

XL C/C++ Enterprise Edition for AIX



プログラミング・ガイド

バージョン 7.0

XL C/C++ Enterprise Edition for AIX



プログラミング・ガイド

バージョン 7.0

ご注意

本書の情報およびそれによってサポートされる製品を使用する前に、111 ページの『特記事項』に記載する一般情報をお読みください。

本書は、XL C/C++ Enterprise Edition for AIX のバージョン 7.0 (プロダクト番号 5724-I11)、および新しい版で特に明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

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

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

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

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

(URL は、変更になる場合があります)

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

原 典： SC09-7888-00
XL C/C++ Enterprise Edition for AIX
Programming Guide
Version 7.0

発 行： 日本アイ・ビー・エム株式会社

担 当： ナショナル・ランゲージ・サポート

第1刷 2004.7

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

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

© Copyright International Business Machines Corporation 1998, 2004. All rights reserved.

© Copyright IBM Japan 2004

目次

本書について	v
文書規則	vi
強調表示の規則	vi
アイコン	vi
構文図の読み方	vii

第 1 章 32 ビット・モードおよび 64 ビット・モードの使用

long 値の割り当て	2
long 変数への定数値の割り当て	3
long 値のビット・シフト	3
ポインタの割り当て	4
集合体データ位置合わせ	4
Fortran コードの呼び出し	5

第 2 章 集合体内のデータの位置合わせ

位置合わせモードおよび修飾子の使用	7
位置合わせの一般的規則	10
位置合わせの例	10
ビット・フィールドの使用と位置合わせ	11
natural 位置合わせの規則	12
power 位置合わせの規則	12
Mac68K 位置合わせの規則	12
ビット・パック位置合わせの規則	13
ビット・フィールド位置合わせの例	13

第 3 章 浮動小数点演算の処理

乗加法演算の処理	15
浮動小数点丸めの処理	15
浮動小数点例外の処理	16
単精度のパフォーマンスと倍精度のパフォーマンス	17
数学関数加速サブシステム (MASS) の使用	17
スカラー・ライブラリーの使用	17
ベクトル・ライブラリーの使用	18
MASS によるプログラムのコンパイルとリンク	22

第 4 章 メモリー・ヒープの使用

複数のヒープを持つメモリーの管理	25
ユーザー作成ヒープの管理のための関数	26
ヒープの作成	27
ヒープの拡張	28
ヒープの使用	30
ヒープに関する情報の取得	30
ヒープのクローズおよび破棄	31
プログラムで使用されるデフォルト・ヒープの変更	32
ユーザー作成ヒープによるプログラムのコンパイルおよびリンク	32
ユーザー・ヒープの作成および使用例	32
メモリー・ヒープのデバッグ	37

メモリー・ヒープのチェックのための関数	38
メモリー・ヒープのデバッグのための関数	39
メモリー割り当て充てんパターンの使用	40
ヒープ・チェックのスキップ	41
スタック・トレースの使用	41

第 5 章 C++ テンプレートの使用

-qtempinc コンパイラー・オプションの使用	44
-qtempinc の例	44
テンプレート・インスタンス化ファイルの再生成	46
共用ライブラリーでの -qtempinc の使用	46
-qtemplateregistry コンパイラー・オプションの使用	47
関連コンパイル単位の再コンパイル	47
-qtempinc から -qtemplateregistry への切り替え	48

第 6 章 スレッド・セーフティーの確保 (C++)

テンプレート・オブジェクトのスレッド・セーフティーの確保	49
ストリーム・オブジェクトのスレッド・セーフティーの確保	50

第 7 章 ライブラリーの構成

ライブラリーのコンパイルとリンク	51
静的ライブラリーのコンパイル	51
共用ライブラリーのコンパイル	51
共用ライブラリー間のリンク	54
ライブラリー内の静的オブジェクトの初期化 (C++)	54
オブジェクトへの優先順位の割り当て	54
ライブラリー間のオブジェクト初期化の順序	57
共用ライブラリーの動的なロード	58
loadAndInit 関数を使用したモジュールのロードおよび初期化	59
terminateAndUnload 関数を使用したモジュールの終了およびアンロード	60

第 8 章 C++ ユーティリティーの使用

コンパイル済み C++ 名のデマングリング	63
コンパイル済み C++ 名の c++filt によるデマングリング	63
コンパイル済み C++ 名のデマングル・クラス・ライブラリーによるデマングリング	65
makeC++SharedLib ユーティリティーによる共用ライブラリーの作成	66
linkxlc ユーティリティーとのリンク	68

第 9 章 アプリケーションの最適化

最適化レベルの使用	70
最適化レベル 2 および 3 の最大活用	73
システム・アーキテクチャーの最適化	73
ターゲット・マシンのオプションの最大活用	74

高位ループ分析および変換の使用	75
-qhot の最大活用	76
共用メモリの並列処理の使用	76
-qsmp の最大活用	77
プロシージャ間分析の使用	78
-qipa の最大活用	79
プロファイル指示フィードバックの使用	79
pdf および showpdf によるコンパイルの例	81
その他の最適化オプション	82
最適化およびパフォーマンスに関するオプションの要約	83

第 10 章 パフォーマンスを向上させるためのアプリケーションのコーディング . . 85

高速入出力手法の検索	85
関数呼び出しによるオーバーヘッドの削減	86
効率的なメモリの管理	87
変数の最適化	88
効率的なストリングの操作	89
式とプログラム・ロジックの最適化	89
64 ビット・モードでの演算の最適化	90

付録. メモリー・デバッグ・ライブラリー

関数 93

メモリー割り当てデバッグ関数	93
_debug_calloc — メモリーの割り当てと初期化	93
_debug_free — 割り当てられたメモリーの解放	94

_debug_heapmin — デフォルト・ヒープ内の未使用メモリーの解放	95
_debug_malloc — メモリーの割り当て	96
_debug_calloc — ユーザーが作成したヒープからのメモリーの予約と初期化	97
_debug_uheapmin — ユーザーが作成したヒープ内の未使用メモリーの解放	98
_debug_umalloc — ユーザーが作成したヒープからのメモリー・ブロックの予約	99
_debug_realloc — メモリー・ブロックの再割り当て	100
ストリング処理デバッグ関数	102
_debug_memcpy — バイトのコピー	102
_debug_memmove — バイトのコピー	103
_debug_memset — 値へのバイトの設定	104
_debug_strcat — ストリングの連結	105
_debug_strcpy — ストリングのコピー	106
_debug_strncat — ストリングの連結	107
_debug_strncpy — ストリングのコピー	108
_debug_strnset — ストリングでの文字の設定	109
_debug_strset — ストリングでの文字の設定	109

特記事項. 111

プログラミング・インターフェース情報	112
商標	113
業界標準	113

本書について

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

本書では、以下のトピックを取り上げます。

- 1 ページの『第 1 章 32 ビット・モードおよび 64 ビット・モードの使用』では、既存の 32 ビット・アプリケーションを 64 ビット・モードに移植する際に生じる一般的な問題について説明し、それを回避するために推奨される方法について述べます。
- 7 ページの『第 2 章 集合体内のデータの位置合わせ』では、集合体 (構造体やクラスなど) 内のデータの位置合わせを制御する際にあらゆるプラットフォームで利用できるさまざまなコンパイラー・オプションについて説明します。
- 15 ページの『第 3 章 浮動小数点演算の処理』では、コンパイラーによる浮動小数点演算の処理方法を制御する際に使用できるオプションについて説明します。
- 25 ページの『第 4 章 メモリー・ヒープの使用』では、カスタム・メモリー・ヒープの使用やヒープ・メモリーの検証およびデバッグを含む、ヒープ・メモリー管理のコンパイラー・ライブラリー関数について説明します。
- 43 ページの『第 5 章 C++ テンプレートの使用』では、C++ テンプレートが組み込まれているプログラムをコンパイルする場合のさまざまなオプションについて説明します。
- 49 ページの『第 6 章 スレッド・セーフティーの確保 (C++)』では、入出力ストリーム、および標準テンプレートを含む、C++ クラス・ライブラリーに関するスレッド・セーフティー問題について説明します。
- 51 ページの『第 7 章 ライブラリーの構成』では、静的ライブラリーと共用ライブラリーのコンパイルおよびリンク方法について、および C++ プログラムで静的オブジェクトの初期化順序を指定する方法について説明します。
- 63 ページの『第 8 章 C++ ユーティリティの使用』では、コンパイルされたシンボル名をデマングリングし、共用ライブラリーを作成し、C++ モジュールをリンクするための、XL C/C++ Enterprise Edition for AIX と同梱される一部の追加ユーティリティについて説明します。
- 69 ページの『第 9 章 アプリケーションの最適化』では、プログラム最適化のためにコンパイラーが提供する各種オプションについて説明し、それぞれのオプションの推奨される使用方法を紹介します。
- 85 ページの『第 10 章 パフォーマンスを向上させるためのアプリケーションのコーディング』では、プログラムのパフォーマンスおよびコンパイラーの最適化機能との互換性を高めるために推奨されるプログラミングの実例とコーディング手法について説明します。

- 93 ページの『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および例を提供します。

文書規則

強調表示の規則

本書では、以下の強調表示規則を使用します。

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

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

モノスペース プログラム・コード例を示します。

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

アイコン

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

▶ AIX

AIX® プラットフォームでサポートされている機能を示します。

▶ Linux

Linux® プラットフォームでサポートされている機能を示します。

▶ Mac OS X

Mac OS X プラットフォームでサポートされている機能を示します。

▶ C++

C++ 言語でのみサポートされている機能を示します。

▶ C

C 言語でのみサポートされている機能を示します。

構文図の読み方

- 構文図は、左から右、上から下に、線のパスに従って読んでください。

▶▶— は、コマンド、ディレクティブ、またはステートメントの先頭を示します。

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

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

—▶◀ 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以外の構文単位の図は、▶— 記号で始まり、—▶ 記号で終わります。

注: 次の図で、`statement` は、C または C++ コマンド、ディレクティブ、またはステートメントを表しています。

- 必須項目は、水平線 (メインパス) 上に記述されます。

▶▶—`statement`—`required_item`—▶◀

- オプション項目は、メインパスの下に記述されます。

▶▶—`statement`—
|
|—`optional_item`—▶◀

- 2 つ以上の項目から選択可能な場合は、スタック内に垂直に記述されます。

いずれか 1 つの項目の選択が必須の場合は、スタック内の項目のいずれか 1 つがメインパス上に記述されます。

▶▶—`statement`—
|
|—`required_choice1`—
|
|—`required_choice2`—▶◀

いずれか 1 つの項目の選択がオプションの場合は、スタック全体がメインパスの下に記述されます。

▶▶—`statement`—
|
|—`optional_choice1`—
|
|—`optional_choice2`—▶◀

デフォルト項目は、メインパスの上に記述されます。

▶▶—`statement`—
|
|—`default_item`—
|
|—`alternate_item`—▶◀

- メインパスの線の上の左に戻る矢印は、繰り返し可能な項目を示します。

▶▶—`statement`—
|
|—`repeatable_item`—▶◀

スタックの上の繰り返し矢印は、スタック内の項目から複数の項目を選択するか、1 つの項目を繰り返し選択できることを示しています。

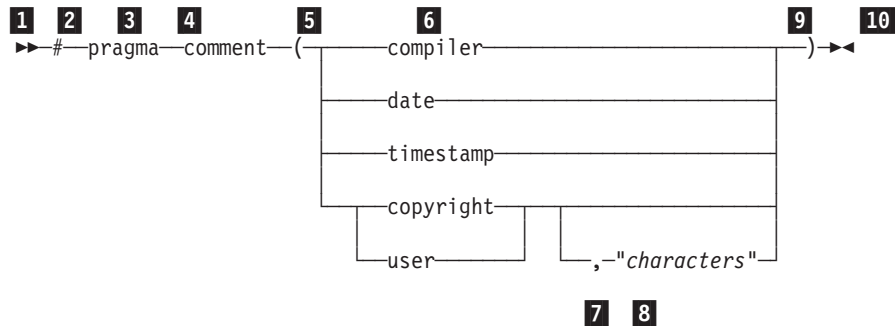
構文図の読み方

- キーワードは、非イタリック体で記述されています。示されているとおりに正確に入力する必要があります (例えば `extern`)。

変数は、イタリック体の小文字で記述されます (例えば、*identifier*)。変数は、ユーザー提供の名前または値を表します。

- 構文図に、句読記号、小括弧、算術演算子、または、ほかの同様な記号が示されている場合は、構文の一部としてこれらの文字を入力する必要があります。

以下の構文図の例では、**#pragma comment** ディレクティブの構文を示しています。



- 1 構文図の始まりを示します。
- 2 記号 `#` を最初に記述します。
- 3 キーワード `pragma` は、記号 `#` の次に記述されます。
- 4 プラグマの名前 `comment` は、キーワード `pragma` の次に記述します。
- 5 左括弧が必要です。
- 6 コメントの型を、表示されている `compiler`、`date`、`timestamp`、`copyright`、または `user` のうちいずれか 1 つだけ入力します。
- 7 コンマが、コメントの型 `copyright` または `user` とオプションの文字ストリングの間に必要です。
- 8 文字ストリングをコンマの次に記述します。文字ストリングは二重引用符で囲みます。
- 9 右小括弧は必須です。
- 10 これが、構文図の終わりを示します。

次の **#pragma comment** ディレクティブの例は、上記のダイアグラムに従っており、構文上正しい例です。

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

第 1 章 32 ビット・モードおよび 64 ビット・モードの使用

XL C/C++ を使用すると、32 ビット・アプリケーションと 64 ビット・アプリケーションの両方を開発することができます。それには、コンパイル時に、それぞれ、**-q32** (デフォルト) または **-q64** と指定してください。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/C++ long データ型および pointer データ型のサイズと位置合わせが、この 2 つのモード間で異なることに起因します。次の表は、その違いをまとめたものです。

表 1. 32 ビット・モードおよび 64 ビット・モードにおけるデータ型のサイズと位置合わせ

データ型	32 ビット・モード		64 ビット・モード	
	サイズ	位置合わせ	サイズ	位置合わせ
long, unsigned long	4 バイト	4 バイト境界	8 バイト	8 バイト境界
pointer	4 バイト	4 バイト境界	8 バイト	8 バイト境界
size_t (システム定義の unsigned long)	4 バイト	4 バイト境界	8 バイト	8 バイト境界
ptrdiff_t (システム定義の long)	4 バイト	4 バイト境界	8 バイト	8 バイト境界

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

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

32 ビット・モードまたは 64 ビット・モードでコンパイルする場合は、アプリケーションの移植に関連する一部の問題を診断するのに役立つ、**-qwarn64** オプションを使用することができます。いずれのモードでも、不具合 (切り捨てやデータ損失など) が発生した場合は、コンパイラーが即時に警告を発します。

64 ビット・モードでパフォーマンスを向上させるための提案については、90 ページの『64 ビット・モードでの演算の最適化』を参照してください。

関連資料

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

- 「コンパイラー・リファレンス」の「環境変数を設定して、64 ビット・モードまたは 32 ビット・モードを選択する」

long 値の割り当て

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

表 2. 32 ビット・モードおよび 64 ビット・モードにおける長整数の定数限界

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

この違いにより、次のような現象が生じます。

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

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

- 明示的な型キャストによって型を変更し、暗黙のうちに型変換されることのないようにする。
- long 型を戻す関数がすべて適切にプロトタイプ化されていることを確認する。
- long パラメーターを、それが渡される関数によって受け入れられるようにする。

long 変数への定数値の割り当て

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

- 定数 \geq **UINT_MAX**
- 定数 $<$ **INT_MIN**
- 定数 $>$ **INT_MAX**

境界での予期しない副次作用の例をいくつか、次の表に示します。

表 3. long 型に割り当てられる定数の、境界での予期しない結果

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

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

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

long 値のビット・シフト

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

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

表 4. `long` 値のビット・シフトの結果

初期値	シンボリック 定数	ビット・シフト後の値	
		32 ビット・モード	64 ビット・モード
0x7FFFFFFFL	INT_MAX	0xFFFFFFFFE	0x00000000FFFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x00000000100000000
0xFFFFFFFFFL	UINT_MAX	0xFFFFFFFFE	0x1FFFFFFFFFE

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。

次のようなコード構文

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

は、32 ビット・モードでは、`calloc` に対する関数プロトタイプがなくても有効ですが、コンパイラーは、この関数が `int` を戻すと想定するので、`a` は自動的に切り捨てられ、その後、符号拡張されます。`calloc` がメモリー内で割り振ったアドレスは、戻されるときに既に切り捨てられているため、結果をキャストする型は、切り捨てを免れることはできません。この例での適切な解決策は、`calloc` のプロトタイプを含む、該当するヘッダー・ファイル `stdlib.h` を組み込むことです。

