XL C/C++ Advanced Edition V7.0 for Linux

プログラミング・ガイド

バージョン 7.0

SC88-9976-01
(英文原典：SC09-7943-01)

XL C/C++ Advanced Edition V7.0 for Linux

プログラミング・ガイド

バージョン 7.0

SC88-9976-01
(英文原典：SC09-7943-01)

ご注意

本書および本書で紹介する製品をご使用になる前に、 55 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、XL C/C++ Advanced Edition V7.0 for Linux のバージョン 7.0 (プロダクト番号 5724-K77)、および新しい版で特に明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

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

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

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

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

(URL は、変更になる場合があります)

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

原　典： SC09-7943-01
XL C/C++ Advanced Edition V7.0 for Linux
Programming Guide
Version 7.0

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2005.1

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

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

目次

本書について

本書では、IBM® XL C/C++ Advanced Edition V7.0 for Linux® コンパイラーの使用に関する高度なトピックを、プログラムの移植性と最適化に重点を置いて説明しています。本書では、コンパイラーの機能を最大限に引き出すための参照情報および実用的なヒントを、推奨されるプログラミングの実例とコンパイル・プロシージャーによって提供します。また、XL C/C++ Advanced Edition V7.0 for Linux の資料セットに含まれる他のリファレンス・ガイドの関連セクションとの相互参照も充実しています。

本書では、以下のトピックを取り上げます。

- 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』では、既存の 32 ビット・アプリケーションを 64 ビット・モードに移植する際に生じる一般的な問題について説明し、それを回避するために推奨される方法について述べます。
- 7 ページの『第 2 章 集合体内のデータの位置合わせ』では、集合体 (構造体やクラスなど) 内のデータの位置合わせを制御する際にあらゆるプラットフォームで使用できるさまざまなコンパイラー・オプションについて説明します。
- 13 ページの『第 3 章 浮動小数点演算の処理』では、コンパイラーによる浮動小数点演算の処理方法を制御する際に使用できるオプションについて説明します。
- 17 ページの『第 4 章 C++ テンプレートの使用』では、 C++ テンプレートが組み込まれているプログラムをコンパイルする場合のさまざまなオプションについて説明します。
- 23 ページの『第 5 章 ライブラリーの構成』では、静的ライブラリーと共用ライブラリーのコンパイルおよびリンク方法について、および C++ プログラム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。
- 31 ページの『第 6 章 アプリケーションの最適化』では、プログラム最適化のためにコンパイラーが提供する各種オプションについて説明し、それぞれのオプションの推奨される使用方法を紹介します。
- 47 ページの『第 7 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』では、プログラムのパフォーマンスおよびコンパイラーの最適化機能との互換性を高めるために推奨されるプログラミングの実例とコーディング手法について説明します。

文書の規則

強調表示の規則

本書では、以下の強調表示規則を使用します。

太字 コマンド、キーワード、ファイル、ディレクトリー、およびシステムによって名前が事前定義されているパス名、環境変数、実行可能ファイル名、その他の項目を示します。

イタリック　その実際の名前または値がプログラマーによって提供されるパラメーターを識別します。*イタリック* は、新規用語を最初に言及する際にも使用されます。

モノスペース　プログラム・コード例を示します。

これらの例は、言語の使用方法を説明するもので、実行時間の最小化、ストレージの節約、エラーのチェックを行うためのものではありません。例では、使用しうるすべての言語構成の使用法を例示しているわけではありません。一部の例では、コードの一部分のみを示すだけに留まり、コードを追加しなければコンパイルできません。

アイコン

一般に本書では、XL C/C++ 機能を Linux プラットフォームでインプリメントされているとおりに説明しています。ただし、他のプラットフォームへの移植性に影響を与える問題について説明している箇所では、以下のアイコンを使用します。

AIX® プラットフォームでサポートされている機能を示します。

Linux プラットフォームでサポートされている機能を示します。

Mac OS X プラットフォームでサポートされている機能を示します。

C++ 言語でのみサポートされている機能を示します。

C 言語でのみサポートされている機能を示します。

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++ を使用すると、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両方を開発することができます。それには、コンパイル時に、それぞれ、**-q32** (デフォルト) または **-q64** と指定してください。

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++ 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起因します。次の表は、その違いをまとめたものです。

表 1. 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、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 (システム定義の unsigned long)	4 バイト	4 バイト境界	8 バイト	8 バイト境界
ptrdiff_t (システム定義の long)	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・モードまたは 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、不具合 (切り捨てやデータ損失など) が発生した場合は、コンパイラーが即時に警告を発します。

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については、52 ページの『64 ビット・モードでの演算の最適化』を参照してください。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-q32/-q64**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qwarn64**

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 2. 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 型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、『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、引き数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。

他の変数に割り当てたり、関数に渡されるときに long 型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 パラメーターを、それが渡される関数によって受け入れられるようにする。

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のない定数を「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・システムで許容される限界を超える値

を表します。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。

- 定数 >= **UINT_MAX**
- 定数 < **INT_MIN**
- 定数 > **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 3. 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	**同等の値**	**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 型定数の場合) または **UL** (unsigned long 型定数の場合) 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。

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。下記の表の例は、以下のコード・セグメントを使用して long 型定数でビット・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 4. long 値のビット・シフトの結果

初期値	**シンボリック定数**	**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	0x1FFFFFFFE

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。

32 ビット・モードでは、以下のようなコード構文は、

```
a=(char*) calloc(25);
```

calloc に対する関数プロトタイプがなくても有効ですが、コンパイラーは、この関数が **int** を戻すものと想定するので、a は自動的に切り捨てられ、その後、符号拡張されます。**calloc** がメモリー内で割り振ったアドレスは、戻されるときに既に切り捨てられているため、結果をキャストする型は、切り捨てを免れることはできません。この例での適切な解決策は、**calloc** のプロトタイプを含む、該当するヘッダー・ファイル **stdlib.h** を組み込むことです。

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。

- ポインターを戻す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。

集合体データ位置合わせ

構造体は、32 ビット・モードでも 64 ビット・モードでも、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、変更されるか、さもなければ位置合わせが破壊されます。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。

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、7 ページの『第 2 章 集合体内の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

アプリケーションには、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するものが少なくありません。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 サイドで行うよりも 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 5. 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
		POINTER (4 バイト)
		POINTER*8 (8 バイト)

第 2 章 集合体内の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、あるいは 32 ビット・モードと 64 ビット・モードの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。

『位置合わせモードおよび修飾子の使用』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位置合わせのデフォルト設定、集合体および集合体メンバーの位置合わせの制御に使用できるオプション、および集合体位置合わせの一般規則について説明します。

10 ページの『ビット・フィールドの使用と位置合わせ』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。

位置合わせモードおよび修飾子の使用

さまざまなデータ型を含む集合体 (C および C++ の構造体や共用体、C++ のクラスなど) 内部では、XL C/C++ がサポートする各データ型は、次のように、プラットフォーム固有のデフォルトに従って、バイト境界に沿って位置合わせされます。

- AIX **power** または **full** (これらは等価です)
- Linux **linuxppc**
- Mac OS X **power**

上記の各設定は、8 ページの表 6 で定義されています。

データの位置合わせは、位置合わせモード と位置合わせ*修飾子* で明示的に制御することもできます。位置合わせモード では、次のようなことができます。

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集合体の位置合わせを設定する

この方法を使用するためには、コンパイル時に、**-qalign** コンパイラー・オプションを指定します。8 ページの表 6 に、 **-qalign** の有効なサブオプションがプラットフォームごとに示されています。

ファイル内の単一または複数の集合体の位置合わせを設定する

この方法を使用するには、ソース・ファイルで、**#pragma align** または **#pragma options align** ディレクティブを指定します。8 ページの表 6 に、 **#pragma align** の有効なサブオプションがプラットフォームごとに示されています。各ディレクティブは、別のディレクティブに遭遇するまで、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に従うすべての集合体で有効な位置合わせ規則を変更します。

