが大きく、ソース・ファイルを変更する必要がありません。詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の『-qtemplateregistry (C++ のみ)』を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

 -qtempinc (C++ のみ)

 -qtemplateregistry (C++ のみ)

 -qtmplinst (C++ のみ)

 #pragma implementation (C++ のみ)

-qtempinc の使用例

テンプレートの宣言と定義が別個のファイルにある場合に、コンパイラーが暗黙的なインスタンス化を管理する方法を以下の例に示します。この例には、次のソース・ファイルが含まれています。

- テンプレート宣言ファイル: a.h
- 対応するテンプレート実装ファイル: a.t
- メインプログラムのソース・ファイル: a.cpp

テンプレート宣言ファイル: a.h

```
struct IC {
  virtual void myfunc() = 0;
};

template <class T> struct C : public IC{
  virtual void myfunc();
};

#ifndef __TEMPINC__
  #include "a.t"
#else
  #pragma implementation("a.t")
#endif
```

テンプレート実装ファイル: a.t

このファイルには、main プログラムから呼び出される myfunc 関数のテンプレート定義が含まれています。

```
template <class T> void C<T>::myfunc() {}
```

メインプログラム・ファイル: a.cpp

このファイルは、暗黙的なインスタンス化を必要とするオブジェクトを作成します。

```
#include "a.h"

int main() {
  IC* pIC = new C<int>();
  pIC->myfunc();
}
```

次のコマンドを使用して、メインプログラム a.cpp をコンパイルできます。

```
xlC -qtempinc a.cpp
```

注:

- **-qnotempinc** が指定されると、テンプレート実装ファイルが組み込まれます。それ以外の場合、**-qtempinc** が指定されると、`#pragma implementation` ディレクティブが、テンプレート実装ファイルへのパスをコンパイラーに通知します。
- `#pragma implementation` ディレクティブが指定されない場合、コンパイラーは、デフォルトでは `a.c` をテンプレート実装ファイルとして検索します。

テンプレート・インスタンス化ファイルの再生成

コンパイラーは、個々のテンプレート・インプリメンテーション・ファイルに対応する TEMPINC ディレクトリーに、テンプレート・インスタンス化ファイルを作成します。 コンパイルを行うたびに、コンパイラーはそのファイルに情報を追加することはできても、そのファイルから情報を除去することはありません。

プログラムを開発する際には、テンプレート関数参照を除去したり、プログラムを再編成したりして、テンプレート・インスタンス生成ファイルの内容が古くなることがあります。 TEMPINC 宛先を定期的に削除し、プログラムを再コンパイルしてください。

共用ライブラリーでの -qtempinc の使用

従来のアプリケーション開発環境では、異なるアプリケーション同士がソース・ファイルとコンパイル済みファイルを共用することができます。テンプレートを使用すると、アプリケーションはソース・ファイルを共用できますが、コンパイル済みファイルは共用できません。

-qtempinc を使用する場合は、次のことに注意してください。

- アプリケーションごとに、独自の TEMPINC 宛先が必要です。
- アプリケーションのソース・ファイルの一部が既に別のアプリケーション用にコンパイルされている場合も、すべてのソース・ファイルをコンパイルする必要があります。

-qtemplateregistry コンパイラー・オプションの使用

テンプレート・レジストリーでは、「先着順サービス」のアルゴリズムが使用されます。

- コンパイラーが暗黙的なインスタンス生成を初めて実行するとき、そのプログラムのインスタンスは、それが発生するコンパイル単位でインスタンス生成されます。
- 別のコンパイル単位が同じ暗黙のインスタンス生成を実行すると、そのコンパイル単位のインスタンスは生成されません。つまり、プログラム全体で生成されるコピーは 1 つだけです。

インスタンス生成情報は、テンプレート・レジストリー・ファイルに保管されます。1 つのプログラムでは、同じテンプレート・レジストリー・ファイルを使用しなければなりません。2 つのプログラムで、テンプレート・レジストリー・ファイルを共用することはできません。

テンプレート・レジストリー・ファイルのデフォルトのファイル名は `templateregistry` ですが、他の有効なファイル名を指定して、このデフォルト名を

オーバーライドすることもできます。プログラム・ビルド環境を消去してから新たにビルドを開始する場合は、古いオブジェクト・ファイルとともにレジストリー・ファイルも削除してください。

同じテンプレート・レジストリー・ファイルを使用して、コンパイル時間に与える影響を最小にして、複数のコンパイルを並列に実行できます。

また、プログラムを再コンパイルするときは、テンプレートのインスタンス生成がないことが原因で、ソース・ファイルの再コンパイルがリンク・エラーを誘発する可能性があるかどうかを判断するために、テンプレート・レジストリー・ファイルの情報も使用されます。以下の条件が真である場合、ソース・ファイルを再コンパイルするときに、コンパイラーは 1 つ以上のソース・ファイルの再コンパイルをスケジュールに入れます。

- ソース・ファイルが、別のソース・ファイルがインスタンス化したテンプレートをインスタンス化した。
- テンプレートを実際にインスタンス化するために、テンプレート・レジストリーによってソース・ファイルが選択された。
- ソース・ファイルがテンプレートをインスタンス化しなくなった。

前述の条件がすべて真の場合、コンパイラーは別のソース・ファイルを選択して、テンプレートをインスタンス化します。そのファイルは、リンク・ステップ中に再コンパイルされるようにスケジュールに入れられます。リンク・ステップ中に再コンパイルされるようにスケジュールに入れられたソース・ファイルを再コンパイルした場合は、スケジュールに入れられた再コンパイルが取り消されます。

-qnotemplaterecompile を使用して、リンク・ステップ中にスケジュールに入れられた再コンパイルを無効にできます。詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の『-qtemplaterecompile (C++ のみ)』を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

-qtempinc (C++ のみ)

-qtemplaterecompile (C++ のみ)

-qtemplateregistry (C++ のみ)

関連コンパイル単位の再コンパイル

2 つのコンパイル単位、A と B が同じインスタンス生成を参照する場合、**-qtemplateregistry** コンパイラー・オプションを指定すると、次のような影響があります。

- A を最初にコンパイルすると、A のオブジェクト・ファイルにインスタンス生成のコードが含まれます。
- 次に B をコンパイルすると、B のオブジェクト・ファイルにはインスタンス生成のコードは含まれません。オブジェクト A に既に含まれているためです。
- あとで、このインスタンス生成を参照しないように A を変更すると、オブジェクト B の参照に、未解決のシンボル・エラーが発生します。A を再コンパイルすると、コンパイラーはこの問題を検出して、次のように処理します。

– **-qtemplaterecompile** コンパイラー・オプションが有効であれば、コンパイラーは A で指定したのと同じコンパイラー・オプションを使用して、リンク・ステップ時に自動的に B を再コンパイルします (ただし、個別のコンパイル・ステップとリンク・ステップを使用する場合は、リンク・ステップにコンパイル・オプションを組み込んで、B の正しいコンパイルを確認する必要があります)。
– **-qnotemplaterecompile** コンパイラー・オプションが有効であれば、コンパイラーが警告を出すので、B を手動で再コンパイルしてください。

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**


-qtemplateregistry (C++ のみ)


-qtemplaterecompile (C++ のみ)

-qtempinc から -qtemplateregistry への切り替え

-qtemplateregistry コンパイラー・オプションでは、アプリケーションのファイル構造にまったく制限がないため、その管理オーバーヘッドは、**-qtempinc** より少なくなります。次の方法で、切り替えを行うことができます。

- アプリケーションが **-qtempinc** でも **-qnotempinc** でも正常にコンパイルされる場合は、変更する必要はありません。
- アプリケーションが **-qtempinc** では正常にコンパイルされるが **-qnotempinc** ではコンパイルされない場合は、 **-qnotempinc** でも正常にコンパイルされるように、変更する必要があります。 __TEMPINC__ マクロが定義されていない場合は、個々のテンプレート定義ファイルに、条件付きで対応するテンプレート・インプリメンテーション・ファイルを組み込んでください。 32 ページの『-qtempinc の使用例』の図を参照してください。

明示的インスタンス生成宣言の使用 (C++11)

注: IBM は、C++11 (標準化前の呼称は C++0x) の一部の機能をサポートしています。IBM は引き続き、この標準の機能を開発および実装する予定です。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含めて、C++11 のすべての機能の実装を IBM が完了するまでは、リリースごとに実装が変更される可能性があります。IBM は、ソース、バイナリー、リストなどのコンパイラー・インターフェースにおいて、新しい C++11 機能の IBM 実装の旧リリースとの互換性を維持することを試みません。

構文的には、*明示的インスタンス生成宣言* は、先頭に `extern` キーワードが付いた *明示的インスタンス生成定義* です。「*XL C/C++ ランゲージ・リファレンス*」の『明示的インスタンス化 (C++ のみ)』を参照してください。

明示的インスタンス生成宣言機能を使用する場合は、以下の点を考慮してください。

- IBM クラス・テンプレート特殊化の明示的インスタンス生成宣言では、 前述の特殊化の暗黙的インスタンス生成を行いません。 IBM
- IBM 変換単位において、ユーザー定義のインライン関数が明示的インスタンス生成宣言の対象だが明示的インスタンス生成定義の対象でない場合、コンパイ

ラー・オプション **-qkeepinlines** が指定されているかどうかにかかわらず、その関数の一致しないコピーがその変換単位内で生成されることはありません。

注: この規則は、コンパイラーで暗黙的に生成される関数の動作を制限しません。デフォルト・コンストラクター、コピー・コンストラクター、デストラクター、およびコピー割り当て演算子などの暗黙的に宣言された特殊メンバーはインラインであり、コンパイラーがそれらをインスタンス化することがあります。具体的には、その関数の一致しないコピーが生成される場合があります。

IBM

- インラインでない関数で明示的インスタンス生成宣言の対象である関数に対して、インライン化が行われる量が低下する場合があります。
- 非純粋仮想メンバー関数が明示的インスタンス生成宣言の対象である場合、直接にまたはそのクラスを介して、プログラム全体のどこかで仮想メンバー関数が明示的インスタンス生成定義の対象となる必要があります。そうしないと、リンク時に未解決のシンボル・エラーが発生する可能性があります。
- クラス・テンプレート特殊化の暗黙のインスタンス生成が可能な場合、そのクラス特殊化の全仮想メンバー関数の暗黙のインスタンス生成が生じるかのようにユーザー・プログラムを作成する必要があります。そうしないと、リンク時に仮想メンバー関数について未解決のシンボル・エラーが発生する可能性があります。
- クラス・テンプレート特殊化について、その暗黙のインスタンス生成が可能であり、明示的インスタンス生成宣言の対象である場合、ユーザー・プログラム内のどこかで、そのクラス・テンプレート特殊化が明示的インスタンス生成定義の対象となる必要があります。そうしないと、リンク時に未解決のシンボル・エラーが発生する可能性があります。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qtempinc (C++ のみ)

#pragma implementation (C++ のみ)

-qlanglvl

「*XL C/C++ ランゲージ・リファレンス*」の関連情報

明示的インスタンス生成 (C++ のみ)

第 7 章 ライブラリーの構成

C および C++ アプリケーションには、静的および共用ライブラリーを組み込むことができます。

『ライブラリーのコンパイルとリンク』では、ソース・ファイルをコンパイルしてオブジェクト・ファイルを作成し、ライブラリーに組み込む方法、ライブラリーをメインプログラムにリンクする方法、およびあるライブラリーを別のライブラリーにリンクする方法について説明します。

38 ページの『ライブラリー内の静的オブジェクトの初期化 (C++)』 では、優先順位を使用して、C++ アプリケーションに含まれる複数のファイルのオブジェクト初期化の順序を制御する方法について説明します。

ライブラリーのコンパイルとリンク

このセクションでは、ソース・ファイルをコンパイルしてオブジェクト・ファイルを作成し、ライブラリーに組み込む方法、ライブラリーをメインプログラムにリンクする方法、およびあるライブラリーを別のライブラリーにリンクする方法について説明します。

関連情報:

動的および静的リンク

静的ライブラリーのコンパイル

静的ライブラリーをコンパイルするには、次のようにします。

1. 各ソース・ファイルをコンパイルして、リンクを持たないオブジェクト・ファイルを作成する。次に例を示します。

   ```
   xlc -c test.c example.c
   ```

2. ar コマンドを実行して、生成されたオブジェクト・ファイルを、アーカイブ・ライブラリー・ファイルに追加する。 次に例を示します。

   ```
   ar -rv libex.a test.o example.o
   ```

共用ライブラリーのコンパイル

共用ライブラリーをコンパイルするには:

1. ソース・ファイルをコンパイルして、リンクを持たないオブジェクト・ファイルを作成します。共用ライブラリーのコンパイルの場合は、**-qpic** コンパイラー・オプションも使用されることに注意してください。次に例を示します。

   ```
   xlc -qpic -c foo.c
   ```

2. **-qmkshrobj** コンパイラー・オプションを使用し、生成されたオブジェクト・ファイルから共用オブジェクトを作成します。次に例を示します。

   ```
   xlc -qmkshrobj -o libfoo.so foo.o
   ```

   「*XL C/C++ コンパイラー・リファレンス*」の関連情報

© Copyright IBM Corp. 1996, 2014

-qpic

-qmkshrobj

ライブラリーとアプリケーションとのリンク

静的ライブラリーまたは共用ライブラリーをメインプログラムにリンクするには、同じコマンド・ストリングを使用することができます。次に例を示します。

```
xlc -o myprogram main.c -Ldirectory [-Rdirectory] -ltest
```

ここで、*directory* は、ライブラリー `libtest.a` が含まれるディレクトリーのパスです。

-l オプションを使用して、**-L** オプション (共用ライブラリーの場合は **-R** オプションも) で指定したディレクトリー内で `libtest.so` を検索するようリンカーに指示しますが、これが見つからないと、リンカーは `libtest.a` を検索します。その他のリンケージ・オプション (デフォルトの動作を変更するオプションなど) については、オペレーティング・システム **ld** の資料 を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-l

-L

-R

共用ライブラリー間のリンク

モジュールをアプリケーションにリンクするのと同様、共用ライブラリー同士をリンクすれば、その間に依存関係を作成することができます。次に例を示します。

```
xlc -qmkshrobj -o mylib.so myfile.o -Ldirectory -Rdirectory -ltest
```

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qmkshrobj

-R

-L

ライブラリー内の静的オブジェクトの初期化 (C++)

C++ 言語定義は、C++ プログラムの `main` 関数を実行する前に、そのプログラムに組み込まれたすべてのファイルから、コンストラクターを持つすべての非ローカル・オブジェクトが適切に構成されるよう指定します。言語定義は、ファイル内のこれらのオブジェクトの初期化順序 (これは、そのオブジェクトが宣言された順序に従います) を指定しますが、複数のファイルやライブラリー間のオブジェクトの初期化順序は指定しません。プログラム内のさまざまなファイルやライブラリーで宣言された静的オブジェクトの初期化順序を指定することもできます。

オブジェクトの初期化順序を指定するには、オブジェクトに相対的な優先順位番号を割り当てます。ファイル全体や、ファイル内のオブジェクトの優先順位を指定できるメカニズムについては、 『オブジェクトへの優先順位の割り当て』で説明します。複数のモジュール間でオブジェクトの初期化順序を制御できるメカニズムについては、 41 ページの『ライブラリー間のオブジェクト初期化の順序』で説明します。

関連情報:

『オブジェクトへの優先順位の割り当て』

41 ページの『ライブラリー間のオブジェクト初期化の順序』

オブジェクトへの優先順位の割り当て

単一ライブラリー内のオブジェクトおよびファイルには、優先順位番号を割り当てることができます。オブジェクトは、その優先順位に従って実行時に初期化されます。ただし、モジュールのロード方法が異なり、オブジェクトが異なるプラットフォーム上で初期化されるため、優先順位を割り当てられるレベルは、次のように、プラットフォームによって違います。

ファイル全体に優先順位を設定する

この方法を使用するには、コンパイル時に **-qpriority** コンパイラー・オプションを指定します。デフォルトでは、単一ファイル内のオブジェクトはすべて同じ優先順位に割り当てられ、宣言された順序で初期化され、宣言とは逆の順序で終了します。

ファイル内のオブジェクトに優先順位を設定する

この方法を使用するには、ソース・ファイルに **#pragma priority** ディレクティブを組み込みます。個々の **#pragma priority** ディレクティブは、別の pragma ディレクティブが指定されるまで、そのあとに続くすべてのオブジェクトに優先順位を設定します。ファイル内では、最初の **#pragma priority** ディレクティブは、 **-qpriority** オプション (使用される場合は) で指定される番号より大きくしなければなりません。そしてそれ以後の **#pragma priority** ディレクティブの番号は、昇順にする必要があります。単一ファイル内のオブジェクトの相対優先順位は、そのオブジェクトの宣言順序のままですが、pragma ディレクティブは、オブジェクトが複数ファイル間で初期化される場合の順序に影響を与えます。オブジェクトは、その優先順位に従って初期化され、その逆の順序で終了します。

個々のオブジェクトの優先順位を設定する

この方法を使用するには、ソース・ファイルで、`init_priority` 変数属性を使用します。 `init_priority` 属性は、 **#pragma priority** ディレクティブより優先され、任意の宣言順序でオブジェクトに適用できます。 Linux では、オブジェクトは優先順位に従って初期化され、いくつかのコンパイル単位にわたって、その逆の順序で終了します。

優先順位番号の使用

優先順位番号の範囲は 101 ～ 65535 です。指定できる最小の優先順位番号は 101 で、この番号のものが最初に初期化されます。最大の優先順位番号は 65535 であり、この番号のものが最後に初期化されます。優先順位が指定されていない場合、デフォルトの優先順位は 65535 になります。

以下の例は、単一ファイル内のオブジェクト、および 2 つのファイル間のオブジェクトの優先順位を指定する方法を示したものです。 41 ページの『ライブラリー間のオブジェクト初期化の順序』 には、オブジェクトの初期化順序に関する詳細情報が記載されています。

ファイル内のオブジェクトの初期化の例

次の例は、ソース・ファイル内のいくつかのオブジェクトの優先順位の指定方法を示しています。

```
...
#pragma priority(2000) //Following objects constructed with priority 2000
...

static Base a ;

House b ;
...
#pragma priority(3000) //Following objects constructed with priority 3000
...

Barn c ;
...
#pragma priority(2500) // Error - priority number must be larger
                       // than preceding number (3000)
...
#pragma priority(4000) //Following objects constructed with priority 4000
...

Garage d ;
...
```

複数ファイル間のオブジェクト初期化の例

次の例は、farm.C および zoo.C の 2 つのファイル内のオブジェクトの初期化の順序を説明しています。 2 つのファイルはともに同じ共用モジュールに含まれていて、**-qpriority** コンパイラー・オプションおよび **#pragma priority** ディレクティブを使用します。

farm.C -qpriority=1000

```
...
Dog a ;
Dog b ;
...
#pragma priority(6000)
...
Cat c ;
Cow d ;
...
#pragma priority(7000)
Mouse e ;
...
```

zoo.C -qpriority=2000

```
...

Bear m ;
...
#pragma priority(5000)
...
Zebra n ;
Snake s ;
...
#pragma priority(8000)
Frog f ;
...
```

実行時には、これらのファイル内のオブジェクトは、次の順序で初期化されます。

シーケンス	オブジェクト	優先順位の値	コメント
1	Dog a	1000	option の優先順位 (1000) を使用。
2	Dog b	1000	同じ優先順位で続く。

シーケンス	オブジェクト	優先順位の値	コメント
3	Bear m	2000	option の優先順位 (2000) を使用。
4	Zebra n	5000	pragma の優先順位 (5000) を使用。
5	Snake s	5000	同じ優先順位で続く。
6	Cat c	6000	次の優先順位番号。
7	Cow d	6000	同じ優先順位で続く。
8	Mouse e	7000	次の優先順位番号。
9	Frog f	8000	次の優先順位番号 (最後に初期化)。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報


-qpriority / #pragma priority (C++ のみ)


-qmkshrobj

「*XL C/C++ ランゲージ・リファレンス*」の関連情報


init_priority 変数属性

ライブラリー間のオブジェクト初期化の順序

静的ライブラリーおよび共用ライブラリーはそれぞれ、そのすべての従属関係がロードされ、初期化されると、実行時に逆リンク順にロードされ、初期化されます。リンク順とは、個々のライブラリーがメインアプリケーションへのリンク中にコマンド・ラインにリストされた順序のことです。例えば、ライブラリー A がライブラリー B を呼び出す場合、ライブラリー B はライブラリー A より前にロードされています。

個々のモジュールがロードされると、オブジェクトは、 39 ページの『オブジェクトへの優先順位の割り当て』で概説した規則に従って、優先順位の順序で初期化されます。オブジェクトに優先順位が割り当てられていない場合、あるいは同じ優先順位が割り当てられている場合は、オブジェクト・ファイルはリンク順の逆順で初期化され (リンク順とは、ライブラリーへのリンクの際にコマンド・ラインでファイルを指定した順序のことです)、そのファイル内のオブジェクトは宣言の順序に従って初期化されます。オブジェクトは、その構成とは逆の順序で終了されます。 .

複数ライブラリー間のオブジェクト初期化の例

この例では、以下のモジュールが使用されています。

- `main.out` は、main 関数を含む実行可能モジュールです。
- `libS1` と `libS2` は、どちらも共用ライブラリーです。
- `libS3` と `libS4` は、どちらも共用ライブラリーで、 `libS1` と依存関係にあります。
- `libS5` と `libS6` は、どちらも共用ライブラリーで、 `libS2` と依存関係にあります。

ソース・ファイルは下記のコマンド・ストリングでオブジェクト・ファイルにコンパイルされる。

```
xlC -qpriority=101 -c fileA.C -o fileA.o
xlC -qpriority=150 -c fileB.C -o fileB.o
xlC -c fileC.C -o fileC.o
xlC -c fileD.C -o fileD.o
xlC -c fileE.C -o fileE.o
xlC -c fileF.C -o fileF.o
xlC -qpriority=300 -c fileG.C -o fileG.o
xlC -qpriority=200 -c fileH.C -o fileH.o
xlC -qpriority=500 -c fileI.C -o fileI.o
xlC -c fileJ.C -o fileJ.o
xlC -c fileK.C -o fileK.o
xlC -qpriority=600 -c fileL.C -o fileL.o
```

従属ライブラリーは、次のコマンド・ストリングによって作成されます。

```
xlC -qmkshrobj -o libS3.so fileE.o fileF.o
xlC -qmkshrobj -o libS4.so fileG.o fileH.o
xlC -qmkshrobj -o libS5.so fileI.o fileJ.o
xlC -qmkshrobj -o libS6.so fileK.o fileL.o
```

従属ライブラリーは、次のコマンド・ストリングによって親ライブラリーとリンクされます。

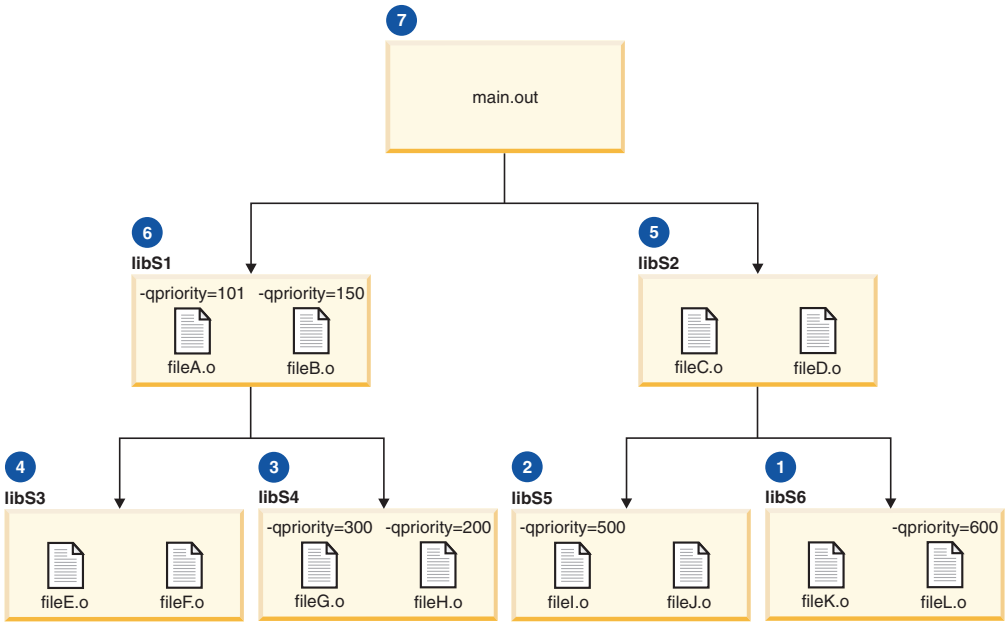
```
xlC -qmkshrobj -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
xlC -qmkshrobj -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

親ライブラリーは、次のコマンド・ストリングによってメインプログラムとリンクされます。