上記のような問題を回避するためには、次のようにします。

- ポインターを戻す関数をプロトタイプ化する。
- 関数 (ポインターまたは `int`) 呼び出しで渡すパラメーターの型が、呼び出される関数で予想される型と一致するようにする。
- ポインターを整数型として処理するアプリケーションでは、32 ビット・モードでも 64 ビット・モードでも、`long` 型または `unsigned long` 型を使用する。

集合体データ位置合わせ

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

ポインターまたは `long` 型を含む構造体は、32 ビット・アプリケーションと 64 ビット・アプリケーションで共用することはできません。`long` 型と `int` 型を共用するか、あるいはポインターを `int` 型にオーバーレイする共用体は、変更されるか、

さもなければ位置合わせが破壊されます。通常は、最も単純なものを除くすべての構造体について、位置合わせとサイズの依存関係を調べる必要があります。

64 ビット・モードでは、値によって **va_arg** 引き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。これは、オペレーティング・システムの既知の制限です。

データ構造体 (ビット・フィールドを含む構造体など) の位置合わせについて詳しくは、7 ページの『第 2 章 集合体内のデータの位置合わせ』を参照してください。

Fortran コードの呼び出し

アプリケーションには、C と C++ と Fortran を併用して、お互いを呼び出したり、ファイルを共用したりするものが少なくありません。そのようなアプリケーションでデータのサイズや型を変更する場合、現時点では、Fortran サイドで行うよりも C サイドで行う方が簡単です。次の表は、C および C++ の型とそれに相当する Fortran の型を、モード別に示したものです。

表 5. C/C++ と Fortran の同等のデータ型

C/C++ の型	Fortran の型	
	32 ビット	64 ビット
signed int	INTEGER	INTEGER
signed long	INTEGER	INTEGER*8
unsigned long	LOGICAL	LOGICAL*8
pointer	INTEGER	INTEGER*8
		POINTER (4 バイト)
		POINTER*8 (8 バイト)

第 2 章 集合体内のデータの位置合わせ

XL C/C++ では、個々の変数、集合体のメンバー、集合体全体、およびコンパイル単位全体の各レベルでデータ位置合わせを指定するためのさまざまなメカニズムを用意しています。異なるプラットフォーム間で、あるいは 32 ビット・モードと 64 ビット・モードの間でアプリケーションの移植を行う場合は、それぞれの環境で利用できる位置合わせの設定の違いを考慮して、データの破損やパフォーマンスの低下を防ぐようにしてください。

『位置合わせモードおよび修飾子の使用』では、各種プラットフォームおよびアドレッシング・モデルでのすべてのデータ型に対する位置合わせのデフォルト設定、集合体および集合体メンバーの位置合わせの制御に使用できるオプション、および集合体位置合わせの一般規則について説明します。また、さまざまな位置合わせオプションに基づく構造体レイアウトの例も示します。

11 ページの『ビット・フィールドの使用と位置合わせ』では、その他の規則と、ビット・フィールドの使用と位置合わせに関する考慮事項について説明し、ビット・パック位置合わせの例を示します。

位置合わせモードおよび修飾子の使用

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

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

上記の各設定は、8 ページの表 6 で定義されています。

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

コンパイル・プロセスで、単一または複数ファイル内のすべての集合体の位置合わせを設定する

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

ファイル内の単一または複数の集合体の位置合わせを設定する

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

単一集合体の位置合わせを設定する

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

- 構造体宣言に **__attribute__((aligned(n)))** 型属性を組み込む。 n の値は、2 の正の累乗でなければなりません。 **__attribute__((aligned))** を集合体の型属性に使用する場合は正しい構文については、「C/C++ ランゲージ・リファレンス」の『型属性』を参照してください。
- 構造体宣言に **__align(n)** 指定子を組み込む。 n の値は 2 の正の累乗です。

位置合わせ修飾子 では、次のようなことができます。

集合体内のすべてのメンバーの位置合わせを設定する

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

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

集合体内の単一メンバーの位置合わせを設定する

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


表 6. 位置合わせ設定

データ型	ストレージ	位置合わせ設定とサポートされるプラットフォーム							
		natural	power	full	mac68k	twobyte	linuxppc	bit_packed ³	packed ³
		AIX Mac	AIX Mac	AIX	AIX Mac	AIX	Linux	AIX Mac Linux	AIX
_Bool (C), bool (C++)	1 バイト	1 バイト	1 バイト		1 バイト		適用外	1 バイト	
char, signed char, unsigned char	1 バイト	1 バイト	1 バイト		1 バイト		1 バイト	1 バイト	
wchar_t (32 ビット・モード)	2 バイト	2 バイト	2 バイト		2 バイト		2 バイト	1 バイト	
wchar_t (64 ビット・モード)	4 バイト	4 バイト	4 バイト		サポート対象外 ²		4 バイト	1 バイト	
int, unsigned int	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	

表 6. 位置合わせ設定 (続き)

データ型	ストレージ	位置合わせ設定とサポートされるプラットフォーム							
		natural	power	full	mac68k	twobyte	linuxppc	bit_packed ³	packed ³
		AIX Mac	AIX Mac	AIX	AIX Mac	AIX	Linux	AIX Mac Linux	AIX
short int, unsigned short int	2 バイト	2 バイト	2 バイト		2 バイト		2 バイト	1 バイト	
long int, unsigned long int (32 ビット・モード)	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
long int, unsigned long int (64 ビット・モード)	8 バイト	8 バイト	8 バイト		サポート対象外 ²		8 バイト	1 バイト	
long long	8 バイト	8 バイト	8 バイト		2 バイト		8 バイト	1 バイト	
float	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
double	8 バイト	8 バイト	注を参照 ¹		2 バイト		8 バイト	1 バイト	
long double	8 バイト	8 バイト	注を参照 ¹		2 バイト		8 バイト	1 バイト	
long double (-qlongdouble を指定)	16 バイト	16 バイト	注を参照 ¹		2 バイト		適用外	1 バイト	
pointer (32 ビット・モード)	4 バイト	4 バイト	4 バイト		2 バイト		4 バイト	1 バイト	
pointer (64 ビット・モード)	8 バイト	8 バイト	8 バイト		サポート対象外 ²		8 バイト	1 バイト	
注: 1. これらの型は、集合体の最初のメンバーに対しては natural 位置合わせを使用し、2 番目以降のメンバーに対しては 4 バイトまたは natural 位置合わせ (いずれか値の小さい方) を使用します。 2. この型のメンバーで集合体を宣言し、この位置合わせ設定でコンパイルしようとする、コンパイラーは警告メッセージを出し、該当するプラットフォームのデフォルトの位置合わせ設定を使用してコンパイルを行います。 3. 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。									

あるプラットフォーム上のアプリケーションでデータを生成し、そのデータを別のプラットフォーム上のアプリケーションで読み取る場合は、プラットフォームに依存しない位置合わせモード (**#pragma pack**, **qalign=bit_packed** など) を使用する必要があります。

注:  C++ コンパイラーは、基本クラスまたは仮想関数を含むクラスに対して、余分なフィールドを生成することがあります。これらの型のオブジェクトは、集合体に対する通常のマッピングに準拠していない可能性があります。

位置合わせの一般的規則

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

- すべての位置合わせ設定で、集合体のサイズ は、その位置合わせ値の倍数のうち、集合体のすべてのメンバーを内包することのできる最小の倍数となる。
- **mac68k** を除くすべての位置合わせ設定で、集合体の位置合わせ は、そのメンバーの最大の位置合わせ値と等しい。
- **mac68k** の位置合わせでは、すべての集合体は、その集合体のメンバーのデータ型にかかわらず、2 バイトで位置合わせされる。
- 位置合わせされる集合体はネストすることができ、ネストされた個々の集合体に適用できる位置合わせ規則は、ネストされた集合体の宣言時に有効になっている位置合わせモードによって決まる。

ビット・フィールドを含む集合体の位置合わせ規則については、11 ページの『ビット・フィールドの使用と位置合わせ』を参照してください。

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=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:

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|

関連資料

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

ビット・フィールドの使用と位置合わせ



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

▶ **C** C 言語では、ビット・フィールドを、**int** ではなく **char** または **short** に指定することができますが、XL C/C++ はそれらを **unsigned int** としてマップします。ビット・フィールドの長さは、その基本型の長さを超えることはできません。拡張モードでは、ビット・フィールドに対して **sizeof** 演算子を使用することができます。(ビット・フィールドに作用する **sizeof** 演算子は、常に 4 を返します。)



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

ただし、ビット・フィールドを含む集合体の位置合わせ規則は、指定する位置合わせ設定によって異なります。この規則については、以下で説明します。

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、次のメンバーを次の 8 バイトの境界に移動する 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、次のモードが 4 バイトの境界で開始するようになります。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。
-  **C** 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。
-  **C++** 長さ 0 のビット・フィールドのみを含む集合体では、宣言されたビット・フィールドの型とコンパイル・モードに応じて、長さが 4 バイトまたは 8 バイトになります。

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) 位置合わせする。
- ビット・フィールドは、現在のワードに圧縮される。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。
-  **C** 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。
-  **C++** 長さ 0 のビット・フィールドのみを含む集合体は、長さ 1 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・フィールド位置合わせの例

ビット・パックの例

下の例では、

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

```
pragma options align=reset
```

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

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

第 3 章 浮動小数点演算の処理

XL C/C++ は、およそ 10^{-38} から 10^{+38} の範囲で、10 進法で約 7 桁の精度を持つ単精度の浮動小数点数と、およそ 10^{-308} から 10^{+308} の範囲で、10 進法で約 16 桁の精度を持つ倍精度の浮動小数点数をサポートしています。4 倍精度の値の範囲は倍精度の値の範囲と同じですが、その精度は 10 進法で約 29 桁になります。

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

- 『乗加法演算の処理』
- 『浮動小数点丸めの処理』
- 16 ページの『浮動小数点例外の処理』
- 17 ページの『単精度のパフォーマンスと倍精度のパフォーマンス』
- 17 ページの『数学関数加速サブシステム (MASS) の使用』

乗加法演算の処理

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

関連資料

- 「コンパイラー・リファレンス」の **-qfloat**

浮動小数点丸めの処理

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

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

- **-qfloat=nofold** コンパイラー・オプションを使用して、浮動小数点計算のコンパイル時のフォールディングをすべて抑制する。

- **-y** コンパイラー・オプションを使用して、実行時に使用する丸めモードと一致する IEEE コンパイル時の丸めモードを指定する。特に別の値を指定しない限り、デフォルトでは、最近似値 への丸めモードになっています (別の値を指定するには、**builtins.h** ファイルで宣言した、XL C/C++ 組み込み関数 **__setrnd** を使用します)。

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

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

    volatile float one, three;

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

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

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

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

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

関連資料

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

浮動小数点例外の処理

デフォルトでは、ゼロによる除法、無限大による除法、オーバーフロー、アンダーフローなどの無効な演算は、実行時には無視されます。ただし、**-qflttrap** オプションを使用すると、このようなタイプの例外を検出することができます。さらに、適切なサポート・コードをプログラムに追加すると、例外が発生してもプログラムの実行を続けて、例外の原因となった演算の結果を修正することができます。

しかし、定数を含む浮動小数点計算は、コンパイル時にはフォールディングされるのが普通であるため、実行時に発生する可能性のある例外は生じません。 **-qflttrap**

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

関連資料

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

単精度のパフォーマンスと倍精度のパフォーマンス

-qarch=com オプションのデフォルト値、または値 **pwr**、**pwr2**、**pwrx**、**pwr2s**、または **p2sc** のいずれかを使用してアプリケーションをコンパイルする場合、サポートされているのは倍精度の計算のみです。これらのアーキテクチャーでは、結果を単精度に変換する必要がある場合は、そのとき有効になっている丸めモードに従って、丸めが適用されます。

これらのアーキテクチャーでは、明示的な丸め操作が要求されるので、単精度の計算は倍精度の計算よりも遅くなることがよくあります。**-qarch** のこれ以外のすべての値では、単精度命令は単精度演算に対して使用され、倍精度演算と同じ速度で実行されます。

PowerPC の浮動小数点プロセッサについて詳しくは、「*AIX Assembler Language Reference*」を参照してください。

関連資料

- 「コンパイラー・リファレンス」の **-qarch**

数学関数加速サブシステム (MASS) の使用

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

AIX の MASS ライブラリーは、『スカラー・ライブラリーの使用』で説明されているスカラー関数のライブラリー、および 18 ページの『ベクトル・ライブラリーの使用』で説明されている特定のアーキテクチャー用に調整されたベクトル・ライブラリーのセットで構成されます。22 ページの『MASS によるプログラムのコンパイルとリンク』では、MASS ライブラリーを使用するプログラムのコンパイル方法とリンク方法、および MASS スカラー・ライブラリー関数を通常の **libm.a** スカラー関数とともに選択的に使用方法について説明します。

スカラー・ライブラリーの使用

MASS スカラー・ライブラリー **libmass.a** には、AIX システム・ライブラリー **libm.a** で頻繁に使用される数学組み込み関数の上級セットが含まれています。これらの関数は、すべて倍精度のパラメーターを受け入れ、倍精度の結果を戻します。18 ページの表 7 にまとめられています。関数のプロトタイプを提供するには、ソース・ファイルに **math.h** を追加します。

表 7. MASS スカラー・ライブラリー関数

関数	説明	プロトタイプ
sqrt	x の平方根を返す	double sqrt (double x);
rsqrt	x の平方根の逆数を返す	double rsqrt (double x);
exp	x の指数関数を返す	double exp (double x);
log	x の自然対数を返す。	double log (double x);
sin	x のサインを返す。	double sin (double x);
cos	x のコサインを返す。	double cos (double x);
tan	x のタンジェントを返す。	double tan (double x);
atan	x のアークタンジェントを返す。	double atan (double x);
atan2	x/y のアークタンジェントを返す。	double atan2 (double x, double y);
sinh	x の双曲線サインを返す。	double sinh (double x);
cosh	x の双曲線コサインを返す。	double cosh (double x);
tanh	x の双曲線タンジェントを返す。	double tanh (double x);
dnint	x (倍数) の最も近い整数を返す。	double dnint (double x);
pow	x の y 乗を返す。	double pow (double x, double y);

三角関数 (**sin**、**cos**、**tan**) は、大きい引き数 ($\text{abs}(x) > 2 \times 50 \times \pi$) の NaN (非数字) を返します。

注: 場合によっては、MASS 関数は、**libm.a** のように正確ではなく、エッジ・ケース (例えば、**sqrt(Inf)**) を異なる方法で処理する可能性があります。

ベクトル・ライブラリーの使用

MASS ベクトル・ライブラリーは、以下のアーカイブで提供されます。

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。POWER5 を使用する場合は、このライブラリーを選択することをお勧めします。

いくつかの関数 (以下で説明されています) を除き、**libmassv.a**、**libmassvp3.a**、および **libmassvp4.a** のすべての関数は、倍精度または単精度のベクトル入力パラメーター、倍精度または単精度の出力パラメーター、および整数の `vector-length` パラメーターの 3 つのパラメーターを受け入れます。これらの関数の形式は、`function_name (y,x,n)` です。ここで、`x` はソース・ベクトル、`y` はターゲット・ベクトル、および `n` はベクトルの長さです。パラメーター `y` および `x` は、プレフィックス **v** が付いた関数では倍精度、プレフィックス **vs** が付いた関数では単精度を想定します。例えば、以下のコードのようになります。

```
#include <massv.h>
```

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

$i=0, \dots, 499$ の場合に、エレメントが $\exp(x[i])$ である長さ 500 のベクトル y を出力します。

ベクトル・ライブラリーに含まれている単精度関数と倍精度関数については、表 8 にまとめられています。関数のプロトタイプを提供するには、ソース・ファイルに **massv.h** を追加します。C および C++ アプリケーションでは、スカラー引き数を使用する場合でも、サポートされているのは参照による呼び出しのみです。

表 8. MASS ベクトル・ライブラリー関数

倍精度関数	単精度関数	説明	倍精度関数プロトタイプ	単精度関数プロトタイプ
vacos	vsacos	$i=0, \dots, n-1$ 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);
vasin	vsasin	$i=0, \dots, n-1$ 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);
vatan2	vsatan2	$i=0, \dots, n-1$ 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);
vcos	vscos	$i=0, \dots, n-1$ 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	$i=0, \dots, n-1$ 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	$i=0, \dots, n-1$ の場合に、 $y[i]$ の実数部を $x[i]$ のコサインに、 $y[i]$ 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		$i=0, \dots, n-1$ の場合に、 $y[i]$ を $x[i]$ の整数切り捨てに設定する	void vdint (double y[], double x[], int *n);	