単一集合体の位置合わせを設定する

#pragma align ディレクティブに加えて、ソース・ファイルでは以下を使用できます。

- 構造体宣言に **__attribute__((aligned(n)))** 型属性を組み込む。*n* の値は、2 の正の累乗でなければなりません。 **__attribute__((aligned))** を集合体の型属性に使用する場合の正しい構文については、「*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。
- 構造体宣言に **__align(n)** 指定子を組み込む。 *n* の値は 2 の正の累乗です。

位置合わせ*修飾子* では、次のようなことができます。

集合体内のすべてのメンバーの位置合わせを設定する

この方法を使用するには、ソース・ファイルで以下のいずれかを使用してください。

- 構造体宣言の前に **#pragma pack** ディレクティブを組み込む。このディレクティブの有効な値については、「*XL C/C++ コンパイラー・リファレンス*」の **#pragma pack** を参照してください。
- 構造体宣言に **__attribute__((packed))** 型属性を組み込む。 **__attribute__((packed))** を型属性に使用する場合の正しい構文については、「*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。

集合体内の単一メンバーの位置合わせを設定する

この方法を使用するには、**__attribute__((packed))** または **__attribute__((aligned(n)))** 型属性または変数属性を、構造体宣言に組み込みます。 **__attribute__((aligned(n)))** の *n* の値は、 2 の正の累乗でなければなりません。変数属性について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『aligned 変数属性』および『packed 変数属性』を参照してください。型属性については、「*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。

注: **__align** 指定子および **__attribute__((aligned))** 属性では、 vector 型の位置合わせは変更されません。

表 6. 位置合わせ設定

		位置合わせ設定とサポートされるプラットフォーム							
		natural	**power**	**full**	**mac68k**	**twobyte**	**linuxppc**	**bit_packed**	**packed**
データ型	**ストレージ**	**AIX** **Mac**	**AIX** **Mac**	**AIX**	**AIX** **Mac**	**AIX**	**Linux**	**AIX** **Mac** **Linux**	**AIX**
_Bool (C)、bool (C++)、_Bool	1 バイト	適用外	適用外		適用外		1 バイト	1 バイト	適用外
char、signed char、unsigned char	1 バイト	1 バイト	1 バイト		1 バイト		1 バイト	1 バイト	
wchar_t (32 ビット・モード)	2 バイト	2 バイト	2 バイト		2 バイト		2 バイト	1 バイト	
wchar_t (64 ビット・モード)	4 バイト	4 バイト	4 バイト		サポート対象外[2]		4 バイト	1 バイト	

表 6. 位置合わせ設定 (続き)

データ型	ストレージ	位置合わせ設定とサポートされるプラットフォーム							
		natural	**power**	**full**	**mac68k**	**twobyte**	**linuxppc**	**bit_packed**	**packed**
		AIX **Mac**	**AIX** **Mac**	**AIX**	**AIX** **Mac**	**AIX**	**Linux**	**AIX** **Mac** **Linux**	**AIX**
int、unsigned int	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
short int、unsigned short int	2 バイト	2 バイト	2 バイト		2 バイト		2 バイト	1 バイト	
long int、unsigned long int (32 ビット・モード)	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
long int、unsigned long int (64 ビット・モード)	8 バイト	8 バイト	8 バイト		サポート対象外[2]		8 バイト	1 バイト	
long long	8 バイト	8 バイト	8 バイト		2 バイト		8 バイト	1 バイト	
float	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
double	8 バイト	8 バイト	注を参照 [1]		2 バイト		8 バイト	1 バイト	
long double	8 バイト	8 バイト	注を参照 [1]		2 バイト		8 バイト	1 バイト	
pointer (32 ビット・モード)	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
pointer (64 ビット・モード)	8 バイト	8 バイト	8 バイト		サポート対象外[2]		8 バイト	1 バイト	
ベクトル型 [3]	16 バイト	16 バイト	16 バイト	適用外	16 バイト	適用外	16 バイト	1 バイト	適用外

注:

1. これらの型は、集合体の最初のメンバーに対しては **natural** 位置合わせを使用し、2 番目以降のメンバーに対しては 4 バイトまたは **natural** 位置合わせ (いずれか値の小さい方) を使用します。
2. この型のメンバーで集合体を宣言し、この位置合わせ設定でコンパイルしようとすると、コンパイラーは警告メッセージを出し、該当するプラットフォームのデフォルトの位置合わせ設定を使用してコンパイルを行います。
3. **-qaltivec** コンパイラー・オプションが有効になっている場合に限り、サポートされます。

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、プラットフォームに依存しない位置合わせモード (**#pragma pack**、**qalign=bit_packed** など) を使用する必要があります。

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ていない可能性があります。

位置合わせの一般的規則

集合体の位置合わせを 8 ページの表 6 にリストされている設定のいずれかで制御する場合は、以下の規則が適用されます。

- すべての位置合わせ設定で、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内包することのできる最小の倍数となる。
- **mac68k** を除くすべての位置合わせ設定で、集合体の*位置合わせ* は、そのメンバーの最大の位置合わせ値と等しい。
- 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。

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・フィールドの使用と位置合わせ』を参照してください。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qalign**
- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma align**
- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma pack**
- 「*XL C/C++ ランゲージ・リファレンス*」の『宣言』内の、『**aligned** 変数属性』、『**packed** 変数属性』、『**__align** 指定子』、および『型属性』
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qaltivec**

ビット・フィールドの使用と位置合わせ

ビット・フィールドは、**_Bool** (C)、 **bool** (C++)、**char**、**signed char**、**unsigned char**、**short**、 **unsigned short**、**int**、**unsigned int**、**long**、**unsigned long**、 **long long**、または **unsigned long long** データ型として宣言することができます。 ビット・フィールドは、宣言される基本型とコンパイル・モード (32 ビットまたは 64 ビット) に応じて、常に 4 バイトまたは 8 バイトになります。

C C 言語では、ビット・フィールドを、**int** ではなく **char** または **short** として指定することができますが、XL C/C++ はそれらを、**unsigned int** としてマップします。ビット・フィールドの長さは、その基本型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・フィールドに作用する **sizeof** 演算子は、常に 4 を返します。)

C++ ビット・フィールドの長さは、その基本型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。

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、指定する位置合わせ設定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・フィールド位置合わせの例

ビット・パックの例

下の例では、

```
#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

第 3 章 浮動小数点演算の処理

XL C/C++ は、およそ 10^{-38} から 10^{+38} の範囲で、 10 進法で約 7 桁の精度を持つ単精度の浮動小数点数と、およそ 10^{-308} から 10^{+308} の範囲で、 10 進法で約 16 桁の精度を持つ倍精度の浮動小数点数をサポートしています。

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。

- 『乗加法演算の処理』
- 『浮動小数点丸めの処理』
- 14 ページの『浮動小数点例外の処理』
- 15 ページの『数学関数加速サブシステム (MASS) の使用』

乗加法演算の処理

デフォルトでは、コンパイラーは、パフォーマンスを向上させるために、特定の IEEE 754 浮動小数点規則に違反することになっています。例えば、デフォルトで乗算 - 加算命令が生成されるのは、その方が、乗算命令と加算命令を個別に行うよりも速く、しかも正確な結果が得られるためです。他のシステムで可能な精度との高度な互換性が必要な場合は、 **-qfloat=nomaf** オプションを使用すると、これらの乗算 - 加算命令を生成しないようにすることができます。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qfloat**

浮動小数点丸めの処理

デフォルトでは、コンパイラーは、コンパイル時に可能な限り算術演算を実行しようとします。オペランドが定数の浮動小数点演算では、フォールディング が行われます。つまり、算術式の代わりにコンパイル時の結果が表示されます。最適化が使用可能になっている場合には、フォールディングが増大することがあります。ただし、コンパイル時の計算結果は、実行時に計算した場合の結果とわずかに異なる場合があります。これは、コンパイル時には丸め操作がより多く発生するためです。例えば、実行時に乗算/加算融合 (MAF) 演算が使用されて丸めが少なくなるような部分では、コンパイル時に乗算と加算が別々に行われることがあり、その場合は結果にわずかな違いを生じます。

コンパイル時の丸めによる予期しない結果を避けるには、次の 2 つの方法があります。

- **-qfloat=nofold** コンパイラー・オプションを使用して、浮動小数点計算のコンパイル時のフォールディングをすべて抑制する。
- **-y** コンパイラー・オプションを使用して、実行時に使用する丸めモードと一致する IEEE コンパイル時の丸めモードを指定する。特に別の値を指定しない限り、

デフォルトでは、*最近似値* への丸めモードになっています (別の値を指定するには、**builtins.h** ファイルで宣言した、XL C/C++ 組み込み関数 **__setrnd** を使用します)。

例えば、次のコード例を **-yz** (丸めモードを切り捨てに指定するコマンド) でコンパイルすると、 u.x の 2 つの結果は少し違うものになります。

