IBM XL C/C++ Advanced Edition for Linux, V9.0

プログラミング・ガイド

SC88-4646-00
(英文原典：SC23-5890-00)

IBM XL C/C++ Advanced Edition for Linux, V9.0

プログラミング・ガイド

SC88-4646-00
(英文原典：SC23-5890-00)

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、 95 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++ Advanced Edition for Linux, V9.0 (プログラム番号 5724-S73) 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。必ず、製品のレベルに合った正しい版を使用してください。

IBM 発行のマニュアルに関する情報のページ

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

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

(URL は、変更になる場合があります)

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

原　典： SC23–5890–00
IBM XL C/C++ Advanced Edition for Linux, V9.0
Programming Guide

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 2007.6

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

このガイドは、IBM® XL C/C++ Advanced Edition for Linux®, V9.0 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。

本書の使い方

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++ Advanced Edition for Linux, V9.0 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検討しています。
- 11 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。
- 19 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。
- 23 ページの『第 5 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。
- 29 ページの『第 6 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。

- 37 ページの『第 7 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。
- 65 ページの『第 9 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。
- 73 ページの『第 10 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。
- 87 ページの『第 11 章 プログラムの並列処理』 は、IBM XL C/C++ Advanced Edition for Linux, V9.0 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。

本書で使用する規則

活字規則

下記の表では本書で使用する活字規則を説明します。

表 1. 活字規則

字体	指示	例
太字	小文字のコマンド、実行可能プログラム名、コンパイラー・オプションおよびディレクティブ。	**-O3** を指定すると、コンパイラーは **-qhot=level=0** を前提とします。 **-O3** のすべての HOT 最適化を抑制するには、**-qnohot** を指定する必要があります。
イタリック	実際の名前や値がユーザーによって供給される、パラメーターまたは変数を識別します。イタリック体は新規用語を紹介するためにも使用されます。	要求を超えるサイズ を戻す場合は、*size* パラメーターを更新する必要があります。
モノスペース	プログラミング・キーワードとライブラリー関数、コンパイラー組み込み機能、プログラム・コードの例、コマンド・ストリング、またはユーザー定義名。	`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します。

アイコン

本書内に記載されるすべての機能は、C および C++ 言語に適用されます。ある機能が 1 つの言語に専用の場合、または機能性が言語によって異なる場合は、以下のアイコンが使用されます。

C

テキストは C 言語のみによってサポートされるフィーチャーを説明します。または C 言語のみに特定な動作を説明します。

C++

テキストは C++ 言語のみによってサポートされるフィーチャーを説明します。または C++ 言語のみに特定な動作を説明します。

構文図

本書では、XL C/C++ の構文を構文図で示しています。このセクションでは、構文図の見方と使い方について説明します。

- 構文図は、左から右、上から下に、線のパスに従って読んでください。

  ▸▸── 記号は、コマンド、ディレクティブ、または文の開始を示します。

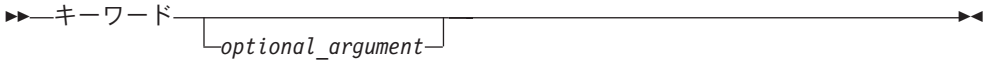
  ──▸ 記号は、コマンド、ディレクティブ、または文構文が次の行に続いていることを示します。

  ▸── 記号は、コマンド、ディレクティブ、または文構文が前のライン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、または文の終了を示します。

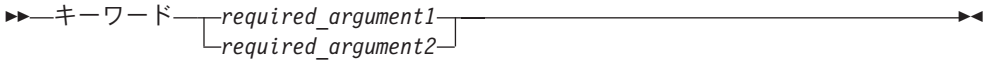
  完全なコマンド、ディレクティブ、または文以外の構文単位の図である断片は、|── 記号で開始して、──| 記号で終了します。
- 必須項目は水平線 (メインパス) 上に示されています。

- オプション項目は、メインパスの下側に表示されます。

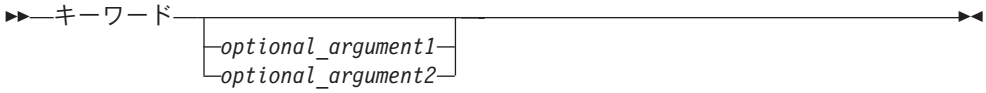


- 2 項目以上から選択できる場合は、縦に積み重ねて表示されます。

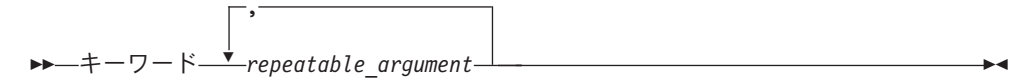
  1 項目の選択が必要 な場合は、スタックの 1 項目がメインパスに表示されます。



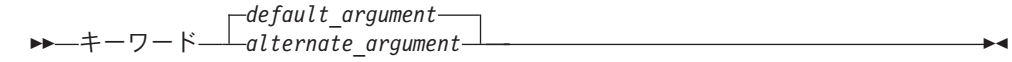
  項目の 1 つの選択がオプショナルの場合は、全体のスタックがメインパスの下側に表示されます。



- メインラインの左上に戻る矢印 (繰り返し矢印) は、スタックされた項目から複数を選択できるか、同じ項目を繰り返し選択できることを示します。区切り文字も、ブランク以外であれば示されています。



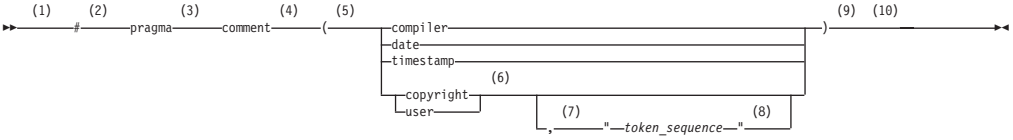
- デフォルトである項目はメインパスの上側に表示されます。



- キーワードは非イタリック体で表示され、表示されたとおり正確に入力されなければなりません。
- 変数はイタリック体を使用した小文字で表示されます。変数はユーザーが指定する名前または値を表します。
- 句読記号、括弧、算術演算子、または他の示されている記号は、構文の一部としてユーザーが入力する必要があります。

構文図の例

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



注:

1 これは構文図の開始です。

2 記号 # は先頭になければなりません。

3 キーワード pragma は # 記号の後になければなりません。

4 キーワード pragma の後にはプラグマの名前 comment が必要です。

5 左括弧が必要です。

6 コメント・タイプは、示されたタイプの中の 1 つ、つまり compiler、date、timestamp、copyright、または user としてのみ入力できます。

7 コメント・タイプ copyright または user とオプションの文字ストリングの間にはコンマが必要です。

8 コンマの後には文字ストリングが必要です。文字ストリングは二重引用符で囲まれなければなりません。

9 右括弧が必要です。

10 これは構文図の終了です。

下記の **#pragma comment** ディレクティブの例は、上記の図に従えば構文的には正しい表示です。

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

例

この文書の例は、特に注記がない限り単純なスタイルでコード化されていて、ストレージを節約し、エラーのチェック、高速なパフォーマンス、または特定の結果を達成するためにすべての可能性のあるメソッドを実証するようなことは行いません。

関連情報

以下のセクションでは、XL C/C++ 関連の資料に関する情報を提供します。

- 『IBM XL C/C++ の資料』
- x ページの『標準と仕様に関する資料』
- xi ページの『その他の IBM 資料』
- xi ページの『その他の資料』

IBM XL C/C++ の資料

XL C/C++ には、下記の形式の製品資料があります。

- README ファイル

  README ファイルには、製品資料についての変更や訂正を含む最新情報が入っています。 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・ディレクトリーに入っています。

- インストール可能マニュアル・ページ

  コンパイラー呼び出しおよびすべてのコマンド行ユーティリティーについては、マニュアル・ページが製品に付属して提供されています。マニュアル・ページのインストールおよびアクセスの説明については、「*XL C/C++ Installation Guide*」を参照してください。

