IBM XL C for AIX, V10.1

最適化およびプログラミング・ガイド

SC88-4920-00
(英文原典：SC23-8883-00)

IBM XL C for AIX, V10.1

最適化およびプログラミング・ガイド

SC88-4920-00
(英文原典：SC23-8883-00)

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、 125 ページの『特記事項』に記載されている情報をお読みください。

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

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

原典： SC23-8883-00
IBM XL C for AIX, V10.1
Optimization and Programming Guide
Version 10.1

発行： 日本アイ・ビー・エム株式会社

担当： ナショナル・ランゲージ・サポート

第1刷 2008.5

目次

本書について

このガイドは、IBM® XL C for AIX®, V10.1 コンパイラーの使用法に関する上級者向けのトピックを、特にプログラムの移植性と最適化に重点を置いて説明したものです。このガイドは、お勧めするプログラミング手法とコンパイル・プロシージャーを用いて、コンパイラーの能力を最大に引き出すための、参照情報と実用的ヒントの両方を提供しています。

本書の対象読者

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

本書の読み方

本書では各トピックの説明のために「タスク指向」アプローチを採用して、各章ごとに特定のプログラミングやコンパイルの問題に焦点を集中しています。各トピックにはまた IBM XL C for AIX, V10.1 文書セット内の解説書にある関連セクションに対する豊富な相互参照が含まれていて、コンパイラー・オプションやプラグマ、そして特定の言語拡張についての説明が提供されています。

本書の構成

このガイドは下記のトピックが記載されています。

- 1 ページの『第 1 章 32 ビット・モードおよび 64 ビット・モードの使用』 は、既存の 32 ビットのアプリケーションを 64 ビット・モードに移植するときに発生する共通問題について説明し、その問題を避けるための勧告を提供しています。
- 7 ページの『第 2 章 XL C の Fortran での使用』 では XL C プログラムから FORTRAN コードを呼び出す際の考慮事項について説明しています。
- 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 のような、集合体内におけるデータの位置合わせのコントロールに使用できる別のコンパイラー・オプションについて説明しています。
- 21 ページの『第 4 章 浮動小数点演算の処理』 は、コンパイラーが取り扱う浮動小数点操作の方法のコントロールに使用可能なオプションを説明します。
- 27 ページの『第 5 章 メモリー・ヒープの使用』 は、カスタム・メモリー・ヒープの使用やデバッグ用ヒープ・メモリーの妥当性検査を含む、ヒープ・メモリー管理のためのコンパイラー・ライブラリー関数について説明します。
- 45 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について説明します。

- 49 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。
- 79 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。
- 85 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 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。
- 99 ページの『第 11 章 プログラムの並列処理』 は、XL C が提供する、マルチスレッド・プログラム (IBM SMP および 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。
- 107 ページの『第 12 章 メモリー・デバッグ・ライブラリー関数』 は、すべてのコンパイラー・デバッグ・メモリー・ライブラリー関数の参照リストを提供しています。

表記規則

書体の規則

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

表 1. 書体の規則

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

構文図

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

- 構文図は、線の経路に沿って左から右、上から下に読みます。

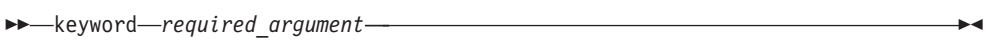
  ▶▶── 記号は、コマンド、ディレクティブ、またはステートメントの始まりを示します。

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

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

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

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



- オプション項目はメインパスの下側に示されます。

- 複数の項目から選択できる場合、それらの項目は縦に並べて表示されます。

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

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

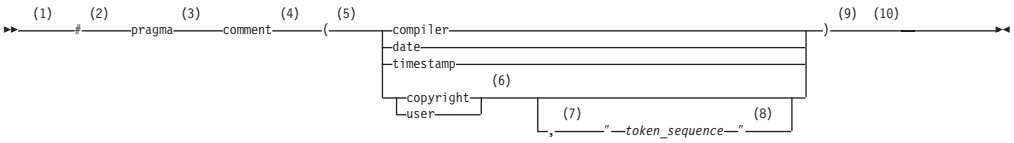
- メインライン上の左に戻る矢印 (繰り返し矢印) は、縦に並んだ項目から複数の項目を選択できること、または同じ項目を繰り返すことができることを示します。ブランク以外の分離文字も、次のように示されます。

- デフォルトの項目は、メインパスの上側に表示されます。

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

構文図の例

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



注:

1 これが構文図の始まりです。

2 記号 # で始まります。

3 キーワード pragma は # 記号の後に続けます。

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

5 ここに左括弧が必要です。

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

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

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

9 ここに右括弧が必要です。

10 これが構文図の終わりです。

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

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

本書での例

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

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

関連情報

以下のセクションには、XL Cの関連情報が記載されています。

IBM XL C の資料

XL C には、次の形式の製品情報が用意されています。

- README ファイル

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

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

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

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

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

以下が、XL C のすべての製品情報を構成するファイルです。

表 2. XL C PDF ファイル

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

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

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

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

標準および仕様

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

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*、*C89* とも呼ばれています。
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*、*C99* とも呼ばれています。
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. このドラフト技術レポートは、C 標準委員会によって受諾されており、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf から入手できます。

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

その他の IBM 資料

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

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

その他の資料

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

技術サポート

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

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

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

第 1 章 32 ビット・モードおよび 64 ビット・モードの使用

XL 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。あるいは、*OBJECT_MODE* 環境変数を 32 または 64 に設定する方法もあります。

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 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、64 ビット・モードでの演算の最適化を参照してください。

「*XL C コンパイラー・リファレンス*」の関連情報

-q32、-q64

-qwarn64

コンパイル時およびリンク時の環境変数

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 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。

64 ビット・モードでは、値によって va_arg 引数に渡される構造体のメンバー値には、その構造体のサイズが 8 バイトの倍数でないと、正しくアクセスできないことがあります。

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 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 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 と Fortran の同等のデータ型

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

関連情報

7 ページの『第 2 章 XL C の Fortran での使用』

第 2 章 XL C の Fortran での使用

XL C を使って FORTRAN で書かれた関数を、ユーザーの 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『関数ポインター』
- 9 ページの『サンプル・プログラム: Fortran を呼び出す 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 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 および Fortran 間のデータ型の対応

C データ型	Fortran データ型
_Bool	LOGICAL(1)
char	CHARACTER
signed char	INTEGER*1
unsigned char	LOGICAL*1
signed short int	INTEGER*2

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 および Fortran 間のデータ型の対応 (続き)

C データ型	Fortran データ型
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 (default)	REAL*8 DOUBLE PRECISION
long double (-qlongdouble または -qldbl128 つき)	REAL*16
float _Complex	COMPLEX*8 または COMPLEX(4)
double _Complex	COMPLEX*16 または COMPLEX(8)
long double _Complex (デフォルト)	COMPLEX*16 または COMPLEX(8)
long double _Complex(-qlongdouble または -qldbl128 つき)	COMPLEX*32 または COMPLEX(16)
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コンパイラー・リファレンス*」**の関連情報**

-qldbl128

-qalign

文字および集合データ

大部分の数値データ型は、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。 9 ページの表 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 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 ランゲージ・リファレンス*」を参照してください。

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

関数ポインター

関数ポインターは、その値が関数のアドレスであるデータ型です。 Fortran では、EXTERNAL ステートメント内に現れる仮引数が関数ポインターです。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

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 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 interlanguage call example

C Compile separately, then link to 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 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。このセクションではまた、さまざまな位置合わせモードに基づく集合体レイアウトの例も示します。

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

「*XL C コンパイラー・リファレンス*」**の関連情報**

-qaltivec

関連外部情報

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

位置合わせモードの使用

XL C がサポートする各々のデータ型は、プラットフォーム固有のデフォルト位置合わせモードに従って、バイト境界に沿って位置合わせされます。AIX では、デフォルトの位置合わせモードは **power** または **full** です。これらは同等のものです。

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)

		位置合わせの設定				
データ型	**ストレージ**	**natural**	**power、full**	**mac68k、twobyte[3]**	**bit_packed[2]**	**packed[2]**
_Bool (32 ビット・モード)	1	1	1	1	1	1
_Bool (64 ビット・モード)	1	1	1	サポートされない	1	1
char、signed char、unsigned char	1	1	1	1	1	1
wchar_t (32 ビット・モード)	2	2	2	2	1	1
wchar_t (64 ビット・モード)	4	4	4	サポートされない	1	1
int、unsigned int	4	4	4	2	1	1
short int、unsigned short int	2	2	2	2	1	1
long int, unsigned long int (32 ビット・モード)	4	4	4	2	1	1
long int, unsigned long int (64 ビット・モード)	8	8	8	サポートされない	1	1
_Decimal32	4	4	4	2	1	1
_Decimal64	8	8	8	2	1	1
_Decimal128	16	16	16	2	1	1
long long	8	8	8	2	1	1
float	4	4	4	2	1	1
double	8	8	注を参照 [1]	2	1	1
long double	8	8	注を参照 [1]	2	1	1

表 10. 位置合わせの設定 (値はバイト数で示される) (続き)

データ型	ストレージ	位置合わせの設定				
		natural	**power、full**	**mac68k、twobyte**[3]	**bit_packed**[2]	**packed**[2]
long double with **-qldbl128**	16	16	注を参照 [1]	2	1	1
pointer (32 ビット・モード)	4	4	4	2	1	1
pointer (64 ビット・モード)	8	8	8	サポートされない	1	1
vector types	16	16	16	16	1	1

注:

1. 集合体内ではこのデータ型の最初のメンバーがその natural 位置合わせ値に従って位置合わせされます。集合体の後続のメンバーは 4 バイト境界に位置合わせされます。
2. `packed` 位置合わせでは、ビット・フィールド・メンバーはビット・レベルでパックされません。ビット・フィールドをビット・レベルでパックする必要がある場合は、`bit_packed` 位置合わせを使用してください。
3. **mac68k** 位置合わせの場合、集合体にベクトル・メンバーが含まれなければ、位置合わせは 2 バイトになります。集合体にベクトル・メンバーが含まれる場合は、位置合わせはすべてのメンバーの中で最大規模になります。

`double`、`long long`、または `long double` データ型を含む集合体について作業をしている場合は、集合体の各メンバーはその natural 位置合わせ値に従って位置合わせされているので、最高のパフォーマンスのためには **natural** モードを使用してください。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。 15 ページの『ビット・フィールドの位置合わせ』では、その他の規則と、ビット・フィールドの使用と位置合わせに関する考慮事項について説明し、ビット・パック位置合わせの例を示します。

「*XL C コンパイラー・リファレンス*」の関連情報

-qalign

-qldbl128

#pragma options

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。
- 空の集合体はサイズ 0 バイトが割り当てられている。
- **mac68k** を除く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。
- **mac68k** 位置合わせの場合、集合体にベクトル・メンバーが含まれなければ、位置合わせは 2 バイトになります。集合体にベクトル・メンバーが含まれる場合は、位置合わせはすべてのメンバーの中で最大規模になります。
- 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。

下記のテーブルには、位置合わせモードによるいくつかの集合体のサイズの実例が示されています。

表 11. 位置合わせと集合体サイズ

例	集合体のサイズ		
	-qalign=power	**-qalign=natural**	**-qalign=packed**

struct Struct1 {
double a1;
char a2;
};
	16 バイト (最大の位置合わせ要求を持つメンバーは a1 である。したがって、a2 には 7 バイト埋め込まれる。)	16 バイト (最大の位置合わせ要求を持つメンバーは a1 である。したがって、a2 には 7 バイト埋め込まれる。)	9 バイト (各メンバーはその natural 位置合わせにパックされる。埋め込みは追加されない。)

struct Struct2 {
char buf[15];
};
	15 バイト	15 バイト	15 バイト

struct Struct3 {
char c1;
double c2;
};
	12 バイト (最大の位置合わせ要求を持つメンバーは c2; である。しかし、そのメンバーが double であって最初のメンバーではないので、4 バイト規則が適用される。 c1 には 3 バイトが埋め込まれる。)	16 バイト (最大の位置合わせ要求を持つメンバーは c2 である。したがって、c1 には 7 バイト埋め込まれる。)	9 バイト (各メンバーはその natural 位置合わせにパックされる。埋め込みは追加されない。)

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、15 ページの『ビット・フィールドの位置合わせ』を参照してください。

位置合わせの例

次の例は、以下の記号を使用して、埋め込みと境界を表示しています。

p = 埋め込み

| = ハーフワード (2 バイト) 境界

: = バイト境界

Mac68K の例

下の例では、

```
#pragma options align=mac68k
struct B {
    char a;
    double b;
     };
#pragma options align=reset
```

B のサイズは 10 バイトです。B の位置合わせは 2 バイトです。B のレイアウトは次のようになります。

```
|a:p|b:b|b:b|b:b|b:b|
```

Packed の例

下の例では、

```
#pragma options align=bit_packed
struct {
   char a;
   double b;
     } B;
#pragma options align=reset
```

B のサイズは 9 バイトです。B のレイアウトは次のようになります。

```
|a:b|b:b|b:b|b:b|b:
```

ネストされた集合体の例

下の例では、

```
#pragma options align=mac68k
struct A {
  char a;
  #pragma options align=power
  struct B {
     int b;
     char c;
     } B1;    // <-- B1 laid out using power alignment rules
  #pragma options align=reset    // <-- has no effect on A or B,
                                        but on subsequent structs
  char d;
};
#pragma options align=reset
```

A のサイズは 12 バイトです。A の位置合わせは 2 バイトです。 A のレイアウトは次のようになります。

```
|a:p|b:b|b:b|c:p|p:p|d:p|
```

ビット・フィールドの位置合わせ

ビット・フィールドは、_Bool、 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。

注: long long および unsigned long long は、AIX の C 言語では使用できません。

C 言語では、ビット・フィールドを、 int でなく char または short として指定することができますが、 XL C はそれらを unsigned int としてマップします。ビット・フィールドの長さは、その基本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 4 を戻します。

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。

natural 位置合わせの規則

- 長さ 0 のビット・フィールドがあると、そのビット・フィールドの基底宣言型の次の位置合わせ境界まで埋め込まれます。これによって、次のメンバーが、64 ビット・モードの long および 32 ビット・モードと 64 ビット・モードの両方の long long を除くすべての型について、4 バイトの境界で開始するようになり、それらについては、次のメンバーが次の 8 バイト境界に移動されます。直前のメンバーの記憶域レイアウトが適切な境界上で終了した場合は、埋め込みはされません。
- 長さ 0 のビット・フィールドのみを含む集合体は、長さは 0 バイトであり、4 バイトで位置合わせされます。

power 位置合わせの規則

- ビット・フィールドを含む集合体は 4 バイト (ワード) 位置合わせをする。
- ビット・フィールドは、現在のワードにパックされる。ビット・フィールドがワード境界に交差する場合は、そのビット・フィールドは次のワード境界から開始する。
- 長さが 0 のビット・フィールドがあると、その直後のビット・フィールドは、宣言型とコンパイル・モードに応じて、次のワード境界または 8 バイトに位置合わせされる。長さが 0 のビット・フィールドがワード境界にある場合は、その次のビット・フィールドはこの境界から開始される。
- 長さ 0 のビット・フィールドのみを含む集合体は、長さは 0 バイトです。

Mac68K 位置合わせの規則

- ビット・フィールドは、1 ワードにパックされ、2 バイト境界で位置合わせされる。
- ワード境界を交差するビット・フィールドは、ビット・フィールドがすでにハーフワード境界で開始していても、次のハーフワードに移動させられる。 (ビット・フィールドは、ワード境界を交差して終了することもある。)
- 長さが 0 のビット・フィールドは、現在このビット・フィールドがハーフワード境界にあったとしても、次のメンバーを、たとえそれがビット・フィールドでなくても、強制的に次のハーフワード境界から開始させる。
- 長さが 0 のビット・フィールドしか含んでいない集合体の長さは、長さ 0 のビット・フィールドの数の 2 倍 (バイト換算) である。

- 特殊な例として、最大のエレメントが長さ 16 以下のビット・フィールドを持つ共用体のサイズは、 2 バイトである。ビット・フィールドの長さが 16 よりも大きい場合は、共用体のサイズは 4 バイトになります。

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法

XL 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)。**注**: AIX 上では **#pragma pack** をビット・フィールドの共用体メンバーに適用できません。

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。

n: *n* には 1、2、4、8、または 16 を使用できます。つまり、構造体メンバーは *n* バイト境界または natural 位置合わせ境界のいずれか小さい値に合わせて位置合わせされます。 *nopack:* パッキングを使用不可にします。*pop*: **#pragma pack** によって追加された前の値を削除します。**注**: 空の大括弧を使用することには、*pop* と同一の機能があります。

__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 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 ランゲージ・リファレンス*」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)

「*XL Cコンパイラー・リファレンス*」**の関連情報**

#pragma pack

「*XL C ランゲージ・リファレンス*」**の関連情報**

位置合わせされた型属性

packed 型属性

__align 型修飾子

型属性

位置合わせされた変数属性

packed 変数属性

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 = maximum(#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』
- 22 ページの『乗加法演算の処理』
- 22 ページの『厳密な IEEE 準拠のためのコンパイル』
- 23 ページの『浮動小数点定数の折り畳みと丸めの処理』
- 25 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 は以下の 2 進浮動小数点フォーマットをサポートします。