```
xlC main.C -o main.out -L. -R. -lS1 -lS2
```

次の図は、共用ライブラリーの初期化順序を示したものです。

図 1. Linux 上のオブジェクトの初期化順序



オブジェクトは、次のように初期化されます。

シーケンス	オブジェクト	優先順位の値	コメント
1	libS6	適用外	libS2 は、main とリンクされる際、コマンド・ラインで最後に入力されました。したがって、 libS1 より前に初期化されます。ただし、libS5 および libS6 は libS2 と依存関係にあるので、この両者が最初に初期化されます。libS6 は、libS2 を作成するためにリンクされたときコマンド・ラインで最後に入力されたので、最初に初期化されます。このライブラリーのオブジェクトは、その優先順位に従って初期化されます。
2	fileL	600	fileL 内のオブジェクトは次に初期化されます (このモジュール内で最も低い優先順位番号)。
3	fileK	65535	fileK 内のオブジェクトは次に初期化されます (このモジュール内で次の優先順位番号 (デフォルト優先順位は 65535))。
4	libS5	適用外	libS5 は、libS2 とリンクされる際、コマンド・ラインで libS6 より前に入力されたため、次に初期化されます。このライブラリーのオブジェクトは、その優先順位に従って初期化されます。
5	fileI	500	fileI 内のオブジェクトは次に初期化されます (このモジュール内で最も低い優先順位番号)。
6	fileJ	65535	fileJ 内のオブジェクトは次に初期化されます (このモジュール内で次の優先順位番号 (デフォルト優先順位は 65535))。
7	libS4	適用外	libS4 は、libS1 に従属していて、libS1 を作成するためにリンクされたときコマンド・ラインで最後に入力されたので、次に初期化されます。このライブラリーのオブジェクトは、その優先順位に従って初期化されます。
8	fileH	200	fileH 内のオブジェクトは次に初期化されます (このモジュール内で最も低い優先順位番号)。
9	fileG	300	fileG 内のオブジェクトは次に初期化されます (このモジュール内で次の優先順位番号)。
10	libS3	適用外	libS3 は、libS1 と依存関係にあり、 libS1 とリンクする際に、コマンド・ラインで最初に入力されたため、次に初期化されます。このライブラリーのオブジェクトは、その優先順位に従って初期化されます。
11	fileF	65535	fileF と fileE には、ともにデフォルトの優先順位である 65535 が割り当てられます。ただし、fileF はオブジェクト・ファイルが libS3 にリンクされたとき最後にコマンド・ラインにリストされたため、fileF が最初に初期化されます。
12	fileE	65535	次に初期化。
13	libS2	適用外	libS2 は次に初期化されます。このライブラリーのオブジェクトは、その優先順位に従って初期化されます。

シーケンス	オブジェクト	優先順位の値	コメント
14	fileD	65535	`fileD` と `fileC` には、ともにデフォルトの優先順位である 65535 が割り当てられます。ただし、`fileD` はオブジェクト・ファイルが `libS2` にリンクされたとき最後にコマンド・ラインにリストされたため、`fileD` が最初に初期化されます。
15	fileC	65535	次に初期化。
16	libS1		`libS1` は次に初期化されます。このライブラリーのオブジェクトは、その優先順位に従って初期化されます。
17	fileA	101	`fileA` 内のオブジェクトは次に初期化されます (このモジュール内で最も低い優先順位番号)。
18	fileB	150	`fileB` 内のオブジェクトは次に初期化されます (このモジュール内で次の優先順位番号)。
19	main.out	適用外	最後に初期化されます。`main.out` のオブジェクトは、その優先順位に従って初期化されます。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qmkshrobj

-W

第 8 章 アプリケーションの最適化

XL コンパイラーは、多層 PowerPC アーキテクチャーを活用した包括的な一連のパフォーマンス向上方法を提供することで、ハイパフォーマンスな 32 ビットおよび 64 ビットのアプリケーションの開発を可能にします。これらのパフォーマンス上の利点は、すぐれたプログラミング手法、完璧なテストおよびデバッグ、そして最適化とチューニングに基づいています。

最適化と調整の区別

最適化と調整を別々に、または組み合わせて使用することで、アプリケーションのパフォーマンスを向上させることができます。これらの違いを理解することが、さまざまなレベル、設定、および技法がどのようにしてパフォーマンスを向上させるかを理解する第一歩です。

最適化

最適化とは、ソース・コードを再構成する機会を探すコンパイラー主導プロセスで、開発時間に大きな影響を及ぼすことなく、アプリケーション全体の実行時パフォーマンスを向上させます。コンパイラー・オプションおよびディレクティブを使用して制御する XL コンパイラー最適化スイートは、徹底したデバッグおよびテスト・プロセスをすでに経た、しっかりしたソース・コードに最適のものです。この最適化変換では以下のことを行うことができます。

- アプリケーションが重要な演算のために実行する命令の数を減らす。
- オブジェクト・コードを再編成して、PowerPC アーキテクチャーの使用を最適化する。
- メモリー・サブシステムの使用率を改善する。
- このアーキテクチャーの能力を活用して大容量の共用メモリー並列化を扱う。

それぞれの基本的な最適化技法はパフォーマンスを向上させることができますが、すべての最適化がすべてのアプリケーションに恩恵をもたらすわけではありません。アプリケーションのパフォーマンスを向上させるために使用できる一般的な手順の概要については、46 ページの『最適化プロセスのステップ』を参照してください。

調整

最適化では、すべてのサポート対象環境内のすべてのアプリケーションのパフォーマンスを向上させるように設計された一般的な変換が適用されますが、調整では、お使いのアプリケーションの特定の特性やターゲット実行環境を調整して、そのパフォーマンスを向上させることができます。たとえ最適化レベルは低くても、アプリケーションおよびターゲット・アーキテクチャーのための調整は、パフォーマンスにプラスの影響をもたらすことができます。適切な調整を行えば、コンパイラーは次のことを行うことができます。

- より効率的なマシン・インストラクションを選択する。

© Copyright IBM Corp. 1996, 2014

- アプリケーションにとってより適切な命令シーケンスを生成する。
- より集中的な最適化を選択してコードを向上させる。

詳細な説明については、 53 ページの『システム・アーキテクチャーのための調整』を参照してください。

最適化プロセスのステップ

最適化プロセスを開始するに当たっては、すべての最適化手法がすべてのアプリケーションに適合するわけではないことを忘れないでください。場合によっては、コンパイル時間の増加、デバッグ能力の減少、および最適化が提供できる改善点の間にトレードオフが生じることがあります。

最善のパフォーマンスを達成する一方で、さまざまな最適化手法を知って試してみることが、XL コンパイラー・アプリケーションの適正なバランスを取るのに役立ちます。また、コードを手動で最適化する必要はありませんが、最適化プロセスにはコンパイラーに適したプログラミングが大いに役立ちます。特異な構文は、アプリケーションの特性を覆い隠し、パフォーマンスの最適化を困難にすることがあります。このセクションで説明するステップを参考にして、アプリケーションの最適化を行ってください。

1. 『基本最適化』ステップは、レベル 0 および 2 の最適化プロセスを開始します。
2. 『拡張最適化』ステップでは、より強固な最適化レベル 3、4 および 5 でアプリケーションの最適化を実施します。
3. 『上位ループ分析および変換の使用』ステップは、ループ実行時間を制限するのに役立ちます。
4. 『プロシージャー間分析の使用』ステップは、アプリケーション全体を一度に最適化できます。
5. 『プロファイル指示フィードバックの使用』ステップは、最適化をアプリケーションの特定の特性に集中的に適用します。
6. 『最適化コードのデバッグ』ステップは、最適化されたコードで発生する可能性のある問題の識別に役立ちます。

基本最適化

XL コンパイラーは、いくつかの最適化レベルをサポートしています。オプション・レベルが上がるごとに、その下のレベルを基礎として徐々に積極的な変換が適用され、その結果、より多くのマシン・リソースが使用されます。

必ず、まず低い最適化レベルでアプリケーションを適切にコンパイルして実行してから、その上でより積極的な最適化を試みるようにしてください。このトピックでは、2 つの最適化レベルについて説明しますが、下の「*基本最適化*」表にコンパイルの補完オプションのリストを示してあります。この表の後ろの欄では、その最適化レベルで、アプリケーションによってはパフォーマンスの改善効果がある場合があるコンパイラー・オプションも示しています。

表 12. 基本最適化

最適化レベル	デフォルトで暗黙指定される追加オプション	補足オプション	有効と考えられるその他のオプション
-O0	なし	**-qarch**	なし
-O2	**-qmaxmem=**8192	**-qarch** **-qtune**	**-qmaxmem=**-1 **-qhot=level=0**

レベル 0 での最適化

レベル 0 での利点

- マシン・リソースへの影響が最小である最小限のパフォーマンス向上。
- デバッグ・プロセスに役立ついくつかのソース・コード問題を明らかにします。

コンパイラーがデフォルトですでに指定している **-O0** で最適化プロセスを開始します。このレベルでは、明らかに冗長なコードを除去して基本的な分析に基づく最適化を行い、コンパイル時間では良好な結果を得ることができます。また、ユーザー・コードがアルゴリズム的に正しいことを確認し、さらに複雑な最適化に進むことができるようになります。**-O0** には、一部の冗長な命令の除去と、定数の折り畳みも含まれます。オプション **-qfloat=nofold** を使用すれば、浮動小数点の定数結合のみを抑制することができます。このレベルできちんと最適化を行うことにより、デバッグ用情報がすべて保存され、変数が初期化されていない、キャストが適切でないなどの現在のコードの問題を明らかにすることができます。

さらに、このレベルで **-qarch** を指定すると、アプリケーションのターゲットが特定のマシンに定められるため、アプリケーションが適用可能なすべての構造上の利点を利用するようにすることで、大幅にパフォーマンスを向上させることができます。

注: SMP プログラムについては、追加のオプション **-qsmp=noopt** を追加する必要があります。

調整の詳細については、 53 ページの『システム・アーキテクチャーのための調整』を参照してください。

レベル 2 での最適化

レベル 2 での利点

- 重複コードを除去します
- 基本ループ最適化
- **-qarch** および **-qtune** 設定を利用するようにコードを構造化できます

-O0 を使用して、アプリケーションを正常にコンパイル、実行、およびデバッグした後、**-O2** で再コンパイルすると、サブプログラムまたはコンパイル単位のスコープに適用され、何らかのインライン化を含むことができる一連の包括的な低レベル変換にアプリケーションが開放されます。 **-O2** での最適化は、パフォーマンスを大きくする一方でコンパイル時間およびシステム・リソースへの影響を制限する相対的なバランスです。 **-qmaxmem** オプションの値を大きくすることによって、**-O2** ポートフォリオ内の一部の最適化に利用できるメモリーを増やすことができま

す。**-qmaxmem=-1** を指定すると、最適化プログラムは制限を確認することなく必要に応じてメモリーを使用できるようになりますが、最適化プログラムが **-O2** でアプリケーションに適用する変換は変更されません。

C では、アプリケーションがライブラリー関数と同じ名前の関数を定義していない限り、**-qlibansi** でコンパイルしてください。**-O2** を指定して問題が発生する場合は、最適化を無効にする代わりに、**-qalias=noansi** を使用することを検討してください。

また、C コード内のポインターは次の型制限に従うようにしてください。

- 汎用ポインターには `char*` または `void*` が可能である
- すべての共用変数と共用変数を指すポインターには `volatile` を指定する

O2 での調整の開始

-O2 以上では、正しいハードウェア・アーキテクチャー・ターゲットまたはターゲットのファミリーを選択することが一層重要になります。適切なハードウェアをターゲットにすれば、最適化プログラムは使用可能なハードウェア機構を最大限に活用できます。ハードウェア・ターゲットのファミリーを選択すれば、**-qtune** オプションを使用して、アーキテクチャー選択と一致するコードを出力するようコンパイラーに指示できますが、このオプションは選択された調整ハードウェア・ターゲットで最適に実行されます。このオプションを使用すると、汎用ターゲットのセット向けにコンパイルできますが、コードは特定のターゲットで最適に実行させることができます。

-qarch および **-qtune** オプションの詳細については、53 ページの『システム・アーキテクチャーのための調整』セクションを参照してください。

-O2 オプションでは、次のような多数の追加最適化を実行できます。

- 共通副次式の除去: 重複した命令を除去します。
- 定数の伝搬: 定数式をコンパイル時に評価します。
- デッド・コードの除去: 特定の制御フローが到達しない命令や、使用されない結果を生成する命令を除去します。
- 不要格納の除去: 不必要な変数割り当てを除去します。
- グラフ色分けレジスターの割り振り: ユーザー変数を全体的にレジスターに割り当てます。
- 値の番号付け: 重複計算を除去することによって代数式を簡略化します。
- ターゲット・マシンの命令スケジューリング。
- ループのアンロールとソフトウェアのパイプライン
- ループ・インバリアント・コードをループの外に移動。
- 制御フローの単純化。
- アドレス・モードの強度縮小および有効利用。
- 隣接するロード/ストアとその他の演算をマージする拡張。
- その他の最適化を強化するためのポインター別名割り当ての改善。

-O2 を指定して最適化を行う場合でも、**-g** を指定すると、ユーザー・ソース・コードに関する有用な情報がデバッガーに提供されます。より高い **-g** レベルを使用す

ると、デバッガーに提供される情報量が増えますが、実行可能な最適化が少なくなります。 逆に、高い最適化レベルでは、コード変換の度合いが高すぎて、デバッグ情報が正確ではなくなる場合があります。そのような情報は慎重に使用してください。

拡張最適化

最適化レベルが高いほどパフォーマンスに大きな影響を及ぼすことができますが、コード・サイズ、コンパイル時間、リソース要件、および数値やアルゴリズムの精度の観点からすると何らかのトレードオフが生じる可能性があります。

46 ページの『基本最適化』を適用して、ユーザー・アプリケーションを正常にコンパイルおよび実行した後に、さらに強力な最適化ツールを適用することができます。XL コンパイラー最適化ポートフォリオには拡張最適化を指示するオプションが多数含まれており、アプリケーションが受ける変換はほとんど制御できます。 表 13 に示されている各最適化レベルの説明には、パフォーマンス上の利点に関する情報だけでなく、考えられるトレードオフや、アプリケーションに対する最適な解決策を見つけるよう最適化プログラムに指示するのに役立つ情報も含まれています。

表 13. 拡張最適化

最適化レベル	暗黙指定される追加オプション	補足オプション	有効と考えられるオプション
-O3	**-qnostrict** **-qmaxmem=-1** **-qhot=level=0**	**-qarch** **-qtune**	**-qpdf**
-O4	**-qnostrict** **-qmaxmem=-1** **-qhot** **-qipa** **-qarch=auto** **-qtune=auto** **-qcache=auto**	**-qarch** **-qtune** **-qcache**	**-qpdf** **-qsmp=auto**
-O5	**-O4** のすべて **-qipa=level=2**	**-qarch** **-qtune** **-qcache**	**-qpdf** **-qsmp=auto**

プログラムを下記のいずれかのオプション・セットを使ってコンパイルするときは、

- **-qhot -qignerrno -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

コンパイラーは、Mathematical Acceleration Subsystem ライブラリー (MASS) 内の同等のベクトル関数を呼び出すことにより、自動的にシステム数学関数の呼び出しのベクトル化を試行します。ただし、関数 `vdnint`、`vdint`、`vcosisin`、`vscosisin`、`vqdrt`、`vsqdrt`、`vrqdrt`、`vsrqdrt`、`vpopcnt4`、および `vpopcnt8` を除きます。ベクトル化できない場合、コンパイラーは自動的に等価 MASS スカラー関数の呼び出し

を試行します。自動ベクトル化または自動スカラー化の場合、コンパイラーはシステム・ライブラリー libxlopt.a に含まれている MASS 関数のバージョンを使用します。

前述のオプション・セットに加えて、**-qipa** オプションが有効になっているときにコンパイラーがベクトル化できないと、MASS スカラー関数のインライン化を試行してから呼び出しを決定します。

レベル 3 での最適化

レベル 3 での利点

- 詳細なメモリー・アクセス分析
- 優れたループ・スケジューリング
- 上位ループ分析および変換 (**-qhot=level=0**)
- デフォルトでのコンパイル単位内の小さいプロシージャーのインライン化
- 暗黙的なコンパイル時メモリー使用量制限の除去

-O3 を指定すると、**-O2** に存在する制限の多くを除去する、より強力な低レベル変換が開始されます。例えば、最適化プログラムは、デフォルトとして **-qmaxmem=-1** を使用することによって、メモリー制限を検査しなくなります。さらに、最適化はより大きなプログラム領域を網羅するとともに、より詳細な分析を試みようとします。最適化プログラムがある程度のパフォーマンス増大を提供する機会はすべてのアプリケーションに含まれているわけではありませんが、ほとんどのアプリケーションはこの種の分析から恩恵を受けることができます。

レベル 3 における潜在的なトレードオフ

-O3 の詳細な分析では、コンパイル時間とメモリー・リソースの観点からトレードオフが生じます。また、**-O3** は **-qnostrict** を暗黙指定するため、最適化プログラムは実行速度を得ようとしてアプリケーション内のある種の浮動小数点のセマンティクスを変更することがあります。これは一般に次のような精度のトレードオフを伴います。

- 浮動小数点計算の順序変更。
- ゼロ除算やオーバーフローなど潜在的な例外の順序変更または除去。
- 計算結果の精度がやや落ちる、または NaN と無限大を同じ方法で処理しない、代替の計算方法の使用。

それにもかかわらず、**-qstrict** を指定することによって、正確な浮動小数点セマンティクスを保持している間は、**-O3** の大部分の利点を得ることができます。浮動小数点計算において **-O0**、**-O2** または **-qnoopt** の結果で得るのと同様の絶対的な精度を要求する場合は、**-qstrict** でコンパイルする必要があります。オプション **-qstrict=ieeefp** はまた、浮動小数点演算に関するすべての IEEE セマンティクスへの順守も確保します。ユーザー・アプリケーションが浮動小数点例外、または浮動小数点算術の評価順序に依存する場合は、 **-qstrict**、 **-qstrict=exceptions**、または **-qstrict=order** を指定してコンパイルすることにより、確実に正確な結果を得ることができます。浮動小数点の計算精度に関して **-qstrict=precision** サブオプション・グループの影響を考慮する必要もあります。精度サブオプション・グループには個別

サブオプションの、**subnormals**、**operationprecision**、**association**、**reductionorder**、および **library** があります (「*XL C/C++ コンパイラー・リファレンス*」の **-qstrict** オプションに記載)。

-qstrict を指定しなければ、どのソース・レベル演算についても計算の差は、 46 ページの『基本最適化』に比べて非常にわずかなものです。差が加法的になるループ構造に演算がある場合は、わずかな差でも複合することがありますが、ほとんどのアプリケーションは浮動小数点セマンティクスで発生する可能性のある変更には依存しません。

-O レベルの構文については、 「*XL C/C++ コンパイラー・リファレンス*」の『-O -qoptimize』を参照してください。

中間ステップ: レベル 3 での -qhot サブオプションの追加

-O3 では、パフォーマンスを大きくするため **level=0** の最小 **-qhot** ループ変換が最適化に含まれます。レベルを上げること、したがって **-qhot** の積極性を増すことで、さらにパフォーマンス上の利点を増やすことができます。サブオプションなしで **-qhot** を指定するか、**-qhot=level=1** を指定してみてください。

-qhot について詳しくは、 56 ページの『上位ループ分析および変換の使用』を参照してください。

一方、アプリケーションでループ処理配列 (**-qhot** によって向上) を使用しない場合、**-O3** の後に **-qnohot** を指定することにより、通常はパフォーマンスの低下を最小限に抑えながらコンパイル速度を大幅に高めることができます。

レベル 4 での最適化

レベル 4 での利点

- コンパイル単位間でのグローバルおよび引数値の伝搬
- コンパイル単位から別のコンパイル単位へのコードのインライン化
- グローバル・データ構造の再編成または除去
- 別名割り当て分析の精度の増加

-O4 での最適化は、プロシージャー間分析 (IPA) を行う **-qipa=level=1** を起動することによって **-O3** に基づいて構築され、アプリケーション全体が 1 単位として最適化されます。このオプションは、頻繁に使用される多数のルーチンを含むアプリケーションに特に関係があります。

IPA 最適化を最大限に活用するには、コンパイル時とリンク時の両方のステージでプロシージャー間分析が発生したとき、アプリケーション・ビルドのコンパイルおよびリンク・ステップで **-O4** を指定する必要があります。

レベル 4 における潜在的なトレードオフ

-O3 で既に解説したトレードオフに加えて、**-qipa** を指定すると、特にリンク・ステップで、コンパイル時間が著しく増加することがあります。

IPA プロセス

1. コンパイル時に最適化がファイル単位で発生するほか、リンク・ステージの準備も行われます。IPA は分析情報を、コンパイラーが作成するオブジェクト・ファイルに直接書き込みます。
2. リンク・ステージで、IPA はオブジェクト・ファイルから情報を読み取って、アプリケーション全体を分析します。
3. この分析により、最適化プログラムはアプリケーションの再作成と再構成および適切な **-O3** レベル最適化の適用が可能になります。

IPA サブオプションの詳細を含む IPA に関する詳しい情報は、『 60 ページの『プロシージャー間分析の使用』』セクションに記載されています。

-qipa を超えると、**-O4** は他の最適化オプションを使用可能にします。

- **-qhot**

  より積極的な HOT 変換を使用可能にして、ループ構成体および配列言語を最適化します。

- **-qarch=**auto および **-qtune=**auto

  ビルド・マシンと同じハードウェア・アーキテクチャーで実行するようにアプリケーションを最適化します。ビルド・マシンのアーキテクチャーがアプリケーションの実行環境と非互換である場合は、**-O4** オプションの後に別の **-qarch** サブオプションを指定する必要があります。これは **-qarch=auto** をオーバーライドします。

- **-qcache=auto**

  キャッシュ構成を、特定のハードウェア・アーキテクチャーで実行するように最適化します。 **auto** サブオプションは、ビルド・マシンのキャッシュ構成が実行アーキテクチャーの構成と同じであることを前提としています。キャッシュ構成を指定すると、プログラムのパフォーマンスが向上することがあります。特に、ループ操作の場合は、データ・キャッシュに一度に収まる量のデータのみを処理するように操作を制限してパフォーマンスを高めることができます。

  アプリケーションを別のマシン上で実行する場合は、適切なキャッシュ値を指定してください。

レベル 5 での最適化

レベル 5 での利点

- ほとんどの積極的最適化が可能です
- ループ最適化と IPA を最大限に活用します

最高の最適化レベルとして、**-O5** には **-O4** のすべての最適化が含まれ、**-qipa** レベルを 2 に上げることによってプログラム全体の分析が深化されます。また、**-O5** でコンパイルすると、最適化プログラムが別名割り当ての改善を追及する積極性も増します。さらに、C/C++ コードと、XL コンパイラーを使用してコンパイルする Fortran コードがアプリケーションに混在している場合は、**-O5** オプションでコンパイルおよびリンクすることによって、パフォーマンスを向上させることができます。

レベル 5 における潜在的なトレードオフ

-O5 でのコンパイルには、他のどの最適化レベルよりも多くのコンパイル時間とマシン・リソースが必要です (特に、IPA リンク・ステップに **-O5** を指定した場合)。**-O5** でのコンパイルは、**-O4** でアプリケーションを正常にコンパイルおよび実行した後、最適化プロセスの最終フェーズとして行ってください。

システム・アーキテクチャーのための調整

コンパイラーには、指定したマイクロプロセッサーまたはアーキテクチャー・ファミリーで、最適に実行されるコードを生成するように指示することができます。該当するターゲット・マシン用オプションを選択することで、可能な限り広範囲にわたるターゲット・プロセッサーや、指定したプロセッサー・アーキテクチャー・ファミリー内の一定範囲のプロセッサー、あるいは特定のプロセッサーをそれぞれ選択できるように、最適化することができます。

次の表は、ターゲット・マシンの個々の特徴に影響を与える最適化オプションのリストです。事前定義の最適化レベルを使用すると、それぞれのオプションに対するデフォルト値が設定されます。

表 14. ターゲット・マシンのオプション

オプション	振る舞い
-q32	32 ビット (4 バイトの int 型/4 バイトの long 型/4 バイトの pointer 型) アドレッシング・モデル用のコードを生成します (32 ビット実行モード)。これがデフォルトの設定値です。
-q64	64 ビット (4 バイトの int 型/ 8 バイトの long 型/ 8 バイトの pointer 型) アドレッシング・モデル用のコードを生成します (64 ビット実行モード)。
-qarch	命令コードを生成するプロセッサー・アーキテクチャー・ファミリーを選択します。 このオプションによって、生成される命令セットは PowerPC アーキテクチャー向け命令セットのサブセットに制限されます。すべての Linux ディストリビューションにおいてデフォルトは -qarch=ppc64grsq です。 -O4 または -O5 を使用すると、デフォルトが -qarch=auto に設定されます。このオプションの詳しい情報については、 54 ページの『ターゲット・マシンのオプションの最大活用』を参照ください。
-qtune	指定したマイクロプロセッサー上で実行するように、最適化にバイアスをかけます。この際、ターゲットとして使用する命令セット・アーキテクチャーにはまったく影響は及びません。このオプションの詳しい情報については、 54 ページの『ターゲット・マシンのオプションの最大活用』を参照ください。
-qcache	特定のキャッシュまたはメモリー形状を定義します。デフォルトは、-qtune の設定によって決まります。このオプションの詳しい情報については、 54 ページの『ターゲット・マシンのオプションの最大活用』を参照ください。

ハードウェア関連の有効なサブオプションおよびサブオプションの組み合わせの完全なリストについては、「*XL C/C++ コンパイラー・リファレンス*」の **-qtune** セ

クションの『*受け入れ可能な -qarch/-qtune の組み合わせ*』、および「*XL C/C++ コンパイラー・リファレンス*」の『*アーキテクチャー固有のコンパイラー・オプションの指定*』を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

-q32, -q64

-qarch

-qipa

-qtune

-qcache

アーキテクチャー固有の 32 ビットまたは 64 ビットのコンパイルでのコンパイラー・オプションの指定

ターゲット・マシンのオプションの最大活用

-qarch オプションの使用

-qarch コンパイラー・オプションを使用して、特定のマシン・アーキテクチャー用に最適化された命令を生成します。例えば、POWER8™ 用に最適化された命令を含むオブジェクト・コードを生成するには、**-qarch=pwr8** を使用します。アプリケーションのコンパイルと実行を同じマシン上で行う場合は、**-qarch=auto** オプションを指定して、コンパイルするマシンの特定のアーキテクチャーを自動的に検出し、そのマシンのみ (またはそれと同等のプロセッサー・アーキテクチャーをサポートするシステム) を対象とした命令を利用するコードを生成することができます。そうでない場合は、**-qarch** オプションを使用して、コードを合理的に実行できる最小限の可能なマシン・ファミリーを指定してください。

特定の機能をサポートするシステム・アーキテクチャー上でアプリケーションを実行する場合は、対応する **-qarch** サブオプションを指定して、そのシステム・アーキテクチャー用のオブジェクト・コードを生成する必要があります。例えば、POWER7®、POWER7+™、または POWER8 マシン上にアプリケーションをデプロイして、VMX ベクトル処理と大規模ページ・サポートを完全に活用するには、POWER7 または POWER7+ の場合は **-qarch=pwr7**、POWER8 の場合は **-qarch=pwr8** をコンパイル・マシンで指定する必要があります。**-qarch=auto** または **-qarch** を指定すると、必要なサポートが用意されません。ただし、アプリケーションを POWER7 と POWER8 の両方でデプロイする場合は、**-qarch** を最も低い共通のアーキテクチャーに設定します。これによって、アプリケーションには、アプリケーションをデプロイするすべてのプロセッサーに共通の命令だけが含まれます。この例では、最も低い共通のアーキテクチャーは POWER7 であるため、**-qarch=pwr7** を使用する必要があります。 **-qarch** とそのサブオプションについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の -qarch を参照してください。各 **-qarch** サブオプションによってサポートされる対応するシステム・アーキテクチャーについて詳しくは、-qarchの表『プロセッサー・アーキテクチャーでのサポート機能』を参照してください。

-qtune オプションの使用

-qtune コンパイラー・オプションを使用して、マシン・アーキテクチャー用に最適化された命令のスケジューリングを制御します。 **-qarch** を指定して特定のアーキテクチャーを指定すると、**-qtune** は、自動的に、そのアーキテクチャーで最高のパフォーマンスを出す命令シーケンスを生成するサブオプションを選択します。 **-qarch** を使用してアーキテクチャーのグループ を指定する場合は、**-qtune=auto** でコンパイルすると、指定したグループ内のすべてのアーキテクチャーで実行されるコードが生成されますが、命令シーケンスは、コンパイルするマシンのアーキテクチャーで最高のパフォーマンスを出すようになっています。

コンパイラーが最高のパフォーマンスを目指し、なおかつ、 **-qarch** オプションで指定したすべてのアーキテクチャー上に作成されたオブジェクト・ファイルを実行できるような特定のアーキテクチャーを指定するには、**-qtune** を試してみてください。 **-qarch** および **-qtune** の有効な組み合わせについては、「*XL C/C++ コンパイラー・リファレンス*」の **-qtune** セクションの『*受け入れ可能な -qarch/-qtune の組み合わせ*』を参照してください。

広範囲の PowerPC ハードウェアで実行される単一バイナリー・ファイルを作成する場合は、**-qtune=balanced** オプションの使用を検討してください。このオプションを使用すると、コンパイラーによる最適化の判断が、特定バージョンのハードウェアに向けられなくなります。代わりに、一般的に広範囲のハードウェアで役立つ機能を追加することにより、一部のハードウェアに障害をもたらす可能性がある最適化を回避する調整が行われます。

注: -qtune=balanced オプションを使用してコンパイルされたコードは、配布する前にパフォーマンスを確認する必要があります。

-qtune=balanced と、**-qtune=auto** などの他の **-qtune** サブオプションとの違いは、他のサブオプションを指定した場合、指定されたバージョンのハードウェア・アーキテクチャー用に最適化され、他では正常に実行されない可能性がある命令をコンパイラーが生成する点です。**-qtune=balanced** オプションを指定すると、コンパイラーは、さまざまな Power® ハードウェアで問題なく実行される命令を生成します。詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の -qtune を参照してください。

-qcache オプションの使用

-qcache オプションを使用する前に、まず **-qlistopt** オプションを指定して現行設定のリストを生成し、それで問題ないかどうかを確認します。独自の **-qcache** サブオプションを指定する場合は、これと一緒に **-qhot** または **-qsmp** も使用してください。サブオプションの完全セット、オプション構文、および使用のためのガイドラインについては、「*XL C/C++ コンパイラー・リファレンス*」の **-qcache** を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qhot

-qsmp

 -qcache

 -qlistopt

 -qarch

 -qtune

上位ループ分析および変換の使用

上位変換は、交換、融合、アンロールなどの手法を用いて、特にループのパフォーマンスを向上させる最適化です。

これらのループ最適化の目的は次のとおりです。

- キャッシュとアドレス変換検索バッファーを効果的に使用して、メモリー・アクセスのコストを削減する。
- ハードウェアによって提供されるデータの事前取り出し機能を有効に利用して、計算とメモリー・アクセスを並行させる。
- 相補的なリソース要件を持つ命令の使用を再配列および平衡化して、マイクロプロセッサー・リソースの使用率を改善する。
- SIMD ベクトル命令を生成する。
- ベクトル数学ライブラリー関数への呼び出しを生成する。

上位ループ分析および変換を使用可能にするには、最適化レベル **-O2** を暗黙指定する **-qhot** オプションを使用します。次の表は、**-qhot** で使用できるサブオプションのリストです。

表 15. -qhot のサブオプション

サブオプション	**振る舞い**
level=0	コンパイラーに上位の変換のサブセットを実行するように指示して、データの局所性を改善してパフォーマンスを向上させます。このサブオプションは、**-qhot=novector** および **-qhot=noarraypad** を暗黙指定します。**-O3** でコンパイルすると、このレベルは自動的に使用可能になります。
level=1	**-qhot** をサブオプションなしで指定すると、これはデフォルトのサブオプションになります。 **-O4** または **-O5** を使ってコンパイルすると、このレベルもまた自動的に使用可能になります。 これは、**-qhot=vector** を指定することと等価です。
level=2	**-qsmp** と一緒に使用すると、ネストされたループで、**-qhot=level=1** に加えて、追加の変換をいくつか実行することをコンパイラーに指示します。ループの分析と変換の結果によっては、より多くのキャッシュの再利用とループの並列化が生じる場合があります。

表 15. -qhot のサブオプション (続き)

サブオプション	振る舞い
vector	**-qnostrict** および **-qignerrno**、または **-O3** 以上の最適化レベルと一緒に指定されると、MASS ライブラリーに含まれる各種数学関数の最適化バージョンを使用してシステム・バージョンを使用しないように一部のループを変換するよう、コンパイラーに指示します。最適化バージョンは、パフォーマンスと、正確さや例外処理の観点でさまざまなトレードオフをもたらします。**-qhot** をサブオプションなしで指定すると、このサブオプションはデフォルトによって使用可能になります。また、**-O3** での **-qhot=vector** の指定は、**-qhot=level=1** を暗黙指定したことになります。
arraypad	コンパイラーに、メリットがあると推測される配列を、必要なだけ埋め込むように指示します。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qhot

-qstrict

-qignerrno

-qarch

-qsimd

-qhot の最大活用

以下に、**-qhot** を使用する場合の提案事項を挙げます。

- すべてのコードに対して、**-qhot** を **-O3** と併せて使用してみてください。このオプションは、変換を行う機会がない場合は、効果が出ない設計になっています。ただし、このオプションによってコンパイル時間が長くなり、プログラム内にループ処理のベクトルや配列が存在しない場合、ほとんどメリットが得られない可能性があります。この場合は、**-O3 -qnohot** を使用する方が、より適切と思われます。
- 自動インライン化とメモリー局所性の最適化によって、コードの実行時パフォーマンスに大きなメリットが期待できる場合は、**-O4** を **-qhot=level=0** または **-qhot=novector** と一緒に使用してみてください。
- コンパイル時間が許容できないほど長くなる場合は (これは、複雑にネストされたループで起こることがあります)、**-qhot=level=0** または **-qnohot** を試してください。
- コード・サイズが許容できないほど大きい場合は、インライン化のレベルを減らしてみるか、**-qcompact** を **-qhot** とともに使用してみてください。
- **-qhot** オプションを指定して一部のソース・ファイルをコンパイルし、**-qhot** オプションなしで残りのソース・ファイルをコンパイルすることができます。これによって、コンパイラーは、最適化が必要なコード部分だけを改善できます。
- **-qreport** と **-qsimd=auto** を併用して、ループ変換リストを生成してください。リスト・ファイルは、`LOOP TRANSFORMATION SECTION` のマークが付けられたセクシ

ョンでループがどのようにトランスフォームされたかを示します。ご使用のプログラム内のループがどのようにトランスフォームされるかについてのフィードバックとしてリスト情報を使用してください。このリスト情報に基づいて、コンパイラーがより効率的にループをトランスフォームできるように、ご使用のコードを調整することができます。例えば、このセクションのリストを使用して、ループのベクトル化を防ぐ可能性があるストライド 1 でない参照を識別できます。

- **-qreport** を **-qhot** または **-qhot** を暗黙指定する任意の最適化オプションと併用して、ネストされたループについての情報をリスト・ファイルの LOOP TRANSFORMATION SECTION に生成してください。さらに、**-qprefetch=assistthread** を使用してプリフェッチ支援スレッドを生成すると、レポートのこのセクションにメッセージ「データ・プリフェッチの支援スレッドが生成されました。」も表示されます。ループ・ネストで実行された積極的なループ変換および並列処理のリストをリスト・ファイルの LOOP TRANSFORMATION SECTION に生成するには、**-qhot=level=2** と **-qsmp** を **-qreport** と一緒に使用してください。
- **-qassert=refalign** を指定すると、コンパイル単位内のすべてのポインターが、もともとポインター型の長さを基準にして調整されているデータのみを指すことを、コンパイラーにアサートすることになります。このアサーションにより、コンパイラーはより効果的なコードを生成できます。このアサーションは、特に **-qsimd=auto** オプションと一緒に **-qhot=level=0** または **-qhot=level=1** が指定された SIMD アーキテクチャーをターゲットとする場合に役立ちます。

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

-qcompact

-qhot

-qsimd

-qprefetch

-qstrict

共用メモリーの並列処理 (SMP) の使用

大部分のIBM pSeries のマシンは、共用メモリーの並列処理ができます。 **-qsmp** でコンパイルすると、この機能の活用に必要なスレッド化されたコードを生成することができます。 **-qsmp** オプションは、**-qhot** オプションと **-O2** 以上の最適化レベルを暗黙指定します。

次の表は、最もよく使用されるサブオプションのリストです。すべてのサブオプションの説明と構文は、「*XL C/C++ コンパイラー・リファレンス*」の **-qsmp** にあります。自動並列処理と、OpenMP ディレクティブの概要については、『 137 ページの『第 12 章 プログラムの並列処理』』を参照してください。

表 16. 一般に使用される -qsmp のサブオプション

サブオプション	振る舞い
auto	コンパイラーに、可能ならユーザー支援なしに自動で並列コードを生成するように指示します。ソース・コード内のあらゆる SMP プログラミング構成 (OpenMP ディレクティブを含む) も認識されます。 **-qsmp** のサブオプションを指定しない場合は、これがデフォルト設定となり、このデフォルト設定で、 **opt** サブオプションも暗黙指定されます。
omp	コンパイラーに対し、OpenMP API の明示的な並列処理の指定について厳密な整合性を強制するように指示をします。 OpenMP 標準に準拠する言語構造体のみが認識されます。 **-qsmp=omp** と **-qsmp=auto** は、では互換性がありません。
opt	コンパイラーに、最適化と同時に並行処理も行うように指示します。最適化は、他の最適化オプションがない場合は、 **-O2 -qhot** に相当します。
noopt	すべての最適化がオフになります。開発作業中は、デバッグができるように最適化をオフにすると便利です。
fine_tuning	サブオプションの他の値は、スレッド・スケジューリングやロッキングなどの制御に使用されます。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-O, -qoptimize

-qsmp

-qhot

-qsmp の最大活用

以下に、**-qsmp** オプションを使用する場合の提案事項を挙げます。

- 自動並行処理で **-qsmp** を使用する前に、最適化と **-qhot** を単一スレッド方式で使用して、プログラムをテストしてください。
- OpenMP プログラムをコンパイルするけれど、自動並行処理は必要ないという場合は、**-qsmp=omp:noauto** を使用します。
- **-qsmp** を使用する場合は、常に thread-safe compiler invocations (**_r** 呼び出し) を使用してください。
- デフォルトでは、ランタイム環境で使用可能なプロセッサーがすべて使用されます。使用可能なプロセッサー数より少ないプロセッサーを使用するのでない限り、 *XLSMPOPTS=PARTHDS* または *OMP_NUM_THREADS* 環境変数は設定しないでください。実行スレッドの数を、小さい数または 1 に設定して、デバッグを容易にすることもできます。
- 専用のマシンまたはノードをご使用の場合は、*SPINS* および *YIELDS* 環境変数 (*XLSMPOPTS* 環境変数のサブオプション) を 0 に設定することも検討してください。そうすることにより、オペレーティング・システムが、バリアなどの同期境界を越えてスレッドのスケジューリングに介入することを防ぎます。

- OpenMP プログラムをデバッグする場合は、**-qsmp=noopt** を使用して (**-O** は指定しない)、コンパイラーが作成するデバッグ情報をより正確にするよう努力してください。

  「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

  -qsmp

  -qhot

  コンパイラーの呼び出し

  XLSMPOPTS

  並列処理用の環境変数

プロシージャー間分析の使用

プロシージャー間分析 (IPA) を使用すると、コンパイラーに、複数の異なるファイル間の最適化 (プログラム全体の分析) ができるようになり、その結果、パフォーマンスが大幅に向上します。

プロシージャー間分析は、コンパイル・ステップのみ、あるいはコンパイル・ステップとリンク・ステップの両方 (「プログラム全体」モード) で指定できます。 プログラム全体モードは、最適化の範囲をプログラム単位全体にまで拡張するモードで、実行可能オブジェクトの場合も共用オブジェクトの場合もあります。 IPA はコンパイル時間をかなり増大するので、IPA の使用は、開発過程の最終的なパフォーマンス調整段階に限定する方がよいでしょう。

IPA は、**-qipa** オプションを指定して使用可能にします。最も一般的に使用されるサブオプションとその効果を、次の表に示します。サブオプションおよび構文の完全セットについては、「*XL C/C++ コンパイラー・リファレンス*」の『**-qipa**』セクションを参照してください。

IPA を使用する手順は次のとおりです。

1. **-qipa** オプションでコンパイルする前に、準備段階としてパフォーマンス分析と調整を行います。なぜなら、IPA 分析では、コンパイルおよびリンク時間が長くなる 2 パス・メカニズムが使用されるからです。 **-qipa=noobject** オプションを使用すれば、コンパイルおよびリンク・オーバーヘッドをかなり削減できます。
2. アプリケーション全体の (またはアプリケーションのできるだけ多くの部分の) コンパイル・ステップとリンク・ステップの両方で **-qipa** オプションを指定します。サブオプションを使用して、**-qipa** でコンパイルされていない プログラムの部分について行う前提を指定します。

表 17. 一般に使用される **-qipa** *のサブオプション*

サブオプション	振る舞い
level=0	プログラム区画と簡単なプロシージャー間の最適化。その内容は次のとおりです。 • 標準ライブラリーの自動認識。 • 静的にバインドされた変数およびプロシージャーのローカライズ。 • 呼び出し関係によるプロシージャーの区分化およびレイアウト。 (相互に頻繁に呼び出すプロシージャーは、メモリー内の比較的近いところにまとめて配置されます。) • 一部の最適化、特にレジスターの割り振りの拡大。
level=1	インライン化およびグローバル・データ・マッピング。主な機能は次のとおりです。 • プロシージャーのインライン化。 • 参照の類縁性による、静的データの区分化およびレイアウト。 (頻繁に合わせて参照されるデータは、メモリー内の比較的近いところにまとめて配置されます。) -qipa オプションでサブオプションを指定しない場合は、これがデフォルト・レベルになります。
level=2	グローバル別名分析、特殊化、プロシージャー間データ・フロー: • プログラム全体の別名分析。このレベルには、ポインター間接参照と間接関数呼び出しの明確化、および関数呼び出しの副次作用に関する情報の細分が含まれます。 • 集中的なプロシージャー間最適化。これは、値の番号付け、コードの伝搬および単純化、条件へのコードの移動またはループ外へのコードの移動、および冗長の除去という形で行われます。 • プロシージャー間の定数伝搬、デッド・コードの除去、ポインター分析、関数間のコード動作、プロシージャー間の強度の低減。 • プロシージャーの特殊化 (クローン作成)。 • 全プログラム・データの再編成。
inline=*suboptions*	関数のインライン化が正確に制御できます。
fine_tuning	**-qipa** には、ほかに、ライブラリー・コードの振る舞いを指定する機能、プログラムの区分化を調整する機能、ファイルからコマンドを読み取る機能などを提供する値があります。

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**


-qipa

-qipa の最大活用

-qipa を指定してすべてをコンパイルする必要はありませんが、プログラムのできる限り多くの部分に適用してみてください。以下は提案事項です。

- アプリケーション全体のコンパイル・ステップとリンク・ステップの両方で **-qipa** オプションを指定します。ライブラリー、共用オブジェクト、および実行可能ファイルに対しても **-qipa** を使用できますが、 main 関数およびエクスポート機能をコンパイルする場合は、必ず **-qipa** を使用してください。
- コンパイルとリンクを個別に行う場合は、高速コンパイルのコンパイル・ステップで、**-qipa=noobject** を使用します。

- Make ファイルで最適化オプションを指定する場合は、必ず、コンパイラー・ドライバー (**xlc**) を使用してリンクし、リンク・ステップですべてのコンパイラー・オプションを組み込むようにしてください。
- IPA で、従来のコンパイルより非常に大きなオブジェクト・ファイルを生成できるので、 /tmp ディレクトリーに十分なスペース (少なくとも 200 MB) があることを確認してください。 TMPDIR 環境変数を使用すれば、十分なフリー・スペースを持つディレクトリーを指定できます。
- リンク時間が長すぎる場合は、**level** サブオプションを変えてみてください。 **-qipa=level=0** を指定したコンパイルは、追加リンク時間が短い場合に非常に有益です。
- **-qipa=list=long** を使用して、前にインライン化された関数のレポートを生成します。インライン化された関数が少なすぎる、あるいは多すぎる場合は、**-qinline** または **-qnoinline** の使用を検討してください。 C 個々の関数のインライン化を制御するには、**-qinline+***function_name* または **-qinline-***function_name* を使用します。 C
- リスト・ファイル内にデータ再編成情報を生成するには、最適化レベル **-qipa=level=2** または **-O5** を **-qreport** と一緒に指定します。IPA リンク・パス時に、 プログラム変数データについてのデータ再編成メッセージが、ラベル `DATA REORGANIZATION SECTION` 付きのリスト・ファイルのデータ再編成セクションに作成されます。再編成には、配列分割、配列転置、メモリー割り振りマージ、配列インターリービング、および配列合体が含まれます。

注: IPA のプロシージャー間最適化によってプログラムのパフォーマンスを大幅に向上させることができますが、その一方で、正しくはないもののそれまで機能していたプログラムが機能しなくなることもあります。以下に示すように、最適化を行わなければ偶然作動できるが、IPA によって表面化するプログラミング事例がいくつかあります。

- 割り振り順序、または自動変数のロケーションへの依存、例えば、自動変数のアドレスを決めて、後でそれを別のローカル変数と比較してスタックの成長方向を決めることがあります。 C 言語では、自動変数が割り振られている場所、またはその位置が別の自動変数に相対的である場合は保証しません。このような関数を IPA でコンパイルしないでください。
- 無効、または配列境界を越える昇順のポインター。 IPA はグローバルなデータ構造を再編成することができるので、以前に未使用メモリーを変更した可能性のある不規則ポインターが、現時点のユーザー割り振りのストレージと競合している可能性があります。
- 互換性のない型にキャストされたポインターの逆参照。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qinline

-qlist

-qipa

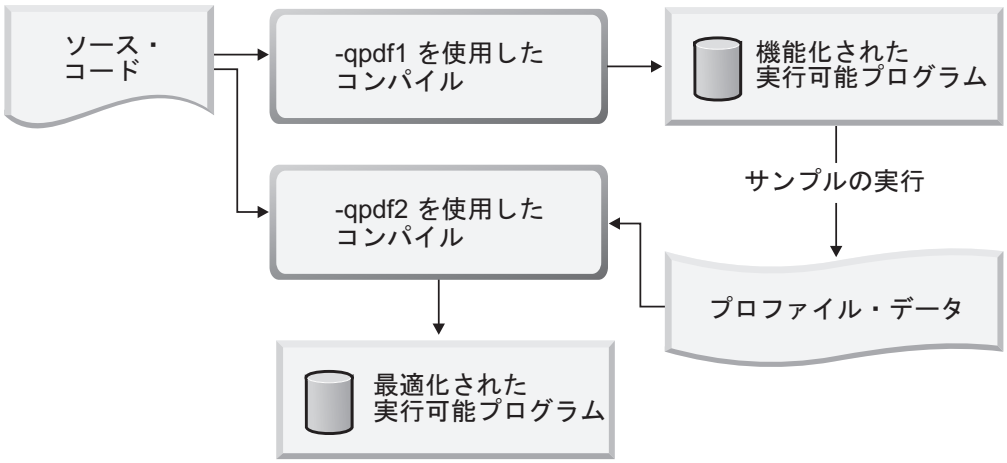
プロファイル指示フィードバックの使用

プロファイル指示フィードバック (PDF) を使用すると、アプリケーションのパフォーマンスを通常の使用例に合うように調整することができます。コンパイラーは、分岐の頻度やコード・ブロックの実行の頻度の分析に基づいて、アプリケーションを最適化します。

PDF プロセスは、アプリケーションを実稼働環境に投入する前に、最適化の最後のステップの 1 つとして使用します。**-O2** レベル以上の最適化では、PDF による恩恵を得られます。**-qipa** オプションや最適化レベル **-O4** および **-O5** などのその他の最適化を PDF プロセスで併用すると、さらにメリットがある場合があります。

次の図は、PDF プロセスを示しています。

図 2. プロファイル指示フィードバック



PDF プロセスを使用してアプリケーションを最適化するには、以下の手順に従います。

1. **-qpdf1** オプションを使用して、プログラム内の一部またはすべてのソース・ファイルをコンパイルします。最適化レベル **-O2** 以上を指定する必要があります。

   注:

   - このステップでは、PDF マップ・ファイルが生成されます。これは、**showpdf** ユーティリティーでプロファイル情報の一部分をテキストまたは XML フォーマットで表示するために使用されます。詳しくは、67 ページの『showpdf によるプロファイル情報の表示』を参照してください。プロファイル情報を表示する必要がない場合は、このステップで **-qnoshowpdf** オプションを指定して、 PDF マップ・ファイルが生成されないようにしてください。**-qnoshowpdf** について詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の**-qshowpdf** を参照してください。
   - PDF 最適化 (-qpdf) は、 **-O2** といった初期レベルで指定することもできますが、 PDF 最適化は **-O4** 以上で行うことをお勧めします。

- 必ずしもプログラムのすべてのファイルを、**-qpdf1** オプションを指定してコンパイルする必要はありません。大規模なアプリケーションでは、最適化の効果が最もよく現れるコード領域に的を絞って最適化を行うこともできます。
- **-O4** オプション、**-O5** オプション、またはいずれかのレベルの **-qipa** オプションが有効になっている場合に、**-qpdf1** オプションをリンク・ステップで指定し、コンパイル・ステップでは指定しない場合、コンパイラーは警告メッセージを発行します。このメッセージでは、すべてのプロファイル情報を取得するためプログラムを再コンパイルする必要があることが示されます。

制約事項: **-qpdf1** を指定してコンパイルしたアプリケーションを実行するときは、通常の方法 (main 関数の実行の末尾に到達すること、および C/C++ プログラムの libc (stdlib.h) にある exit() 関数を呼び出すことを含む) でアプリケーションを終了させる必要があります。 システム・コール `exit()`、`_Exit()`、 および `abort()` は、異常な終了方法とみなされ、サポートされていません。異常なプログラムの終了処理を使用すると、まったく生成されていない PDF ファイルまたは PDF データが使用されて、不完全な計測データが生成される場合があります。

2. 生成されるアプリケーションを標準的なデータ・セットを使用して実行します。アプリケーションが終了すると、1 つ以上の PDF ファイルにプロファイル情報が書き込まれます。 さまざまなデータ・セットを使用して、生成されるアプリケーションを何度でもテスト実行することができます。使用された入力データに基づいた、分岐の頻度やコード・ブロックの実行頻度のカウントを示すプロファイル情報が累積されます。 このステップは、PDF トレーニング・ステップと呼ばれます。デフォルトでは、PDF ファイルは、`._pdf` という名前が付けられ、現行作業ディレクトリー、または PDFDIR 環境変数で指定されたディレクトリーに格納されます。PDFDIR 環境変数を設定した場合に、指定したディレクトリーが存在しないと、コンパイラーから警告メッセージが出されます。 デフォルト値をオーバーライドするには、**-qpdf1=pdfname** または **-qpdf1=exename** オプションを使用します。

   -qpdf1=level=0 オプションまたは **-qpdf1=level=1** オプションを使用してプログラムを再コンパイルする場合、単一パスのプロファイル作成がサポートされます。コンパイラーは、新しいアプリケーションを生成する前に、既存の PDF ファイルを削除します。

   -qpdf1=level=2 オプションを使用してプログラムを再コンパイルする場合は、マルチパスのプロファイル作成がサポートされます。プログラムのコンパイルと、生成されたアプリケーションのテスト実行を繰り返し行うことができます。その際、新しい PDF ファイルが最大で 5 回生成されます。

   注:

   - **-qpdf1** または **-qpdf2** オプションを使用してプログラムをコンパイルする場合、デフォルトで、一緒に **-qipa** オプションが **level=0** で呼び出されます。
   - コンパイルおよび実行時間の浪費を避けるため、PDFDIR 環境変数が絶対パスに設定されていることを確認してください。そうでないと、誤ったディレクトリーからアプリケーションを実行するおそれがあり、コンパイラーがプロファイル情報ファイルを見つけることができません。この場合、プログラムを正しく最適化できなかったり、セグメンテーション障害によってプログラムが停止

される可能性があります。セグメンテーション障害は、PDF プロセスの完了前に PDFDIR 環境変数の値を変更して、アプリケーションを実行した場合にも起こることがあります。

- 標準的でないデータを使用すると、実行頻度の低いコード・パスに偏った分析にゆがめられてしまうため、そのようなデータの使用は避けてください。

3. 複数の PDF ファイルが存在する場合は、 **mergepdf** ユーティリティーを使用して、それらのファイルを 1 つの PDF ファイルに結合します。例えば、時間の 53%、32%、15% にそれぞれ発生する使用パターンを表す 3 つの PDF ファイルを作成する場合は、次のコマンドが使用してください。

   ```
   mergepdf -r 53 path1  -r 32 path2  -r 15 path3
   ```

   注:

   - バージョンまたは PTF レベルが異なる XL C/C++ コンパイラーによって作成された PDF ファイルを混用しないようにしてください。
   - 結果のアプリケーションによって生成された PDF ファイルは編集できません。PDF ファイルを編集すると、生成される実行可能アプリケーションのパフォーマンスや機能に影響を与える可能性があります。

4. 前と同じコンパイラー・オプションを使用してプログラムを再コンパイルします。ただし、**-qpdf1** は **-qpdf2** に変更します。この 2 回目のコンパイルでは、最適化を微調整するために、累積されたプロファイル情報が使用されます。結果のプログラムはプロファイル作成オーバーヘッドを含んでいないため、フルスピードで実行されます。

   注:

   - コンパイラーがこのステップでどの PDF ファイルも読み取れない場合、コンパイラーは重大なエラー・メッセージを発行して、コンパイルを停止します。
   - 特定のプログラムのコンパイル手順ではすべて、同じ最適化レベルを使用することを強くお勧めします。そうでないと、PDF プロセスがプログラムを正しく最適化できないだけでなく、プログラムの速度を低下させるおそれがあります。構成ファイルでの設定も含めて、最適化に影響するすべてのコンパイラー設定が一致している必要があります。
   - ソース・コードを変更し、**-qpdf1** および **-qpdf2** オプションを使用して、プログラムをコンパイルすることができます。古いプロファイル情報を保存しておいて、PDF プロセスの第 2 段階で使用することができます。コンパイラーから警告リストが出されますが、コンパイルは停止しません。古いプロファイル情報の古さを示す 0 から 100 までの番号が付いた通知メッセージも出されます。
   - **-O4** オプション、**-O5** オプション、または任意レベルの **-qipa** オプションが有効になっている場合に、**-qpdf2** オプションをリンク・ステップで指定する一方でコンパイル・ステップでは指定しない場合、コンパイラーは警告メッセージを発行します。このメッセージでは、すべてのプロファイル情報を取得するためプログラムを再コンパイルする必要があることが示されます。
   - **-qpdf2** オプションと共に **-qreport** オプションを使用した場合、リスト・ファイル内にプログラムの調整に役立つ追加情報を取得することができます。この情報は、`PDF Report` セクションに書き込まれます。

5. PDF 情報を削除する場合は、**cleanpdf** または **resetpdf** ユーティリティーを使用します。

ステップ 4 の代わりに、**-qpdf2** フェーズでプログラムを再コンパイルせずに、**-qpdf1** フェーズ中に作成されたオブジェクト・ファイルを **-qpdf2** オプションを使用してリンクさせることができます。この代替方法を使用すると時間を大幅に節約することができ、大規模なアプリケーションを最適化のために調整するのに役立ちます。

例

以下の例は、**-qpdf1** オプションを使用してアプリケーションのコード全体をコンパイルするのではなく、最適化の効果が最もよく現れるコードに的を絞ってコンパイルを実行できることを示しています。

