Intel® C/C++ コンパイラ ユーザーズ・ガイド

ストリーミング SIMD 拡張命令対応

Copyright © 1996-1999 Intel Corporation
All Rights Reserved

資料番号 718195J-001

Intel® C/C++ コンパイラ ユーザーズ・ガイド ストリーミング SIMD 拡張命令対応

資料番号： 718195J-001

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

● 本ドキュメントの内容を予告なしに変更することがあります。

●インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

●インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

●本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

● いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複写することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社　資料販売センタ

〒 305-8603　つくば学園郵便局　私書箱 115 号

Fax: 0120-478832

『Intel© C/C++ コンパイラ ユーザーズ・ガイド』およびこれに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、そのライセンスの許諾範囲内でのみ使用または複製することができます。本書の情報は情報提供の目的でのみ提供されるもので、予告なしに変更されることがあります。本書の情報はインテルが約定として構成したものではありません。本書の内容、および本書の内容に関連して掲載されているソフトウェア製品の誤りに関して、インテルは一切の責任や義務を負いません。

ライセンス契約で許可されている場合を除き、インテルからの文書による承諾なく、本書のいかなる部分も複製したり、検索システムに保持したり、他の形式や媒体によって転送したりすることは禁じられています。

* 一般にブランド名または商品名は各社の商標または登録商標です。

Copyright© 1996-1999, Intel Corporation, All Rights Reserved

このオンライン・マニュアルの使用方法

 しおりが表示されているときにクリックすると、サブトピックが表示されるか、非表示になります。	 前のページに戻ります。
 しおりが表示されているときにダブルクリックすると、トピックにジャンプします。	 次のページに進みます。
 しおりとページを表示します。	 最後のページに移動します。
 サムネールとページを表示します。	 変更 (ジャンプ、拡大／縮小) 前のビューに戻ります。このボタンは、ジャンプまたは拡大／縮小した後、元のビューに戻る必要がある場合に使用します (下記を参照)。
 ページのみを表示します。	 戻る前のビューに進みます。
 手のひらツールを選択します。ページを上下左右にドラッグします (ページのみ表示の場合は上下にのみドラッグ)。	 ページ・ビューのサイズを 100% に設定します。
 ズームインツールを選択します。ページ上をクリックまたはドラッグするとビューが拡大します。	 ページ全体をウィンドウ内に表示します。
 ズームアウトツールを選択します。ページ上をクリックまたはドラッグするとビューが縮小します。	 ウィンドウの幅に合わせて表示します。
 テキスト選択ツールを選択します。ページ上でドラッグしてテキストを選択します。	 単語を検索するためのダイアログを開きます。
 マニュアルの最初のページに移動します。	

オンライン・ファイルの印刷： オンライン・ファイルを印刷するには、[ファイル] メニューの [印刷] を選択します。表示されたダイアログで、全ページ、指定範囲のページ、または選択したページを印刷できます。

複数のオンライン・マニュアルの表示： [ファイル] メニューの [開く] を選択し、希望する PDF ファイルを開きます。複数のファイルを表示するには、[ウィンドウ] メニューの [重ねて表示] を選択します。

しおり表示領域のサイズ変更： 領域の境界 (矩形の角) にカーソルを合わせ (カーソルの両端が矢印になる)、任意の方向にドラッグします。

トピックへのジャンプ： このマニュアルでは、関数名やセクションの見出しの多くが緑色の下線付きスタイルで表示されています。これは、そのトピックにジャンプできることを示しています。

次のページでは、ジャンプ・テキストの例を示します。

ジャンプ・テキストの例

以下のテキストは、マニュアルから抜き出したものです。この例では、セクション名「最適化の制限」が緑の下線付きフォントで表示されています。このセクション名をクリックすると、このセクションの最初のページが表示されます。

このページに戻るには、ツール・バー上の  アイコンをクリックします。

> この章では、アプリケーションの性能を向上させる方法と、標準コンパイラ・オプションがプログラムに及ぼす効果について説明します。「最適化の制限」の項では、特定のアプリケーションやデバッグに対して最適化を行わない方法を説明します。

目次

第 6 章 コンパイル出力の指定

第 7 章 前処理

第 8 章 C/C++ 言語機能

第 9 章 Microsoft との互換性

第 10 章 診断情報

第 11 章 ライブラリ

第 12 章 コンパイラ生成コードの制御

第 13 章 MMX® テクノロジとストリーミング SIMD 拡張命令のサポート

例

図

表

本書について

本書では、Intel® C/C++ コンパイラ (Win32* システム対応版) の使い方を説明しています。Intel C/C++ コンパイラは、Windows NT* または Windows* 95 オペレーティング・システム上で動作します。

- 第 1 章から第 3 章までは、初めてのユーザのための説明です。
- 第 2 章の「コンパイラ・オプション・クイック・ガイド」では、コンパイラ・オプションをアルファベット順に一覧表示し、本文での詳しい説明箇所の参照ページを示してあります。
- 第 4 章では、一般によく使用されるパフォーマンスの最適化について説明します。
- 第 5 章では、プロシージャ間の最適化、およびプロファイルによる最適化について説明します。
- 第 6 章から第 9 章では、前処理および言語への適合の方法について説明します。
- 第 10 章では診断情報について説明します。
- 第 11 章ではライブラリについて説明します。
- 第 12 章では、コンパイラ生成コードを制御する方法を説明します。
- 第 13 章では、MMX® テクノロジおよびストリーミング SIMD 拡張命令のサポートについて説明します。
- 第 14 章では、ベクトル化のオプションについて説明します。
- 付録 A ではコンパイラの制限を示す。
- 付録 B では実験的なパフォーマンス調整について説明します。このマニュアルには、用語集と索引も含まれています。

本書は、C プログラミング言語について説明するものではありません。ソフトウェア開発プロセスにおける C/C++ およびアセンブリ言語プログラムの役割を理解していることが前提となります。また、ホスト・コンピュータのオペレーティング・システムおよび Intel® プロセッサ・アーキテクチャについての知識も必要です。

Intel C/C++ コンパイラは、できる限り Microsoft Visual C++* と同様に動作するよう設計されています。多くの場合、Microsoft Visual C++ コンパイラについての機能説明は、Intel コンパイラにも適用できます。本書では、適用されない場合についてはその旨を記述しています。Intel C/C++ コンパイラとシステム固有のプログラミング・サポート・ツールの関係については、第 1 章の「アプリケーション開発」を参照してください。

参考文献

Intel C/C++ コンパイラについては、以下のドキュメントも参照してください。

- 『The Annotated C++ Reference Manual』第 1 版、Margaret Ellis、Bjarne Stroustrup 著、Addison Wesley 刊、1991 年。C++ プログラミング言語について解説しています。
- 『The C Programming Language』第 2 版、Brian W. Kernighan、Dennis W. Ritchie 著、Prentice Hall 刊、1988 年。C 言語の K&R 定義について解説しています。
- 『C: A Reference Manual』第 3 版、Samual P. Harbison、Guy L. Steele 著、Prentice Hall 刊、1991 年。C 言語の ANSI 標準および拡張機能について解説しています。
- 環境についての詳細は、Microsoft Visual C++ for Windows32 ビット版に付属のマニュアルまたはオンライン・ヘルプを参照してください。
- Win32 固有の情報については、『Microsoft Win32 Software Development Kit, Version 3.1』に付属のマニュアルを参照してください。

ターゲット・アーキテクチャに関する情報は、Intel および専門書店で入手できます。次のドキュメントが役に立ちます。

- 『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、上巻：基本アーキテクチャ』インテル株式会社、資料番号 243190J
- 『インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻：命令セット・リファレンス』インテル株式会社、資料番号 243191J
- 『インテル・プロセッサの識別と CPUID 命令』インテル株式会社、資料番号 241618J
- 『インテル・アーキテクチャ MMX テクノロジ・プログラマーズ・リファレンス・マニュアル』インテル株式会社、資料番号 443007J
- 『Pentium® PRO ファミリ・デベロッパーズ・マニュアル（3 巻セット）』インテル株式会社、資料番号 242690J、242691J、242692J
- 『Pentium II Processor Developer's Manual』Intel Corporation、資料番号 243502-001

- 『Pentium® プロセッサ・仕様アップデート』インテル株式会社、資料番号 242480J
- 『Pentium® ファミリー・デベロッパーズ・マニュアル、上巻：データブック』インテル株式会社、資料番号 241428J
- Intel のほとんどのドキュメントは、Intel Corporation のウェブ・サイト (www.intel.com) からも入手できます。

以下に、参考 URL を示します。

- 『Developer's Insight』 *http://developer.intel.com/sites/developer/*
- 『開発者のためのホームページ』 *http://www.intel.co.jp/jp/design/*

表記の規則

本書での表記は、以下の規則に従います。

`This type style`	構文の要素、予約語、キーワード、ファイル名、コンピュータ出力、プログラム例の一部分のいずれかを表します。テキストは、大文字に意味がない限り小文字で表記します。 例では、`l` は小文字の L、1 は数字の 1 です。また、`O` は大文字の O、`0` は数字の 0 です。
`This type style`	入力としてユーザがタイプする文字そのものを表します。
`This type style`	識別子、式、文字列、記号、値のいずれかのプレースホルダを表します。これらのアイテムのいずれかをプレースホルダと置き換えてください。
`[items]`	角括弧で囲まれたアイテムはオプションです。
`{item | item}`	中括弧内のアイテムのリストから 1 つだけ選択します。縦の線 (|) はアイテムの区切りです。
`...` (ellipses)	省略記号は、前のアイテムを複数指定できることを表します。

概要

1

この章では、Intel C/C++ コンパイラ (Win32 システム対応版) の概要を説明します。このコンパイラは、すべての 32 ビット Intel アーキテクチャのマシン上で動作するよう設計されています。しかし、このコンパイラで最適化コードを生成できるのは、Intel486™、Pentium®、Pentium Pro、Pentium II、Celeron™、Pentium II Xeon™、および Pentium III プロセッサについてです。

必要なツール

このコンパイラでプログラムをコンパイルするには、以下のツールが必要です。

- Windows NT、Windows 95 または Windows 98 オペレーティング・システム
- ホストのオペレーティング・システム上で使用できる Microsoft Visual C++ Version 4.2、またはそれ以上。