- 0 および 10^{-38} から 10^{+38} の近似絶対範囲および近似絶対正規化範囲で、10 進法で約 7 桁の精度を持つ 32 ビット単精度の浮動小数点数
- 0 および 10^{-308} から 10^{+308} の近似絶対範囲および近似絶対正規化範囲で、10 進法で約 16 桁の精度を持つ 64 ビット倍精度の浮動小数点数
- 倍精度値と同じ範囲であるが、10 進法で約 32 桁の精度を持つ 128 ビット拡張精度の浮動小数点数

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

V9.0 以降は、選択したハードウェアおよびオペレーティング・システムのレベルに基づいて、コンパイラーは以下の 10 進浮動小数点フォーマットもサポートします。

- およそ 10^{-101} から 10^{+90} の範囲で、10 進法で 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^{-398} から 10^{+369} の範囲で、10 進法で 16 桁の精度を持つ 64 ビット倍精度の浮動小数点数
- およそ 10^{-6176} から 10^{+6111} の範囲であるが、10 進法で 34 桁の精度を持つ 128 ビット拡張精度の浮動小数点数

「*XL C コンパイラー・リファレンス*」の関連情報

-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。

注: 10 進浮動小数点は、乗加法の命令には使用しません。

「*XL C*コンパイラー・リファレンス」の関連情報

-qfloat

厳密な IEEE 準拠のためのコンパイル

XL 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

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

  「*XL C*コンパイラー・リファレンス」の関連情報

  -qfloat

  -qstrict

  -qinitauto

浮動小数点定数の折り畳みと丸めの処理

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

- コンパイル時の丸めモードが実行時の丸めモードと異なるとき。デフォルトでは、両方の場合とも最も近い値への丸めですが、プログラムが実行時に異なった結果を避けるために丸めモードを変更する場合は、以下のいずれかを行ってください。
  - 適切な **-y** オプションを使用して、コンパイル時の丸めモードを実行時のモードと一致するように変更してください。詳細な情報と例示については 『コンパイル時と実行時の丸めモードのマッチング』を参照してください。
  - **-qfloat=nofold** オプションを使ってコンパイルすることにより、折り畳みを抑止します。
- *a+b*c* のような式については、コンパイル時に部分的または全面的に評価を行います。実行時の乗加法命令は中間の丸めを全く使用しないにもかかわらず、*b*c* は *a* に加えられる前に丸められる場合があるため、実行時に得られた結果とは異なる可能性があります。異なった結果を避けるには下記のどちらかを行います。
  - **-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

25 ページの『浮動小数点例外の処理』

「*XL C*コンパイラー・リファレンス」**の関連情報**

-qfloat

-qstrict

-qflttrap

コンパイル時と実行時の丸めモードのマッチング

コンパイル時と実行時に使用されるデフォルトの丸めモードは、最も近い値への丸めです。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 [1] 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 */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to zero = %.17f\n", one_third);

 fesetround (FE_TONEAREST);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to nearest = %.17f\n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 fesetround (FE_DOWNWARD);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to -infinity = %.17f\n", one_third);

 return 0;
 }
```

注:

1. AIX 上では、fenv.h ヘッダー・ファイル内で宣言されている関数とマクロを得るため、この例をシステム数学ライブラリー libm にリンクする必要があります。
2. printf の呼び出し前に丸めモードを再設定する説明については、25 ページの『丸めモードおよび標準ライブラリー関数』 を参照してください。

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、あるいは 10 進浮動小数点のみ丸めモードに変更する場合は、**-qfloat=rrm** オプションも使用することをお勧めします。

「*XL C*コンパイラー・リファレンス」**の関連情報**

 -qfloat

 -y

丸めモードおよび標準ライブラリー関数

AIX 上では、C の入出力関数と変換関数が、有効な丸めモードをその関数による入力または出力の値に適用します。これらの関数は、 printf、scanf、atof、および ftoa、を組み込みます。

例えば、現行の丸めモードが「無限への丸め」の場合、すでに計算として行われた丸めに追加して、printf 関数はその丸めモードを、印刷しようとする浮動小数点桁ストリング値に適用します。以下の例で説明しています。

```
#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;

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
 printf ("1/3 with execution-time rounding to +infinity = %.17f¥n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);
 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 execution-time rounding to +infinity = 0.33333333333333338
1/3 with execution-time rounding to -infinity = 0.33333333333333337
```

最初の計算では、戻された値は大きい値 0.33333333333333337 に丸められていますが、 printf 関数は再度大きい値に丸めて 0.33333333333333338 を出力しています。 2 番目の計算で使用されたこの問題の解決策は、ライブラリー関数の呼び出しの直前に丸めモードを最も近い値にリセットすることです。

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。

「*XL C*コンパイラー・リファレンス」の関連情報

 -qfloat

 -qflttrap

10 進浮動小数点プログラムのコンパイル

10 進浮動小数点フォーマットをプログラムで使用する場合、コンパイルに **-qdfp** オプションを使用します。例えば、次の Hello World プログラム dfp_hello.c をコンパイルする場合、コンパイラー呼び出しは以下のようになります。

```
xlc -qdfp dfp_hello.c
```

```
#include <stdio.h>
#include <float.h>
int main() {
   printf("Hello DFP World\n");
   printf("DEC32_MAX = %Hf\n",DEC32_MAX);
   float f = 12.34df;
   printf("12.34df as a float = %f\n",f);
 }
```

「*XL C*コンパイラー・リファレンス」の関連情報

 **-qdfp**

第 5 章 メモリー・ヒープの使用

ANSI によって定義されたメモリー管理関数に加えて、XL C はプログラム・パフォーマンスとプログラムのデバッグの改善を支援するメモリー管理関数の拡張バージョンを提供します。これらの関数によって下記の項目が可能になります。

- ユーザー作成ヒープとして知られる、複数の、カスタム定義のメモリーのプールから、メモリーを割り振る。
- デフォルトのランタイム・ヒープ内でメモリー問題のデバッグを行う。
- ユーザー作成ヒープ内でメモリー問題のデバッグを行う。

メモリー管理関数のすべてのバージョンは同じ方法で処理します。異なるところは、どのヒープから割り当てられるか、およびメモリー問題をデバッグするのに役に立つ情報を保管するかどうかだけです。これらのすべての関数によって割り当てられたメモリーは、どのタイプのオブジェクトを保管する場合にも適切に位置合わせされます。

『複数のヒープを持つメモリーの管理』では、複数のユーザー作成ヒープの利点を説明し、ユーザー作成ヒープを管理するための使用可能な関数をまとめ、ユーザー定義ヒープの作成、拡張、使用、そして破棄する手順を示し、そして正規メモリーと共用メモリーの両方を使ってヒープを作成するプログラムの例を提供しています。

40 ページの『メモリー・ヒープのデバッグ』では、デフォルト・ヒープおよびユーザー作成ヒープのチェックおよびデバッグに使用可能な関数を説明しています。

複数のヒープを持つメモリーの管理

XL C を使用することにより、デフォルトの XL C ランタイム・ヒープの代わり、または追加として、ユーザー独自のメモリー・ヒープを作成して操作することができます。

正規メモリーや共用メモリーのヒープを作成できます。また、どんなタイプのヒープでも任意の数だけ持つことができます。唯一の制限としては、オペレーティング・システム上での使用可能なスペース (マシンのメモリーとスワッパーのサイズから、他の稼働中のアプリケーションに必要なメモリーを引いたもの) があるのみです。また、デフォルトのランタイム・ヒープを、ユーザー作成の 1 つのヒープに変更することもできます。

ユーザー独自のヒープを使用するかどうかは任意です。アプリケーションは、XL C ランタイム・ライブラリーが提供 (および使用) するデフォルトのメモリー管理を使用して問題なく動作します。ただし、複数のヒープを使用すればより効果的であり、次のようないろいろな理由で、プログラムのパフォーマンスが向上し、メモリーの無駄を削減することができます。

- 単一のヒープから割り当てを行うと、メモリー・ブロックがメモリーの異なるページに分散してしまう場合があります。例えば、リストにノードを追加するたびに、メモリーを割り当てるようなリンク・リストがあるとします。ノードを追加

している間に、メモリーを別のデータ用に割り当てると、ノード用のメモリー・ブロックが多くの異なるページに分散してしまう可能性があります。リスト内のデータにアクセスするには、システムは多くのページをスワップしなければならないため、プログラムのスピードはかなり低下します。

複数のヒープを使用すれば、どのヒープから割り当てるかを指定することができます。例えば、リンク・リスト用に特定したヒープを作成することができます。リストのメモリー・ブロック、およびそれらに含まれるデータは、少数のページにかたまって存在し、必要なスワップの量を減少させます。

- マルチスレッドのアプリケーションでは、確実にメモリーが安全に割り当てられ、そして解放されるように、一度に 1 つのスレッドのみがヒープにアクセスできます。例えば、スレッド 1 はメモリーを割り当て中であり、スレッド 2 は 解放する ための呼び出しを行っているとします。スレッド 2 がヒープにアクセスするにはその前に、スレッド 1 が割り当てを終了するのを待たなければなりません。また、このことによって、特にユーザー・プログラムが多くのメモリー操作を行う場合は、パフォーマンスが低下します。

  各スレッドごとに別個のヒープを作成した場合には、それらのヒープから同時に割り当てることができ、ヒープへのアクセスを逐次化するために必要な待ち時間とオーバーヘッドの両方をなくすことができます。
- 単一のヒープを使用した場合、割り当てる各ブロックを明示的に解放しなければなりません。各ノードにメモリーを割り当てるリンク・リストがある場合には、リスト全体を通して、各ブロックを個々に解放しなければならないために時間がかかります。

  そのリンク・リスト用に別個のヒープを作成した場合には、そのヒープを単一の呼び出しで破棄でき、一度にすべてのメモリーを解放できます。
- ヒープが 1 つだけの場合は、すべてのコンポーネント (XL C ランタイム・ライブラリー、ベンダー・ライブラリー、およびユーザー独自のコード) が、そのヒープを共用します。 1 つのコンポーネントがヒープを破壊した場合は、別のコンポーネントが失敗することがあります。このため、問題の原因やヒープの損傷箇所を突き止めるのがめんどうになります。

  複数のヒープを使用すれば、各コンポーネントに別個のヒープを作成することができます。そのため、いずれか 1 つがヒープを損傷させても (例えば、解放されたポインターを使用して)、その他のコンポーネントは影響を受けずに継続することができます。また、問題を訂正するためにどこを調べればよいかもわかります。

ユーザー作成ヒープを管理する関数

`libhu.a` ライブラリーは、ユーザー作成ヒープの管理を可能にする関数セットを提供しています。これらの関数はすべて `_u` ("user" ヒープを示す) によるプレフィックスがあり、これらはヘッダー・ファイル `umalloc.h` 内で宣言されています。下記の表では、ユーザー定義ヒープの作成および管理のために使用可能な関数がまとめられています。

表 12. メモリー・ヒープを管理する関数

デフォルトのヒープ関数	対応するユーザー作成ヒープ関数	説明
適用外	_ucreate	ヒープを作成する。 『ヒープの作成』を参照してください。
適用外	_uopen	プロセスが使用するヒープをオープンする。 32 ページの『ヒープの使用』を参照してください。
適用外	_ustats	ヒープに関する情報を提供する。 33 ページの『ヒープに関する情報の獲得』を参照してください。
適用外	_uaddmem	メモリー・ブロックをヒープに追加する。 31 ページの『ヒープの拡張』を参照してください。
適用外	_uclose	プロセスによるこれ以上のヒープの使用をクローズする。 33 ページの『ヒープのクローズおよび破棄』を参照してください。
適用外	_udestroy	ヒープを破棄する。 33 ページの『ヒープのクローズおよび破棄』を参照してください。
calloc	_ucalloc	作成されたヒープからメモリーを割り振って初期化する。 32 ページの『ヒープの使用』を参照してください。
malloc	_umalloc	作成されたヒープからメモリーを割り振る。 32 ページの『ヒープの使用』を参照してください。
_heapmin	_uheapmin	未使用のメモリーをシステムに戻す。 33 ページの『ヒープのクローズおよび破棄』を参照してください。
適用外	_udefault	デフォルトのランタイム・ヒープを、ユーザー作成ヒープに変更する。 34 ページの『プログラムで使用されるデフォルト・ヒープの変更』を参照してください。

注: `realloc` または `free` には、ユーザー作成ヒープのバージョンはありません。これらの標準関数は、どのヒープからメモリーを割り当てるかを常に判別し、ユーザー作成ヒープとデフォルト・メモリー・ヒープの両方で使用できます。

ヒープの作成

固定サイズ・ヒープ、または動的サイズ変更ヒープ (dynamically-sized heap) を作成できます。固定サイズ・ヒープを使用するとき、メモリーの初期ブロックは、この初期ブロックに対して行われるすべての割り当て要求を満たすのに十分な大きさが必要です。動的サイズ変更ヒープ (dynamically-sized heap) により、プログラムの要求に応じて、ヒープを拡張および縮小することが可能です。

固定サイズ・ヒープの作成

固定サイズ・ヒープを作成する時は、そのヒープおよびヒープの管理に必要な内部情報を保持して、それにハンドルを割り当てるために十分な大きさのメモリーのブロックをあらかじめ割り当てておく必要があります。 次に例を示します。

```
Heap_t fixedHeap;    /* this is the "heap handle" */
/* get memory for internal info plus 5000 bytes for the heap */
static char block[_HEAP_MIN_SIZE + 5000];
```

内部情報には、_HEAP_MIN_SIZE マクロ (umalloc.h に定義されている) で指定されている最小のバイト・セットが必要です。必要なブロックのサイズを判別するには、プログラムで必要とされるメモリー量をこの値に加算してください。ブロックが一度満杯になるまで割り振られてしまうと、そのヒープに対する以降の割り当て要求は失敗します。

メモリーのブロックを割り当てた後で、_ucreate を使ってヒープを作成し、ヒープのメモリー・タイプ、正規メモリーまたは共用メモリーを指定します。次に例を示します。

```
fixedHeap = _ucreate(block, (_HEAP_MIN_SIZE+5000),  /* block to use */
                     !_BLOCK_CLEAN,   /* memory is not set to 0   */
                     _HEAP_REGULAR,   /* regular memory           */
                     NULL, NULL);     /* functions for expanding and shrinking
                                         a dynamically-sized heap */
```

!_BLOCK_CLEAN パラメーターは、ブロック内のメモリーが 0 に初期化されていないことを示します。これが 0 に設定されていた (例えば、memset を使って) 場合には、_BLOCK_CLEAN を指定します。 calloc および _ucalloc 関数は、この情報を使用して関数の効率を高めます。すなわち、メモリーがすでに 0 に初期化されている場合には、関数でメモリーを初期化する必要はありません。

4 番目のパラメーターは、ヒープに含まれるメモリーのタイプが正規 (_HEAP_REGULAR) であるか、または共用 (_HEAP_SHARED) であるかを示します。

正規メモリーには _HEAP_REGULAR を使用します。大半のプログラムは正規メモリーを使用します。このタイプは、デフォルトのランタイム・ヒープによって提供されます。共用メモリーには _HEAP_SHARED を使用します。共用メモリーのヒープは、プロセス間またはアプリケーション間で共有できます。

固定サイズ・ヒープの場合、最後の 2 つのパラメーターは常に NULL です。

動的サイズ変更ヒープ (dynamically-sized heap) の作成

XL C デフォルト・ヒープでは、malloc 要求で十分なストレージが使用できないときには、ランタイム環境はシステムから追加ストレージを取得します。同様に、_heapmin を使ってヒープを最小化したとき、またはプログラムが終了したときには、そのランタイム環境はオペレーティング・システムにメモリーを戻します。

拡張可能ヒープを作成するときは、この作業を行うために、ユーザーが選択するいずれかの名前でユーザー独自の関数を指定します。これらの関数に対するポインターを、_ucreate への最後の 2 つのパラメーターとして (固定サイズ・ヒープを作成するために使用した NULL ポインターの代わりに) 指定します。 次に例を示します。

```
Heap_t growHeap;
static char block[_HEAP_MIN_SIZE];  /* get block */

growHeap = _ucreate(block, _HEAP_MIN_SIZE,   /* starting block */
                    !_BLOCK_CLEAN,      /* memory not set to 0 */
```

```
_HEAP_REGULAR,      /* regular memory      */
expandHeap,     /* function to expand heap */
shrinkHeap);    /* function to shrink heap */
```

注: ヒープが同じタイプのメモリーを使用し、関数が 1 つのヒープ用に特定して作成されたものでない限り、複数のヒープに対して同一の拡張および縮小関数を使用することができます。

ヒープの拡張

ヒープのサイズを大きくするには、下記の関数を使ってメモリー・ブロックを追加します。

- 固定サイズ・ヒープ、または動的サイズ変更ヒープ (dynamically-sized heaps) については、_uaddmem 関数を呼び出します。
- 動的サイズ変更ヒープ (dynamically-sized heaps) のみについては、ヒープを拡張する関数の書き込みは、ヒープからメモリーを割り振る時はいつでも必要に応じてシステムが自動的に呼び出します。

メモリー・ブロックのヒープへの追加

_uaddmem を使用して、メモリー・ブロックを固定サイズ・ヒープまたは動的サイズ変更ヒープ (dynamically-sized heap) に追加することができます。条件に応じて割り当てられる大きなメモリーがある場合に、この関数は役立ちます。開始ブロックと同じように、まず、メモリー・ブロック用のメモリーを割り当てておく必要があります。このブロックは現行のヒープに追加されます。そのため、追加するブロックが、このブロックの追加先のヒープと同じタイプのメモリーであることを確認する必要があります。例えば、fixedHeap に 64K を追加するには、次のようにします。