```
#Set the PDFDIR variable
export PDFDIR=$HOME/project_dir

#Compile most of the files with -qpdf1
xlc -qpdf1 -O3 -c file1.c file2.c file3.c

#This file does not need optimization
xlc -c file4.c

#Non-PDF object files such as file4.o can be linked
xlc -qpdf1 -O3 file1.o file2.o file3.o file4.o

#Run several times with different input data
./a.out < polar_orbit.data
./a.out < elliptical_orbit.data
./a.out < geosynchronous_orbit.data

#No need to recompile the source of non-PDF object files
#(file4.c).
xlc -qpdf2 -O3 -c file1.c file2.c file3.c

#Link all the object files into the final application
xlc -qpdf2 -O3 file1.o file2.o file3.o file4.o
```

以下の例では、**-qpdf2** オプションを使用したソースの再コンパイルを回避しています。

```
#Compile source with -qpdf1
xlc -c -qpdf1 -O3 file1.c file2.c

#Link object files
xlc -qpdf1 -O3 file1.o file2.o

#Run with one set of input data
./a.out < sample.data

#Link the mix of pdf1 and pdf2 objects
xlc -qpdf2 -O3 file1.o file2.o
```

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qpdf1、-qpdf2

-O, -qoptimize

ランタイム環境変数

showpdf によるプロファイル情報の表示

showpdf ユーティリティーを使用すると、アプリケーションから収集された以下のタイプのプロファイル情報を表示することができます。

- ブロック・カウンターのプロファイル
- 呼び出しカウンターのプロファイル
- 値のプロファイル
- キャッシュ・ミスのプロファイル (**-qpdf1** フェーズで **-qpdf1=level=2** オプションを指定した場合)

最初の 2 つのタイプのプロファイル情報は、テキスト・フォーマットまたは XML フォーマットのいずれかで表示できます。しかし、値のプロファイルとキャッシュ・ミスのプロファイル情報は、XML フォーマットでしか表示できません。

構文

パラメーター

pdfdir
プロファイル指示フィードバック (PDF) ファイルを格納するディレクトリーです。 **-qpdf1** フェーズの後に PDFDIR 環境変数を変更していない場合、 PDF マップ・ファイルもこのディレクトリーに格納されます。このパラメーターを指定しない場合、コンパイラーは PDFDIR 環境変数の値をディレクトリーの名前として使用します。

pdfname
PDF ファイルの名前。このパラメーターを指定しない場合、コンパイラーは `._pdf` を PDF ファイルの名前として使用します。

pdfmapdir
PDF マップ・ファイルを含むディレクトリー。このパラメーターを指定しない場合、コンパイラーは PDFDIR 環境変数の値をディレクトリーの名前として使用します。

-xml
PDF 情報の表示フォーマットを決定します。このパラメーターを指定した場合、 PDF 情報は XML フォーマットで表示されます。指定しない場合は、テキスト・フォーマットで表示されます。値のプロファイルおよびキャッシュ・ミスのプロファイル情報は、XML フォーマットでしか表示できないため、XML フォーマットの PDF レポートは、テキスト・フォーマットのレポートよりも情報量が多くなります。

使用法

静的情報を格納する PDF マップ・ファイルは、**-qpdf1** フェーズ中に生成されます。これに対して PDF ファイルは、結果として生成されたアプリケーションの実行中に生成されます。 **showpdf** ユーティリティーでは、PDF 情報をテキストと XML のいずれかのフォーマットで表示するために、PDF ファイルと PDF マップ・ファイルの両方を必要とします。

-qpdf1 フェーズで **-qpdf1=level=2** オプションを指定した場合、複数の PDF ファイルおよび PDF マップ・ファイルが生成されることがあります。この場合は、プロファイル情報を表示するときに、PDF ファイルと PDF マップ・ファイルのペアごとに **showpdf** ユーティリティーを実行する必要があります。

デフォルトでは、PDF ファイルには `._pdf` という名前が付けられ、PDF マップ・ファイルには `._pdf_map` という名前が付けられます。PDFDIR 環境変数が設定されている場合、PDF ファイルおよび PDF マップ・ファイルは、コンパイラーによって PDFDIR で指定されたディレクトリーに配置されます。PDFDIR 環境変数を設定しなかった場合、コンパイラーはこれらのファイルを現行作業ディレクトリーに格納します。 PDFDIR 環境変数を設定した場合に、指定したディレクトリーが存在しないと、コンパイラーから警告メッセージが出されます。 デフォルト値をオーバーライドするには、 **-qpdf1=pdfname** オプションを使用して、PDF ファイルおよび PDF マップ・ファイルのパスと名前を指定します。例えば、 **-qpdf1=pdfname=/home/joe/func** オプションを指定すると、結果の PDF ファイルの名前は `func` となり、 PDF マップ・ファイルの名前は `func_map` となります。どちらのファイルも、 `/home/joe` ディレクトリーに格納されます。

-qpdf1 フェーズの後、結果のアプリケーションの実行前に PDFDIR 環境変数を変更した場合、PDF ファイルと PDF マップ・ファイルが別々のディレクトリー内に生成されます。この場合は、この両方のファイルのディレクトリーを **showpdf** ユーティリティーに指定する必要があります。

注:

- 同じコンパイル・インスタンスから、PDF ファイルと PDF マップ・ファイルを生成する必要があります。そうしないと、コンパイラーからエラーが出されます。
- 同じプロファイル作成プロセス中に、PDF ファイルと PDF マップ・ファイルを生成する必要があります。これは、異なるプロファイル作成プロセスから生成された PDF ファイルと PDF マップ・ファイルを組み合わせて使用できないことを意味します。
- 同じバージョンおよび PTF レベルのコンパイラーを使用して、PDF ファイルと PDF マップ・ファイルを生成する必要があります。
- **showpdf** ユーティリティーは、バイナリー・フォーマットの PDF ファイルのみを受け入れます。
- PDF_WL_ID 環境変数を使用して、ユーザー・プログラムを複数回トレーニング実行して生成された複数セットの PDF カウンターを区別できます。

以下の例では、**showpdf** ユーティリティーを使用して、「Hello World」アプリケーションのプロファイル情報を表示する方法を示しています。

プログラム・ファイル `hello.c` のソースは次のとおりです。

```
#include <stdio.h>
void HelloWorld()
{
   printf("Hello World");
}
main()
```

```
{
   HelloWorld();
   return 0;
}
```

1. ソース・ファイルをコンパイルします。

   ```
   xlc -qpdf1 -O hello.c
   ```

2. 標準的なデータ・セットを 1 つ以上使用して、結果の実行可能プログラム **a.out** を実行します。
3. 実行可能ファイルのプロファイル情報をテキスト・フォーマットで表示する場合は、**showpdf** ユーティリティーをパラメーターを指定せずに実行します。

   ```
   showpdf
   ```

   結果は、次のようになります。

   ```
   HelloWorld(67):  1 (hello.c)

   Call Counters:
   4 | 1  printf(69)

   Call coverage = 100% ( 1/1 )

   Block Counters:
   2-4 | 1
   5 |
   5 | 1

   Block coverage = 100% ( 2/2 )

   ----------------------------------
   main(68):  1 (hello.c)

   Call Counters:
   8 | 1  HelloWorld(67)

   Call coverage = 100% ( 1/1 )

   Block Counters:
   6-9 | 1
   10 |

   Block coverage = 100% ( 1/1 )

   Total Call coverage = 100% ( 2/2 )
   Total Block coverage = 100% ( 3/3 )
   ```

   プロファイル情報を XML フォーマットで表示する場合は、**showpdf** ユーティリティーを **-xml** パラメーターを指定して実行します。

   ```
   showpdf -xml
   ```

   結果は、次のようになります。

   ```
     <?xml version="1.0" encoding="UTF-8" ?>
   - <XLTransformationReport xmlns="http://www.ibm.com/2010/04/CompilerTransformation" version="1.0">
     - <CompilationStep name="showpdf">
       - <ProgramHierarchy>
         - <FileList>
           - <File id="1" name="hello.c">
             - <RegionList>
                 <Region id="67" name="HelloWorld" startLineNumber="2" />
                 <Region id="68" name="main" startLineNumber="6" />
               </RegionList>
             </File>
           </FileList>
         </ProgramHierarchy>
   ```

```
    <TransformationHierarchy />
  - <ProfilingReports>
    - <BlockCounterList>
      - <BlockCounter regionId="67" execCount="1" coveredBlock="2" totalBlock="2">
        - <BlockList>
            <Block index="3" execCount="1" startLineNumber="2" endLineNumber="4" />
            <Block index="2" execCount="0" startLineNumber="5" endLineNumber="5" />
            <Block index="4" execCount="1" startLineNumber="5" endLineNumber="5" />
          </BlockList>
        </BlockCounter>
      - <BlockCounter regionId="68" execCount="1" coveredBlock="1" totalBlock="1">
        - <BlockList>
            <Block index="3" execCount="1" startLineNumber="6" endLineNumber="9" />
            <Block index="2" execCount="0" startLineNumber="10" endLineNumber="10" />
          </BlockList>
        </BlockCounter>
      </BlockCounterList>
    - <CallCounterList>
      - <CallCounter regionId="67" execCount="1" coveredCall="0" totalCall="0">
        - <CallList>
            <Call name="printf" execCount="1" lineNumber="4" />
          </CallList>
        </CallCounter>
      - <CallCounter regionId="68" execCount="1" coveredCall="0" totalCall="0">
        - <CallList>
            <Call name="HelloWorld" execCount="1" lineNumber="8" />
          </CallList>
        </CallCounter>
      </CallCounterList>
      <ValueProfileList />
      <CacheMissList />
    </ProfilingReports>
  </CompilationStep>
</XLTransformationReport>
```

「*XL C/C++ コンパイラー・リファレンス*」の関連情報


-qpdf1、-qpdf2


-qshowpdf

オブジェクト・レベルのプロファイル指示フィードバック

このタスクについて

実行可能ファイル全体の最適化に加えて、個々のオブジェクト・ファイル に Profile-Directed Feedback (PDF) を適用することもできます。これは、パッチや更新が実行可能ファイルではなくオブジェクト・ファイルまたはライブラリーとして配布されるアプリケーションにおいて利点となり得ます。また、アプリケーション全体を再リンクするプロセスを経ずにアプリケーション内の個々の機能領域を最適化することもできます。したがって、大規模なアプリケーションでは、アプリケーションの再リンクに費やされていた時間と労力を節約できます。

オブジェクト・レベルの PDF を使用するプロセスは、**-qpdf2** ステップにわずかな変更があることを除けば、基本的に標準 PDF プロセスと同じです。オブジェクト・レベルの PDF の場合、**-qpdf1** オプションを使用してプログラムをコンパイルし、結果のアプリケーションを典型的なデータで実行し、**-qpdf2** オプションを使用して再度プログラムをコンパイルしますが、ここでは、リンク・ステップをスキップするために、**-qnoipa** オプションも使用します。

以下の手順は、このプロセスの概要を説明したものです。

1. **-qpdf1** オプションを使用してプログラムをコンパイルします。次に例を示します。

   ```
   xlc -c -O3 -qpdf1 file1.c file2.c file3.c
   ```

この例では、最適化レベル **-O3** を使用して、中レベルの最適化を行うように指示しています。

2. オブジェクト・ファイルをリンクして、装備された実行可能ファイルを取得します。

   ```
   xlc -O3 -qpdf1 file1.o file2.o file3.o
   ```

3. 最適化するデータに典型的なサンプル・データを使用して、装備された実行可能ファイルを実行します。

   ```
   a.out < sample_data
   ```

4. **-qpdf2** オプションを使用してプログラムを再びコンパイルします。リンク・ステップがスキップされ、PDF 最適化が実行可能ファイル全体ではなくオブジェクト・ファイルに適用されるようにするため、**-qnoipa** オプションを指定します。

   ```
   xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
   ```

   このステップの結果の出力は、元の装備された実行可能ファイルが処理したサンプル・データ用に最適化されたオブジェクト・ファイルです。この例では、最適化されたオブジェクト・ファイルは file1.o、file2.o、および file3.o です。システム・ローダー **ld** を使用するか、**-qpdf2** のステップで **-c** オプションを省略すると、これらのファイルをリンクできます。

注:

- すべてのステップで同じ最適化レベルを使用する必要があります。この例では、最適化レベルは **-O3** です。
- 作成されるプロファイルのファイル名を指定したい場合は、**-qpdf1** ステップと **-qpdf2** ステップの両方で **pdfname** サブオプションを使用してください。次に例を示します。