```
int main ()
 {
  union uu
   {
    float x;
    int i;
    } u;

    volatile float one, three;

    u.x=1.0/3.0;
    printf("1/3=%8X ¥n", u.i);

    one=1.0;
    three=3.0;
    u.x=one/three;
    printf ("1/3=%8X ¥n", u.i);
    return 0;
 }
```

これは、1.0/3.0 の計算が、コンパイル時には切り捨てによってフォールディングされるのに対して、実行時には one/three がデフォルトの最近値への丸めモードで計算されるためです。 (変数 one および three を **volatile** と宣言することで、最適化を行っても、コンパイラーによるフォールディングは抑制されます。)このプログラムの出力は、次のようになります。

```
1/3=3EAAAAAA
1/3=3EAAAAAB
```

この例でコンパイル時の結果と実行時の結果に整合性を持たせるためには、オプション **-yn** (これがデフォルトです) を指定してコンパイルします。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qfloat**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-y**
- 「*XL C/C++ コンパイラー・リファレンス*」の『付録 B: 組み込み関数』に記載の **__setrnd**

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と、このようなタイプの例外を検出することができます。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、コンパイル時にはフォールディング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、コンパイル時のフォールディングをすべて抑止することを検討してください。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qfloat**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qflttrap**

数学関数加速サブシステム (MASS) の使用

XL C/C++ Advanced Edition V7.0 for Linux は、数学関数加速サブシステム (MASS) を出荷しています。これは、対応する **libm.a** ライブラリー関数で改善されたパフォーマンスを提供する、調整された数学組み込み関数のライブラリー・セットです。 MASS 関数と **libm.a** 関数では、精度と例外処理が異なる場合があります。

Linux の MASS ライブラリーは、『ベクトル・ライブラリーの使用』で説明されているベクトル関数のライブラリーで構成されます。 16 ページの『MASS によ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について説明します。

ベクトル・ライブラリーの使用

Linux 用の MASS ライブラリーは、AIX 用 MASS ライブラリーのサブセットです。Linux では、32 ビットのオブジェクトと 64 ビットのオブジェクトを 1 つのライブラリーに混在させてはなりません。そこで、MASS ライブラリーには 2 つのバージョン、**libmassvp4.a** (32 ビット) と **libmassvp4_64.a** (64 ビット) が用意されています。

ベクトル・ライブラリーに含まれている単精度関数と倍精度関数については、表 7 にまとめられています。関数のプロトタイプを提供するには、ソース・ファイルに **massv.h** を追加します。 C および C++ アプリケーションでは、スカラー引き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。

表 7. MASS ベクトル・ライブラリー関数

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
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);
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);
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);

MASS ベクトル関数の整合性

一定の入力値は、ベクトル内の位置やベクトルの長さに関係なく、常に同じ結果をもたらすという点で、MASS ベクトル・ライブラリー内のすべての関数には一貫性があります。

MASS によるプログラムのコンパイルとリンク

MASS ライブラリーでルーチンを呼び出すアプリケーションをコンパイルするには、**-l** リンカー・オプションで、**massvp4** (32 ビット) または **massvp4_64** (64 ビット) を指定します。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。

```
xlc progc.c -o progf -lmassvp4
xlc progc.c -o progf -lmassvp4_64 -q64
```

MASS 関数を実行する際には、最近似値への丸めモードにし、浮動小数点例外トラッピングを使用不可にしてください。 (これが、デフォルトのコンパイル設定です。)

第 4 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

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。その場合の引き数は、同じであっても異なっていてもかまいません。同じ引き数を使用する場合は、繰り返されるインスタンス生成は冗長になります。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 4 つの方法があります。

固有のインスタンス生成のためのコーディングを行う

ソース・コードを、オブジェクト・ファイルに必要なインスタンス生成ごとにインスタンスが 1 つだけ含まれ、未使用のインスタンス生成が含まれないように編成します。この方法は、最も使用頻度の低いものです。この方法をとるには、個々のテンプレートがどこで定義され、個々のテンプレート・インスタンス生成がどこで必要になるかを知っている必要があるためです。

出現するたびにインスタンスを生成する

-qnotempinc および **-qnotemplateregistry** コンパイラー・オプションを使用します (これらはデフォルト設定です)。すると、コンパイラーは、インスタンス生成が必要になるたびにそのためのコードを生成します。この方法では、冗長なインスタンス生成の欠点は改善されません。

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き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。この方法については、 18 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。

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き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。**-qtemplateregistry** コンパイラー・オプションには、 **-qtempinc** コンパイラ

ー・オプションと同様の利点がありますが、テンプレート定義ファイルおよびテンプレート・インプリメンテーション・ファイルのための特定の構造は必要ありません。この方法については、 21 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。

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、同時に使用することはできません。

-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・ヘッダー・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。
- テンプレート宣言ファイルごとに、テンプレート・インプリメンテーション・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。

-qtempinc の例

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: `stack.h`
- それ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: `stack.c`
- 関数プロトタイプ: `stackops.h` (関数テンプレートではありません)
- それに対応する関数インプリメンテーション・ファイル: `stackops.cpp`

- メインプログラムのソース・ファイル: stackadd.cpp

この例では、

1. どちらのソース・ファイルにも、テンプレート宣言ファイル stack.h が組み込まれています。
2. どちらのソース・ファイルにも、関数プロトタイプ stackops.h が組み込まれています。
3. テンプレート宣言ファイルには、プログラムが **-qnotempinc** でコンパイルされている場合は、テンプレート・インプリメンテーション・ファイル stack.c が組み込まれています。

テンプレート宣言ファイル: stack.h

このヘッダー・ファイルは、クラス Stack のクラス・テンプレートを定義するものです。

```
    #ifndef STACK_H
    #define STACK_H

    template <class Item, int size> class Stack {
      public:
          void push(Item item);  // Push operator
          Item pop();            // Pop operator
          int isEmpty(){
              return (top==0);   // Returns true if empty, otherwise false
      }
          Stack() { top = 0; }   // Constructor defined inline
      private:
          Item stack[size];      // The stack of items
          int   top;             // Index to top of stack
};

    #ifndef __TEMPINC__          //  3
    #include "stack.c"           //  3
    #endif                       //  3
    #endif
```

テンプレート・インプリメンテーション・ファイル: stack.c

このファイルは、クラス Stack のクラス・テンプレートのインプリメンテーションを提供するものです。

```
      template <class Item, int size>
      void Stack<Item,size>::push(Item item) {
        if (top >= size) throw size;
        stack[top++] = item;
      }
   template <class Item, int size>
      Item Stack<Item,size>::pop() {
         if (top <= 0) throw size;
         Item item = stack[--top];
         return(item);
      }
```

関数宣言ファイル: stackops.h

このヘッダー・ファイルには、add 関数のプロトタイプが含まれています。このプロトタイプは、stackadd.cpp および stackops.cpp で使用されます。

```
    void add(Stack<int, 50>& s);
```

関数インプリメンテーション・ファイル: stackops.cpp

このファイルは、add 関数のインプリメンテーションを提供するものです。このインプリメンテーションは、メインプログラムから呼び出されます。