表 8. MASS ベクトル・ライブラリー関数 (続き)

vdiv	vsdiv	i=0,...,*n-1 の場合に、z[i] を x[i]/y[i] に設定する	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdnint		i=0,...,n-1 の場合に、y[i] を x[i] に最も近い整数に設定する	void vdnint (double y[], double x[], int *n);	
vexp	vsexp	i=0,...,*n-1 の場合に、y[i] を x[i] の指数関数に設定する	void vexp (double y[], double x[], int *n);	void vsexp (float y[], float x[], int *n);
vexpm1	vsexpm1	i=0,...,*n-1 の場合に、y[i] を (x[i] の指数関数)-1 に設定する	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);
vlog	vslog	i=0,...,*n-1 の場合に、y[i] を x[i] の自然対数に設定する	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog10	vslog10	i=0,...,*n-1 の場合に、y[i] を x[i] の底が 10 の対数に設定する	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	i=0,...,*n-1 の場合に、y[i] を (x[i]+1) の自然対数に設定する	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vpow	vspow	i=0,...,*n-1 の場合に、z[i] を x[i] の y[i] 乗に設定する	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vrec	vsrec	i=0,...,*n-1 の場合に、y[i] を x[i] の逆数に設定する	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);
vrsqrt	vsrsqrt	i=0,...,*n-1 の場合に、y[i] を x[i] の平方根の逆数に設定する	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
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、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);

表 8. MASS ベクトル・ライブラリー関数 (続き)

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);
<p>注:</p> <p>1. デフォルトでは、これらの関数は、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ない、__Complex データ型を使用します。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。</p>				

関数 **vdiv**、**vsincos**、および **vatan2** は、4 つのパラメーターを取ります。関数 **vdiv** および **vatan2** は、パラメーター (z, x, y, n) を取ります。関数 **vdiv** は、 $i=0, \dots, *n-1$ の場合にエレメントが $x[i]/y[i]$ である、ベクトル z を出力します。関数 **vatan2** は、 $i=0, \dots, *n-1$ の場合にエレメントが $\text{atan}(x[i]/y[i])$ である、ベクトル z を出力します。関数 **vsincos** は、パラメーター (y, z, x, n) を取り、エレメントがそれぞれ $\sin(x[i])$ と $\cos(x[i])$ である、2 つのベクトル y と z を出力します。

vcosisin(y,x,n) では、 x は、 n **double** エレメントのベクトルであり、関数は、形式 $(\cos(x[i]), \sin(x[i]))$ の n **double complex** エレメントのベクトル y を出力します。**-D__nocomplex** を使用する場合は (19 の「注」を参照)、出力ベクトルは、 $i=0, \dots, *n-1$ の場合に $y[0][i] = \cos(x[i])$ および $y[1][i] = \sin(x[i])$ となります。

MASS ベクトル関数の整合性

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、最小重みビットのみ) 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。

ただし、**libmassvp4.a** ライブラリーは、整合性のある新しいバージョンの関数を提供しています。これらの整合性のある関数とは、**vsqrt**、**vssqrt**、**vlog**、**vrec**、**vdiv**、**vsin**、**vcos**、**vacos**、**vasin**、**vatan2**、**vrsqrt**、**vscos**、**vsdiv**、**vexp**、**vsrec**、**vssin** です。

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。

ベクトル・ライブラリーとの整合性および不整合の回避、パフォーマンスと精度のデータについて詳しくは、MASS Web サイト

(<http://www.ibm.com/software/awdtools/vacpp/mass>) を参照してください。

関連資料

- 「コンパイラー・リファレンス」の **-D**

MASS によるプログラムのコンパイルとリンク

これらのライブラリーでルーチン呼び出すアプリケーションをコンパイルするには、**-l** リンカー・オプションで、**mass** および **massv** (あるいは **massvp3** または **massvp4**) を指定します。例えば、デフォルト・ディレクトリー **/usr/lib** に MASS ライブラリーがインストールされている場合は、次を指定します。

```
xlc prog.c -o progf -lmass -lmassv
```

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

libm.a での libmass.a の使用

一部の関数で **libmass.a** スカラー・ライブラリーを使用して、他の関数で通常の **libm.a** を使用したい場合は、以下の手順に従って、ご使用のプログラムをコンパイルおよびリンクしてください。

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 **fast_tan.exp** というファイルを作成します。

```
tan
```

2. AIX **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します。次に例を示します。

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

3. AIX **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 fast_tan.o
```

4. **xlc** を使用して、最終的な実行可能ファイルを作成し、標準の数学ライブラリー **libm.a** の前に MASS 関数が含まれているオブジェクト・ファイルを指定します。これは、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)、および標準のシステム・ライブラリーの数学関数の残りの関数のみをリンクします。次に例を示します。

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

注: MASS **cos** 関数は、MASS **sin** をエクスポートすると自動的にリンクされます。MASS **atan2** は、MASS **atan** をエクスポートすると、自動的にリンクされます。

関連資料:

- 「AIX コマンド解説書」の **ld**
- 「AIX コマンド解説書」の **ar**

第 4 章 メモリー・ヒープの使用

ANSI で定義されたメモリー管理関数に加えて、XL C/C++ では、プログラム・パフォーマンスの向上およびプログラムのデバッグに役立つメモリー管理関数の拡張バージョンが提供されます。これらの関数によって、以下のことができます。

- ユーザー作成ヒープとして知られる、メモリーの複数のカスタム定義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・ヒープでメモリー問題をデバッグする。
- ユーザー作成ヒープでメモリー問題をデバッグする。

メモリー管理関数のすべてのバージョンは、実際に同じように動作します。異なるところは、どのヒープから割り当てるか、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』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の管理に使用できる関数をまとめ、ユーザー定義ヒープの作成、拡張、使用および破壊の手順を提供し、正期メモリーおよび共用メモリーを使用してユーザーのヒープを作成するプログラムの例を提供します。

37 ページの『メモリー・ヒープのデバッグ』は、デフォルトのユーザー作成ヒープの検査およびデバッグに使用可能な関数について説明します。

複数のヒープを持つメモリーの管理

XL C/C++ を使用して、デフォルトの XL C ランタイム・ヒープの代わりに、またはそれに追加して、ユーザー独自のメモリー・ヒープを作成および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・ヒープを作成したヒープに変更することができます。

ユーザー独自のヒープを使用するかどうかはオプションです。アプリケーションは、XL C/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良好に動作します。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、およびそのブロックに含まれるデータは、少数のページにかたまっているため、スワッピングの必要量は減少します。

- 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、または解放されるように、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 が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場合は、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ることができます。

- 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、リスト全体を探索して、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 つだけある場合は、すべてのコンポーネント (XL C/C++ ランタイム・ライブラリー、バンダー・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ろになります。

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。

以下のセクションでは、複数のヒープを使用するために利用可能な関数、複数のヒープの作成、使用、および破壊のためのプログラミングのガイドライン、および複数のヒープをインプリメントするコードの例について説明します。

ユーザー作成ヒープの管理のための関数

libhu.a ライブラリーでは、ユーザー作成ヒープを管理できる関数セットを提供します。これらの関数はすべてプレフィックス **_u** (「ユーザー」ヒープ用) が付いていて、ヘッダー・ファイル **umalloc.h** で宣言されています。以下の表で、ユーザー定義ヒープの作成および管理に使用できる関数についてまとめています。

表 9. メモリー・ヒープの管理のための関数

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
適用外	<code>_ucreate</code>	ヒープを作成。『ヒープの作成』に説明。
適用外	<code>_uopen</code>	プロセスによって使用のためにヒープを開く。30 ページの『ヒープの使用』に説明。
適用外	<code>_ustats</code>	ヒープに関する情報を提供。30 ページの『ヒープに関する情報の取得』に説明。
適用外	<code>_uaddmem</code>	ヒープにメモリー・ブロックを追加。28 ページの『ヒープの拡張』に説明。
適用外	<code>_uclose</code>	プロセスによってさらなる使用からヒープを閉じる。31 ページの『ヒープのクローズおよび破棄』に説明。
適用外	<code>_udestroy</code>	ヒープを破棄。31 ページの『ヒープのクローズおよび破棄』に説明。
<code>calloc</code>	<code>_ucalloc</code>	作成したヒープからメモリーを割り振り、初期化する。30 ページの『ヒープの使用』に説明。
<code>malloc</code>	<code>_umalloc</code>	作成したヒープからメモリーを割り振る。30 ページの『ヒープの使用』に説明。
<code>_heapmin</code>	<code>_uheapmin</code>	システムに未使用のメモリーを戻す。31 ページの『ヒープのクローズおよび破棄』に説明。
適用外	<code>_udefault</code>	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。32 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』に説明。

注: **realloc** にも **free** にも、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が割り当てられるかを常に判別し、ユーザー作成ヒープとデフォルトのメモリー・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・ヒープを作成できます。固定サイズ・ヒープでは、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・ヒープでは、プログラムの要求に応じて、ヒープは拡張および縮小できます。両タイプのヒープを作成する手順は、以下に提供されています。

固定サイズ・ヒープの作成

固定サイズ・ヒープを作成する場合は、そのヒープを保持し、ヒープ管理に必要な内部情報を保持するの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、それにハンドルを割り振ります。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];
```

内部情報は、**_HEAP_MIN_SIZE** マクロ (**umalloc.h** に定義) によって指定される、最小セットのバイトが必要です。必要なブロックのサイズを判別するには、プ

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、正規または共用を指定します。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。0 に設定されている (例えば、**memset** によって) 場合は、**_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 に初期化されていれば、これらの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、共用 (**_HEAP_SHARED**) であるかを示します。

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・ヒープの作成

XL C/C++ デフォルト・ヒープを使用すると、**malloc** 要求を満たすために十分なストレージが使用できないときに、ランタイム環境がシステムから追加ストレージを取得します。同様に、**_heapmin** を実行し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、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独自の関数を指定し、これにはユーザーが選択するどのような名前でも付けることができます。これらの関数のポインターを、**_ucreate** の最後の 2 つのパラメーターとして (固定サイズ・ヒープを作成するのに使用した **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・サイズを増やすには、以下を実行して、それにメモリーの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・ヒープの場合、**_uaddmem** 関数を呼び出す。
- 動的サイズ・ヒープのみの場合、ヒープを拡張し、必要に応じて、ヒープからメモリーを割り振るときはいつでも、システムによって自動的に呼び出されることができる関数を書き込む。

両方のオプションについては、以下に説明しています。

ヒープへのメモリーのブロックの追加

_uaddmem を使用してメモリーのブロックを固定サイズ・ヒープまたは動的サイズ・ヒープ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。オーバーヘッドの総量を少なくするためには、小さなメモリー・ブロックをたくさん追加するよりも、大きなメモリー・ブロックを少しだけ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・ヒープに対して **_umalloc** (または同様の関数) を呼び出すと、**_umalloc** は、**_ucreate** で指定した初期ブロックからメモリーを割り当てようとします。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** にパラメーターとして指定したヒープ拡張関数を呼び出します。次に関数は、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。

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);
```

ここで、*functionName* は関数 (希望の名前を付けることができます) を示し、*uh* は拡張するヒープ、*size* は **_umalloc** によって渡される割り当て要求のサイズです。数回の割り当てを満たすため、十分なメモリーを一度に戻すことができます。そうしないと、その後の割り当てのたびにヒープ拡張関数を呼び出さなければならなくなり、これによってプログラムの実行速度が落ちます。要求した *size* を超えるメモリーに戻す場合には、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。

以下のコーディング部分では、ヒープ拡張関数の例を示します。


```
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、そのヒープを開いて使用することができます。

```
_uopen(fixedHeap);
```

これで、その特定プロセスに対してヒープが開きます。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の割り当ておよび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、使用するヒープも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られた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取得

_mheap を呼び出して、割り当てられたオブジェクトから、ヒープを特定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。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の大きさ
- ヒープにあるメモリのタイプ
- ヒープから使用可能な最大の連続メモリ部分のサイズ

ヒープのクローズおよび破棄

ヒープを使用してプロセスが終了したら、**_uclose** を実行し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리를割り当てることも、そのヒープにメモ리를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、以下を実行します。

- 固定サイズ・ヒープの場合、**_udestroy** を呼び出す。メモリ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・ヒープの場合、**_uheapmin** を呼び出してヒープを合体させるか (完全に解放されたヒープにあるすべてのブロックをシステムに戻す)、**_udestroy** を呼び出して破壊する。これら両方の関数がヒープ縮小関数を呼び出します。(以下参照。)

ヒープを破棄した後で、そのヒープ用のメモリー (**_ucreate** で指定した初期メモリー・ブロック、および **_uaddmem** で追加したその他のブロック) 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・ヒープを合体または破壊するために **_uheapmin** または **_udestroy** を呼び出すと、これらの関数はメモ리를システムに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インプリメントするかは、ユーザー次第です。

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);
```

ここで、*functionName* は関数 (希望の名前を付けることができます) を示し、*uh* は、縮小するヒープを示します。ポインター *block* と、その *size* は、**_uheapmin** または **_udestroy** によってユーザー関数に渡されます。ユーザー関数は *block* で指示されたメモ리를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

正規メモリ管理関数 (**malloc** など) は、そのスレッドに現在のデフォルト・ヒープを常に使用します。すべての XL C/C++ アプリケーションの初期デフォルト・ヒープは、XL C/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・ヒープからメモリを割り当てます。

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されます。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/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 (ペンダー・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。また、**_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って、デフォルトを XL C/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当てすることもできます。

ユーザー作成ヒープによるプログラムのコンパイルおよびリンク

任意のユーザー作成ヒープ関数 (プレフィックス **_u**) を呼び出すアプリケーションをコンパイルするには、**-l** リンカー・オプションで **hu** を指定します。例えば、**libhu.a** ライブラリーがデフォルトのディレクトリーにインストールされている場合は、以下のように指定します。

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

ユーザー・ヒープの作成および使用例

正規メモリーによるユーザー・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 - 親プロセスの例

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ます。まず、メインプログラムは **init** 関数を呼び出し、(**CreateFileMapping** を使用して) オペレーティング・システムから共用メモリーを割り振り、メモリーに名前を付け、他のプロセスが名前ですれを使用できるようにします。その後、**init** 関数がヒープを作成し、開きます。メインプログラムのループは、ヒープで操作を実行し、他のプロセスも開始します。プログラムは、**term** 関数を呼び出し、ヒープを閉じて破壊します。