  ```
  xlc -O3 -qpdf1=pdfname=myprofile file1.c file2.c file3.c
  ```

  pdfname サブオプションを指定しないと、デフォルトによりファイル名は `._pdf` となります。ファイルの格納場所は、現行作業ディレクトリーまたは PDFDIR 環境変数を使用して設定したディレクトリーになります。PDFDIR 環境変数を設定した場合に、指定したディレクトリーが存在しないと、コンパイラーから警告メッセージが出されます。
- この **-qpdf2** ステップでは、オブジェクト・ファイルのリンクをスキップするために、**-qnoipa** オプションを指定する必要があるため、プロシージャー間分析 (IPA) の最適化とオブジェクト・レベルの PDF を同時に使用することはできません。

詳細については、「XL C/C++ コンパイラー・リファレンス」の『-qpdf1、-qpdf2』セクションを参照してください。

目次 (TOC) オーバーフローの処理

目次 (TOC) オーバーフローを処理するには、グローバル・シンボルの数を減らすか、TOC アクセス範囲を拡大するか、またはプロシージャー間分析を適用します。TOC は、64 ビット・モードのみが対象です。

プログラム内のグローバル・シンボルのアドレスは、TOC と呼ばれるデータ構造体に格納されています。グローバル・シンボルにアクセスするには、そのグローバル・シンボルのアドレスを TOC から取得する必要があります。デフォルト TOC データ構造体は、一定数のグローバル・シンボルを格納できる固定サイズになっています。例えば、IBM PowerPC アーキテクチャーでは、間接アドレス計算のための符号付き 16 ビット・オフセットを伴う命令が使用され、TOC のサイズが 64 KB に制限されます。32 ビット・モードでは、最大で 16 K のエントリーを TOC に格納でき、64 ビット・モードでは最大で 8 K のエントリーを格納できます。

大規模なアプリケーションでは、デフォルト TOC に格納できるよりも多くのグローバル・シンボルが保持されていることがよくあります。TOC に格納できるよりも多くの TOC エントリーがアプリケーションに含まれている場合、リンカーは TOC オーバーフローを報告して、代替のメカニズムを使用する必要があることを通知します。TOC オーバーフローを処理するには、以下の方法を使用してください。

- 以下の方法で、プログラム内のグローバル・シンボルの数を減らします。
  - ソース・コードを変更します。これは、グローバル・シンボルの数を減らすための最適な方法です。
  - **-qminimaltoc**オプションを指定します。
  - **-qipa** オプションを指定することで、プロシージャー間分析を適用します。このオプションについて詳しくは、 60 ページの『プロシージャー間分析の使用』 および 61 ページの『-qipa の最大活用』 を参照してください。
- **-qpic=large** オプションを指定して、TOC アクセス範囲を拡大します。

グローバル・シンボルの数を減らすためのオプション

目次 (TOC) オーバーフローを処理するための最適な方法は、グローバル・シンボルの数を減らすことで、必要な TOC エントリーの数も減らすことです。

ソース・コードを変更することで、グローバル・シンボルの数を減らすことができます。可能であれば、ソース・コードを変更して、不要なグローバル変数および関数を削除するか、静的なものとしてマーク付けするか、またはグローバル変数を構造体にグループ化します。ただし、ソース・コードの変更には、時間がかかったり間違いが伴ったりすることがあります。実際には、リンク時の最適化が高い最適化レベル (**-O4** および **-O5**) で使用されている場合や、コンパイル時とリンク時の両方で **-qipa** を使用して最適化が明示的に適用されている場合、コンパイラーはこれらのタスクを自動的に完了できます。この最適化の結果は、ソースの変更を通じて実現できるものとほぼ同じですが、広範囲にわたる手動のソース変更は伴いません。

グローバル・シンボルの数を減らすには、**-qminimaltoc** オプションを指定することもできます。このオプションを指定すると、コンパイラーはソース・ファイルごとに別々のテーブルを作成します。このテーブルには、該当ソース・ファイルで使用されている各グローバル・シンボルのアドレスが含まれています。このオプションを指定した場合、コンパイラーはコンパイル単位ごとに 1 つの TOC エントリーのみを作成します。このオプションが効果を持つのは、TOC エントリーが複数のソース・ファイルに分散されている場合のみです。単一のソース・ファイルに、TOC オ

ーバーフローを発生させるのに十分な数のグローバル・シンボルが含まれている場合、このオプションには TOC オーバーフローへの対処法としての効果はありません。

注:

- **-qminimaltoc** は、コンパイル単位ごとに指定する必要はありません。
- **-qminimaltoc** は、パフォーマンスを低下させる可能性があるため、慎重に使用してください。**-qminimaltoc** を使用すると間接参照が生じるため、グローバル・シンボルへのアクセスにかかる時間が長くなります。もう 1 つの欠点として、アプリケーションに必要なメモリー量が増大する可能性があります。このオプションによって影響を受けるパフォーマンスについて詳しくは、74 ページの『TOC オーバーフロー処理のパフォーマンス考慮事項』 を参照してください。

TOC アクセス範囲を拡大するためのオプション

目次 (TOC) アクセス範囲の拡大は、TOC オーバーフローを処理するための効果的な方法です。2 つの命令を使用して TOC にアクセスし、TOC の範囲を連続する TOC 領域にグループ化します。IBM PowerPC 上の最大 16 ビット・オフセットは、64 K の TOC 領域で構成される大きな TOC をサポートしています。各 TOC 領域内に最大 64 K のエントリーが含まれている場合、大きい TOC は 4 GB に達することがあります。これにより、32 ビット環境では 1 G のグローバル・シンボルという制限が生じ、64 ビット環境では 500 M のグローバル・シンボルという制限が生じます。POWER8 ベースのシステムでは、多くの場合 2 つの命令は、1 つの命令として、まとめて同時に実行されます。

TOC を拡大するには、**-qpic=large** オプションを指定できます。これらのオプションを指定する前に、TOC エントリーの数を減らしてください。TOC エントリーが多いと、生成されたコードが含まれたプログラムのパフォーマンスが低下する可能性があるからです。

-qpic=large

-qpic=large を指定すると、コンパイラーは常に、TOC オーバーフローが発生するかどうかにかかわらず、シンボルのアドレスを取得するための 2 つの命令を生成します。このオプションを指定した場合、ベース TOC 内のシンボルを含むすべてのシンボルは、アドレスを計算するための追加の命令を必要とします。通常の 64 KB ベースの TOC サイズ内のオフセットの場合、リンカーは、1 つ目の命令を何も実行しない命令に変換するため、実行時間を要しません。 POWER8 では、これら 2 つの命令は、多くの場合、より大きな変位により、1 つの命令にマージされます。リンカーは、オフセットのためのアウト・オブ・ライン・コードへの分岐を挿入しません。このオプションがパフォーマンスに与える影響について詳しくは、74 ページの『TOC オーバーフロー処理のパフォーマンス考慮事項』 を参照してください。

注: -qpic=large は、コンパイル単位ごとに指定する必要はありません。

TOC オーバーフロー処理のパフォーマンス考慮事項

目次 (TOC) オーバーフローを処理する際は、パフォーマンスを考慮する必要があります。TOC オーバーフローをバイパスする際は、実行時のパフォーマンスに対する悪影響を最小限に抑えてください。

TOC オーバーフローが発生する場合、最適な解決策は、ソース・コードを変更してグローバル・シンボルの数を減らすことです。または、要求されるパフォーマンスに応じて、以下のオプションの使用を検討することもできます。

-qipa

- **利点:** プロシージャー間分析 (IPA) プロセスを適用すると、TOC 負荷が大幅に低減されます。ほとんどの場合、これにより TOC オーバーフローは完全に解消されます。IPA はこのことを実現するため、グローバル・シンボルの数が減るようにプログラムを再構築します。
- **考慮事項:** IPA は最適化レベル **-O4** および **-O5** では暗黙指定されますが、これらの最適化レベルには、商用アプリケーション開発には適さない他の複雑な最適化も含まれています。推奨される代替策の 1 つは、最低限レベルのプログラム全体の最適化を適用する **-qipa=level=0** を使用することですが、大規模なアプリケーションの場合は **-qipa=level=1** が必要になる場合があります。レベル 1 では、TOC 要件がより積極的に低減される代わりに、コンパイル・プロセスにかかる時間が長くなります。

注: プログラム全体の分析を行うには、**-qipa** オプションをコンパイルとリンクの両方のコマンド・ラインで指定する必要があります。

-qminimaltoc

- **利点:** ソース・コードに、パフォーマンスに影響を与えにくいグローバル・シンボルしか含まれていない場合、**-qminimaltoc** を使用してファイルをコンパイルしてください。このオプションを指定すると、ソース・ファイル内のすべてのグローバル・シンボルが別個のデータ構造体に格納されるため、TOC に対する負荷全体を簡単に低減できます。
- **考慮事項:** このオプションは慎重に使用する必要があり、頻繁に実行されるコードが含まれたファイルの場合は特に注意が必要です。パフォーマンスに影響を与えやすいグローバル・シンボルが含まれたファイルをコンパイルする場合にこのオプションを使用すると、パフォーマンスが多大な影響を受けることがあります。**-qminimaltoc** を指定した場合、プログラムは以前より大きくなり低速になります。

注: -qminimaltoc は、すべてのコンパイル単位で指定する必要はありません。このオプションをパフォーマンスと無関係なコンパイル単位のみで使用することで、パフォーマンスの低下を最小限に抑えてください。

-qpic=large

- **利点: -qpic=large** オプションは、通常、推奨される解決策です。このオプションを使用すると、ベース TOC 内と拡張 TOC 内の両方のシンボルへのアクセスについて、最適なバランスが実現されるからです。
- **考慮事項:** このオプションは、TOC オーバーフローが発生するかどうかにかかわらず、2 つの命令を使用してシンボルのアドレスを取得するため、パフォーマン

スに影響を与える可能性があります。POWER8 ベースのシステムでは、多くの場合 2 つの命令は、1 つの命令として、まとめて同時に実行されます。

最適化の機会を診断するためのコンパイラー・レポートの使用

-qlistfmt オプションを使用すると、プログラムに対してどのような最適化が行われたかを詳しく示す、XML または HTML フォーマットのコンパイラー・レポートを生成できます。**genhtml** ツールを使用して、既存の XML レポートを HTML フォーマットに変換することもできます。この情報を使用して、アプリケーションのコードを理解し、コードを調整してパフォーマンスをさらに向上させることができます。

XML フォーマットのコンパイラー・レポートは、XSLT をサポートするブラウザーで表示できます。スタイルシート・サブオプション **-qlistfmt=xml=all:stylesheet=xlstyle.xsl** を指定してコンパイルすると、 XML を読みやすくするスタイルシートへのリンクがレポートに組み込まれ、コードをより効果的に最適化するための情報が得られます。また、この情報を構文解析するためのツールを作成することもできます。

デフォルトでは、このレポートの名前は `a.html` (HTML フォーマット) および `a.xml` (XML フォーマット) です。**-qlistfmt=filename** オプションを使用すると、このデフォルト名とは異なる名前を指定できます。

インライン・レポート

-qinline と、**-qlistfmt=xml=inlines**、**-qlistfmt=html=inlines**、**-qlistfmt=xml**、または **-qlistfmt=html** のいずれかを指定してコンパイルした場合、生成されるコンパイラー・レポートには、コンパイル中に試行されたインライン化のリストが含まれます。このレポートには、試行のタイプとその結果も示されます。

コンパイラーがインライン化を試行した関数ごとに、インライン化が成功したかどうかが示されます。レポートには、正常にインライン化されなかった名前付き関数の説明が任意の数含まれていることがあります。説明の例をいくつか示します。

- FunctionTooBig: 関数が大きすぎてインライン化できません。
- RecursiveCall: 関数が再帰的であるためインライン化されません。
- ProhibitedByUser: ユーザー指定のプラグマまたはディレクティブが理由でインライン化が行われませんでした。
- CallerIsNoopt: 呼び出し元が最適化なしでコンパイルされたため、インライン化が行われませんでした。
- WeakAndNotExplicitlyInline: 呼び出し側関数が弱く、インラインとしてマークが付けられていません。

表示される可能性がある説明を網羅したリストについては、XML スキーマ・ヘルプ・ファイル `XMLContent.html` 内の `Inline optimization types` セクションを参照してください。このファイルは、日本語版と中国語版の `XMLContent-Japanese.utf8.html` および `XMLContent-Chinese.utf8.html` も格納されている `/opt/ibm/xlC/13.1.0/listings/` ディレクトリーにあります。

ループ変換

-qhot と、**-qlistfmt=xml=transforms**、 **-qlistfmt=html=transforms**、 **-qlistfmt=xml**、または **-qlistfmt=html** のいずれかを指定してコンパイルした場合、生成されるコンパイラー・レポートには、コンパイル時にそのファイル内の全ループで実行された変換のリストが含まれます。レポートでは、変換が行われなかったケースについてはその理由もリストされます。

- ループが自動的に並列化できない理由
- ループのアンロールができない理由
- SIMD ベクトル化に失敗した理由

表示される可能性がある変換上の問題を網羅したリストについては、XML スキーマ・ヘルプ・ファイル `XMLContent.html` 内の `Loop transformation types` セクションを参照してください。このファイルは、日本語版と中国語版の `XMLContent-Japanese.utf8.html` および `XMLContent-Chinese.utf8.html` も格納されている `/opt/ibm/xlC/13.1.0/listings/` ディレクトリーにあります。

データの再編成

-qhot と、**-qlistfmt=xml=data**、 **-qlistfmt=html=data**、**-qlistfmt=xml**、または **-qlistfmt=html** のいずれかを指定してコンパイルした場合、生成されるコンパイラー・レポートには、コンパイル時にプログラムで実行されたデータ再編成のリストが含まれます。データ再編成の例として、以下のものがあります。

- 配列分割
- 配列合体
- 配列インターリービング
- 配列入れ替え
- メモリー・マージ

これらの再編成ごとに、データ名、ファイル名、行番号、および領域名に関する詳細がレポートに記載されます。

Profile-Directed Feedback レポート

-qpdf2 と、 **-qlistfmt=xml=pdf**、**-qlistfmt=html=pdf**、 **-qlistfmt=xml**、または **-qlistfmt=html** のいずれかを指定してコンパイルした場合、生成されるコンパイラー・レポートには、以下の情報が含まれます。

- ループの繰り返し数
- ブロック数と呼び出し数
- キャッシュ・ミス (**-qpdf1=level=2** でコンパイルした場合)

開発ツールを使用したコンパイラー・レポート構文解析

XML 形式で作成されたコンパイラー・レポートを構文解析するソフトウェア開発ツールを作成できます。このようなツールは、アプリケーションのパフォーマンスを向上させる機会をユーザーに与えてくれます。

コンパイラーには、XML スキーマが含まれています。これを使用してコンパイラー・レポートを構文解析するツールを作成し、パフォーマンスを向上させる機会を

示す可能性のあるコードの特性を表示することができます。このスキーマ xllisting.xsd は、/opt/ibm/xlC/13.1.0/listings/ ディレクトリーにあります。このスキーマは、レポートからの情報をツリー構造で示すのに役立ちます。

このディレクトリーには、スキーマの詳細を理解するのに役立つスキーマ・ヘルプ・ファイル XMLContent.html と、このファイルの日本語版および中国語版である XMLContent-Japanese.utf8.html および XMLContent-Chinese.utf8.html が見つかります。

変数にローカル変数またはインポートされた変数としてのマークを付ける

コンパイラーでは、アプリケーション内のすべての変数がインポートされたものであると想定されますが、-qdatalocal および -qdataimported を使用することで、変数にローカル変数またはインポートされた変数としてマークを付けることができます。コンパイラーは、プログラム変数の静的バインディングまたは動的バインディングの指定に基づいて、アプリケーションを最適化します。

-qdatalocal

ローカル変数は、プログラムまたは共用ライブラリーに一意的にバインドされている、メモリーの特別なセグメントに格納されます。コンパイル済みプログラムまたは共用ライブラリーに対して変数がローカル変数として扱われるようにするには、-qdatalocal オプションを指定します。このオプションをパラメーターなしで指定すると、すべての該当する変数がローカル変数として扱われるようになります。または、コロンで区切った名前のリストをオプションに付加して、プログラム実引数のサブセットのみがローカル変数として扱われるようにすることもできます。

可能な場合は、ローカル変数としてマーク付けされた変数は、メモリー内の別のグローバルな部分にではなく、目次 (TOC) と呼ばれる構造体に直接組み込まれます。変数が TOC に組み込まれるためには、変数のストレージがポインターのサイズ以下であることが前提となります。通常、データへのポインターは、TOC に格納されます。-qdatalocal オプションを指定すると、データを TOC に直接格納できるため、ロード命令を 2 回から 1 回にして、データ・アクセスを減らすことができます。

-qdataimported

インポートされた変数は、デフォルトのメモリー割り振りスキームに従って格納されます。-qdataimported オプションは、デフォルトのデータ・バインディングの仕組みです。このオプションを指定することは、リンクされている他のプログラムまたは共用ライブラリーからデータが可視であることを意味します。そのため、変数名を -qdataimported オプションに引数として指定しても、引数を指定せずに -qdataimported オプションを単独で使用してコンパイルしても、効果がありません。

-qdataimported オプションは、-qdatalocal オプションと組み合わせて使用すると便利です。すべてのデータを TOC に保管しなければならないようなことはあり得ないため、プログラムまたは共用ライブラリーの外部の変数について、-qdatalocal オプションの指定を-qdataimported オプションでオーバーライドできます。例え

ば、-qdatalocal -qdataimported=<*variable*> というオプションを指定すると、<*variable*> を除く、すべてのグローバル・データが TOC に格納されます。

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**


-qdataimported、-qdatalocal、-qtocdata

-qdatalocal の最大活用

-qdatalocal オプションの使用方法を説明したいくつかの例を示します。

次のプログラム・ファイルのソースでは、A1 および A2 は、グローバル変数です。

```
int A1;
int A2;
int main(){
  A2=A1+1;
  return A2;
}
```

以下は、-qdatalocal を指定せずに、-qlist を指定した場合に作成されるリスト・ファイルの抜粋です。

```
    | 000000                          PDEF     main
  4 |                                 PROC
  5 | 000000 lwz      80620004   1    L4A       gr3=.A1(gr2,0)
  5 | 000004 lwz      80630000   1    L4A       gr3=A1(gr3,0)
  5 | 000008 addi     38630001   1    AI        gr3=gr3,1
  5 | 00000C lwz      80820008   1    L4A       gr4=.A2(gr2,0)
  5 | 000010 stw      90640000   1    ST4A      A2(gr4,0)=gr3
```

以下は、-qdatalocal を指定して、-qlist を指定した場合に作成されるリスト・ファイルの抜粋です。

```
    | 000000                          PDEF     main
  4 |                                 PROC
  5 | 000000 lwz      80620004   1    L4A       gr3=A1(gr2,0)
  5 | 000004 addi     38630001   1    AI        gr3=gr3,1
  5 | 000008 stw      90620008   1    ST4A      A2(gr2,0)=gr3
```

-qdatalocal を指定した場合、A1 変数および A2 変数は TOC に組み込まれるため、データへのアクセスは単一のロード命令によって行われます。-qdatalocal を指定しない場合、A1 変数および A2 変数へのアクセスは、2 つのロード命令によって行われます。この例では、-qdatalocal=A1:A2 という形式を使用して、ローカル変数を個々に指定することができます。

-qlist によって作成された .lst ファイルの >>>>> OPTIONS SECTION <<<<< は、常に表示可能であり、これらのオプションの使用を確認できます。例えば、オプションが指定されている場合、DATALOCAL=<*variables*> または DATALOCAL を表示できます。

注:

- 64 ビット Linux では、TOC エントリーは、ポインター・サイズです。引数なしで -qdatalocal を指定した場合、ポインター・サイズより大きい変数のオプションは無視されます。逆に、ポインター・サイズよりも小さいデータは、ワード境界合わせが行われます。 char (r3) がローカルとしてマーク付けされている場合に表示される、以下の objdump の抜粋の例をご覧ください。バイトと次のデータ (r4) の間のオフセットは、4 バイトのまま変わりません。このデータは、通常の

ロードではなく、ロード・バイト命令によってアクセスされます。TOC がデータを格納する方法について詳しくは、『目次 (TOC) オーバーフローの処理』を参照してください。

```
10000380:       88 62 00 20     lbz     r3,32(r2)
10000384:       80 82 00 24     l       r4,36(r2)
r2 (base address of the TOC), r3 (char), r4 (int)
```

- -qdatalocal オプションは、32 ビットの Linux では効果がありません。
- 適切でない変数を -qdatalocal にパラメーターとして指定した場合、-qdatalocal は無視されます。適切でない変数には、ポインター・サイズのバイトよりも大きいデータや、存在しない変数などがあります。TOC 候補でない変数に -qdatalocal を指定すると、その変数のストレージは、デフォルトで -qdataimported に指定され、変数は TOC に格納されません。
- C++ ローカル変数を指定する場合、マングルされた名前を使用する必要があります。そうしないと、エラー・メッセージが表示される場合があります。 C++
- 変数をローカル変数としてマーク付けする場合は注意が必要です。引数なしで -qdatalocal を指定した場合、すべてのグローバル変数が TOC への直接格納の候補になります。それらの変数が外部変数としてマーク付けされていても関係ありません。静的リンケージを持つ変数に同様の問題はありません。
- TOC 構造体は、1 つのモジュールまたは共用ライブラリーに固有であるため、-qdatalocal オプションの効果は、そのモジュールまたは共用ライブラリー内のデータに限定されます。
- 複数のモジュールを持つプログラムの場合、複数の TOC 構造体間で切り替えを行うと、このオプションを使用することによる高速化の効果が薄れる場合があります。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qdataimported、-qdatalocal、-qtocdata

その他の最適化オプション

オプションは、最適化の特定の局面を制御する際に使用できます。これらのオプションは、グループとして使用可能にされることもよくあり、また、もっと汎用的な最適化オプションまたはレベルを使用可能にすると、デフォルト値を与えられることもあります。

詳しくは、「*XL C/C++ コンパイラー・リファレンス*」に記載の各オプションの見出しを参照してください。

表 18. パフォーマンスの最適化のために選択されるコンパイラー・オプション

オプション	説明
-qignerrno	コンパイラーが、errno はライブラリー関数呼び出しによって変更されないので、そのような呼び出しは最適化できると判断できるようにします。また、ライブラリー関数の呼び出しではなく、インライン・コードの生成によって、平方根演算の最適化を行うことも可能になります。 (sqrt をサポートするプロセッサーの場合)

表 18. パフォーマンスの最適化のために選択されるコンパイラー・オプション (続き)

オプション	説明
-qsmallstack	コンパイラーに、スタック・ストレージを圧縮するように指示します。これにより、ヒープ使用量が増大し、実行時間が延びる場合があります。ただし、プログラムを実行するか、またはプログラムを最適にマルチスレッド化しなければならない場合があります。
-qinline	インライン化を制御します。
-qunroll	ループのアンロールを独自に制御します。-O3 では、 **-qunroll** が暗黙のうちにアクティブになっています。
-qtbtable	トレースバック・テーブル情報の生成を制御します。
C++ **-qnoeh**	C++ 例外がスローされないこと、クリーンアップ・コードが省略できることを、コンパイラーに通知します。プログラムが C++ 例外をスローしない場合は、このオプションを使用して、例外処理コードを除去し、プログラムを圧縮してください。
-qnounwind	このコンパイル内のルーチンがアクティブである間はスタックがアンワインドされないことを、コンパイラーに通知します。このオプションを選択すると、不揮発性レジスターの保管と復元の最適化を改善できます。C++ では、**-qnounwind** オプションには **-qnoeh** オプションが暗黙指定されます。プログラムで `setjmp/longjmp` や他の形式の例外処理を使用する場合、このオプションは使用しないでください。
-qstrict	プログラムのセマンティクスを変更する変換をすべて無効にします。一般に、**-qstrict** およびいずれかのレベルの最適化を指定してプログラムを正しくコンパイルすると、最適化を指定しない場合と同じ結果が生成されます。
-qnostrict	コンパイラーが、浮動小数点計算と、除外される可能性のある命令をリオーダーするのを許可します。除外される可能性のある命令は、誤った実行 (例えば、浮動小数点のオーバーフロー、メモリー・アクセス違反など) によって割り込みを引き起こすことがあります。 **-qnostrict** は、デフォルトで **-O3** 以上の最適化レベルで使用されます。
-qprefetch	コンパイル済みコードにプリフェッチ命令を挿入して、コードのパフォーマンスを改善します。高いキャッシュ・ミス率が生成されるアプリケーションで作業している状態では、サブオプション **assistthread** を使用して (例えば、`-qprefetch=assistthread` のように)、プリフェッチ支援スレッドを生成できます。**-qnoprefetch** は、デフォルトのオプションです。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qignerrno

-qsmallstack

-qinline

-qunroll / #pragma unroll

- -qinlglue
- -qtbtable
- -qeh (C++ のみ)
- -qunwind
- -qstrict
- -qprefetch

第 9 章 最適化コードのデバッグ

最適化されたプログラムをデバッグする際には、デバッグ機能が使いづらくなるという特有の問題が生じます。最適化によって、演算の順序が変更されたり、コードが追加または削除されたり、変数データの場所が変更されたりするなど、生成されたコードを元のソース・ステートメントと関連付けることを難しくするような変換が行われることがあります。

次に例を示します。

データ位置の問題
最適化されたプログラムでは、変数の最新の値が置かれる場所が必ずしも定かではないことがあります。例えば、最新の値がレジスターに格納されている場合、メモリー内の値は現行値ではない可能性があります。ほとんどのデバッガーは、変数に関連付けられているストアの削除を追跡できないため、デバッガーにとっては、その変数がまったく更新されていないか、場合によってはまったく設定されていないと認識されます。これと対照的に、最適化を行わない場合はすべての値がメモリーに戻されるので、デバッグはより効率的に、より使いやすくなります。

命令スケジューリングの問題
最適化されたプログラムでは、コンパイラーが命令の順序を変更することがあります。つまり、元のソース・コード内の行の順序に基づいてプログラマーが予想する順序では、命令が実行されない可能性があります。また、ステートメントの命令シーケンスが連続していない可能性もあります。デバッガーでプログラムをステップスルーすると、プログラムは既に実行されたコード行に戻っているように見えることがあります (命令のインターリービング)。

変数値の整理
最適化を行うと、変数の除去や統合が行われることがあります。例えば、プログラムが 2 つの式を使用して、同じ値を 2 つの異なる変数に割り当てている場合、コンパイラーはその代わりに単一の変数を使用することがあります。この場合、プログラマーは存在すると思っている変数が、最適化されたプログラムでは使用できなくなっているために、デバッグ機能の使いやすさの妨げになります。

デバッグ機能を改善しながら、プログラムの最適化も行うためには、次に示すように数種類のアプローチを使用できます。

最適化されていないコードをまずデバッグする
最適化されていないバージョンのプログラムを最初にデバッグしてから、必要な最適化オプションを指定してそのプログラムを再コンパイルします。この方法で行う際に役に立つコンパイラー・オプションについては、85 ページの『最適化におけるデバッグ』を参照してください。

© Copyright IBM Corp. 1996, 2014

-g レベルを使用する

-g レベル・サブオプションを使用して、有効なデバッグ情報の量を制御することができます。この情報量を増やすとデバッグ機能が向上しますが、一部の最適化が無効になります。

-qoptdebug を使用する

-O3 レベル以上の最適化を伴うコンパイル時に、コンパイラー・オプション **-qoptdebug** を使用して、最適化されたプログラム内での命令および変数値の動作により正確にマップされる疑似コード・ファイルを生成します。このオプションを指定した場合は、プログラムをデバッガーにロードする際に、最適化されたプログラムの疑似コードをデバッグします。詳しくは、86 ページの『最適化プログラムをデバッグするのに役立つ -qoptdebug の使用』を参照してください。

最適化されたプログラムにおける異なる結果の理解

最適化されたプログラムが最適化プロセスを経ていないプログラムとは異なる結果を生じる理由のいくつかを次に示します。

- プログラムが無効なコードを含んでいる場合は、最適化されたコードが失敗する可能性があります。 最適化プロセスは、アプリケーションが言語標準に準拠していることを前提とします。
- 最適化なしで機能するプログラムが最適化時に失敗した場合は、プログラムの相互参照リストと実行フローから、初期化される前に使用される変数を調べてください。 **-qinitauto=***hex_value* オプションでコンパイルして、一貫して誤った結果を起こしてみてください。例えば、**-qinitauto=FF** を使用すると、「負の非数字」(-NAN) という初期値が変数に与えられます。これらの変数を使用した演算の結果も NAN 値になります。他のビット・パターン (***hex_value***) の場合はさまざまな結果が考えられ、その状況についてさらに有力な手掛かりが与えられます。コンパイラーが行うデフォルト解釈のため、未初期化の変数を含むプログラムは最適化なしでコンパイルされたとき正しく機能するように見えることがありますが、最適化時には失敗する可能性があります。同様に、最適化後にプログラムが正しく機能するように見えることがありますが、最適化レベルが低くなったり別の環境で実行されたときには失敗します。また、**-qcheck=unset** オプションおよび **-qinfo=unset** オプションを使用して、初期化されていない変数や初期化されていない可能性のある変数を検出することもできます。
- 所有関数がスコープから外れた後で自動ストレージ変数をそのアドレスによって参照すると、新規関数が呼び出されて他の自動変数がスコープに入ったときに上書きできるメモリー・ロケーションが参照されます。

-g8 オプションまたは **-g9** オプションが指定されており、最適化レベルが **-O2** である場合を除き、ストレージ内の値を調べることが前提となっているデバッグ技法を使用する際は、注意してください。コンパイラーが共通の式評価を削除したり移動したりした可能性があります。また、変数をレジスターに割り当てた結果、変数がストレージにまったく現れないことがあります。

最適化におけるデバッグ

必要な最適化オプションを使用して、プログラムのデバッグとコンパイルを行います。最適化されたプログラムを実稼働環境に移行する前に、プログラムのテストを実行します。 最適化されたコードが期待した結果を出さなかった場合は、デバッグ・セッションで個々の最適化問題の分離を試みることができます。

次のリストは専門情報を提供するオプションを示しています。これらのオプションは、最適化されたコードの作成時に役立つものです。

-qlist オブジェクト・リストの出力をコンパイラーに指示します。オブジェクト・リストには、生成された命令、トレースバック・テーブル、およびテキスト定数の 16 進および疑似アセンブリー表現が含まれます。

-qreport
コンパイラーが実行したループ変換、プログラムがどのように並列処理されたのか、どのようなインライン化が行われたのか、および他のいくつかの変換に関するレポートを作成するようコンパイラーに指示します。 **-qreport** によってリスト・ファイルを生成するには、オプション **-qreport**、**-qhot**、**-qsmp**、**-qinline**、**-qsimd**、または他の最適化オプションも指定する必要があります。

-qinfo=mt
並列コード内の潜在的な同期問題を報告します。詳しくは、「-qinfo」を参照してください。

-qinfo=unset
使用される自動変数を設定前に検出し、コンパイル時に通知メッセージによってフラグを立てます。詳しくは、「-qinfo」を参照してください。

-qipa=list
IPA 最適化の情報を提供するオブジェクト・リストの出力をコンパイラーに指示します。

-qcheck
特定のタイプのランタイム検査を実行するコードを生成します。

-qsmp=noopt
SMP コードをデバッグする場合は、**-qsmp=noopt** を指定すると、コンパイラーはコードの並列処理に必要な最小変換のみを行い、最大デバッグ機能を保持します。

-qoptdebug
高水準の最適化を指定して使用すると、デバッガーによって読み込まれる、最適化された疑似コードを含むファイルを作成します。

-qkeepparm
最適化時にもプロシージャー・パラメーターがスタックに格納されるようにします。これは実行パフォーマンスにマイナスの影響を与える可能性があります。そこで、**-qkeepparm** オプションは、単にそれらの値をスタックに保存することで、デバッガーなどのツールが入力されるパラメーターの値にアクセスできるようにします。

-qinitauto
すべての自動変数を与えられた値に初期化するコードの出力をコンパイラーに指示します。

-g シンボリック・デバッガーで使用するデバッグ情報を生成します。異なる **-g** レベルを使用して、デバッガー内の選択されたソース・ロケーションでアクセス可能な変数を表示または変更することにより、最適化されたコードをデバッグすることができます。**-g** のレベルを高くすると、より詳細なデバッグ・サポートが提供されますが、レベルを低くすると、実行時のパフォーマンスが向上します。詳しくは、**-g** を参照してください。

さらに、**snapshot** プラグマを使用して、アプリケーション内の各点で一定の変数がデバッガーに見えるようにすることもできます。詳しくは、『**#pragma ibm snapshot**』を参照してください。

最適化プログラムをデバッグするのに役立つ -qoptdebug の使用

-qoptdebug コンパイラー・オプションの目的は、最適化されたプログラムのデバッグを支援することにあります。これを行うため、元のソース・コードより厳密に、最適化されたプログラムの命令と値にマップされる疑似コードを作成します。このオプションでコンパイルされたプログラムがデバッガーにロードされたら、元のソースではなく疑似コードをデバッグします。疑似コード内で最適化を明示的にすることにより、最適化されたプログラムの実際の動作をいっそうよく理解することができます。プログラムの疑似コードを含むファイルは、ファイル・サフィックス `.optdbg` を付けて生成されます。この機能については行デバッグのみがサポートされます。

次の例に倣ってプログラムをコンパイルします。