- インフォメーション・センター

  検索可能な HTML ファイルのあるインフォメーション・センターをネットワークに起動して、リモートまたはローカル側からアクセスできます。オンライン・インフォメーション・センターのインストールおよびアクセスの説明については、「*XL C/C++ Installation Guide*」を参照してください。インフォメーション・センターは、次の Web サイトからも表示できます。http://publib.boulder.ibm.com/infocenter/lnxphelp/v9v111/index.jsp

- PDF 文書

  PDF 文書は、デフォルトでは /opt/ibmcmp/vacpp/9.0/doc/*LANG*/pdf/ ディレクトリーに入っています。ここで、*LANG* は en_US、zh_CN、 または ja_JP です。PDF ファイルは Web からも入手できます (http://www.ibm.com/software/awdtools/xlcpp/library)。

XL C/C++ 製品マニュアルは、すべて下記のファイルに収められています。

表 2. 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++ Advanced Edition for Linux, V9.0 インストール・ガイド, GC88-4647-00	install.pdf	XL C/C++ のインストールおよび基本的なコンパイルとプログラム実行に必要な環境の構成に関する情報が含まれています。
IBM XL C/C++ Advanced Edition for Linux, V9.0 はじめに, GC88-4645-00	getstart.pdf	XL C/C++ 製品の入門が用意されていて、お客様の環境の設定と構成に関する情報、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のための情報など含まれています。
IBM XL C/C++ Advanced Edition for Linux, V9.0 Compiler Reference, SC23-5889-00	compiler.pdf	各種のコンパイラー・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 (並列処理用のものも含む) に関する情報が含まれています。
IBM XL C/C++ Advanced Edition for Linux, V9.0 Language Reference, SC23-5892-00	langref.pdf	IBM がサポートする C および C++ プログラミング言語 (所有権のない標準に対する移植性と適合性のための言語拡張を含む) に関する情報が含まれています。
IBM XL C/C++ Advanced Edition for Linux, V9.0 プログラミング・ガイド, SC88-4646-00	proguide.pdf	アプリケーションの移植、Fortran コードを使った言語間呼び出し、ライブラリー開発、アプリケーションの最適化と並列化、および XL C/C++ ハイパフォーマンス・ライブラリーといった、高度なプログラミング・トピックに関する情報が含まれています。

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、(ライセンス条項に従うことにより) Adobe Web サイト (http://www.adobe.com) からダウンロードできます。

Redbook、ホワイト・ペーパー、チュートリアルその他の文献を含む XL C/C++ 関連の豊富な資料を下記の Web サイトから入手できます。

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

標準と仕様に関する資料

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。本書に記載されているフィーチャーの一部の厳密な定義については、これらの標準を参照してください。

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*、*C89* ともいいます。
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*、*C99* ともいいます。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*、*C++98* ともいいます。

- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003(E)*、*Standard C++* ともいいます。
- *Information Technology - Programming languages - Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. このドラフトの技術レポートは、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いて、http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1836.pdf で入手可能です。
- *AltiVec Technology Programming Interface Manual*、Motorola Inc. このベクトル・データ型の仕様は、ベクトル処理テクノロジーをサポートするものであり、http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf で入手可能です。
- *OpenMP Application Program Interface Version 2.5*、http://www.openmp.org で入手可能です。

その他の IBM 資料

- *ESSL for Linux on POWER V4.2 Guide and Reference*, SA22-7904、http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp で入手可能です。

その他の資料

- *Using the GNU Compiler Collection*、http://gcc.gnu.org/onlinedocs で入手可能です。

テクニカル・サポート

追加のテクニカル・サポートは、XL C/C++ サポート・ページ (http://www.ibm.com/software/awdtools/xlcpp/support) から入手できます。このページでは、さまざまなテクニカル・サポート FAQ その他のサポート文書に対する検索機能を備えたポータルを提供します。

もし、見つからない場合は、compinfo@ca.ibm.com 宛に電子メールをご送付ください。

XL C/C++ に関する最新の情報については、製品情報サイト (http://www.ibm.com/software/awdtools/xlcpp)をご覧ください。

第 1 章 32 ビットおよび 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。そのためには、コンパイル中に **-q32** (デフォルト) または **-q64** を各々指定します。

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++ 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 3. 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、70 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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-q32/-q64** および **-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 4. 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。

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。

- 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 5. 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) または UL (unsigned 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。

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 6. 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、以下のコード構成は有効です。

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

calloc に対する関数プロトタイプがなくても、同じコードが 64 ビット・モードでコンパイルされるとき、コンパイラーは関数が int を戻すと想定し、したがって a が無言で切り捨てられ、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、結果の型キャスト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、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。

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、 11 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 7. C/C++ と Fortran の同等のデータ型

C/C++ type	**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』
- 『対応するデータ型』
- 8 ページの『文字および集合データ』
- 9 ページの『関数呼び出し、およびパラメーター引き渡し』
- 9 ページの『ポインターから関数へ』
- 9 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。

関連情報

- 5 ページの『Fortran コードの呼び出し』

ID

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 ID 内の有効文字の最大数は 250 です。[1]

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 8. 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
bool (C++)_Bool (C)	LOGICAL(1)
char	CHARACTER
signed char	INTEGER*1
unsigned char	LOGICAL*1

1. Fortran 90 および 95 言語標準には、31 文字を超えない ID が必要です。 Fortran 2003 標準には、63 文字を超えない ID が必要です。

表 8. 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
signed short int	INTEGER*2
unsigned short int	LOGICAL*2
signed long int	INTEGER*4
unsigned long int	LOGICAL*4
signed long long int	INTEGER*8
unsigned long long int	LOGICAL*8
float	REAL REAL*4
double	REAL*8 DOUBLE PRECISION
long double	REAL*8 DOUBLE PRECISION
float _Complex	COMPLEX*8 または COMPLEX(4)
double _Complex	COMPLEX*16 または COMPLEX(8)
long double _Complex	COMPLEX*16 または COMPLEX(8)
構造体または共用体	派生型
enumeration	INTEGER*4
char[n]	CHARACTER*n
タイプへの配列ポインター、または type []	次元つき変数 (転置)
関数へのポインター	関数パラメーター
構造 (**-qalign=packed** つき)	シーケンス派生タイプ

- 「*XL C/C++ Compiler Reference*」の **-qldbl128**
- 「*XL C/C++ Compiler Reference*」の **-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。
- 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 9 には、「C 内の A 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 9. 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 の両方に対して同一に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。詳細については「*XL Fortran Language Reference*」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。

ポインターから関数へ

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。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。

```
#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);
}
```

以下は Fortran サブルーチンです。

```
C Fortran function add.f - for C/C++ interlanguage call example

C Compile separately, then link to C/C++ program

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
```

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、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。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。つまり、ベクトルは 16 バイト境界に従って位置合わせする必要があります。詳しくは、「*AltiVec Technology Programming Interface Manual*」を参照してください。

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可能となります。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可能になります。

『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。

15 ページの『位置合わせ修飾子の使用』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。

関連情報

- 「*AltiVec Technology Programming Interface Manual*」、http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手可能
- 「*XL C/C++ Compiler Reference*」の **-qaltivec**

位置合わせモードの使用

XL C/C++ でサポートされる各データ型は、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位置合わせされます。Linux は、デフォルト位置合わせモードは **linuxppc** です。

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ためには、コンパイル時に、表 10 内にリストされているいずれか 1 つのサブオプションを持つ **-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ためには、ソース・ファイル内で、表 10 にリストされているサブオプションの 1 つを持つ **#pragma align** 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 10 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 10. 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。

『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。 『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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qalign** および **#pragma align**

集合体の位置合わせ

12 ページの表 10 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。さらに追加して、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に対して全体として (修飾子が何も存在しない時) 適用されます。