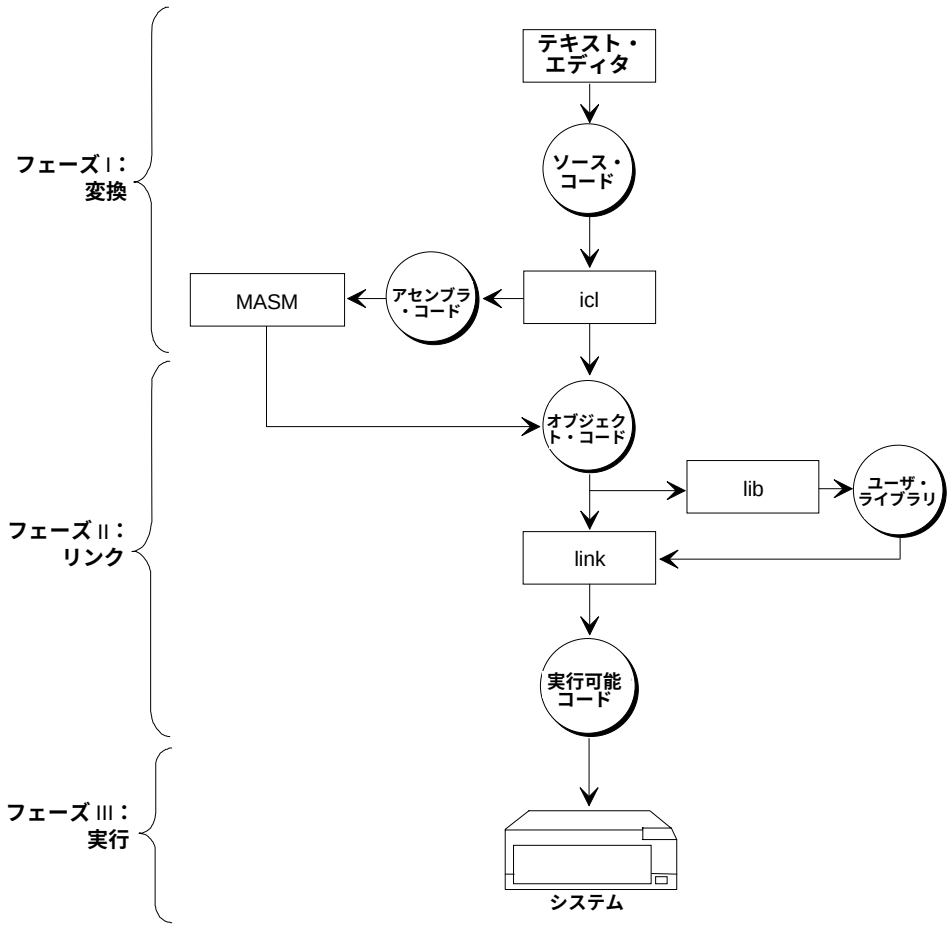
注：*Intel C/C++ コンパイラは、Microsoft Visual C++ のヘッダ・ファイル、ライブラリ、およびその他の多くのコンポーネントを使用するプラグイン製品です。*

アセンブラで生成したファイルをアセンブルする場合、アセンブラが必要です。アセンブラとしては Microsoft Macro Assembler (MASM) のバージョン 6.12 を推奨しますが、ストリーミング SIMD 拡張命令を使用する予定がない場合は、バージョン 6.11 でもかまいません。どのアセンブラを使用する場合でも、その名前を `ML.EXE` にして、デフォルトのパスに保存しなければなりません。

アプリケーション開発

図 1-1 に、Microsoft アプリケーション開発環境で、どのように Intel コンパイラ (icl) を使用するかを示します。このコンパイラは、C または C++ 言語のソースを処理し、アセンブリまたはオブジェクト・モジュールのいずれかを生成します。入力と出力は、コンパイラの実行時にオプションを設定して決定します。

図 1-1 アプリケーション開発サイクル



コンパイラの操作 2

この章では、コンパイラの実行方法、コンパイル・オプションの選択方法、およびコンパイラのデフォルトの動作について説明します。

この章の「コンパイラ・オプション・クイック・ガイド」には、Intel C/C++ コンパイラで使用可能なすべてのオプションを掲載してあります。オプションを指定することによって、以下の点についてコンパイラのデフォルトの動作が変更されます。

- コンパイル・フローの制御
- コンパイラの出力の制御
- ファイルの前処理
- 最適化の制御
- 言語準拠の指定
- デバッグの準備
- リンカの制御
- 特殊なケースの管理
- プロファイルの準備

Intel C/C++ コンパイラで使用可能なオプションについては、第 3 章以降で詳しく説明します。

コンパイラのコマンド・ライン構文

DOS のコマンド・ラインからコンパイラを起動するには、次のコマンドを入力します。

```
prompt> icl [options] [path]filenames
```

options	1 つまたは複数のコマンド・ライン・オプションを示します。コンパイラは、ハイフン (-) またはスラッシュ (/) に続く 1 つ以上の文字をオプションとして認識します。
filenames	コンパイル・システムで処理する 1 つまたは複数のソース・ファイルを示します。複数の [*path*]*filename* を指定できます。複数のファイル名はスペースで区切らなければなりません。

Microsoft Visual C++ 統合開発環境内で Intel® C/C++ コンパイラを使用する

Microsoft Visual C++ V4.2 またはそれ以上がインストールされているシステムに Intel C/C++ コンパイラをインストールすると、自動的に Intel C/C++ コンパイラがマイクロソフト開発環境 (IDE) に統合されます。これにより、Intel C/C++ コンパイラと Microsoft C/C++ Compiler 間のスイッチ・オプションの互換性が得られます。さらに、Intel の最適化機能により、Intel アーキテクチャの能力を十分引き出すことができます。マイクロソフト IDE から Intel C/C++ コンパイラを起動するには以下の手順に従ってください。

1. [ツール] メニューから、[コンパイラ選択] をクリックします。
2. [コンパイラ選択] ウィンドウがマイクロソフト開発スタジオに表示されますので、コンパイラを選択します。
3. [Intel コンパイラ] を選択するため、チェック・ボックスをクリックし、下向き矢印のメニューからバージョンを指定します。
4. OK ボタンをクリックし、マイクロソフト IDE 環境に戻ります。

注：*Microsoft コンパイラに切り替えるには、Intel C/C++ コンパイラのチェック・ボックスをクリアします。*

以上で、マイクロソフト IDE 環境において、選択したコンパイラを使用したアプリケーション開発を行う準備ができました。Intel C/C++ コンパイラの詳細な使用方法については、Visual C++ 開発環境の [ヘルプ] ボタンをクリックしてください。

ファイル名拡張子

デフォルトでは、このコンパイラは拡張子が .cc、.cpp、.cxx のファイルを C++ のファイルとして認識します。本書で例を挙げて示す場合には、C++ のファイルについては拡張子 .cpp を使用しています。拡張子が .c のファイルは、C のファイルとして認識されます。さらに、Intel C/C++ コンパイラは、表 2-1 に示すデフォルトのファイル名拡張子を認識します。

表 2-1 デフォルトのファイル名拡張子

ファイル名	コンパイラの認識	処理
filename.lib	**オブジェクト・ライブラリ**	LINK.exe **に渡されます。**
filename.i	C/C++ **プリプロセッサにより前処理および展開された** C **または** C++ **のソース**	**コンパイラに渡されます。**
filename.obj	**コンパイル済みのオブジェクト・モジュール**	LINK.exe **に渡されます。**
filename.asm	**アセンブリ・ファイル**	MASM **によりアセンブルされます。**

-TP *[files]* の形式でオプションを指定すると、コンパイラはコマンド・ラインに入力されているすべてのファイルを C/C++ ファイルとして処理します。例えば、次のコマンド・ラインでは、x.i も y.c も共に C++ のファイルとして処理されます。

```
prompt> icl -c -TP x.i y.c
```

ただし、**-Tp** *file* の形式でオプションを指定した場合は、拡張子に関係なく、コマンド・ラインで -TP の直後に指定されたファイルが C++ のファイルとして処理されます。例えば、x.i を C++ のファイル、y.c を C のファイルとして処理するには、次のように入力します。

```
prompt> icl -c y.c -Tp x.i
```

注：*通常、コマンド・ラインでは、1 つのオプションを 1 つのファイル名にだけ対応付けることはできません。指定したオプションはすべてのファイルに適用されます。例えば、次のコマンド・ラインの場合、-Za オプションは x.cpp と y.cpp の両方のファイルに適用されます。*

```
prompt> icl -c x.cpp -Za y.cpp
```

ただし、-Tc オプションおよび -Tp オプションは例外です。これらのオプションは、コマンド・ラインでその直後に指定されているファイルだけに適用されます。

コンパイラ・オプション・クイック・ガイド

表 2-2 に、すべてのコンパイル制御オプションおよび一部のリンカ制御オプションを示します。

表 2-2 コマンド・ライン・オプション一覧

オプション	説明	デフォルト	参照先
`-?, -help`	コンパイラ・オプションの一覧を出力します。	オフ	10-1 ページ
`-c`	オブジェクト・ファイルが生成された後、コンパイル処理を停止します。コンパイラは、C または C++ の各ソース・ファイルまたは前処理されたソース・ファイルからオブジェクト・ファイルを生成します。	オフ	6-4 ページ
`-C`	前処理されたソースの出力にコメントを配置します。	オフ	7-2 ページ
`-Dname[=value]`	マクロ名を定義し、そのマクロを指定された値に関連付けます。	オフ	7-3 ページ
`-E`	C または C++ のソース・ファイルが前処理された後、コンパイル処理を停止し、結果を `stdout` に書き込みます。	オフ	7-2 ページ
`-EHa`	非同期型 C++ 例外処理モデルを有効にします。	オフ	*
`-EHc`	`extern` C 関数が例外処理を throw しないように、指定します。	オン	*
`-EHs`	同期型 C++ 例外処理モデルを有効にします。	オフ	*
`-EP`	C または C++ のソース・ファイルが前処理された後、コンパイル処理を停止し、結果を `stdout` に書き込みます。`#line` ディレクティブは除去されます (出力されません)。	オフ	7-2 ページ
`-F n`	`-stack:`*n* をリンカに渡すことで、プログラムに予約するスタックの量を設定します。	オフ	*

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-FA[c s]	アセンブリ出力ファイルを生成します。引数 s は、Intel C コンパイラでは無効です。	オフ	*
-Fa[filename]	指定したファイル名 (ファイル名を指定しなかった場合はデフォルトのファイル名) でアセンブリ出力ファイルを生成します。	オフ	6-5 ページ
-Fe[filename]	指定したファイル名 (ファイル名を指定しなかった場合はデフォルトのファイル名) で実行可能出力ファイルを生成します。	オン	6-5 ページ
-Fm[filename]	リンカに対してマップ・ファイルの生成を指示します。	オフ	*
-Fo[filename]	指定したファイル名 (ファイル名を指定しなかった場合はデフォルトのファイル名) でオブジェクト出力ファイルを生成します。	オン	6-5 ページ
-G4	最適化のターゲットを Intel i486 プロセッサにします。	オフ	4-3 ページ
-G5	最適化のターゲットを Intel Pentium プロセッサにします。	オフ	4-3 ページ
-G6	最適化のターゲットを Intel Pentium Pro および Pentium II プロセッサにします。	オン	4-3 ページ
-GA	Microsoft Windows アプリケーション用に最適化します。	オフ	*
-Ge	スタック・チェックを有効にします。	オン	*
-Gf	文字列プールの最適化を有効にします。	オン	*
-GF	読み取り専用の文字列プールを有効にします。	オン	*
-Gh	ユーザ定義の __penter ルーチンの呼び出しを各関数プロローグに追加します。	オフ	*

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-GR	C++ 実行時タイプ情報 (RTTI) を有効にします。	オフ	*
-Gr	デフォルトの読み出し規則として Microsoft _fastcall を選択します。	オフ	*
-GR-	C++ RTTI を無効にします。	オン	*
-Gs *n*	*n* バイト以上のローカル変数およびコンパイラの一時変数を使用するルーチンのスタック・チェックを有効にします。	オフ	*
-GT	ファイバーセーフのスレッド・ローカル・ストレージ (TLS) を有効にします。	オフ	*
-GX	C++ の例外処理を有効にします。	オフ	*
-GX-	C++ の例外処理を無効にします。	オン	*
-Gy	リンク時の関数の順序変更を有効にします。	オン	*
-H *n*	外部シンボル名の長さを *n* 文字に制限します。	オフ	*
-Idirectory	インクルード・ファイルを検索する追加ディレクトリを指定します。	オフ	3-4 ページ
-J	デフォルトの char 型を符号なしにします。	オフ	*
-LD	DLL を生成します。	オフ	*
-LDd	デバッグ・バージョンの DLL を生成します。	オフ	*
-link	オプションをリンカに渡します。	オフ	3-5 ページ
-MD	コンパイル後、ダイナミック・ライブラリ (マルチスレッドの C のランタイム・ライブラリ) にリンクします。	オフ	*
-MDd	コンパイル後、ダイナミック・ライブラリ (マルチスレッド、デバッグ・バージョンの C のランタイム・ライブラリ) にリンクします。	オフ	*