```
xlc myprogram.c -O3 -qhot -g -qoptdebug
```

この例では、ソース・ファイルが a.out にコンパイルされます。最適化されたプログラムの疑似コードは `myprogram.optdbg` という名前のファイルに書き込まれます。このファイルはプログラムのデバッグ中に参照できます。

注:

- コンパイルされた実行可能ファイルをデバッグ可能にするためには、**-g** または **-qlinedebug** オプションも指定する必要があります。ただし、上記のいずれのオプションも指定されなかった場合でも、最適化された疑似コードを含む疑似コード・ファイル `<output_file>.optdbg` が生成されます。
- **-qoptdebug** オプションが影響を持つのは、最適化オプション **-qhot**、 **-qsmp**、**-qpdf** または **-qipa** の 1 つ以上が指定されたときか、これらのオプションを暗黙に示す最適化レベル、つまり最適化レベル **-O3**、**-O4**、および **-O5** が指定されたときに限られます。上記の例は、最適化オプション **-qhot** および **-O3** を示しています。

最適化されたプログラムのデバッグ

以下の例から、コンパイラーが最適化を単純プログラムに適用する方法およびそのデバッグと元のソースのデバッグとの相違点を理解できます。

例 1: 単純プログラムの元の最適化されていないコードを表します。これは、コンパイラーに最適化機会を与えます。ループはアンロールが可能です。最適化されたソースでは、明示的にリストされたループの反復を確認できます。

例 2: デバッガーに表示される、最適化されたソースのリストを表します。アンロールされたループ、および x + y 式によって割り当てられた値の統合に注意してください。

例 3: デバッガーを使用して、最適化されたソースをステップスルーする例を示します。元のソースの行番号と比べると、最適化されたソース内のステートメントの行番号の間にはもはや対応がないことに注意してください。

例 1: 元のコード

```
#include "stdio.h"

void foo(int x, int y, char* w)
{
 char* s = w+1;
 char* t = w+1;
 int z = x + y;
 int d = x + y;
 int a = printf("TEST¥n");

 for (int i = 0; i < 4; i++)
  printf("%d %d %d %s %s¥n", a, z, d, s, t);
 }

int main()
{
 char d[] = "DEBUG";
 foo(3, 4, d);
 return 0;
}
```

例 2: gdb デバッガー・リスト

```
(gdb) list
1            3 |  void foo(long x, long y, char * w)
2            4 |  {
3            9 |    a = printf("TEST¥n");
4           12 |    printf("%d %d %d %s %s¥n",a,x + y,x + y,
                         ((char *)w  + 1),((char *)w  + 1));
5                  printf("%d %d %d %s %s¥n",a,x + y,x + y,
                         ((char *)w  + 1),((char *)w  + 1));
6                  printf("%d %d %d %s %s¥n",a,x + y,x + y,
                         ((char *)w  + 1),((char *)w  + 1));
7                  printf("%d %d %d %s %s¥n",a,x + y,x + y,
                         ((char *)w  + 1),((char *)w  + 1));
8           13 |   return;
9                 } /* function */
10
11
12          15 |  long main()
13          16 |  {
14          17 |    d$init$0 = "DEBUG";
15          18 |    $$PARM.x0 = 3;
16                 $$PARM.y1 = 4;
17                 $$PARM.w2 = &d;
18           9 |    a = printf("TEST¥n");
19          12 |    printf("%d %d %d %s %s¥n",a,$$PARM.x0 +
                          $$PARM.y1,$$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                          ((char *)$$PARM.w2  + 1));
20                 printf("%d %d %d %s %s¥n",a,$$PARM.x0 + $$PARM.y1,
```

```
                          $$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                          ((char *)$$PARM.w2  + 1));
21                  printf("%d %d %d %s %s¥n",a,$$PARM.x0 + $$PARM.y1,
                          $$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                          ((char *)$$PARM.w2  + 1));
22                  printf("%d %d %d %s %s¥n",a,$$PARM.x0 + $$PARM.y1,
                          $$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                          ((char *)$$PARM.w2  + 1));
23          19 |    rstr = 0;
24                  return rstr;
25          20 |  } /* function */
```

例 3: 最適化されたソースのステップスルー

```
(gdb) break 17
Breakpoint 2 at 0x10000694: file myprogram.o.optdbg, line 17.
(gdb) run
Starting program: /nfs/r3lp1/home/gklou/tmp/optdbg/a.out

Breakpoint 2, main () at myprogram.o.optdbg:18
18           9 |    a = printf("TEST¥n");
(gdb) continue
Continuing.
TEST
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG

Program exited normally.
```

第 10 章 パフォーマンスを向上させるためのアプリケーションのコーディング

45 ページの『第 8 章 アプリケーションの最適化』では、最小限のコーディングでコードを最適化するために XL C/C++ コンパイラーが提供する各種のコンパイラー・オプションについて説明しています。アプリケーションをもう一歩進めて、コンパイラーの最適化を補完したり最大限に利用したりする場合は、以下の節で説明する C および C++ プログラミング手法を使用すれば、コードのパフォーマンスを向上させることができます。

- 『高速入出力手法の検出』
- 90 ページの『関数呼び出しによるオーバーヘッドの低減』
- 91 ページの『委任コンストラクター (C++11) の使用』
- 92 ページの『テンプレート明示的インスタンス生成宣言の使用 (C++11)』
- 92 ページの『効率的なメモリーの管理』
- 93 ページの『変数の最適化』
- 94 ページの『効率的なストリングの操作』
- 94 ページの『式とプログラム・ロジックの最適化』
- 95 ページの『64 ビット・モードでの演算の最適化』
- 96 ページの『コード内のトレース関数』
- 101 ページの『右辺値参照の使用 (C++11)』
- 103 ページの『可視属性の使用 (IBM 拡張)』

高速入出力手法の検出

プログラムの入出力のパフォーマンスを向上するには、いくつか方法があります。

- ファイル入出力アクセスが局所性を持っていない場合 (つまり、データベース内でのアクセスのように、まったくのランダム・アクセスである場合)、独自のバッファー・メカニズムまたはキャッシュ・メカニズムを下位の入出力関数に実装します。
- ユーザー独自の入出力バッファリングを行う場合、バッファーをページの最小サイズである 4 K の倍数にする。
- バッファーされた入出力を使用してテキスト・ファイルを処理する。
- ファイル全体を処理する必要があることを認識している場合は、読み取られるデータのサイズを判別し、これを読み取る単一バッファーを割り当て、`read` を使用してファイル全体をそのバッファーに一度に読み取り、それからバッファー内のデータを処理する。 過度のスワッピングが起こるほどファイルが大きくなければ、これでディスク入出力が削減されます。ファイルにアクセスするために `mmap` 関数を使用することも考えてください。

© Copyright IBM Corp. 1996, 2014

関数呼び出しによるオーバーヘッドの低減

関数を作成するか、またはライブラリー関数を呼び出すときには、次のガイドラインを考慮してください。

- 関数ポインターを使用する代わりに、関数を直接呼び出す。
- 可能であれば、インライン化された関数で const の引数を常に使用する。定数の引数を持つ関数によって、最適化の機会が広がります。
- **#pragma expected_value** プリプロセッサー・ディレクティブを使用して、関数に使われる共通値をコンパイラーで最適化できるようにする。
- **#pragma isolated_call** プリプロセッサー・ディレクティブを使用して、副次作用がなく、副次作用に依存しない関数をリストする。
- 同じメモリーをポイントする可能性がないポインターについては、restrict キーワードを使用する。
- ポインターの関数内の **#pragma disjoint**、または同じメモリーを指すことのできない参照パラメーターを使用する。
- 可能であれば、非メンバー関数を static として宣言する。これによって関数の呼び出しが高速になり、関数がインライン化される可能性が高まります。
- C++ 通常は、すべての仮想関数をインラインで宣言することはしない。クラス内の仮想関数がすべてインラインである場合は、仮想関数テーブルとすべての仮想関数本体が、クラスを使用する各コンパイル単位で複製されます。
- C++ 可能であれば、関数を宣言するときにはいつも const 指定子を使用する。
- C すべての関数を完全にプロトタイプ化する。完全なプロトタイプは、コンパイラーおよび最適化プログラムに、パラメーターの型について完全な情報を与えます。結果として、広げられない型から広げられた型へのプロモーションは必要なく、パラメーターが適切なレジスターに渡されます。
- C プロトタイプ化されていない変数引数の関数の使用を避ける。
- パラメーターの数を少なくし、最も頻繁に使用されるパラメーターが、関数プロトタイプの一番左端に位置するように関数を設計する。
- 値を使用して大きな構造体または共用体を関数仮パラメーターとして渡すことを避ける、あるいは、大きな構造体または共用体を返すことを避ける。 このような集合体を渡すと、コンパイラーは多数の値のコピーおよび保管を行わなければならなくなります。このことは、クラス・オブジェクトが値によって渡される C++ プログラムではより不適切です。コンストラクターとデストラクターは、関数が呼び出されるときに呼び出されるからです。その代わりに、ポインターを構造体か共用体に渡すか、または戻すか、あるいは参照によってこれを渡すようにします。
- 可能な場合は、int や short などの非集合体の型、または小さな集合体は、参照では渡すのではなく、常に値で渡すようにする。
- ある関数が、その関数に渡されたものと同じパラメーターを指定して、別の関数の値を戻すことによって終了する場合は、関数プロトタイプ内でそのパラメーターを同じ順序にする。これにより、コンパイラーは、他の関数に直接ブランチすることができます。

- 独自の関数をコーディングする代わりに、ストリング処理、浮動小数点、および三角関数を含む、組み込み関数を使用します。組み込み関数では、必要なオーバーヘッドはより少なく、関数呼び出しより高速であり、またコンパイラーがよりよい最適化を実行できる場合があります。

  C++ C++ 標準ライブラリーの多くの関数は、コンパイラーにより、最適化された組み込み関数にマップされます。

  C `string.h` と `math.h` の多くの関数は、コンパイラーにより、最適化された組み込み関数にマップされます。
- `inline` キーワードを使用して、インライン用の関数を選択的にマーク付けする。インラインになった関数は、必要なオーバーヘッドがより少なく、一般的に関数呼び出しより高速です。インライン化の最も有力な候補は、少数の場所から頻繁に呼び出される小さな関数、あるいは、 1 つ以上のコンパイル時の定数パラメーター、特に `if` 文、`switch` 文、または `for` 文に影響を与えるパラメーターで呼び出される関数です。これらの関数は、ヘッダー・ファイルに入れることもできます。そうすると、最適化レベルが低い場合でも、ファイル境界を越えて自動インライン化ができるようになります。単に値をロードまたは保管するだけの関数は、すべてインライン化するようにしてください。あるいは、比較演算子や算術演算子のような単純な演算子を使用してください。大きな関数やめったに呼び出されない関数は、通常、インラインの候補としては適しません。多くの場所から呼び出される中規模の関数も、適していません。
- プログラムを多くの小さな関数に分けることは避ける。小さな関数を使用する必要がある場合は、 **-qipa** コンパイラー・オプションの使用を真剣に検討してください。このオプションを使用すると、このような関数を自動でインライン化することができ、関数間の呼び出しを最適化するその他の手法が使用されます。
- C++ クラス拡張性のために必要な場合を除いて、仮想関数および仮想継承は避ける。これらの言語機能は、オブジェクト・スペースおよび関数呼び出しのパフォーマンスの点で負担がかかります。

  「*XL C/C++ コンパイラー・リファレンス*」の関連情報

  #pragma expected_value

  -qisolated_call / #pragma isolated_call

  #pragma disjoint

  -qipa

委任コンストラクター (C++11) の使用

注: IBM は、C++11 (標準化前の呼称は C++0x) の一部の機能をサポートしています。IBM は引き続き、この標準の機能を開発および実装する予定です。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含めて、C++11 のすべての機能の実装を IBM が完了するまでは、リリースごとに実装が変更される可能性があります。IBM は、ソース、バイナリー、リストなどのコンパイラー・インターフェースにおいて、新しい C++11 機能の IBM 実装の旧リリースとの互換性を維持することを試みません。

委任コンストラクター機能を使用して、共通の初期化を 1 つのコンストラクターに集中させます。この機能は、コード・サイズを減少させてプログラムの可読性および保守可能性を高めるのに役立ちます。

この手法については、27 ページの『委任コンストラクター (C++11) の使用』で説明します。

テンプレート明示的インスタンス生成宣言の使用 (C++11)

注: IBM は、C++11 (標準化前の呼称は C++0x) の一部の機能をサポートしています。IBM は引き続き、この標準の機能を開発および実装する予定です。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含めて、C++11 のすべての機能の実装を IBM が完了するまでは、リリースごとに実装が変更される可能性があります。IBM は、ソース、バイナリー、リストなどのコンパイラー・インターフェースにおいて、新しい C++11 機能の IBM 実装の旧リリースとの互換性を維持することを試みません。

明示的インスタンス生成宣言機能を使用して、テンプレート特殊化およびそのメンバーの暗黙のインスタンス生成を抑制します。この機能は、オブジェクト・ファイルの総体サイズを減少させ、コンパイル時間を短縮するのに役立ちます。

この手法については、35 ページの『明示的インスタンス生成宣言の使用 (C++11)』で説明します。

効率的なメモリーの管理

C++ オブジェクトは、しばしばヒープから割り当てられ、有効範囲が制限されているため、C++ プログラムでのメモリー使用は、C プログラムでのメモリー使用よりもパフォーマンスに影響を与えます。そのため、C++ アプリケーションを開発するときは、以下のガイドラインを考慮してください。

- 構造体では、最もサイズの大きい位置合わせ済みのメンバーから順に宣言する。可能な場合、類似した位置合わせのメンバーをグループ化する必要があります。
- 構造体では、一緒に使用する頻度が高い変数は、それぞれ互いに近くに置く。
- C++ 必要なくなったオブジェクトが、確実に解放されるか、そうでなければ再利用のために使用できるようにする。これを行う方法の 1 つに、オブジェクト・マネージャー の使用があります。オブジェクトのインスタンスを作成するたびに、そのオブジェクトへのポインターをオブジェクト・マネージャーに渡します。オブジェクト・マネージャーは、それらのポインターのリストを保守します。オブジェクトにアクセスするには、オブジェクト・マネージャーのメンバー関数を呼び出して、ユーザーまで情報を戻させます。するとオブジェクト・マネージャーは、メモリーの使用量やオブジェクト再使用を管理します。
- ストレージ・プールは、オブジェクト・マネージャーまたは参照カウントに頼らずに、使用されているメモリーを追跡する (そしてそれを再利用する) にはよい方法です。自明でない (non-trivial) デストラクターを持つオブジェクトに対してストレージ・プールを使用しないでください。ほとんどの実装では、そのストレージ・プールがクリアされたときに、これらのデストラクターを実行できないからです。

- C++ 大きな、複雑なオブジェクトのコピーは避ける。
- C++ シャロー・コピー だけが必要な場合は、ディープ・コピー を実行しないようにする。他のオブジェクトへのポインターを含むオブジェクトについて、シャロー・コピーはポインターだけをコピーし、それらが指すオブジェクトをコピーしません。その結果、同じものが含まれたオブジェクトを指す 2 つのオブジェクトができます。しかしディープ・コピーは、そのオブジェクト内に含まれているすべてのポインターやオブジェクトなどと同様に、ポインターとそれが指すオブジェクトをコピーします。ディープ・コピーは共有と同期の妨げになるため、マルチスレッド環境で実行する必要があります。
- C++ どうしても必要なときにのみ仮想メソッドを使用する。
- C++ 「Resource Acquisition is Initialization: RAII (リソース確保は初期化である)」パターンを使用する。
- `shared_ptr` と `weak_ptr` を使用する。

変数の最適化

次のガイドラインを考慮してください。

- できるかぎりローカル変数 (自動変数が望ましい) を使用する。

  コンパイラーは、グローバル変数について、いくつかのワーストケースの想定をする必要があります。例えば、ある関数で外部変数を使用して外部関数を呼び出す場合、コンパイラーは、それぞれの外部関数の呼び出しで、それぞれの外部変数の値が使用または変更される可能性があるものと想定します。グローバル変数がどの関数呼び出しからも読み込まれず、影響も受けないこと、および、分散された関数呼び出しでこの変数が複数回にわたって読み込まれることが分かっている場合は、そのグローバル変数をローカル変数にコピーしてから、このローカル変数を使用してください。
- グローバル変数を使用しなければならない場合は、可能であれば、外部変数ではなく、ファイル有効範囲を指定した静的変数を使用してください。複数の関連する関数と静的変数を指定したファイルでは、変数の受ける影響について、最適化プログラムがより多くの情報を集めて使用することができます。
- 外部変数を使用しなければならない場合には、そうすることに意味があれば、外部データを構造体または配列にグループ化する。外部構造のエレメントはすべて、同じ基底アドレスを使用します。アドレスが取られている変数を、アドレスが取られていない変数と一緒にグループ化しないでください。
- **#pragma isolated_call** プリプロセッサー・ディレクティブは、コンパイラーに、外部変数および静的変数のストレージについてあまり悲観的でない前提事項を作成させることによって、最適化コードの実行時パフォーマンスを向上することができます。 定数または不変ループのパラメーターを指定した Isolated_call 関数がループから移動し、同じパラメーターを指定した複数の呼び出しが単一呼び出しで置き換えられます。
- 変数のアドレスをとることを避ける。ローカル変数を一時変数として使用しており、そのアドレスをとらなければならない場合、その一時変数を別の目的で再利用することは避けてください。ローカル変数のアドレスをとると、アドレスをとらなければその変数にかかわる計算で行われるはずの最適化が禁止されることがあります。

- 可能な場合は変数の代わりに定数を使用する。最適化プログラムは、代わりにコンパイル時にこれを行って、実行時の計算を削減し、よりよいジョブの実行を可能にします。例えば、ループ本体で反復回数が定数である場合は、ループ条件に定数を使用して、最適化を向上させます (`for (i=0; i<4; i++)` は、`for (i=0; i<x; i++)` よりもよく最適化されます)。列挙型宣言を使用して名前付き定数を宣言すると、保守容易性を向上することができます。
- 64 ビット・モードでの各変更後における符号拡張命令を避けるため、スカラーにはレジスター・サイズの整数 (`long` データ型) を使用する。大きな整数配列には、1 バイトまたは 2 バイトの整数、あるいはビット・フィールドの使用を検討してください。
- 計算に適した、最小の浮動小数点精度を使用する。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報


-qisolated_call / #pragma isolated_call

効率的なストリングの操作

ストリング操作の処理が、プログラムのパフォーマンスに影響を与えることがあります。

- 割り当てられたストレージにストリングを保管するときは、ストリングの開始を 8 バイトの境界に位置合わせする。
- ストリングの長さを常に把握しておく。ストリングの長さが分かっている場合は、`str` 関数の代わりに `mem` 関数を使用することができます。例えば、`memcpy` が `strcpy` より高速なのは、ストリングの終わりを検索する必要がないためです。
- ソースとターゲットがオーバーラップしないことが確かな場合には、`memmove` の代わりに、`memcpy` を使用する。これは、`memcpy` がソースから宛先に直接コピーするのに対し、 `memmove` は、ソースをメモリー内の一時ロケーションにコピーしてから、宛先にコピーすることがあるためです (ストリングの長さによります)。
- `mem` 関数を使用してストリングを処理するときに、*count* パラメーターが変数でなく定数であれば、より高速なコードが生成される。これは、小カウント値の場合に特に当てはまります。
- 可能であれば、ストリング・リテラルを読み取り専用にする。同じストリングを複数回使用する場合、そのストリングを読み取り専用にすることで、特定の最適化技法の向上、メモリー使用量の削減、コンパイル時間の短縮を図ることができます。ストリングを明示的に読み取り専用に設定するには、ソース・ファイルで **#pragma strings (readonly)** を使用するか、ソース・ファイルの変更を避ける場合は **-qro** (デフォルトで有効になっていますが C cc を使用したコンパイル時は除きます C) を使用します。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報


-qro / #pragma strings

式とプログラム・ロジックの最適化

次のガイドラインを考慮してください。

- ある式のコンポーネントが別の式で使用され、そのコンポーネントに関数呼び出しが含まれている場合、またはそのコンポーネントを使用してから次に使用するまでの間に関数呼び出しが存在する場合は、重複する値をローカル変数に割り当てる。
- コンパイラーに、整数と浮動小数点の内部表記の間で数字を変換するように強制することは避ける。次に例を示します。

```
float array[10];
float x = 1.0;
int i;
for (i = 0; i< 9; i++)  {     /* No conversions needed */
     array[i] = array[i]*x;
     x = x + 1.0;
    }
for (i = 0; i< 9; i++)  {       /* Multiple conversions needed */
     array[i] = array[i]*i;
    }
```

  混合モードの算術演算を使用しなければならないときは、可能ならば、整数と浮動小数点の算術演算を別の計算でコーディングしてください。
- グローバル変数をループ指標または境界として使用しないようにします。
- ループの真ん中にジャンプする goto 文は避けます。このような文は決まった最適化を禁止します。
- フォールスルー・パスの発生確率を高めることによって、コードの予測可能性を向上させる。次のコードがあるとした場合には、

```
if (error) {handle error} else {real code}
```

  次のようにコーディングする必要があります。

```
if (!error) {real code} else {error}
```

- switch 文の 1 つか 2 つのケースが一般的に他のケースより頻繁に実行される場合は、switch 文の前に別々に処理して、これらのケースを抜き出す。配列から取得される範囲内に値が収まっているかどうかを確認し、可能であれば、switch 文を置き換えてください。
- C++ 最適化が禁止される可能性があるため、必要なときにだけ、例外ハンドリングに try ブロックを使用する。
- 配列指標式は可能な限り単純にする。

64 ビット・モードでの演算の最適化

ディスク入出力に頼らず、物理メモリー内で直接大量のデータを処理できるということは、おそらく、 64 ビット・マシンのパフォーマンス上最大の利点でしょう。しかし、アプリケーションによっては、64 ビット・モードで再コンパイルしたときよりも、32 ビット・モードでコンパイルした方が優れたパフォーマンスを示す場合があります。これには次のようないくつかの理由があります。

- 64 ビット・プログラムの方が大きい。プログラム・サイズの増加により、物理メモリーの要求がより大きくなります。
- 64 ビットの long 型除法の方が、32 ビットの整数除法よりも時間がかかる。
- 32 ビットの符号付き整数を配列指標またはループ・カウントとして使用する 64 ビット・プログラムの場合、配列が参照されるたび、またはループ・カウントが増分されるたびに、符号拡張を行うための追加の命令が必要になる場合がある。

64 ビット・プログラムのパフォーマンス上の不利を補うには、次のような方法があります。

- 32 ビットと 64 ビット混合の演算を行わないようにする。例えば、 32 ビットのデータ型と 64 ビットのデータ型を加算する場合は、32 ビットの方を符号拡張し、レジスターの上位 32 ビットをクリアまたは設定する必要があります。このため、計算が遅くなります。
- 頻繁にアクセスされる変数 (ループ・カウンター、配列指標など) には、`signed`、`unsigned`、および単純な `int` などの型ではなく、 `long` 型を使用する。このようにすると、コンパイラーが、配列参照、関数呼び出し中のパラメーター、および戻される関数結果を、切り捨てたり符号拡張したりする必要がなくなります。

コード内のトレース関数

ユーザー定義のトレース関数の呼び出しを挿入するようにコンパイラーに指示して、その他の関数をデバッグしたり、それらの実行のタイミングをとったりする際に役立てることができます。

プログラムでトレース関数を使用するには、以下のことを行う必要があります。

1. トレース関数を作成する。
2. **-qfunctrace** オプションで、トレース対象の関数を指定する。

-qfunctrace オプションを使用すると、コンパイラーが、これらのトレース関数の呼び出しを関数本体のキー・ポイントに挿入します。ただし、これらのトレース関数はユーザーが定義する必要があります。以下のリストで、トレース関数が呼び出されるポイントについて説明します。

- コンパイラーは、トレース関数の呼び出しを関数の入り口点に挿入します。ルーチンに渡される行番号は、機能化関数の最初の実行可能ステートメントの行番号です。
- コンパイラーは、トレース関数の呼び出しを関数の出口点に挿入します。関数に渡される行番号は、機能化関数を終了させる原因となったステートメントの行番号です。
- キャッチ・トレース関数は、例外が発生したときに、C++ の catch ブロックの最初で呼び出されます。

-qnofunctrace コンパイラー・オプションまたは `#pragma nofunctrace` プラグマを使用して、関数のトレースを無効にできます。

トレース関数の作成方法

コード内の関数をトレースするには、以下のトレース関数を定義します。

- `__func_trace_enter` は、入り口点トレース関数です。
- `__func_trace_exit` は、出口点トレース関数です。
- `__func_trace_catch` は、キャッチ・トレース関数です。

これらの関数のプロトタイプは、以下のとおりです。

- `void __func_trace_enter(const char *const function_name, const char *const file_name, int line_number, void **const user_data);`

- `void __func_trace_exit(const char *const function_name, const char *const file_name, int line_number, void **const user_data);`
- `void __func_trace_catch(const char *const function_name, const char *const file_name, int line_number, void **const user_data);`

これらのトレース関数の変数について、以下で説明します。

- `function_name` は、トレースする対象の関数の名前です。
- `file_name` は、ファイルの名前です。
- `line_number` は、関数の入り口点または出口点の行番号です。これは、4 バイトの数値です。
- `user_data` は、静的ポインター変数のアドレスです。静的ポインター変数は、コンパイラーが生成し、`NULL` に初期化されます。また、ポインター変数が静的であるため、そのアドレスは、同じ関数内のすべてのインスツルメンテーション呼び出しに対して同じです。

注:

- 関数が異常終了した場合、出口機能は呼び出されません。 異常終了の原因となり得るアクションは、シグナルの発行や、C++ 例外のスロー、exit() 関数または abort() 関数の呼び出しなどです。
- **-qfunctrace** オプションは、`setjmp` および `longjmp` をサポートしません。例えば、function1 を終了して function2 の `setjmp()` から戻る `longjmp()` を呼び出した場合、function1 の `__func_trace_exit` と function2 の `__func_trace_enter` は呼び出されません。
- キャッチ関数は、C++ の例外がユーザー・コードでキャッチされたポイントで呼び出されます。
- C++ プログラムでトレース関数を定義するには、関数定義の前で `extern "C"` リンケージ・ディレクティブを使用します。
- 関数呼び出しは、関数定義にのみ挿入されます。関数がインライン化される場合、インライン化されたコード内ではトレースが行われません。
- マルチスレッド・プログラムを開発する場合、トレース関数が適切に同期され、デッドロックを引き起こさないようにする必要があります。。トレース関数の呼び出しは、スレッド・セーフではありません。
- このオプションで、存在しない関数を指定した場合、その関数は無視されます。

規則

コード内の関数をトレースするときは、以下の規則が適用されます。

- 最適化が有効である場合、行番号は正確でないことがあります。
- トレース関数では、機能化関数を一切呼び出さないでください。呼び出すと、無限ループが発生する可能性があります。
- 再帰的関数をトレースするようにコンパイラーに指示する場合は、トレース関数が再帰を処理できることを確認してください。
- インライン化された関数は、機能化されません。
- トレース関数は、機能化されません。

- コンパイラー生成関数は機能化されません。ただし、OpenMP など、最適化で生成されてアウトライン化された関数は除きます。この場合、アウトライン化された関数の名前には、元のユーザー関数の名前が接頭部として含まれます。
- トレース関数は、静的初期化で呼び出されることがあります。トレース関数で使用されるすべてのものが、トレース関数の最初の呼び出しより前に初期化されるように、十分に注意してください。

例

以下の C の例で、関数プロトタイプを使用してコードの関数をトレースする方法を示します。function1 と function2 の入り口点と出口点のトレースを行い、以下のコードでコンパイラーがトレースに消費した時間をトレースするものとします。

メインプログラム・ファイル: t1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#ifdef __cplusplus
extern "C"
#endif
void __func_trace_enter(const char *function_name, const char *file_name,
                        int line_number, void** const user_data){
   if((*user_data)==NULL)
       (*user_data)=(time_t *)malloc(sizeof(time_t));
   (*(time_t *)*user_data)=time(NULL);
   printf("begin function: name=%s file=%s line=%d¥n",function_name,file_name,
          line_number);
}
#ifdef __cplusplus
extern "C"
#endif
void __func_trace_exit(const char *function_name, const char*file_name,
                       int line_number, void** const user_data){
   printf("end function: name=%s file=%s line=%d. It took %g seconds¥n",
          function_name,file_name,line_number, difftime(time(NULL),
          *(time_t *)*user_data));
}
void function2(void){
   sleep(3);
}
void function1(void){
   sleep(5);
   function2();
}
int main(){
   function1();
}
```

次のようにメインプログラムのソース・ファイルをコンパイルします。

```
xlc t1.c -qfunctrace+function1:function2
```

実行可能ファイル a.out を実行して、関数のトレース結果を出力します。

```
begin function: name=function1 file=t.c line=27
begin function: name=function2 file=t.c line=24
end function: name=function2 file=t.c line=25. It took 3 seconds
end function: name=function1 file=t.c line=29. It took 8 seconds
```

前の例でわかるように、時間計算の基本としてシステム時刻を使用するために、user_data パラメーターが定義されています。以下のステップで、user_data を定義してこの目的を達成する方法を説明します。

1. 関数で、user_data の値を保管するメモリー領域を予約します。
2. システム時刻を user_data の値として使用します。
3. __func_trace_exit 関数で、difftime 関数が user_data を使用して時差を計算します。結果は It took %g seconds の形式で出力に表示されます。

以下の C++ の例で、トレース関数を呼び出す方法を示します。以下の例では、クラス myStack と関数 foo をトレースし、#pragma nofunctrace で int main() のトレースを無効にしています。

メインプログラム・ファイル: t2.cpp

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
extern "C"
void __func_trace_enter(const char *function_name, const char *file_name,
                        int line_number, void** const user_data){
  if((*user_data)==NULL)
    (*user_data)=(time_t *)malloc(sizeof(time_t));
  (*(time_t *)*user_data)=time(NULL);
  printf("enter function: name=%s file=%s line=%d¥n",function_name,file_name,
         line_number);
}
extern "C"
void __func_trace_exit(const char *function_name, const char*file_name,
                       int line_number, void** const user_data){
  printf("exit function: name=%s file=%s line=%d. It took %g seconds¥n",
         function_name, file_name, line_number, difftime(time(NULL),
         *(time_t *)*user_data));
}
extern "C"
void __func_trace_catch(const char *function_name, const char*file_name,
                        int line_number, void** const user_data){
  printf("catch function: name=%s file=%s line=%d. It took %g seconds¥n",
         function_name, file_name,line_number, difftime(time(NULL),
         *(time_t *)*user_data));
}

template <typename T> class myStack{
  private:
  std::vector<T> elements;
  public:
  void push(T const&);
  void pop();
};

template <typename T>
void myStack<T>::push(T const& value){
  sleep(3);
  std::cout<< "¥tpush(" << value << ")" <<std::endl;
  elements.push_back(value);
}
template <typename T>
void myStack<T>::pop(){
  sleep(5);
  std::cout<< "¥tpop()" <<std::endl;
```