- 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。
- 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。

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` のデータ型として宣言することができます。 ビット・フィールドの位置合わせは、その基本タイプとコンパイル・モード (32 ビットまたは 64 ビット) に依存します。

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` 演算子は常に基本タイプのサイズを戻します。

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、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

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

位置合わせ修飾子の使用

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、宣言のレベル、または個々の変数定義のレベルの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。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。[2]

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))` として指定可能です。これは最大のシステム位置合わせを指定したことと同一です (すべての UNIX® プラットフォームで 16 バイト)。

__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、共用体、またはクラスに適用されます。 [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。またこの変数に使用されているストレージ容量を効果的に増大させます。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++ ではユーザーが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++ Language Reference*」の『型属性』を参照してください。

位置合わせの修飾子を使用する際には、修飾子とモード間の相互作用、および複数の修飾子間の相互作用が複雑になる場合があります。後続のセクションでは、下記の変数の型について位置合わせ修飾子の優先順位ガイドラインの概要を説明しています。

- 集合体 (構造体、共用体、またはクラス) のメンバー、および `typedef` ステートメントで作成されたユーザー定義の型などを含む、単純変数またはスカラー変数。
- 集合体変数 (構造体、共用体、またはクラス)

関連情報

- 「*XL C/C++ Language Reference*」の 『位置合わせ変数属性』、『パック変数属性』、 『位置合わせ型属性』、『パック型属性』、および 『__align 指定子』
- 「*XL C/C++ Compiler Reference*」の **#pragma pack** および **-qpack_semantic**

スカラー変数の位置合わせを決定するガイドライン

以下の公式では、*non-embedded* (スタンドアロン) スカラー変数および *embedded* スカラー (集合体のメンバーとして宣言されている変数) の両方の位置合わせ修飾子の存在を前提として、「トップダウン・アプローチ」で「位置合わせ」について決定します。

変数の位置合わせ = maximum(*有効な型の位置合わせ、変更された位置合わせ値*)

ここで、*有効な型の位置合わせ* = maximum(maximum(位置合わせした型属性値、`__align` 指定子の値)、minimum(*型の位置合わせ*、`packed` 型属性値))

そして *変更された位置合わせ値* = maximum(位置合わせした 変数の属性値、`packed` 変数の属性値)

型の位置合わせは、変数が宣言されたとき、または位置合わせ値が `typedef` 文内の型に適用された場合における、現在有効な *型の位置合わせ* モードです。

さらに追加して、組み込み変数について、**#pragma pack** ディレクティブによって変更されることがある場合は以下の規則が適用されます。

変数の位置合わせ = minimum(**#pragma pack** 値、maximum(*有効な型の位置合わせ、変更された位置合わせ値*))

注: もし、同じ種類の型属性および変数属性の両方が 1 つの宣言に指定された場合、2 番目の属性は無視されます。

集合体変数の位置合わせを決定するガイドライン

以下の公式は集合体変数、すなわち構造体、共用体、およびクラスの位置合わせについて決定します。

変数の位置合わせ = maximum(*有効な型の位置合わせ、変更された位置合わせ値*)

ここで、*有効な型の位置合わせ* = maximum(maximum(位置合わせした 型属性値、`__align` 指定子の値)、minimum(*集合体型の位置合わせ*、`packed` 型属性値))

そして *変更された位置合わせ値* = maximum(位置合わせした 変数の属性値、`packed` 変数の属性値)

そして *集合体型位置合わせ* = maximum (すべてのメンバーの位置合わせ)

注: もし、同じ種類の型属性および変数属性の両方が 1 つの宣言に指定された場合、2 番目の属性は無視されます。

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』
- 20 ページの『厳密な IEEE 準拠のためのコンパイル』
- 20 ページの『浮動小数点定数の折り畳みと丸めの処理』
- 22 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。

- 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 32 ビット単精度の浮動小数点数
- 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 64 ビット倍精度の浮動小数点数
- 倍精度値と同じ範囲であるが、10 進法で約 29 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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qldbl128**

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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qfloat**

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するには以下の方法を行います。

- **-qfloat=nomaf** コンパイラー・オプションを使用します。
- プログラムで実行時に丸めモードを変更する場合には、**-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、**-qfloat=nans** オプションを使用します。 (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。)
- **-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。

関連情報

- 41 ページの『拡張最適化』
- 「*XL C/C++ Compiler Reference*」の **-qfloat**
- 「*XL C/C++ Compiler Reference*」の **-qinitauto**
- 「*XL C/C++ Compiler Reference*」の **-qstrict**
- 「*XL C/C++ Compiler Reference*」の **-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 21 ページの『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。

- 操作は無限または NaN 結果になります。たとえ、**-qflttrap** オプションでコンパイルしても、コンパイル時折り畳みは例外の実行時検出を防いでしまいます。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

- 22 ページの『浮動小数点例外の処理』
- 「*XL C/C++ Compiler Reference*」の **-qfloat** および **-qstrict**

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、最も近い値への丸めです。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「最も近い値へ丸め」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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qfloat** および **-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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qfloat** および **-qflttrap**

第 5 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、あまり使用されません。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、 24 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、 27 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、同時には使用できません。

関連情報

- -qtmplinst

-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。

1. 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。
2. 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。
3. 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。
4. (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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qtempinc** および **#pragma implementation**

-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、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

-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、「先着順サービス」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インスタンス生成されます。
- 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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qtemplateregistry** および **-qtemplaterecompile**

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。

-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。 25 ページの『-qtempinc の例』の図を参照してください。

第 6 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。

30 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

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。例えば次のようになります。

   ```
   xlc -c bar.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。 例えば次のようになります。

   ```
   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。共用ライブラリーをコンパイルする場合は、**-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。例えば次のようになります。

   ```
   xlc -qpic -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。例えば次のようになります。

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

関連情報

- 「*XL C/C++ Compiler Reference*」の **-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。例えば次のようになります。

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

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。例えば次のようになります。

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

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qmkshrobj**、**-l**、**-R**、および **-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、 32 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やファイルに優先順位番号を割り当てることができ、オブジェクトは優先順位の順序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。

関連情報

- 「*XL C/C++ Language Reference*」の『init_priority 変数属性』

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。 32 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 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、このファイル内のオブジェクトが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。
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)。

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、 30 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。

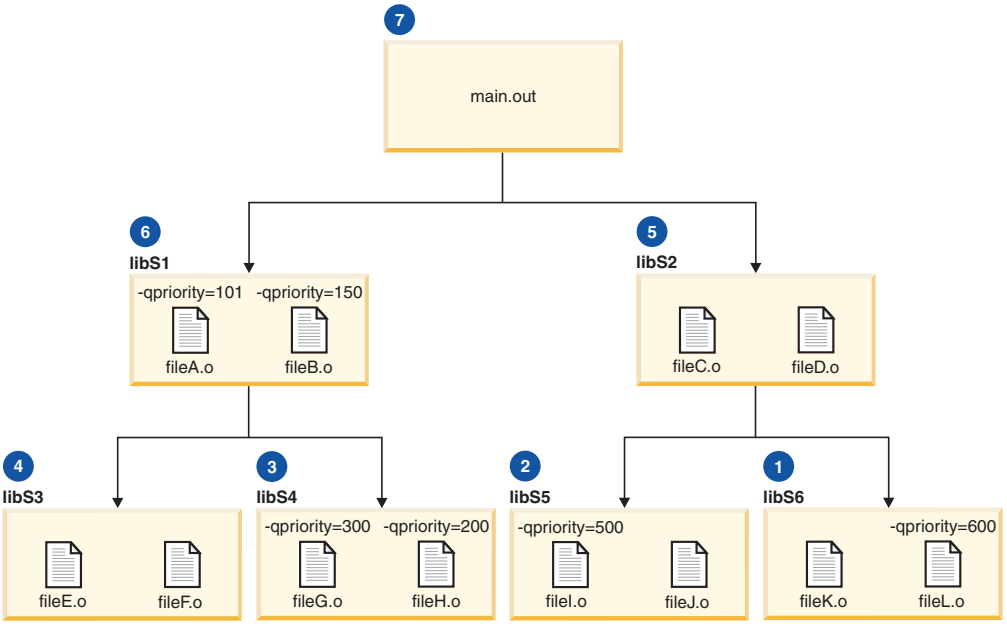
```
xlC -qmkshrobj -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
xlC -qmkshrobj -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