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-ML	コンパイル後、スタティック・ライブラリ (シングルスレッドの C のランタイム・ライブラリ) にリンクします。	オン	*
-MLd	コンパイル後、スタティック・ライブラリ (シングルスレッド、デバッグ・バージョンの C のランタイム・ライブラリ) にリンクします。	オフ	*
-MT	コンパイル後、スタティック・ライブラリ (マルチスレッドの C のランタイム・ライブラリ) にリンクします。	オフ	*
-MTd	コンパイル後、スタティック・ライブラリ (マルチスレッド、デバッグ・バージョンの C のランタイム・ライブラリ) にリンクします。	オフ	*
-O1	速度について最適化を行いますが、コード・サイズを大きくするだけでさほどの高速化にはつながらない最適化を無効にします。-O1 オプションは、-Og、-Oi-、-Os、-Oy、-Ob1、-Gf、-Gs、および -Gy オプションを指定するのと同じ効果があります。	オフ	4-1 ページ
-O2	速度について最適化を行います。-O2 オプションは、-Og、-Oi、-Ot、-Oy、-Ob1、-Gf、-Gs、および -Gy オプションを指定するのと同じ効果があります。	オン	4-1 ページ

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-Ob*n*	コンパイラのインライン展開を制御します。実行されるインライン展開の範囲は、次に示す *<n>* の値によって異なります。 0 インライン展開を無効にします。 1 _inline キーワードによって宣言された関数についてのインライン展開を有効にします。また、C++ 言語に準拠したインライン展開を有効にします。 2 あらゆる関数のインライン展開を有効にします。ただし、どの関数をインライン展開するかはコンパイラが判断します。このオプションによってプロシージャ間の最適化が有効となり、-Qip オプションを指定するのと同じ効果が得られます。	オフ	5-4 ページ
-Od	最適化を無効にします。	オフ	4-1 ページ
-Og	広域的な最適化を有効にします。	オン	*
-Oi	標準ライブラリ関数のインライン展開を有効にします。	オン	4-6 ページ
-Oi-	標準ライブラリ関数のインライン展開を無効にします。	オフ	4-6 ページ
-Op	浮動小数点演算を ANSI C および IEEE 754 に準拠させることを指定します。NaN 比較に関する処理は、その対象とはなりません。	オフ	4-7 ページ
-Op-	浮動小数点演算について、ANSI C および IEEE 754 への準拠よりも最適化を優先します。	オン	4-7 ページ
-Os	ほとんどの速度の最適化を有効にしますが、コード・サイズを大きくするだけでさほどの高速化にはつながらない最適化を無効にします。	オフ	*

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-Ot	すべての速度の最適化を有効にします。	オン	*
-Ox	-Gf および -Gy オプションを指定しない -O2 オプションと同じです。	オフ	*
-Oy	最適化において ebp レジスタの使用を有効にします。	オン	6-6 ページ
-Oy-	最適化において ebp レジスタの使用を無効にします。代わりに、レジスタはフレーム・ポインタとして使用されます。	オフ	6-6 ページ
-P	C または C++ のソース・ファイルが前処理された後、コンパイル処理を停止し、コンパイラのデフォルトのファイル命名規則に従って名前が付けられたファイルに結果を書き込みます。	オフ	7-2 ページ
-QA-	すべての事前定義マクロ (先頭が __ のものを除く) およびすべての表明を非アクティブにします。(-u と同じ。)	オフ	7-3 ページ
-QAname[(values)]	シンボル名を指定した値のシーケンスと関連付けます。#assert 前処理ディレクティブと等価です。	オフ	7-3 ページ
-Qax[extensions]	プロセッサ固有の拡張命令に対し、専用コードおよび汎用コードを生成します。	オフ	4-4 ページ
-QH	インクルード・ファイルの依存関係を、1 行に 1 つずつ stdout に出力します。	オフ	7-5 ページ
-QI0f	旧型プロセッサをターゲットとしたコードについて、特定の 0f 命令の不正なデコードを回避します。	オフ	11-4 ページ
-QIfdiv	Pentium プロセッサのステップ実行に存在する浮動小数点除算のソフトウェア・パッチを有効にします。	オフ	11-3 ページ
-QIfdiv-	Pentium プロセッサのステップ実行に存在する浮動小数点除算のソフトウェア・パッチを無効にします。	オン	11-3 ページ

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-Qip	プロシージャ間の最適化を有効にします。	オフ	5-2 ページ
-Qip_no_inlining	-Qipまたは-Ob2オプションを指定した場合のプロシージャ間の最適化によって実行されるインライン展開を無効にします。ただし、これ以外の場合のプロシージャ間の最適化には影響しません。また、ユーザ指定によるインライン展開 (-Ob1) にも影響しません。	オフ	5-4 ページ
-Qipo	複数のファイルにわたるプロシージャ間の最適化を有効にします。	オフ	5-2 ページ
-Qkscalar	デフォルトの x87 命令ではなく、ストリーミング SIMD 拡張命令を使用して、すべての 32 ビット浮動小数点演算を実行します。	オフ	14-3 ページ
-Qlocation,*tool*,*path*	*path* にパスを指定して、代替ツール *tool* を指定します。	オフ	3-4 ページ
-Qlong_double	long double データ型のデフォルトのサイズを 64 ビットから 80 ビットに変更します。	オフ	4-10 ページ
-QM	ソース・ファイルの #include 行に基づいて、各ソース・ファイルについて makefile 依存性の行を生成します。	オフ	7-6 ページ
-Qnobss_init	DATA セクションに 0 で初期化された変数を配置します。	オフ	12-2 ページ
-Qoption,*tool*,*list*	アセンブラやリンカなどのコンパイル・シーケンスにおける他のプログラムに引数並びを渡します。	オフ	3-4 ページ
-Qpc *nn*	浮動小数点数の仮数部の精度の制御を有効にします。nn の値は、仮数部の値を正しいビット数に丸めるために使用されます。*nn* の値は、32、64、または 80 のいずれかです。	nn=64	4-8 ページ
-Qpf[options]	プリフェッチにより、キャッシュ使用率を向上させます。	オフ	4-5 ページ

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-Qprec	-Op ほどの速度は得られませんが、浮動小数点数の精度を向上させます。	オフ	4-8 ページ
-Qprec_div	浮動小数点の除算演算から乗算演算への最適化機能を無効にします。	オフ	4-8 ページ
-Qprof_dir *dirname*	プロファイル情報を保持するディレクトリを指定します。	オフ	5-5 ページ
-Qprof_gen	基本的なブロックの実行回数を収集するプログラムを組み込みます。	オフ	5-5 ページ
-Qprof_genx	機能が組み込まれたオブジェクト・ファイルを生成し、新しい静的プロファイル情報ファイル (.spi) を作成します。	オフ	5-5 ページ
-Qprof_use	動的フィードバック情報を使用します。	オフ	5-5 ページ
-Qrcd	FPU の丸め制御の変更を無効にします。	オフ	4-10 ページ
-Qrestrict	restrict 修飾子と一緒に指定して、ポインタの明確化を有効にします。	オフ	14-3 ページ
-Qsfalign-	すべての関数に対して、スタックのアライメントを無効にします。	オフ	13-50 ページ
-Qsfalign16	16 バイトの変数と一緒に指定して、関数用のスタックのアライメントを行います。	オフ	13-50 ページ
-Qsfalign8	8 バイトまたは 16 バイトの変数と一緒に指定して、関数用のスタックのアライメントを行います。	オン	13-50 ページ
-Qunroll*n*	ループのアンロール回数の上限を指定します。	*n* = 0	4-6 ページ
-Quse_asm	ソース・ファイルをアセンブリ・ファイルにコンパイルし、MASM を呼び出してオブジェクト・ファイルを生成します。	オフ	6-4 ページ
-Qvc4	Visual C++ 4.x との互換性を有効にします。	オフ	9-2 ページ
-Qvc5	Visual C++ 5.0 との互換性を有効にします。	オン	9-2 ページ

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-Qvc6	Visual C++ 6.0 との互換性を有効にします。	オン	9-2 ページ
-Qvec	ベクトライザを有効にします。	オフ	14-2 ページ
-Qvec_alignment	ベクトル化可能なデータに対するデフォルトのアライメントを制御します。	オフ	14-2 ページ
-Qvec_emms[-]	ループをベクトル化した後に、コンパイラが自動的に EMMS 命令を挿入するかどうかを制御します。	オフ	14-3 ページ
-Qvec_no_alias[-]	名前の異なるオブジェクト間ではエイリアシングがないものと仮定します。	オフ	14-4 ページ
-Qvec_no_arg_alias[-]	エントリ時にプロシージャの引数がエイリアシングされないものと仮定します。	オフ	14-4 ページ
-Qvec_verbose	ベクトライザによって設定される診断レベルを制御します。	オフ	14-3 ページ
-Qwd*tag*	*tag* に対応するソフト診断を無効にします。	オフ	10-5 ページ
-Qwe*[tag]*	*tag* に対応するソフト診断の重要度をエラーに変更します。	オフ	10-5 ページ
-Qwn*num*	コンパイルを中断するまでに表示されるエラーの数を *num* に制限します。	100	10-6 ページ
-Qwr*[tag]*	*tag* に対応するソフト診断の重要度をリマークに変更します。	オフ	10-5 ページ
-Qww*[tag]*	*tag* に対応するソフト診断の重要度を警告に変更します。	オフ	10-5 ページ
-Qx[extensions]	プロセッサ固有の拡張命令に対し、専用コードを生成します。	オフ	4-4 ページ
-S	アセンブリ・ソースが生成された後、コンパイル処理を停止します。コンパイラは、C または C++ の各ソース・ファイルまたは前処理されたソース・ファイルに基づいて、アセンブリ・コード・ソースを出力ファイル (名前は命名規則に従って付けられる) に書き込みます。	オフ	6-2 ページ

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-TC	すべてのソース・ファイルまたは認識されていないタイプのファイルを C のソース・ファイルとしてコンパイルします。	オフ	*
-Tc *file*	*file* を C のソース・ファイルとして処理します。	オフ	*
-TP	すべてのソース・ファイルまたは認識されていないタイプのファイルを C++ のソース・ファイルとしてコンパイルします。	オフ	*
-Tp *file*	*file* を C++ のソース・ファイルとして処理します。	オフ	*
-u	すべての事前定義マクロ (__ で始まるものを除く) と表明を無効にします。(-QA- と同じ。)	オフ	*
-Uname	マクロ名のすべての定義を抑止します。#undef 前処理ディレクティブと等価です。	オフ	7-3 ページ
-V *text*	バージョン文字列を *text* に設定します。	オフ	*
-vd0	C++ の隠し vtordisp コンストラクタ / デストラクタ置換メンバーを抑止します。	オフ	*
-vd1	C++ の隠し vtordisp コンストラクタ / デストラクタ置換メンバーを生成します。	オン	*
-vmb	メンバーへのポインタの最小表現を選択します。このオプションは、クラスのメンバーのポインタを宣言する前に、各クラスの定義をする場合に使用します。	オン	*
-vmg	メンバーへのポインタの一般的な表現を選択します。このオプションは、対応するクラスを定義する前に、メンバーへのポインタを宣言する場合に使用します。	オフ	*