```
#include "stack.h"            // 1
#include "stackops.h"         // 2

void add(Stack<int, 50>& s) {
  int tot = s.pop() + s.pop();
  s.push(tot);
  return;
}
```

メインプログラム・ファイル: stackadd.cpp

このファイルで、Stack オブジェクトが作成されます。

```
#include <iostream.h>
#include "stack.h"            // 1
#include "stackops.h"         // 2

main() {
  Stack<int, 50> s;           // create a stack of ints
  int left=10, right=20;
  int sum;

  s.push(left);               // push 10 on the stack
  s.push(right);              // push 20 on the stack
  add(s);                     // pop the 2 numbers off the stack
                              // and push the sum onto the stack
  sum = s.pop();              // pop the sum off the stack

  cout << "The sum of: " << left << " and: " << right << " is: " << sum << endl;

  return(0);
}
```

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、ソース・ファイルは共用できますが、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。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qtempinc**
- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma implementation**

-qtemplateregistry コンパイラー・オプションの使用

-qtempinc とは異なり、 **-qtemplateregistry** コンパイラー・オプションでは、ソース・コードの編成に特定の要件を必要としません。 **-qnotempinc** で正常にコンパイルできるプログラムなら、 **-qtemplateregistry** でもコンパイルできます。

テンプレート・レジストリーでは、「先着順」のアルゴリズムが使用されます。

- プログラムが新規のインスタンス生成を初めて参照するとき、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生成されます。
- 別のコンパイル単位が同じインスタンス生成を参照すると、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。

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、コンパイラーはリンク・ステップ時に自動的に B を再コンパイルして、 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。

-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。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場合は、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。 18 ページの『-qtempinc の例』の図を参照してください。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qtemplateregistry**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qtemplaterecompile**

第 5 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。

24 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*によって、 C++ アプリケーションに含まれる複数のファイルで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

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。
2. GCC **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。

たとえば次のようになります。

```
xlc -c bar.c example.c
ar -rv libfoo.a bar.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、次のようにします。

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・ファイルを作成する。たとえば次のようになります。

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

2. **-qmkshrobj** コンパイラー・オプションを使用して、生成したオブジェクト・ファイルから共用オブジェクトを作成する。たとえば次のようになります。

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

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。たとえば次のようになります。

```
xlc -o myprogram main.c -Ldirectory -lfoo
```

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

-l オプションを使用すると、リンカーは、 **-L** オプションで指定したディレクトリーで `libfoo.so` を検索します。見つからない場合は、`libfoo.a` を検索します。その他のリンケージ・オプション (デフォルトの振る舞いを変更するオプションなど) については、GCC **ld** の資料を参照してください。

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。たとえば次のようになります。

```
xlc -qmkshrobj -o mylib.so myfile.o -Ldirectory -lfoo
```

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qmkshrobj**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-l**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-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オブジェクトが適切に構成されるように指定します。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、 27 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。

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。

AIX Linux **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。

AIX Linux Mac OS X **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** ディレクティブの優先順位番号は、 **-qpriority** オプションを使用する場合は、そこで指定される番号より大きくしなければなりません。また、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。

Linux Mac OS X **個々のオブジェクト別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。 Mac OS X では、オブジェクトは優先順位に従って初期化され、 1 つのコンパイル単位内 でのみ、その逆の順序で終了します。

AIX AIX に限って、それ以外に、 **-qmkshrobj** コンパイラー・オプションの優先順位サブオプションを使用して、共用ライブラリー全体の優先順位を設定することもできます。AIX では、ロードと初期化は別々のプロセスとして発生するので、ファイル (またはファイル内のオブジェクト) に割り当てられる優先順位番号は、ライブラリーに割り当てられる優先順位番号とはまったく無関係であり、シーケンスに従う必要はありません。

優先順位番号の使用

AIX 優先順位番号の範囲は、 -2147483643 から 2147483647 までです。ただし、-2147483648 ～ -2147482624 までの番号はシステム用になっています。指定できる最小の優先順位番号は -2147482623 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 2147483647 で、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 0 (ゼロ) になります。

Linux Mac OS X 優先順位番号の範囲は 101 から 65535 までです。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。 27 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、 Linux プラットフォームでの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 と 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記述したものです。 2 つのファイルは、ともに **#pragma priority** ディレクティブを使用し、 **-qpriority** オプションでコンパイルされています。

farm.C -qpriority=2000

```
#pragma priority(3000)
...
Dog a ;
Dog b ;
...
#pragma priority(6000)
...
Cat c ;
Cow d ;
...
#pragma priority(7000)
Mouse e ;
...
```