```

#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 - 子プロセスの例

次のプログラムは、親プロセスのループで開始されたプロセスを示しています。このプロセスは **OpenFileMapping** を使用して名前でも共用メモリーにアクセスし、親プロセスが作成したヒープのヒープ・ハンドルを抽出します。その後、プロセスはヒープを開き、それをデフォルト・ヒープにし、ループのヒープで一部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・ヒープを置き換え、ユーザー・ヒープを閉じて終了します。

```

#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 run-time 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/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。（38 ページの『メモリー・ヒープのチェックのための関数』で説明）
- すべてのメモリー管理関数のデバッグ・バージョン。（39 ページの『メモリー・ヒープのデバッグのための関数』で説明）

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。選択するものは、ユーザーのプログラム、問題、好みによって異なります。

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。ユーザーは、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作成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。

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当て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コードを変更する必要はありません。 **-qheapdebug** コンパイラー・オプションを指定だけです。

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方法をお勧めします。ヒープが破壊されていることがわかった場合には、 **-qheapdebug** オプションを指定して再作成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。

関連資料

- 93 ページの『メモリー・デバッグ・ライブラリー関数』

メモリー・ヒープのチェックのための関数

ヘッダー・ファイル **umalloc.h** は、ユーザー作成ヒープを検証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。これらの関数の正規バージョンは、プレフィックス **_u** なしで、デフォルト・ヒープの検査にも使用可能です。ヒープ・チェック関数については、次の表にまとめています。

表 10. メモリー・ヒープのチェックのための関数

デフォルトのヒープ関数	ユーザー作成ヒープ関数	説明
<code>_heapchk</code>	<code>_uheapchk</code>	最小の整合性についてヒープ全体をチェックする。
<code>_heapset</code>	<code>_uheapset</code>	最小の整合性についてヒープ内の空きメモリーをチェックし、ヒープ内の空きメモリーを指定した値に設定する。
<code>_heap_walk</code>	<code>_uheap_walk</code>	ヒープを全探索して、割り当て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する。

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、32 ページの『ユーザー作成ヒープによるプログラムのコンパイルおよびリンク』を参照してください。

メモリー・ヒープのデバッグのための関数

デバッグ・バージョンは正規メモリー管理関数とユーザー定義ヒープ・メモリー管理関数の両方に使用可能です。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、それらを共用メモリーを含む、どんなタイプのヒープにも使用できます。デバッグ関数に行う各呼び出しはまた、**_heap_check** を呼び出すことによって自動的にヒープをチェックし (以下に説明)、ファイル名および行番号を含む情報を提供するため、メモリー問題をデバッグするために使用することができます。ユーザー定義デバッグ・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**)、この名前は **umalloc.h** で定義されます。

すべてのデバッグ・メモリー管理関数についての完全なリストおよび詳細については、93 ページの『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 11. メモリー・ヒープのデバッグのための関数

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
<code>_debug_calloc</code>	<code>_debug_ucalloc</code>
<code>_debug_malloc</code>	<code>_debug_umalloc</code>
<code>_debug_heapmin</code>	<code>_debug_uheapmin</code>
<code>_debug_realloc</code>	適用外
<code>_debug_free</code>	適用外

これらのデバッグ・バージョンを使用するには、以下のいずれかを行うことができます。

- ・ソース・コードで、任意のデフォルトまたはユーザー定義ヒープ・メモリー管理関数にプレフィックス **_debug_** を付ける。
- ・ソース・コードを変更したくない場合は、**-qheapdebug** オプションで単純にコンパイルする。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対応するデバッグ・バージョンにマップします。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、32 ページの『ユーザー作成ヒープによるプログラムのコンパイルおよびリンク』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** と一緒に使用しないでください。

3. **#pragma strings (読み取り専用)** 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・ヒープのデバッグのための追加関数

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。これらについては、次の表にまとめています。

表 12. メモリー・ヒープのデバッグのための追加関数

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
<code>_dump_allocated</code>	<code>_udump_allocated</code>	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。
<code>_dump_allocated_delta</code>	<code>_udump_allocated_delta</code>	_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振られた各メモリー・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。
<code>_heap_check</code>	<code>_uheap_check</code>	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。この関数を明示的に呼び出すこともできます。すると、それ以後は、**_dump_allocated** または **_dump_allocated_delta** を使用して、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ことができます。これらの関数を明示的に呼び出す必要があります。

メモリー割り当て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、デフォルトでは、割り当てたメモリーを **0xAA** の繰り返し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 **0xFB** の繰り返し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せんが、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** 環境変数で、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。

他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、

<i>increment</i>	実行しているヒープ・チェック間でスキップするデバッグ関数呼び出し数を指定します。
<i>start</i>	ヒープ・チェックを開始する前にスキップするデバッグ関数呼び出し数を指定します。

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交換条件となります。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 5 章 C++ テンプレートの使用

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

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

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

冗長なインスタンス生成の問題に対処するには、基本的に次の 4 つの方法があります。

固有のインスタンス生成のためのコーディングを行う

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

出現するたびにインスタンスを生成する

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

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

-qtempinc コンパイラー・オプションを使用します。テンプレート定義ファイルとテンプレート・インプリメンテーション・ファイルの構造が所定のものである場合は、テンプレートで生成された個々のインスタンスはテンプレート・インクルード・ディレクトリーに保管されます。コンパイラーは、同じテンプレートのインスタンスを同じ引き数で再び生成するように要求されると、新たに生成する代わりに保管したバージョンを使用します。この方法については、44 ページの『**-qtempinc** コンパイラー・オプションの使用』で説明します。

コンパイラーに、インスタンス生成情報をレジストリーに保管するよう指示する

-qtemplatergistry コンパイラー・オプションを使用します (これが、デフォルト設定です)。テンプレートによる個々のインスタンス生成に関する情報が、テンプレート・レジストリーに保管されます。コンパイラーは、同じテンプレートのインスタンスを同じ引き数で再び生成するように要求されると、新たに生成する代わりに、最初のオブジェクト・ファイルにあるインスタンス生成をポイントします。 **-qtemplatergistry** コンパイラー・オプション

ョンには、**-qtempinc** コンパイラー・オプションと同様の利点がありますが、テンプレート定義ファイルおよびテンプレート・インプリメンテーション・ファイルのための特定の構造は必要ありません。この方法については、47 ページの『**-qtemplateregistry** コンパイラー・オプションの使用』で説明します。

注: **-qtempinc** コンパイラー・オプションと **-qtemplateregistry** コンパイラー・オプションは、同時に使用することはできません。

-qtempinc コンパイラー・オプションの使用

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

- テンプレート・ヘッダー・ファイルで、クラス・テンプレートと関数テンプレートを拡張子 **.h** を付けて宣言する。
- テンプレート宣言ファイルごとに、テンプレート・インプリメンテーション・ファイルを作成する。このファイルの名前は、テンプレート宣言ファイルの名前と同じで拡張子が **.c** または **.t** であるか、あるいは **#pragma implementation** ディレクティブで指定する必要があります。クラス・テンプレートの場合は、インプリメンテーション・ファイルがメンバー関数と静的データ・メンバーを定義します。関数テンプレートの場合は、インプリメンテーション・ファイルはその関数を定義します。
- ソース・プログラムで、個々のテンプレート宣言ファイルに対して **#include** ディレクティブを指定する。
- (オプション) コードが **-qtempinc** コンパイルと **-qnotempinc** コンパイルの両方に適用できることを確認するためには、個々のテンプレート宣言ファイルに、**__TEMPINC__** マクロが定義されていない ことを条件に、対応するテンプレート・インプリメンテーション・ファイルを組み込む。(このマクロは、**-qtempinc** コンパイル・オプションを使用すると、自動的に定義されます。)

こうすると、次のような結果が得られます。

- **-qnotempinc** を指定してコンパイルすると、必ず、テンプレート・インプリメンテーション・ファイルが組み込まれます。
- **-qtempinc** を指定してコンパイルすると、コンパイラーは、テンプレート・インプリメンテーション・ファイルを組み込みません。その代わりに、コンパイラーは、特定のインスタンス生成が最初に必要になったときに、テンプレート・インプリメンテーション・ファイルと名前が同じで、拡張子が **.c** であるファイルを探します。これ以後は、同じインスタンス生成が必要になると、コンパイラーは、テンプレート・インクルード・ディレクトリーに保管されているコピーを使用します。

-qtempinc の例

この例には、次のソース・ファイルが含まれています。

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

- それに対応する関数インプリメンテーション・ファイル: stackops.cpp
- メインプログラムのソース・ファイル: stackadd.cpp

この例では、

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

テンプレート宣言ファイル: `stack.h`

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

```
#ifndef STACK_H
#define STACK_H

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

#ifdef __USE_STL_TEMPINC__ // 3
#include "stack.c" // 3
#endif
```

テンプレート・インプリメンテーション・ファイル: `stack.c`

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

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

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

関数宣言ファイル: `stackops.h`

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

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


関数インプリメンテーション・ファイル: stackops.cpp

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

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

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

メインプログラム・ファイル: stackadd.cpp

このファイルで、Stack オブジェクトが作成されます。

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

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

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

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

    return(0);
}
```

テンプレート・インスタンス化ファイルの再生成

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

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

共用ライブラリーでの -qtempinc の使用

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

-qtempinc を使用する場合は、次のことに注意してください。

- アプリケーションごとに、独自の **TEMPINC** 宛先が必要です。

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

関連資料

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

-qtemplateregistry コンパイラー・オプションの使用

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

テンプレート・レジストリーでは、「先着順」のアルゴリズムが使用されます。

- プログラムが新規のインスタンス生成を初めて参照するとき、そのプログラムのインスタンスは、それが発生するコンパイル単位で生成されます。
- 別のコンパイル単位が同じインスタンス生成を参照すると、そのコンパイル単位のインスタンスは生成されません。つまり、プログラム全体で生成されるコピーは 1 つだけです。

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

テンプレート・レジストリー・ファイルのデフォルトのファイル名は **templateregistry** ですが、他の有効なファイル名を指定して、このデフォルト名をオーバーライドすることもできます。プログラム・ビルド環境を消去してから新たにビルドを開始する場合は、古いオブジェクト・ファイルとともにレジストリー・ファイルも削除してください。

関連コンパイル単位の再コンパイル

2 つのコンパイル単位、A と B が同じインスタンス生成を参照する場合、

-qtemplateregistry コンパイラー・オプションを指定すると、次のような影響があります。

- A を最初にコンパイルすると、A のオブジェクト・ファイルにインスタンス生成のコードが含まれます。
- 次に B をコンパイルすると、B のオブジェクト・ファイルにはインスタンス生成のコードは含まれません。オブジェクト A に既に含まれているためです。
- あとで、このインスタンス生成を参照しないように A を変更すると、オブジェクト B の参照に、未解決のシンボル・エラーが発生します。A を再コンパイルすると、コンパイラーはこの問題を検出して、次のように処理します。
 - **-qtemplatererecompile** コンパイラー・オプションが有効であれば、コンパイラーはリンク・ステップ時に自動的に B を再コンパイルして、A で指定したのと同じコンパイラー・オプションを使用します。(ただし、個別のコンパイ

ル・ステップとリンク・ステップを使用する場合は、リンク・ステップにコンパイル・オプションを組み込んで、B の正しいコンパイルを確認する必要があります。)

- **-qnotemplaterecompile** コンパイラー・オプションが有効であれば、コンパイラーが警告を出すので、B を手動で再コンパイルしてください。

-qtempinc から -qtemplateregistry への切り替え

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

- アプリケーションが **-qtempinc** でも **-qnotempinc** でも正常にコンパイルされる場合は、変更する必要はありません。
- アプリケーションが **-qtempinc** では正常にコンパイルされるが **-qnotempinc** ではコンパイルされない場合は、**-qnotempinc** でも正常にコンパイルされるように、変更する必要があります。個々のテンプレート宣言ファイルに、
__TEMPINC__ マクロが定義されていない場合は、対応するテンプレート・インプリメンテーション・ファイルを組み込んでください。44 ページの『**-qtempinc** の例』の図を参照してください。

関連資料

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

第 6 章 スレッド・セーフティの確保 (C++)

マルチスレッドの C++ アプリケーションをビルドしている場合、C++ 標準テンプレート・ライブラリーおよびストリーム・クラスで定義されるオブジェクトを使用する際に考慮する必要のあるスレッド・セーフティ問題が幾つかあります。

テンプレート・オブジェクトのスレッド・セーフティの確保

標準 テンプレート・ライブラリーの以下のヘッダーは再入可能です。

- **algorithm**
- **deque**
- **functional**
- **iterator**
- **list**
- **map**
- **memory**
- **numeric**
- **queue**
- **set**
- **stack**
- **string**
- **unordered_map**
- **unordered_set**
- **utility**
- **valarray**
- **vector**

XL C/C++ では、複数のスレッドから同時に単一のオブジェクトを安全に読み取ることができる限り、再入可能性がサポートされます。この再入可能性は、組み込み済みのレベルです。ロックや他のグローバル割り振りリソースを使用することはできません。

ただし、以下の場合はヘッダーを再入することはできません。

- 複数のスレッドによって同時に単一のコンテナ・オブジェクトが書き込まれる。
- 単一のコンテナ・オブジェクトが、1 つのスレッドで書き込まれ、1 つ以上の他のスレッドで読み込まれる。

複数のスレッドが単一のコンテナに書き込みを行う場合、または単一のスレッドが単一のコンテナに書き込みを行う際に他のスレッドがそのコンテナから読み取りを行う場合は、ユーザーの責任においてこのコンテナに対するアクセスを直列化してください。複数のスレッドが単一のコンテナから読み取りを行い、そのコンテナに対する書き込みが行われない場合は、直列化を行う必要はありません。

ストリーム・オブジェクトのスレッド・セーフティーの確保

iostream 標準ライブラリーで宣言されるすべてのクラスは再入可能で、単一ロックを使用して、デッドロックが起こらないようにしながらスレッド・セーフティーを確保します。しかし、マルチプロセッサ・マシンでは、2 つの異なるスレッドが同時に共有ストリーム・オブジェクトにアクセスしようとするか、または (例えば、キーボードからの) 入力を待つ間にストリーム・オブジェクトがロックを保持すると、まれにライブ・ロックが起こることがあります。ライブ・ロックが起こるのを回避したい場合は、コンパイル時に以下のマクロを使用して、入力ストリーム・オブジェクト、出力ストリーム・オブジェクト、またはその両方でロックを使用不可にすることができます。

__NOLOCK_ON_INPUT

入力ロックを使用不可にします。

__NOLOCK_ON_OUTPUT

出力ロックを使用不可にします。

これらのマクロの 1 つまたは両方を使用するには、コンパイル・コマンド行で **-D** オプションを使用して、マクロ名にプレフィックスを付けます。次に例を示します。

```
x1C_r -D__NOLOCK_ON_INPUT -D__NOLOCK_ON_OUTPUT a.C
```

ただし、入力オブジェクトまたは出力オブジェクトでロックを使用不可にしている場合、お客様の責任で、ストリーム・オブジェクトがスレッド間で共有されているときに、ソース・コードに適切なロック・メカニズムを提供してください。そうしない場合は、振る舞いが未定義になり、データが破壊されたり、アプリケーションが破損する可能性があります。

注: ロック使用不可化マクロとともに、OpenMP ディレクティブまたは **-qsmp** オプションを使用して、入出力ストリーム・オブジェクトを共有するコードを自動的に並列化すると、Pthreads または他のマルチスレッド構造をインプリメントするコードと同じ危険を冒すことになるため、スレッドを同期化する必要があります。

関連資料

- ・「コンパイラー・リファレンス」の **-D**
- ・「コンパイラー・リファレンス」の **-qsmp**

第 7 章 ライブラリーの構成

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

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

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

58 ページの『共用ライブラリーの動的なロード』では、実行時に C++ 共用ライブラリーのロード、初期化、アンロード、および終了を行うために、アプリケーション・コードで利用できる 2 つの機能について説明します。

ライブラリーのコンパイルとリンク

静的ライブラリーのコンパイル

静的 (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。
2. AIX **ar** コマンドを実行して、生成されたオブジェクト・ファイルを、アーカイブ・ライブラリー・ファイルに追加する。

たとえば次のようになります。

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

共用ライブラリーのコンパイル

静的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. 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。たとえば次のようになります。

```
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。
 - 53 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・ステップで使ったエクスポート・ファイルの基本を作成する。たとえば次のようになります。

```
xlc -qmkshrojb -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、**-qmkshrojb** 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。(C++ オブジェクト・ファイルから共用ライブラリーを作成している場合は、54 ページの『オブジェクトへの優先順位の割り当て』で説明されているように、初期化の優先順位を共用ライブラリーに割り当てることもできます。)たとえば次のようになります。

```
xlc -qmkshrojb 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。)

別の方法として、共用ライブラリーを C++ オブジェクト・ファイルから作成している場合、66 ページの『makeC++SharedLib ユーティリティーによる共用ライブラリーの作成』で説明されているように、**makeC++SharedLib** ユーティリティーを使用することができます。ただし、**-qmkshrojb** メソッドは C++ テンプレート・インスタンス化を自動的に処理する機能、および **-O5** 最適化オプションとの互換性などの利点が幾つかあるため、優先されます。

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。たとえば次のようになります。

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

5. 53 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、次のようになります。

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。(C++ 共用オブジェクトの優先順位を指定する場合、**-qmkshrojb** オプションを使用することもできます。54 ページの『ライブラリー内の静的オブジェクトの初期化 (C++)』を参照してください。) たとえば次のようになります。

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

3. 53 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。

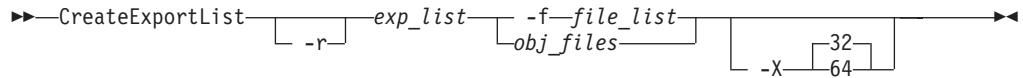


共用ライブラリーを動的にロードするときに、システムで静的な初期化を実行する場合は、58 ページの『共用ライブラリーの動的なロード』で説明されているロード関数とアンロード関数を使用します。

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** オブジェクトの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。
- 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。

ライブラリーとアプリケーションとのリンク

静的ライブラリーまたは共用ライブラリーをメインプログラムにリンクするには、同じコマンド・ストリングを使用することができます。たとえば次のようになります。

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

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

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

共用ライブラリー間のリンク

モジュールをアプリケーションにリンクするのと同様、共用ライブラリー同士をリンクすれば、その間に依存関係を作成することができます。たとえば次のようになります。