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-vmm	-vmg と一緒に指定して、単一または多重継承のクラス・メンバーへのポインタを有効にします。	オフ	*
-vms	-vmg と一緒に指定して、単一継承のクラス・メンバーへのポインタを有効にします。	オフ	*
-vmv	-vmg と一緒に指定して、任意の継承タイプのクラス・メンバーへのポインタを有効にします。	オフ	*
-W0	エラー・メッセージだけを表示するように指定します。	オフ	10-4 ページ
-W1, -W2, -W3	エラーおよび警告メッセージを表示することを指定します。	オン	10-5 ページ
-W4	エラー、警告、およびリマーク・メッセージを表示することを指定します。	オフ	10-6 ページ
-X	インクルード・ファイルを検索するディレクトリのリストから標準ディレクトリを削除します。	オフ	3-3 ページ
-Za	C の ANSI 標準への厳密な準拠を強制します。	オフ	8-2 ページ
-Zd	オブジェクト・ファイルに行番号情報のみ書き込みます (デバッグ用)。	オフ	*
-Ze	拡張 C 言語を受け入れます。	オン	8-2 ページ
-Zi	ソース・レベル・デバッガで使用するための、シンボリック・デバッグ情報をオブジェクト・コードに生成します。	オフ	6-6 ページ
-Zl	オブジェクト・ファイルへのデフォルト・ライブラリの埋め込みを無効にします。	オフ	*
-Zp[*number*]	構造体および共用体に、1、2、4、8、16 バイトのいずれかの、最も厳密な整列条件を指定します。	8	12-1 ページ

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

表 2-2 コマンド・ライン・オプション一覧 (続き)

オプション	説明	デフォルト	参照先
-Zs	プログラムの構文をチェックし、C または C++ のソース・ファイルおよび前処理されたソース・ファイルを構文解析した後、コンパイル処理を停止します。コードおよび出力ファイルは生成しません。警告およびメッセージは stderr に出力されます。	オフ	6-2 ページ

* 「参照先」の欄にアスタリスクが示されているオプションは、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラで結果が同じになります。これらのオプションについての詳細は、Microsoft Visual C++ に付属のマニュアルを参照してください。

コンパイラのデフォルトの動作

Intel C/C++ コンパイラを起動する際、オプションは何も指定しなくてもかまいません。オプションを何も指定しなかった場合、コンパイラはデフォルトの設定を使用します。コンパイラのドライバは、デフォルトで以下の処理を実行します。

- 実行可能ファイルを生成します。ファイル名は、コマンド・ラインの最初のソース・ファイルまたはオブジェクト・ファイルの名前に拡張子 `.exe` が付いたものになります。
- `-G6` オプションを使用して、Pentium Pro または Pentium II プロセッサを最適化します。
- `INCLUDE` 変数を使用してインクルード・ファイルを検索します。
- カレント・ディレクトリにライブラリ・ファイルが見つからない場合は、`LIB` 変数で指定されたディレクトリでライブラリ・ファイルを検索します。
- 構造体の最も厳密なアライメント境界として 16 バイトを設定します。
- エラー・メッセージおよび警告メッセージを表示します。
- C のソース・ファイルに対して ANSI および拡張機能 (`-Ze`) を適用します。
- 第 4 章の「最適化の選択肢」で説明されている、デフォルトの `-O2` オプションを使用して標準的な最適化を実行します。

コンパイラがコマンド・ライン・オプションを認識しない場合、そのオプションは無視され、警告が表示されます。システム・メッセージについての詳細は、第 10 章「診断情報」を参照してください。

コンパイル環境の変更 3

この章では、コンパイル環境をカスタマイズする方法について説明します。カスタマイズでは、次の項目を指定できます。

- コンパイル・ツールの名前と場所
- 各コンパイルで使用するオプション
- 個々のプロジェクトで使用するオプションとファイル
- インクルード・ファイルやライブラリを検索するディレクトリ

環境変数

環境のカスタマイズにあたって、ライブラリやインクルード・ファイルのようなファイルをコンパイラが検索する際のパスを指定します。Win32* システムでは、Systems Properties(システム・プロパティ) ウィンドウの環境タブで、これらの変数を指定します。コントロール・パネルのシステム・アイコンをクリックすると、このウィンドウが開きます。コンパイル環境に関連する変数には以下のものがあります。

- `LIB` は、数値演算ライブラリへのディレクトリ・パスを指定します。また、コンパイラは Microsoft のリンカである `LINK.exe` を呼び出して、オブジェクト・ファイルから実行可能ファイルを作成します。このリンカは、この `LIB` 環境変数に指定されているパスでライブラリを検索します。
- `PATH` は、コンパイラの実行可能ファイルへのディレクトリ・パスを指定します。
- `INCLUDE` は、インクルード・ファイルへのディレクトリ・パスを指定します。
- `TMP` は、一時ファイルを保存するディレクトリを指定します。`TMP` で指定したディレクトリが存在しない場合、コンパイラはカレント・ディレクトリに一時ファイルを作成します。

設定ファイル

icl.exe がインストールされたディレクトリに、icl.cfg という名前のユーザ設定ファイルを作成できます。このファイルでは、各コンパイルで使用する共通のオプションを指定できます。設定ファイルのオプションは指定された順序で処理され、その後、コンパイラ起動時にコマンド・ラインで指定したオプションが処理されます。

設定ファイルを使用してコマンド・ラインの入力を自動化することによって、コマンド・ライン・オプションを入力する時間を短縮し、一貫性を維持できます。しかし、設定ファイル内のオプションは、コンパイラを実行するたびに指定されることに注意してください。プロジェクトごとに異なるオプションを指定しなければならない場合は、この章の「応答ファイル」を参照してください。

このファイルには、有効な任意のコマンド・ライン・オプションを指定できます。ポンド記号「#」を使用すると、行の残りの部分はコメントとして処理されます。

例 3-1 に、icl.cfg ファイルのサンプルを示します。

例 3-1 icl.cfg ファイルのサンプル

```
## Sample icl.cfg file.
##
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
##
## Additional directories to be searched for include
## files, before the default.
-Ic:¥project¥include
##
## Use the static, multi-threaded C run-time library.
-MT
```

応答ファイル

応答ファイルを使用することによって、特定のコンパイルで使用するオプションを指定し、その情報を個別のファイルに保存できます。応答ファイルは、コマンド・ラインでオプションとして呼び出されます。応答ファイル内のオプションは、コマンド・ライン上で応答ファイルが呼び出された場所に挿入されます。

応答ファイルを使用してコマンド・ラインの入力を自動化することによって、コマンド・ライン・オプションを入力する時間を短縮し、一貫性を維持できます。さらに、個別の応答ファイルに特定のプロジェクト用のオプションを保持することができ、プロジェクトを別のプロジェクトに変更したときに設定ファイルを編集する必要がありません。

応答ファイルの 1 行には、オプションやファイル名をいくつでも指定できます。1 つのコマンド・ラインで複数の応答ファイルを指定することもできます。このファイルでは、有効な任意のコマンド・ライン・オプションを指定できます。ポンド記号「#」を使用すると、行の残りの部分はコメントとして処理されます。

応答ファイルを使用する構文は次のとおりです。

```
prompt> icl @response_file filenames
```

コマンド・ライン上の応答ファイル名の前には、アットマーク (@) を入力する必要があります。

インクルード・ファイル

デフォルトでは、コンパイラは環境変数 INCLUDE で指定されたディレクトリで標準インクルード・ファイルを検索します。「インクルード・ディレクトリを指定する (-I)」で説明するオプションを使用して、設定ファイル icl.cfg でインクルード・ファイルの場所を指定することもできます。

インクルード・ディレクトリを指定する (-I)

-Idirectory オプションを使用することによって、インクルード・ファイルを検索するディレクトリを追加できます。インクルードされるファイルは、#include プリプロセッサ・ディレクティブによってプログラムに取り込まれます。コンパイラは、次の順序でディレクトリからインクルード・ファイルを検索します。

- include を含むソース・ファイルのディレクトリ
- -I オプションで指定されたディレクトリ
- 環境変数 INCLUDE で指定されたディレクトリ

インクルード・ディレクトリを除外する (-X)

コンパイラが環境変数 INCLUDE で指定されたデフォルトのパスを検索しないようにするには、-X オプションを使用します。

-X オプションと -I オプションを使用することによって、コンパイラがデフォルトのパスを検索しないようにし、別のパスでインクルード・ファイルを検索するよう指定できます。例えば、コンパイラに対して、デフォルトのパスではなく ¥alt¥include を検索するよう指示するには、次のようにします。

```
prompt> icl -X -I ¥alt¥include newmain.cpp
```

代替ツールと代替パスの指定方法

Intel のコンパイラでは、デフォルトのパスやツール以外にも、前処理、コンパイル、アセンブリ、リンクを行うための代替ツールを指定できます。さらに、コマンド・ラインから、代替ツール専用のオプションを呼び出すこともできます。この機能を利用するには、-Qlocation および -Qoption オプションを指定します。

代替コンポーネントを指定する (-Qlocation,*tool*,*path*)

ツールの代替パスを指定するには、-Qlocation オプションを使用します。このオプションは、次の構文によって 2 つの引数を受け取ります。

```
prompt> -Qlocation,tool,path
```

tool　1 つまたは複数の引数を受け取るコンパイル・ツールを表す、以下のいずれかの文字が入ります。

- cpp　コンパイラのフロントエンド・プリプロセッサを指定します。
- c　C/C++ コンパイラを指定します。
- asm　アセンブラを指定します。
- link　リンカを指定します。

他のプログラムにオプションを渡す (-Qoption, *tool*, *optlist*)

-Qoption オプションを使用して、*optlist* で指定するオプションをツール *tool* に渡します。*optlist* は、カンマで区切られたオプションの並びです。このコマンドの構文は、次のようになります。

```
Qoption,tool,optlist
```

tool　1 つまたは複数の引数を受け取るコンパイル・ツールを表す、以下のいずれかの文字が入ります。

- cpp　コンパイラのフロントエンド・プリプロセッサを指定します。
- c　C/C++ コンパイラを指定します。
- asm　アセンブラを指定します。

link	リンカを指定します。
optlist	指定されたプログラムの有効な引数を 1 つまたは複数指定します。引数がコマンド・ライン・オプションの場合はハイフンも含めなければなりません。引数にスペースやタブ文字が含まれている場合は、引数全体を引用符 (" ") で囲まなければなりません。複数の引数はカンマで区切ります。

次の例は、コンパイラがソースから実行可能ファイルを生成する際に、リンカに対してメモリ・マップを作成するよう指示します。

```
prompt> icl -Qoption,link,-map:proto.map proto.cpp
```

上記の例の `-Qoption,link` オプションは、**`-map`** オプションをリンカに渡します。これは、コンパイル処理で引数を他のツールに渡すための明示的な指定方法です。

次のコマンドを使用することによって、暗黙に同じ結果を得ることもできます。

```
prompt> icl -Fmproto.map proto.cpp
```

リンカにオプションを渡す (-link)

オプションの渡し先をリンカに特定するには、`-link` オプションを使用します。例えば、`a.cpp` ファイルをコンパイルし、リンカに対し、`a.obj` と `libfoo.lib` をリンクさせて `a.exe` を作成するよう指示するには、コマンド・ラインに次のように入力します。

```
prompt> icl a.cpp -link libfoo.lib
```

注：*コンパイラは、`-link` オプションの後に入力されたものはすべてリンカだけに渡します。したがって、これ以外のコンパイラ・オプションを使用する場合は、すべて `-link` オプションより前で指定しなければなりません。*

最適化

4

この章では、プログラムの最適化オプションを使用して、アプリケーションのパフォーマンスを向上させる方法について説明します。このほか、最適化の関連情報として、第 5 章「プロシージャ間の最適化とプロファイルによる最適化」、およびコードを調整するための実験的な方法について説明した、付録 B「実験的なパフォーマンス調整」も参照してください。