zoo.C -qpriority=2000

```
...
Lion k ;
#pragma priority(4000)
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	Lion k	2000	ファイル zoo.o の優先順位番号 (2000) を使用 (最初に初期化)。
2	Dog a	3000	pragma の優先順位 (3000) を使用。
3	Dog b	3000	Dog a と同じ。
4	Bear m	4000	pragma で指定された、次の優先順位番号 (4000)。
5	Zebra n	5000	pragma で指定された、次の優先順位番号 (5000)。
6	Snake s	5000	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
7	Cat c	6000	次の優先順位番号。
8	Cow d	6000	同じ優先順位で続く。
9	Mouse e	7000	次の優先順位番号。
10	Frog f	8000	次の優先順位番号 (最後に初期化)。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qpriority**
- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma priority**
- 「*XL C/C++ ランゲージ・リファレンス*」の『宣言』に記載の『**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

実行時にアプリケーションのすべてのモジュールがロードされると、モジュールは、その優先順位に従って初期化されます (**main** 関数を含む実行可能プログラムには、常に優先順位 0 が割り当てられます)。ライブラリー内でオブジェクトが初期化される場合は、初期化の順序は、 24 ページの『オブジェクトへの優先順位の割り当て』で説明されている規則に従います。 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・ファイルはランダムに初期化され、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。

静的ライブラリーと共用ライブラリーはそれぞれ、その依存関係がすべてロードされ、初期化された後に、実行時にはリンク順序とは*逆* の順序でロードおよび初期化されます。リンク順序とは、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行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、 24 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序の逆順で初期化され (リンク順序とは、ファイルがライブラリーにリンクするときにコマンド行で与えられた順序のことです)、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 関数に含まれる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。

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 fileE.o fileF.o
xlC -qmkshrobj -o libS4 fileG.o fileH.o
xlC -qmkshrobj -o libS5 fileI.o fileJ.o
xlC -qmkshrobj -o libS6 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。

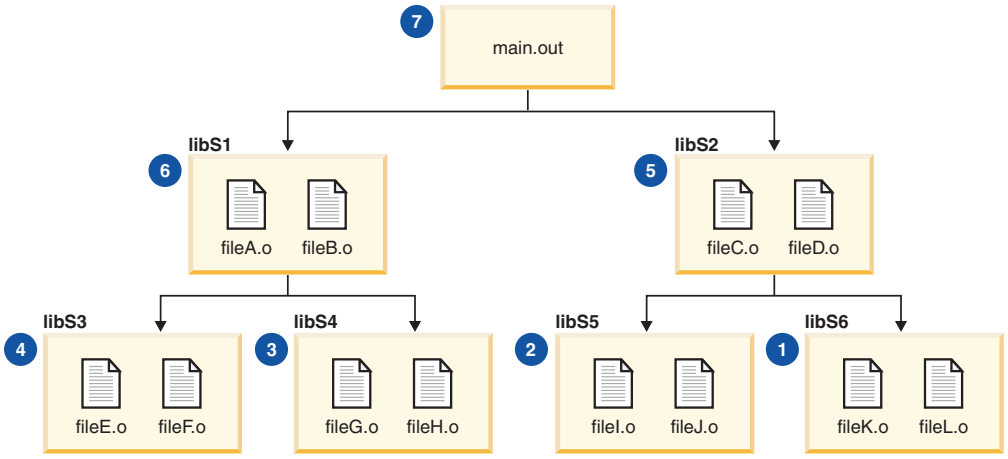
```
xlC -qmkshrobj libS1 fileA.o fileB.o -L. -lS3 -lS4
xlC -qmkshrobj libS2 fileC.o fileD.o -L. -lS5 -lS6
```

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. -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 でのオブジェクト初期化順序



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	コメント
1	libS6	libS2 は、main とリンクされる際、コマンド行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 とリンクされる際にコマンド行で最後に入力されたため、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。 (優先順位が割り当てられていない場合、fileL 内のオブジェクトが fileK 内のオブジェクトよりも前に初期化されます。これは、オブジェクト・ファイルが libS6 にリンクされたときにコマンド行で fileL が最後にリストされたためです。)
2	libS5	libS5 は、libS2 とリンクされる際、コマンド行で 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。 (優先順位が割り当てられていない場合、fileJ 内のオブジェクトが fileI 内のオブジェクトよりも前に初期化されます。これは、オブジェクト・ファイルが libS5 にリンクされたときにコマンド行で fileJ が最後にリストされたためです。)
3	libS4	libS4 は、libS1 と依存関係にあり、 libS1 とリンクする際に、コマンド行で最後に入力されたため、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。 (優先順位が割り当てられていない場合、fileH 内のオブジェクトが fileG 内のオブジェクトよりも前に初期化されます。これは、オブジェクト・ファイルが libS4 にリンクされたときにコマンド行で fileH が最後にリストされたためです。)
4	libS3	libS3 は、libS1 と依存関係にあり、 libS1 とリンクする際に、コマンド行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。 (優先順位が割り当てられていない場合、fileF 内のオブジェクトが fileE 内のオブジェクトよりも前に初期化されます。これは、オブジェクト・ファイルが libS3 にリンクされたときにコマンド行で fileF が最後にリストされたためです。)
5	libS2	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。 (優先順位が割り当てられていない場合、fileD 内のオブジェクトが fileC 内のオブジェクトよりも前に初期化されます。これは、オブジェクト・ファイルが libS2 にリンクされたときにコマンド行で fileD が最後にリストされたためです。)
6	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。 (優先順位が割り当てられていない場合、fileB 内のオブジェクトが fileA 内のオブジェクトよりも前に初期化されます。これは、オブジェクト・ファイルが libS1 にリンクされたときにコマンド行で fileB が最後にリストされたためです。)

シーケンス	オブジェクト	コメント
7	main.out	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。

第 6 章 アプリケーションの最適化

デフォルトでは、標準コンパイルでコードに関してごく基本的なローカルの最適化が実行されるのみですが、さらに、高速コンパイルと完全デバッグがサポートされています。コードをいったん開発、テスト、デバッグした後は、 XL C/C++ が提供する高度な最適化機能を利用することができます。この機能により、手動で再コーディングしなくても、パフォーマンスをかなり向上させることができます。実際、コードの手動最適化を過度に行うこと (例えばループの手動アンロール) はお勧めできません。構成を誤ると、コンパイラーが混乱し、新しいマシンに対するアプリケーションの最適化が難しくなるためです。

手動最適化の代わりに、コンパイラー・オプションのセットを使用することで、XL C/C++ コンパイラーの最適化を制御することができます。これらのオプションでは、次のような方法でコードを最適化できます。

- 特定タイプの最適化を実行するオプションには、次のようなものがあります。
  - システム・アーキテクチャー。アプリケーションが特定のハードウェア構成上で実行される場合、コンパイラーは、マイクロプロセッサー・アーキテクチャー、キャッシュまたはメモリー形状、アドレッシング・モデルも含めたターゲット・マシン用に最適化される命令を生成することができます。これらのオプションについては、 35 ページの『システム・アーキテクチャーの最適化』で説明します。
  - 共用メモリーの並列処理。アプリケーションが、共用メモリーの並列化をサポートするハードウェア上で実行される場合は、コンパイラーに、スレッド化されたコードの自動生成、あるいは OpenMP 標準プログラミング構文の認識を命令することができます。プログラム並列化のオプションについては、 38 ページの『共用メモリーの並列処理の使用』で説明します。
  - 高位のループ分析および変換。コンパイラーは、さまざまな手法を駆使してループを最適化します。これらのオプションについては、 37 ページの『高位ループ分析および変換の使用』で説明します。
  - プロシージャー間分析 (IPA)。コンパイラーは、コード・セクションを再編成して関数間の呼び出しを最適化します。 IPA オプションについては、 40 ページの『プロシージャー間分析の使用』で説明します。
  - プロファイル指示フィードバック (PDF)。コンパイラーは、呼び出し数とブロック数、および実行時間を基にして、コードのセクションを最適化することができます。 PDF オプションについては、 41 ページの『プロファイル指示フィードバックの使用』で説明します。
  - その他のタイプの最適化。ループのアンロール、関数のインライン化、スタック・ストレージの圧縮など、多数。これらのオプションについては、 44 ページの『その他の最適化オプション』に簡単な説明があります。
- 最適化レベル を使用できます。これは、いくつかの手法をバンドルしたもので、この中に、前述の特定の最適化オプションを 1 つ以上組み込むことができます。4 つの最適化レベルがあり、順に、コードに対してより積極的な最適化を実行するようになっています。最適化レベルについては、 32 ページの『最適化レベルの使用』で説明します。

- 最適化オプションと最適化レベルを結合すると、望みどおりの結果を得ることができます。上記で言及した各節では、その方法についても説明しています。

プログラムの最適化は一種のトレードオフであり、最適化の結果、コンパイル時間は長くなり、プログラム・サイズとディスク使用量は大きくなり、デバッグ機能は低下することに注意してください。最適化のレベルを高くすると、プログラム・セマンティクスが影響を受け、そのため、最適化の前までは正常に実行されていたコードが予想どおりに実行されなくなることがあります。つまり、すべてのアプリケーションにとって、あるいはアプリケーションのすべての部分にとって、最適化が無条件に望ましいわけではありません。計算的に密ではないプログラムの場合、最適化によって命令シーケンスを高速化することよりは、プログラムの容量を小さくしてページングやキャッシュのパフォーマンスを向上させることの方が、場合によっては重要です。

パフォーマンス向上によって恩恵が得られるコードのモジュールを確認するには、**-p** または **-pg** オプションを指定して、選択したファイルをコンパイルし、オペレーティング・システム・プロファイラー **gprof** を使用して、「ホット・スポット」であり、計算的に密である関数を識別します。サイズと速度がどちらも重要である場合は、ホット・スポットが含まれるモジュールを最適化し、それ以外のモジュールではコード・サイズを圧縮したままにしておきます。適切なバランスを検出するには、手法の組み合わせをいろいろ試してみる必要があります。

最適化で使用できるオプションをすべて網羅し、カテゴリー別に編成したリストが、45 ページの『最適化およびパフォーマンスに関するオプションの要約』に記載されています。

最後に、アプリケーションを手動で調整してコンパイラーが使用する最適化手法を補う場合は、47 ページの『第 7 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』に記載されているコーディングのパフォーマンスに対する提案と、その最良実例を参考にしてください。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-p**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-pg**

最適化レベルの使用

コンパイラーがデフォルトで実行するのは、ローカルでの簡単な最適化 (定数のフォールディング、ローカルでの共通部分式の除去など) のみですが、完全なデバッグもサポートされています。プログラムは、さまざまな最適化レベルを指定することにより最適化できますが、最適化によってアプリケーションのパフォーマンスが向上すると、逆にプログラム・サイズやデバッグ・サポートは大きくなります。次の表に、指定できるオプションをまとめてあります。また、それぞれの最適化レベルで使用される手法の詳細説明は、後述します。

表 8. 最適化レベル

オプション	振る舞い
-O、**-O2**、**-qoptimize**、または **-qoptimize=2**	低レベルの包括的最適化。部分的なデバッグ・サポート。
-O3 または **-qoptimize=3**	より広範囲にわたる最適化。精度とのトレードオフあり。
-O4 または **-qoptimize=4**	プロシージャー間の最適化。ループの最適化。自動マシン調整。
-O5 または **-qoptimize=5**	

最適化レベル 2 で使用される手法

最適化レベル 2 では、コンパイラーが適用する最適化手法は保守的であり、プログラムの正確さに影響を及ぼさないものとされます。最適化レベル 2 では、次の手法が使用されます。

- 後続の式で再計算される共通の部分式を除去します。例えば、次を指定した場合、