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 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	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 を作成するためにリンクされたときコマンド行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。
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))。

シーケンス	オブジェクト	優先順位の値	コメント
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行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。
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行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。
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行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。

第 7 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する広範囲なパフォーマンス強化手法を提供することで、Linux オペレーティング・システム向けのハイパフォーマンス 32 ビットおよび 64 ビット・アプリケーションの開発を可能にします。このようなパフォーマンスの優位性を実現するには、優れたプログラミング手法、徹底したテストとデバッグ (後に最適化が続く)、および調整が必要です。

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。
- PowerPC アーキテクチャーを最適に活用するようにオブジェクト・コードを再構成する。
- 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。

すべての最適化がすべてのアプリケーションに利益をもたらすわけではありませんが、たとえ基本的な最適化手法でもパフォーマンスの向上につながることがあります。アプリケーションのパフォーマンスの向上に役立つ一般的な手順については、『最適化プロセスのステップ』を参照してください。

調整

最適化はサポートされている環境でアプリケーションのパフォーマンスの向上を目指した積極的な変換を徐々に適用していくことですが、調整は、パフォーマンスを向上させたり、特定の実行環境をターゲットとするようにアプリケーションの特性を順応させる機会を提供することです。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。
- 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。

説明については、『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』ステップは、アプリケーションをレベル 3 から 5 のより強力な最適化に向かわせます。
3. 『高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瞬時に最適化できます。
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. 『ハイパフォーマンス・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ます。各オプション・レベルは徐々に積極的になる変換によって下位レベルの上に構成され、その結果より多くのマシン・リソースを使用するようになります。より積極的な最適化を試みる前に、アプリケーションが低い最適化レベルで正しくコンパイルおよび実行されるようにしてください。このセクションでは、*基本最適化*の表で補足オプションと一緒にリストされている 2 つの最適化レベルについて説明します。この表には、一部のアプリケーションの場合にその最適化レベルでパフォーマンス上の利点があるコンパイラー・オプションの列も示されています。

表 11. 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	利点をもたらす可能性のあるその他のオプション
-O0	なし	**-qarch**	**-g**
-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。 SMP プログラムの場合は、**-O0** に最も近いのは **-qsmp=noopt** です。このレベルでは、明らかに重複するコードを除去することによって基本分析最適化を実行するため、コンパイル時間を短縮できますが、より複雑な最適化へ進むことができるようにコードのアルゴリズムが正しいことを保証する必要があります。 **-O0** には定数の折り畳みも含まれます。オプション **-qfloat=nofold** を使用すると、浮動小数点折り畳み操作を抑制できます。このレベルでの最適化では、すべてのデバッグ情報が正確に保持されるため、未初期化の変数や間違ったキャストといった既存のコードの問題を明らかにできます。

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。

調整の詳細については、『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、オプティマイザー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、オプティマイザーが **-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

02 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、オプティマイザーは有効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。つまり、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、『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
- インバリアント・コードをループから移動します。
- 制御フローを簡素化します。
- アドレス・モードの強度縮小および有効利用。

-O2 最適化の場合でも、**-g** を指定すれば、デバッガーはソース・コードに関して何らかの役に立つ情報を取得できます。逆に言えば、より高い最適化レベルでは、デバッグ情報がもはや正確でない程度にコードが変換される可能性があります。そのような情報は慎重に使用してください。

拡張最適化

基本最適化を適用して、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、さらに強力な最適化ツールを適用できます。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。 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。 41 ページの表 12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最適な解決策を見つけるようオプティマイザーに指示するのに役立つ情報も含まれています。

表 12. 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	利点をもたらす可能性のあるオプション
-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**

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
- 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

-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、オプティマイザーは、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。オプティマイザー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、オプティマイザー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。

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。 **-qstrict** コンパイラー・オプションはまた、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。アプリケーションが浮動小数点例外や、浮動小数点演算の評価順序に依存する場合は、**-qstrict** でのコンパイルが正確な結果の確保に役立ちます。 **-qstrict** を指定しなければ、特定のソース・レベル演算に関する計算の差は、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。

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 の詳細については、『高位ループ分析および変換の使用』を参照してください。

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パラメーター値の伝搬
- 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。

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、オプティマイザー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、『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します。特に、ループ演算の場合がそうです。 データ・キャッシュに収まるだけのデータ量を処理するようにループ演算をブロックします。

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

レベル 4 における潜在的なトレードオフ

-O3 ですでに述べたトレードオフのほかに、**-qipa** を指定するとコンパイル時間が (特にリンク・ステップで) 大幅に増えることがあります。

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、オプティマイザー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、XL 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 13. 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、45 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。

表 13. ターゲット・マシンのオプション (続き)

オプション	振る舞い
-qipa=clonearch	ユーザーは、そのために命令セットを生成する、複数の特定のプロセッサー・アーキテクチャーを指定することができます。実行時には、アプリケーションはオペレーティング環境の特定のアーキテクチャーを検出し、そのアーキテクチャーのために専門化した命令セットを選択します。このオプションの優位性は、各ターゲット・アーキテクチャーごとにコードを再コンパイルする必要なしに、いくつかのアーキテクチャーの最適化が可能になる点です。このオプションの詳しい情報については、49 ページの『プロシージャー間分析の使用』を参照してください。
-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、『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、『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、『アーキテクチャー固有の 32 または 64 ビットでコンパイルするコンパイラー・オプションの指定』、および 「*XL C/C++ Compiler Reference*」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。

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** を使用して、コードを合理的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。または、複数アーキテクチャー用の命令を生成する **-qipa=clonearch** オプションを使用してください。 **-qipa=clonearch** を使用する場合、**-qarch** の値は **clonearch** サブオプションで指定されたアーキテクチャー・ファミリー内のものであることが必要であることに注意してください。

-qtune オプションの使用

-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の情報については、「*XL C/C++ Compiler Reference*」の、『受け入れ可能な -qarch/-qtune の有効な組み合わせ』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。 **-qtune=balanced** オプションを使用してコンパイルされたコードは、パフォーマンスを確認してから配布してください。

-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++ Compiler Reference*」の **-qcache** を参照してください。

関連情報

- 73 ページの『Mathematical Acceleration Subsystem (MASS) ライブラリーの使用』
- *XL C/C++ Compiler Reference*の **-qarch**、**-qcache**、**-qtune**、および**-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。
- ベクトル命令を生成する。

高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 14. **-qhot** *のサブオプション*

サブオプション	振る舞い
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** と **-qhot=simd** を指定することと等価です。

表 14. **-qhot** *のサブオプション (続き)*

サブオプション	振る舞い
level=0	コンパイラーに高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**、**-qhot=noarraypad** および **-qhot=nosimd** 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。
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。
simd	コンパイラーに SIMD 自動ベクトル化を試みるよう指示します。つまり、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 命令の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。 Linux では、**-qarch** を、VMX 命令をサポートするターゲット・アーキテクチャーに設定すると (デフォルトでは **-qenablevmx** が有効である)、このサブオプションはデフォルト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**-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。
- 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** を試してください。
- コード・サイズが許容できないほど大規模の場合は、**-qcompact** を **-qhot** と共に使ってみてください。
- 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ます。

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qhot**、**-qenablevmx**、 および **-qstrict**