最適化の選択肢

プログラムで必要な最適化のタイプを指定するために、以下のコマンド・ライン・オプションを使用します。

`-Od`	最適化を無効にします。
`-O1`	`-Og`、`-Oi-`、`-Os`、`-Oy`、`-Ob1`、`-Gf`、`-Gs`、および `-Gy` オプションを有効にします。しかし、`-O1` は、ライブラリ関数のインライン展開など、コード・サイズを大きくするだけで高速化にはつながらないオプションを無効にします。このオプションは Microsoft Visual C++ コンパイラの `-O1` オプションほど徹底していません。ほとんどの場合、`-O1` よりも `-O2` オプションを推奨します。`-O2` オプションはインライン展開を有効にするので、プログラムで多くの関数呼び出しを使用できるからです。
`-O2`	`-Og`、`-Oi-`、`-Ot`、`-Oy`、`-Ob1`、`-Gf`、`-Gs`、および `-Gy` オプションを有効にします。最適化をプロシージャ・レベルに制限します。最適化の全項目については、表 4-1 を参照してください。デフォルトでは、この `-O2` オプションが有効になっています。

-Od オプションを使用して最適化を制限する

-Od オプションを指定すると、最適化は無効になります。表 4-1 は、オプション欄に示されている最適化オプションを指定して最適化を実行したときに、コンパイラがプログラムに適用する最適化の種類を示しています。オプション欄が「自動」となっている項目については、-Od を指定しても無効にすることはできません。

表 4-1　最適化項目の一覧

最適化	プログラムへの影響	オプション
定数の伝搬 (constant propagation)	**定数と式の評価**	-O1 / -O2
コピーの伝搬 (copy propagation)	**定数と式の評価**	-O1 / -O2
無効なコードの除去 (dead-code elimination)	**命令のシーケンス**	-O1 / -O2
グローバル・レジスタの割り当て (global register allocation)	**レジスタの使用**	-O1 / -O2
命令のスケジュール (instruction scheduling)	**命令の順序変更**	-O1 / -O2
ループのアンロール (loop unrolling)	**命令のシーケンス**	-O1 / -O2
ループ不変コードの移動 (loop-invariant code movement)	**命令のシーケンス**	-O1 / -O2
コード選択の最適化 (optimized code selection)	**命令選択 / アドレス指定モード**	**自動**
部分的な冗長の除去 (partial redundancy elimination)	**定数と式の評価**	-O1 / -O2
ストレングスの減少／帰納的変数の簡略化 (strength reduction/induction variable simplification)	**命令選択 / シーケンス** **定数と式の評価**	-O1 / -O2
変数名の変更 (variable renaming)	**レジスタの使用**	-O1 / -O2

-Oy に対する -Od の効果とデバッグ

-Od オプションを指定すると、自動的に -Oy オプションが無効になります。-Oy は、デフォルトで、あるいは -O1 または -O2 オプションが指定されたときに有効になるオプションです。-Oy オプションを指定すると、コンパイラは edp レジスタを最適化における汎用レジスタとして使用できるようになります。ただし、ほとんどのデバッガは、edp をスタック・フレーム・ポインタとして使用されるものと認識しているため、汎用レジスタとして使用される場合には、スタックのバックトレースを作成することができません。-Oy- オプションでは、-O1 または -O2 オプションの指定時に自動的に設定される他の最適化オプションは有効のままで、edp をスタック・

フレーム・ポインタとして保持および使用できるコードを生成するよう、コンパイラに命令します。その結果、デバッガは、現状態のままスタックのバックトレースを作成できるようになります。ただし、-Oy- オプションを使用すると、使用可能な汎用レジスタの数が 1 つ減るため、コードの効率がやや低下する可能性があります。

対象とするプロセッサの指定 (-G*n*)

アプリケーションを実行するプロセッサを特定し、最高のパフォーマンスを得るには、-G*n* オプションを使用します。選択した -G*n* のサブオプションに関係なく、生成されたバイナリコードは、すべての 32 ビット Intel アーキテクチャ (IA-32) プロセッサ上で動作します。-G*n* のサブオプションは、次のとおりです。

G4 アプリケーションを主として Intel486 プロセッサ上で実行する場合は、G4 を選択します。

G5 アプリケーションを主として Pentium プロセッサ上でのみ実行する場合は、G5 を選択します。

G6 アプリケーションを主として Pentium Pro または Pentium II プロセッサ上で実行する場合は、G6 を選択します。G6 はデフォルトでオンになっています。

例えば、次のコマンドは、ソース・プログラム prog.cpp をコンパイルし、Pentium Pro プロセッサに合った最適化を行います。

```
prompt> icl prog.cpp
prompt> icl -G6 prog.cpp
```

自動プロセッサ・ディスパッチ機能のサポート (-Qx*[extensions]* 、-Qax*[extensions]*)

-Qx*[extensions]* および -Qax*[extensions]* オプションを使用して、指定したプロセッサ命令拡張子に対応する専用コードを生成することができます。

-Qx*[extensions]* 拡張子で指定されたプロセッサでのみ実行される専用コードを生成します。

-Qax*[extensions]* 指定された拡張子に対応する専用コードを生成し、さらに汎用の IA-32 コードも生成します。通常は、汎用コードの方が処理速度が遅くなります。

表 4-2 に示すように、どちらのオプションにも同じ拡張子を指定できます。

表 4-2 プロセッサ・ディスパッチ機能の拡張子のオプション

拡張子	拡張子で指定されるプロセッサと機能
i	CMOV 命令と FCMOV 命令対応の Pentium® Pro および Pentium II プロセッサ
M	MMX® テクノロジ命令対応の Pentium II プロセッサ
K	ストリーミング SIMD 拡張命令 (i と M の命令セットを含む) 対応の Pentium IIIプロセッサ

-Qx*[extensions]* を使用して専用コードを生成する

`-Qx[extensions]` オプションは、プログラムを実行するプロセッサ上で必要な最小限の拡張命令セットを指定するためのものです。生成されるコードには、指定したプロセッサの拡張命令を無条件に使用することができます。`-Qx[extensions]` オプションを使用した場合は、コンパイラによって生成されたコードを IA-32 プロセッサ上で実行することはできません。IA-32 プロセッサには指定した拡張命令がサポートされていないからです。次のコマンド例は、拡張子 `i` を使用して、`myprog.cpp` ファイル内のプログラムをコンパイルします。

```
prompt> icl -O2 -G6 -Qxi myprog.cpp
```

コンパイルされたプログラム `myprog.exe` は、Pentium プロセッサでは実行できませんが、Pentium Pro および Pentium II プロセッサでは実行可能です。

注意： *`-Qx[extensions]` オプションでコンパイルされたプログラムは、指定した拡張命令をサポートしていないプロセッサで実行すると、無効命令例外または他の予期せぬ動作によって異常終了する可能性があります。*

-Qax*[extensions]* を使用して専用コードと汎用コードを生成する

特定のプロセッサ拡張命令セットに対し、自動的にコードを特定できるようにするには、`-Qax[extensions]` オプションを使用します。コンパイラは、現在のプロセッサでサポートされている拡張命令を実行時に検出するためのコードを生成します。検出された拡張命令がプロセッサでサポートされている場合には、専用コードが実行されます。コンパイラはさらに、動作内容は同じであるが通常は処理速度が遅い別のコードを生成します。これが汎用コードです。したがって、`-Qax[extensions]` オプションでコンパイルされたプログラムはすべての IA-32 プロセッサで実行できますが、`-Qx[extensions]` オプションを指定した場合には、実行可能なプロセッサが制限されます。

次のコマンド例は、myprog.cpp ファイル内のプログラムをコンパイルし、コンパイラに専用コードを生成させて、CMOV 命令や MMX® テクノロジ命令などの Pentium II プロセッサ・アーキテクチャ用の機能を活用できるようにします。

```
prompt> icl -O2 -G6 -QaxiM myprog.cpp
```

コンパイルされたプログラム myprog.exe は、すべての IA-32 プロセッサで実行できます。

注意： *-Qax[extensions] オプションと -Qx[extensions] オプションを一緒に使用すると、-Qx[extensions] オプションで指定した拡張命令がコンパイラで無条件に使用される可能性があるため、生成されたプログラムには、正しく実行するためのプロセッサ拡張命令が必要となります。*

-Qpf*[options]* を使用してプリフェッチを行う

Pentium IIIプロセッサ上に自動的にプリフェッチ機能を組み入れるには、-Qpf オプションを使用します。このオプションには、以下に示す 3 つのサブオプションを使用できます。いずれのサブオプションも、キャッシュの性能を向上させるためのものです。次のコマンド例は、すべてのサブオプション機能を持つ 1 つのオプションとして、-Qpf を呼び出します。

```
prompt> icl -Qpf[options] a.cpp
```

一部の機能だけを有効にするには、次のサブオプションを指定します。

-Qpf_loop	インデックス変数の単純な線形関数を添え字として持つ配列参照に対し、プリフェッチ命令を自動的に挿入します。さらに、ループのアンロールによって、ループが反復されるごとにプリフェッチが 1 回ずつ実行されたり、ループ本体のサイズが起こりうるプリフェッチの遅延に十分対応しうる大きさになるように、挿入したプリフェッチ処理の効率を高めます。アンロールされるループの反復回数は、データ要素のサイズ、ループのストライド、およびループ本体のサイズによって決まります。
-Qpf_call	struct へのポインタを引数として使用して関数を呼び出す場合には、-Qpf オプションにより、struct が関数内で使用されるものと仮定され、その struct のプリフェッチが生成されます。

`-Qpf_sstore`	ストリーミング格納命令を生成できるようにします。ストリーミング格納命令を使用すると、キャッシュに余分なものが格納される可能性が低くなるため、メモリ・バスの使用率を高めることができます。

-Qunroll*n* を使用してループをアンロールする

ループのアンロール回数の上限を指定するには、`-Qunroll`*n* オプションを使用します。例えば、ループのアンロール回数を 4 回以下に設定するには、次のコマンドを入力します。

```
prompt> icl -Qunroll4 a.cpp
```

ループのアンロールを無効にするには、*n* を `0` に指定します。次の例は、ループのアンロールを無効にするコマンドを示しています。

```
prompt> icl -Qunroll0 a.cpp
```

ライブラリ関数のインライン展開 (-Oi、-Oi-)

デフォルトでは、コンパイラは標準 C、C++、および数値演算ライブラリの関数をインライン化します。

通常は、これによってプログラムの処理が高速になります。ただし、ライブラリ関数のインライン展開においては、予期せぬ結果をもたらすこともあります。インライン展開されたライブラリ関数では、`errno` 変数は設定されません。したがって、`errno` 変数の設定に依存しているコードでは、`-Oi-` オプションを使用する必要があります。また、コンパイラに提供されているライブラリ関数と同じ名前の関数を使用すると、コンパイラはその関数をライブラリ関数とみなし、その関数の呼び出しをインライン展開された呼び出しに置き換えてしまいます。そのため、プログラムに標準のライブラリ・ルーチンと同じ名前の関数が定義されている場合は、`-Oi-` オプションを使用して、そのプログラムの関数が必ず使用されるように指定しなければなりません。

例えば、次のオプションと `-Oi-` を一緒に使用すると、インライン展開を無効にすることができます。

- IEEE 754 への準拠 (`-Op`)
- ANSI 標準への準拠 (`-Za`)
- 最適化の制限 (`-Od`)
- デバッグ情報 (`-Zi`)