```
xlc -qmkshrojb [-G] -o mylib.so myfile.o -ldirectory -lfoo
```

関連資料

- ・「コンパイラー・リファレンス」の **-qmkshrojb**
- ・「コンパイラー・リファレンス」の **-l**
- ・「コンパイラー・リファレンス」の **-L**
- ・「AIX コマンド解説書」の **ar**
- ・「AIX コマンド解説書」の **ld**
- ・「コンパイラー・リファレンス」の **-G**
- ・「コンパイラー・リファレンス」の **-brtl**
- ・「コンパイラー・リファレンス」の **-qexpfile**

ライブラリー内の静的オブジェクトの初期化 (C++)

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

オブジェクトの初期化順序を指定するには、オブジェクトに相対的な優先順位 番号を割り当てます。ファイル全体や、ファイル内のオブジェクトの優先順位を指定できるメカニズムについては、『オブジェクトへの優先順位の割り当て』で説明します。複数のモジュール間でオブジェクトの初期化順序を制御できるメカニズムについては、57 ページの『ライブラリー間のオブジェクト初期化の順序』で説明します。

オブジェクトへの優先順位の割り当て

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

 AIX

 Linux

ファイル全体に優先順位を設定する

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

▶ AIX ▶ Linux ▶ Mac OS X ファイル内のオブジェクトに優先順位を設定する

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

▶ Linux ▶ Mac OS X 個々のオブジェクト別に優先順位を設定する

この方法を使用するには、ソース・ファイルで、**init_priority** 変数属性を使用します。**init_priority** 属性は、**#pragma priority** ディレクティブより優先され、任意の宣言順序でオブジェクトに適用できます。**Linux** では、オブジェクトは優先順位に従って初期化され、いくつかのコンパイル単位にわたって、その逆の順序で終了します。**Mac OS X** では、オブジェクトは優先順位に従って初期化され、1つのコンパイル単位内でのみ、その逆の順序で終了します。

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

優先順位番号の使用

▶ AIX 優先順位番号の範囲は、-2147483643 から 2147483647 までです。ただし、-2147483648 ～ -2147482624 までの番号はシステム用になっています。指定できる最小の優先順位番号は -2147482623 で、この番号のものが最初に初期化されません。最大の優先順位番号は 2147483647 で、この番号のものが最後に初期化されません。優先順位が指定されていない場合、デフォルトの優先順位は 0（ゼロ）になります。

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

以下の例は、単一ファイル内のオブジェクト、および 2 つのファイル間のオブジェクトの優先順位を指定する方法を示したものです。57 ページの『ライブラリー間のオブジェクト初期化の順序』には、AIX プラットフォームでのオブジェクトの初期化順序に関する詳細情報が記載されています。

ファイル内のオブジェクトの初期化の例

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

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

static Base a ;

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

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

Garage d ;
...
```

複数ファイル間のオブジェクト初期化の例

次の例は、farm.C と zoo.C の 2 つのファイル内のオブジェクトの初期化の順序を記述したものです。2 つのファイルは、ともに **#pragma priority** ディレクティブを使用し、**-qpriority** オプションでコンパイルされています。

farm.C -qpriority=2000

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

zoo.C -qpriority=2000

```
...
Lion k ;
#pragma priority(4000)
Bear m ;
...
#pragma priority(5000)
...
Zebra n ;
Snake s ;
...
#pragma priority(8000)
Frog f ;
...
```

実行時には、これらのファイル内のオブジェクトは、次の順序で初期化されます。

シーケンス	オブジェクト	優先順位の値	コメント
1	Lion k	2000	ファイル zoo.o の優先順位番号 (2000) を使用 (最初に初期化)。
2	Dog a	3000	pragma の優先順位 (3000) を使用。
3	Dog b	3000	Dog a と同じ。
4	Bear m	4000	pragma で指定された、次の優先順位番号 (4000)。
5	Zebra n	5000	pragma で指定された、次の優先順位番号 (5000)。

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

関連資料

- 「コンパイラー・リファレンス」の **-qmkshrobj**
- 「コンパイラー・リファレンス」の **-qpriority**
- 「コンパイラー・リファレンス」の **#pragma priority**

ライブラリー間のオブジェクト初期化の順序

実行時にアプリケーションのすべてのモジュールがロードされると、モジュールは、その優先順位に従って初期化されます (**main** 関数を含む実行可能プログラムには、常に優先順位 0 が割り当てられます)。ライブラリー内でオブジェクトが初期化される場合は、初期化の順序は、54 ページの『オブジェクトへの優先順位の割り当て』で説明されている規則に従います。オブジェクトに優先順位が割り当てられていない場合、あるいは同じ優先順位が割り当てられている場合は、オブジェクト・ファイルはランダムに初期化され、そのファイル内のオブジェクトは宣言の順序に従って初期化されます。オブジェクトは、その構成とは逆の順序で終了されます。

複数ライブラリー間のオブジェクト初期化の例

この例では、以下のモジュールが使用されています。

- **main.out** は、**main** 関数に含まれる実行可能モジュールです。
- **libS1** と **libS2** は、どちらも共用ライブラリーです。
- **libS3** と **libS4** は、どちらも共用ライブラリーで、**libS1** と依存関係にあります。
- **libS5** と **libS6** は、どちらも共用ライブラリーで、**libS2** と依存関係にあります。

従属ライブラリーは、次のコマンド・ストリングによって作成されます。

```
x1C -qmkshrobj=50 -o libS3 fileE.o fileF.o
x1C -qmkshrobj=-600 -o libS4 fileG.o fileH.o
x1C -qmkshrobj=-200 -o libS4 fileI.o fileJ.o
x1C -qmkshrobj=-150 -o libS6 fileK.o fileL.o
```

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

```
x1C -qmkshrobj=-300 main.c -o main.out -L. -lA
x1C -qmkshrobj=100 main.c -o main.out -L. -lB
```

次の図は、共用ライブラリー内のオブジェクトの初期化順序を示したものです。

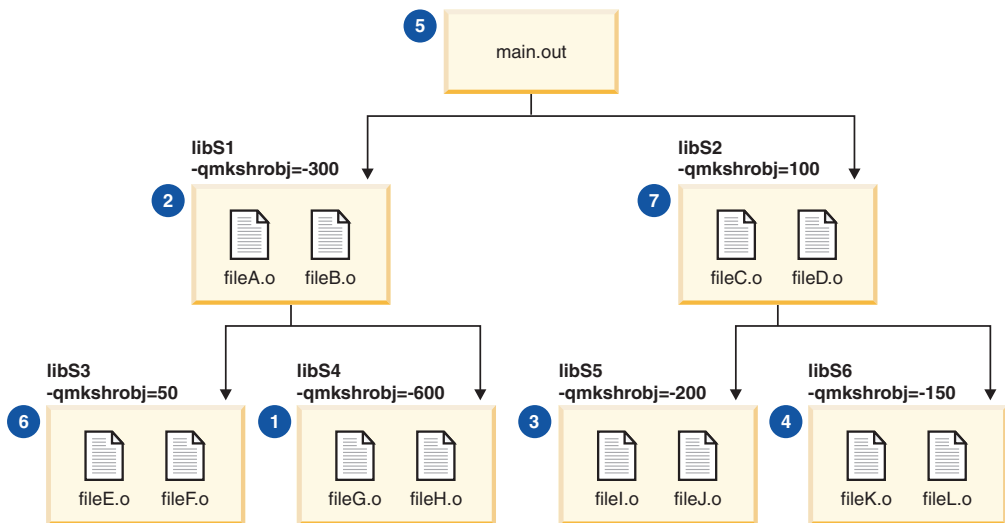


図 1. AIX 上のオブジェクトの初期化順序

オブジェクトは、次のように初期化されます。

シーケンス	オブジェクト	優先順位の値	コメント
1	libS4	-600	最初に初期化 (優先順位番号が最も低い)。
2	libS1	-300	次に初期化 (次の優先順位番号)。
3	libS5	-200	次に初期化 (次の優先順位番号)。
4	libS6	-150	次に初期化 (次の優先順位番号)。
5	main.out	0	次に初期化 (次の優先順位番号)。メインプログラムの優先順位は、常に 0 です。
6	libS3	50	次に初期化 (次の優先順位番号)。
7	libS2	100	最後に初期化 (次の優先順位番号)。
8	すべてのライブラリーのすべてのオブジェクトは、その優先順位番号に従って初期化されます。		

関連資料

- 「コンパイラー・リファレンス」の **-qmkshrobj**

共用ライブラリーの動的なロード

共用ライブラリー内に含まれる C++ オブジェクトのロードおよび初期化をプログラマチックに制御する場合、XL C/C++ によって提供されている 2 つの関数 **loadAndInit** および **terminateAndUnload** を使用することができます。これらの関数はヘッダー・ファイル **load.h** で宣言され、メイン・プログラムから呼び出して、任意の名前付き共用ライブラリーのロード、初期化、終了、およびアンロードを実行することができます。これらの関数は AIX **load** および **unload** ルーチンと同じ方法で機能しますが、さらに C++ オブジェクトの初期化を実行します。

注: 移植性を確保するために、初期化と終了処理を実行して、**loadAndInit** および **terminateAndUnload** と正しく対話する、POSIX **dlopen** および **dlclose** 関

数を使用することができます。**dlopen** および **dlclose** について詳しくは、「AIX テクニカル・リファレンス: 基本オペレーティング・システムおよび拡張機能」を参照してください。

loadAndInit 関数を使用したモジュールのロードおよび初期化

loadAndInit は、**load** ルーチンと同じパラメーターを受け取り、同じ値およびエラー・コードを返します。詳しくは、「AIX テクニカル・リファレンス: 基本オペレーティング・システムおよび拡張機能」にある **load** ルーチンを参照してください。

形式

```
#include <load.h>
int (*loadAndInit(char *FilePath, unsigned int Flags, char *LibraryPath))();
```

説明

loadAndInit 関数は、AIX **load** ルーチンを呼び出し、指定したモジュール (共用ライブラリー) を呼び出し側のプロセスのアドレス・スペースにロードします。共用ライブラリーが正常にロードされたなら、C++ の初期化がある場合は行われます。**loadAndInit** 関数により、ライブラリーをロードするために **dlopen** が使用される場合でも、共用ライブラリーは一度だけ初期化されます。以降に同じ共用ライブラリーをロードしても、共用ライブラリーの初期化は行われません。

共用ライブラリーをロードしたことにより、他の共用ライブラリーがロードされた場合は、それらの共用ライブラリーの初期化も実行されます (すでに実行されていない場合)。共用ライブラリーをロードしたことにより、複数の共用ライブラリーが初期化される場合は、初期化の順序は、共用ライブラリーが作成されたときに割り当てられた優先順位によって決定されます。同じ優先順位の共用ライブラリーは、ランダムな順序に初期化されます。

共用ライブラリーを終了してアンロードするには、下記で説明する、**terminateAndUnload** 関数を使用します。

C++ 初期化では、解決するのに AIX **loadbind** ルーチンを呼び出す必要のあるシンボルは、参照しないでください。これは、**loadbind** ルーチンは通常、**loadAndInit** 関数が戻された後に呼び出されるためです。

パラメーター

FilePath

ロードする共用ライブラリーの名前、またはアーカイブのメンバーを指定します。相対または絶対パス名 (つまり、1 つ以上の / 文字を含む名前) を指定する場合、ファイルは直接使用され、*LibraryPath* で指定したディレクトリーの検索は行われません。ベース名 (つまり、/ 文字が含まれない名前) を指定する場合、*LibraryPath* パラメーターで指定したディレクトリーの検索は実行されます (下記を参照)。

フラグ **loadAndInit** の動作を変更します。特殊な動作が必要でない場合は、値を 0 (または 1) に設定してください。設定可能なフラグは、以下のものです。

L_LIBPATH_EXEC

loadAndInit 呼び出し中で指定されるすべてのライブラリー・パス

の前に、プログラム実行時に使用されるライブラリー・パスを付加することを指定します。このフラグを使用することが推奨されます。

L_NOAUTODEFER

延期されたインポートは **loadbind** ルーチンを使用して明示的に解決する必要があることを指定します。

L_LOADMEMBER

FilePath はアーカイブに含まれるメンバーの名前であることを指定します。形式は、*archivename.a(member)* です。

LibraryPath

デフォルトのライブラリー検索パスを指します。

戻り値

正常終了すると、**loadAndInit** 関数は、共用ライブラリーの入り口点 (またはデータ・セクション) の関数へのポインターを戻します。

loadAndInit 関数が失敗すると、NULL ポインターが戻され、モジュールはロードおよび初期化されず、エラーを示すグローバル変数 **errno** が設定されます。

terminateAndUnload 関数を使用したモジュールの終了およびアンロード

terminateAndUnload 関数は、**unload** ルーチンと同じパラメーターを受け取り、同じ値およびエラー・コードを返します。詳しくは、「AIX テクニカル・リファレンス: 基本オペレーティング・システムおよび拡張機能」にある **unload** ルーチンを参照してください。

形式

```
#include <load.h>
int terminateAndUnload(int (*FunctionPointer)());
```

説明

terminateAndUnload 関数は、必要な C++ の終了処理を行い、モジュール (共用ライブラリー) をアンロードします。**loadAndInit** ルーチンによって戻された関数ポインターは、**terminateAndUnload** 関数のパラメーターとして使用されます。この共用ライブラリーが最後にアンロードされるとき、この共用ライブラリーと、同じく最後にアンロードされる他の共用ライブラリー (存在する場合) に対し、C++ の終了処理が行われます。**terminateAndUnload** 関数により、ライブラリーをアンロードするために **dlclose** が使用される場合でも、共用ライブラリーは一度だけ終了します。終了の順序は、**loadAndInit** 関数による初期化の順序と逆の順序です。C++ の終了処理中にキャッチされていない例外が発生した場合は、終了処理は停止され、共用ライブラリーはアンロードされます。

ある共用ライブラリーに対して、**loadAndInit** 関数が **terminateAndUnload** より多く呼び出されると、共用ライブラリーに対して C++ の終了処理が実行されることはありません。**terminateAndUnload** 関数の呼び出し時に C++ 終了処理が行われることを前提としているならば、**terminateAndUnload** 関数を呼び出す回数が **loadAndInit** 関数を呼び出す数と一致していることを確認してください。プログラ

ムが終了するときに **loadAndInit** 関数でロードされた共用ライブラリーがまだ使用中である場合、C++ 終了処理が実行されます。

loadAndInit 関数でロードしなかった共用ライブラリーは、**terminateAndUnload** 関数でアンロードしても、終了処理が行われません。

パラメーター

FunctionPointer

loadAndInit 関数が返す関数名を指定します。

戻り値

terminateAndUnload 関数が正常に完了すると、値 0 が戻されます。共用ライブラリーが使用中であるためにアンロードされず、C++ 終了処理が実行されなかった場合でもこの値が戻されます。

terminateAndUnload 関数が失敗すると、値 -1 が戻され、エラーを表すために **errno** を設定します。

第 8 章 C++ ユーティリティの使用

XL C/C++ Enterprise Edition for AIX は、C++ アプリケーションの管理に使用できる追加ユーティリティ・セットと同梱されています。

- オブジェクト・ファイル内のコンパイル済みシンボル名をデマングリングするためのフィルター。『コンパイル済み C++ 名の c++filt によるデマングリング』に説明。
- マングルされた名前をデマングリングおよび取り扱いするためのクラスのライブラリー。65 ページの『コンパイル済み C++ 名のデマングル・クラス・ライブラリーによるデマングリング』に説明。
- ライブラリー・ファイルから共用ライブラリーを作成するための分配可能なシェルスクリプト。66 ページの『makeC++SharedLib ユーティリティによる共用ライブラリーの作成』に説明。
- C++ オブジェクト・ファイルおよびアーカイブをリンクするための分配可能なシェルスクリプト。68 ページの『linkx1C ユーティリティとのリンク』に説明。

コンパイル済み C++ 名のデマングリング

XL C/C++ が C++ プログラムをコンパイルするとき、すべての関数名および他の ID を、タイプ情報およびスコープ情報を含めてエンコード (マングル) します。このネーム・マングリングは、多重定義された C++ 関数および演算子を調整するために必要です。リンカーは、マングルされた名前を使用することで、重複シンボルを解決し、タイプ・セーフなリンケージを確保します。これらのマングルされた名前は、オブジェクト・ファイルおよび最終的な実行可能ファイル中で使用されています。

ファイルを操作できるツール (例えば、AIX **dump** ユーティリティ) は、マングルされた名前のみを持ち、元のソース・コードの名前を保持しません。これらのツールの出力には、マングルされた名前が使用されます。名前が認識可能ではなくなっているため、この出力はお勧めできません。