共用メモリーの並列処理 (SMP) 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++ Compiler Reference*」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、87 ページの『第 11 章 プログラムの並列処理』を参照してください。

表 15. 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 SMP プログラミング構成も、OpenMP ディレクティブを含めて認識可能です。 **-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、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。
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、ネストされた並列処理、ロッキング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、常に reentrant 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。

関連情報

- 「*XL C/C++ Compiler Reference*」の 『並列処理の環境変数』
- 「*XL C/C++ Compiler Reference*」の『コンパイラーの呼び出し』

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・ステップの両方で (「プログラム全体」モード) 指定できます(**clonearch** および **cloneproc** サブオプションを例外として、リンク・ステップで指定する必要があります)。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++ Compiler Reference*」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 16. 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。

表 16. 一般に使用される **-qipa** *のサブオプション (続き)*

サブオプション	振る舞い
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るようになります。
clonearch=*arch_list*	最適化命令を生成できる、複数アーキテクチャーを指定できるようになります。サポートされるアーキテクチャーの値は、**PWR4**、**PWR5**、**PWR6**、および **PPC970** です。プログラムの各関数について、コンパイラーは有効な **-qarch** 値にしたがって、命令セットの汎用バージョンを生成し、そして適切であれば、ユーザーがこのサブオプションに指定するアーキテクチャーの命令セットの *clones* 特殊バージョンを生成します。コンパイラーは、プロセッサー・アーキテクチャーをランタイムにチェックするために、ユーザーのアプリケーションにコードを挿入し、生成される命令の、ランタイム環境に最適化されたバージョンを選択します。
cloneproc=*func_list*	**clonearch** サブオプションによって、指定アーキテクチャーにクローンされる、正確な関数の指定を可能にします。
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。

- 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、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、**-qipa=inline** または **-qipa=noinline** の使用を検討してください。個々の関数のインライン化を制御するには、**-qipa=[no]inline=***function_name* を使用します。

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。

関連情報

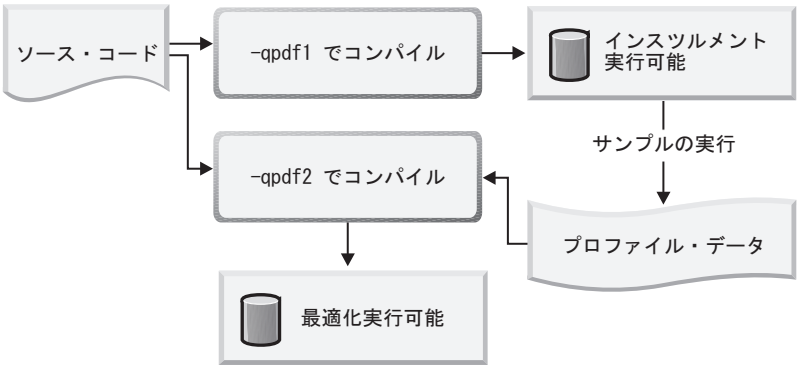
- 「*XL C/C++ Compiler Reference*」の **-qipa**、**-Q**、および **-qlist**

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。 PDF プロセスは、他のデバッグや調整が完了した後で使用されることを意図していて、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されています。 **-qipa** や最適化レベル **-O4** および **-O5** など他の最適化は、PDF と併用したときもメリットがあります。

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 **-O2** 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。 **-qpdf2** オプションを指定して、プログラムをもう一度コンパイルします。これにより、プロファイル・データに基づいてプログラムが最適化されます。代わりに、**-qpdf2** ステップのフル再コンパイルをしないことで多くの時間を節約したい場合は、単純に **-qpdf1** ステップによって作成されたオブジェクト・ファイルに再リンクしてください。

PDF を使用するには、以下の手順に従います。

1. プログラム内のソース・ファイルの一部または全部を **-qpdf1** オプションでコンパイルします。少なくとも **-O2** 最適化オプションを指定する必要があります。また、少なくとも有効な **-O2** とリンクする必要があります。ファイルのコンパイルに使用するコンパイラー・オプションに注意してください。後で同じオプションを使用する必要があります。
2. 完了したプログラムの正常実行時に使用されるデータを代表するデータを使用してプログラムを最後まで実行します。プログラムは、終了時にプロファイル情報を記録します。さまざまなデータ・セットを使用してプログラムを何回でも実行できます。プロファイル情報が累積され、分岐の頻度カウントやコード・ブロックの実行頻度が、使用された入力データに基づいて提供されます。アプリケーションが終了すると、デフォルトでは、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ある PDF ファイルにプロファイル情報が書き込まれます。インスツルメンテーション・ファイルのデフォルト名は ._pdf です。デフォルトをオーバーライドするには、**-qpdf1** ステップで **-qipa=pdfname** オプションを使用します。
3. 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・オーバーヘッドを含んでいないので、フルスピードで実行されます。

   注: オプション **-L**、**-l**、およびその他のいくつかはリンカー・オプションです。それらはこの時点で変更できます。

-qpdf2 を中間ステップとして使用すれば、**-qpdf1** パスで作成されたオブジェクト・ファイルをリンクできるので、**-qpdf2** パスでソースを再コンパイルする必要がありません。これはかなりの時間の節約となり、大規模なアプリケーションを最適化の

ために微調整するのに役立ちます。 **-qpdf2** パスでさまざまなオプションを使用することで、PDF 最適化バイナリーのさまざまなフレーバーを作成してテストすることができます。

注:

- 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集中することもできます。
- プログラムを **-qpdf1** または **-qpdf2** でコンパイルすると、デフォルトでは、**-qipa** オプションも **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、間違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、プロファイル・データ・ファイルを見つけることができなくなります。そうなると、プログラムを正しく最適化できなくなるか、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、PDFDIR 変数の値を変更して、PDF プロセスを完了する前にアプリケーションを実行した場合にも起こる可能性があります。
- 特定のプログラムについては、すべてのコンパイル・ステップで同じコンパイラー・オプションを使用しなければなりません。そうでないと、PDF はプログラムを正しく最適化できなくなり、速度が低下するおそれがあります。構成ファイルが提供するものも含めて、すべてのコンパイラー設定が同じでなければなりません。
- XL C/C++ の現行バージョン・レベルで作成された PDF ファイルと、コンパイラーの他のバージョン・レベルで作成された PDF ファイルとを混用しないようにしてください。
- プログラムを **-qpdf1** でコンパイルする場合は、プログラムの実行時にプロファイル情報が生成され、それがかなりのパフォーマンス・オーバーヘッドを伴うことを忘れないでください。このオーバーヘッドは、**-qpdf2** で再コンパイルするか、まったく PDF なしで再コンパイルすると解決されます。

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 **-O2** 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。デフォルトでは、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。 PDF ファイルのデフォルト名は ._pdf です。
3. PDFDIR 環境変数または **-qipa=pdfname** オプションで指定した PDF ファイルの場所を変更して、別の場所に PDF ファイルが作成されるようにする。
4. アプリケーションの再コンパイルまたは再リンクを、**-qpdf1** および最小レベル **-O2** を使って行う。
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 ファイルを結合する。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. アプリケーションの再コンパイルまたは再リンクを、**-qpdf2** および最小レベル **-O** を使って行う。

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。

showpdf によるプロファイル情報の表示

関数呼び出しおよびブロック統計に関する詳細情報を収集して表示するには、**-qshowpdf** オプションでコンパイルした後、**showpdf** ユーティリティーを使用します。次の例は、**showpdf** ユーティリティーでプロファイル指示フィードバック (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します。

   ```
   xlc -qpdf1 -qshowpdf -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. **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します。コンパイル時に **-qipa=pdfname** オプションを使用した場合は、**-f** オプションを使用してインスツルメンテーション・ファイルを指定します。

   ```
   showpdf -f instr1
   ```

結果は次のようになります。