  ```
  a = c + d;
  f = c + d + e;
  ```

  共通の式 `c + d` は、最初の計算が行われた時点で保管され、次のステートメントで使用されて `f` の値を決定します。
- 代数式を単純化します。例えば、コンパイラーは、同じ式で使用される複数の定数を結合します。
- コンパイル時に定数を評価します。
- 次のような、未使用または冗長なコードは除去します。
  - 到達できないコード
  - 結果が後で使用されることのないコード
  - 値が後で使用されることのない保管命令
- プログラム・コードを再編成して、ロジックの分岐を最小限にし、物理的に個別のコード・ブロックを結合し、実行時間をできるだけ短くします。
- グラフ・カラーリング・アルゴリズムを使用して、変数と式を使用可能なハードウェア・レジスターに割り振ります。
- 効率の低い命令を、より効率的な命令に置き換えます。例えば、配列の添え字処理で、乗算命令を加算命令に置き換えます。
- 次のような、不変のコードをループの外に移動します。
  - ループ内で値が変化しない式。
  - ループ内で値が変化しない変数に基づく分岐コード。
  - 保管命令。
- いくつかのループをアンロールします (**-qunroll** コンパイラー・オプションの使用に相当)。
- いくつかのループをパイプラインにします。

最適化レベル 3 で使用される手法

最適化レベル 3 以上では、コンパイラーはよりアグレッシブになり、変更によって別の結果が生じるリスクを冒しても、プログラム・セマンティクスを変更してパフォーマンスの向上を図ります。次に、いくつか例を挙げます。

- 場合によっては、X*Y*Z を、(X*Y)*Z ではなく X*(Y*Z) として計算します。こうすると、丸めのために結果が異なる場合があります。
- 場合によっては、負のゼロ値の符号を省略します。こうすると、その値に無限大を掛けた場合に結果が異なる可能性があります。

35 ページの『最適化レベル 2 および 3 の最大活用』には、このリスクを少なくするための提案があります。

最適化レベル 3 では、最適化レベル 2 で使用されるすべての手法に加えて、次のような手法が使用されます。

- より深いループをアンロールし、ループ・スケジューリングを改善します。
- 最適化の範囲を拡張します。
- 効果の微小な最適化や、一部の条件下でのみ有用な最適化など、すべてのプログラムで役立つとは限らない最適化も実行します。
- コンパイル時間やスペースをかなり消費する最適化も実行します。
- 一部の浮動小数点計算の順序を変更します。これによって精度差が生じたり、または浮動小数点関連の例外の発生に影響を及ぼす可能性があります (**-qnostrict** オプションによるコンパイルに相当)。
- 暗黙のメモリー使用量制限を除去します (**-qmaxmem=-1** オプションによるコンパイルに相当)。
- 自動インライン化を拡張します。
- 構造体のコピーを介して定数および値を伝搬します。
- 可能であれば、他の最適化のあとで「address taken」属性を除去します。
- 隣接する集合体メンバーでのロード、保管、その他のオペレーションをグループ化します。その際、場合によっては、VMX ベクトル登録オペレーションを使用します。

最適化レベル 4 および 5 で使用される手法

最適化レベル 4 および 5 では、最適化レベル 2 および 3 で使用されるすべての手法に加えて、次のような手法が使用されます。

- プロシージャー間分析。リンク時に最適化プログラムを呼び出して、複数のソース・ファイル間で最適化を実行します (**-qipa** オプションによるコンパイルに相当)。
- 高位変換。ループ・ネストと配列言語構造体の最適化処理を行います (**-qhot** オプションによるコンパイルに相当)。
- ハードウェア固有の最適化 (**-qarch=auto**、**-qtune=auto**、および **-qcache=auto** オプションによるコンパイルに相当)。

- 最適化レベル 5 では、さらに詳細なプロシージャー間分析 (**-qipa=level=2** オプションによるコンパイルに相当)。レベル 2 IPA では、高位の変換 (**-qhot** によるコンパイルに相当) は、プログラム全体の情報が収集された後は、リンク時まで遅らせられます。

最適化レベル 2 および 3 の最大活用

ここでは、最適化レベル 2 および 3 を使用する際の、推奨される方法について説明します。

1. 可能であれば、まず最適化しないでコードをテストおよびデバッグしてから、**-O2** を使用します。
2. ご使用のコードが言語標準に準拠していることを確認します。
3. C C コードでは、ポインターの使用が次の制約事項に従っていることを確認してください。つまり、汎用ポインターは **char*** または **void*** でなければなりません。また、すべての共用変数および共用変数に対するポインターは、**volatile** でマークされている必要があります。
4. C C では、プログラムが独自の関数をライブラリー関数と同じ名前で定義しているのでない限り、 **-qlibansi** コンパイラー・オプションを使用します。
5. コードのできるだけ多くの部分を、**-O2** でコンパイルします。
6. **-O2** を使用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。
7. 次に、**-O3** をできるだけ多くのコードに対して使用します。
8. 問題が発生したり、パフォーマンスが低下したりする場合は、必要に応じて、**-O3** と一緒に **-qstrict** または **-qcompact** を使用することを検討してください。
9. **-O3** の使用時の問題が改善されない場合は、ファイルの一部に対しては **-O2** に切り替えますが、 **-qmaxmem=-1**、**-qnostrict** のいずれか、または両方を使用することを検討してください。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-O**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qnostrict**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qmaxmem**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qunroll**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qalias**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qlibansi**

システム・アーキテクチャーの最適化

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 9. 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 (4/4/4) 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 (4/8/8) 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、 PowerPC® アーキテクチャー向け命令のサブセットに対して生成される命令セットが制限されます。Y-HPC を除くすべての Linux プラットフォームで、デフォルトは **-qarch=ppc64grsq** です。Y-HPC でのデフォルトは **-qarch=ppc970** 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。
-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。デフォルトは **-qtune=pwr4** です。
-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。
-qenablevmx	コンパイラーに対し、任意のコンパイラー・フェーズで VMX (AltiVec) コードを生成するよう指示します。 SUSE 9、Y-HPC、および Red Hat 4 Linux では、このオプションはデフォルトで使用可能になっています。Red Hat 3 Linux では、デフォルトは **-qnoenablevmx** です。

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、『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コンパイラー・オプションを指定する』、および 『コンパイラー・モードおよびプロセッサーのアーキテクチャーの有効な組み合わせ』を参照してください。

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

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** オプションを使用して、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** を使用して、コードを十分に実行できる最小限のマシン・ファミリーを指定してください。

-qtune オプションの使用

-qarch を使用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** と **-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、『コンパイラー・モードおよびプロセッサーのアーキテクチャーの有効な組み合わせ』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** オプションを使用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。独自に **-qcache** サブオプションを使用することに決めた場合は、**-qhot** または **-qsmp** をそのサブオプションと併用します。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。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qarch**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qcache**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qtune**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qenablevmx**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qlistopt**