```
static char newblock[65536];

_uaddmem(fixedHeap,       /* heap to add to   */
         newblock, 65536, /* block to add     */
         _BLOCK_CLEAN);   /* sets memory to 0 */
```

注: 追加するそれぞれのメモリー・ブロックごとに、内部情報を保管するために、このブロックから数バイトが使用されます。オーバーヘッドの合計を減らすためには、小さなブロックをたくさん追加するよりも、大きなメモリーのブロックを少なく追加することをお勧めします。

ヒープ拡張関数の書き込み

動的サイズ変更ヒープ (dynamically-sized heap) のために _umalloc (または類似の関数) を呼び出す時は、 _umalloc は _ucreate に提供されている初期ブロックからメモリーの割り振りを試みます。初期ブロックに十分なメモリーがない場合、この関数は _ucreate のパラメーターとして指定されたヒープ拡張関数を呼び出します。その間数は次にオペレーティング・システムからさらにメモリーを取得し、そのメモリーをヒープに追加します。これをどのような方法で行うかは、ユーザー次第です。

そのユーザーの関数には、次のプロトタイプが必要です。

```
void *(*functionName)(Heap_t uh, size_t *size, int *clean);
```

functionName がその関数 (任意の名前を付けることが可能) を識別する場合、 *uh* が拡張されるヒープであり、*size* が _umalloc によって渡される割り振り要求のサイズです。ユーザーは、数回の割り振りに足るような十分なメモリーを一度に戻すほうが良いでしょう。そうしないとその後の割り振りのたびごとにヒープ拡張関数を呼び出すことになってプログラムの速度を低下させてしまいます。要求した以上の *size* を戻す場合には、必ず *size* パラメーターを更新してください。

ユーザー関数では、*_BLOCK_CLEAN* (メモリーが 0 に設定されていることを示す) または !_BLOCK_CLEAN (メモリーが初期設定されていないことを示す) のいずれかに clean パラメーターを設定することも必要です。

以下のフラグメントにはヒープ拡張関数の例が示されています。

```
static void *expandHeap(Heap_t uh, size_t *length, int *clean)
{
   char *newblock;
   /* round the size up to a multiple of 64K *  /
   *length = (*length / 65536) * 65536 + 65536;

   *clean = _BLOCK_CLEAN;  /* mark the block as "clean" */
   return(newblock);       /* return new memory block   */
}
```

ヒープの使用

一度ヒープを作成したら、_uopen を呼び出して、そのヒープをオープンして使用することができます。

```
 _uopen(fixedHeap);
```

これにより、その特定プロセスに対してヒープがオープンされます。ヒープが共用の場合は、そのヒープを使用する各プロセスがそれぞれ個別に _uopen を呼び出す必要があります。

次に、デフォルト・ヒープの場合と同様に、ユーザー固有のヒープからメモリーを割り当て、そして解放を行うことができます。メモリーを割り当てるには、_ucalloc または _umalloc を使用します。これらの関数は、必要なブロック・サイズだけでなく、使用するヒープを指定する点を除いて、calloc および malloc と同じように動作します。例えば、fixedHeap から 1000 バイトを割り当てるには、次のようにします。

```
void *up;
up = _umalloc(fixedHeap, 1000);
```

メモリーの再割り当ておよび解放を行うには、正規の realloc および free 関数を使用します。これらの関数は両方とも、どのヒープからメモリーを割り振るかを常にチェックしているため、使用するヒープをユーザーが指定する必要はありません。例えば、以下のコード・フラグメントにある realloc および free の呼び出しでは、デフォルト・ヒープとユーザー・ヒープのどちらに対してもまったく同じように見えます。

```
void *p, *up;
p = malloc(1000);   /* allocate 1000 bytes from default heap */
up = _umalloc(fixedHeap, 1000);  /* allocate 1000 from fixedHeap */

realloc(p, 2000);   /* reallocate from default heap */
realloc(up, 100);   /* reallocate from fixedHeap    */
```

```
free(p);            /* free memory back to default heap */
free(up);           /* free memory back to fixedHeap    */
```

ヒープ関数を呼び出すときは、指定したヒープが有効であることを確認してください。ヒープが無効の場合には、ヒープ関数の振る舞いは未定義となります。

ヒープに関する情報の獲得

ユーザーは、_mheap を呼び出すことにより、いずれかのオブジェクトを割り振った元のヒープを識別することができます。 _ustats を呼び出すことによって、ヒープそのものに関する情報を得ることもできます。この関数により、次のことがわかります。

- ヒープが保持するメモリーの大きさ (オーバーヘッドに使用されるメモリーは除く)
- ヒープから現在割り当てられているメモリーの大きさ
- ヒープにあるメモリーのタイプ
- ヒープから使用可能な最大の連続メモリー部分のサイズ

ヒープのクローズおよび破棄

ヒープの使用を終えたら、_uclose を使ってヒープをクローズします。プロセス内でヒープをいったんクローズすると、そのプロセスは、ヒープからメモリーを割り当てることも、そのヒープにメモリーを戻すこともできなくなります。ほかのプロセスがヒープを共用している場合には、各プロセスでヒープをクローズするまで、そのヒープを使い続けることができます。ヒープをクローズした後でそのヒープに関する操作を行うと、未定義な振る舞いを引き起こすことになります。

ヒープを破棄するには以下を行います。

- 固定サイズのヒープについては、_udestroy を呼び出します。 メモリーのブロックがまだどこかに割り当てられている場合は、強制的に破棄することができます。ヒープを破棄すると、ヒープがほかのプロセスで共用されていたとしても、そのヒープは完全に除去されます。また、ヒープを破棄した後で、そのヒープに関する操作を行うと、未定義な振る舞いを引き起こすことになります。
- 動的サイズ変更ヒープ (dynamically-sized heap) については、_uheapmin を呼び出してヒープを合体させる (ヒープ内のすべてのブロックを戻して、システムに対して完全に解放する)、または _udestroy を使用して破棄します。これら両方の関数は、ユーザーのヒープ縮小関数を呼び出します。 (以下を参照ください。)

ヒープを破棄した後で、そのヒープ用のメモリーを (_ucreate に指定した初期メモリー・ブロック、および _uaddmem で追加したほかのブロックを) システムに戻すかどうかはユーザー次第です。

ヒープ縮小関数の書き込み

動的サイズ変更ヒープ (dynamically-sized heap) を合体させるために _uheapmin を呼び出したり、またはヒープを破棄するために _udestroy を呼び出すと、これらの関数は、システムにメモリーを戻すためにヒープ縮小関数を呼び出します。この関数をどのような方法で実装するかは、ユーザー次第です。

そのユーザーの関数には、次のプロトタイプが必要です。

```
void (*functionName)(Heap_t uh, void *block, size_t size);
```

functionName がその関数 (任意の名前を付けることが可能) を識別する場合、 *uh* 縮小されるヒープを識別します。ポインター *block* およびその *size* が _uheapmin または _udestroy によってユーザーの関数に渡されます。ユーザー関数は *block* で指示されたメモリーをシステムに戻さなければなりません。次に例を示します。

```
static void shrinkHeap(Heap_t uh, void *block, size_t size)
{
  free(block);
  return;
}
```

プログラムで使用されるデフォルト・ヒープの変更

正規メモリー管理関数 (malloc など) は、そのスレッドのデフォルト・ヒープを常に使用します。すべての XL C アプリケーションの初期デフォルト・ヒープは、XL C が提供するランタイム・ヒープです。ただし、_udefault を呼び出して、ユーザー独自のヒープをデフォルトにすることができます。そうすると、正規メモリー管理関数のすべての呼び出しは、デフォルトのランタイム・ヒープからではなく、ユーザー・ヒープから割り振ります。

デフォルト・ヒープは、_udefault を呼び出すスレッドに対してのみ変更します。プログラムの各スレッドごとに異なるデフォルト・ヒープを使用するよう選択することができます。これは、XL C デフォルト・ヒープ以外のヒープを使用するコンポーネント (ベンダー・ライブラリーなど) が必要なときに役立ちますが、ヒープ固有の呼び出しを使うためにソース・コードを実際に変更することはできません。例えば、デフォルト・ヒープを共用ヒープに設定してから、malloc を呼び出すライブラリー関数を呼び出した場合、ライブラリーはストレージを共用メモリーに割り当てます。

_udefault は現行のデフォルト・ヒープを戻すので、この戻り値を保管し、置き換えたデフォルト・ヒープを復元するのにこの戻り値を後で使用することができます。 _udefault を呼び出し、RUNTIME_HEAP マクロ (umalloc.h で定義) を指定することにより、デフォルトを XL C デフォルト・ランタイム・ヒープに戻すこともできます。また、任意のヒープ固有関数でこのマクロを使用し、デフォルト・ランタイム・ヒープから明示的に割り振ることもできます。

ユーザー作成ヒープ使用のプログラムのコンパイルとリンク

いずれかのユーザー作成ヒープ関数 (_u のプレフィックス付き) を呼び出すアプリケーションをコンパイルするために hu を -l リンカー・オプションに指定します。例えば、デフォルト・ディレクトリーに libhu.a ライブラリーがインストールされている場合は、以下のように指定できます。

```
xlc progc.c -o progf -lhu
```

ユーザー・ヒープの作成と使用の例

正規メモリーを持つユーザー・ヒープの例

以下のプログラムでは、正規メモリーを使用するヒープの作成および使用方法を示します。

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
   void *p;
   /* Round up to the next chunk size */
   *length = ((*length) / 65536) * 65536 + 65536;
   *clean = _BLOCK_CLEAN;
    p = calloc(*length,1);
    return (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
   free( p );
   return;
}

int main(void)
{
   void    *initial_block;
   long   rc;
   Heap_t  myheap;
   char    *ptr;
   int     initial_sz;

   /* Get initial area to start heap */
   initial_sz = 65536;
   initial_block = malloc(initial_sz);
   if(initial_block == NULL) return (1);

   /* create a user heap */
   myheap = _ucreate(initial_block, initial_sz, _BLOCK_CLEAN,
                     _HEAP_REGULAR, get_fn, release_fn);
   if (myheap == NULL) return(2);


   /* allocate from user heap and cause it to grow */
   ptr = _umalloc(myheap, 100000);
   _ufree(ptr);

   /* destroy user heap */
   if (_udestroy(myheap, _FORCE)) return(3);

   /* return initial block used to create heap */

   free(initial_block);
   return 0;
}
```

共用ユーザー・ヒープの例 - 親プロセス

次のプログラムでは、1 つの親プロセスと複数の子プロセスの間で共用されるヒープをインプリメントする方法を示します。このプログラムでは、共用ヒープを作成する親プロセスを示しています。最初に、メインプログラムがオペレーティング・システムから (CreateFileMapping を使用して) 共用メモリーを割り振るために init 関数を呼び出し、そして別のプロセスが名前で使用できるようにメモリーに名前を付けます。それから、init 関数がヒープを作成してオープンします。メインプログラム内のループはヒープ上のプログラムを実行し、また他のプロセスも開始します。それからプログラムは term 関数を呼び出してヒープをクローズして破棄します。

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

#define PAGING_FILE  0xFFFFFFFF
#define MEMORY_SIZE  65536
#define BASE_MEM     (VOID*)0x01000000

static HANDLE hFile;       /* Handle to memory file                      */
static void*  hMap;        /* Handle to allocated memory                 */

typedef struct mem_info {
   void  * pBase;
   Heap_t  pHeap;
} MEM_INFO_T;

 /*--------------------------------------------------------------------------*/
 /* inithp:                                                                  */
 /* Function to create and open the heap with a named shared memory object */
 /*--------------------------------------------------------------------------*/
static Heap_t inithp(size_t heap_size)
{
   MEM_INFO_T info;        /* Info structure               */

   /* Allocate shared memory from the system by creating a shared memory   */
   /* pool basing it out of the system paging (swapper) file.              */

   hFile = CreateFileMapping( (HANDLE) PAGING_FILE, NULL, PAGE_READWRITE, 0,
                              heap_size + sizeof(Heap_t), "MYNAME_SHAREMEM" );
   if (hFile == NULL) {
     return NULL;
   }

   /* Map the file to this process' address space, starting at an address  */
   /* that should also be available in child processe(s)                    */

   hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, BASE_MEM );

   info.pBase = hMap;
   if (info.pBase == NULL) {
     return NULL;
   }

   /* Create a fixed sized heap.  Put the heap handle as well as the      */
   /* base heap address at the beginning of the shared memory.            */

   info.pHeap = _ucreate((char *)info.pBase + sizeof(info), heap_size - sizeof(info),
                !_BLOCK_CLEAN, _HEAP_SHARED | _HEAP_REGULAR, NULL, NULL);

   if (info.pBase == NULL) {
     return NULL;
   }

   memcpy(info.pBase, info, sizeof(info));

   if (_uopen(info.pHeap)) {      /* Open heap and check result           */
     return NULL;
   }

   return info.pHeap;

}

/*--------------------------------------------------------------------------*/
/* termhp:                                                                  */
/* Function to close and destroy the heap                                   */
```

```
/*--------------------------------------------------------------------------*/
static int termhp(Heap_t uheap)
{
   if (_uclose(uheap))                      /* close heap                */
      return 1;
   if (_udestroy(uheap, _FORCE))            /* force destruction of heap */
      return 1;

   UnmapViewOfFile(hMap);                   /* return memory to system   */
   CloseHandle(hFile);

   return 0;
}

/*--------------------------------------------------------------------------*/
/* main:                                                                    */
/* Main function to test creating, writing to and destroying a shared       */
/* heap.                                                                    */
/*--------------------------------------------------------------------------*/
int main(void)
{
   int i, rc;                           /* Index and return code         */
   Heap_t uheap;                        /* heap to create                */
   char *p;                             /* for allocating from heap      */

   /*                                                                      */
   /* call init function to create and open the heap                       */

   uheap = inithp(MEMORY_SIZE);
   if (uheap == NULL)                   /* check for success             */
      return 1;                         /* if failure, return non zero   */

   /*                                                                      */
   /* perform operations on uheap                                          */
   /*                                                                      */
   for (i = 1; i <= 5; i++)
   {
      p = _umalloc(uheap, 10);          /* allocate from uheap           */
      if (p == NULL)
         return 1;
      memset(p, 'M', _msize(p));        /* set all bytes in p to 'M'     */
      p = realloc(p,50);                /* reallocate from uheap         */
      if (p == NULL)
        return 1;
      memset(p, 'R', _msize(p));        /* set all bytes in p to 'R'     */
   }

   /*                                                                      */
   /* Start a second process which accesses the heap                       */
   /*                                                                      */
   if (system("memshr2.exe"))
     return 1;

   /*                                                                      */
   /* Take a look at the memory that we just wrote to. Note that memshr.c */
   /* and memshr2.c should have been compiled specifying the               */
   /* alloc(debug[, yes]) flag.                                            */
   /*                                                                      */
    #ifdef DEBUG
      _udump_allocated(uheap, -1);
    #endif

   /*                                                                       */
   /* call term function to close and destroy the heap                      */
   /*                                                                       */
   rc = termhp(uheap);
```