```
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 )
```

関連情報

- 「*XL C/C++ Compiler Reference*」の **-qpdf** および **-showpdf**

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

実行可能ファイル全体を最適化するほか、個々のオブジェクトにプロファイル指示フィードバック (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 の場合は、**-qpdf1** を使用してアプリケーションをコンパイルし、典型的なデータでアプリケーションを実行し、**-qpdf2** を使用して再びコンパイルしますが、現在では、**-qnoipa** オプションも使用してリンク・ステップをスキップするようにします。

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** を使用してアプリケーション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、オプション **-O3** を使用して、中程度の最適化を指定しています。
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
   ```

   注: 同じ最適化オプションを使用する必要があります。この例では、最適化オプションは **-O3** です。
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** を使用してアプリケーション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。**注**: 前の手順で使用したのと同じ最適化オプションを使用する必要があります。この例では、最適化オプションは **-O3** です。

   ```
   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、最適化されるオブジェクト・ファイルは file1.o、file2.o、および file3.o です。これらのファイルをリンクするには、システム・ローダー **ld** を使用するか、**-qpdf2** ステップで **-c** オプションを除外します。

注:

- 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。
- どのコンパイルおよびリンク・ステップでも同じ最適化オプションを使用する必要があります。
- **-qpdf2** ステップでは、オブジェクト・ファイルのリンクがスキップされるように **-qnoipa** を指定する必要があるため、プロシージャー間分析 (IPA) 最適化とオブジェクト・レベルの PDF を同時に使用できなくなります。

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。これらのオプションの詳細については、「*XL C/C++ Compiler Reference*」に記載されている各オプションの見出しを参照してください。

表 17. 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)
-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	低レベル・オプティマイザーによる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。**-qunroll** は、**-O3** のもとで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。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 17. 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。
-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、デフォルトで最適化レベル **-O3** 以上の場合に使用されます。

第 8 章 最適化されたコードのデバッグ

最適化されたプログラムをデバッグすると、特別なユーザビリティー上の問題が生じます。最適化により、演算順序の変更、コードの追加や除去、変数データ位置の変更が行われたり、生成されたコードを元のソース・ステートメントに関連付けることが困難になるその他の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、必ずしも変数の最新値が存在する場所が明らかではありません。例えば、メモリー内の値は、最新の現行値がレジスターに格納されている場合は最新でないことがあります。ほとんどのデバッガーは変数の格納値の除去について行くことができないため、デバッガーにとっては、あたかもその変数が一度も更新されていないか、場合によっては設定さえもされていないように見えることがあります。これは、すべての値がメモリーにフラッシュ・バックされ、デバッグがより効果的で使用可能である非最適化とは対照的です。

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、命令の実行順序が、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期した順序にならないことがあります。また、命令の順序が連続しないこともあります。ユーザーがデバッガーでプログラムをステップスルーすると、あたかもコード内の前に実行された行に戻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、同じ値を 2 つの異なる変数に割り当てる式がプログラムに 2 つあると、コンパイラーは単一の変数に置換することがあります。これはデバッグのユーザビリティーを妨げるおそれがあります。最適化されたプログラムではプログラマーが存在すると予期していた変数がもはや存在しないからです。

プログラムを最適化する一方でデバッグ機能も向上させる方法には次の 2 種類があります。

最適化されていないコードをまずデバッグする

プログラムの最適化されていないバージョンをまずデバッグし、その後で目的の最適化オプションを使用して再コンパイルします。この方法で役立ついくつかのコンパイラー・オプションについては、60 ページの『最適化前のデバッグ』を参照してください。

-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。このオプションを使用した場合は、プログラムをデバッガーにロードしたとき、最適化されたプログラムの疑似コードをデバッグすることになります。詳しくは、61 ページの『最適化されたプログラムのデバッグに役立つ -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** オプションを使用してコンパイルすると、正しくない結果を繰り返し出そうと試みられます。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。
- 未初期化ストレージのバリエーション。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。

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。

最適化前のデバッグ

まずプログラムをデバッグし、その後で目的の最適化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。そして、最適化されたプログラムをテストしてから実動を開始します。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。

-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。

-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。

-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 コンパイラーが行ったループ変換およびプログラムの並列処理のレポートを作成するようコンパイラーに指示します。 **-qreport** でリストを生成する場合は、**-qhot** または **-qsmp** オプションも指定する必要があります。

-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。

-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。

また、**snapshot** プラグマを使用すれば、アプリケーション内の諸点で一定の変数がデバッガーに見えるようにすることができます。

最適化されたプログラムのデバッグに役立つ -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**、**-qipa**、または **-qpdf** 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。

下記の図を足掛かりにして、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単純プログラムのデバッグと元のソースのデバッグとの相違点を理解してください。

図 3 **元のコード**: 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、コンパイラーにいくつかの最適化機会を与えます。 例えば、変数 `z` と `d` はともに等価式 `x + y` によって割り当てられます。したがって、この 2 つの変数は最適化されたソース内では統合が可能です。また、ループをアンロールできます。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となります。

63 ページの図 4 **dbx デバッガー・リスト**: dbx 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。

63 ページの図 5 **最適化されたソースのステップスルー**: dbx 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。

```
#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;
}
```

図 3. **元のコード**

```
dbx> list
     1    3 |  void foo(long x, long y, char * w)
     2    9 |  {
     3           a = printf("TEST/n");
     4   12 |    @CSE0 = x + y;
     5           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     6           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     7           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     8           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     9   13 |    return;
    10         } /* function */
    11   15 |  long main()
    12   17 |  {
    13           d$init$0 = "DEBUG";
    14   18 |    foo(3,4,&d)
    15   19 |    rstr = 0;
    16           return rstr;
    17   20 |  } /* function */
```

図 4. **dbx デバッガー・リスト**

```
dbx> stop at 3
[1] stop at "myprogram.o.optdbg":3
dbx> run
TEST
[1] stopped in foo(int,int,char*) at line 3 in file "myprogram.o.optdbg" ($t1)
    3       16 |    @CSE0 = x + y;
dbx> step
stopped in foo(int,int,char*) at line 4 in file "myprogram.o.optdbg" ($t1)
    4               printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
dbx> step
3 7 7 EBUG EBUG
stopped in foo(int,int,char*) at line 5 in file "myprogram.o.optdbg" ($t1)
    5               printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
dbx> cont
3 7 7 EBUG EBUG
3 7 7 EBUG EBUG
3 7 7 EBUG EBUG