高位ループ分析および変換の使用

高位変換は、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。

高位ループ分析および変換を使用可能にするには、 **-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 10. **-qhot** *のサブオプション*

サブオプション	振る舞い
vector	コンパイラーに、いくつかのループを変換して、組み込みライブラリーにある各種の三角関数や演算 (逆数や平方根など) の、標準バージョンではなく最適化バージョンを使用するように指示します。最適化バージョンを使用すると、精度とパフォーマンスに関して、さまざまなトレードオフが発生します。このサブオプションは、 **-qhot**、**-O4**、または **-O5** を使用すると、デフォルトで使用可能になります。
novector	コンパイラーに、上記の組み込みライブラリー関数を使用する最適化を避けるように指示します。プログラム結果の精度を落としたくない場合は、このサブオプションまたは **-qstrict** を使用してください。
arraypad	コンパイラーに、メリットがあると思われる配列を、必要なだけ埋め込むように指示します。
simd	コンパイラーに、SIMD 自動ベクトル化を試みるように指示します。すなわち、配列の連続するエレメントに適用されるループ内の一定の演算を、VMX 命令呼び出しに変換します。この呼び出しは、いくつかの結果を一度に計算するため、個々の結果を連続して計算するよりは速くなります。このサブオプションは、Y-HPC ではデフォルトで使用可能になります。このサブオプションは、 **-qarch** を **ppc970** に設定すると、SUSE 9 および Red Hat 4 ではデフォルトで使用可能になります。このサブオプションは、 **-qarch** を **ppc970** に設定し、かつ **-qenablevmx** を使用すると、Red Hat 3 U3 で使用可能になります。

-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、 **-O2** および **-O3** と併用してみてください。このオプションは、変換を行う必要がない場合は、影響が起こらないように設計されています。
- **-qhot** を使用したことによってコンパイル時間が許容できないほど長くなったり (これは、複雑なネストされたループで起こることがあります)、性能の低下が見られたりする場合は、**-qhot=novector** を使用するか、あるいは **-qstrict** または **-qcompact** を、**-qhot** と併用してください。
- 必要に応じて、**-qhot** の非アクティブ化を選択して、コードの一部を改善できるようにします。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qhot**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qstrict**

共用メモリーの並列処理の使用

IBM pSeries™ のマシンの中には、共用メモリーの並列処理ができるものがあります。 **-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。このオプションは、少なくとも **-O2** 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*」に記載されています。

表 11. 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。 **-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、明示的並列処理を指定するための OpenMP 言語拡張に従うように指示します。 **-qsmp=omp** と **-qsmp=auto** は、現時点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。
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、ネストされた並列処理、ロックなどの制御に使用されます。

-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、必ず、再入可能コンパイラー呼び出し (*_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。
- 専用マシンまたはノードを使用している場合は、 **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。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qsmp**
- 「*XL C/C++ コンパイラー・リファレンス*」の『並列処理用のランタイム・オプション』
- 「*XL C/C++ コンパイラー・リファレンス*」の『並列処理用の OpenMP ランタイム・オプション』

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** で説明しています。

表 12. 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、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、および関数間のコードの移動。 • プロシージャーの特殊化 (クローン作成)。
inline=*variable*	関数のインライン化が正確に制御できるようになります。
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。

-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。

- **-qipa** オプションは、アプリケーション全体、あるいはそのできるだけ多くの部分の、コンパイルおよびリンク・ステップで指定してください。ライブラリー、共用オブジェクト、および実行可能プログラムに対しても **-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かなり大きなオブジェクト・ファイルを生成したり、 **/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たり、 **TMPDIR** 環境変数を使用して十分なフリー・スペースのある別のディレクトリーを指定したりすることができます。
- 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。
- インライン化された関数のレポートを生成するには、 **-qlist** または **-qipa=list** を使用します。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、**-qipa=inline** または **-qipa=noinline** の使用を検討してください。特定関数のインライン化を制御するには、**-Q+** または **-Q-** を使用します。

関連参照

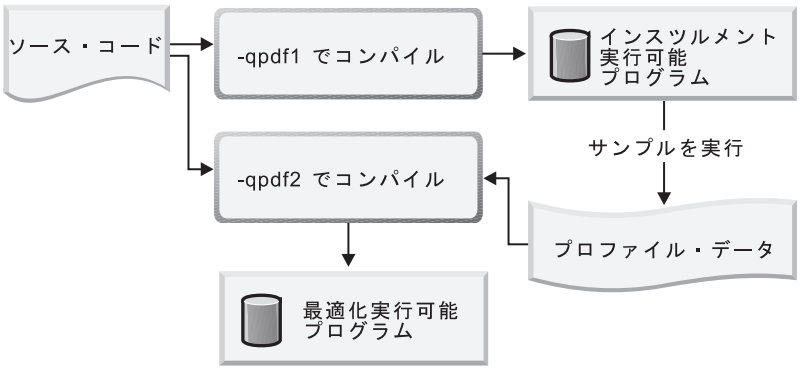
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qipa**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-Q**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qlist**

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。このプロセスは、アプリケーション全体を 2 度コンパイルする必要があるため、他のデバッグやチューニングが終了してから、アプリケーションを実稼働させる前の最後の段階の一部として使用されるようになっています。

次の図は、PDF プロセスを表しています。

図 2. プロファイル指示フィードバック



まず、**-qpdf1** オプションを指定して (最小最適化レベル **-0** を使用) プログラムをコンパイルします。これにより、コンパイル済みプログラムをユーザーが通常使用するのと同じ方法で使用して、プロファイル・データが生成されます。次に、**-qpdf2** オプションを使用して、プログラムをもう一度コンパイルします。これで、 **qipa=level=0** が呼び出され、プログラムはプロファイル・データに基づいて最適化されます。

アプリケーションのすべてのコードを **-qpdf1** オプションでコンパイルしなくても、 PDF プロセスの恩恵は受けられます。大規模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集中することもできます。

-qpdf オプションを使用するには、次のようにします。

1. アプリケーションの一部またはすべてのソース・ファイルを、**-qpdf1** および最小最適化レベル **-O** を指定してコンパイルする。
2. 標準的なデータ・セットを 1 つ以上使用して、アプリケーションを実行する。ここで重要なのは、そのアプリケーションで実際に使用されるデータを代表するようなデータを使用することです。アプリケーションの終了時には、現行作業ディレクトリーまたは **PDFDIR** 環境変数で指定したディレクトリー内の PDF ファイルに、プロファイル情報が書き込まれます。
3. **-qpdf2** を指定してアプリケーションをコンパイルする。

次のようにすれば、PDF ファイルをさらに制御することができます。

1. アプリケーションの一部またはすべてのソース・ファイルを、**-qpdf1** および最小最適化レベル **-O** を指定してコンパイルする。
2. 標準的なデータ・セットを 1 つ以上使用して、アプリケーションを実行する。これによって、現行ディレクトリーに PDF ファイルが作成されます。
3. **PDFDIR** 環境変数で指定したディレクトリーを変更して、別のディレクトリーに PDF ファイルを作成する。
4. **-qpdf1** を指定してアプリケーションを再コンパイルする。
5. ステップ 3 と 4 を必要なだけ繰り返す。
6. **mergepdf** ユーティリティーを使用して、 PDF ファイルを連結する。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
```