```
   #ifdef DEBUG
     printf("memshr ending... rc = %d\n", rc);
   #endif

   return rc;
}
```

共用ユーザー・ヒープの例 - 子プロセス

以下のプログラムは、親プロセス内のループによって開始されるプロセスを示します。このプロセスは OpenFileMapping を使用して共用メモリーに名前でアクセスし、それから親プロセスで作成されたヒープのヒープ・ハンドルを抽出します。そこでプロセスはヒープをオープンしてそれをデフォルト・ヒープにして、ループ内で何らかの操作を実行します。ループの後、プロセスは古いデフォルト・ヒープを置換してユーザー・ヒープをクローズし、終了します。

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

static HANDLE hFile;       /* Handle to memory file                       */
static void*  hMap;        /* Handle to allocated memory                  */

typedef struct mem_info {
   void  * pBase;
   Heap_t  pHeap;
} MEM_INFO_T;

/*---------------------------------------------------------------------------*/
/* inithp:  Subprocess Version                                               */
/* Function to create and open the heap with a named shared memory object */
/*---------------------------------------------------------------------------*/
static Heap_t inithp(void)
{

   MEM_INFO_T info;                       /* Info structure               */

   /* Open the shared memory file by name.  The file is based on the      */
   /* system paging (swapper) file.                                       */

   hFile = OpenFileMapping(FILE_MAP_WRITE, FALSE, "MYNAME_SHAREMEM");

   if (hFile == NULL) {
     return NULL;
   }

   /* Figure out where to map this file by looking at the address in the   */
   /* shared memory where the memory was mapped in the parent process.     */

   hMap = MapViewOfFile( hFile, FILE_MAP_WRITE, 0, 0, sizeof(info) );

   if (hMap == NULL) {
     return NULL;
   }

   /* Extract the heap and base memory address from shared memory          */

   memcpy(info, hMap, sizeof(info));
   UnmapViewOfFile(hMap);

   hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, info.pBase );

   if (_uopen(info.pHeap)) {          /* Open heap and check result       */
      return NULL;
```

```
   }

   return info.pHeap;
}

 /*--------------------------------------------------------------------------*/
 /* termhp:                                                                  */
 /* Function to close my view of the heap                                    */
 /*--------------------------------------------------------------------------*/
 static int termhp(Heap_t uheap)
 {
    if (_uclose(uheap))                      /* close heap               */
       return 1;

    UnmapViewOfFile(hMap);                   /* return memory to system  */
    CloseHandle(hFile);

    return 0;
 }

/*--------------------------------------------------------------------------*/
/* main:                                                                    */
/* Main function to test creating, writing to and destroying a shared     */
/* heap.                                                                    */
/*--------------------------------------------------------------------------*/
int main(void)
{
   int rc, i;                     /* for return code, loop iteration    */
   Heap_t uheap, oldheap;         /* heap to create, old default heap   */
   char *p;                       /* for allocating from the heap       */

   /*                                                                      */
   /* Get the heap storage from the shared memory                          */
   /*                                                                      */
   uheap = inithp();
   if (uheap == NULL)
     return 1;

   /*                                                                      */
   /* Register uheap as default runtime heap, save old default             */
   /*                                                                      */
   oldheap = _udefault(uheap);
   if (oldheap == NULL) {
      return termhp(uheap);
}

   /*                                                                      */
   /* Perform operations on uheap                                          */
   /*                                                                      */
   for (i = 1; i <= 5; i++)
   {
      p = malloc(10);     /* malloc uses default heap, which is now uheap*/
      memset(p, 'M', _msize(p));
   }

   /*                                                                      */
   /* Replace original default heap and check result                       */
   /*                                                                      */
   if (uheap != _udefault(oldheap)) {
      return termhp(uheap);
}

   /*                                                                      */
   /* Close my views of the heap                                           */
   /*                                                                      */
   rc = termhp(uheap);
```

```
  #ifdef DEBUG
    printf("Returning from memshr2 rc = %d\n", rc);
  #endif
  return rc;

}
```

メモリー・ヒープのデバッグ

XL C では、メモリーの問題をデバッグするために、次の 2 つの関数セットが用意されています。

- 他のコンパイラーが提供するものに類似したヒープ・チェック関数。 (41 ページの『メモリー・ヒープをチェックする関数』を参照してください。)
- すべてのメモリー管理関数のデバッグ・バージョン。 (41 ページの『メモリー・ヒープをデバッグする関数』を参照してください。)

どちらのデバッグ関数セットにも利点と欠点があります。どちらを選択するかは、ユーザーのプログラム、問題、好みによって異なります。

一方、ヒープ・チェック関数は、プログラム内の特定のポイントで、より一般的なチェックをヒープに対して行います。どこでチェックを行うかをかなり柔軟に制御することができます。また、ヒープ・チェック関数には、これらの関数を提供する他のコンパイラーとの互換性もあります。さらに、ヒープ・チェック呼び出しが含まれているモジュールのみを再ビルドするだけで済みます。ただし、ソース・コードを変更して、最終コードではおそらく除去することになる、これらの呼び出しを組み込む必要があります。また、ヒープ・チェック関数は、ヒープに整合性があるかどうかを知らせるだけであり、デバッグ・メモリー管理関数によって提供されるような詳細は提供しません。

一方で、デバッグ・メモリー管理関数は、プログラム内で行ったすべての割り振り要求に関する詳細情報を提供します。デバッグ・バージョンを使用するためにプログラム・コードを変更する必要はありません。 **-qheapdebug** オプションを指定することだけが必要です。

お勧めするアプローチは、メモリーの問題の可能性があると思われるところにヒープ・チェック関数の呼び出しを追加することです。ヒープが破壊されていることがわかった場合には、**-qheapdebug** を指定して再ビルドすることができます。

どのデバッグ関数を選択しても、プログラムには、これらの関数の内部情報を保持するための追加のメモリーが必要です。固定サイズのヒープを使用している場合は、デバッグ関数を使用するために、場合によってはヒープのサイズを増やす必要があります。

関連情報

107 ページの『第 12 章 メモリー・デバッグ・ライブラリー関数』

「*XL C コンパイラー・リファレンス*」**の関連情報**

-qheapdebug

メモリー・ヒープをチェックする関数

ヘッダー・ファイル umalloc.h は、ユーザー作成ヒープの妥当性検査をする関数セットを宣言します。これらの関数はコンパイラー・オプションで制御されることがないため、いつでもプログラム内で使用することができます。 _u プレフィックスのない、これらの関数の正規バージョンも、デフォルト・ヒープのチェックのために使用可能です。ヒープ・チェック関数は下記の表にまとめられています。

表 13. メモリー・ヒープをチェックする関数

デフォルトのヒープ関数	ユーザー作成ヒープ関数	説明
_heapchk	_uheapchk	ヒープ全体を最小整合性についてチェックする。
_heapset	_uheapset	ヒープ内のフリー・メモリーの最小整合性についてチェックして、ヒープ内のフリー・メモリーをユーザー指定の値に設定する。
_heap_walk	_uheap_walk	ヒープを通して、割り振られたか、または解放された各オブジェクトの情報をユーザー提供のコールバック関数に提供します。

ユーザー作成ヒープ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、 34 ページの『ユーザー作成ヒープ使用のプログラムのコンパイルとリンク』 を参照してください。

メモリー・ヒープをデバッグする関数

正規メモリー管理関数およびユーザー定義のヒープ・メモリー管理関数の両方のためのデバッグ・バージョンが使用可能です。各デバッグ・バージョンは、対応する非デバッグ・バージョンと同じ機能を実行します。ユーザーは、共用メモリー・ヒープを含むどのタイプのヒープに対しても使用可能です。各々のデバッグ関数の呼び出しも、_heap_check (以下で説明) を呼び出して自動的にヒープをチェックし、メモリー問題のデバッグに使用できる、ファイル名、行番号を含む情報を提供します。 ユーザー定義のデバッグ・バージョンの名前には _debug_u というプレフィックスが付き (例えば、_debug_umalloc)、そして umalloc.h 内に定義されます。

すべてのデバッグのメモリー管理関数の完全リストと詳細については、メモリー・デバッグ・ライブラリー関数を参照してください。

表 14. メモリー・ヒープをデバッグする関数

デフォルトのヒープ関数	対応するユーザー作成ヒープ関数
_debug_calloc	_debug_ucalloc
_debug_malloc	_debug_umalloc
_debug_heapmin	_debug_uheapmin
_debug_realloc	適用外
_debug_free	適用外

これらのデバッグ・バージョンを使用するには、以下の手順のいずれかを行います。

- ソース・コード内で、デフォルト、またはユーザー定義のヒープ・メモリー管理関数のいずれかに、プレフィックス _debug_ を付加します。
- ソース・コードに変更を加えたくない場合は、単純に -qheapdebug オプション付きでコンパイルしてください。このオプションは、すべてのメモリー管理関数の呼び出しを、それらのデバッグ・バージョンのカウンター・パートにマップします。呼び出しがマップされることを避けるには、関数名に括弧を付けます。

ユーザー作成ヒープ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、34 ページの『ユーザー作成ヒープ使用のプログラムのコンパイルとリンク』 を参照してください。

注:

1. **-qheapdebug** オプションを指定すると、すべての関数のローカル変数を*事前初期化*するためのコードが生成されます。これにより、初期化されていないローカル変数が通常のデバッグ・サイクル中に見つかる可能性が、それよりずっと後 (通常はコードが最適化されるとき) に見つかる可能性よりも大きくなります。
2. **-brtl** オプションは **-qheapdebug** と一緒には使用しないでください。
3. **#pragma strings** (readonly) ディレクティブは、デバッグ関数を呼び出す各ソース・ファイルの先頭、または各ソース・ファイルが組み込む共通のヘッダー・ファイル内に置いてください。このディレクティブは必須ではありませんが、このディレクティブによって、デバッグ関数へ受け渡すファイル名が上書きされないことと、オブジェクト・モジュールに組み込まれるファイル名ストリングのコピーが 1 つのみであることが保証されます。

メモリー・ヒープをデバッグする追加の関数

次の 3 つの追加のデバッグ・メモリー管理関数には、対応する正規の関数はありません。それらは下記の表にまとめられています。

表 15. メモリー・ヒープをデバッグする追加の関数

デフォルトのヒープ関数	対応するユーザー作成ヒープ関数	説明
_dump_allocated	_udump_allocated	デバッグ関数によって現在割り当てられている各メモリー・ブロックに関する情報を stderr に印刷します。
_dump_allocated_delta	_udump_allocated_delta	_dump_allocated または _dump_allocated_delta の最後の呼び出し以降に、デバッグ関数によって割り当てられた各メモリー・ブロックに関する情報をファイル記述子 2 に印刷します。
_heap_check	_uheap_check	デバッグ関数によって割り当てられた、または解放されたすべてのメモリー・ブロックをチェックして、割り当てられたブロックの境界の外部、または空きメモリー・ブロック内に上書きが発生していないことを確認します。

_heap_check 関数はデバッグ関数によって自動的に呼び出されます。ユーザーは明示的にこの関数を呼び出すこともできます。 次に、現在割り当てられているメモリー・ブロックの情報を表示するため、_dump_allocated または _dump_allocated_delta を使用することができます。これらの関数は明示的に呼び出す必要があります。

関連情報

107 ページの『第 12 章 メモリー・デバッグ・ライブラリー関数』

「*XL C*コンパイラー・リファレンス」**の関連情報**


-brtl


-qheapdebug


-qro / #pragma strings

メモリー割り振り充てんパターンの使用

デバッグ関数の中には、割り当てるメモリーすべてを、指定した充てんパターンに設定するものがあります。これにより、プログラムで使用するメモリー内の領域を簡単に配置することができます。

debug_malloc、debug_realloc、および debug_umalloc 関数は、割り振ったメモリーをデフォルトの繰り返し 0xAA 充てんパターンに設定します。この充てんパターンを使用可能にするには、HD_FILL 環境変数をエクスポートします。

debug_free 関数は、すべての空きメモリーを繰り返し 0xFB 充てんパターンに設定します。

ヒープ・チェックのスキップ

各デバッグ関数は、ヒープをチェックするのに _heap_check (または _uheap_check) を呼び出します。これは便利なものではありますが、これもまたプログラムのメモリー要件を増やしたり、プログラムの実行速度を低下させるおそれがあります。

それぞれのデバッグ・メモリー管理関数でヒープをチェックするためのオーバーヘッドを減らすために、HD_SKIP 環境変数を使って、関数がヒープをチェックする頻度を制御することができます。アプリケーションが極端にメモリー集中型でない限り、ほとんどのアプリケーションはこの制御を行う必要はありません。

ほかの環境変数と同様に、HD_SKIP を設定してください。HD_SKIP の構文は次のとおりです。

```
set HD_SKIP=increment,[start]
```

ここで、

increment
: ヒープ・チェックを行う間にデバッグ関数呼び出しをスキップする回数を指定する。

start
: ヒープ・チェックを開始する前にデバッグ関数呼び出しをスキップする回数を指定する。

注: パラメーターを分離するコンマの指定はオプションです。

例えば、次を指定した場合、

```
set HD_SKIP=10
```

デバッグ・メモリー関数を 10 回呼び出すごとに、ヒープ・チェックを 1 回行います。次を指定した場合、

```
set HD_SKIP=5,100
```

デバッグ・メモリー関数を 100 回呼び出した後で、5 回の呼び出しごとにだけヒープ・チェックを行います。

ヒープ・チェックのスキップを開始するのに *start* パラメーターを使用する場合は、暗黙的に行われるヒープ・チェックとプログラム実行速度の間の取引となります。そのため、少ない増分 (例えば 5) から開始し、アプリケーションが使用可能である範囲で、徐々にその数を増やすようにしてください。

スタック・トレースの使用

割り当てられたそれぞれのメモリー・オブジェクトごとに、スタックの内容がトレースされます。オブジェクトのスタックの内容が変更されると、トレースされた内容がダンプされます。

トレース・サイズは、HD_STACK 環境変数によって制御されます。この変数が設定されていない場合、コンパイラーはスタック・サイズを 10 と見なします。スタック・トレースを使用不可にするには、HD_STACK 環境変数を 0 に設定します。