execution completed
```

図 5. **最適化されたソースのステップスルー**

第 9 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

37 ページの『第 7 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』
- 66 ページの『関数呼び出しによるオーバーヘッドの低減』
- 67 ページの『効率的なメモリーの管理』
- 68 ページの『変数の最適化』
- 69 ページの『効率的なストリングの操作』
- 69 ページの『式とプログラム・ロジックの最適化』
- 70 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。

- 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する。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固有です。低水準の関数に対しては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。
- `scanf` および `fscanf` の代わりに、`fgets` を使用してストリング内を読み取り、それから `atoi`、`atol`、`atof`、または `_atold` のうち 1 つを使用してそれを適切な形式に変換する。
- 複雑なフォーマット設定にのみ `sprintf` を使用する。ストリングの連結など、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。
- 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数を使用する。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。
- 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、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。
- 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、いつも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++ 関数は、XL C/C++ ヘッダー・ファイルを組み込むと、自動的に組み込み関数にマップされます。

  C 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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **#pragma isolated_call**、**#pragma disjoint**、および **-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。
- 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て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。
- **#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。

関連情報

- 「*XL C/C++ Compiler Reference*」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。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。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。

関連情報

- 「*XL C/C++ Compiler Reference*」の **#pragma strings (読み取り専用)** および **-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。
- 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。

- 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。

第 10 章 ハイパフォーマンス・ライブラリーの使用

IBM XL C/C++ Advanced Edition for Linux, V9.0 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 82 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

MASS ライブラリーは、スカラー XL C/C++ 関数 のライブラリー (『スカラー・ライブラリーの使用』を参照) と、特定のアーキテクチャー用にチューニングされたベクトル・ライブラリー・セット (76 ページの『ベクトル・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。

81 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とリンク、および 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・スカラ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結合できないため、スカラー・ライブラリーとベクトル・ライブラリーは、それぞれ 2 種類のバージョン (32 ビット・アプリケーションの場合は libmass.a と libmassv.a、64 ビット・アプリケーションの場合は libmass_64.a と libmassv_64.a) がコンパイラーと一緒に出荷されます。

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改善された、頻繁に使用される数学組み込み関数の調整されたセットが含まれています。プログラムを下記のいずれかのオプションを使ってコンパイルするときは、これらの関数を使用できます。

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

ほとんどの数学ライブラリー関数について、コンパイラーが自動的により高速な MASS 関数を使用します。実際には、コンパイラーは最初に数学ライブラリー関数を等価な MASS ベクトル関数で置き換えることにより vectorize 呼び出しを試みます。それができない場合は、MASS スカラー関数を使います。コンパイラーが数学ライブラリー関数の自動置換を行うときは、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。 MASS 関数に対する特別の呼び出しをコードに追加したり、`libxlopt` ライブラリーにリンクしたりする必要はありません。

上記の最適化オプションをいずれも使用しないで、明示的に MASS スカラー関数を呼び出したい場合は、次のようにしてそれを行うことができます。

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。説明については、 81 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 18 に示します。

表 18. 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戻す	double acos (double x);	float acosf (float x);
acosh	acoshf	x の双曲線アークコサインを戻す	double acosh (double x);	float acoshf (float x);
	anint	x の丸められた整数値を戻す		float anint (float x);
asin	asinf	x のアークサインを戻す	double asin (double x);	float asinf (float x);
asinh	asinhf	x の双曲線アークサインを戻す	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戻す	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戻す	double atanh (double x);	float atanhf (float x);
cbrt	cbrtf	x の立方根を戻す	double cbrt (double x);	float cbrtf (float x);

表 18. 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
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 のコサインを戻す	double cos (double x);	float cosf (float x);
cosh	coshf	x の双曲線コサインを戻す	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 の誤差関数を戻す	double erf (double x);	float erff (float x);
erfc	erfcf	x の補完的な誤差関数を戻す	double erfc (double x);	float erfcf (float x);
exp	expf	x の指数関数を戻す	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戻す	double lgamma (double x);	float lgammaf (float x);
log	logf	x の自然対数を戻す	double log (double x);	float logf (float x);
log10	log10f	x の対数 (底 10) を戻す	double log10 (double x);	float log10f (float x);
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);
rsqrt		x の平方根の逆数を戻す	double rsqrt (double x);	
sin	sinf	x のサインを戻す	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 の双曲線サインを戻す	double sinh (double x);	float sinhf (float x);

表 18. 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
sqrt		x の平方根を戻す	double sqrt (double x);	
tan	tanf	x のタンジェントを戻す	double tan (double x);	float tanf (float x);
tanh	tanhf	x の双曲線タンジェントを戻す	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。
- あるケースでは、MASS 関数は libm.a ライブラリーの場合ほど正確でなく、エッジに対する処理が異なることがあります (sqrt(Inf) など)。

ベクトル・ライブラリーの使用

プログラムを下記のいずれかのオプション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、等価 MASS ベクトル関数 (関数 vdnint、vdint、vsincos、vssincos、vcosisin、vscosisin、vqdrt、vsqdrt、vrqdrt、vsrqdrt、vpopcnt4、および vpopcnt8 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 (vectorize) 呼び出しを試みます。自動ベクトル化の場合、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。MASS 関数に対する特別の呼び出しをコードに追加したり、libxlopt ライブラリーにリンクしたりする必要はありません。

上記の最適化オプションをいずれも使用しないで、明示的に MASS ベクトル関数を呼び出したい場合は、ソース・ファイルに XL C/C++ ヘッダー massv.h ファイルを組み込み、アプリケーションを適切なベクトル・ライブラリーとリンクさせることで、それを行うことができます。 (リンクについては、81 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。)

libmassvp4.a

POWER4™ アーキテクチャー用に調整された関数を含みます。 PPC970 マシンを使用している場合は、このライブラリーを選択することをお勧めします。

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。

Linuxでは、32 ビットおよび 64 ビット・オブジェクトを単一ライブラリーに混在させてはなりません。したがって、別個の 64 ビット・バージョンの各ベクトル・ライブラリー (libmassvp4_64.a、libmassvp5_64.a、および libmassvp6_64.a) が提供されます。

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、78 ページの表 19 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、80 ページの表 20 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
- 整数のベクトル長さ (vector-length) 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、プレフィックス v が付いた関数では倍精度、プレフィックス vs が付いた関数では単精度を想定します。下記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) はパラメーターを 4 つ使います。関数 vdiv、vpow、および vatan2 はパラメーター (*z*,*x*,*y*,*n*) を使います。関数 vdiv はベクトル *z* を出力します。この要素は 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。この要素は $x[i]^{y[i]}$ (i=0,..,*n-1) です。関数 vatan2 はベクトル *z* を出力します。この要素は atan(x[i]/y[i]) (i=0,..,*n-1) です。関数 vsincos はパラメーター (*y*,*z*,*x*,*n*) を使用して、2 つのベクトル *y* および *z* を出力します。この要素はそれぞれ sin(x[i]) および cos(x[i]) です。

表 19. 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] の整数切り捨てに設定する	void vdint (double y[], double x[], int *n);	

表 19. 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
vdiv	vsdiv	i=0,..,*n-1 の場合に、z[i] を x[i]/y[i] に設定する	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);	
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);
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);
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);
vlog10	vslog10	i=0,..,*n-1 の場合に、y[i] を x[i] の底が 10 の対数に設定する	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);
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);
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);

表 19. 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
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 20. 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	合計数の 1 ビットをバイナリー表記 x[i] (ただし i=0,..,*n-1) の連結で戻します。ここで x は 32 ビットのオブジェクトのベクトルです。	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	合計数の 1 ビットをバイナリー表記 x[i] (ただし i=0,..,*n-1) の連結で戻します。ここで x は 64 ビットのオブジェクトのベクトルです。	unsigned int vpopcnt8 (void *x, int *n)

入力と出力の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 (disjoint) の入力および出力ベクトルを使って呼び出されます。これにより 2 つのベクトルがメモリー内でオーバーラップすることはありません。別の一般的な使用手順では、入力と出力パラメーターの両方に同一のベクトルを使って呼び出します (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。

  上記の制限は、ベクトル `y1`、`x` と `y2`、`x` の両ペアに適用されます。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。

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意味で、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 を使用するプログラムのコンパイルとリンク

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、**-l** リンカー・オプションで、**mass** と **massvp4**、**massvp5**、**massvp6** (32 ビット) のいずれか、または **mass_64** と **massvp4_64**、**massvp5_64**、**massvp6_64** (64 ビット) のいずれかを指定します。

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 progc -lmass -lmassvp4
xlc progc.c -o progc -lmass_64 -lmassvp4_64 -q64
```

MASS 関数は、デフォルトの丸めモードおよび浮動小数点例外トラッピング設定で実行する必要があります。

数学システム・ライブラリーでの libmass.a の使用

一部の関数には libmass.a (または libmass_64.a) スカラー・ライブラリーを使用して、その他の関数には通常の数学ライブラリー libm.a を使用したい場合は、次の手順に従ってプログラムのコンパイルとリンクを行ってください。