マングルされた名前を、元のソース・コードでの名前に変換するユーティリティが 2 つあります。

c++filt マングルされた名前をデマングル (デコード) するフィルターです。

demangle.h マングルされた名前の操作ツールを開発するための、クラス・ライブラリーです。

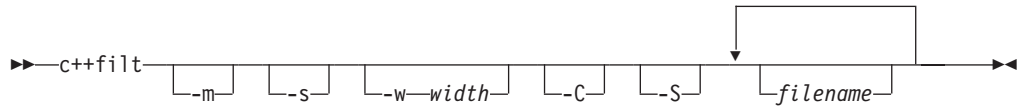
両方とも次のセクションで説明されています。

コンパイル済み C++ 名の c++filt によるデマングリング

c++filt ユーティリティはフィルターで、ファイル名または標準入力から、標準出力へ文字をコピーし、その際にすべてのマングルされた名前を、対応するデマング

ルされた名前に置き換えます。ファイル名引き数とそのフィルターを直接使用でき、フィルターはファイル内のすべてのマングルされた名前のデマングルされた名前を出力します。または、特定のマングルされた名前などのテキストを入力するシェル・コマンドを使用してそれをフィルターにパイプ接続すると、フィルターは指定された名前のデマングルされた名前を提供します。

c++filt ユーティリティ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。

- m** 左の列にデマングルされた名前と右の列にそれに対応するマングルされた名前を横並びにしたリストを含む、シンボル・マップを作成します。
- s** それぞれのデマングルされた名前のすぐ後にそのマングルされた名前が続くリストを作成します。
- w width** デマングルされた名前を、幅が *width* 文字分のフィールドでプリントします。名前が *width* より短い場合は、右側が空白で埋められ、より長い場合は、*width* の幅になるように切り捨てられます。
- C** Q2_1X1Y のようなスタンドアロン・クラス名をデマングルします。
- S** __vft1X (仮想関数を示す) などの特別なコンパイラ生成シンボル名をデマングルします。
- filename* これは、デマングルする、マングルされた名前を含むファイルの名前です。複数のファイル名を指定することが可能です。

例えば、次のコマンドを実行すると、オブジェクト・ファイル `functions.o` に含まれるシンボルが表示され、マングルされた名前とデマングルされた名前を横並びにしたリストが、幅 40 文字のフィールドに出力されます。

```
c++filt -m -w 40 functions.o
```

出力は次のように表示されます。

C++ Symbol Mapping	
demangled:	Mangled:
Average::insertValue(double)	insertValue__7AverageFd
Average::getCount()	getCount__7AverageFv
Average::getTotal()	getTotal__7AverageFv
Average::getAverage()	getAverage__7AverageFv

次のコマンドはデマングルされた名前のすぐ後にマングルされた名前を表示します。

```
echo getAverage__7AverageFv | c++filt -s
```

出力は次のように表示されます。

```
Average::getAverage()getAverage__7AverageFv
```

コンパイル済み C++ 名のデマングル・クラス・ライブラリーによるデマングリング

demangle クラス・ライブラリーには、クライアント・プログラムが名前をデマングルし、その結果得られる名前を部分ごとに調べることができる、小規模のクラス階層が含まれます。このクラス・ライブラリーは、C プログラムで使用するための C 言語インターフェースも提供します。このライブラリーは C++ ライブラリーですが、外部の C++ 機能は使用していないので、C プログラムに直接リンクすることができます。 **demangle** ライブラリーは、 **libC.a** の一部として組み込まれているので、 **libC.a** がリンクされていれば、必要に応じて自動的にリンクされます。

ヘッダー・ファイルは基本クラス **Name** およびメンバー関数 **Demangle** を宣言し、マングルされた名前をパラメーターとして取り、それに対応するデマングルされた名前を戻します。ヘッダー・ファイルは 4 つの追加サブクラスを宣言し、名前に関する追加情報を取得できるようそれぞれにメンバー関数が含まれます。これらのクラスは以下のとおりです。

ClassName

独立クラスまたはネスト・クラスの名前を照会するために使用できます。

FunctionName

関数の名前を照会するために使用できます。

MemberVarName

メンバー変数の名前を照会するために使用できます。

MemberFunctionName

メンバー関数の名前を照会するために使用できます。

これらの各クラスでは、名前に関する情報を取得できるよう関数が定義されています。例えば、関数名のために、以下の情報を戻す関数セットが定義されています。

種類 照会される名前のタイプを戻します (つまり、クラス、関数、メンバー変数、またはメンバー関数)。

本文 関数の完全修飾のオリジナル・テキストを戻します。

Rootname

関数の非修飾の元の名前を戻します。

引き数 パラメーター・リストのオリジナル・テキストを戻します。

スコープ

関数の修飾子のオリジナル・テキストを戻します。

IsConst/IsVolatile/IsStatic

これらの型修飾子またはストレージ・クラス指定子に関する真/偽を戻します。

名前 (文字配列として表現されています) をデマングルするには、 **Name** クラスの動的インスタンスを作成し、その際に文字ストリングをクラスのコンストラクターに受け渡します。例えば、コンパイラーが `X::f(int)` をマングルし、マングル名 `f__1XFi` を作成した場合、この名前をデマングルするには次のコードを使用します。

```
char *rest;  
Name *name = Demangle("f__1XFi", rest) ;
```

元の名前がマングルされていなかったため、提供された文字ストリングがデマングルリングを必要とする名前ではない場合、 **Demangle** 関数は **NULL** を返します。

プログラムが **Name** クラスのインスタンスを作成すると、プログラムは、**Kind** メソッドを使用して、インスタンスを照会し、それがどの種類であるかを調べることができます。マングルされた名前 **f_1XFi** の例を使用すると、以下のコードのようになります。

```
name->Kind()
```

MemberFunction を戻します。

返された名前の種類に基づき、プログラムは名前の各部分のテキストを求めたり、名前全体のテキストを求めることができます。次の表は、マングルされた名前 **f_1XFi** を前提とした例を示しています。

戻ります...	...このコードを使用します。	結果
関数の修飾子の名前	<code>((MemberFunctionName *)name)->Scope()->Text()</code>	X
関数の非修飾名	<code>((MemberFunctionName *)name)->RootName()</code>	f
関数の完全修飾名	<code>((MemberFunctionName *)name)->Text()</code>	X::f(int)

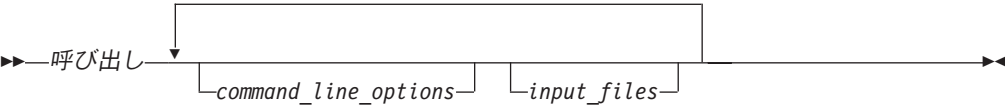
demangle ライブラリーおよび **C++** インターフェースについて詳しくは、そのライブラリーのヘッダー・ファイル **/usr/vacpp/include/demangle.h** のコメントを参照してください。

makeC++SharedLib ユーティリティーによる共用ライブラリーの作成

makeC++SharedLib は、**C++** の **.o** および **.a** ファイルをリンクするシェル・スクリプトです。このスクリプトは再配布可能で、**XL C/C++** をインストールしていないユーザーでも使用することができます。

makeC++SharedLib コマンドではなく、**-qmkshrobj** コンパイラー・オプションを使用することをお勧めします。このオプションを使用することによる利点には、リンク・タイム **C++** テンプレート・インスタンスが自動的に処理されるという点(テンプレート・インクルード・ディレクトリーまたはテンプレート・レジストリーのいずれかを使用)、および **-O5** オプションとの互換性があります。

makeC++SharedLib の構文は以下のとおりです。



- 呼び出し コマンド。前にパスが付きます。以下のコマンドが提供されています。
- **makeC++SharedLib**
 - **makeC++SharedLib_r**
 - **makeC++SharedLib_r7**
 - **makeC++SharedLib128**

コマンド行オプション

- o *shared_file.o*** 共用ファイル情報を保持するファイルの名前です。デフォルトは、**shr.o** です。
- b** **ld** コマンドの **-b** バインダー・オプションを使用します。
- L*lib_dir*** **ld** コマンドの **-L** オプションを使用して、未解決のシンボルを検索するディレクトリーのリストに、ディレクトリー *lib_dir* を追加します。
- l*library*** 未解決のシンボルを検索するライブラリーのリストに、*library* を追加します。
- p *priority*** ファイルの優先順位を指定します。*priority* は、-214782623 (最高の優先順位 - 最初に初期化されます) から 214783647 (最低の優先順位 - 最後に初期化されます) の間の任意の値に指定できます。
-214783648 から -214782624 の間の値は、システム使用のために予約されています。詳しくは、54 ページの『オブジェクトへの優先順位の割り当て』を参照してください。
- I *import_list*** **ld** コマンドの **-bl** オプションを使用して、バインダーによって解決できる、ファイル *import_list* にあるシンボルのリストを解決します。
- E *export_list*** **ld** コマンドの **-bE** オプションを使用して、*export_list* ファイルの外部シンボルをエクスポートします。 **-E *export_list*** を指定しないと、すべてのグローバル・シンボルを含むリストが生成されます。
- e *file*** **-E *export_list*** によって計算されたリストを、*file* に保管します。
- n *name*** 共有実行可能ファイルの入り口名を、*name* に設定します。これは、**ld -e *name*** コマンドを使用するのと同じことです。
- X *mode*** **makeC++SharedLib** が作成するオブジェクト・ファイルの種類を指定します。 *mode* は **32** (32 ビット・オブジェクト・ファイルのみを処理します) または **64** (64 ビット・オブジェクト・ファイルのみを処理します) のいずれかでなければなりません。デフォルトでは、32 ビット・オブジェクト・ファイルを処理します (64 ビット・オブジェクトは無視されます)。 **OBJECT_MODE** 環境変数でモードを設定することもできます。例えば、 **OBJECT_MODE=64** を指定すると、**makeC++SharedLib** は 64 ビット・オブジェクトを処理し、32 ビット・オブジェクトは無視します。 **-X** フラグは、**OBJECT_MODE** 変数に優先します。

入力ファイル

- file.o*** 共用ライブラリーに入れるオブジェクト・ファイルです。
- file.a*** 共用ライブラリーに入れるアーカイブ・ファイルです。

linkx1C ユーティリティーとのリンク

linkx1C は、C++ の **.o** および **.a** ファイルをリンクする、小規模のシェル・スクリプトです。このスクリプトは再配布可能で、XL C/C++ をインストールしていないユーザーでも使用することができます。

linkx1C は、x1C コンパイラー・オプションの次のサブセットをサポートします。

- **-q32** (32 ビット・アプリケーションを作成する)
- **-q64** (64 ビット・アプリケーションを作成する)
- **-b** (リンカー・オプションを **ld** に渡す)
- **-f** (オブジェクト・ファイルのリストを **ld** に渡す)
- **-l** (ライブラリーを **ld** に渡す)
- **-L** (ライブラリー・パスを **ld** に渡す)
- **-o** (出力ファイルを指定する)
- **-s** (出力をストリップする)
- **-qtwolink** (2 段階のリンクを使用可能にする)

linkx1C は、以下のコンパイラー・オプションをサポートしません。

- **-G**
- **-p**
- **-pg**

linkx1C は、他のすべてのコンパイラー・オプションを受け入れ、無視します。

x1C とは異なり、**linkx1C** はランタイム・ライブラリーを指定しません。ユーザー自身がこれらのライブラリーを指定する必要があります。例えば、x1C **a.o** は、次のようになります。

```
linkx1C a.o -L/usr/lpp/vacpp/lib -lc -lm -lc
```

関連資料

- 「AIX コマンド解説書」の **ld**

第 9 章 アプリケーションの最適化

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

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

- 特定タイプの最適化を実行するオプションには、次のようなものがあります。
 - システム・アーキテクチャー。アプリケーションが特定のハードウェア構成上で実行される場合、コンパイラーは、マイクロプロセッサ・アーキテクチャー、キャッシュまたはメモリー形状、アドレッシング・モデルも含めたターゲット・マシン用に最適化される命令を生成することができます。これらのオプションについては、73 ページの『システム・アーキテクチャーの最適化』で説明します。
 - 共用メモリーの並列処理。アプリケーションが、共用メモリーの並列化をサポートするハードウェア上で実行される場合は、コンパイラーに、スレッド化されたコードの自動生成、あるいは OpenMP 標準プログラミング構文の認識を命令することができます。プログラム並列化のオプションについては、76 ページの『共用メモリーの並列処理の使用』で説明します。
 - 高位のループ分析および変換。コンパイラーは、さまざまな手法を駆使してループを最適化します。これらのオプションについては、75 ページの『高位ループ分析および変換の使用』で説明します。
 - プロシージャー間分析 (IPA)。コンパイラーは、コード・セクションを再編成して関数間の呼び出しを最適化します。IPA オプションについては、78 ページの『プロシージャー間分析の使用』で説明します。
 - プロファイル指示フィードバック (PDF)。コンパイラーは、呼び出し数とブロック数、および実行時間を基にして、コードのセクションを最適化することができます。PDF オプションについては、79 ページの『プロファイル指示フィードバックの使用』で説明します。
 - その他のタイプの最適化。ループのアンロール、関数のインライン化、スタック・ストレージの圧縮など、多数。これらのオプションについては、82 ページの『その他の最適化オプション』に簡単な説明があります。
- 最適化レベルを使用できます。これは、いくつかの手法をバンドルしたもので、この中に、前述の特定の最適化オプションを 1 つ以上組み込むことができます。4 つの最適化レベルがあり、順に、コードに対してより積極的な最適化を実行するようになっています。最適化レベルについては、70 ページの『最適化レベルの使用』で説明します。

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

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

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

最適化で利用できるオプションをすべて網羅し、カテゴリー別に編成したリストが、83 ページの『最適化およびパフォーマンスに関するオプションの要約』に記載されています。

最後に、アプリケーションを手動で調整してコンパイラーが使用する最適化手法を補う場合は、85 ページの『第 10 章 パフォーマンスを向上させるためのアプリケーションのコーディング』に記載されているコーディングのパフォーマンスに対する提案と、その最良実例を参考にしてください。

関連資料

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

最適化レベルの使用

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

表 13. 最適化レベル

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

最適化レベル 2 で使用される手法

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

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

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

共通の式 $c + d$ は、最初の計算が行われた時点で保管され、次のステートメントで使用されて f の値を決定します。

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

最適化レベル 3 で使用される手法

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

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

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

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

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

最適化レベル 4 および 5 で使用される手法



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

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

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

最適化レベル 2 および 3 の最大活用

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

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

関連資料

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

システム・アーキテクチャーの最適化

コンパイラーには、指定したマイクロプロセッサ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・プロセッサや、指定したプロセッサ・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、あるいは特定のプロセッサをそれぞれ選択できるように、最適化することができます。次の表は、ターゲット・マシンの個

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

表 14. ターゲット・マシンのオプション

オプション	振る舞い
-q32	32 ビット (4/4/4) アドレッシング・モデル用のコードを生成します (32 ビット実行モード)。これがデフォルトの設定値です。
-q64	64 ビット (4/8/8) アドレッシング・モデル用のコードを生成します (64 ビット実行モード)。
-qarch	命令コードを生成するプロセッサ・アーキテクチャー・ファミリーを選択します。このオプションによって、PowerPC® アーキテクチャー向け命令のサブセットに対して生成される命令セットが制限されます。デフォルトは -qarch=com です。 -O4 または -O5 を使用すると、デフォルトが -qarch=auto に設定されます。
-qtune	指定したマイクロプロセッサ上で実行するように、最適化にバイアスをかけます。この際、ターゲットとして使用する命令セット・アーキテクチャーにはまったく影響は及びません。デフォルトは -qtune=pwr3 です。
-qcache	特定のキャッシュまたはメモリー形状を定義します。デフォルトは、 -qtune の設定によって決まります。

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、「コンパイラー・リファレンス」の、『アーキテクチャー固有の、32 ビットまたは 64 ビットのコンパイルでコンパイラー・オプションを指定する』、および『コンパイラー・モードおよびプロセッサのアーキテクチャーの有効な組み合わせ』を参照してください。

ターゲット・マシンのオプションの最大活用

-qarch オプションの使用

アプリケーションのコンパイルと実行を同じマシン上で行う場合は、**-qarch=auto** オプションを使用して、コンパイルするマシンの特定のアーキテクチャーを自動的に検出し、そのマシンのみ (またはそれと同等のプロセッサ・アーキテクチャーをサポートするシステム) を対象とした命令を利用するコードを生成することができます。そうでない場合は、**-qarch** を使用して、コードを十分に実行できる最小限のマシン・ファミリーを指定してください。

平方根演算を最適化するには、ライブラリー関数を呼び出すのではなくインライン・コードを生成する方法で、**sqrt** 機能をサポートするプロセッサ・ファミリーを指定し、さらに、**-qignerrno** オプション (またはそのプロセッサ・ファミリーを暗黙指定する最適化オプション) も指定する必要があります。

-qarch=ppc64grsq を使用してください。このオプションは、**ppc64grsq** プロセッサ・グループに属するすべてのプロセッサ (RS64 II、RS64 III、POWER3、POWER4、POWER5、PowerPC970) で正しいコードを生成します。

-qtune オプションの使用

-qarch を使用して特定のアーキテクチャーを指定すると、**-qtune** は、自動的に、そのアーキテクチャーで最高のパフォーマンスを出す命令シーケンスを生成するサブオプションを選択します。**-qarch** を使用してアーキテクチャーのグループを指