第 6 章 ライブラリーの構成

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。

関連情報

『ライブラリーのコンパイルとリンク』

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

   ```
   xlc -c 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。次に例を示します。

   ```
   ar -rv libfoo.a bar.o example.o
   ```

共用ライブラリーのコンパイル

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

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

2. オプションとして、以下のいずれかを実行することにより、エクスポートするグローバル・シンボルをリストするエクスポート・ファイルを作成する。

   - 46 ページの『CreateExportList ユーティリティーによるシンボルのエクスポート』で説明されている **CreateExportList** ユーティリティーを使用する。
   - **-qmkshrobj** オプションとともに **-qexpfile** コンパイラー・オプションを使用して、実リンク・ステップで使用したエクスポート・ファイルの基本を作成する。 次に例を示します。

     ```
     xlc -qmkshrobj -qexpfile=exportlist foo.o
     ```

   - エクスポート・ファイルを手動で作成する。必要な場合は、エクスポート・ファイルをテキスト・エディターで編集して、共用ライブラリーを作成する際にエクスポートするシンボルを制御します。

3. ステップ 2 でエクスポート・ファイルを作成した場合は、 **-qmkshrobj** コンパイラー・オプションおよび **-bE** リンカー・オプションを使用して、望ましいオブジェクト・ファイルから共用ライブラリーを作成する。 **-bE** オプションを指定しない場合は、すべてのシンボルがエクスポートされます。次に例を示します。

   ```
   xlc -qmkshrobj foo.o -o mySharedObject -bE:exportlist
   ```

(**-o** オプションを使用して別の名前を指定しない限りは、共用オブジェクトのデフォルト名は `shr.o` です。)

4. オプションとして、AIX **ar** コマンドを使用して、複数の共用オブジェクトまたは静的オブジェクトからアーカイブ・ライブラリー・ファイルを作成する。次に例を示します。

   ```
   ar -rv libfoo.a shr.o anotherlibrary.so
   ```

5. 47 ページの『ライブラリーとアプリケーションとのリンク』で説明されているように、共用ライブラリーをメイン・アプリケーションにリンクする。

ランタイム・リンクを使用する共用ライブラリーを作成するには、次のようにします。

1. 上記で説明した手順のステップ 1 と 2 を実行する。
2. **-G** オプションを使用して、生成されたオブジェクト・ファイルから共用ライブラリーを作成し、ロード時にリンクし、**-bE** リンカー・オプションを使用して、エクスポート・リスト・ファイルの名前を指定する。 次に例を示します。

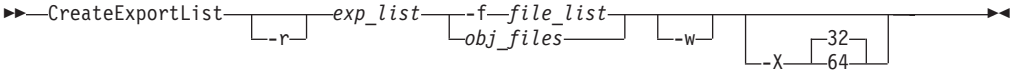
   ```
   xlc -G -o libfoo.so foo1.o foo2.o -bE:exportlist
   ```

3. 47 ページの『ライブラリーとアプリケーションとのリンク』で説明されているように、共用ライブラリーをメイン・アプリケーションにリンクする。

CreateExportList ユーティリティーによるシンボルのエクスポート

CreateExportList は、特定のオブジェクト・ファイル・セットにあるすべてのグローバル・シンボルのリストを含むファイルを作成するシェル・スクリプトです。このコマンドは、**-qexpfile** コマンドで代わりのエクスポート・ファイルを指定しない限り、**-qmkshrobj** オプションを使用するときに、自動的に実行されることに注意してください。

CreateExportList コマンドの構文は以下のとおりです。



以下のオプションの 1 つ以上を指定することができます。

-r これを指定すると、テンプレートの接頭部が除去されます。リソース・リストには、リソース・ファイル・シンボル (`__rsrc`) は追加されません。

exp_list
オブジェクト・ファイルで検出される、グローバル・シンボルのリストを含むファイルの名前です。このファイルは、**CreateExportList** コマンドが実行されるたびに上書きされます。

-f*file_list*
オブジェクト・ファイル名のリストを含むファイルの名前です。

obj_files
オブジェクト・ファイルの 1 つ以上の名前です。

-w エクスポート・リストから弱いシンボルを除外します。

-X32 -f *file_list* または *obj_files* で指定された入力リスト内の 32 ビット・オブジェクト・ファイルから名前を生成します。これはデフォルトです。

-X64 -f *file_list* または *obj_files* で指定された入力リスト内の 64 ビット・オブジェクト・ファイルから名前を生成します。

次のいずれかに該当する場合、**CreateExportList** コマンドは空のリストを作成します。

- **-f** *file_list* にも *obj_files* にもオブジェクト・ファイルが指定されていない。
- **-f** *file_list* パラメーターに指定されているファイルが空である。

関連外部情報

- 「*AIX コマンド・リファレンス 第 1 巻 から 第 6 巻*」の **ar** および **ld**

「*XL Cコンパイラー・リファレンス*」**の関連情報**

 -qexpfile

 -qmkshrobj

 -O、-qoptimize

 -G

 -brtl

ライブラリーとアプリケーションとのリンク

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

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

ライブラリーで実行時リンクを使用する場合は、次のように、このコマンドに **-brtl** オプションを追加します。

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

-l オプションを使用して、**-L** オプション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、AIX **ld** の資料を参照してください。

「*XL Cコンパイラー・リファレンス*」**の関連情報**

 -l

 -L

共用ライブラリー間のリンク

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

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

「*XL C*コンパイラー・リファレンス」の関連情報

 -qmkshrobj

 -L

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、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。
- オブジェクト・コードを再編成して、PowerPC アーキテクチャーの使用を最適化する。
- メモリー・サブシステムの使用率を改善する。
- このアーキテクチャーの能力を活用して大容量の共用メモリー並列化を扱う。

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。

チューニング

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。

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. 『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。このトピックでは、*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 16. 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。

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。

O2 での調整の開始

-O2 以上では、正しいハードウェア・アーキテクチャー・ターゲットまたはターゲットのファミリーを選択することが一層重要になります。適切なハードウェアをターゲットにすれば、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。

-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

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。

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。 XL コンパイラー最適化ポートフォリオには拡張最適化を指示するオプションが多数含まれており、アプリケーションが受ける変換はほとんど制御できます。 表 17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 17. 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

- 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。オプション **-qstrict=ieeefp** はまた、浮動小数点演算に関するすべての IEEE セマンティクスへの順守も確保します。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**、**-qstrict=exceptions**、または**-qstrict=order** でのコンパイルが正確な結果の確保に役立ちます。浮動小数点の計算精度に関して **-qstrict=precision** サブオプション・グループの影響を考慮する必要もあります。精度サブオプション・グループには個別サブオプションの、**subnormals**、**operationprecision**、**association**、**reductionorder**、および **library** があります (「*XL Cコンパイラー・リファレンス*」の **-qstrict** オプションに記載)。

-qstrict を指定しなければ、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。

-O レベルの構文については、「*XL Cコンパイラー・リファレンス*」の-O オプションを参照してください。

中間ステップ: レベル 3 での -qhot サブオプションの追加

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

レベル 4 での最適化

レベル 4 での利点

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

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

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

レベル 4 における潜在的なトレードオフ

-O3 で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。

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。

-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。これにより **-qarch=auto** がオーバーライドされます。

- **-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 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

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、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 18. 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。 -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、57 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。
-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、62 ページの『プロシージャー間分析の使用』を参照ください。
-qtune	指定したマイクロプロセッサー上で実行するように、最適化にバイアスをかけます。この際、ターゲットとして使用する命令セット・アーキテクチャーにはまったく影響は及びません。このオプションの詳しい情報については、57 ページの『ターゲット・マシンのオプションの最大活用』を参照ください。
-qcache	特定のキャッシュまたはメモリー形状を定義します。デフォルトは、-qtune の設定によって決まります。このオプションの詳しい情報については、57 ページの『ターゲット・マシンのオプションの最大活用』を参照ください。

ハードウェア関連の有効なサブオプションおよびサブオプションの組み合わせの完全なリストについては、「*XL C*コンパイラー・リファレンス」の **-qtune** セクションの『*受け入れ可能な -qarch/-qtune の組み合わせ*』、および「*XL C*コンパイラー・リファレンス」の『*アーキテクチャー固有の (32 または 64 ビットの) コンパイル用コンパイラー・オプションの指定*』を参照してください。

「*XL C*コンパイラー・リファレンス」**の関連情報**

-q32、-q64

-qarch

-qipa

-qtune

-qcache

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

ターゲット・マシンのオプションの最大活用

-qarch オプションの使用

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。

ライブラリー関数を呼び出す方法によらずに、インライン・コードを生成することによって平方根演算を最適化するには、平方根機能をサポートするプロセッサー・ファミリーを指定し、さらに、**-qignerrno** オプション (またはそれを暗黙指定する任意の最適化オプション) も指定する必要があります。 **-qarch=ppc64grsq** を指定してください。このオプションは、**ppc64grsq** グループに属するすべてのプロセッサー (RS64 II、RS64 III、POWER3™、POWER4™、POWER5™、POWER6™、および PowerPC970) に対して正しいコードを生成します。

-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については、「*XL C*コンパイラー・リファレンス」の **-qtune** セクションの『*受け入れ可能な -qarch/-qtune の組み合わせ*』を参照してください。

広範囲の PowerPC ハードウェアで実行される単一バイナリーを作成する場合は、**-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*コンパイラー・リファレンス」の **-qcache** を参照してください。

「*XL C*コンパイラー・リファレンス」の関連情報

 -qignerrno

 -qhot

 -qsmp

 -qcache

 -qlistopt

 -qarch

 -qtune

上位ループ分析および変換の使用

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

これらのループ最適化の目的は次のとおりです。

- キャッシュと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。

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

表 19. -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。
level=0	コンパイラーに上位の変換のサブセットを実行するように指示して、データの局所性を改善してパフォーマンスを向上させます。このサブオプションは、-qhot=novector、-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。
ベクトル	**-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、一度に複数の結果を計算するため、それぞれの結果を順番に計算するより処理は速くなります。このサブオプションは、-qarch を ppc970 または pwr6 に設定し、かつ **-qenablevmx** を使用すると、AIX 上で使用可能になります。

「*XL C コンパイラー・リファレンス*」の関連情報

-qhot

-qstrict

-qignerrno

-qarch

-qenablevmx

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

「*XL C*コンパイラー・リファレンス」の関連情報

-qcompact

-qhot

-qenablevmx

-qstrict

共用メモリーの並列処理 (SMP) の使用

IBM pSeries® のマシンの多くは、共用メモリーの並列処理ができます。 **-qsmp** でコンパイルすると、この機能の活用に必要なスレッド化されたコードを生成することができます。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*コンパイラー・リファレンス」の **-qsmp** にあります。自動並列処理と、IBM SMP および OpenMP ディレクティブの概要については、 99 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 20. 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、IBM SMP および 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。

表 20. 一般に使用される -qsmp のサブオプション (続き)

サブオプション	振る舞い
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。

「*XL C*コンパイラー・リファレンス」の関連情報

-O、-qoptimize

-qsmp

-qhot

-qsmp の最大活用

以下に、**-qsmp** オプションを使用する場合の提案事項を挙げます。

- 自動並行処理で **-qsmp** を使用する前に、最適化と **-qhot** を単一スレッド方式で使用して、プログラムをテストしてください。
- OpenMP プログラムをコンパイルするけれど、自動並行処理は必要ないという場合は、**-qsmp=omp:noauto** を使用します。
- **-qsmp** を使用する場合は、常に 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。

「*XL C*コンパイラー・リファレンス」の関連情報

-qsmp

-qhot

コンパイラーの呼び出し

XLSMPOPTS

並列処理の環境変数

プロシージャー間分析の使用

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

プロシージャー間分析は、コンパイル・ステップのみ、あるいはコンパイル・ステップとリンク・ステップの両方 (「プログラム全体」モード) で指定できます(リンク・ステップで指定する必要がある **clonearch** および **cloneproc** サブオプションを除く)。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*コンパイラー・リファレンス」の『**-qipa**』セクションを参照してください。

IPA を使用する手順は次のとおりです。

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

表 21. 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 21. 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。

「*XL C コンパイラー・リファレンス***」の関連情報**

-qipa

-qipa の最大活用

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

- アプリケーション全体のコンパイル・ステップとリンク・ステップの両方で **-qipa** オプションを指定します。ライブラリー、共用オブジェクト、および実行

可能ファイルに対しても **-qipa** を使用できますが、 main 関数およびエクスポート機能をコンパイルする場合は、必ず **-qipa** を使用してください。

- コンパイルとリンクを個別に行う場合は、高速コンパイルのコンパイル・ステップで、**-qipa=noobject** を使用します。
- Make ファイルで最適化オプションを指定する場合は、必ず、コンパイラー・ドライバー (**xlc**) を使用してリンクし、リンク・ステップですべてのコンパイラー・オプションを組み込むようにしてください。
- IPA で、従来のコンパイルより非常に大きなオブジェクト・ファイルを生成できるので、 /tmp ディレクトリーに十分なスペース (少なくとも 200 MB) があることを確認してください。 TMPDIR 環境変数を使用すれば、十分なフリー・スペースを持つディレクトリーを指定できます。
- リンク時間が長すぎる場合は、**level** サブオプションを変えてみてください。 **-qipa=level=0** を指定したコンパイルは、追加リンク時間が短い場合に非常に有益です。
- **-qipa=list=long** を使用して、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。

「*XL C コンパイラー・リファレンス*」の関連情報

-Q、-qinline

-qlist

-qipa

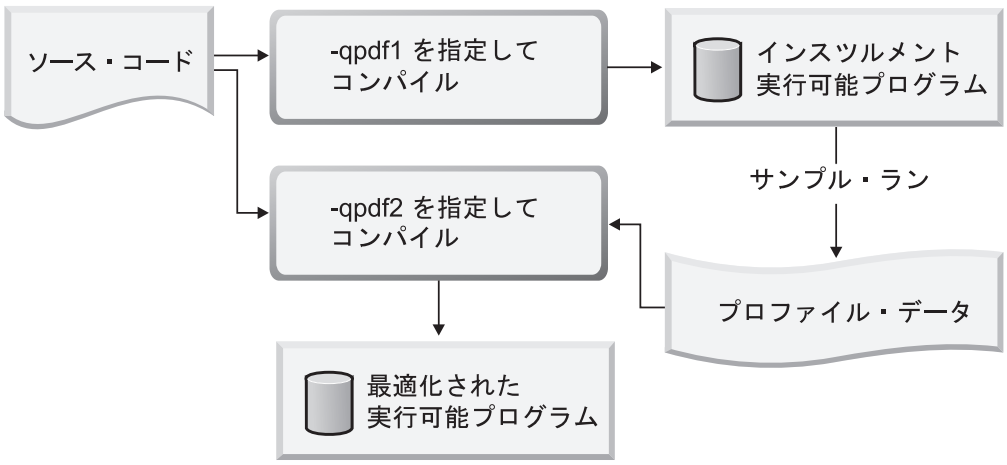
プロファイル指示フィードバックの使用

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

PDF プロセスは、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 1. 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。アプリケーションの終了時には、デフォルトで、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内の 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 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** を使用するか、PDF をまったく使用しないで再コンパイルすると解消されます。

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。

「*XL C*コンパイラー・リファレンス」**の関連情報**

-qpdf1, -qpdf2

-O、-qoptimize

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 )
```