1. **ar** コマンドを使用して、libmass.a または libmass_64.a から求める関数のオブジェクト・ファイルを抽出します。大部分の関数にとって、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) または .s64.o (64 ビット・モード) 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** を指定した **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 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 に含まれています。
2. XL C/C++ pow 関数または XL Fortran ** (指数) 演算子は、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++ 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。

83 ページの『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。詳細な追加情報とこれらの関数の使用例については、「*Engineering and Scientific Subroutine Library Guide and Reference*」を参照してください。これらは http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html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、85 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* が指定する配列の先導ディメンション (leading dimension)。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* が指定する配列の先導ディメンション (leading dimension)。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 は float (sgemm 用) または double (dgemm 用) 値の入力行列 *b* です。

ldb
は、*b* が指定する配列の先導ディメンション (leading dimension)。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 は float (sgemm 用) または double (dgemm 用) 値の出力行列 *c*

ldc は、*c* が指定する配列の先導ディメンション (leading dimension)。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++ 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行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 11 章 プログラムの並列処理

コンパイラーは、共用メモリー・プログラムの並列処理について 3 つの方法を提案しています。それは以下の通りです。

- 計数可能なプログラム・ループの自動並列処理で、『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 89 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、 89 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。並列化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++ Compiler Reference*」の『並列処理のための XLSMPOPTS 環境変数サブオプション』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++ Compiler Reference*」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++ Compiler Reference*」の『並列処理のための組み込み関数』を参照してください。

スレッドの作成方法と使用法に関する完全な説明については、www.openmp.org から使用可能な「*OpenMP Application Program Interface Language Specification*」を参照してください。

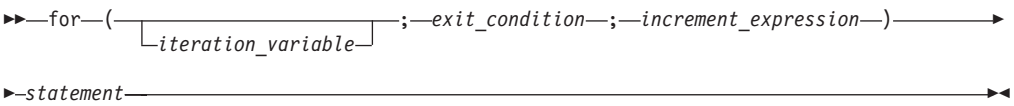
関連情報

- 48 ページの『共用メモリーの並列処理 (SMP) の使用』

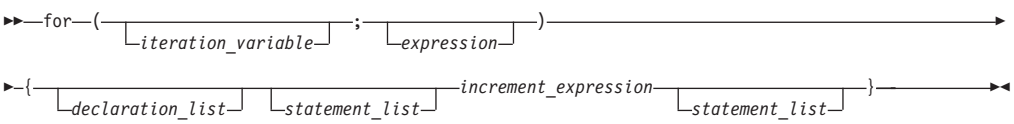
計数可能ループ

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。

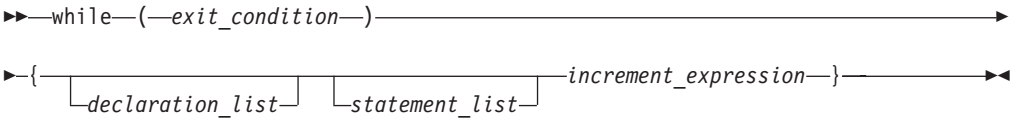
計数可能ループ構文 (Countable for loop syntax)



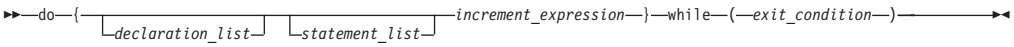
ステートメント・ブロック付きの計数可能ループ構文 (Countable for loop syntax with statement block)



計数可能ループ中構文 (Countable while loop syntax)



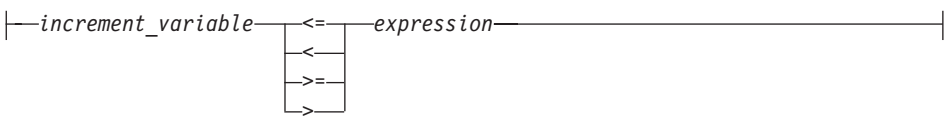
計数可能ループ中 Do 構文 (Countable do while loop syntax)



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、 87 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。
- インクリメントする式がクリティカル・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・モードで呼び出される。

OpenMP ディレクティブの使用

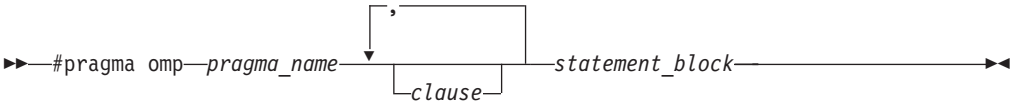
OpenMP ディレクティブは、さまざまなタイプの*並列領域* を定義することによって、共用メモリーの並列処理を有効に活用します。並列領域には、プログラム・コードの反復セグメントと非反復セグメントの両方を組み込むことができます。

プラグマは、4 つ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プラグマ (**#pragma omp section**, **#pragma omp ordered**, **#pragma omp single**)。
3. スレッド間の同期をユーザーがコントロールできるプラグマ (**#pragma omp atomic**、**#pragma omp master**、**#pragma omp barrier**、**#pragma omp critical**、**#pragma omp flush**)。

4. スレッド間のデータの可視性の有効範囲をユーザーが定義できるプラグマ (**#pragma omp threadprivate**)。

OpenMP ディレクティブ構文



プラグマ・ディレクティブは、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。例えば、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 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++ Compiler Reference*」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。

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、関連した並列ループ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。
- 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、並列ループ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 */
 ... }
```

プログラムのセマンティクスを変更しないでそれが可能ならば、コンパイラーはいくつかの共用変数を専用化することができます。例えば、各ループの反復が共用変数の固有値を使う場合、その変数を専用化することができます。専用化された共用変数は、**-qinfo=private** オプションによってレポートされます。このレポートにリストされていないすべての共用変数へのアクセスを同期化するには、クリティカル・セクションを使用してください。

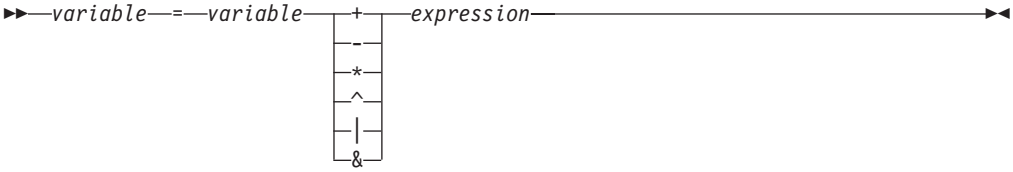
いくつかの OpenMP プリプロセッサー・ディレクティブでは、選択されたデータ変数に対する可視性コンテキストの指定をユーザーが行うことができます。データ有効範囲の属性文節の要約を以下にリストします。

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。
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。
shared	**shared** 文節は、チームのすべてのスレッドのリストに表示される変数を共用します。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** 文節は、ユーザーが変数のデータ有効範囲の属性に影響を与えることを許容します。

詳しい情報については、OpenMP ディレクティブの説明を 「*XL C/C++ Compiler Reference*」の『並列処理のためのプラグマ・ディレクティブ』 または「*OpenMP Application Program Interface Language Specification*」から参照してください。

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。

認識された縮小は **-qinfo=reduction** オプションによってリストされます。 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。

〒106-8711
東京都港区六本木 3-2-12
IBM World Trade Asia Corporation
Intellectual Property Law & Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

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

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

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

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

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

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

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

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

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

著作権使用許諾:

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

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

商標

資料に記載されている会社名、製品名、またはサービス名は、IBM Corporation または各社の商標である可能性があります。IBM Corporation の商標については、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Microsoft および Windows は、Microsoft Corporation の米国およびその他の国における商標です。

Intel は、Intel Corporation またはその子会社の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

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。

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 9899-1990) に準拠しています。
- C 言語は、International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)) にも準拠しています。
- 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 および C++ Application Programming Interface Version 2.5 に準拠しています。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ラ行]

［数字］

B

C

F

I

L

M

O

S

X

［特殊文字］

プログラム番号: 5724-S73

SC88-4646-00

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