上記オプションの結果は、先に実行する最適化によって若干変わります。

注：*ライブラリ関数の自動インライン展開は、コンパイラがプロシージャ間の最適化で実行するインライン展開とは関係ありません。例えば、次のコマンドはライブラリ関数を展開せずにプログラム* `sum.cpp` *をコンパイルします。*

```
prompt> icl -Qip -Oi- sum.cpp
```

浮動小数点演算の精度 (-Op、-Op-、-Qprec、-Qprec_div、-Qpc、-Qlong_double)

この節で説明するオプションは、浮動小数点演算の精度に関するさまざまなレベルの最適化を行います。

-Op を使用する場合

`-Op` オプションは、宣言された精度を維持し、浮動小数点演算を ANSI および IEEE 標準にできるだけ準拠するように最適化を制限します。

ほとんどのプログラムでは、このオプションを指定するとパフォーマンスには逆効果になります。アプリケーションでこのオプションが必要かどうかがわからない場合は、このオプションを指定した場合と指定しない場合で、プログラムをコンパイルして実行し、パフォーマンスと精度に対する効果を評価してください。

このオプションを指定すると、プログラムのコンパイルに以下の影響があります。

- 浮動小数点型として宣言されたユーザ変数がレジスタに割り当てられません。
- 式がスピルされる場合、64 ビット (倍精度) ではなく、80 ビット (拡張精度) としてスピルされます。
- 浮動小数点演算の比較は IEEE 754 に準拠します（`NaN` 比較に関する動作を除く）。
- 演算はコードで指定されたとおりに実行されます。例えば、除算が逆数の乗算に変換されることはありません。
- コンパイラは関連付けを変更せずに、指定された順序で浮動小数点演算を実行します。

- コンパイラは浮動小数点値に関する定数保持による最適化を実行しません。定数保持の最適化は、1 による乗算、1 による除算、0 の加算または減算も除去します。例えば、数値に 0.0 を加算するコードはそのとおりに実行されます。浮動小数点例外も維持されるように、コンパイル時には浮動小数点演算は実行されません。
- 浮動小数点演算は ANSI C に準拠します。`float` 型および `double` 型への代入が行われると、精度は 80 ビット (拡張) から 32 ビット (`float`) または 64 ビット (`double`) に丸められます。`-Op` を指定しない場合は、精度の追加ビットは必ずしも変数が再利用される前に丸められるわけではありません。
- 関数のインライン展開を無効にする `-Oi-` オプションが使用されます。

注： `-Za`(ANSI C への厳密な準拠) オプションを選択した場合、`-Oi-` および `-Op` オプションはデフォルトで有効になります。

-Qprec を使用する場合

`-Qprec` オプションは、浮動小数点数の精度を向上させるために使用します。`-Qprec` は、`-Op` オプションに比べて無効にする最適化が少なく、パフォーマンスに対する影響は小さくなります。

-Qprec_div を使用する場合

`-Qprec_div` は、浮動小数点数の除算から乗算への最適化を無効にするために使用します。Intel C/C++ コンパイラでは、分母の逆数によって浮動小数点数除算計算を乗算に変えることができます。この変更は、浮動小数点数除算計算の結果を変える場合があります。

-Qpc を使用する場合

`-Qpc` *nn* オプションは、浮動小数点数の仮数部の精度を制御するために使用します。一部の浮動小数点アルゴリズムでは、浮動小数点数値の仮数部または小数部の精度が重要となる場合があります。例えば、除算や平方根の算出のように反復によって行われる演算では、`-Qpc` *nn* オプションを使用して精度を低めに設定すると、より高速に処理できます。*nn* を以下のいずれかの値に設定すると、仮数部が指定されたビット数に丸められます。

`-Qpc 32`	24 ビット (単精度)
`-Qpc 64`	53 ビット (倍精度)
`-Qpc 80`	64 ビット (拡張精度)

nn のデフォルト値は、倍精度を示す 64 です。

このオプションでは、完全な最適化を実行できます。このオプションを使用すると、浮動小数点数値の小数部だけが影響を受けるので、-Op オプションを使用した場合のような悪影響はありません。指数の範囲は影響を受けません。

-Qpc *nn* オプションを使用すると、コンパイラは main() 関数をコンパイルするときに浮動小数点精度の制御を変更します。-Qpc *nn* を使用するプログラムはそのエントリ・ポイントとして main() を使用し、main() を含むファイルは -Qpc *nn* を指定してコンパイルしなければなりません。

-Qpc オプションと同じ効果を得る方法

-Qpc *nn* オプションは、プログラムに main() ルーチンが含まれており、Intel コンパイラを使用してこの main() を含むファイルをコンパイルする場合にのみ、期待どおりに機能します。それ以外の場合、これらのオプションは役に立ちません。

Intel コンパイラで main() をコンパイルしないで -Qpc *nn* と同じ効果を得るには、Microsoft C ライブラリ・ルーチンの _controlfp() を使用します。_controlfp() 関数により、浮動小数点数の精度を変更できます。このルーチンを使用するには、Microsoft のヘッダ・ファイル float.h をインクルードしなければなりません。表 4-3 に、_controlfp() を使用して -Qpc *nn* と同じ効果を得る方法をまとめます。

表 4-3 -Qpc と等価な呼び出し

オプション	等価な呼び出し	その他のオプション
-Qpc 32	_controlfp(_PC_24, _MCW_PC);	**なし**
-Qpc 64	_controlfp(_PC_53, _MCW_PC);	**なし**
-Qpc 80	_controlfp(_PC_64, _MCW_PC);	**なし**

注： *よりパフォーマンスを高めるために、_controlfp() の呼び出しを、プログラムの初期化ルーチンに挿入することを推奨します。*

図 4-1 に、プログラムにおける _controlfp( ) の使用例を示します。_controlfp( ) の詳細については、Microsoft のオンライン・ヘルプを参照してください。

図 4-1 _controlfp() 関数の例

```
// init.c - contains initialization code for my application
#include <float.h>
void init_program(void){
// Set FPU precision control to use 24-bit significand
_controlfp( _PC_24, _MCW_PC );

// other initialization code follows...
}
```

-Qlong_double を使用する場合

-Qlong_double は、long double 型のサイズを 80 ビットに変更するために使用します。Microsoft コンパイラとの互換性のために、Intel のコンパイラでは、デフォルトの long double 型のサイズは double 型と同じ 64 ビットです。このオプションを使用すると、このオプションを指定せずにコンパイルしたファイルやライブラリ・ルーチンの呼び出しと互換性がなくなります。したがって、Intel では、このオプションを指定してコンパイルする場合、long double 型の変数は単一ファイルで局所的に使用することを推奨しています。

丸めオプション (-Qrcd)

Intel コンパイラは -Qrcd オプションを使用して、浮動小数点計算を必要とするコードのパフォーマンスを向上させます。最適化は、丸めモードの変更を制御することにより行われます。

システムのデフォルトの浮動小数点丸めモードは最近似値です。つまり、値は浮動小数点計算中に丸められます。しかし、C 言語では、浮動小数点数値は整数への変換時に切り捨てられる必要があります。これを行うために、コンパイラは各浮動小数点変換の前に丸めモードを切り捨てに変更し、後で元に戻さなければなりません。

-Qrcd オプションは、浮動小数点数から整数への変換において、丸めモードから切り捨てモードへの変更を無効にします。つまり、浮動小数点数から整数への変換を含む、すべての浮動小数点計算においてデフォルトの最近似値への丸めを行うことを意味します。このオプションは浮動小数点計算には影響しませんが、整数への変換が C のセマンティクスに準拠しなくなります。

プロシージャ間の最適化とプロファイルによる最適化

5

この章では、固有のプロファイルとプロシージャ間の依存関係に基づいてコードのパフォーマンスを向上させるための、次の 2 つの方法について説明します。

プロシージャ間の最適化 (IPO) — `-Qip` オプションを使用してコードを解析し、個々のソース・ファイル内のプロシージャ間で最適化を実行します。複数のソース・ファイルのプロシージャ間で最適化を行う場合は、`-Qipo` オプションを使って「複数ファイルの IPO」を実行します。

プロファイルによる最適化 (PGO) — ソース・コードとコンパイラの特殊コードから、機能が組み込まれたプログラムを作成します。この機能が組み込まれたコードが実行されるたびに、コンパイラは動的情報ファイルを生成します。次にコンパイルするときには、動的情報ファイルが 1 つのサマリ・ファイルにマージされます。このファイルのプロファイル情報を使用して、コンパイラはプログラム内で最も負荷の大きいパスの実行を最適化しようと試みます。

サイズや速度に対して厳密に行われる最適化オプションとは異なり、IPO と PGO の結果は多様に変動します。これは、それぞれのプログラムによって、プロファイルや最適化を行うタイミングが異なるからです。ここで示すガイドラインは、プログラムに IPO と PGO を実行する価値があるかどうかを見極める 1 つの基準となります。ただし、その評価を行うためには、最適化の原則とソース・コードの特性を理解しておく必要があります。

注意：*場合によっては、`-Qip` および `-Qipo` オプションを使用することにより、コンパイル時間とコード・サイズが極端に増大する可能性もあります。*

-Qip および -Qipo によるプロシージャ間の最適化 (IPO)

プロシージャ間の最適化を有効にするには、-Qip オプションを使用します。このオプションを使用すると、コンパイラはコードを解析し、次の表 5-1 の最適化の種類の中から、ユーザにとって最も有益なものを判断します。

表 5-1 プロシージャ間の最適化 (IPO) 項目の一覧

最適化	プログラムへの影響
関数のインライン展開 (inline function expansion)	**呼び出し、ジャンプ、分岐、ループ**
プロシージャ間の定数の伝播 (interprocedural constant propagation)	**引数、グローバル変数、戻り値**
レジスタ内の引数の受け渡し (passing arguments in registers)	**呼び出し、レジスタ使用率**
ループ不変コードの移動 (loop-invariant code motion)	**次の段階の最適化、ループ不変コード**
無効なコードの除去 (dead code elimination)	**コード・サイズ**
関数特性の伝播 (propagation of function characteristics)	**呼び出しの削除と呼び出しの移動**

関数のインライン展開は、プロシージャ間のオプティマイザによって実行される主な最適化の 1 つです。関数呼び出しが頻繁に行われる場合、コンパイラはその呼び出しの命令を関数そのもののコードに置き換えます。

-Qip オプションを指定すると、コンパイラは現在のソース・ファイルに定義されているプロシージャ内の呼び出しに対し、関数のインライン展開を行います。一方、-Qipo オプションで複数ファイルの IPO を指定すると、コンパイラはそれぞれのファイルに定義されているプロシージャ間での呼び出しに対し、関数のインライン展開を行います。

複数ファイルの IPO (-Qipo)

複数ファイルの IPO は、複数ファイルからなるプログラムの個々のプログラム・モジュールから、潜在的な最適化情報を取り出します。その情報を使用して、コンパイラはモジュール間で最適化を実行します。-Qipo オプションで実行した最適化の結果は、拡張子が .il の一連の情報ファイルとして格納されます。

プログラムの構築は、コンパイルとリンクの 2 つのフェーズに分かれます。複数ファイルの IPO は、コンパイルあるいはリンクのどちらを実行するのかによって、次のように実行内容が異なります。

1. *コンパイル・フェーズ*— ソース・ファイルがコンパイルされるたびに、複数ファイルの IPO によって情報ファイルが 1 つずつ作成されます。このファイルには、オブジェクト・ファイルと同じファイル名に拡張子 `.il` が付けられ、ソース・コードの中間形式のほか、最適化に使用される要約情報が格納されます。コンパイラは、複数ファイルの IPO が有効な場合には、`.obj` ファイルも作成します。
2. *リンク・フェーズ*— `-Qpio` オプションを指定すると、コンパイラはリンカの直前に呼び出されます。コンパイラは、対応する最新の `.il` ファイルが存在するすべてのモジュールに対して、複数ファイルの IPO を実行します。