「*XL C*コンパイラー・リファレンス」の関連情報

-qpdf1, -qpdf2

-qshowpdf

オブジェクト・レベルのプロファイル指示フィードバック

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。例えば次のようになります。

  ```
  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

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

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

表 22. 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。-O3 では、 **-qunroll** が暗黙のうちにアクティブになっています。
-qinlglue	リンカーによって生成され、外部関数の呼び出しまたは関数ポインターを介した呼び出しに使用される「グルー・コード」をインライン化するよう、コンパイラーに命令します。
-qtbtable	トレースバック・テーブル情報の生成を制御します。
-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。
-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。
-qlargepage	ハードウェアの事前取り出しをより効率的に行うことができるように、デフォルトの 4K ページに加えて大容量の 16M ページをサポートします。ヒープおよび静的データが実行時に大容量ページから割り当てられることを、コンパイラーに通知します。

「*XL Cコンパイラー・リファレンス*」の関連情報

-qignerrno

- -qsmallstack
- -Q、-qinline
- -qunroll / #pragma unroll
- -qinlglue
- -qtbtable
- -qunwind
- -qstrict
- -qlargepage

第 8 章 最適化コードのデバッグ

最適化されたプログラムをデバッグすると、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、 74 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、 75 ページの『最適化プログラムをデバッグするのに役立つ -qoptdebug の使用』を参照してください。

最適化されたプログラムにおける異なる結果の理解

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

- プログラムが無効なコードを含んでいる場合は、最適化されたコードが失敗する可能性があります。 最適化プロセスは、アプリケーションが言語標準に準拠していることを前提とします。
- 最適化なしで機能するプログラムが最適化時に失敗した場合は、プログラムの相互参照リストと実行フローから、初期化される前に使用される変数を調べてください。 **-qinitauto=***hex_value* オプション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。

-qextchk
リンカーが外部変数および関数の複数ファイル型検査を実行できるようにする追加シンボル情報を生成します。このオプションを使用するには、リンカー **-btypchk** オプションをアクティブにする必要があります。

-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。

最適化プログラムをデバッグするのに役立つ -qoptdebug の使用

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

次の例に倣ってプログラムをコンパイルします。

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

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

注:

- コンパイルされた実行可能ファイルをデバッグ可能にするためには、**-g** または **-qlinedebug** オプションも指定する必要があります。ただし、上記のいずれのオプ

ションも指定されなかった場合でも、最適化された疑似コードを含む疑似コード・ファイル <output_file>.optdbg が生成されます。

- **-qoptdebug** オプションが影響を持つのは、最適化オプション **-qhot**、**-qsmp**、**-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。

最適化されたプログラムのデバッグ

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 2: 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。

77 ページの図 3: 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。

77 ページの図 4: 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 2. **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 3. **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 4. **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

49 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 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 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』
- 80 ページの『関数呼び出しによるオーバーヘッドの低減』
- 81 ページの『効率的なメモリーの管理』
- 81 ページの『変数の最適化』
- 82 ページの『効率的なストリングの操作』
- 83 ページの『式とプログラム・ロジックの最適化』
- 83 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、関数は 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。
- 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。
- 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。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。

  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。

「*XL C*コンパイラー・リファレンス」の関連情報

#pragma expected_value

-qisolated_call / #pragma isolated_call

#pragma disjoint

-qipa

効率的なメモリーの管理

- 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。
- 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++ 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。 `long double` データ型は、極端に高い精度が要求されるときにのみ使用してください。

「*XL C*コンパイラー・リファレンス」**の関連情報**

-qisolated_call / #pragma isolated_call

効率的なストリングの操作

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

- 割り当てられたストレージにストリングを保管するときは、ストリングの開始を 8 バイトの境界に位置合わせする。
- ストリングの長さを常に把握しておく。ストリングの長さが分かっている場合は、`str` 関数の代わりに `mem` 関数を使用することができます。例えば、`memcpy` が `strcpy` より高速なのは、ストリングの終わりを検索する必要がないためです。
- ソースとターゲットがオーバーラップしないことが確かな場合には、`memmove` の代わりに、`memcpy` を使用する。これは、`memcpy` がソースから宛先に直接コピーするのに対し、 `memmove` は、ソースをメモリー内の一時ロケーションにコピーしてから、宛先にコピーすることがあるためです (ストリングの長さによります)。
- `mem` 関数を使用してストリングを処理するときに、*count* パラメーターが変数でなく定数であれば、より高速なコードが生成される。これは、小カウント値の場合に特に当てはまります。
- 可能であれば、ストリング・リテラルを読み取り専用にする。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。

「*XL C*コンパイラー・リファレンス」**の関連情報**


-qro / #pragma strings

式とプログラム・ロジックの最適化

次のガイドラインを考慮してください。

- ある式のコンポーネントが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。
- 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 for AIX, V10.1 は、ハイパフォーマンス数学計算のライブラリー・セットを同梱して出荷されています。

- Mathematical Acceleration Subsystem (MASS) は調整された数学組み込み関数のライブラリー・セットで、標準システム数学ライブラリー関数を超える向上した性能を提供します。 MASS については 『Mathematical Acceleration Subsystem (MASS) ライブラリーの使用』で説明されています。
- Basic Linear Algebra Subprograms (BLAS) はルーチンのセットで、 PowerPC アーキテクチャー用に調整された、行列ベクトル乗算関数を提供しています。 BLAS 関数については 94 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 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 (87 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。

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

関連外部情報

Mathematical Acceleration Subsystem Web サイト (http://www.ibm.com/software/awdtools/mass/ から入手可能)

スカラー・ライブラリーの使用

MASS スカラー・ライブラリー libmass.a 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。MASS スカラー関数は、libmass.a に明示的にリンクするときに使用されますが、プログラムを下記のいずれかのオプションを使用してコンパイルするときも、これらの関数を自動的に使用できます。

- **-qhot -O3**
- **-O4**
- **-O5**

これらのオプションを使用すると、コンパイラーは、ほとんどの数学ライブラリー関数について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。

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 23 に示します。

表 23. 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);
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);

表 23. 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
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);
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);
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)。
- libm.a との精度比較については、*Mathematical Acceleration Subsystem Web サイト* を参照してください。

関連外部情報

Mathematical Acceleration Subsystem Web サイト (http://www.ibm.com/software/awdtools/mass/ から入手可能)

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 -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 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、 94 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。)

libmassv.a
一般的なベクトル・ライブラリー。

libmassvp3.a
POWER3 アーキテクチャー用に調整された関数をいくつか含みます。残りの関数は、libmassv.a の関数と同一です。

libmassvp4.a
POWER4 アーキテクチャー用に調整された関数をいくつか含みます。残りの関数は、libmassv.a の関数と同一です。 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いくつか含みます。残りの関数は、libmassv.a の関数と同一です。

libmassvp6.a
POWER6 アーキテクチャー用に調整された関数をいくつか含みます。残りの関数は、libmassv.a の関数と同一です。

すべてのライブラリーは 32 ビット・モード、または 64 ビット・モードのいずれかで使用可能です。

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、 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、 92 ページの表 25 にまとまられています。

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

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

関数の形式は次のとおりです。

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

ここで、*y* はターゲット・ベクトル、*x* はソース・ベクトル、*n* はベクトルの長さです。パラメーター *y* および *x* は、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と見なされます。下記にコード例を示します。 エレメントが 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。このエレメントは x[i]/y[i] (i=0,..,*n-1) です。関数 vpow はベクトル *z* を出力します。このエレメントは $x[i]^{y[i]}$ (i=0,..,*n-1) です。関数 vatan2 はベクトル *z* を出力します。このエレメントは atan(x[i]/y[i]) (i=0,..,*n-1) です。関数 vsincos はパラメーター (*y*,*z*,*x*,*n*) を使い、そして 2 つのベクトル、*y* および *z* を出力します。ここでエレメントは各々 sin(x[i]) および cos(x[i]) です。

vcosisin(y,x,n) および vscosisin(y,x,n) では、*x* は*n* 個のエレメントのベクトルであり、関数は、(cos(x[i]),sin(x[i])) の形式の *n* 個の complex エレメントのベクトル *y* を出力します。**-D__nocomplex** を使用する場合 (表 24の「注」を参照)、出力ベクトルは y[0][i] = cos(x[i]) および y[1][i] = sin(x[i]) を保持します。ここで i=0,..,*n-1 です。

表 24. 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);

表 24. 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
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[1]	vscosisin[1]	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);	
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] の対数 (底 10) に設定する	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	i=0,..,*n-1 の場合に、y[i] を (x[i]+1) の自然対数に設定する	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);

表 24. 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
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);
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);

表 24. 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
注:				
1. デフォルトでは、これらの関数は __Complex データ型を使用します。このデータ型は AIX 5.2 以降でのみ使用可能で、古いバージョンのオペレーティング・システムではコンパイルされません。これらの関数の代替プロトタイプを取得するには、**-D__nocomplex** を使用してコンパイルします。これによって、関数は void vcosisin (double y[][2], double *x, int *n); および void vscosisin(float y[][2], float *x, int *n); として定義されます。				

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 25. 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)

入力ベクトルと出力ベクトルのオーバーラップ

大部分のアプリケーションでは MASS ベクトル関数は相互に素 (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については、下記の制限に注意してご使用のアプリケーションに対する正しい操作を確実に行ってください。

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

  ベクトル x[0:n-1] および y[0:n-1] が相互に素または同一でなければならないか、または x[0] のアドレスが y[0] のアドレスよりも大きくなければなりません。 つまり、x と y が同じベクトルでない場合、y[0] のアドレスは、x[0:n-1] で範囲指定されたアドレスの範囲内であってはなりません。
- 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 が x1 と同じベクトルでない場合、y[0] のアドレスは x1[0:n-1] で範囲指定されたアドレスの範囲内であってはなりません。y が x2 と同じベクトルでない場合、y[0] のアドレスは x2[0:n-1] で範囲指定されたアドレスの範囲内であってはなりません。

- 2 つの出力ベクトル (例えば、vsincos (y1, y2, x, &n)) を受け取るベクトル関数の呼び出しには以下に注意します。

  両方のベクトルのペア y1、x および y2、x に対して上記の制限が適用されます。つまり、y1 および x が同じベクトルでない場合、y1[0] のアドレスは x[0:n-1] で範囲指定されたアドレスの範囲内であってはなりません。y2 および x が同じベクトルでない場合、y2[0] のアドレスは x[0:n-1] で範囲指定されたアドレスの範囲内であってはなりません。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

ベクトル関数の精度は、libmass.a での該当するスカラー関数の精度と同程度ですが、結果はビット単位で同一ではない場合があります。

速度を高めるために、MASS ライブラリーは特定のトレードオフを作成します。トレードオフの 1 つでは、特定の MASS ベクトル関数の整合性を必要とします。ある関数では、特定の入力値を計算した結果は、ベクトル内での位置、ベクトルの長さ、および入力ベクトルの隣接エレメントによって、わずかに異なる (通常、最下位ビットのみ) 可能性があります。また、異なる MASS ライブラリーによって作成された結果は、必ずしもビット単位で同一ではありません。

下記の関数はすべてのバージョンのライブラリーにおいて整合しています。 vcbrt、vscbrt、vrcbrt、vsrcbrt、vlog、vsin、vssin、vcos、vscos、vsexp、vacos、vasin、vrqdrt、vsqdrt、vsrqdrt、vacosh、vsacosh、vasinh、vsasinh、vtanh、vstanh 下記の関数は、libmassvp3.a、libmassvp4.a、libmassvp5.a、および libmassvp6.a において整合しています。 vsqrt および vrsqrt 下記の関数は、libmassvp4.a、libmassvp5.a、および libmassvp6.a については整合しています。 vrec、vsrec、vdiv、vsdiv、および vexp 下記の関数は libmassv.a、libmassvp5.a、および libmassvp6.a について整合しています。vsrsqrt 古い整合性のないバージョンの関数が、一部 *Mathematical Acceleration Subsystem for AIX Web* サイト で使用可能です。整合性が不要な場合は、古いバージョンを使った方がパフォーマンス上の利点がある場合があります。ベクトル・ライブラリーとの整合性および不整合の回避についての詳細のほか、パフォーマンスと精度のデータについて詳しくは、*Mathematical Acceleration Subsystem Web* サイト を参照してください。

「*XL C*コンパイラー・リファレンス」**の関連情報**

-D

関連外部情報

Mathematical Acceleration Subsystem for AIX Web サイト (http://www.ibm.com/software/awdtools/mass/aix から入手可能)

Mathematical Acceleration Subsystem Web サイト (http://www.ibm.com/software/awdtools/mass/ から入手可能)

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・オプションに **mass** および **massv** (あるいは **massvp3**、**massvp4**、**massvp5**、または **massvp6**) 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、以下を指定できます。

```
xlc progc.c -o progc -lmass -lmassv
```

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 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. 必要な関数の名前を含む、エクスポート・リスト (フラット・テキスト・ファイルになる) を作成する。例えば、C プログラム sample.c で使用する、 libmass.a の高速なタンジェント関数のみを選択する場合は、以下の行を含めた fasttan.exp というファイルを作成します。

   ```
   tan
   ```

2. **ld** コマンドを使用して、エクスポート・リストから共用オブジェクトを作成し、libmass.a ライブラリーとリンクする。次に例を示します。

   ```
   ld -bexport:fasttan.exp -o fasttan.o -bnoentry -lmass -bmodtype:SRE
   ```

3. **ar** コマンドを使用して、共用オブジェクトをライブラリーにアーカイブする。次に例を示します。

   ```
   ar -q libfasttan.a fasttan.o
   ```

4. **xlc** を使用して、最終的な実行可能ファイルを作成します。その際、標準の数学ライブラリー `libm.a` の*前に* MASS 関数が含まれるオブジェクト・ファイルを指定します。そうすると、そのオブジェクト・ファイルで指定された関数 (この例では、`tan` 関数) と標準の数学ライブラリーの残りの数学関数のみがリンクされます。次に例を示します。

   ```
   xlc sample.c -o sample -Ldir_containing_libfasttan -lfasttan -lm
   ```

注: MASS `cosisin` をエクスポートすると、MASS `sincos` 関数は自動的にリンクされます。MASS sin をエクスポートすると、MASS cos 関数が自動的にリンクされます。MASS `atan` をエクスポートすると、MASS `atan2` は自動的にリンクされます。

関連外部情報

- 「*AIX コマンド・リファレンス 第 1 巻 から 第 6 巻*」の **ar** および **ld**

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 にあります。関数は以下から成り立っています。

- sgemv (単精度) および dgemv (倍精度) があり、これらは一般的な行列、またはその転置 (transpose) の行列ベクトルの積を計算します。
- sgemm (単精度) および dgemm (倍精度) があり、これらは一般的な行列、またはその転置 (transpose) に対する結合行列乗算、および加算を行います。

BLAS ルーチンは Fortran で書かれているため、すべてのパラメーターは参照によって渡され、すべての配列は列優先順位 (column-major order) で保管されます。

注: いくつかのエラー処理コードが libxlopt の BLAS 関数から取り外されて、これらの関数の呼び出しに対するエラー・メッセージは出されません。

『BLAS 関数構文』 は XL C 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、98 ページの『libxlopt ライブラリーへのリンク』 に XL 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 これは 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 これは float (sgemm 用) または double (dgemm 用) 値の出力行列 *c* です。

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 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、 100 ページの『計数可能ループ』 に定義されています。コンパイラーの自動並列処理機能の概要が 101 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。
- IBM SMP ディレクティブを使用した、計数可能ループの明示的並列処理。 IBM SMP ディレクティブの概要は、 102 ページの『IBM SMP ディレクティブの使用』 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 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、 103 ページの『OpenMP ディレクティブの使用』 で提供されています。

プログラム並列処理のすべてのメソッドは、**omp** サブオプションなしに **-qsmp** コンパイラー・オプションが有効なとき、使用可能になります。ユーザーは厳格な OpenMP の整合性を **-qsmp=omp** コンパイラー・オプションによって使用可能にできますが、それを行うと自動並列処理が使用不可にされます。

注: -qsmp オプションはスレッド・セーフ・コンパイラー呼び出しモード (**_r** 接尾部を含むもの) と共に使用されなければなりません。

プログラム・コードの並列領域は、マルチスレッドによって、そしておそらくは複数処理装置上で実行されます。作成されるスレッドの数は、環境変数とライブラリー関数に対する呼び出しによって決定されます。作業は、環境変数で指定されるスケジューリング・アルゴリズムにしたがって、使用可能なスレッドに分散されます。並列処理のどのようなメソッドについても、XLSMPOPTS 環境変数と、スレッド・スケジューリングを制御するためのサブオプションが使用可能です。この環境変数の詳細については、「*XL C*コンパイラー・リファレンス」の *XLSMPOPTS* を参照してください。 OpenMP 構成をご使用中の場合は、OpenMP 環境変数を使用してスレッド・スケジューリングを制御することができます。 OpenMP 環境変数についての詳細については、「*XL C*コンパイラー・リファレンス」の *並列処理のための OpenMP 環境変数*を参照してください。 IBM SMP および OpenMP 組み込み関数の詳細については、「*XL C*コンパイラー・リファレンス」の『*並列処理のための組み込み関数*』を参照してください。

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、http://www.openmp.org から利用可能な「*OpenMP Application Program Interface Language Specification*」を参照してください。

関連情報

60 ページの『共用メモリーの並列処理 (SMP) の使用』

IBM pSeries のマシンの多くは、共用メモリーの並列処理ができます。 **-qsmp**

でコンパイルすると、この機能の活用に必要なスレッド化されたコードを生成することができます。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。

「*XL C*コンパイラー・リファレンス」**の関連情報**

XLSMPOPTS

並列処理の OpenMP 環境変数

並列処理のための組み込み関数

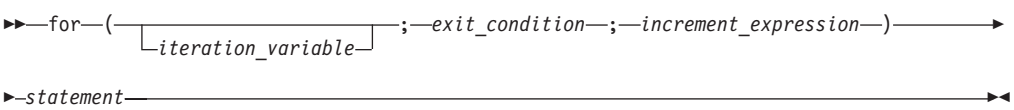
関連外部情報

OpenMP Application Program Interface Language Specification (http://www.openmp.org から入手可能)

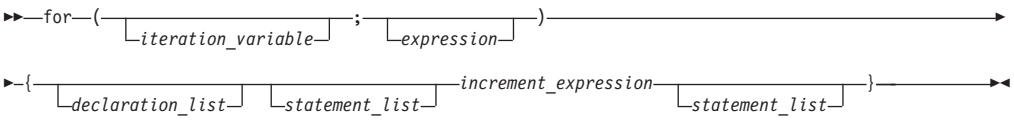
計数可能ループ

ループは、下記のいずれかの形であれば 計数可能 と考えられます。

計数可能ループ構文 (Countable for loop syntax)

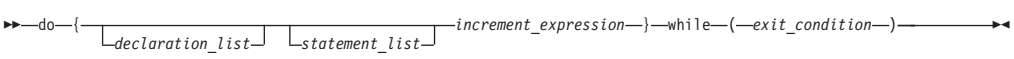


ステートメント・ブロック付きの計数可能ループ構文 (Countable for loop syntax with statement block)



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

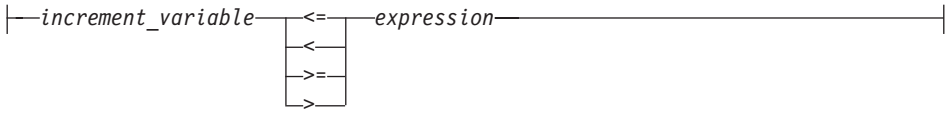
計数可能ループ中 Do 構文 (Countable do while loop syntax)



上記の構文図に以下の定義が適用されます。

iteration_variable
これは、自動または登録ストレージ・クラスを持つ符号付き整数で、決まったアドレスを持たず、 *increment_expression* 以外では、ループ内のどこでも変更されません。

exit_condition
次の書式を持っています。



ここで *expression* はループ・インバリアント符号付き整数式です。 *expression* は、外部変数または静的変数、ポインターまたはポインター式、関数呼び出し、またはアドレスを持つ変数を参照できません。

increment_expression
下記の任意の書式を使用できます。

- *++iteration_variable*
- *--iteration_variable*
- *iteration_variable++*
- *iteration_variable--*
- *iteration_variable += increment*
- *iteration_variable -= increment*
- *iteration_variable = iteration_variable + increment*
- *iteration_variable = increment + iteration_variable*
- *iteration_variable = iteration_variable - increment*

ここで *increment* はループ・インバリアント符号付き整数式です。 expression の値は実行時に知ることができますが 0 ではありません。*increment* は、外部変数または静的変数、ポインターまたはポインター式、関数呼び出し、またはアドレスを持つ変数を参照できません。

自動並列処理の使用可能化

コンパイラーは自動的に計数可能ループを探し、可能な場合はユーザーのプログラム・コード内の計数可能ループをすべて並列処理します。ループは、 100 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。

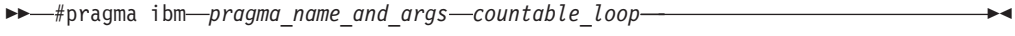
- **-qnostrict_induction** コンパイラー・オプションが有効である。
- **-qsmp=auto** コンパイラー・オプションが有効である。
- コンパイラーは、スレッド・セーフ・コンパイラー・モードで呼び出される。

IBM SMP ディレクティブの使用

IBM SMP ディレクティブは countable loops の並列処理を通じて共用メモリーの並列処理を活用しています。ループは、 100 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れば countable と見なされます。 XL C コンパイラーは、コンパイラーが実行する自動並列処理を強化するために使用する、プラグマ・ディレクティブを提供しています。プラグマは、2 つの一般的なカテゴリーに分かれます。

1. 並列処理に対する明示的なコントロールをユーザーに与えるプラグマ。これらのプラグマを使ってループ (**#pragma ibm parallel_loop** および **#pragma ibm sequential_loop**) の並列処理を強制したり抑制を行い、ループ (**#pragma ibm schedule**) に特定の並列処理アルゴリズムを適用し、そしてクリティカル・セクション (**#pragma ibm critical**) を使用する共用変数へのアクセスを同期化します。
2. プラグマは、ユーザーに特定の計数可能ループ (**#pragma ibm independent_calls**、**#pragma ibm independent_loop**、**#pragma ibm iterations**、**#pragma ibm permutation**) の特性に関するコンパイラー情報を提供させます。コンパイラーはこの情報を使って、さらに効率の高いループの自動並列処理を行います。

IBM SMP ディレクティブ構文



プラグマ・ディレクティブは、それが適用される計数可能なループの直前に現れなければなりません。計数可能ループに、2 つ以上の並列処理プラグマ・ディレクティブを適用することができます。次に例を示します。

```
#pragma ibm independent_loop
#pragma ibm independent_calls
#pragma ibm schedule(static,5)
countable_loop
```

いくつかのプラグマ・ディレクティブ、例えば、parallel_loop sequential_loop ディレクティブは、各々相互に排他的です。同一のループに相互に排他的なプラグマが指定されると、最後に指定されたプラグマがループに適用されます。

他のプラグマが、与えられたループに繰り返し指定されると、追加の影響があります。次に例を示します。