```
  if(elements.empty()){
    throw std::out_of_range("myStack is empty");
  }
  elements.pop_back();
}
void foo(){
  myStack<int> intValues;
  myStack<float> floatValues;
  myStack<double> doubleValues;
  intValues.push(4);
  floatValues.push(5.5f);
  try{
    intValues.pop();
    floatValues.pop();
    doubleValues.pop(); // 例外を発生させる
  } catch(std::exception const& e){
    std::cout<<"¥tException: "<<e.what()<<std::endl;
  }
  std::cout<<"¥tdone"<<std::endl;
}
#pragma nofunctrace(main)
int main(){
  foo();
}
```

次のようにメインプログラムのソース・ファイルをコンパイルします。

```
xlC t2.cpp -qfunctrace+myStack:foo
```

実行可能ファイル a.out を実行して、関数のトレース結果を出力します。

```
  enter function: name=_Z3foov file=t2.cpp line=56
  enter function: name=_ZN7myStackIiE4pushERKi file=t2.cpp line=42
          push(4)
  exit function: name=_ZN7myStackIiE4pushERKi file=t2.cpp line=45. It took 3 seconds
  enter function: name=_ZN7myStackIfE4pushERKf file=t2.cpp line=42
          push(5.5)
  exit function: name=_ZN7myStackIfE4pushERKf file=t2.cpp line=45. It took 3 seconds
  enter function: name=_ZN7myStackIiE3popEv file=t2.cpp line=48
          pop()
  exit function: name=_ZN7myStackIiE3popEv file=t2.cpp line=54. It took 5 seconds
  enter function: name=_ZN7myStackIfE3popEv file=t2.cpp line=48
          pop()
  exit function: name=_ZN7myStackIfE3popEv file=t2.cpp line=54. It took 5 seconds
  enter function: name=_ZN7myStackIdE3popEv file=t2.cpp line=48
          pop()
  catch function: name=_Z3foov file=t2.cpp line=65. It took 21 seconds
          Exception: myStack is empty
          done
  exit function: name=_Z3foov file=t2.cpp line=69. It took 21 seconds
```

関連情報

- **-qfunctrace** コンパイラー・オプションについて詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の -qfunctrace を参照してください。
- #pragma nofunctrace について詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の #pragma nofunctrace を参照してください。

右辺値参照の使用 (C++11)

注: IBM は、C++11 (標準化前の呼称は C++0x) の一部の機能をサポートしています。IBM は引き続き、この標準の機能を開発および実装する予定です。この言語レベルの実装は、IBM による標準の解釈に基づいています。新しい C++11 標準ライブラリーのサポートを含めて、C++11 のすべての機能の実装を IBM が完了するまでは、リリースごとに実装が変更される可能性があります。IBM は、ソース、バイナリー、リストなどのコンパイラー・インターフェースにおいて、新しい C++11 機能の IBM 実装の旧リリースとの互換性を維持することを試みません。

C++11 では、関数を引数の値のカテゴリーに基づいて多重定義でき、同様にテンプレート引数の推論によって検出された左辺値性を保持することができます。右辺値を右辺値参照にバインドし、その右辺値を参照によって変更することもできます。これにより、期限が切れるオブジェクトのリソースの再利用を可能にするプログラミング手法を使用することができます。その結果、特にテンプレート・データ構造体などのクラス型を含む汎用コードを使用する場合に、ライブラリーのパフォーマンスを向上させることができます。また、転送関数を記述するときに、値のカテゴリーを考慮することができます。

移動セマンティクス

一時的な値の使用を最適化したい場合に、移動操作を破壊的コピーと呼ばれる処理で使用することができます。以下の文字列の連結と代入を考えてみます。

```
std::string a, b, c;
c = a + b;
```

このプログラムでは、コンパイラーはまず `a + b` の結果を内部一時変数、すなわち右辺値に格納します。

通常のコピー代入演算子のシグニチャーは、次のようになります。

```
string& operator = (const string&)
```

このコピー代入演算子では、代入は次のステップから成ります。

1. ディープ・コピー操作を使用して、一時変数を `c` にコピーする。
2. 一時変数を破棄する。

一時変数は次のステップで破棄されるため、一時変数を `c` にディープ・コピーすることは効率的ではありません。

一時変数の不必要なコピーを避けるために、変数をコピーする代わりに、変数を移動する代入演算子を実装することができます。つまり、演算子の引数が操作によって変更されます。移動操作はポインター処理によって行われるため、コピーより高速になります。ただし、移動操作には、ソース変数の処理を可能にする参照が必要です。しかし、`a + b` は一時的な値であり、C++11 以前の C++ で多重定義の解決のため const で修飾された値と区別しにくくなっています。

右辺値参照では、移動代入演算子を以下のように作成できます。

```
string& operator= (string&&)
```

この移動代入演算子では、 a + b の結果の中の基になる C スタイルの文字列に割り当てられているメモリーが c に割り当てられます。したがって、c の基になる文字列を格納するために新しいメモリーを割り当て、その内容を新しいメモリーにコピーする必要がありません。

以下のコードは、string 移動代入演算子の実装になります。

```
string& string::operator=(string&& str)
{
  // The named rvalue reference str acts like an lvalue
  std::swap(_capacity, str._capacity);
  std::swap(_length, str._length);

  // char* _str points to a character array and is a
  // member variable of the string class
  std::swap(_str, str._str);
  return *this;
}
```

ただし、この実装では、メモリーが割り当てられる文字列が元々保持していたメモリーは、str が破棄されるまでは解放されません。ローカル変数を使用する以下の実装では、よりメモリー効率が良くなります。

```
string& string::operator=(string&& parm_str)
{
  // The named rvalue reference parm_str acts like an lvalue
  string sink_str;
  std::swap(sink_str, parm_str);
  std::swap(*this, sink_str);
  return *this;
}
```

同様に、以下のプログラムは string 連結演算子の実現可能な実装です。

```
string operator+(string&& a, const string& b)
{
  return std::move(a+=b);
}
```

注: std::move 関数は a+=b の結果を右辺値参照にキャストするだけで、何も移動しません。式 std::move(a+=b) は右辺値であるため、戻り値は、移動コンストラクターを使用して構成されます。移動コンストラクターとコピー・コンストラクター間の関係は、移動代入演算子とコピー代入演算子間の関係に似ています。

完全転送

std::forward 関数は、 std::move とよく似たヘルパー・テンプレートです。この関数は、自身の関数引数への参照を返します。このとき、結果の値のカテゴリーはテンプレート型引数によって決定されています。転送関数テンプレートのインスタンス化では、引数の値のカテゴリーは関連するテンプレート型パラメーターに対して推論された型の一部としてエンコードされます。推論された型は、std::forward 関数に渡されます。

以下の例に示す wrapper 関数は、do_work 関数に転送を行う転送関数テンプレートです。 std::forward は、転送関数内で宛先関数の呼び出し時に使用します。以下の例では、decltype 機能と trailing return type 機能も使用して、do_work 関数のいずれかに転送を行う転送関数を生成します。任意の引数を指定して wrapper 関数を呼び出すと、適切な多重定義関数が存在する場合、do_work 関数が呼び出されま

す。追加のテンポラリー (temporary) は作成されず、転送呼び出しでの多重定義解決は、do_work 関数が直接呼び出された場合と同様に、同じ多重定義に解決されます。

```
struct s1 *do_work(const int&);               // #1
struct s2 *do_work(const double&);            // #2
struct s3 *do_work(int&&);                    // #3
struct s4 *do_work(double&&);                 // #4
template <typename T> auto wrapper(T && a)->
   decltype(do_work(std::forward<T>(*static_cast<typename std::remove_reference<T>
   ::type*>(0))))
{
   return do_work(std::forward<T>(a));
}
template <typename T> void tPtr(T *t);
int main()
{
   int x;
   double y;
   tPtr<s1>(wrapper(x));     // calls #1
   tPtr<s2>(wrapper(y));     // calls #2
   tPtr<s3>(wrapper(0));     // calls #3
   tPtr<s4>(wrapper(1.0));   // calls #4
}
```

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-qlanglvl

「*XL C/C++ ランゲージ・リファレンス*」の関連情報

参照の縮約 (C++11)

decltype(expression) 型指定子 (C++11)

後置戻り型 (C++11)

可視属性の使用 (IBM 拡張)

可視属性は、あるモジュールで定義されているエンティティーを他のモジュールで参照または使用することが可能かどうか、および可能な場合の参照方法と使用方法を示します。可視属性は、外部リンケージを持つエンティティーのみに適用され、他のエンティティーの可視性を高めることはできません。エンティティーの可視属性を指定することで、共用ライブラリーにとって必要なエンティティーのみをエクスポートできます。この機能を使用すると、以下の利点が得られます。

- 共用ライブラリーのサイズを縮小する。
- シンボル競合の可能性が減る。
- コンパイル・フェーズとリンク・フェーズでより多くの最適化を可能にする。
- 動的リンクの効率性を高める。

サポートされているエンティティーのタイプ

C++

コンパイラーは以下のエンティティーの可視属性をサポートしています。

- 関数

- variable
- structure/union/class
- enumeration
- template
- namespace

C++

C

コンパイラーは以下のエンティティーの可視属性をサポートしています。

- 関数
- variable

注: C 言語のデータ型は外部リンケージを持っていないため、C のデータ型の可視属性を指定することはできません。

C

「*XL C/C++ コンパイラー・リファレンス*」の関連情報


-qvisibility (IBM 拡張)


-qmkshrobj


#pragma GCC visibility push、#pragma GCC visibility pop (IBM 拡張)

「*XL C/C++ ランゲージ・リファレンス*」の関連情報


visibility 変数属性 (IBM 拡張)


visibility 関数属性 (IBM 拡張)


visibility 型属性 (C++ のみ) (IBM 拡張)


visibility 名前空間属性 (C++ のみ) (IBM 拡張)


内部リンケージ


外部リンケージ


リンケージなし

可視属性のタイプ

次の表では、各種の可視属性を説明しています。

表 19. 可視属性

属性	説明
デフォルト (default)	外部リンケージ・エンティティーがオブジェクト・ファイル内にデフォルト属性を持っていることを示します。これらのエンティティーは共用ライブラリーにエクスポートされ、これらの代わりに同じ名前の他のエンティティーを優先使用できます。
保護された (protected)	外部リンケージ・エンティティーがオブジェクト・ファイル内に保護された属性を持っていることを示します。これらのエンティティーは共用ライブラリーにエクスポートされますが、これらの代わりに同じ名前の他のエンティティーを優先使用することはできません。
非公開 (hidden)	外部リンケージ・エンティティーがオブジェクト・ファイル内に非公開の属性を持っていることを示します。これらのエンティティーは共用ライブラリーにエクスポートされませんが、これらのエンティティーのアドレスをポインターを介して間接参照できます。
内部 (internal)	外部リンケージ・エンティティーがオブジェクト・ファイル内に内部属性を持っていることを示します。これらのエンティティーは共用ライブラリーにエクスポートされず、これらのエンティティーのアドレスを共用ライブラリー内の他のモジュールが使用することはできません。
注: • このリリースでは、非公開の可視属性と内部可視属性は同じです。これらのいずれかの可視属性が指定されたエンティティーのアドレスは、ポインターを介して間接参照できます。	

例: デフォルト可視属性、保護された可視属性、
非公開可視属性、および内部可視属性の違い

```
//a.c
#include <stdio.h>
void __attribute__((visibility("default"))) func1(){
   printf("func1 in the shared library");
}
void __attribute__((visibility("protected"))) func2(){
   printf("func2 in the shared library");
}
void __attribute__((visibility("hidden"))) func3(){
   printf("func3 in the shared library");
}
void __attribute__((visibility("internal"))) func4(){
   printf("func4 in the shared library");
}

//a.h
extern void func1();
extern void func2();
extern void func3();
extern void func4();

//b.c
#include "a.h"
void temp(){
   func1();
   func2();
}
```

```
//b.h
extern void temp();

//main.c
#include "a.h"
#include "b.h"

void func1(){
   printf("func1 in b.c");
}
void func2(){
   printf("func2 in b.c");
}
void main(){
   temp();
   // func3(); // error
   // func4(); // error
}
```

次のコマンドを使用して、libtest.so という名前の共用ライブラリーを作成できます。

```
xlc -c -qpic a.c b.c
xlc -qmkshrobj -o libtest.so a.o b.o
```

その後、次のコマンドを使用して、実行時に libtest.so を動的にリンクできます。

```
xlc main.c -L. -ltest -brtl -bexpall -o main
./main
```

この例の出力は次のとおりです。

```
func1 in b.c
func2 in the shared library
```

関数 func1() の可視属性はデフォルトであるため、この関数の代わりに main.c 内の同じ名前の関数が優先使用されます。関数 func2() の可視属性は保護されているため、この関数の代わりに同じ名前の別の関数を優先使用することはできません。コンパイラーは、共用ライブラリー libtest.so 内で定義されている func2() を常に呼び出します。関数 func3() の可視属性は非公開であるため、この関数は共用ライブラリーにエクスポートされません。コンパイラーはリンク・エラーを発行して、func3() の定義が見つからないことを通知します。内部可視属性を持つ関数 func4() についても同じ問題が当てはまります。

可視属性のルール

可視属性の優先順位

可視属性の優先順位は、デフォルト (default) 可視属性 < 保護された (protected) 可視属性 < 非公開の (hidden) 可視属性 < 内部 (internal) 可視属性となっています。参考として、例 3 および例 9 を参照してください。

可視属性の決定に関するルール

C

エンティティーの可視属性は、以下のルールによって決定されます。

1. そのエンティティーが明示的に指定された可視属性を持っている場合、その指定された可視属性が適用されます。
2. そうでない場合、そのエンティティーを囲んでいるプラグマ・ディレクティブのペアがある場合は、それらのプラグマ・ディレクティブによって指定された可視属性が適用されます。
3. そうでない場合、**-qvisibility** オプションの設定が適用されます。

C

C++

エンティティーの可視属性は、以下のルールによって決定されます。

1. そのエンティティーが明示的に指定された可視属性を持っている場合、その指定された可視属性が適用されます。
2. そうでない場合、そのエンティティーがテンプレート・インスタンス化またはテンプレート特殊化である場合に、そのテンプレートが可視属性を持っている場合は、エンティティーの可視属性はテンプレートの可視属性から伝搬されます。例 1 を参照してください。
3. そうでない場合、そのエンティティーが以下のいずれかのコンテキストによって囲まれている場合は、そのエンティティーの可視属性は最も近いコンテキストの可視属性から伝搬されます。例 2 を参照してください。伝搬ルールについて詳しくは、112 ページの『伝搬ルール (C++ のみ)』 を参照してください。
   - structure/class
   - enumeration
   - namespace
   - プラグマ・ディレクティブ

   制約事項: プラグマ・ディレクティブは、クラス・メンバーおよびテンプレート特殊化の可視属性に影響を与えません。
4. そうでない場合、そのエンティティーの可視属性は、以下の可視属性設定によって決定されます。優先順位が最も高い可視属性が、そのエンティティーの実際の可視属性です。例 3 を参照してください。可視属性の優先順位については、『可視属性の優先順位』を参照してください。
   - **-qvisibility** オプションの設定
   - そのエンティティーの型の可視属性 (そのエンティティーが変数であり、その変数の型が可視属性を持っている場合)
   - そのエンティティーの戻りの型の可視属性 (そのエンティティーが関数であり、その関数の戻りの型が可視属性を持っている場合)
   - そのエンティティーのパラメーター型の可視属性 (そのエンティティーが関数であり、その関数のパラメーター型が可視属性を持っている場合)
   - そのエンティティーのテンプレート引数またはテンプレート・パラメーターの可視属性 (そのエンティティーがテンプレートであり、そのテンプレートの引数またはパラメーターが可視属性を持っている場合)

例 1

次の例では、テンプレート template<typename T, typename U> B{} は、保護された可視属性を持っています。この可視属性は、テンプレート特殊化 template<> class B<char, char>{} の可視属性、部分的特殊化 template<typename T> class B<T, float>{} の可視属性、およびすべてのタイプのテンプレート・インスタンス化の可視属性に伝搬されます。

```
class __attribute__((visibility("internal"))) A{} vis_v_a;   //internal

//protected
template<typename T, typename U>
class __attribute__((visibility("protected"))) B{
   public:
   void func(){}
};

//protected
template<>
class B<char, char>{
   public:
   void func(){}
};

//protected
template<typename T>
class B<T, float>{
   public:
   void func(){}
};

B<int, int> a;    //protected
B<A, int> b;      //protected
B<char, char> c;  //protected
B<int, float> d;  //protected
B<A, float> e;    //protected

int main(){
   a.func();
   b.func();
   c.func();
   d.func();
   e.func();
}
```

例 2

次の例では、関数 func() を囲んでいるコンテキストのうち最も近いものはクラス B であるため、func() の可視属性はクラス B の可視属性 (非公開) から伝搬されます。クラス A を囲んでいるコンテキストのうち最も近いものは、protected に設定されたプラグマ・ディレクティブであるため、クラス A の可視性は保護されます。

```
namespace __attribute__((visibility("internal"))) ns{
#pragma GCC visibility push(protected)
   class A{
      class __attribute__((visibility("hidden"))) B{
         int func(){};
      };
   };
#pragma GCC visibility pop
};
```

例 3

次の例では、**-qvisibility** オプションによって指定された可視属性は保護されています。vis_v_d 変数の型はクラス CD であり、このクラスの可視属性はデフォルトです。これら 2 つの属性より優先順位が高い可視属性は保護されているため、vis_v_d 変数の実際の可視属性は保護されています。同じルールに基づいて、vis_v_p、vis_v_h、および vis_v_i 変数の可視属性が決定されます。vis_f_fun1、vis_f_fun2、および vis_f_fun3 関数の可視属性は、これらの関数のパラメーター型の可視属性、戻り型の可視属性、および **-qvisibility** オプションの設定によって決定されます。vis_f_template1 および vis_f_template2 テンプレート関数の可視属性は、これらの関数のテンプレート引数の可視属性、テンプレート・パラメーターの可視属性、関数パラメーター型の可視属性、戻り型の可視属性、および **-qvisibility** オプションの設定によって決定されます。優先順位が最も高い可視属性が適用されます。

```
//The -qvisibility=protected option is specified
class __attribute__((visibility("default"))) CD {} vis_v_d;   //protected
class __attribute__((visibility("protected"))) CP {} vis_v_p; //protected
class __attribute__((visibility("hidden"))) CH {} vis_v_h;    //hidden
class __attribute__((visibility("internal"))) CI {} vis_v_i;  //internal

void vis_f_fun1(CH a, CP b, CD c, CI d) {} //internal
void vis_f_fun2(CD a) {} //protected
CH vis_f_fun3(CI a, CP b) {} //internal

template<class T, class U> T vis_f_template1(T t, U u){}
template<class T, int N> void vis_f_template2(T t, int i){}

int main(){
   vis_f_template1<CD, CH>(vis_v_d, vis_v_p); //hidden
   vis_f_template2<CD, 10)(vis_v_p, 10); // protected
}
```

C++

可視属性の使用に関するルールと制約事項

エンティティーの可視属性を指定する際は、以下のルールと制約事項を考慮してください。

- 可視属性は、外部リンケージを持つエンティティーに対してのみ指定できます。他のリンケージを持つエンティティーの可視属性を設定すると、コンパイラーは警告メッセージを発行して、指定された可視属性は無視されます。例 4 を参照してください。
- エンティティーの同じ宣言内や定義内で複数の異なる可視属性を指定することはできません。指定した場合、コンパイラーはエラー・メッセージを発行します。例 5 を参照してください。
- 単一のエンティティーが、それぞれ異なる可視属性が指定された複数の宣言を持っている場合、そのエンティティーの可視属性は、コンパイラーによって最初に処理される可視属性です。例 6 を参照してください。
- typedef ステートメントで可視属性を指定することはできません。 例 7 を参照してください。
- C++ 型 T が可視属性を持っている場合、型 T*、T&、および T&& は、型 T と同じ可視属性を持っています。 例 8 を参照してください。

- C++ あるクラスとそのエンクロージング・クラスが明示的に指定された可視性を持っていない場合に、クラスの可視属性の優先順位が、このクラスの非静的メンバー型の可視属性の優先順位、およびこのクラスの基底クラスの可視属性の優先順位より低い場合、コンパイラーは警告メッセージを発行します。例 9 を参照してください。可視属性の優先順位については、『可視属性の優先順位』を参照してください。 C++
- C++ 名前空間の可視属性は、同じ名前の名前空間には当てはまりません。例 10 を参照してください。 C++
- C++ グローバルの new 属性または delete 演算子の可視属性を指定した場合、コンパイラーは、その可視属性がデフォルトでない場合はその可視属性を無視するという警告メッセージを発行します。 例 11 を参照してください。 C++

例 4

この例では、コンパイラーは、変数 m、i、および j の可視属性を無視します。その理由は、m と i は内部リンケージを持っているからであり、j はリンケージを持っていないからです。

```
static int m __attribute__((visibility("protected")));
int n __attribute__((visibility("protected")));

int main(){
   int i __attribute__((visibility("protected")));
   static int j __attribute__((visibility("protected")));
}
```

例 5

この例では、コンパイラーはエラー・メッセージを発行して、変数 m の定義内で 2 つの異なる可視属性を同時に指定できないことを通知します。

```
//error
int m __attribute__((visibility("hidden"))) __attribute__((visibility("protected")));
```

例 6

この例では、コンパイラーによって最初に処理される fun() 関数の宣言は extern void fun() __attribute__((visibility("hidden"))) であるため、fun() の可視属性は非公開です。

```
extern void fun() __attribute__((visibility("hidden")));
extern void fun() __attribute__((visibility("protected")));

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

例 7

この例では、変数 vis_v_ti の可視属性はデフォルトであり、これは typedef ステートメント内の設定の影響を受けません。

```
//The -qvisibility=default option is specified.
typedef int __attribute__((visibility("protected"))) INT;
INT vis_v_ti = 1;
```

C++

例 8

この例では、クラス CP の可視属性は保護されているため、CP* および CP& の可視属性も保護されています。

```
class __attribute__((visibility("protected"))) CP {} vis_v_p;
class CP* vis_v_p_p = &vis_v_p;  //protected
class CP& vis_v_lr_p = vis_v_p;  //protected
```

例 9

この例では、コンパイラーはクラス Derived1 のデフォルト可視属性を受け付けます。この可視属性は、クラス Derived1 に対して明示的に指定されているからです。コンパイラーは、クラス Derived2 の保護された可視属性も受け付けます。この可視属性は、エンクロージング・クラス A の可視属性から伝搬されたものであるからです。クラス Derived3 は明示的に指定された可視属性を持っておらず、エンクロージング・クラスもないため、このクラスの可視属性はデフォルトです。コンパイラーは警告メッセージを発行します。その理由は、クラス Derived3 の可視属性の優先順位は、親クラス Base の可視属性の優先順位、および非静的メンバー関数 fun() の可視属性の優先順位より低いからです。

```
//The -qvisibility=default option is specified.
//base class
struct __attribute__((visibility("hidden"))) Base{
   int vis_f_fun(){
     return 0;
   }
};

//Ok
struct __attribute__((visibility("default"))) Derived1: public Base{
   int vis_f_fun(){
      return Base::vis_f_fun();
   };
}vis_v_d;

//Ok
struct __attribute__((visibility("protected"))) A{
   struct Derived2: public Base{
      int vis_f_fun(){
         __attribute__((visibility("protected")))
      };
   }
};

//Warning
struct Derived3: public Base{
  //Warning
  int fun() __attribute__((visibility("protected"))){};
};
```

例 10

この例では、名前空間 X の定義の可視属性は、名前空間 X の拡張には適用されません。

```
//The -qvisibility=default option is specified.
//namespace definition
namespace X __attribute__((visibility("protected"))){
  int a;  //protected
  int b;  //protected
}
//namespace extension
namespace X {
  int c;  //default
  int d;  //default
}
//equivalent to namespace X
namespace Y {
  int __attribute__((visibility("protected"))) a;  //protected
  int __attribute__((visibility("protected"))) b;  //protected
  int c;  //default
  int d;  //default
}
```

例 11

この例では、クラス A の外側で定義された new 演算子と delete 演算子はグローバル関数であるため、明示的に指定された非公開の可視属性は適用されません。クラス A 内で定義された new 演算子と delete 演算子はローカル関数であるため、これらの可視属性を指定できます。

```
#include <stddef.h>
//default
void* operator new(size_t) throw (std::bad_alloc) __attribute__((visibility("hidden")))
{
   return 0;
};
void operator delete(void*) throw () __attribute__((visibility("hidden"))){}

class A{
   public:
   //hidden
   void* operator new(size_t) throw (std::bad_alloc) __attribute__((visibility("hidden")))
   {
      return 0;
   };
   void operator delete(void*) throw () __attribute__((visibility("hidden"))){}
};
```

C++

伝搬ルール (C++ のみ)

可視属性は、エンティティー間で伝搬できます。次の表では、可視性伝搬のすべてのケースを示しています。

表 20. 可視属性の伝搬

伝搬元エンティティー	伝搬先エンティティー	例

namespace	伝搬元の名前空間で定義された名前付き名前空間	`namespace A __attribute__((visibility("hidden"))){`
		`  // Namespace B has the hidden visibility attribute,`
		`  // which is propagated from namespace A.`
		`  namespace B{}`
		`  // The unnamed namespace does not have a visibility`
		`  // attribute.`
		`  namespace{}`
		`}`

namespace	伝搬元の名前空間で定義されたクラス	`namespace A __attribute__((visibility("hidden"))){`
		`  // Class B has the hidden visibility attribute,`
		`  // which is propagated from namespace A.`
		`  class B;`
		`  // Object x has the hidden visibility attribute,`
		`  // which is propagated from namespace A.`
		`  class{} x;`
		`}`

namespace	伝搬元の名前空間で定義された関数	`namespace A __attribute__((visibility("hidden"))){`
		`  // Function fun() has the hidden visibility`
		`  // attribute, which is propagated from namespace A.`
		`  void fun(){};`
		`}`

namespace	伝搬元の名前空間で定義されたオブジェクト	`namespace A __attribute__((visibility("hidden"))){`
		`  // Variable m has the hidden visibility attribute,`
		`  // which is propagated from namespace A.`
		`  int m;`
		`}`

クラス	メンバー・クラス	`class __attribute__((visibility("hidden"))) A{`
		`  // Class B has the hidden visibility attribute,`
		`  // which is propagated from class A.`
		`  class B{};`
		`}`

クラス	メンバー関数または静的メンバー変数	`class __attribute__((visibility("hidden"))) A{`
		`  // Function fun() has the hidden visibility`
		`  // attribute, which is propagated from class A.`
		`  void fun(){};`
		`  // Static variable m has the hidden visibility`
		`  // attribute, which is propagated from class A.`
		`  static int m;`
		`}`

表 20. 可視属性の伝搬 (続き)

伝搬元エンティティー	伝搬先エンティティー	例

template	テンプレート・インスタンス化/テンプレート特殊化/テンプレート部分的特殊化	`template<typename T, typename U>`
		`class __attribute__((visibility("hidden"))) A{`
		`  public:`
		`  void fun(){};`
		`};`
		
		`// Template instantiation class A<int, char> has the`
		`// hidden visibility attribute, which is propagated`
		`// from template class A(T,U).`
		`class A<int, char>{`
		`  public:`
		`  void fun(){};`
		`};`
		
		`// Template specification`
		`// template<> class A<double, double> has the hidden`
		`// visibility attribute, which is propagated`
		`// from template class A(T,U).`
		`template<> class A<double, double>{`
		`  public:`
		`  void fun(){};`
		`};`
		
		`// Template partial specification`
		`// template<typename T> class A<T, char> has the`
		`// hidden visibility attribute, which is propagated`
		`// from template class A(T,U).`
		`template<typename T> class A<T, char>{`
		`  public:`
		`  void fun(){};`
		`};`

テンプレートの引数/パラメーター	テンプレート・インスタンス化/テンプレート特殊化/テンプレート部分的特殊化	`template<typename T> void fun1(){}`
		`template<typename T> void fun2(T){}`
		
		`class M __attribute__((visibility("hidden"))){} m;`
		
		`// Template instantiation fun1<M>() has the hidden`
		`// visibility attribute, which is propagated from`
		`// template argument M.`
		`fun1<M>();`
		
		`// Template instantiation fun2<M>(M) has the hidden`
		`// visibility attribute, which is propagated from`
		`// template parameter m.`
		`fun2(m);`
		
		`// Template specification fun1<M>() has the hidden`
		`// visibility attribute, which is propagated from`
		`// template argument M.`
		`template<> void fun1<M>();`

インライン関数	静的ローカル変数	`inline void __attribute__((visibility("hidden")))`
		`  fun(){`
		`  // Variable m has the hidden visibility attribute,`
		`  // which is propagated from inline function fun().`
		`  static int m = 4;`
		`}`

表 20. 可視属性の伝搬 (続き)

伝搬元エンティティー	伝搬先エンティティー	例

type	元の型のエンティティー	`class __attribute__((visibility("hidden"))) A {};`
		
		`// Object x has the hidden visibility attribute,`
		`// which is propagated from class A.`
		`class A x;`

関数の戻り値の型	関数	`class __attribute__((visibility("hidden"))) A{};`
		`// Function fun() has the hidden visibility attribute,`
		`// which is propagated from function return type A.`
		`A fun();`

関数パラメーターの型	関数	`class __attribute__((visibility("hidden"))) A{};`
		`// Function fun(class A) has the hidden visibility`
		`// attribute, which is propagated from function`
		`// parameter type A.`
		`void fun(class A);`

-qvisibility オプションを使用した可視属性の指定

-qvisibility オプションを使用して、プログラム内の外部リンケージ・エンティティーの可視属性をグローバルに設定できます。これらのエンティティーは、プラグマ・ディレクティブ、明示的に指定された属性、または伝搬ルールから可視属性を得ていない場合、**-qvisibility** オプションによって指定された可視属性を持っています。

プラグマ・プリプロセッサー・ディレクティブを使用した可視属性の指定

ソース・プログラム全体にわたって、`#pragma GCC visibility push` および `#pragma GCC visibility pop` というプリプロセッサー・ディレクティブのペアを使用することで、エンティティーの可視属性を選択的に設定できます。

コンパイラーは、ネストされた可視性プラグマ・プリプロセッサー・ディレクティブをサポートしています。エンティティーが、ネストされた `#pragma GCC visibility push` および `#pragma GCC visibility pop` ディレクティブの複数のペアに含まれている場合、それらのうち最も近いディレクティブ・ペアが有効になります。例 1 を参照してください。

ヘッダー・ファイルに対して可視性プラグマ・ディレクティブを指定してはいけません。指定した場合、プログラムは想定外の動作を示す可能性があります。 例 2 を参照してください。

C++ 可視性プラグマ・ディレクティブ `#pragma GCC visibility push` および `#pragma GCC visibility pop` は、名前空間スコープの宣言のみに影響を与えます。クラス・メンバーとテンプレート特殊化は影響を受けません。例 3 および例 4 を参照してください。 C++

例

例 1

この例では、関数と変数は、それぞれに最も近いプラグマ・プリプロセッサー・ディレクティブのペアによって指定された可視属性を持っています。

```
#pragma GCC visibility push(default)
namespace ns
{
         void vis_f_fun() {}       //default
#  pragma GCC visibility push(internal)
         int vis_v_i;              //internal
#    pragma GCC visibility push(protected)
         int vis_v_j;              //protected
#      pragma GCC visibility push(hidden)
         int vis_v_k;              //hidden
#      pragma GCC visibility pop
#    pragma GCC visibility pop
#  pragma GCC visibility pop
}
#pragma GCC visibility pop
```