複数ファイルの IPO 実行可能プログラムの作成

以下の 2 つの手順は、複数ファイルの IPO を実行する方法を示しています。

1. 次の例のように、`-Qpio` オプションを使ってモジュールをコンパイルします。

   ```
   prompt> icl -Qipo -c a.cpp b.cpp c.cpp
   ```

   `.il`、`.obj` ファイルが生成された後、コンパイル処理を停止する場合は、`-c` オプションを指定します。上のコマンド例では、以下のファイルが生成されます。

   情報ファイル： `a.il`、`b.il`、`c.il`

   オブジェクト・ファイル：`a.obj`、`b.obj`、`c.obj`

注： `.il` *ファイルのパスと名前は、オブジェクト・ファイルと一致していなければなりません。*

2. 上のコマンドの結果を用いて次のように入力し、プロシージャ間の最適化を実行します。

   ```
   prompt> icl -Fenu_ip_ofile -Qipo a.obj b.obj c.obj
   ```

   `-Fe` オプションを指定することにより、*`nu_ipo_file.exe`* に実行可能プログラムが格納されます。

   複数ファイルの IPO は、最新の `.il` ファイルが存在するモジュールだけに適用されます。最新の `.il` ファイルがないモジュールでは、オブジェクト・ファイルがリンク・フェーズに渡されます。

作業を効率良くするために、手順 1 と 2 を次のようにまとめて実行することもできます。

```
prompt> icl -Qipo -Fenu_ipofile a.cpp b.cpp c.cpp
```

次の段階の最適化として、プロファイル情報を用いた複数ファイルの IPO を実行する方法については、後述の「プロファイルによる最適化の例」を参照してください。

プロジェクトの makefile を使用して複数ファイルの IPO 実行可能プログラムを作成する

大部分のアプリケーションにおいては、makefile または同様のファイルを使用して、`link.exe` などのリンカを呼び出します。この処理は、`icl` オプションを指定してコンパイルとリンクを実行すると、自動的に行われます。したがって、`-Qipo` オプションを指定してリンク手順を変更したい場合は、Intel のリンカ・ドライバ `xilink.exe` を指定する必要があります。これには、次のようにコマンドを入力します。

```
prompt> xilink -Qipo <LINK_commandline>
```

`-Qipo` により、複数ファイルの IPO が有効になります。

`<LINK_commandline>` は、リンカのコマンド・ラインです。

「複数ファイルの IPO 実行可能プログラムの作成」の節で示した手順 2 の代わりに makefile を使用する場合は、`xilink.exe` の構文を使用します。次の例のようにコマンドを入力すると、複数ファイルの IPO 実行可能プログラムが ***`filename.exe`*** に作成されます。

```
prompt> xilink -Qipo -out:filename.exe a.obj b.obj c.obj
```

注： *`-Qipo` コマンド・ラインの結果としてリンクが行われる場合、オブジェクト・ファイルとリンカの引数が混在する順序は保持されません。リンカの引数は、`-Qipo` で最適化されたオブジェクトおよびリンカのコマンド・ラインで指定したその他のオブジェクトの後に配置されます。*

ユーザ関数のインライン展開の制御 (-Obn、-Qip_no_inlining)

コンパイラは `-Obn` で、関数のインライン展開を制御します。詳細については、表 5-2 を参照してください。

表 5-2　-Obn と -Qip_no_inlining オプションのまとめ

オプション	効果
`-Ob0`	ユーザ定義関数のインライン展開を無効にします。

表 5-2 -Ob*n* と -Qip_no_inlining オプションのまとめ (続き)

オプション	効果
-Ob1	__inline キーワードで宣言された関数のインライン展開を有効にします。また、C++ 言語に準拠したインライン展開も有効にします。-O1、-O2、あるいは -Ox が指定された場合、このオプションはデフォルトになります。
-Ob2	任意の関数のインライン展開を有効にします。しかし、どの関数をインライン展開するかはコンパイラが判断します。このオプションにより、プロシージャ間の最適化が有効となり、-Qip オプションが指定された場合と同様の効果が得られます。
-Qip_no_inlining	このオプションは、-Qip または -Ob2 オプションが指定される場合のみ使用します。この場合、-Qip または -Ob2 によるプロシージャ間の最適化の結果から生じるインライン展開を無効にします。ただし、これ以外のプロシージャ間の最適化に対しては何の影響もありません。また、ユーザ指定によるインライン展開に対して何の影響もありません。(-Ob1)

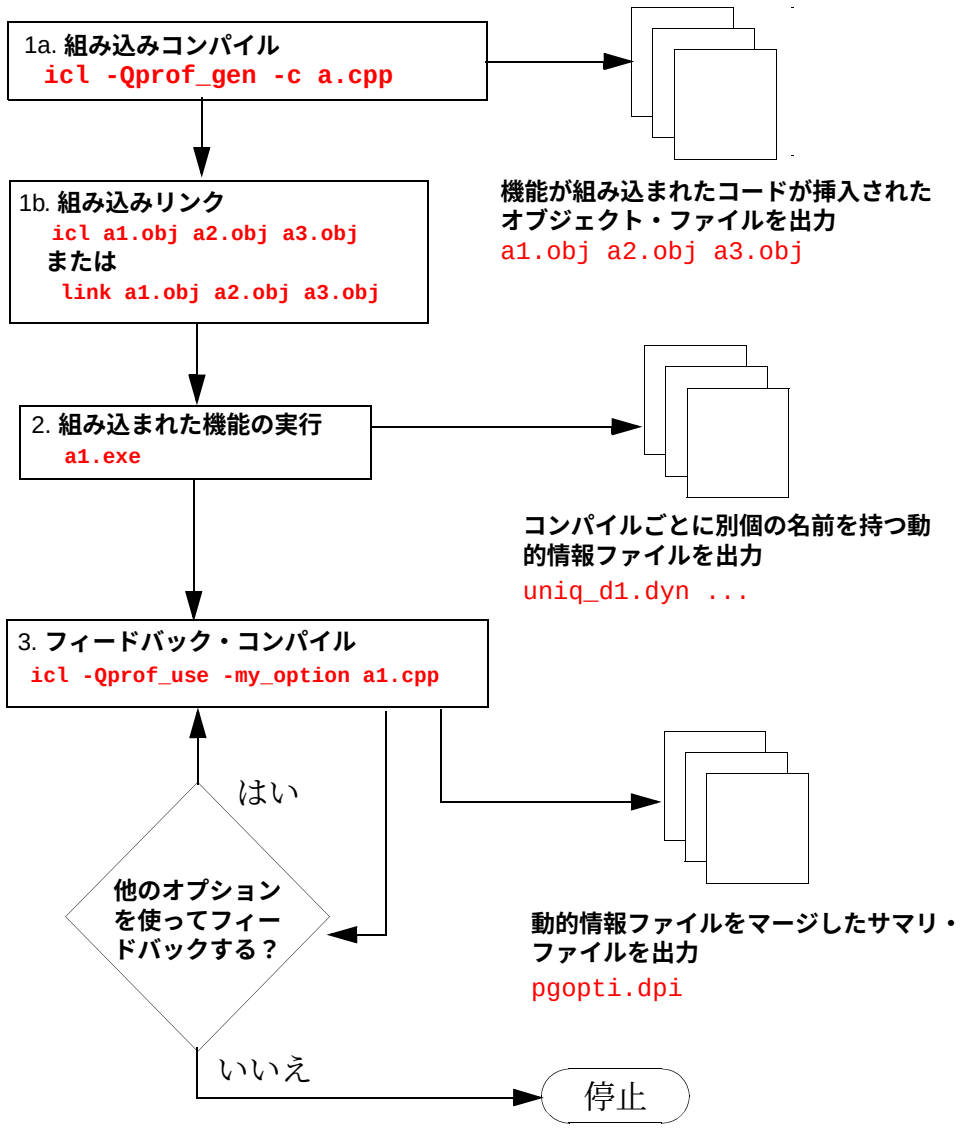
プロファイルによる最適化 (PGO)：3 つのフェーズ

プロファイルによる最適化によって、アプリケーションのどの領域が最も頻繁に実行されるかがコンパイラに通知されます。それらの領域を知ることにより、コンパイラはより的確かつ厳密にアプリケーションを最適化することができます。たとえば、PGO を使用すると、通常、コンパイラは関数のインライン化についてより的確な判断ができるようになるため、プロシージャ間の最適化の効果が上がります。PGO の方法論には、次の 3 つのフェーズが要求されます。

1. -Qprof_genオプションによる*組み込みコンパイルと組み込みリンク*
2. 実行可能プログラムの実行による*組み込まれた機能の実行*
3. -Qprof_use オプションによる*フィードバック・コンパイル*

これらのフェーズの概要については、図 5-1 を参照してください。この図には、各フェーズの説明と例も示されています。PGO を実行するかどうかの 1 つ主要な判断要素は、コード内で最も使用頻度の高いセクションを知ることです。たとえば、プログラムに使用されるデータ・セットが大変類似していて、実行のたびに同じような処理が繰り返されるような場合には、PGO によって、プログラムの動作は効果的に最適化されます。しかし、さまざまなデータ・セットを使用すると、呼び出されるアルゴリズムも多様化する可能性があります。そうなると、プログラムの動作も実行のたびに違ってくる恐れがあります。

図 5-1 プロファイルによる最適化の基本フェーズ



コードの動作が実行のたびに大幅に異なる場合は、プロファイル情報から得られる効果が、最新のプロファイルを維持するために必要となる労力に足るものであるかどうかを確認する必要があります。基本となるプロファイルによる最適化には、2 つのオプションしか使用されません。表 5-3 に示すように、フェーズ 1 で使用される -Qprof_gen オプションと、フェーズ 3 で使用される -Qprof_use オプションです。

表 5-3 プロファイルによる最適化の基本オプション

オプション	説明
-Qprof_gen	組み込まれた機能を実行する準備として、機能が組み込まれたコードをオブジェクト・ファイルに生成するようコンパイラに指示します。 注：動的情報ファイルは、フェーズ 2 で実行可能プログラムを実行した時点で生成されます。
-Qprof_use	プロファイルによって最適化された実行可能プログラムを生成し、使用できる動的情報 (.dyn) ファイルを pgopti.dpi ファイルにマージするようコンパイラに指示します。機能が組み込まれたプログラムを複数回実行すると、-Qprof_use オプションによって動的情報ファイルがそのつどマージされ、元の pgopti.dpi ファイルに上書きされます。

基本的な PGO オプションと環境変数

表 5-4 に、動的情報ファイルを保存するディレクトリの指定や、pgopti.dpi ファイルを上書きするかどうかの指定に使用する環境変数を示します。環境変数の設定方法については、ご使用のオペレーティング・システムのマニュアルを参照してください。

表 5-4 プロファイルによる最適化の環境変数

変数	説明
PROF_DIR	動的情報ファイルを作成するディレクトリを指定します。この変数は、プロファイル処理の 3 つのフェーズすべてに適用されます。
PROF_NO_CLOBBER	フィードバック・コンパイル・フェーズを若干変更します。デフォルトでは、フィードバック・コンパイル・フェーズで、コンパイラがすべての動的情報ファイルのデータをマージし、pgopti.dpi ファイルが既に存在していても、このファイルを新たに作成するように設定されています。この変数を設定すると、コンパイラは既存の pgopti.dpi ファイルを上書きせずに、警告を発行します。新しい動的情報ファイルを使用するには、既存の pgopti.dpi ファイルを削除する必要があります。