```
#pragma ibm permutation (a,b)
#pragma ibm permutation (c)
```

は

```
#pragma ibm permutation
(a,b,c)
```

IBM SMP ディレクティブのプラグマごとの記述については、「*XL C*コンパイラー・リファレンス」の *並列処理のためのプラグマ・ディレクティブ*を参照してください。

「*XL C*コンパイラー・リファレンス」**の関連情報**

並列処理のためのプラグマ・ディレクティブ

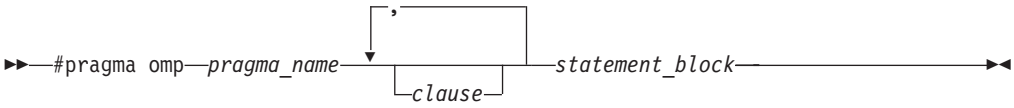
OpenMP ディレクティブの使用

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、5 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**, **#pragma omp task**)。
3. スレッド間の同期をユーザーがコントロールできるプラグマ (**#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**)。
5. タスク同期化用のプラグマ (**#pragma omp taskwait**, **#pragma omp barrier**)

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
          ...
      ....
}
```

IBM SMP ディレクティブのプラグマごとの記述については、「*XL C*コンパイラー・リファレンス」の *並列処理のためのプラグマ・ディレクティブ*を参照してください。

「*XL C*コンパイラー・リファレンス」**の関連情報**

並列処理のためのプラグマ・ディレクティブ

OpenMP 組み込み関数

並列処理の OpenMP 環境変数

並列環境内の共用変数と専用変数

並列環境内の変数は、共用、または専用のコンテキストを持つことができます。共用コンテキスト内の変数は、関連した並列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。

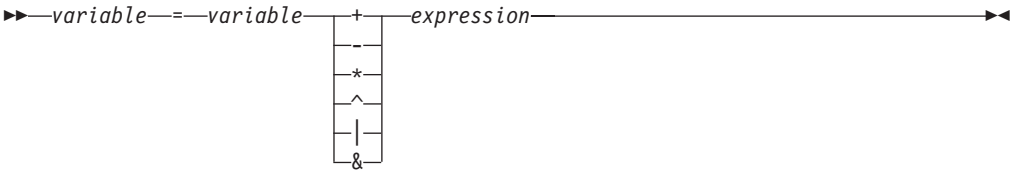
詳細については、「*XL Cコンパイラー・リファレンス*」の『*並列処理のためのプラグマ・ディレクティブ*』または「*OpenMP Application Program Interface Language Specification*」に記載の OpenMP ディレクティブの説明を参照してください。

「*XL C*コンパイラー・リファレンス」**の関連情報**

並列処理のためのプラグマ・ディレクティブ

並列処理ループ内の縮小操作

コンパイラーは、自動、および明示的な並列処理中の、ループ内の縮小操作を大部分認識し、適切に処理することができます。特に、下記の書式のいずれかを持つ縮小文を処理することができます。



ここで、

variable
これは、自動または登録変数を指定する ID で、その決まったアドレスを持たず、ネストされた全ループを含むループ内の他のどこからも参照されません。例えば、次に示すコード内では、ネストされたループ内の S のみが縮小として認識されます。

```
int i,j, S=0;
for (i= 0 ;i < N; i++) {
    S = S+ i;
     for (j=0;j< M; j++) {
        S = S + j;
    }
}
```

expression
これは任意の有効な式です。

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。 IBM ディレクティブを使用するときは、コンパイラーに認識されないすべての縮小変数へのアクセスを同期化するために、クリティカル・セクションを使用してください。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。

第 12 章 メモリー・デバッグ・ライブラリー関数

この付録には、標準 C メモリー管理関数の拡張である、XL C メモリー・デバッグ・ライブラリー関数に関する参照情報が含まれています。この付録は 2 つのセクションに分かれています。

- 『メモリー割り振りのデバッグ関数』 には、ヒープ・メモリーの割り振りに関する標準ライブラリー関数のデバッグ・バージョンが記載されています。
- 116 ページの『ストリング・ハンドリングのデバッグ関数』 には、ストリングの取り扱いに関する標準ライブラリー関数のデバッグ・バージョンが記載されています。

これらのデバッグ・バージョンを使用するには、以下の手順のいずれかを行います。

- ソース・コード内で、デフォルト、またはユーザー定義のヒープ・メモリー管理関数のいずれかに、プレフィックス `_debug_` を付加します。
- ソース・コードに変更を加えたくない場合は、単純に **-qheapdebug** オプション付きでコンパイルしてください。このオプションは、すべてのメモリー管理関数の呼び出しを、それらのデバッグ・バージョンのカウンター・パートにマップします。呼び出しがマップされることを避けるには、関数名に括弧を付けます。

この付録で提供されているすべての事例は、 **-qheapdebug** オプション付きでコンパイルすることを前提にしています。

「*XL C*コンパイラー・リファレンス」の関連情報

-qheapdebug

メモリー割り振りのデバッグ関数

このセクションでは、標準と、ユーザー作成のヒープ・メモリー割り振り関数のデバッグ・バージョンについて記載しています。これらの関数は、ヒープの妥当性を検査するために、すべて自動的に `_heap_check` または `_uheap_check` の呼び出しを行います。それから `_dump_allocated` または `_dump_allocated_delta` 関数を使用して、ヒープ検査関数から戻された情報をプリントします。

関連情報

41 ページの『メモリー・ヒープをデバッグする関数』

_debug_calloc — メモリーの割り当てと初期化

形式

```
#include <stdlib.h>   /* also in <malloc.h> */
void *_debug_calloc(size_t num, size_t size, const char *file, size_t line);
```

目的

これは `calloc` のデバッグ・バージョンです。`calloc` と同様、*num* 個のエレメントを持つ配列に、それぞれ長さ *size* バイトだけ、デフォルト・ヒープからメモリーを

割り当てます。それから各エレメントのすべてのビットを 0 に初期化します。さらに、_debug_calloc は _heap_check への暗黙の呼び出しを行い、ファイル *file* の名前およびストレージが割り当てられる行番号 *line* を格納します。

戻り値

予約されたスペースへポインターを戻します。使用可能なメモリーが不足している場合や、 *num* または *size* が 0 である場合、NULL を戻します。

例

次の例では 100 バイトのストレージが予約されます。続いて、割り当てられなかったストレージへの書き込みを試行します。 _debug_calloc を再び呼び出したときに、_heap_check はエラーを検出し、いくつかのメッセージを生成し、そしてプログラムを停止します。

```
/*  _debug_calloc.c  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *ptr1, *ptr2;

   if (NULL == (ptr1 = (char*)calloc(1, 100))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   memset(ptr1, 'a', 105);         /* overwrites storage that was not allocated */
   ptr2 = (char*)calloc(2, 20);      /* this call to calloc invokes _heap_check */
   puts("_debug_calloc did not detect that a memory block was overwritten.");
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 6161616161616161.
This memory block was (re)allocated at line number 9 in _debug_calloc.c.
Heap state was valid at line 9 of _debug_calloc.c.
Memory error detected at line 14 of _debug_calloc.c.
```

_debug_free — 割り当てられたメモリーの解放

形式

```
#include <stdlib.h>   /* also in <malloc.h> */
void _debug_free(void *ptr, const char *file, size_t line);
```

目的

これは free のデバッグ・バージョンです。 free と同様、*ptr* でポイントされるメモリー・ブロックを解放します。 _debug_free もまた、解放したメモリーの各ブロックを 0xFB に設定します。したがって、解放されたメモリー内のデータをプログラムが使用する場所で、インスタンスを容易に見つけることができます。さらに、_debug_free は、_heap_check 関数の暗黙の呼び出しを行い、メモリーが解放された場所のファイル名 *file* と行番号 *line* を格納します。

_debug_free は、どのヒープからメモリーが割り当てられたかを常にチェックします。したがって、メモリー管理関数の正規バージョン、ヒープ固有バージョン、またはデバッグ・バージョンを使用して割り当てたメモリー・ブロックを解放することができます。ただし、メモリーがメモリー管理関数によって割り当てられていなかった場合や、以前にメモリーが解放されていた場合は、 _debug_free はエラー・メッセージを生成し、プログラムは終了します。

戻り値

戻り値はありません。

例

次の例では、2 つのブロック (1 つは 10 バイトで、もう 1 つは 20 バイト) を予約します。続いて、最初のブロックを解放し、解放されたストレージへの上書きを試みます。 _debug_free が再び呼び出されたときに、_heap_check はエラーを検出し、いくつかのメッセージを印刷し、プログラムを停止します。

```
/*  _debug_free.c  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *ptr1, *ptr2;

   if (NULL == (ptr1 = (char*)malloc(10)) || NULL == (ptr2 = (char*)malloc(20))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   free(ptr1);
   memset(ptr1, 'a', 5);      /* overwrites storage that has been freed       */
   free(ptr2);                 /* this call to free invokes _heap_check       */
   puts("_debug_free did not detect that a freed memory block was overwritten.");
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
Free heap was overwritten at 0x00073890.
Heap state was valid at line 12 of _debug_free.c.
Memory error detected at line 14 of _debug_free.c.
```

_debug_heapmin — デフォルト・ヒープ内の未使用メモリーの解放

形式

```
#include <stdlib.h>   /* also in <malloc.h> */
int _debug_heapmin(const char *file, size_t line);
```

目的

これは _heapmin のデバッグ・バージョンです。 _heapmin と同様、デフォルトのランタイム・ヒープからオペレーティング・システムにすべての未使用のメモリーを戻します。さらに、_debug_heapmin は、_heap_check 関数の暗黙の呼び出しを行い、メモリーが戻された場所のファイル名 *file* と行番号 *line* を格納します。

戻り値

成功した場合は 0 を戻します。成功しなかった場合は -1 を戻します。

例

次の例では、10000 バイトのストレージを割り当て、ストレージ・サイズを 10 バイトに変更し、続いて _debug_heapmin を使用して、未使用のメモリーをオペレーティング・システムに戻します。続いて、プログラムは、割り当てられなかったメモリーへの上書きを試みます。 _debug_heapmin を再び呼び出したときに、_heap_check はエラーを検出し、いくつかのメッセージを生成し、そしてプログラムを停止します。

```
/*   _debug_heapmin.c  */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *ptr;

   /* Allocate a large object from the system */
   if (NULL == (ptr = (char*)malloc(100000))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   ptr = (char*)realloc(ptr, 10);
   _heapmin();               /* No allocation problems to detect             */

   *(ptr - 1) = 'a';         /* Overwrite memory that was not allocated      */
   _heapmin();               /* This call to _heapmin invokes _heap_check    */

   puts("_debug_heapmin did not detect that a non-allocated memory block"
        "was overwritten.");
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
Header information of object 0x000738b0 was overwritten at 0x000738ac.
The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
This memory block was (re)allocated at line number 13 in _debug_heapmin.c.
Heap state was valid at line 14 of _debug_heapmin.c.
Memory error detected at line 17 of _debug_heapmin.c.
```

_debug_malloc — メモリーの割り当て

形式

```
#include <stdlib.h>   /* also in <malloc.h> */
void *_debug_malloc(size_t size, const char *file, size_t line);
```

目的

これは malloc のデバッグ・バージョンです。 malloc と同様、デフォルトのヒープから *size* バイトのストレージ・ブロックを予約します。また、_debug_malloc は、割り当てるすべてのメモリーを 0xAA に設定します。したがって、最初にそれを初期化しなくても、メモリー内のデータをプログラムが使用する場所で、インスタンスを容易に見つけることができます。さらに、_debug_malloc は、_heap_check 関数の暗黙の呼び出しを行い、ファイル名 *file* とメモリーが割り振られた場所の行番号 *line* を格納します。

戻り値

予約されたスペースへポインターを戻します。使用可能なメモリーが不足している場合や、*size* が 0 である場合、NULL を戻します。

例

次の例では 100 バイトのストレージを割り当てます。続いて、割り当てられなかったストレージへの書き込みを試行します。 _debug_malloc を再び呼び出したときに、_heap_check はエラーを検出し、いくつかのメッセージを生成し、そしてプログラムを停止します。

```
/*   _debug_malloc.c  */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
      char *ptr1, *ptr2;

      if (NULL == (ptr1 = (char*)malloc(100))) {
      puts("Could not allocate memory block.");
            exit(EXIT_FAILURE);
   }
      *(ptr1 - 1) = 'a';        /* overwrites storage that was not allocated    */
      ptr2 = (char*)malloc(10); /* this call to malloc invokes _heap_check      */
   puts("_debug_malloc did not detect that a memory block was overwritten.");
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
Header information of object 0x00073890 was overwritten at 0x0007388c.
The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
This memory block was (re)allocated at line number 8 in _debug_malloc.c.
Heap state was valid at line 8 of _debug_malloc.c.
Memory error detected at line 13 of _debug_malloc.c.
```

_debug_ucalloc — ユーザー作成ヒープからのメモリーの予約と初期化

形式

```
#include <umalloc.h>
void *_debug_ucalloc(Heap_t heap, size_t num, size_t size, const char *file, size_t line);
```

目的

これは _ucalloc のデバッグ・バージョンです。 _ucalloc と同様、*num* 個のエレメントの配列に対して、それぞれ長さ *size* バイトだけ、指定された *heap* からメモリーを割り当てます。それから各エレメントのすべてのビットを 0 に初期化します。さらに、_debug_ucalloc は _uheap_check への暗黙の呼び出しを行い、ファイル *file* の名前およびストレージが割り当てられる行番号 *line* を格納します。

heap に要求に対する十分なメモリーがない場合、_debug_ucalloc は、_ucreate を使用してヒープを作成したときに指定したヒープ拡張関数を呼び出します。

注: 有効でないヒープを _debug_ucalloc に渡すと、未定義な振る舞いを引き起こすことになります。

戻り値

予約されたスペースへポインターを戻します。 *size* または *num* がゼロに指定されていた場合、あるいはヒープ拡張関数が十分なメモリーを用意できない場合は NULL を戻します。

例

次の例では、ユーザー・ヒープを作成し、_debug_ucalloc を使用して、そのヒープからメモリーを割り当てます。続いて、割り当てられなかったメモリーへの書き込みを試行します。 _debug_free を呼び出したときに、_uheap_check はエラーを検出し、いくつかのメッセージを生成し、そしてプログラムを停止します。

```
/*  _debug_ucalloc.c  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 105);   /* Overwrites storage that was not allocated */
   free(ptr);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 7878787878787878.
This memory block was (re)allocated at line number 14 in _debug_ucalloc.c.
Heap state was valid at line 14 of _debug_ucalloc.c.
Memory error detected at line 19 of _debug_ucalloc.c.
```

_debug_uheapmin — ユーザー作成ヒープ内の未使用メモリーの解放

形式

```
#include <umalloc.h>
int _debug_uheapmin(Heap_t heap, const char *file, size_t line);
```

目的

これは _uheapmin のデバッグ・バージョンです。 _uheapmin と同様、すべての未使用メモリー・ブロックを、指定された *heap* からオペレーティング・システムへ戻します。

メモリーを戻すために、_debug_uheapmin は、_ucreate を使用してヒープを作成するときに用意するヒープ縮小関数を呼び出します。ヒープ縮小関数を指定しないと、_debug_uheapmin は何の効果もなく、0 を戻します。

さらに、_debug_uheapmin はヒープを妥当性検査するために、暗黙の _uheap_check 呼び出しを行います。

戻り値

成功すると、0 を戻します。ゼロ以外の戻り値は失敗を示します。指定したヒープが有効でない場合には、_debug_uheapmin の呼び出しが行われたファイル名と行番号が含まれるエラー・メッセージを生成します。

例

次の例では、ヒープを作成して、メモリーをそこから割り当て、続いて _debug_heapmin を使用してメモリーを解放します。

```
/*  _debug_uheapmin.c   */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <umalloc.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   /* Allocate a large object */
   if (NULL == (ptr = (char*)_umalloc(myheap, 60000))) {
      puts("Cannot allocate memory from user heap.\n");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 60000);
   free(ptr);

   /* _debug_uheapmin will attempt to return the freed object to the system */
   if (0 != _uheapmin(myheap)) {
      puts("_debug_uheapmin returns failed.\n");
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

_debug_umalloc - ユーザー作成ヒープからのメモリーの予約

形式

```
#include <umalloc.h>
void *_debug_umalloc(Heap_t heap, size_t size, const char *file, size_t line);
```

目的

これは _umalloc のデバッグ・バージョンです。 _umalloc と同様、指定された *heap* から、*size* バイトのブロックのストレージ・スペースを予約します。また、_debug_umalloc は、割り当てるすべてのメモリーを 0xAA に設定します。したがって、最初にそれを初期化しなくても、メモリー内のデータをプログラムが使用する場所で、インスタンスを容易に見つけることができます。

さらに、_debug_umalloc は、_uheap_check への暗黙の呼び出しを行い、ファイル *file* の名前およびストレージが割り当てられる行番号 *line* を格納します。

heap に要求に対する十分なメモリーがない場合、_debug_umalloc は、_ucreate を使用してヒープを作成したときに指定したヒープ拡張関数を呼び出します。

注: 有効でないヒープを _debug_umalloc に渡すと、未定義な振る舞いを引き起こすことになります。

戻り値

予約されたスペースへポインターを戻します。 *size* にゼロが指定された場合、またはヒープ拡張関数が十分なメモリーを用意できない場合は、NULL を戻します。

例

次の例では、ヒープ myheap を作成し、_debug_umalloc を使用して、そこから 100 バイトを割り当てます。続いて、割り当てられなかったストレージへの上書きを試行します。_debug_free を呼び出すと _uheap_check を呼び出すことになり、エラーを検出し、メッセージを生成して、プログラムを終了します。