定する場合は、**-qtune=auto** でコンパイルすると、指定したグループ内のすべてのアーキテクチャーで実行されるコードが生成されますが、命令シーケンスは、コンパイルするマシンのアーキテクチャーで最高のパフォーマンスを出すようになっています。

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

-qcache オプションの使用

-qcache オプションを使用する前に、まず **-qlistopt** オプションを使用して現行設定のリストを生成し、それで問題ないかどうかを確認します。独自に **-qcache** サブオプションを使用することに決めた場合は、**-qhot** または **-qsmp** をそのサブオプションと併用します。サブオプションの完全セット、オプション構文、および使用のためのガイドラインについては、「コンパイラー・リファレンス」の **-qcache** を参照してください。

関連資料

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

高位ループ分析および変換の使用

高位変換は、交換、融合、アンロールなどの手法を用いて、特にループのパフォーマンスを向上させる最適化です。これらのループ最適化の目的は次のとおりです。

- キャッシュと変換検索バッファーを効果的に使用して、メモリー・アクセスのコストを削減する。
- ハードウェアによって提供されるデータの事前取り出し機能を有効に利用して、計算とメモリー・アクセスを並行させる。
- 相補的なリソース要件を持つ命令の使用を再配列および平衡化して、マイクロプロセッサ・リソースの使用率を改善する。

高位ループ分析および変換を使用可能にするには、**-qhot** オプションを使用します。次の表は、**-qhot** で使用できるサブオプションのリストです。

表 15. **-qhot** のサブオプション

サブオプション	振る舞い
vector	コンパイラーに、いくつかのループを変換して、組み込みライブラリーにある各種の三角関数や演算（逆数や平方根など）の、標準バージョンではなく最適化バージョンを使用するように指示します。最適化バージョンを使用すると、精度とパフォーマンスに関して、さまざまなトレードオフが発生します。このサブオプションは、 -qhot 、 -O4 、または -O5 を使用すると、デフォルトで使用可能になります。
novector	コンパイラーに、上記の組み込みライブラリー関数を使用する最適化を避けるように指示します。プログラム結果の精度を落としたいくない場合は、このサブオプションまたは -qstrict を使用してください。
arraypad	コンパイラーに、メリットがあると思われる配列を、必要なだけ埋め込むように指示します。

-qhot の最大活用

以下に、**-qhot** を使用する場合の提案事項を挙げます。

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

関連資料

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

共用メモリーの並列処理の使用

IBM pSeries™ のマシンの中には、共用メモリーの並列処理ができるものがあります。**-qsmp** でコンパイルすると、この機能の活用に必要なスレッド化されたコードを生成することができます。このオプションは、少なくとも **-O2** の最適化レベルを暗黙指定します。

次の表は、最もよく使用されるサブオプションのリストです。すべてのサブオプションの説明と構文は、「コンパイラー・リファレンス」に記載されています。

表 16. 一般に使用される **-qsmp** のサブオプション

サブオプション	振る舞い
auto	コンパイラーに、可能ならユーザー支援なしに自動で並列コードを生成するように指示します。 -qsmp のサブオプションを指定しない場合は、これがデフォルト設定となり、このデフォルト設定で、 opt サブオプションも暗黙指定されます。
omp	コンパイラーに、明示的並列処理を指定するための OpenMP 言語拡張に従うように指示します。 -qsmp=omp と -qsmp=auto は、現時点では互換性がありません。
opt	コンパイラーに、最適化と同時に並行処理も行うように指示します。最適化は、他の最適化オプションがない場合は、 -O2 -qhot に相当します。
<i>fine_tuning</i>	サブオプションの他の値は、スレッド・スケジューリング、ネストされた並列処理、ロックなどの制御に使用されます。

-qsmp の最大活用

以下に、**-qsmp** オプションを使用する場合の提案事項を挙げます。

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

関連資料

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

プロシージャ間分析の使用

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

IPA は、**-qipa** オプションを指定して使用可能にします。最も一般的に使用されるサブオプションとその効果を、次の表に示します。サブオプションおよび構文の完全セットについては、「コンパイラー・リファレンス」の **-qipa** で説明しています。

表 17. 一般に使用される **-qipa** のサブオプション

サブオプション	振る舞い
level=0	プログラム区画と簡単なプロシージャ間の最適化。その内容は次のとおりです。 <ul style="list-style-type: none">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のローカライズ。呼び出し関係によるプロシージャ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は、メモリー内の比較的近いところにまとめて配置されます。)一部の最適化、特にレジスターの割り振りの拡大。
level=1	インライン化およびグローバル・データ・マッピング。主な機能は次のとおりです。 <ul style="list-style-type: none">プロシージャのインライン化。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、プロシージャ間データ・フロー： <ul style="list-style-type: none">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間最適化。これは、値の番号付け、コードの伝搬および単純化、条件へのコードの移動またはループ外へのコードの移動、冗長の除去という形で行われます。プロシージャ間の定数伝搬、デッド・コードの除去、ポインター分析、および関数間のコードの移動。プロシージャの特殊化 (クローン作成)。
inline=variable	関数のインライン化が正確に制御できるようになります。
<i>fine_tuning</i>	-qipa には、ほかに、ライブラリー・コードの振る舞いを指定する機能、プログラムの区分化を調整する機能、ファイルからコマンドを読み取る機能などを提供する値があります。

-qipa の最大活用

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

- **-qipa** オプションは、アプリケーション全体、あるいはそのできるだけ多くの部分の、コンパイルおよびリンク・ステップで指定してください。ライブラリー、共用オブジェクト、および実行可能プログラムに対しても **-qipa** を使用できますが、`main` 関数およびエクスポート機能をコンパイルする場合は、必ず **-qipa** を使用してください。
- コンパイルとリンクを個別に行う場合は、高速コンパイルのコンパイル・ステップで、**-qipa=noobject** を使用します。
- `Make` ファイルで最適化オプションを指定する場合は、必ず、コンパイラー・ドライバ (**xlc**) を使用してリンクし、リンク・ステップですべてのコンパイラー・オプションを組み込むようにしてください。
- IPA で、従来のコンパイルよりかなり大きなオブジェクト・ファイルを生成したり、`/tmp` ディレクトリーに十分なスペース (少なくとも 200 MB) があることを確認したり、`TMPDIR` 環境変数を使用して十分なフリー・スペースのある別のディレクトリーを指定したりすることができます。
- リンク時間が長すぎる場合は、**level** サブオプションを変えてみてください。**-qipa=level=0** を指定したコンパイルは、追加リンク時間が短い場合に非常に有益です。
- インライン化された関数のレポートを生成するには、**-qlist** または **-qipa=list** を使用します。インライン化された関数が少なすぎる、あるいは多すぎる場合は、**-qipa=inline** または **-qipa=noinline** の使用を検討してください。特定関数のインライン化を制御するには、**-Q+** または **-Q-** を使用します。

関連資料

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

プロファイル指示フィードバックの使用

プロファイル指示フィードバック (PDF) を使用すると、アプリケーションのパフォーマンスを通常の使用例に合うように調整することができます。コンパイラーは、分岐の頻度やコード・ブロックの実行の頻度の分析に基づいて、アプリケーションを最適化します。このプロセスは、アプリケーション全体を 2 度コンパイルする必要があるため、他のデバッグやチューニングが終了してから、アプリケーションを実稼働させる前の最後の段階の一部として使用されるようになっています。

次の図は、PDF プロセスを表しています。

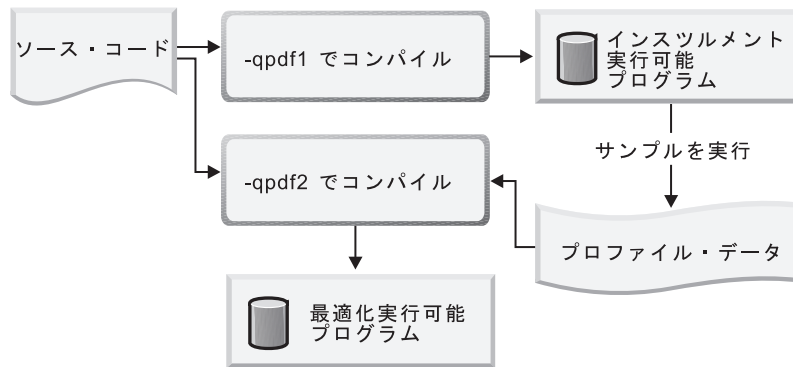


図2. プロファイル指示フィードバック

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

アプリケーションのすべてのコードを **-qpdf1** オプションでコンパイルしなくても、PDF プロセスの恩恵は受けられます。大規模アプリケーションでは、最適化の効果が最もよく現れるコード領域に集中することもできます。

-qpdf オプションを使用するには、次のようにします。

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

次のようにすれば、PDF ファイルをさらに制御することができます。

1. アプリケーションの一部またはすべてのソース・ファイルを、**-qpdf1** を指定してコンパイルする。
2. 標準的なデータ・セットを 1 つ以上使用して、アプリケーションを実行する。これによって、現行ディレクトリーに PDF ファイルが作成されます。
3. **PDFDIR** 環境変数で指定したディレクトリーを変更して、別のディレクトリーに PDF ファイルを作成する。
4. **-qpdf1** を指定してアプリケーションを再コンパイルする。
5. ステップ 3 と 4 を必要なだけ繰り返す。
6. **mergepdf** ユーティリティを使用して、PDF ファイルを連結する。例えば、時間の 53%、32%、15% にそれぞれ発生する使用パターンを表す 3 つの PDF ファイルを作成する場合は、次のコマンドが使用してください。

```
mergepdf -r 53 path1 -r 32 path2 -r 15 path3
```
7. **-qpdf2** を指定してアプリケーションをコンパイルする。

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

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

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

pdf および showpdf によるコンパイルの例

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

プログラム・ファイル `hello.c` のソースは次のとおりです。

```
#include <stdio.h>
void HelloWorld()
{
    printf("Hello World");
}
main()
{
    HelloWorld();
    return 0;
}
```

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

```
showpdf
```

結果は以下のようになります。

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

```
Call Counters:
5 | 1 printf(6)
```

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

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

```
Block coverage = 100% ( 2/2 )
```

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

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

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

Block Counters:

8-11 | 1

11 |

Block coverage = 100% (1/1)

Total Call coverage = 100% (2/2)

Total Block coverage = 100% (3/3)

関連資料

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

その他の最適化オプション

以下のオプションは、最適化の特定の局面を制御する際に使用できます。これらのオプションは、グループとして使用可能にされることもよくあり、また、もっと汎用的な最適化オプションまたはレベルを使用可能にすると、デフォルト値を与えられることもあります。詳しくは、「コンパイラー・リファレンス」に記載の各オプションの見出しを参照してください。

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


オプション	説明
-qcompact	コード・サイズの肥大につながる最適化 (ループのアンロール、関数のインライン化など) を抑制します。
-qignerrno	コンパイラーが、 errno はライブラリー関数呼び出しによって変更されないで、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	名前付き関数のインライン化を制御します。コンパイル時、リンク時、またはその両方で使用できます。 -qipa の使用時には、 -qinline と -qipa=inline は同義です。
-qunroll	ループのアンロールを独自に制御します。 -O3 では、暗黙のうちにアクティブになっています。
-qinlglue	リンカーによって生成され、外部関数の呼び出しまたは関数ポインターを介した呼び出しに使用される「グルー・コード」をインライン化するよう、コンパイラーに命令します。64 ビット・モード専用です。
-qtbtable	トレースバック・テーブル情報の生成を制御します。64 ビット・モード専用です。
 -qnoeh	C++ 例外がスローされないこと、クリーンアップ・コードが省略できることを、コンパイラーに通知します。プログラムが C++ 例外をスローしない場合は、このオプションを使用して、例外処理コードを除去し、プログラムを圧縮してください。

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

オプション	説明
-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windされないことを、コンパイラーに通知します。このオプションを選択すると、不揮発性レジスターの保管と復元の最適化を改善できます。C++ では、 -qnounwind オプションには -qnoeh オプションが暗黙指定されます。
-qnostrict	コンパイラーが、浮動小数点計算と、除外される可能性のある命令をリオーダーするのを停止します。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。
-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。

最適化およびパフォーマンスに関するオプションの要約

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

表 19. 最適化およびパフォーマンス調整に関するオプション

最適化フラグ	最適化制限オプション
-O/-qoptimize -qagrrcopy	-qkeepparm -qnoprefetch -qstrict -qcompact -qmaxmem
関数のインライン化	コード・サイズの削減
-Q -qinline -qinlglue	-bmaxdata -s -qnoeh
副次作用	ループの最適化
-qignerrno -qisolated_call	-qhot -qreport -qnostrict_induction -qunroll
プログラム全体の分析	プロセッサおよびアーキテクチャーの最適化
-qipa	-qarch -qcache -qtune -qdirectstorage

表 19. 最適化およびパフォーマンス調整に関するオプション (続き)

パフォーマンス・データ収集	その他の最適化オプション
-qfdpr -p -qpdf1 -qpdf2 -pg -qshowpdf	-qlargepage -qtocdata -qtocmerge -qsmallstack -qspill

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

69 ページの『第 9 章 アプリケーションの最適化』では、最小限のコーディングでコードを最適化するために XL C/C++ が提供する各種のコンパイラー・オプションについて説明します。アプリケーションをもう一歩進めて、コンパイラーの最適化を補完したり最大限に利用したりする場合は、以下の節で説明する C および C++ プログラミング手法を使用すれば、コードのパフォーマンスを向上させることができます。

- 『高速入出力手法の検索』
- 86 ページの『関数呼び出しによるオーバーヘッドの削減』
- 87 ページの『効率的なメモリーの管理』
- 88 ページの『変数の最適化』
- 89 ページの『効率的なストリングの操作』
- 89 ページの『式とプログラム・ロジックの最適化』
- 90 ページの『64 ビット・モードでの演算の最適化』





高速入出力手法の検索

プログラムの入出力のパフォーマンスを向上するには、いくつか方法があります。

- テキスト・ストリームの代わりにバイナリー・ストリームを使用する。バイナリー・ストリームでは、入力または出力時にデータは変更されません。
- **open** および **close** のような、低水準の入出力関数を使用します。これらの関数は、**fopen** および **fclose** のようなストリーム入出力関数と比べて、より高速で、よりアプリケーションに特定です。低レベルの関数に対してユーザー独自のバッファリングを提供しなければなりません。
- ユーザー独自の入出力バッファリングを行う場合、バッファをページのサイズである 4K の倍数にする。
- 入力を読み取るときは、一度に 1 つの文字ではなく、行全体を同時に読み取る。
- ファイル全体を処理する必要があることが分かっている場合は、読み取られるデータのサイズを判別し、これを読み取る単一バッファを割り当て、**read** を使用してファイル全体をそのバッファに一度に読み取り、それからバッファ内のデータを処理する。過度のスワッピングが起こるほどファイルが大きくなければ、これでディスク入出力が削減されます。ファイルにアクセスするために **mmap** 関数を使用することも考えてください。
- **scanf** および **fscanf** の代わりに、**fgets** を使用して文字列内を読み取り、それから **atoi**、**atol**、**atof**、または **_atold** のいずれかを使用してそれを適切な形式に変換します。
- 複雑なフォーマット設定にのみ **sprintf** を使用する。ストリングの連結のようなより単純なフォーマット設定では、より特定のストリング関数を使用します。

関数呼び出しによるオーバーヘッドの削減

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

- 関数ポインターを使用する代わりに、関数を直接呼び出す。
- 関数にグローバル変数から値を取らせるのではなく、関数に引き数として値を渡す。
- 可能であれば、インライン化された関数で定数の引き数を使用する。定数の引き数を持つ関数によって、最適化の機会が広がります。
- **#pragma isolated_call** プリプロセッサ・ディレクティブを使用して、副次作用がなく、副次作用に依存しない関数をリストする。
- ポインターの関数内の **#pragma disjoint**、または同じメモリーを指すことのできない参照パラメーターを使用する。
- 可能であれば、非メンバー関数を **static** として宣言する。これによって、関数の呼び出しを高速にできます。
-  通常は、仮想関数をインラインで宣言しない。クラス内の仮想関数がすべてインラインである場合は、仮想関数テーブルとすべての仮想関数本体が、クラスを使用する各コンパイル単位で複製されます。
-  可能であれば、関数を宣言するときに **const** 指定子を使用する。
-  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き数の関数の使用を避ける。
- 最も頻繁に使用されるパラメーターが、関数プロトタイプが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、コンパイラは多数の値のコピーおよび保管を行わなければなりません。このことは、クラス・オブジェクトが値によって渡される C++ プログラムでは不適切です。コンストラクターとデストラクターは、関数が呼び出されるときに呼び出されるからです。その代わりに、ポインターを構造体か共用体に渡すか、または戻すか、あるいは参照によってこれを渡すようにします。
- 可能であれば、**int** および **short** のような非集合体の型は、参照によって渡すのではなく、値で渡すようにする。
- ある関数が、その関数に渡されたものと同じパラメーターを指定して、別の関数の値を戻すことによって終了する場合は、関数プロトタイプ内でそのパラメーターを同じ順序にする。これにより、コンパイラは、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、またコンパイラがよりよい最適化を実行できる場合があります。