例 2

この例では、コンパイラーはリンク・エラー・メッセージを発行して、printf() ライブラリー関数の定義が見つからないことを通知しています。

```
#pragma GCC visibility push(hidden)
#include <stdio.h>
#pragma GCC visibility pop

int main(){
   printf("hello world!");
   return 0;
}
```

C++

例
3

この例では、クラス・メンバー vis_v_i および vis_f_fun() の可視属性は非公開になっています。この可視属性は、クラスの可視属性から伝搬されていますが、プラグマ・ディレクティブの影響は受けません。

```
class __attribute__((visibility("hidden"))) A{
#pragma GCC visibility push(protected)
   public:
    static int vis_v_i;
    void vis_f_fun() {}
#pragma GCC visibility pop
} vis_v_a;
```

例 4

この例では、vis_f_fun() 関数の可視属性は非公開になっています。この可視属性は、テンプレート特殊化または部分的特殊化の可視属性から伝搬されていますが、プラグマ・ディレクティブの影響は受けません。

```
namespace ns{
   #pragma GCC visibility push(hidden)
   template <typename T, typename U> class TA{
      public:
      void vis_f_fun(){}
   };
```

```
#pragma GCC visibility pop

#pragma GCC visibility push(protected)
//The visibility attribute of the template specialization is hidden.
template <> class TA<char, char>{
   public:
   void vis_f_fun(){}
};
#pragma GCC visibility pop

#pragma GCC visibility push(default)
//The visibility attribute of the template partial specialization is hidden.
template <typename T> class TA<T, long>{
   public:
   void vis_f_fun(){}
};
#pragma GCC visibility pop
```

C++

第 11 章 ハイパフォーマンス・ライブラリーの使用

IBM XL C/C++ for Linux, V13.1 は、ハイパフォーマンス数学計算のライブラリー・セットを同梱して出荷されています。

- Mathematical Acceleration Subsystem (MASS) は調整された数学組み込み関数のライブラリー・セットで、標準システム数学ライブラリー関数を超える向上した性能を提供します。 MASS については 『Mathematical Acceleration Subsystem (MASS) ライブラリーの使用』で説明されています。
- Basic Linear Algebra Subprograms (BLAS) はルーチンのセットで、 PowerPC アーキテクチャー用に調整された、行列ベクトル乗算関数を提供しています。 BLAS 関数については 132 ページの『Basic Linear Algebra Subprograms (BLAS) の使用』で説明されています。

Mathematical Acceleration Subsystem (MASS) ライブラリーの使用

XL C/C++ は、ハイパフォーマンス数学計算の Mathematical Acceleration Subsystem (MASS) ライブラリー・セットを同梱して出荷されています。

MASS ライブラリーは、スカラー C/C++ 関数のライブラリー (『 120 ページの『スカラー・ライブラリーの使用』』を参照) 、特定のアーキテクチャー用に調整されたベクトル・ライブラリーのセット (『 122 ページの『ベクトル・ライブラリーの使用』』を参照)、および特定のアーキテクチャー用に調整された SIMD ライブラリーのセット (『 127 ページの『SIMD ライブラリーの使用』』を参照) から構成されます。 スカラーとベクトルの両方のライブラリーに含まれている関数は、ある最適化のレベルで自動的に呼び出されますが、ユーザーがプログラム内で明示的に呼び出すことも可能です。 MASS 関数とシステム・ライブラリー関数では、正確性と例外処理が異なる場合がありますのでご注意ください。

MASS 関数は、デフォルトの丸めモードで、浮動小数点例外トラッピングを設定して実行する必要があります。

プログラムを下記のいずれかのオプション・セットを使ってコンパイルするときは、

- **-qhot -qignerrno -qnostrict**
- **-qhot -qignerrno -qstrict=nolibrary**
- **-qhot -O3**
- **-O4**
- **-O5**

コンパイラーは、等価 MASS ベクトル関数 (関数 `vdnint`、`vdint`、`vcosisin`、`vscosisin`、`vqdrt`、`vsqdrt`、`vrqdrt`、`vsrqdrt`、`vpopcnt4`、`vpopcnt8`、`vexp2`、`vexp2m1`、`vsexp2`、`vsexp2m1`、`vlog2`、`vlog21p`、`vslog2`、および `vslog21p` を例外として) の呼び出しによって、自動的にシステム数学関数に対するベクトル化呼び出しを試みます。ベクトル化できない場合は、自動的に、等価 MASS スカラー関数の

© Copyright IBM Corp. 1996, 2014

呼び出しを試行します。自動ベクトル化または自動スカラー化の場合、コンパイラーは XLOPT ライブラリー libxlopt.a に含まれている MASS 関数のバージョンを使用します。

前述のオプション・セットに加えて、**-qipa** オプションが有効になっているときにコンパイラーがベクトル化できないと、MASS スカラー関数のインライン化を試行してから呼び出しを決定します。

131 ページの『MASS を使用するプログラムのコンパイルとリンク』では、MASS ライブラリーを使用するプログラムのコンパイル方法とリンク方法、ならびに MASS スカラー・ライブラリー関数を通常のシステム・ライブラリー関数と連携して選択的に使用する方法について説明します。

注: Linux では、32 ビット・オブジェクトと 64 ビット・オブジェクトを同じライブラリー内で組み合わせることができないため、以下のように 2 つのバージョンのスカラー・ライブラリー、ベクトル・ライブラリー、および SIMD ライブラリーのそれぞれ、ベクトル・ライブラリー、および SIMD ライブラリーがコンパイラーと一緒に出荷されます。

アプリケーション・タイプ	スカラー・ライブラリー	ベクトル・ライブラリー	SIMD ライブラリー
32 ビット・アプリケーション	libmass.a	libmassvp*.a	libmass_simd*.a
64 ビット・アプリケーション	libmass_64.a	libmassvp*_64.a	libmass_simd_64.a

関連外部情報

Mathematical Acceleration Subsystem の Web サイト、http://www.ibm.com/software/awdtools/mass/ からアクセス可能

スカラー・ライブラリーの使用

MASS スカラー・ライブラリー libmass.a (32 ビット) および libmass_64.a (64 ビット) には、対応する標準システム・ライブラリー関数よりもパフォーマンスが向上した、高速化された一連の頻繁に使用される数学組み込み関数が含まれています。 MASS スカラー関数は、libmass.a または libmass_64.a に明示的にリンクするときに使用されます。

明示的に MASS スカラー関数を呼び出したい場合は、以下のステップを実行します。

1. math.h をソース・ファイルに組み込むことで、関数 (anint、cosisin、dnint、sincos および rsqrt を除く) のプロトタイプを提供する。
2. mass.h をソース・ファイルに組み込むことで、関数 (anint、cosisin、dnint、sincos および rsqrt を除く) のプロトタイプを提供する。
3. MASS スカラー・ライブラリー libmass.a (または 64 ビット・バージョンの libmass_64.a) をアプリケーションとリンクさせる。詳細な説明については、131 ページの『MASS を使用するプログラムのコンパイルとリンク』を参照してください。

倍精度の結果を 2 つ戻す sincos 以外の MASS スカラー関数は、倍精度のパラメーターを受け入れて倍精度の結果を戻すか、単精度のパラメーターを受け入れて単精度の結果を戻します。これらの MASS スカラー関数の要約を表 21 に示します。

表 21. MASS スカラー関数

倍精度関数	単精度関数	説明	倍精度関数プロトタイプ	単精度関数プロトタイプ
acos	acosf	x のアークコサインを返す	double acos (double x);	float acosf (float x);
acosh	acoshf	x の双曲線アークコサインを返す	double acosh (double x);	float acoshf (float x);
	anint	x の丸めた整数値を返す		float anint (float x);
asin	asinf	x のアークサインを返す	double asin (double x);	float asinf (float x);
asinh	asinhf	x の双曲線アークサインを返す	double asinh (double x);	float asinhf (float x);
atan2	atan2f	x/y のアークタンジェントを戻す	double atan2 (double x, double y);	float atan2f (float x, float y);
atan	atanf	x のアークタンジェントを返す	double atan (double x);	float atanf (float x);
atanh	atanhf	x の双曲線アークタンジェントを返す	double atanh (double x);	float atanhf (float x);
cbrt	cbrtf	x の立方根を返す	double cbrt (double x);	float cbrtf (float x);
copysign	copysignf	符号 y を持つ x を戻す	double copysign (double x,double y);	float copysignf (float x);
cos	cosf	x のコサインを返す	double cos (double x);	float cosf (float x);
cosh	coshf	x の双曲線コサインを返す	double cosh (double x);	float coshf (float x);
cosisin		実数部が x のコサインで虚数部が x のサインである複素数を戻す	double_Complex cosisin (double);	
dnint		x (倍数) の最も近い整数を戻す	double dnint (double x);	
erf	erff	x の誤差関数を返す	double erf (double x);	float erff (float x);
erfc	erfcf	x の相補誤差関数を返す	double erfc (double x);	float erfcf (float x);
exp	expf	x の指数関数を返す	double exp (double x);	float expf (float x);
expm1	expm1f	(x の指数関数) - 1 を戻す	double expm1 (double x);	float expm1f (float x);
hypot	hypotf	$(x^2 + y^2)$ の平方根を戻す	double hypot (double x, double y);	float hypotf (float x, float y);
lgamma	lgammaf	x のガンマ関数の絶対値の自然対数を返す	double lgamma (double x);	float lgammaf (float x);
log	logf	x の自然対数を返す	double log (double x);	float logf (float x);
log10	log10f	x の常用対数を返す	double log10 (double x);	float log10f (float x);

表 21. MASS スカラー関数 (続き)

倍精度関数	単精度関数	説明	倍精度関数プロトタイプ	単精度関数プロトタイプ
log1p	log1pf	(x + 1) の自然対数を戻す	double log1p (double x);	float log1pf (float x);
pow	powf	x の y 乗を戻す	double pow (double x, double y);	float powf (float x, float y);
rsqrt		x の平方根の逆数を返す	double rsqrt (double x);	
sin	sinf	x のサインを返す	double sin (double x);	float sinf (float x);
sincos		*s を x のサインに、*c を x のコサインに設定する	void sincos (double x, double* s, double* c);	
sinh	sinhf	x の双曲線サインを返す	double sinh (double x);	float sinhf (float x);
sqrt		x の平方根を返す	double sqrt (double x);	
tan	tanf	x のタンジェントを返す	double tan (double x);	float tanf (float x);
tanh	tanhf	x の双曲線タンジェントを返す	double tanh (double x);	float tanhf (float x);

注:

- 三角関数 (sin、cos、tan) は、大きな引数 (絶対値が 2^{50} パイより大きい) の場合は NaN (数値ではない) を戻します。
- 場合によっては、MASS 関数は libm.a ライブラリー内の関数ほど精度が高くないため、異なる方法で境界ケースを処理することがあります (例: sqrt(Inf))。
- libm.a との精度比較については、Mathematical Acceleration Subsystem Web サイト を参照してください。

関連外部情報

Mathematical Acceleration Subsystem の Web サイト、http://www.ibm.com/software/awdtools/mass/ からアクセス可能

ベクトル・ライブラリーの使用

いずれかの MASS ベクトル関数を明示的に呼び出したい場合、massv.h をソース・ファイルに含めて、アプリケーションを適切なベクトル・ライブラリーにリンクすることで、それを行うことができます。 (リンクについては、 131 ページの『MASS を使用するプログラムのコンパイルとリンク』を参照してください。)

libmassvp5.a
POWER5 アーキテクチャー用に調整された関数を含みます。

libmassvp6.a
POWER6® アーキテクチャー用に調整された関数を含みます。

libmassvp7.a
POWER7 アーキテクチャー用に調整された関数を含みます。

libmassvp8.a

POWER8 アーキテクチャー用に調整された関数を含みます。

ベクトル・ライブラリーに含まれている、単精度と倍精度の浮動小数点関数については、 124 ページの表 22 にまとめられています。ベクトル・ライブラリーに含まれている整数関数については、 126 ページの表 23 にまとまられています。 C および C++ アプリケーションでは、スカラー引数を使用する場合でも、サポートされているのは参照による呼び出しのみです。

いくつかの関数を除いて (下記を参照)、ベクトル・ライブラリー内のすべての浮動小数点関数は次の 3 つのパラメーターを受け入れます。

- 倍精度 (倍精度関数用) または単精度 (単精度関数用) のベクトル出力パラメーター。
- 倍精度 (倍精度関数用) または単精度 (単精度関数用) のベクトル入力パラメーター。
- 整数ベクトル長パラメーター。

関数の形式は次のとおりです。

function_name (*y*,*x*,*n*)

ここで、*y* はターゲット・ベクトル、*x* はソース・ベクトル、*n* はベクトルの長さです。パラメーター *y* および *x* は、接頭部 v が付いた関数では倍精度、接頭部 vs が付いた関数では単精度と見なされます。以下にコード例を示します。

```
#include <massv.h>

double x[500], y[500];
int n;
n = 500;
...
vexp (y, x, &n);
```

エレメントが exp(x[i]) (ここで i=0,...,499) である長さ 500 のベクトル *y* を出力します。

関数 vdiv、vsincos、vpow、および vatan2 (およびそれらの単精度バージョン、vsdiv、vssincos、vspow、および vsatan2) は、4 つの引数をとります。関数 vdiv、vpow、および vatan2 は、引数 (*z*,*x*,*y*,*n*) をとります。関数 vdiv は、ベクトル *z* を出力します。このエレメントは x[i]/y[i] (i=0,..,*n–1) です。関数 vpow はベクトル *z* を出力します。このエレメントは $x[i]^{y[i]}$ (i=0,..,*n–1) です。関数 vatan2 はベクトル *z* を出力します。このエレメントは atan(x[i]/y[i]) (i=0,..,*n–1) です。関数 vsincos は引数 (*y*,*z*,*x*,*n*) をとり、2 つのベクトル *y* および *z* を出力します。このエレメントはそれぞれ sin(x[i]) および cos(x[i]) です。

vcosisin(y,x,n) および vscosisin(y,x,n) では、*x* は *n* 個のエレメントのベクトルであり、関数は、(cos(x[i]),sin(x[i])) 形式の *n* 個の __Complex エレメントのベクトル *y* を出力します。

表 22. MASS 浮動小数点ベクトル関数

倍精度関数	単精度関数	説明	倍精度関数 プロトタイプ	単精度関数 プロトタイプ
vacos	vsacos	i=0,..,*n-1 の場合に、y[i] を x[i] のアークコサインに設定する	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);
vacosh	vsacosh	i=0,..,*n-1 の場合に、y[i] を x[i] の双曲線アークコサインに設定する	void vacosh (double y[], double x[], int *n);	void vsacosh (float y[], float x[], int *n);
vasin	vsasin	i=0,..,*n-1 の場合に、y[i] を x[i] のアークサインに設定する	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vasinh	vsasinh	i=0,..,*n-1 の場合に、y[i] を x[i] の双曲線アークサインに設定する	void vasinh (double y[], double x[], int *n);	void vsasinh (float y[], float x[], int *n);
vatan2	vsatan2	i=0,..,*n-1 の場合に、z[i] を x[i]/y[i] のアークタンジェントに設定する	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vatanh	vsatanh	i=0,..,*n-1 の場合に、y[i] を x[i] の双曲線アークタンジェントに設定する	void vatanh (double y[], double x[], int *n);	void vsatanh (float y[], float x[], int *n);
vcbrt	vscbrt	i=0,..,*n-1 の場合に、y[i] を x[i] の立方根に設定する	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	i=0,..,*n-1 の場合に、y[i] を x[i] のコサインに設定する	void vcos (double y[], double x[], int *n);	void vscos (float y[], float x[], int *n);
vcosh	vscosh	i=0,..,*n-1 の場合に、y[i] を x[i] の双曲線コサインに設定する	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin	vscosisin	i=0,..,*n-1 の場合に、y[i] の実数部を x[i] のコサインに、そして y[i] の虚数部を x[i] のサインに設定する	void vcosisin (double _Complex y[], double x[], int *n);	void vscosisin (float _Complex y[], float x[], int *n);
vdint		i=0,..,*n-1 の場合に、y[i] を x[i] の切り捨て整数部に設定する	void vdint (double y[], double x[], int *n);	
vdiv	vsdiv	i=0,..,*n-1 の場合に、z[i] を x[i]/y[i] に設定します。	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdnint		i=0,..,*n-1 の場合に、y[i] を x[i] に最も近い整数に設定する	void vdnint (double y[], double x[], int *n);	
verf	vserf	i=0,..,*n-1 の場合に、y[i] を x[i] の誤差関数に設定する	void verf (double y[], double x[], int *n)	void vserf (float y[], float x[], int *n)

表 22. MASS 浮動小数点ベクトル関数 (続き)

倍精度関数	単精度関数	説明	倍精度関数 プロトタイプ	単精度関数 プロトタイプ
verfc	vserfc	i=0,..,*n-1 の場合に、y[i] を x[i] の相補誤差関数に設定する	void verfc (double y[], double x[], int *n)	void vserfc (float y[], float x[], int *n)
vexp	vsexp	i=0,..,*n-1 の場合に、y[i] を x[i] の指数関数に設定する	void vexp (double y[], double x[], int *n);	void vsexp (float y[], float x[], int *n);
vexp2	vsexp2	i=1,..,*n-1 の場合に、y[i] を x[i] の 2 のべき乗に設定する	void vexp2 (double y[], double x[], int *n);	void vsexp2 (float y[], float x[], int *n);
vexpm1	vsexpm1	i=0,..,*n-1 の場合に、y[i] を (x[i] の指数関数) - 1 に設定する	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);
vexp2m1	vsexp2m1	i=1,..,*n-1 の場合に、y[i] を (x[i] の 2 のべき乗) - 1 に設定する	void vexp2m1 (double y[], double x[], int *n);	void vsexp2m1 (float y[], float x[], int *n);
vhypot	vshypot	i=0,..,*n-1 の場合に、z[i] を x[i] の二乗と y[i] の二乗の合計の平方根に設定する	void vhypot (double z[], double x[], double y[], int *n)	void vshypot (float z[], float x[], float y[], int *n)
vlog	vslog	i=0,..,*n-1 の場合に、y[i] を x[i] の自然対数に設定する	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog2	vslog2	i=0,..,*n-1 の場合に、y[i] を x[i] の対数 (底 2) に設定する	void vlog2 (double y[], double x[], int *n);	void vslog2(float y[], float x[], int *n);
vlog10	vslog10	i=0,..,*n-1 の場合に、y[i] を x[i] の対数 (底 10) に設定する	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	i=0,..,*n-1 の場合に、y[i] を (x[i]+1) の自然対数に設定する	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vlog21p	vslog21p	i=1,..,*n-1 の場合に、y[i] を (x[i]+1) の対数 (底 2) に設定する	void vlog21p (double y[], double x[], int *n);	void vslog21p (float y[], float x[], int *n);
vpow	vspow	i=0,..,*n-1 の場合に、z[i] を x[i] の y[i] 乗に設定する	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vqdrt	vsqdrt	i=0,..,*n-1 の場合に、y[i] を x[i] の 4 乗根に設定する	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	i=0,..,*n-1 の場合に、y[i] を x[i] の立方根の逆数に設定する	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], int *n);
vrec	vsrec	i=0,..,*n-1 の場合に、y[i] を x[i] の逆数に設定する	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);

表 22. MASS 浮動小数点ベクトル関数 (続き)

倍精度関数	単精度関数	説明	倍精度関数 プロトタイプ	単精度関数 プロトタイプ
vrqdrt	vsrqdrt	i=0,..,*n-1 の場合に、y[i] を x[i] の 4 乗根の逆数に設定する	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);
vrsqrt	vsrsqrt	i=0,..,*n-1 の場合に、y[i] を x[i] の平方根の逆数に設定する	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsin	vssin	i=0,..,*n-1 の場合に、y[i] を x[i] のサインに設定する	void vsin (double y[], double x[], int *n);	void vssin (float y[], float x[], int *n);
vsincos	vssincos	i=0,..,*n-1 の場合に、y[i] を x[i] のサインに、そして z[i] を x[i] のコサインに設定する	void vsincos (double y[], double z[], double x[], int *n);	void vssincos (float y[], float z[], float x[], int *n);
vsinh	vssinh	i=0,..,*n-1 の場合に、y[i] を x[i] の双曲線サインに設定する	void vsinh (double y[], double x[], int *n);	void vssinh (float y[], float x[], int *n);
vsqrt	vssqrt	i=0,..,*n-1 の場合に、y[i] を x[i] の平方根に設定する	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);
vtan	vstan	i=0,..,*n-1 の場合に、y[i] を x[i] のタンジェントに設定する	void vtan (double y[], double x[], int *n);	void vstan (float y[], float x[], int *n);
vtanh	vstanh	i=0,..,*n-1 の場合に、y[i] を x[i] の双曲線タンジェントに設定する	void vtanh (double y[], double x[], int *n);	void vstanh (float y[], float x[], int *n);

整数関数の形式は *function_name* (*x*[], **n*) です。ここで、x[] は 4 バイト (vpopcnt4 用) または 8 バイト (vpopcnt8 用) の数値オブジェクト (整数または浮動小数点) のベクトル、*n はベクトルの長さです。

表 23. MASS 整数ベクトル・ライブラリー関数

関数	説明	プロトタイプ
vpopcnt4	i=0,..,*n-1 の場合に、x[i] のバイナリー表現連結内の 1 ビットの総数を戻します。ここで、x は 32 ビット・オブジェクトのベクトルです。	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	i=0,..,*n-1 の場合に、x[i] のバイナリー表現連結内の 1 ビットの総数を戻します。ここで、x は 64 ビット・オブジェクトのベクトルです。	unsigned int vpopcnt8 (void *x, int *n)

入力ベクトルおよび出力ベクトルのオーバーラップ

大部分のアプリケーションでは、MASS ベクトル関数は、相互に素な入力ベクトルと出力ベクトルを指定して呼び出されます。すなわち、2 つのベクトルは、メモリ

一内でオーバーラップしません。別の一般的な使用シナリオでは、入力パラメーターと出力パラメーターの両方に対し、同じベクトルを指定して呼び出されます(例えば、vsin (y, y, &n))。他の種類のオーバーラップ (入力および出力ベクトルが相互に素でも同一でもない) は、予期しない結果を生じることがあるため、使用しないようにしてください。

- 1 つの入力ベクトルと 1 つの出力ベクトル (例えば、vsin (y, x, &n)) を受け取るベクトル関数の呼び出しは以下のことに注意します。

  ベクトル x[0:n-1] と y[0:n-1] は、相互に素または同一でなければなりません。そうでないと、予期しない結果となる場合があります。
- 2 つの入力ベクトル (例えば、vatan2 (y, x1, x2, &n)) を受け取るベクトル関数の呼び出しには以下に注意します。

  両方のベクトルのペア y、x1 および y、x2 に対して直前の制限が適用されます。つまり、y[0:n-1] および x1[0:n-1] は相互に素または同一でなければなりません。また、y[0:n-1] および x2[0:n-1] も相互に素または同一でなければなりません。
- 2 つの出力ベクトル (例えば、vsincos (y1, y2, x, &n)) を受け取るベクトル関数の呼び出しには以下に注意します。

  両方のベクトルのペア y1、x および y2、x に対して上記の制限が適用されます。つまり、y1[0:n-1] および x[0:n-1] は相互に素または同一でなければなりません。また、y2[0:n-1] および x[0:n-1] も相互に素または同一でなければなりません。また、ベクトル y1[0:n-1] と y2[0:n-1] は相互に素でなければなりません。

入力ベクトルおよび出力ベクトルの位置合わせ

POWER7 および POWER8 のベクトル・ライブラリーから最高のパフォーマンスを得るには、入力ベクトルと出力ベクトルを 16 バイト境界上で位置合せします。

MASS ベクトル関数の整合性

MASS ベクトル・ライブラリーのすべての関数は、ベクトルの位置、ベクトル長に関係なく、一定の入力値では常時同じ結果になるという意味では一貫性があります。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

-D

関連外部情報

Mathematical Acceleration Subsystem の Web サイト、http://www.ibm.com/software/awdtools/mass/ からアクセス可能

SIMD ライブラリーの使用

MASS SIMD ライブラリー

libmass_simdp7.a、libmass_simdp7_64.a、libmass_simdp8.a、または libmass_simdp8_64.a には、頻繁に使用される数学組み込み関数のセットが含まれて

います。このセットにより、対応する標準システム・ライブラリー関数よりもパフォーマンスが向上します。 MASS SIMD 関数を使用する場合、以下のようにして使用できます。

1. ソース・ファイルに mass_simd.h をインクルードすることにより、各関数のプロトタイプを指定します。
2. アプリケーションを MASS SIMD ライブラリー libmass_simdp7.a、libmass_simdp7_64.a、libmass_simdp8.a、または libmass_simdp8_64.a とリンクします。詳細な説明については、 131 ページの『MASS を使用するプログラムのコンパイルとリンク』を参照してください。

単精度/倍精度の MASS SIMD 関数は 単精度/倍精度の引数を受け入れて、単精度/倍精度の結果を戻します。これらの MASS スカラー関数の要約を 表 24 に示します。

表 24. MASS SIMD 関数

倍精度関数	単精度関数	説明	倍精度関数 プロトタイプ	単精度関数 プロトタイプ
acosd2	acosf4	vx の各エレメントのアークコサインを計算する	vector double acosd2 (vector double vx);	vector float acosf4 (vector float vx);
acoshd2	acoshf4	vx の各エレメントのアーク双曲線コサインを計算する	vector double acoshd2 (vector double vx);	vector float acoshf4 (vector float vx);
asind2	asinf4	vx の各エレメントのアークサインを計算する	vector double asind2 (vector double vx);	vector float asinf4 (vector float vx);
asinhd2	asinhf4	vx の各エレメントのアーク双曲線サインを計算する	vector double asinhd2 (vector double vx);	vector float asinhf4 (vector float vx);
atand2	atanf4	vx の各エレメントのアークタンジェントを計算する	vector double atand2 (vector double vx);	vector float atanf4 (vector float vx);
atan2d2	atan2f4	vx/vy の各エレメントのアークタンジェントを計算する	vector double atan2d2 (vector double vx, vector double vy);	vector float atan2f4 (vector float vx, vector float vy);
atanhd2	atanhf4	vx の各エレメントのアーク双曲線タンジェントを計算する	vector double atanhd2 (vector double vx);	vector float atanhf4 (vector float vx);
cbrtd2	cbrtf4	vx の各エレメントの立方根を計算する	vector double cbrtd2 (vector double vx);	vector float cbrtf4 (vector float vx);
cosd2	cosf4	vx の各エレメントのコサインを計算する	vector double cosd2 (vector double vx);	vector float cosf4 (vector float vx);
coshd2	coshf4	vx の各エレメントの双曲線コサインを計算する	vector double coshd2 (vector double vx);	vector float coshf4 (vector float vx);

表 24. MASS SIMD 関数 (続き)

倍精度関数	単精度関数	説明	倍精度関数 プロトタイプ	単精度関数 プロトタイプ
cosisind2	cosisinf4	x の各エレメントのコサインおよびサインを計算し、次のように結果を y および z に保管する cosisind2 (x,y,z) は、y と z を x={x1,x2} である {cos(x1), sin(x1)} および {cos(x2), sin(x2)} に設定する cosisinf4 (x,y,z) は、y と z を x={x1,x2,x3,x4} である {cos(x1), sin(x1), cos(x2), sin(x2)} および {cos(x3), sin(x3), cos(x4), sin(x4)} に設定する	void cosisind2 (vector double x, vector double *y, vector double *z)	void cosisinf4 (vector float x, vector float *y, vector float *z)
divd2	divf4	商 vx/vy を計算する	vector double divd2 (vector double vx, vector double vy);	vector float divf4 (vector float vx, vector float vy);
erfcd2	erfcf4	vx の各エレメントの補完的な誤差関数を計算する	vector double erfcd2(vector double vx);	vector float erfcf4(vector float vx);
erfd2	erff4	vx の各エレメントの誤差関数を計算する	vector double erfd2(vector double vx);	vector float erff4(vector float vx);
expd2	expf4	vx の各エレメントの指数関数を計算する	vector double expd2 (vector double vx);	vector float expf4 (vector float vx);
exp2d2	exp2f4	vx の各エレメントの 2 のべき乗を計算する	vector double exp2d2 (vector double vx);	vector float exp2f4 (vector float vx);
expm1d2	expm1f4	(vx の各エレメントの指数関数) - 1 を計算する	vector double expm1d2 (vector double vx);	vector float expm1f4 (vector float vx);
exp2m1d2	exp2m1f4	(vx の各エレメントの 2 のべき乗) - 1 を計算する	vector double exp2m1d2 (vector double vx);	vector float exp2m1f4 (vector float vx);
hypotd2	hypotf4	vx の各エレメントおよび対応する vy のエレメントについて、sqrt(x*x+y*y) を計算する	vector double hypotd2 (vector double vx, vector double vy);	vector float hypotf4 (vector float vx, vector float vy);
lgammad2	lgammaf4	vx の各エレメントのガンマ関数の絶対値の自然対数を計算する	vector double lgammad2(vector double vx);	vector float lgammaf4(vector float vx);

表 24. MASS SIMD 関数 (続き)

倍精度関数	単精度関数	説明	倍精度関数プロトタイプ	単精度関数プロトタイプ
logd2	logf4	vx の各エレメントの自然対数を計算する	vector double logd2 (vector double vx);	vector float logf4 (vector float vx);
log2d2	log2f4	vx の各エレメントの対数 (底 2) を計算する	vector double log2d2 (vector double vx);	vector float log2f4 (vector float vx);
log10d2	log10f4	vx の各エレメントの対数 (底 10) を計算する	vector double log10d2 (vector double vx);	vector float log10f4 (vector float vx);
log1pd2	log1pf4	(vx +1) の各エレメントの自然対数を計算する	vector double log1pd2(vector double vx);	vector float log1pf4(vector float vx);
log21pd2	log21pf4	(vx +1) の各エレメントの対数 (底 2) を計算する	vector double log21pd2 (vector double vx);	vector float log21pf4 (vector float vx);
powd2	powf4	vx の各エレメントを、対応する vy のエレメントで累乗した数を計算する	vector double powd2 (vector double vx, vector double vy);	vector float powf4 (vector float vx, vector float vy);
qdrtd2	qdrtf4	vx の各エレメントの 4 乗根を計算する	vector double qdrtd2 (vector double vx);	vector float qdrtf4 (vector float vx);
rcbrtd2	rcbrtf4	vx の各エレメントの立方根の逆数を計算する	vector double rcbrtd2 (vector double vx);	vector float rcbrtf4 (vector float vx);
recipd2	recipf4	vx の各エレメントの逆数を計算する	vector double recipd2 (vector double vx);	vector float recipf4 (vector float vx);
rqdrtd2	rqdrtf4	vx の各エレメントの 4 乗根の逆数を計算する	vector double rqdrtd2 (vector double vx);	vector float rqdrtf4 (vector float vx);
rsqrtd2	rsqrtf4	vx の各エレメントの平方根の逆数を計算する	vector double rsqrtd2 (vector double vx);	vector float rsqrtf4 (vector float vx);
sincosd2	sincosf4	vx の各エレメントのサインおよびコサインを計算する	void sincosd2(vector double vx, vector double *vs, vector double *vc);	void sincosf4(vector float vx, vector float *vs, vector float *vc);
sind2	sinf4	vx の各エレメントのサインを計算する	vector double sind2 (vector double vx);	vector float sinf4 (vector float vx);
sinhd2	sinhf4	vx の各エレメントの双曲線サインを計算する	vector double sinhd2 (vector double vx);	vector float sinhf4 (vector float vx);
sqrtd2	sqrtf4	vx の各エレメントの平方根を計算する	vector double sqrtd2 (vector double vx);	vector float sqrtf4 (vector float vx);
tand2	tanf4	vx の各エレメントのタンジェントを計算する	vector double tand2 (vector double vx);	vector float tanf4 (vector float vx);
tanhd2	tanhf4	vx の各エレメントの双曲線タンジェントを計算する	vector double tanhd2 (vector double vx);	vector float tanhf4 (vector float vx);