7. **-qpdf2** を指定してアプリケーションをコンパイルする。

関数呼び出しおよびブロック統計についてさらに詳しい情報を収集するには、次のようにします。

1. **-qpdf1 -qshowpdf -O** を指定して、アプリケーションをコンパイルする。
2. 標準的なデータ・セットを 1 つ以上使用して、アプリケーションを実行する。アプリケーションは、より詳細なプロファイル情報を PDF ファイルに書き込みます。
3. **showpdf** ユーティリティーを使用して、 PDF ファイル内の情報を表示する。

PDF ディレクトリー内の情報を消去するには、 **cleanpdf** ユーティリティーまたは **resetpdf** ユーティリティーを使用します。

pdf および showpdf によるコンパイルの例

次の例は、**showpdf** ユーティリティーで PDF を使用して、「Hello World」アプリケーションの呼び出しおよびブロック統計を表示する方法を示したものです。

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する。

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

2. その結果できる実行可能ファイル **a.out** を実行する。
3. **showpdf** ユーティリティーを実行して、その実行可能ファイルに対する呼び出し数およびブロック数を表示する。

   ```
   showpdf
   ```

結果は以下のようになります。

```
HelloWorld(4):  1 (hello.c)

Call Counters:
5 | 1  printf(6)

Call coverage = 100% ( 1/1 )

Block Counters:
3-5 | 1
6 |
6 | 1

Block coverage = 100% ( 2/2 )

-----------------------------------
main(5):  1 (hello.c)

Call Counters:
10 | 1  HelloWorld(4)
```

```
Call coverage = 100% ( 1/1 )

Block Counters:
8-11 | 1
11 |

Block coverage = 100% ( 1/1 )

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

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **-qpdf**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-showpdf**

その他の最適化オプション

以下のオプションは、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 13. 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
-qcompact	コード・サイズの肥大につながる最適化 (ループのアンロール、関数のインライン化など) を抑制します。
-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。
-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。そうすることにより、ヒープの使用量が増大する場合があります。
-qinline	名前付き関数のインライン化を制御します。コンパイル時、リンク時、またはその両方で使用できます。 **-qipa** の使用時には、**-qinline** と **-qipa=inline** は同義です。
-qunroll	ループのアンロールを独自に制御します。**-O3** では、暗黙のうちにアクティブになっています。
-qinlglue	リンカーによって生成され、外部関数の呼び出しまたは関数ポインターを介した呼び出しに使用される「グルー・コード」をインライン化するよう、コンパイラーに命令します。 64 ビット・モード専用です。
-qtbtable	トレースバック・テーブル情報の生成を制御します。64 ビット・モード専用です。
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。

表 13. 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
-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。

最適化およびパフォーマンスに関するオプションの要約

次の表は、最適化およびパフォーマンス調整を扱うコンパイラー・オプションを要約したものです。オプションは、タイプ別にグループ化されています。各オプションの説明、完全な構文、および使用法については、「*XL C/C++ コンパイラー・リファレンス*」の、該当するオプションの見出しを参照してください。

表 14. 最適化およびパフォーマンス調整に関するオプション

最適化フラグ	最適化制限オプション
-O/-qoptimize **-qagrrcopy**	**-qkeepparm** **-qnoprefetch** **-qstrict** **-qcompact** **-qmaxmem**
関数のインライン化	**コード・サイズの削減**
-Q **-qinline** **-qinlglue**	**-s** **-qnoeh**
副次作用	**ループの最適化**
-qignerrno **-qisolated_call**	**-qhot** **-qreport** **-qnostrict_induction** **-qunroll**
プログラム全体の分析	**プロセッサーおよびアーキテクチャーの最適化**
-qipa	**-qarch** **-qcache** **-qtune** **-qdirectstorage** **-qenablevmx**
パフォーマンス・データ収集	**その他の最適化オプション**
-p **-qpdf1** **-qpdf2** **-pg** **-qshowpdf**	**-qtocdata** **-qsmallstack** **-qspill**

第 7 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

31 ページの『第 6 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++ 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索』
- 48 ページの『関数呼び出しによるオーバーヘッドの削減』
- 49 ページの『効率的なメモリーの管理』
- 50 ページの『変数の最適化』
- 51 ページの『効率的なストリングの操作』
- 51 ページの『式とプログラム・ロジックの最適化』
- 52 ページの『64 ビット・モードでの演算の最適化』

高速入出力手法の検索

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。

- テキスト・ストリームの代わりにバイナリー・ストリームを使用する。バイナリー・ストリームでは、入力または出力時にデータは変更されません。
- **open** および **close** のような、低水準の入出力関数を使用します。これらの関数は、**fopen** および **fclose** のようなストリーム入出力関数と比べて、より高速で、よりアプリケーションに特定です。低レベルの関数に対してユーザー独自のバッファリングを提供しなければなりません。
- 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サイズである 4K の倍数にする。
- 入力を読み取るときは、一度に 1 つの文字ではなく、行全体を同時に読み取る。
- ファイル全体を処理する必要があることが分かっている場合は、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。
- **scanf** および **fscanf** の代わりに、**fgets** を使用して文字列内を読み取り、それから **atoi**、**atol**、**atof**、または **_atold** のいずれかを使用してそれを適切な形式に変換します。
- 複雑なフォーマット設定にのみ **sprintf** を使用する。ストリングの連結のようなより単純なフォーマット設定では、より特定なストリング関数を使用します。

関数呼び出しによるオーバーヘッドの削減

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。
- 関数にグローバル変数から値を取らせるのではなく、関数に引き数として値を渡す。
- 可能であれば、インライン化された関数で定数の引き数を使用する。定数の引き数を持つ関数によって、最適化の機会が広がります。
- **#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。
- 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、関数の呼び出しを高速にできます。
- C++ 通常は、仮想関数をインラインで宣言しない。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 **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き数の関数の使用を避ける。
- 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。
- 関数仮パラメーターとして値の構造体または共用体を渡すこと、あるいは構造体または共用体を戻すことを避ける。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。
- 可能であれば、**int** および **short** のような非集合体の型は、参照によって渡すのではなく、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++ ユーザー関数は、XL C/C++ ヘッダー・ファイルを組み込むと、組み込み関数に自動的にマップされます。

C ユーザー関数は、math.h および string.h を組み込むと、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、インラインの候補としては適しません。
- 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。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma isolated_call**
- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma disjoint**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qipa**

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メンバーから順に宣言する。
- 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、メモリーの使用法やオブジェクトの再利用を管理します。

- 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。
- 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。
- 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。

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、外部関数も呼び出す場合には、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にも影響を受けないこと、および混在する関数呼び出しでこの変数が複数回にわたって読み取られることが分かっている場合には、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 static 変数を使用してください。複数の関連する関数と static 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。
- **#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 static 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る。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は、一時変数の再利用は避けてください。ローカル変数のアドレスをとると、別の状況では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)。

- 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。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **#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。たとえば、**memcpy** が **strcpy** より高速なのは、文字列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、短精度の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。こうすれば、ある種の最適化手法が改善され、同じストリングを複数回使用する場合に、メモリーの使用量が削減されます。ストリングを明示的に読み取り専用に設定することができます。ストリングを読み取り専用に設定するには、ソース・ファイルで **#pragma strings (読み取り専用)** を使用するか、ソース・ファイルの変更を避ける場合は、 **-qro** (デフォルトで使用可能になっています) を使用します。

関連参照

- 「*XL C/C++ コンパイラー・リファレンス*」の **#pragma strings (読み取り専用)**
- 「*XL C/C++ コンパイラー・リファレンス*」の **-qro**

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ほかの式で使用されている場合には、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。
- 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。
- 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、いくつかのアプリケーションは、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良いパフォーマンスを示します。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、物理メモリーの負荷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。
- 64 ビット・プログラムで、配列指標に 32 ビットの符号付き整数を使用する場合は、配列を参照するたびに、符号拡張を行うための追加の命令が必要となる場合がある。

64 ビット・プログラムのパフォーマンス上のマイナス影響を補正するには、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 ビット型の方を符号拡張し、レジスターの上位 32 ビットをクリアする必要があります。このため、計算が遅くなります。

- 可能な限り、long 型の除法を使用しない。乗算は、多くの場合、除法よりも高速です。同じ除数で多くの除法を実行する必要がある場合は、除数の逆数を一時変数に割り当て、すべての除法を一時変数に対する乗算に変更します。例えば、次の関数を考えてみます。

  ```
  double preTax(double total)
  {
  return total * (1.0 / 1.0825);
  }
  ```

  この方が、次の直接除法よりも実行が速くなります。

  ```
  double preTax(double total)
  {
  return total / 1.0825;
  }
  ```

  その理由は、除法 `(1.0 / 1.0825)` は、コンパイル時にのみ評価され、フォールディングされるためです。
- 頻繁に使用される変数 (ループ・カウンター、配列指標など) には、 **signed**、**unsigned**、および簡潔な **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。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
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。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。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。

プログラミング・インターフェース情報

プログラミング・インターフェース情報は、プログラムを使用してアプリケーション・ソフトウェアを作成する際に役立ちます。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

注: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

以下は、IBM Corporation の商標です。

- AIX
- IBM
- IBM (ロゴ)
- PowerPC
- pSeries

Linux は、Linus Torvalds の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

業界標準

次の規格がサポートされます。

- C 言語は、International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)) に準拠しています。
- C++ 言語は、International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998) に準拠しています。
- C++ 言語は、International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)) にも準拠しています。
- C 言語および C++ 言語は、 OpenMP C and C++ Application Programming Interface Version 2.0 に準拠しています。

プログラム番号: 5724-K77

Printed in Japan

SC88-9976-01

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12