▶ **C++** ユーザー関数は、XL C/C++ ヘッダー・ファイルを組み込むと、組み込み関数に自動的にマップされます。

▶ **C** ユーザー関数は、math.h および string.h を組み込むと、組み込み関数にマップされます。

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




関連資料

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

効率的なメモリーの管理

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

- 構造体では、最もサイズの大きいメンバーから順に宣言する。
- 構造体では、一緒に使用する頻度が高い変数は、それぞれ互いに近くに置く。
- ▶ **C++** 必要なくなったオブジェクトが、確実に、解放されるか、そうでなければ再利用のために使用できるようにする。これを行う方法の 1 つに、オブジェクト・マネージャーの使用があります。オブジェクトのインスタンスを作成するたびに、そのオブジェクトへのポインターをオブジェクト・マネージャーに渡します。オブジェクト・マネージャーは、それらのポインターのリストを保守します。オブジェクトにアクセスするには、オブジェクト・マネージャーのメンバー関数を呼び出して、ユーザーまで情報を戻させます。するとオブジェクト・マネージャーは、メモリーの使用法やオブジェクトの再利用を管理します。

- ストレージ・プールは、オブジェクト・マネージャーまたは参照カウントに頼らずに、使用されているメモリーを追跡する（そしてそれを再利用する）にはよい方法です。
-  **C++** 大きな、複雑なオブジェクトのコピーは避ける。
-  **C++** シャロー・コピー だけが必要な場合は、ディープ・コピー を実行しないようにする。他のオブジェクトへのポインターを含むオブジェクトについて、シャロー・コピーはポインターだけをコピーし、それらが指すオブジェクトをコピーしません。その結果、同じものが含まれたオブジェクトを指す 2 つのオブジェクトができます。しかしディープ・コピーは、そのオブジェクト内に含まれているすべてのポインターやオブジェクトなどと同様に、ポインターとそれが指すオブジェクトをコピーします。
-  **C++** どうしても必要なときにのみ仮想メソッドを使用する。

変数の最適化

次のガイドラインを考慮してください。

- できるかぎりローカル変数（自動変数が望ましい）を使用する。

コンパイラーは、グローバル変数について、いくつかのワーストケースの想定をする必要があります。例えば、ある関数で外部変数を使用し、外部関数も呼び出す場合には、コンパイラーは、それぞれの外部関数の呼び出しで、それぞれの外部変数の値が変更される可能性があるものと想定します。グローバル変数がどの関数呼び出しにも影響を受けないこと、および混在する関数呼び出しでこの変数が複数回にわたって読み取られることが分かっている場合には、そのグローバル変数をローカル変数にコピーしてから、このローカル変数を使用してください。

- グローバル変数を使用しなければならない場合は、可能であれば、外部変数ではなく、ファイル有効範囲を指定した `static` 変数を使用してください。複数の関連する関数と `static` 変数を指定したファイルでは、変数の受ける影響について、最適化プログラムがより多くの情報を集めて使用することができます。
- 外部変数を使用しなければならない場合には、そうすることに意味があれば、外部データを構造体または配列にグループ化する。外部構造の要素はすべて、同じ基底アドレスを使用します。
- **#pragma isolated_call** プリプロセッサー・ディレクティブは、コンパイラーに、外部変数および `static` 変数のストレージについてあまり悲観的でない前提事項を作成させることによって、最適化コードの実行時パフォーマンスを向上させることができる。定数または不変ループのパラメーターを指定した `Isolated_call` 関数がループから移動し、同じパラメーターを指定した複数の呼び出しが単一呼び出しで置き換えられます。
- 変数のアドレスをとることを避ける。ローカル変数を一時変数として使用しており、そのアドレスをとらなければならない場合は、一時変数の再利用は避けてください。ローカル変数のアドレスをとると、別の状況ではその変数にかかわる計算で行われるはずの最適化が禁止されます。
- 可能な場合は変数の代わりに定数を使用する。最適化プログラムは、代わりにコンパイル時にこれを行って、実行時の計算を削減し、よりよいジョブの実行を可

能にします。例えば、ループ本体で反復回数が定数である場合は、ループ条件に定数を使用して、最適化を向上させます (for (i=0; i<4; i++) は、for (i=0; i<x; i++) よりもよく最適化されます)。

- スカラーには、レジスター・サイズの整数 (**long** データ型) を使用する。大きな整数配列には、1 バイトまたは 2 バイトの整数、あるいはビット・フィールドの使用を検討してください。
- 計算に適した、最小の浮動小数点精度を使用する。 **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。

関連資料

- 「コンパイラー・リファレンス」の **#pragma isolated_call**

効率的なストリングの操作

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

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

関連資料

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

式とプログラム・ロジックの最適化

次のガイドラインを考慮してください。

- ある式のコンポーネントがほかの式で使用されている場合には、重複する値をローカル変数に割り当てる。
- コンパイラーに、整数と浮動小数点の内部表記の間で数字を変換するように強制することは避ける。次に例を示します。

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


混合モードの算術演算を使用しなければならないときは、可能ならば、整数と浮動小数点の算術演算を別の計算でコーディングしてください。

- ループの真ん中にジャンプする **goto** 文は避けます。このような文は決まった最適化を禁止します。
- フォールスルー・パスの発生確率を高めることによって、コードの予測可能性を向上させる。次のコードがあるとした場合には、

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

次のようにコーディングする必要があります。

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

- **switch** 文の 1 つか 2 つのケースが一般的に他のケースより頻繁に実行される場合は、**switch** 文の前に別々に処理して、これらのケースを抜き出す。
-  **C++** 最適化が禁止される可能性があるため、必要なときにだけ、例外ハンドリングに **try** ブロックを使用する。
- 配列指標式は可能な限り単純にする。

64 ビット・モードでの演算の最適化

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

- 64 ビット・プログラムの方が大きい。プログラム・サイズの増加により、物理メモリーの負荷がより大きくなります。
- 64 ビットの **long** 型除法の方が、32 ビットの整数除法よりも時間がかかる。
- 64 ビット・プログラムで、配列指標に 32 ビットの符号付き整数を使用する場合は、配列を参照するたびに、符号拡張を行うための追加の命令が必要となる場合がある。

64 ビット・プログラムのパフォーマンス上のマイナス影響を補正するには、次のような方法があります。

- 32 ビットと 64 ビット混合の演算を行わないようにする。例えば、32 ビットのデータ型と 64 ビットのデータ型を加算する場合は、32 ビット型の方を符号拡張し、レジスターの上位 32 ビットをクリアする必要があります。このため、計算が遅くなります。
- 可能な限り、long 型の除法を使用しない。乗算は、多くの場合、除法よりも高速です。同じ除数で多くの除法を実行する必要がある場合は、除数の逆数を一時変数に割り当て、すべての除法を一時変数に対する乗算に変更します。例えば、次の関数を考えてみます。

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

この方が、次の直接除法よりも実行が速くなります。

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

その理由は、除法 (1.0 / 1.0825) は、コンパイル時にのみ評価され、フォールディングされるためです。

- 頻繁に使用される変数 (ループ・カウンタ、配列指標など) には、**signed**、**unsigned**、および簡潔な **int** などの型ではなく、**long** 型を使用する。このようにすると、コンパイラーが、配列参照、関数呼び出し中のパラメーター、および戻される関数結果を、切り捨てたり符号拡張したりする必要がなくなります。

付録. メモリー・デバッグ・ライブラリー関数

この付録には、XL C/C++ メモリー・デバッグ・ライブラリー関数についての参照情報が含まれています。この関数は、標準 C メモリー管理関数を拡張したものです。この付録は、2 つのセクションに分けられています。

- 『メモリー割り当てデバッグ関数』では、ヒープ・メモリーを割り当てるための、標準ライブラリー関数のデバッグ・バージョンについて説明します。
- 102 ページの『ストリング処理デバッグ関数』では、ストリングを取り扱うための、標準ライブラリー関数のデバッグ・バージョンについて説明します。

これらのデバッグ・バージョンを使用するには、以下のいずれかを行うことができます。

- ソース・コードで、任意のデフォルトまたはユーザー定義ヒープ・メモリー管理関数にプレフィックス `_debug_` を付ける。
- ソース・コードを変更したくない場合は、`-qheapdebug` オプションで単純にコンパイルする。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対応するデバッグ・バージョンにマップします。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る実例はすべて、`-qheapdebug` オプションを使用したコンパイルを前提としています。

関連資料

- 「コンパイラー・リファレンス」の `-qheapdebug`

メモリー割り当てデバッグ関数

このセクションでは、標準およびユーザーが作成したヒープ・メモリー割り当て関数のデバッグ・バージョンについて説明します。これらの関数はすべて、`_heap_check` または `_uheap_check` の呼び出しを自動的に作成し、ヒープの妥当性を検査します。次に、`_dump_allocated` 関数または `_dump_allocated_delta` 関数を使用して、ヒープ・チェック関数によって戻される情報を印刷することができます。

`_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);
```

説明

これは、`calloc` のデバッグ・バージョンです。`calloc` と同様、`num` 個の要素を持つ配列に、それぞれ `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。次に、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の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** 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生成して、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;
}
```

出力は次のようになります。

```
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);
```

説明

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。また、**_debug_free** は、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 *file* の名前と行番号 *line* を格納します。

_debug_free は、どのタイプ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するので、この関数を使用して、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・バージョンによって割り当てられたメモリー・ブロックを解放することができます。ただし、メモリー管理関数によってメモリーが割り当てられなかった場合、または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;
}
```

出力は次のようになります。

```
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);
```

説明

これは、**_heapmin** のデバッグ・バージョンです。**_heapmin** と同様、これは、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** の暗黙の呼び出しを行い、ファイル *file* の名前とメモリーが戻される行番号 *line* を格納します。

戻り値

成功した場合、0 を返します。成功しなかった場合は、-1 を返します。

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` 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生成して、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(10000))) {
        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;
}
```

次のような出力になります。

```
Header information of object 0x000738b0 was overwritten at 0x000738ac.
The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAA.
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);
```

説明

これは、**malloc** のデバッグ・バージョンです。**malloc** と同様、これは、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` の暗黙の呼び出しを行い、ファイル *file* の名前と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の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** 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生成して、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;
}
```

次のような出力になります。

```
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 — ユーザーが作成した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);
```

説明

これは、**_ucalloc** のデバッグ・バージョンです。 **_ucalloc** と同様、これは、*num* 個の要素を持つ配列に、それぞれ *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。次に、各要素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の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** が呼び出されたとき、**_uheap_check** はエラーを検出し、いくつかのメッセージを生成して、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;
}
```

出力は次のようになります。

```
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 — ユーザーが作成した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);
```

説明

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したときに指定した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 を返します。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 — ユーザーが作成したヒープからのメモリー・ブロックの予約

形式

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

説明

これは、**_umalloc** のデバッグ・バージョンです。 **_umalloc** と同様、これは、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** 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のポインターを戻します。*size* 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;
}
```

出力は次のようになります。

```
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);
```

説明

これは、**realloc** のデバッグ・バージョンです。**realloc** と同様、これは、*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、**_debug_realloc** は、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** の暗黙の呼び出しを行い、ファイル *file* の名前とストレージが再割り当てされる行番号 *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、メモリー管理関数によってメモリーが割り当てられなかった場合、または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・ブロックへのポインターを戻します。*ptr* 引き数は、戻り値と同じではありません。**_debug_realloc** は、メモリーが再割り当てされる前に解放されなかっ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 である場合、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 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生成して、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;
}
```

出力は次のようになります。


```
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.
```

関連資料

- ・ 39 ページの『メモリー・ヒープのデバッグのための関数』

ストリング処理デバッグ関数

このセクションでは、標準 C ストリング処理ライブラリーのストリング処理およびメモリー関数のデバッグ・バージョンについて説明します。これらの関数は、現行のデフォルト・ヒープのみをチェックし、ユーザーが作成した複数のヒープを使用するアプリケーション内のすべてのヒープをチェックするわけではないことに注意してください。

`_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);
```

説明

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` は、`_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

dest へのポインターを戻します。

例

次の例には、プログラミング・エラーがあります。コピー先を初期化するために使用される **memcpy** の呼び出しでは、*count* がコピー先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;
}

```

出力は次のようになります。

```

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_memmove` — バイトのコピー

形式

```

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

```

説明

これは、**memmove** のデバッグ・バージョンです。**memmove** と同様、*src* の *count* バイトを *dest* にコピーします。オーバーラップしたオブジェクト間でもコピーが可能です。

_debug_memmove 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_memmove は、暗黙の **_heap_check** 呼び出しを行います。

_debug_memmove は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

dest へのポインターを戻します。

例

次の例には、プログラミング・エラーがあります。**memmove** 呼び出しで指定された *count* は 5 ではなく 15 であるため、**memmove** 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_memmove.c */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define SIZE 21

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

    target = (char*)malloc(SIZE);
    strcpy(target, "a shiny white sphere");
    p = target+8;          /* p points at the starting character

```

```

                                of the word we want to replace */
source = target+2;                /* start of "shiny" */

printf("Before memmove, target is \"%s\\n\"", target);
memmove(p, source, 15);
printf("After memmove, target becomes \"%s\\n\"", target);
return 0;

}

```

出力は次のようになります。

```

Before memmove, target is "a shiny white sphere"
End of allocated object 0x00073c80 was overwritten at 0x00073c95.
The first eight bytes of the memory block (in hex) are: 61207368696E7920.
This memory block was (re)allocated at line number 11 in _debug_memmove.c.
Heap state was valid at line 12 of _debug_memmove.c.
Memory error detected at line 18 of _debug_memmove.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);

```

説明

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** は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

dest へのポインターを戻します。

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;

}
```

出力は次になります。

```
Buffer contents: AAAAAAAAAA
End of allocated object 0x00073c80 was overwritten at 0x00073c95.
The first eight bytes of the memory block (in hex) are: 41414141414141.
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tringの連結

形式

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

説明

これは、**strcat** のデバッグ・バージョンです。**strcat** と同様、これは、*string2* を *string1* に連結して、結果のStringをヌル文字で終わらせます。

_debug_strcat は、String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** は、**_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、そのファイル *file* の名前と行番号 *file* をメッセージで報告します。

戻り値

連結されたString *string1* へのポインターを戻します。

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、String “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;
}
```

出力は次のようになります。

```
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tringのコピー

形式

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

説明

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

_debug_strcpy は、コピー先にString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** は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

コピーされたString *string1* へのポインターを戻します。

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String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;  
}
```

出力は次のようになります。

```
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tring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);
```

説明

これは、**strncat** のデバッグ・バージョンです。**strncat** と同様、これは、*string2* の最初の *count* 個の文字を、*string1* へ付加し、結果のStringをヌル文字 (NUL) 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** は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

結合されたString *string1* へのポインターを戻します。

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、String " 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;
}
```

出力は次のようになります。

```
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tring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);
```

説明

これは、**strncpy** のデバッグ・バージョンです。**strncpy** と同様、これは、*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String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String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 は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

string1 へのポインターを戻します。

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String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_strncpy */
#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);
    strncpy(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;
}
```

出力は次のようになります。

```
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tring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);
```

説明

これは、**strnset** のデバッグ・バージョンです。**strnset** と同様、これは、*string* の多くとも最初の *n* 個の文字を *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、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_strnset** は、暗黙の **_heap_check** 呼び出しを行います。**_debug_strnset** は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

変更された *string* へのポインターを戻します。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。String *str* は、String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、ヌル終止符なしでは、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;
}
```

出力は次になります。

```
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 — String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);
```

説明

これは、**strset** のデバッグ・バージョンです。**strset** と同様、これは、末尾のヌル文字 (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、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_strset は、暗黙の **_heap_check** 呼び出しを行います。**_debug_strset** は、**_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、そのファイル *file* の名前と行番号 *line* をメッセージで報告します。

戻り値

変更されたストリングへのポインターを戻します。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;
}
```

出力は次になります。

```
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。

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

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

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

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

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

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

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

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

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

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

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

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

著作権使用許諾:

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

プログラミング・インターフェース情報

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

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

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

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

商標

以下は、IBM Corporation の商標です。

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

UNIX は、The Open Group がライセンスしている米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

業界標準

次の規格がサポートされます。

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



プログラム番号: 5724-I11

Printed in Japan

SC88-9956-00



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