MASS を使用するプログラムのコンパイルとリンク

スカラー MASS ライブラリー、SIMD MASS ライブラリー、またはベクトル MASS ライブラリー内の関数を呼び出すアプリケーションをコンパイルするには、必要な 1 つ以上の次のライブラリー名を **-l** リンカー・オプションで指定します。

- 32 ビット: **libmass**、**mass_simdp7** または **mass_simdp8**、および/または次のいずれか:**massvp4**、**massvp5**、**massvp6**、**massvp7**、**massvp8**
- 64 ビット: **libmass_64**、**mass_simdp7_64** または **mass_simdp8_64**、および/または次のいずれか: **massvp4_64**、**massvp5_64**、**massvp6_64**、**massvp7_64**、**massvp8_64**

例えば、デフォルト・ディレクトリーに MASS ライブラリーがインストールされている場合は、以下のいずれかを指定できます。

スカラー・ライブラリー libmass.a およびベクトル・ライブラリー libmassvp8.a とのリンク (32 ビット・コード)

```
xlc -qarch=pwr8 progc.c -o progc -lmass -lmassvp8
```

SIMD ライブラリー libmass_simdp8.a とのリンク (32 ビット・コード)

```
xlc -qarch=pwr8 progc.c -o progc -lmass_simdp8
```

スカラー・ライブラリー libmass_64 およびベクトル・ライブラリー libmassvp8_64.a とのリンク (64 ビット・コード)

```
xlc -qarch=pwr8 progc.c -o progc -lmass_64 -lmassvp8_64 -q64
```

SIMD ライブラリー libmass_simdp8_64.a とのリンク (64 ビット・コード)

```
xlc -qarch=pwr8 progc.c -o progc -lmass_simdp8_64 -q64
```

数学システム・ライブラリーでの libmass.a の使用

libmass.a または libmass_64.a スカラー・ライブラリーを一部の関数用、および標準数学ライブラリー libm.a をその他の関数用に使用する場合、次の手順に従ってユーザー・プログラムをコンパイルしてリンクします。

1. **ar** コマンドを使用して、libmass.a または libmass_64.a から必要な関数のオブジェクト・ファイルを抽出します。大部分の関数において、オブジェクト・ファイル名は、関数名に .s32.o (32 ビット・モードの場合) または .s64.o (64 ビット・モードの場合) が続いたものです。[1]例えば、32 ビット・モードの tan 関数のオブジェクト・ファイルを抽出するコマンドは次のようになります。

   ```
   ar -x tan.s32.o libmass.a
   ```

2. 抽出したオブジェクト・ファイルを別のライブラリーにアーカイブする場合は以下になります。

   ```
   ar -qv libfasttan.a tan.s32.o
   ranlib libfasttan.a
   ```

3. **-lmass** の代わりに **-lfasttan** を指定して、**xlc** を使用して、最終的な実行可能ファイルを作成します。

   xlc sample.c -o sample -L*dir_containing_libfasttan* -lfasttan

   これは、tan 関数のみを MASS (現在は libfasttan.a 内にある) にリンクするもので、数学関数の残りは標準システム・ライブラリーからになります。

例外:

1. sin 関数と cos 関数は、両方ともオブジェクト・ファイル sincos.s32.o および sincos.s64.o に含まれています。cosisin 関数と sincos 関数は、両方ともオブジェクト・ファイル cosisin.s32.o および cosisin.s64.o に含まれています。
2. XL C/C++ pow 関数は、オブジェクト・ファイル dxy.s32.o および dxy.s64.o に含まれています。

注: cos 関数と sin 関数は、一方がエクスポートされると両方ともエクスポートされます。 cosisin 関数と sincos 関数は、一方がエクスポートされると両方ともエクスポートされます。

Basic Linear Algebra Subprograms (BLAS) の使用

4 つの Basic Linear Algebra Subprograms (BLAS) 関数は libxlopt ライブラリー内の XL C/C++ コンパイラーにあります。関数は以下から成り立っています。

- sgemv (単精度) および dgemv (倍精度) があり、これらは一般的な行列、またはその転置 (transpose) の行列ベクトルの積を計算します。
- sgemm (単精度) および dgemm (倍精度) があり、これらは一般的な行列、またはその転置 (transpose) に対する結合行列乗算、および加算を行います。

BLAS ルーチンは Fortran で書かれているため、すべてのパラメーターは参照によって渡され、すべての配列は列優先順位 (column-major order) で保管されます。

注: いくつかのエラー処理コードが libxlopt の BLAS 関数から取り外されて、これらの関数の呼び出しに対するエラー・メッセージは出されません。

『BLAS 関数構文』 は XL C/C++ BLAS 関数のプロトタイプとパラメーターについて説明しています。これらの関数に対するインターフェースは、IBM の Engineering and Scientific Subroutine Library (ESSL) から出荷された同等の BLAS 関数のものと同じです。追加情報とこれらの関数の使用法については、「*Engineering and Scientific Subroutine Library Guide and Reference*」を参照してください。これらは Web ページの Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL から利用可能です。

サード・パーティーの BLAS ライブラリーも使用中の場合は、135 ページの『libxlopt ライブラリーへのリンク』 に XL C/C++ libxlopt ライブラリーへのリンク方法が説明されているので参考になります。

BLAS 関数構文

sgemv 関数および dgemv 関数のプロトタイプを以下に示します。

```
void sgemv(const char *trans, int *m, int *n, float *alpha,
     void *a, int *lda, void *x, int *incx,
     float *beta, void *y, int *incy);

void dgemv(const char *trans, int *m, int *n, double *alpha,
     void *a, int *lda, void *x, int *incx,
      double *beta, void *y, int *incy);
```

パラメーターを以下に示します。

trans
これは単一文字で入力行列の書式 *a* を示します。ここで、

- 'N' または 'n' は、*a* が計算に使用されることを示しています。
- 'T' または 't' は *a* の転置 (transpose) が計算に使用されることを示しています。

m 以下を表しています。

- 入力行列 *a* 内の行数
- ベクトル *y* の長さ、もし 'N' または 'n' が *trans* パラメーターに使用されている場合。
- ベクトル *x* の長さ、もし 'T' または 't' が *trans* パラメーターに使用されている場合。

行数はゼロ以上で、行列 *a* (*lda* に指定済み) の先導ディメンション (leading dimension) より小でなければならない。

n 以下を表しています。

- 入力行列 *a* 内の列数
- ベクトル *x* の長さ、もし 'N' または 'n' が *trans* パラメーターに使用されている場合。
- ベクトル *y* の長さ、もし 'T' または 't' が *trans* パラメーターに使用されている場合。

列数はゼロ以上でなければならない。

alpha
これは行列 *a* のスケーリング定数です。

a これは float (sgemv 用) または double (dgemv 用) 値の入力行列です。

lda
これは、*a* で指定される配列の先導ディメンション (leading dimension) です。先導ディメンション (leading dimension) はゼロより大でなければなりません。先導ディメンションは (leading dimension) 1 以上で、*m* で指定された値以上でなければなりません。

x これは、float (sgemv 用) の入力ベクトル、または double (dgemv 用) 値です。

incx
これはベクトル *x* のストライド。任意の値を持てる。

beta
これはベクトル *y* のスケーリング定数です。

y これは、float (sgemv 用) の出力ベクトル、または double (dgemv 用) 値です。

incy
これはベクトル *y* のストライド。ゼロであってはならない。

注: ベクトル *y* は、行列 *a*、またはベクトル *x* と共通エレメントを持ってはならない。そうでない場合、結果は予測不能です。

sgemm および dgemm 関数のプロトタイプを以下に示します。

```
void sgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, float *alpha,
   const void *a, int *lda, void *b, int *ldb,
   float *beta, void *c, int *ldc);

void dgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, double *alpha,
   const void *a, int *lda, void *b, int *ldb,
   double *beta, void *c, int *ldc);
```

パラメーターを以下に示します。

transa
これは単一文字で入力行列の書式 *a* を示します。ここで、

- 'N' または 'n' は、*a* が計算に使用されることを示しています。
- 'T' または 't' は *a* の転置 (transpose) が計算に使用されることを示しています。

transb
これは単一文字で入力行列の書式 *b* を示します。ここで、

- 'N' または 'n' は、*b* が計算に使用されることを示しています。
- 'T' または 't' は *b* の転置 (transpose) が計算に使用されることを示しています。

l 出力行列 *c* の行数を表します。行数はゼロ以上で、*c* の先導ディメンション (leading dimension) より小でなければならない。

n 出力行列 *c* の列数を表します。列数はゼロ以上でなければならない。

m 以下を表しています。

- 行列 *a* の列数。'N' または 'n' が *transa* パラメーターに対して使用されている場合
- 行列 *a* の行数。'T' または 't' が *transa* パラメーターに対して使用されている場合

そして、

- 行列 *b* の行数。'N' または 'n' が *transb* パラメーターに対して使用されている場合
- 行列 *b* の列数。'T' または 't' が *transb* パラメーターに対して使用されている場合

m はゼロ以上でなければなりません。

alpha
これは行列 *a* のスケーリング定数です。

a これは float (sgemm 用) または double (dgemm 用) 値の入力行列 *a* です。

lda
これは、*a* で指定される配列の先導ディメンション (leading dimension) です。先導ディメンション (leading dimension) はゼロより大でなければなりません。もし *transa* が 'N' または 'n' として指定されると、先導ディメンション (leading dimension) は 1 以上でなければなりません。もし *transa* が 'T' または 't' として指定されると、先導ディメンション (leading dimension) は *m* で指定された値以上でなければなりません。

b　これは float (sgemm 用) または double (dgemm 用) 値の入力行列 *b* です。

ldb
これは、*b* で指定される配列の先導ディメンション (leading dimension) です。先導ディメンション (leading dimension) はゼロより大でなければなりません。もし *transb* が 'N' または 'n' として指定されると、先導ディメンション (leading dimension) は *m* で指定された値以上でなければなりません。もし *transa* が 'T' または 't' として指定されると、先導ディメンション (leading dimension) は *n* で指定された値以上でなければなりません。

beta
これは行列 *c* のスケーリング定数です。

c　これは float (sgemm 用) または double (dgemm 用) 値の出力行列 *c* です。

ldc
これは、*c* で指定される配列の先導ディメンション (leading dimension) です。先導ディメンション (leading dimension) はゼロより大でなければなりません。もし *transb* が 'N' または 'n' として指定されると、先導ディメンション (leading dimension) は 0 以上で、 *l* で指定された値以上でなければなりません。

注: 行列 *c* は、行列 *a* または *b* と共通エレメントを持ってはならない。そうでない場合、結果は予測不能です。

libxlopt ライブラリーへのリンク

デフォルトで、libxlopt ライブラリーは XL C/C++ コンパイラーでコンパイルするいずれかのアプリケーションともリンクしています。しかし、ユーザーがサード・パーティー BLAS ライブラリーを使用している場合に、 libxlopt と一緒に出荷された BLAS ルーチンを使いたい場合には、 libxlopt ライブラリーを他のいずれかの BLAS ライブラリーの前に、リンク時にコマンド・ライン上で指定しなければなりません。例えば、他の BLAS ライブラリーが libblas.a と呼ばれる場合は、次のコマンドでコードをコンパイルします。

```
xlc app.c -lxlopt -lblas
```

コンパイラーは、 sgemv、dgemv、sgemm、および dgemm 関数を libxlopt ライブラリーから呼び出し、他のすべての BLAS 関数を libblas.a ライブラリーから呼び出します。

第 12 章 プログラムの並列処理

コンパイラーは、共用メモリー・プログラムの並列化のインプリメント方法として以下を提案します。

- 計数可能なプログラム・ループの自動並列処理で、 138 ページの『計数可能ループ』 に定義されています。コンパイラーの自動並列処理機能の概要が 139 ページの『自動並列処理の使用可能化』で提供されています。
- OpenMP アプリケーション・プログラム・インターフェース仕様に準拠したプラグマ・ディレクティブを使用した、 C および C++ プログラム・コードの明示的並列処理です。 OpenMP ディレクティブの概要は、 141 ページの『OpenMP ディレクティブの使用』 で提供されています。

プログラム並列処理のすべてのメソッドは、**omp** サブオプションなしに **-qsmp** コンパイラー・オプションが有効なとき、使用可能になります。ユーザーは厳格な OpenMP の整合性を **-qsmp=omp** コンパイラー・オプションによって使用可能にできますが、それを行うと自動並列処理が使用不可にされます。

注: -qsmp オプションはスレッド・セーフ・コンパイラー呼び出しモード (**_r** 接尾部を含むもの) と共に使用されなければなりません。

プログラム・コードの並列領域は、マルチスレッドによって、そしておそらくは複数処理装置上で実行されます。作成されるスレッドの数は、環境変数とライブラリー関数に対する呼び出しによって決定されます。作業は、環境変数で指定されるスケジューリング・アルゴリズムにしたがって、使用可能なスレッドに分散されます。並列処理のどのようなメソッドについても、XLSMPOPTS 環境変数と、スレッド・スケジューリングを制御するためのサブオプションが使用可能です。この環境変数の詳細については、「*XL C/C++ コンパイラー・リファレンス*」の *XLSMPOPTS* を参照してください。OpenMP 構成をご使用中の場合は、OpenMP 環境変数を使用してスレッド・スケジューリングを制御することができます。OpenMP 環境変数についての詳細については、「*XL C/C++ コンパイラー・リファレンス*」の *並列処理のための OpenMP 環境変数*を参照してください。 OpenMP 組み込み関数の詳細については、「*XL C/C++ コンパイラー・リファレンス*」の『*並列処理のための組み込み関数*』を参照してください。

OpenMP の構文、環境変数、およびランタイム・ルーチンについて詳しくは、http://www.openmp.org で提供されている「*OpenMP Application Program Interface Specification*」を参照してください。

関連情報:

58 ページの『共用メモリーの並列処理 (SMP) の使用』

「*XL C/C++ コンパイラー・リファレンス*」**の関連情報**

XLSMPOPTS

並列処理用の OpenMP 環境変数

並列処理のための組み込み関数

© Copyright IBM Corp. 1996, 2014

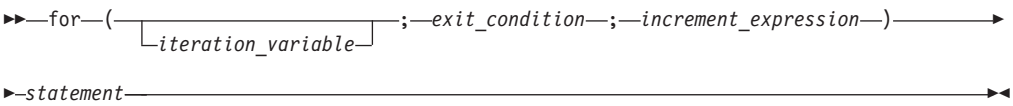
関連外部情報

「OpenMP Application Program Interface Language Specification」、http://www.openmp.org から入手可能

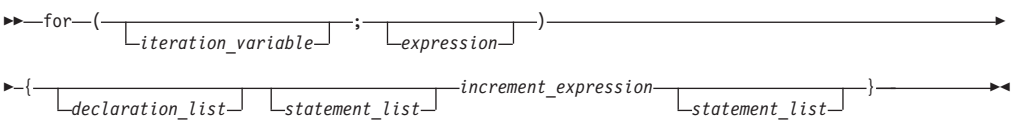
計数可能ループ

ループは、下記のいずれかの形であれば計数可能と考えられます。

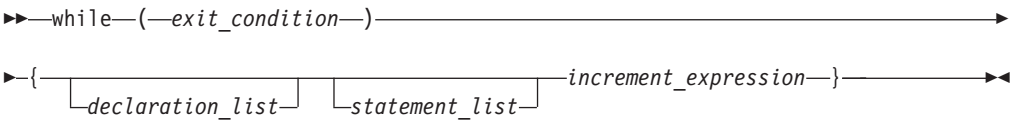
計数可能ループ構文 (Countable for loop syntax)



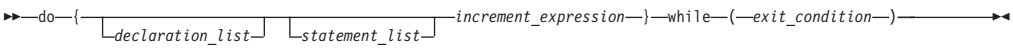
ステートメント・ブロック付きの計数可能ループ構文 (Countable for loop syntax with statement block)



計数可能ループ中構文 (Countable while loop syntax)



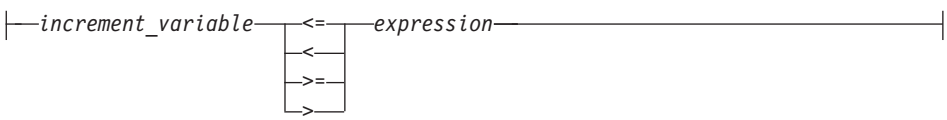
計数可能ループ中 Do 構文 (Countable do while loop syntax)



上記の構文図に以下の定義が適用されます。

iteration_variable
これは、自動または登録ストレージ・クラスを持つ符号付き整数で、決まったアドレスを持たず、 *increment_expression* 以外では、ループ内のどこでも変更されません。

exit_condition
次の書式を持っています。



ここで *expression* はループ・インバリアント符号付き整数式です。 *expression* は、外部変数または静的変数、ポインターまたはポインター式、関数呼び出し、またはアドレスを持つ変数を参照できません。

increment_expression
下記の任意の書式を使用できます。

- *++iteration_variable*
- *--iteration_variable*
- *iteration_variable++*
- *iteration_variable--*
- *iteration_variable += increment*
- *iteration_variable -= increment*
- *iteration_variable = iteration_variable + increment*
- *iteration_variable = increment + iteration_variable*
- *iteration_variable = iteration_variable - increment*

ここで *increment* はループ・インバリアント符号付き整数式です。 expression の値は実行時に知ることができますが 0 ではありません。*increment* は、外部変数または静的変数、ポインターまたはポインター式、関数呼び出し、またはアドレスを持つ変数を参照できません。

自動並列処理の使用可能化

コンパイラーは自動的に計数可能ループを探し、可能な場合はユーザーのプログラム・コード内の計数可能ループをすべて並列処理します。ループは、 138 ページの『計数可能ループ』 に示されているいずれかの書式であり、そして以下の条件に合っていれば countable と見なされます。

- ループに入出するブランチがない。
- *increment_expression* がクリティカル・セクションではない。

一般的に、計数可能ループは、下記のすべての条件が満たされたときにのみ自動的に並列処理されます。

- ループの反復が開始または終了する順序は、プログラムの結果に影響しない。
- ループには入出力操作がない。
- ループ内側の浮動小数点縮小は、**-qnostrict** オプションが有効でない限り、丸めエラーの影響を受けない。
- **-qnostrict_induction** コンパイラー・オプションが有効である。
- **-qsmp=auto** コンパイラー・オプションが有効である。
- コンパイラーは、スレッド・セーフ・コンパイラー呼び出しモードで呼び出される (**_r** 接尾部が含まれているもの)。

データ共用属性の規則

データ共用属性の規則に基づいて、`parallel` ディレクティブ、`task` ディレクティブ、および作業共用領域で参照される変数の属性が決定されます。

構文内で参照される変数のデータ共用属性規則

構文内で参照される変数のデータ共用属性は、次のカテゴリーに分類できます。

- 事前定義済みのデータ共用属性

- 明示的に決定されたデータ共用属性
- 暗黙的に決定されたデータ共用属性

囲まれた構文の firstprivate、lastprivate、または reduction 文節で変数を指定すると、囲んでいる構文内の変数への暗黙参照が開始されます。このような暗黙参照も、データ共用属性の規則に従います。

一部の変数やオブジェクトは、次のような事前定義済みのデータ共用属性を持っています。

- threadprivate ディレクティブで指定された変数は threadprivate 変数です。
- 構文内のスコープ内で宣言された自動ストレージ期間を持つ変数は専用です。
- 動的ストレージ期間を持つオブジェクトは共用されます。
- 静的データ・メンバーは共用されます。
- for 構文や parallel for 構文の関連付けられた for ループ内のループ反復変数は専用です。
- mutable メンバーを持っていない const 修飾型の変数は共用されます。
- 静的ストレージ期間を持つ変数は、構文内のスコープ内で宣言されている場合、共用されます。

事前定義済みのデータ共用属性を持つ変数をデータ共用属性文節内で指定することはできません。ただし、以下の場合は、事前定義済みの変数をデータ共用属性文節内で指定することが可能であり、その結果としてその変数の事前定義済みのデータ共用属性がオーバーライドされます。

- for 構文や parallel for 構文の関連付けられた for ループ内のループ反復変数は、private 文節内や lastprivate 文節内で指定できます。
- mutable メンバーを持っていない const 修飾型の変数は、firstprivate 文節内で指定できます。

以下の条件を満たす変数は、明示的に決定されたデータ共用属性を持っています。

- それらの変数が、任意の構文内で参照されている。
- それらの変数が、その構文上のデータ共用属性文節内で指定されている。

以下のすべての条件を満たす変数は、暗黙的に決定されたデータ共用属性を持っています。

- それらの変数が、任意の構文内で参照されている。
- それらの変数が、事前定義済みのデータ共用属性を持っていない。
- それらの変数が、その構文上のデータ共用属性文節内で指定されていない。

暗黙的に決定されたデータ共用属性を持っている変数の場合、規則は以下のとおりです。

- parallel または task 構文内では、変数のデータ共用属性は、default 文節によって決定されます (この文節がある場合)。
- parallel 構文内では、default 文節がない場合、変数は共用されます。
- task 以外の構文では、default 文節がない場合、変数のデータ共用属性は、囲んでいるコンテキストから継承されます。

- task 構文では、default 文節がない場合、囲んでいるコンテキスト内で共用されると決定された変数は、現在のチームにバインドされているすべての暗黙的タスクによって共用されます。
- task 構文では、default 文節がない場合、上記の規則によって決定されていないデータ共用属性を持つ変数は firstprivate 変数です。

領域内で参照されるが構文内では参照されない変数のデータ共用属性規則

領域内で参照されるが構文内では参照されない変数のデータ共用属性は、次のように決定されます。

- 静的ストレージ期間を持つ変数が、その領域内の呼び出し先ルーチンで宣言されている場合、それらの変数は共用されます。
- mutable メンバーを持っていない const 修飾型の変数が、呼び出し先ルーチンで宣言されている場合、それらの変数は共用されます。
- その領域内の呼び出し先ルーチンで参照されているファイル・スコープ変数や名前空間スコープ変数は、threadprivate ディレクティブで指定されていない限り、共用されます。
- 動的ストレージ期間を持つオブジェクトは共用されます。
- 静的データ・メンバーは、threadprivate ディレクティブで指定されていない限り、共用されます。
- その領域内の呼び出し先ルーチンの仮引数のうち、参照によって渡されるものは、関連付けられた実際の引数のデータ共用属性を継承します。
- その領域内の呼び出し先ルーチンで宣言された他の変数は専用です。

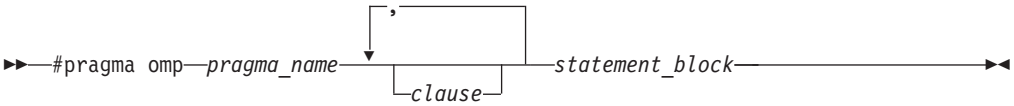
OpenMP ディレクティブの使用

OpenMP ディレクティブは、さまざまなタイプの並列領域を定義することによって、共用メモリーの並列処理を有効に活用します。並列領域には、プログラム・コードの反復セグメントと非反復セグメントの両方を組み込むことができます。

#pragma omp プラグマは、以下の一般的なカテゴリーに分かれます。

1. スレッドによって作業が並列に行われる並列領域をユーザーが定義できる **#pragma omp** プラグマ (**#pragma omp parallel**)。大部分の OpenMP ディレクティブは、静的または動的に囲まれている並列領域にバインドします。
2. 並列領域内で作業がどのように分散されるか、またはスレッドを共有するかをユーザーが定義できる **#pragma omp** プラグマ (**#pragma omp sections**、**#pragma omp for**、**#pragma omp single**、**#pragma omp task**)。
3. スレッド間の同期をユーザーがコントロールできる **#pragma omp** プラグマ (**#pragma omp atomic**、**#pragma omp master**、**#pragma omp barrier**、**#pragma omp critical**、**#pragma omp flush**、**#pragma omp ordered**)。
4. 同じスレッド内の複数の並列領域にまたがってデータ可視性の範囲を定義できる **#pragma omp** プラグマ (**#pragma omp threadprivate**)
5. 同期化用の **#pragma omp** プラグマ (**#pragma omp taskwait**, **#pragma omp barrier**)

OpenMP ディレクティブ構文



#pragma omp プラグマに特定の文節を追加することで、並列領域や作業共用領域の動作を微調整できます。例えば num_threads 文節を使用して、並列領域プラグマを制御することができます。

#pragma omp プラグマは、一般的にはそれが適用されるコードのセクションの直前に表示されます。以下の例は、for ループの反復を並列に実行できる並列領域を定義しています。

```
#pragma omp parallel
{
  #pragma omp for
    for (i=0; i<n; i++)
      ...
}
```

この例は、2 つ以上の非反復のプログラム・コードのセクションが並列に実行できる並列領域を定義しています。

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
       structured_block_1
            ...
    #pragma omp section
       structured_block_2
            ...
        ....
  }
}
```

OpenMP ディレクティブのプラグマごとの記述については、「*XL C/C++ コンパイラー・リファレンス*」の『*並列処理のためのプラグマ・ディレクティブ*』を参照してください。

「*XL C/C++ コンパイラー・リファレンス*」の関連情報

並列処理のためのプラグマ・ディレクティブ

OpenMP 組み込み関数

並列処理用の OpenMP 環境変数

並列環境内の共用変数と専用変数

並列環境内の変数は、共用、または専用のコンテキストを持つことができます。共用コンテキスト内の変数は、関連した並列領域内で実行中のすべてのスレッドに対して可視です。専用コンテキスト内の変数は、他のスレッドからは非表示です。各スレッドは自身の変数の専用コピーを持ち、スレッドがそのコピーに行った変更は他のスレッドには見えません。

変数のデフォルトのコンテキストは、下記の規則で決定されます。

- 静的ストレージ期間を持つ変数は共用されます。
- 動的に割り振られるオブジェクトは共用されます。
- 並列領域内で宣言された自動ストレージ期間を持つ変数は、専用です。
- ヒープ割り振りメモリー内の変数は共用です。共用ヒープは 1 つしかありません。
- 並列構成の外側で定義された変数は、並列領域に遭遇すると共用になります。
- ループ反復変数は、それらのループ内では専用です。ループの後の反復変数の値は、ループが連続して実行されたときと同一です。
- 並列ループ内で `alloca` 関数によって割り振られたメモリーは、そのループの 1 つの反復継続時間のみに対して主張し、各スレッドに対しては専用です。

以下のコード・セグメントは、これらのデフォルト・ルールの例を示します。

```
int E1;                          /* shared static     */

void main (argvc,...) {          /* argvc is shared   */
   int i;                         /* shared automatic  */

void *p = malloc(...);          /* memory allocated by malloc   */
                                   /* is accessible by all threads */
                                   /* and cannot be privatized     */

#pragma omp parallel firstprivate (p)
   {
     int b;                       /* private automatic  */
     static int s;                /* shared static      */

     #pragma omp for
     for (i =0;...) {
       b = 1;                     /* b is still private here !    */
       foo (i);                   /* i is private here because it */
                                  /* is an iteration variable     */
      }


#pragma omp parallel
     {
       b = 1;                     /* b is shared here because it  */
                                  /* is another parallel region   */
     }
   }
 }


int E2;                          /*shared static */

void foo (int x) {               /* x is private for the parallel */
```

```
                                /* region it was called from     */
int c;                         /* the same */
 ... }
```

いくつかの OpenMP 文節では、選択されたデータ変数に対する可視性コンテキストの指定をユーザーが行うことができます。データ有効範囲の属性文節の要約を以下にリストします。

データ有効範囲の属性文節	説明
private	**private** 文節は、チームの各スレッドについて専用となるため、変数をリストに宣言します。
firstprivate	**firstprivate** 文節は、専用文節が用意した機能性のスーパーセットを提供します。専用変数は、並列構文が検出されたときに、その変数の元の値によって初期化されます。
lastprivate	**lastprivate** 文節は、専用文節が用意した機能性のスーパーセットを提供します。専用変数は、並列構文の終了後に更新されます。
shared	**shared** 文節は、リスト内の変数をチーム内のすべてのスレッドによって共用されるものとして宣言します。 チームのすべてのスレッドは、同一のストレージ域の共用変数にアクセスします。
reduction	**reduction** 文節は、リストに表示されるスカラー変数について、指定された演算子を使って縮小を行います。
default	**default** 文節を使用すると、並列構文に含まれた変数のデータ共用属性を制御できます。

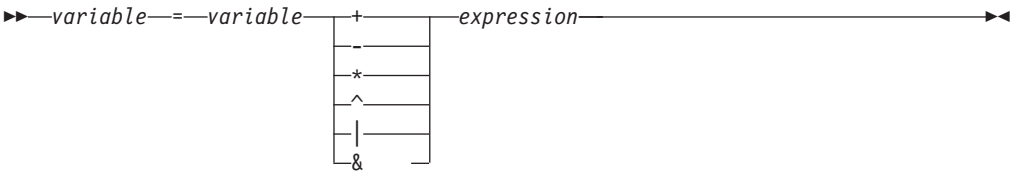
詳細については、「*XL C/C++ コンパイラー・リファレンス*」の『並列処理のためのプラグマ・ディレクティブ』に記載の OpenMP ディレクティブの説明を参照してください。また、「*OpenMP Application Program Interface Language Specification*」(http://www.openmp.org で入手可能) も参照してください。

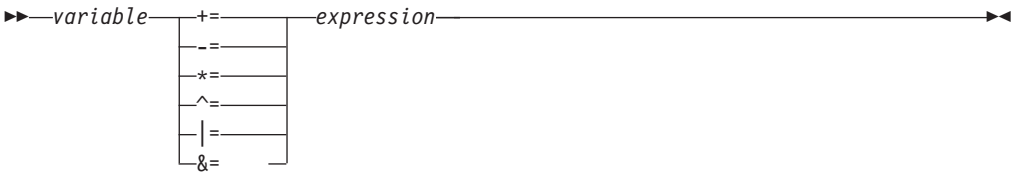
「*XL C/C++ コンパイラー・リファレンス*」の関連情報

並列処理のためのプラグマ・ディレクティブ

並列処理ループ内の縮小操作

コンパイラーは、自動、および明示的な並列処理中の、ループ内の縮小操作を大部分認識し、適切に処理することができます。特に、下記の書式のいずれかを持つ縮小文を処理することができます。





ここで、

variable
これは、自動または登録変数を指定する ID で、その決まったアドレスを持たず、ネストされた全ループを含むループ内の他のどこからも参照されません。例えば、次に示すコード内では、ネストされたループ内の S のみが縮小として認識されます。

```
int i,j, S=0;
for (i= 0 ;i < N; i++) {
    S = S+ i;
     for (j=0;j< M; j++) {
        S = S + j;
    }
}
```

expression
これは任意の有効な式です。

OpenMP ディレクティブは、縮小変数を明示的に指定する仕組みを提供しています。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒103-8510
東京都中央区日本橋箱崎町19番21号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

© Copyright IBM Corp. 1996, 2014

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、

利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、 IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. 1998, 2014.

商標

IBM、IBM ロゴおよび ibm.com® は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Adobe 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登録商標です。

UNIX は The Open Group の米国およびその他の国における登録商標です。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

© Copyright IBM Corp. 1996, 2014

[ラ行]

[数字]

B

C

F

I

L

M

O

S

T

X

[特殊文字]

プログラム番号: 5765-J08; 5725-C73

Printed in Japan

SA88-5402-00

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21