```
/*  _debug_umalloc.c  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
      puts("Cannot allocate memory from user heap.\n");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 105);   /* Overwrites storage that was not allocated */
   free(ptr);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 7878787878787878.
This memory block was (re)allocated at line number 14 in _debug_umalloc.c.
Heap state was valid at line 14 of _debug_umalloc.c.
Memory error detected at line 19 of _debug_umalloc.c.
```

_debug_realloc — メモリー・ブロックの再割り当て

形式

```
#include <stdlib.h>   /* also in <malloc.h> */
void *_debug_realloc(void *ptr, size_t size, const char *file, size_t line);
```

目的

これは realloc のデバッグ・バージョンです。 realloc と同様、*ptr* でポイントされたメモリーのブロックを、新しい *size* (バイト数で指定) に再割り当てします。また、いずれかの割り当てる新しいメモリーを 0xAA に設定します。したがって、最初にそれを初期化しなくても、メモリー内のデータをプログラムが使用する場所

で、インスタンスを容易に見つけることができます。さらに、_debug_realloc は、_heap_check 関数の暗黙の呼び出しを行い、ストレージが再割り振りされた場所のファイル名 *file* と行番号 *line* を格納します。

ptr が NULL の場合、_debug_realloc は _debug_malloc (または malloc) と類似した動作を行い、メモリー・ブロックを割り当てます。

_debug_realloc は、どのヒープからメモリーが割り当てられたかを常にチェックします。したがって、_debug_realloc を使用して、メモリー管理関数の正規バージョンまたはデバッグ・バージョンを使用して割り当てたメモリー・ブロックを再割り当てすることができます。ただし、メモリーがメモリー管理関数によって割り当てられていなかった場合や、以前にメモリーが解放されていた場合は、_debug_realloc はエラー・メッセージを生成し、プログラムは終了します。

戻り値

再割り当てされたメモリー・ブロックへポインターを戻します。 *ptr* 引数は、戻り値と同じではありません。 _debug_realloc メモリーが再割り当てされる前に解放されたメモリーへの参照を見つけやすくするために、常にメモリー・ロケーションを変更します。

size が 0 の場合、NULL を戻します。指定されたサイズにブロックを拡張するのに十分なメモリーが使用可能でない場合、元のブロックは未変更のままで、NULL が戻されます。

例

次の例では、_debug_realloc を使用して、100 バイトのストレージを割り当てます。続いて、割り当てられなかったストレージへの書き込みを試行します。_debug_realloc を再び呼び出したときに、_heap_check はエラーを検出し、いくつかのメッセージを生成し、そしてプログラムを停止します。

```
/*   _debug_realloc.c   */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *ptr;

   if (NULL == (ptr = (char*)realloc(NULL, 100))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'a', 105);     /* overwrites storage that was not allocated    */
   ptr = (char*)realloc(ptr, 200);        /*  realloc invokes _heap_check     */
   puts("_debug_realloc did not detect that a memory block was overwritten." );
   return 0;

   }
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 6161616161616161.
This memory block was (re)allocated at line number 8 in _debug_realloc.c.
Heap state was valid at line 8 of _debug_realloc.c.
Memory error detected at line 13 of _debug_realloc.c.
```

ストリング・ハンドリングのデバッグ関数

このセクションでは、標準 C ストリング・ハンドリング・ライブラリーのストリング処理とメモリー関数のデバッグ・バージョンについて記述しています。これらの関数は現在のデフォルト・ヒープのみについて検査します。複数のユーザー作成ヒープを使用するアプリケーション内のヒープはすべて検査されないことにご注意ください。

_debug_memcpy — バイトのコピー

形式

```
#include <string.h>
void *_debug_memcpy(void *dest, const void *src, size_t count, const char *file,
                                          size_t line);
```

目的

これは memcpy のデバッグ・バージョンです。 memcpy と同様、*src* の *count* バイトを *dest* にコピーします。ここで、オーバーラップしたオブジェクト間でコピーすると、未定義な振る舞いを引き起こすことになります。

_debug_memcpy は、コピー先にバイトをコピーしたのちにヒープを妥当性検査して、コピー先がヒープ内にある場合にのみこのチェックを行います。
_debug_memcpy は、暗黙の _heap_check 呼び出しを行います。 _debug_memcpy が、_heap_check を呼び出したときに破壊されたヒープを検出すると、_debug_memcpy はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターを *dest* へ戻します。

例

次の例には、プログラミング・エラーがあります。ターゲット・ロケーションを初期化するために使用される memcpy 呼び出しでは、カウントがターゲット・オブジェクトのサイズを超えているため、 memcpy 操作は割り当てられたオブジェクトの最後を超えてバイトをコピーします。

```
/*   _debug_memcpy.c  */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define  MAX_LEN       10

int main(void)
{
   char *source, *target;

   target = (char*)malloc(MAX_LEN);
   memcpy(target, "This is the target string", 11);
```

```
   printf("Target is \"%s\"\n", target);
   return 0;

}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 5468697320697320.
This memory block was (re)allocated at line number 11 in _debug_memcpy.c.
Heap state was valid at line 11 of _debug_memcpy.c.
Memory error detected at line 12 of _debug_memcpy.c.
```

_debug_memset - 値へのバイトの設定

形式

```
#include <string.h>
void *_debug_memset(void *dest, int c, size_t count, const char *file, size_t line);
```

目的

これは memset のデバッグ・バージョンです。 memset と同様、*dest* の最初の *count* バイトを値 *c* に設定します。 *c* の値は符号なし文字に変換されます。

_debug_memset は、バイトを設定したのちにヒープを妥当性検査して、コピー先がヒープ内にある場合にのみこのチェックを行います。 _debug_memset は、暗黙の _heap_check 呼び出しを行います。 _debug_memset が、_heap_check を呼び出したときに破壊されたヒープを検出すると、 _debug_memset はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターを *dest* へ戻します。

例

次の例には、プログラミング・エラーがあります。 'B' をバッファーに書き込む memset の呼び出しは、誤った count を指定しているため、バッファーの最後を超えてバイトを保管します。

```
/*  _debug_memset.c  */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define  BUF_SIZE      20

int main(void)
{
   char *buffer, *buffer2;
   char *string;

   buffer = (char*)calloc(1, BUF_SIZE+1);    /* +1 for null-terminator */

   string = (char*)memset(buffer, 'A', 10);
   printf("\nBuffer contents: %s\n", string);
   memset(buffer+10, 'B', 20);
```

```
   return 0;

   }
```

出力は以下に類似した表示になるはずです。

```
Buffer contents: AAAAAAAAAA
End of allocated object 0x00073c80 was overwritten at 0x00073c95.
The first eight bytes of the memory block (in hex) are: 4141414141414141.
This memory block was (re)allocated at line number 12 in _debug_memset.c.
Heap state was valid at line 14 of _debug_memset.c.
Memory error detected at line 16 of _debug_memset.c.
```

_debug_strcat — ストリングの連結

形式

これは strcat のデバッグ・バージョンです。 strcat と同様、*string2* を *string1* に連結して、結果のストリングの最後にヌル文字を付けます。

_debug_strcat は、ストリングを連結したのちにヒープを妥当性検査して、ターゲットがヒープ内にある場合にのみこのチェックを行います。 _debug_strcat は、暗黙の _heap_check 呼び出しを行います。 _debug_strcat が、_heap_check を呼び出したときに破壊されたヒープを検出すると、 _debug_strcat はそのファイル名 *file* と行番号 *file* をメッセージで報告します。

目的

```
#include <string.h>
char *_debug_strcat(char *string1, const char *string2, const char *file, size_t file);
```

戻り値

ポインターを連結されたストリング *string1* へ戻します。

例

次の例には、プログラミング・エラーがあります。 buffer1 オブジェクトは、ストリング「program」が連結された後の結果を保管するだけの十分な大きさがありません。

```
/*   _debug_strcat.hc   */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define  SIZE    10

int main(void)
{
   char *buffer1;
   char *ptr;

   buffer1 = (char*)malloc(SIZE);
   strcpy(buffer1, "computer");

   ptr = strcat(buffer1, " program");
   printf("buffer1 = %s\n", buffer1);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
This memory block was (re)allocated at line number 12 in _debug_strcat.c.
Heap state was valid at line 13 of _debug_strcat.c.
Memory error detected at line 15 of _debug_strcat.c.
```

_debug_strcpy — ストリングのコピー

形式

```
#include <string.h>
char *_debug_strcpy(char *string1, const char *string2, const char *file, size_t line);
```

目的

これは strcpy のデバッグ・バージョンです。 strcpy と同様、*string2* (末尾のヌル文字を含む) を、*string1* で指定されたロケーションへコピーします。

_debug_strcpy は、コピー先へストリングをコピーしたのちにヒープを妥当性検査して、コピー先がヒープ内にある場合にのみこのチェックを行います。 _debug_strcpy は、暗黙の _heap_check 呼び出しを行います。 _debug_strcpy が、_heap_check を呼び出したときに破壊されたヒープを検出すると、_debug_strcpy はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターをコピーされたストリング *string1* へ戻します。

例

次の例には、プログラミング・エラーがあります。コピー元のストリングがコピー先のバッファーには長すぎるため、strcpy 操作はヒープに損傷を与えます。

```
/*  _debug_strcpy.c  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define  SIZE          10

int main(void)
{
   char *source = "1234567890123456789";
   char *destination;
   char *return_string;

   destination = (char*)malloc(SIZE);
   strcpy(destination, "abcdefg"),

   printf("destination is originally = '%s'\n", destination);
   return_string = strcpy(destination, source);
   printf("After strcpy, destination becomes '%s'\n\n", destination);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
destination is originally = 'abcdefg'
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 3132333435363738.
```

```
This memory block was (re)allocated at line number 13 in _debug_strcpy.c.
Heap state was valid at line 14 of _debug_strcpy.c.
Memory error detected at line 17 of _debug_strcpy.c.
```

_debug_strncat — ストリングの連結

形式

```
#include <string.h>
char *_debug_strncat(char *string1, const char *string2, size_t count,
                                          const char *file, size_t line);
```

目的

これは strncat のデバッグ・バージョンです。 strncat と同様、*string2* の最初の count 個の文字を、*string1* へ付加し、結果のストリングをヌル文字 (0) で終わらせます。 *count* が *string2* の長さよりも大きい場合は、*string2* の長さが *count* の代わりに使用されます。

_debug_strncat は、文字を付加したのちにヒープを妥当性検査し、ターゲットがヒープ内にある場合にのみこのチェックを行います。 _debug_strncat は、暗黙の _heap_check 呼び出しを行います。 _debug_strncat が、_heap_check を呼び出したときに破壊されたヒープを検出すると、 _debug_strncat はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターを結合されたストリング *string1* へ戻します。

例

次の例には、プログラミング・エラーがあります。 buffer1 オブジェクトは、ストリング" programming"からの 8 文字が連結された後の結果を保管するには十分な大きさではありません。

```
/*  _debug_strncat.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define  SIZE          10

int main(void)
{
   char *buffer1;
   char *ptr;

   buffer1 = (char*)malloc(SIZE);
   strcpy(buffer1, "computer");

   /* Call strncat with buffer1 and " programming"                              */

   ptr = strncat(buffer1, " programming", 8);
   printf("strncat: buffer1 = \"%s\"\n", buffer1);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
This memory block was (re)allocated at line number 12 in _debug_strncat.c.
Heap state was valid at line 13 of _debug_strncat.c.
Memory error detected at line 17 of _debug_strncat.c.
```

_debug_strncpy — ストリングのコピー

形式

```
#include <string.h>
char *_debug_strncpy(char *string1, const char *string2, size_t count,
                     const char *file, size_t line);
```

目的

これは strncpy のデバッグ・バージョンです。 strncpy と同様、*string2* の *count* 個の文字を *string1* にコピーします。 *count* が *string2* の長さと等しいか、それ以下である場合は、コピーされたストリングにヌル文字 (0) は付加されません。 *count* が *string2* の長さより大きい場合、*string1* の結果は、*count* の長さになるまで、ヌル文字 (0) が埋め込まれます。

_debug_strncpy は、コピー先にストリングをコピーしたのちにヒープを妥当性検査して、コピー先がヒープ内にある場合にのみこのチェックを行います。 _debug_strncpy は、暗黙の _heap_check 呼び出しを行います。 _debug_strncpy が、_heap_check を呼び出したときに破壊されたヒープを検出すると、_debug_strncpy はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターを *string1* へ戻します。

例

次の例には、プログラミング・エラーがあります。コピー元のストリングがコピー先のバッファーには長すぎるため、 strncpy 操作はヒープに損傷を与えます。

```
/*  _debug_strncopy  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define  SIZE          10

int main(void)
{
   char *source = "1234567890123456789";
   char *destination;
   char *return_string;
   int index = 15;

   destination = (char*)malloc(SIZE);
   strcpy(destination, "abcdefg"),

   printf("destination is originally = '%s'\n", destination);
   return_string = strncpy(destination, source, index);
   printf("After strncpy, destination becomes '%s'\n\n", destination);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
destination is originally = 'abcdefg'
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 3132333435363738.
This memory block was (re)allocated at line number 14 in _debug_strncpy.c.
Heap state was valid at line 15 of _debug_strncpy.c.
Memory error detected at line 18 of _debug_strncpy.c.
```

_debug_strnset — ストリングでの文字の設定

形式

```
#include <string.h>
char *_debug_strnset(char *string, int c, size_t n, const char *file, size_t line);
```

目的

これは strnset のデバッグ・バージョンです。 strnset と同様、多くても *string* の最初の *n* 文字を *c* (char に変換される) に設定します。ただし、*n* が *string* の長さより大きい場合は、*string* の長さが *n* の代わりに使用されます。

_debug_strnset は、バイトを設定したのちにヒープを妥当性検査して、コピー先がヒープ内にある場合にのみこのチェックを行います。 _debug_strnset は、暗黙の _heap_check 呼び出しを行います。 _debug_strnset が、_heap_check を呼び出したときに破壊されたヒープを検出すると、 _debug_strnset はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターを変更された *string* へ戻します。エラー戻り値はありません。

例

次の例には、2 つのプログラム・エラーがあります。ストリング str は、ストリングの最後のマークであるヌル終止符がないまま作成されており、終了文字なしでは、カウントが 10 の strnset は、割り当てられたオブジェクトの最後を超えてバイトを保管します。

```
/*  _debug_strnset   */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *str;

   str = (char*)malloc(10);

   printf("This is the string after strnset: %s\n", str);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 7878787878797979.
This memory block was (re)allocated at line number 9 in _debug_strnset.c.
Heap state was valid at line 11 of _debug_strnset.c.
```

_debug_strset — ストリングでの文字の設定

形式

```
#include <string.h>
char *_debug_strset(char *string, size_t c, const char *file, size_t line);
```

目的

これは strset のデバッグ・バージョンです。 strset と同様、終了のヌル文字 (0) を除いて、*string* のすべての文字を、 *c* (char へ変換される) に設定します。

_debug_strset は、*string* のすべての文字を設定したのちにヒープを妥当性検査し、コピー先がヒープ内にある場合にのみこのチェックを行います。 _debug_strset は、暗黙の _heap_check 呼び出しを行います。 _debug_strset が、_heap_check を呼び出したときに破壊されたヒープを検出すると、 _debug_strset はそのファイル名 *file* と行番号 *line* をメッセージで報告します。

戻り値

ポインターを変更された string へ戻します。 エラー戻り値はありません。

例

次の例には、プログラミング・エラーがあります。ストリング str は、ヌル終止符なしで作成されているため、strset はヌル終止符と考えられるものを検出するまで、文字 'k' を設定し続けます。

```
/*  file: _debug_strset.c  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *str;

   str = (char*)malloc(10);

   strnset(str, 'x', 5);
   strset(str+5, 'k');
   printf("This is the string after strset: %s\n", str);
   return 0;
}
```

出力は以下に類似した表示になるはずです。

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 78787878786B6B6B.
This memory block was (re)allocated at line number 9 in _debug_strset.c.
Heap state was valid at line 11 of _debug_strset.c.
Memory error detected at line 12 of _debug_strset.c.
```

特記事項

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

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

〒106-8711
東京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

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

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

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

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

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

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

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

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

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

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

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

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

著作権使用許諾:

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

利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、 IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

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

商標

IBM、IBM ロゴ、および ibm.com は、International Business Machines Corporation の米国およびその他の国における商標です。この資料が公開された時点で、米国において IBM が所有する登録商標または商標には、初出時に商標記号 (® または ™) を付けて示しています。このような商標は、その他の国においても、登録商標または商標である可能性があります。現時点での IBM の商標については、http://www.ibm.com/legal/copytrade.shtml の「Copyright and trademark information」をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

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

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

Cell Broadband Engine, Cell/B.E は、米国およびその他の国における Sony Computer Entertainment, Inc. の商標であり、同社の許諾を受けて使用しています。

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

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

索引

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

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ラ行]

[数字]

B

F

H

I

L

M

O

S

X

[特殊文字]

プログラム番号: 5724-U80

Printed in Japan

SC88-4920-00

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