プロファイルによる最適化の例

以下に、基本的な PGO フェーズの例を示します。

1. 組み込みコンパイルと組み込みリンク — `-Qprof_gen` オプションを指定して、機能が組み込まれた情報で実行可能プログラムを生成します。

   ```
   prompt> icl -Qprof_gen -c a1.cpp a2.cpp a3.cpp
   prompt> icl a1.obj a2.obj a3.obj
   ```

   2 番目のコマンドの代わりに、リンカ (`link.exe`) を使用して、機能が組み込まれたプログラムを直接生成することもできます。その場合は、必ず `libirc.lib` ライブラリにリンクしなければなりません。
2. 組み込まれた機能の実行 — 試験的なデータ・セットを入力して機能が組み込まれたプログアムを実行し、動的情報ファイルを作成します。

   ```
   prompt> a1.exe
   ```

   作成される動的情報ファイルには、`a1.exe` を実行するたびに別個の名前と `.dyn` 拡張子が付けられます。機能が組み込まれたファイルは、プログラムが特定のデータ・セットを使ってどのように動作するのかを予測するのに役立ちます。入力データを変更して、プログラムを繰り返し回実行することもできます。
3. フィードバック・コンパイル — `-Qprof_use` オプションを指定してソース・ファイルのコンパイルとリンクを実行し、動的情報を使って、プロファイルに応じてプログラムを最適化します。

   ```
   prompt> icl -Qprof_use -Qip a1.cpp a2.cpp a3.cpp
   ```

   最適化のほかに、コンパイラは `pgopti.dpi` ファイルも生成します。一般に、フェーズ 1 ではデフォルトの最適化オプション (`-O2`) を指定し、フェーズ 3 では、さらに高度の最適化オプション (`-Qip` または `-Qipo`) を指定します。上の例では、フェーズ 1 で `-O2` オプションを指定し、フェーズ 3 で `-Qip` オプションを指定しています。

注： *`-Qip` または `-Qipo` オプションは、`-Qprof_gen` オプションと一緒に使用すると、コンパイラによって無視されます。*

PGO の使用ガイドライン

PGO を使用するときは、以下のガイドラインに従ってください。

- 組み込まれた機能が実行されてからフィードバック・コンパイルを行うまでの間は、プログラムに加える変更を最小限にします。フィードバック・コンパイルでは、情報が生成された後に変更された関数の動的情報は無視されます。

注：*コンパイラは、使用しようとしている動的情報が、変更された関数に対応していることを示す警告を発行します。*

- 組み込まれた機能が実行されてからフィードバック・コンパイルを行うまでの間にソース・ファイルに多数の変更を加える場合は、組み込みコンパイルを繰り返します。

関数順序リストを使用したプロファイルによる最適化

関数順序リストは、リンカがプログラムの非静的関数をリンクする際の順序を指定するテキスト・ファイルです。このリストにより、ページングが減少し、コードのアクセス性が高まるため、プログラムのパフォーマンスが向上します。プロファイルによる最適化においては、Microsoft Visual C++ リンカで使用される関数順序リストを生成できるようにサポートされています。コンパイラはプロファイル情報を使用して、順序を判定します。

Intel C/C++ コンパイラおよび `proforder` ツールを有効にして関数順序リストを生成するには、`-Qprof_genx` および `-Qprof_dir` オプションを使用する必要があります。これらのオプションについては、表 5-5 で説明しています。

表 5-5　プロファイルによる最適化の拡張オプション

オプション	説明
`-Qprof_genx`	**機能が組み込まれたオブジェクト・ファイルを生成して、静的プロファイル情報ファイル (`.spi`) を作成します。`-Qprof_gen`オプションの代わりに`-Qprof_genx`オプションを使用すると、`proforder` ツールでリンカ用の関数順序リストを作成できます。ただし、組み込みコンパイル時に `-Qprof_genx` を使用するには、より多くの実行時メモリが必要になります。また、作成される `.dyn` ファイルのサイズも大きくなるため、`-Qprof_genx` を使用するときは、同時に** makefile **を実行することはできません。**

表 5-5 プロファイルによる最適化の拡張オプション (続き)

オプション	説明
-Qprof_dir <dirname>	.dyn ファイルを作成するディレクトリを指定します。デフォルトは、プログラムがコンパイルされるディレクトリです。指定するディレクトリは既に存在していなければなりません。組み込みコンパイルとフィードバック・コンパイルには、どちらも同じ -Qprof_dir オプションを指定する必要があります。また、.dyn ファイルを移動する場合は、新しいパスを指定する必要があります。

この章の「プロファイルによる最適化のユーティリティ」で後述するユーティリティ profmerge および proforder を使用する必要があります。

関数順序リストの使用ガイドライン

関数順序リストを作成するには、以下のガイドラインに従ってください。

1. 順序リストは非静的関数の順序にのみ影響します。
2. -Qprof_genx オプションを使用して、同じプログラムの 2 つのファイルを同時にコンパイルするようなことはしないでください。つまり、-Qprof_genx オプションと並列的な makefile ユーティリティをともに使用することはできません。
3. 関数レベルのリンクを実行するには、-Gy オプションを指定してコンパイルする必要があります。このオプションは、-O1 または -O2 オプションを指定すると有効になります。

関数順序リストの例

file1.c および file2.c で構成される C のプログラムがあり、C:¥profdata にプロファイル・データ・ファイル用のディレクトリを作成したとします。この場合、関数順序リストを生成して使用するのは次のとおりです。

1. -Qprof_dir オプションおよび -Qprof_genx オプションを指定して、プログラムをコンパイルします。
   ```
   prompt> icl -FeMYPROG -Qprof_genx -Qprof_dir
   c:¥profdata file1.c file2.c
   ```
2. 機能が組み込まれたプログラムを 1 つまたは複数の入力データ・セットについて実行します。
   ```
   prompt> MYPROG.exe
   ```
   プログラムが実行されるたびに、.dyn ファイルが作成されます。
3. profmerge ツールを使用して、機能が組み込まれたプログラムの実行のデータをマージし、pgopti.dpi ファイルを作成します。
   ```
   prompt> profmerge -Qprof_dir c:¥profdata
   ```

4. proforder ツールを使用して関数順序リストを生成します。デフォルトの場合、関数順序リストは、proford.txt ファイルに作成されます。

```
prompt> proforder -Qprof_dir c:¥profdata
-o MYPROG.txt
```

5. -Qprof_use オプションと、リンカに対して /ORDER オプションを指定し、プロファイル・フィードバックによりアプリケーションをコンパイルします。ここでも、-Qprof_dir オプションを使用してプロファイル・ファイルの場所を指定します。

```
prompt> icl -FeMYPROG -Qprof_use -Qprof_dir
c:¥profdata file1.c file2.c -link /ORDER:@MYPROG.txt
```

プロファイルによる最適化のユーティリティ

profmerge および proforder を使用することによって、動的情報ファイルをマージし、プログラムのパフォーマンスを向上させることができます。

profmerge ユーティリティ

このツールは、動的情報ファイル (.dyn) をマージするために使用します。-Qprof_use を指定すると、コンパイラはフィードバック・コンパイル・フェーズにおいて自動的にこのツールを実行します。Microsoft Visual C++ のリンカの /ORDER オプションとともに使用する関数順序リストを生成する場合は、このツールが必要です。profmerge をコマンド・ラインで使用する方法は次のとおりです。

```
prompt> profmerge [-nologo] [-prof_dir <dir_name>]
```

これにより、カレント・ディレクトリ、または -prof_dir で指定したディレクトリ内のすべての .dyn ファイルがマージされ、サマリ・ファイル pgopti.dpi が作成されます。

proforder ユーティリティ

リンカの /ORDER オプションで使用するための関数順序リストを生成します。このツールを実行するためのコマンド・ライン構文は次のとおりです。

```
prompt> proforder [-nologo] [-prof_dir <dir_name>]
[-o <order_file>]
```

<dir_name> は、プロファイル・ファイル (.dpi、.dyn、.spi) が格納されるディレクトリです。

<order_file> は、関数順序リスト・ファイルの名前です (省略可能)。ファイル名を省略すると、デフォルトの proford.txt になります。

proforder ツールは、フィードバック・コンパイル・フェーズの一部として Microsoft Visual C++ リンカと一緒に使用され、プログラムのパフォーマンスを向上させます。

プロファイル・データを明示的にダンプする関数の呼び出し

プロファイルによる実行フェーズにおいて、プログラムはプロファイル・データを動的情報ファイル (.dyn ファイル) に書き込みます。このファイルは、プログラムが main() から正常に戻るか、C の標準関数 exit を呼び出した後で書き込まれます。プログラムが正常に終了しない場合に備えて、_PGOPTI_Prof_Dump 関数が用意されています。組み込みコンパイル (-Qprof_gen) 時に、プログラムにこの関数の呼び出しを追加できます。この呼び出しの前に次の関数プロトタイプを追加します。

```
void _cdecl _PGOPTI_Prof_Dump(void);
```

注： *フィードバック・コンパイル (-Qprof_use) の前に、この呼び出しを削除しなければなりません。また、フィードバック・コンパイルの実行時に動的情報ファイルを上書きするかどうかに応じてコメント・アウトすることにより、この呼び出しの有効 / 無効を切り替えることもできます。*

コンパイル出力の指定 6

この章では、Intel C/C++ コンパイラにより作成される出力ファイルを指定するオプションについて説明します。デフォルトでは、コンパイラはソース・コードを直接実行可能ファイルに変換します。しかし、特定のオプションによってコンパイラにより作成される出力ファイルを指定することができます。

出力ファイルを制御することにより、アセンブリ・ファイル、オブジェクト・ファイル、または実行可能ファイルなど、任意のコンパイル・フェーズのファイルを作成できることを意味します。処理中にエラーが発生しなかった場合は、特定のフェーズの出力ファイルを、後でコンパイラを起動するときの入力ファイルとして使用できます。表 6-1 に、出力ファイルを指定するためのオプションを示します。

表 6-1　コンパイラの入力ファイルおよび出力ファイルの一覧

直前のフェーズ	オプション	入力ファイル	出力ファイル
前処理	-P、-E、 **または** -EP	**ソース・ファイル**	**前処理済みのファイル (これらのオプションについての詳細は、第 7 章の「前処理のみ行う (-E、-EP、および -P)」を参照してください。)**
構文チェック	-Zs	C **または** C++ **のソース・ファイル** **前処理済みのファイル**	**診断リスト**
コンパイル	-S	**ソース・ファイル** **前処理済みのファイル**	**アセンブリ言語ファイル**

続く

表 6-1 コンパイラの入力ファイルおよび出力ファイルの一覧 (続き)

直前のフェーズ	オプション	入力ファイル	出力ファイル
アセンブリ	-c	ソース・ファイル 前処理済みのファイル アセンブリ言語ファイル	リンクされていないオブジェクト・ファイル
リンク	(デフォルト)	ソース・ファイル 前処理済みのファイル アセンブリ・ファイル オブジェクト・ファイル ライブラリ	実行可能ファイル マップ・ファイル リンク可能オブジェクト・ファイル
コンパイル、リンク、またはアセンブリ	-Fa, -Fo, -Fe	ソース、アセンブリ、またはオブジェクト・ファイル	選択した出力ファイルに名前を割り当てられる

構文解析のみ (-Zs)

`-Zs` オプションは、C/C++ 言語のエラーがないか構文解析した後、ソース・ファイルの処理を停止するために使用します。このオプションは、ソースの構文またはセマンティクスが正しいか迅速にチェックできます。コンパイラは出力ファイルを作成しません。次の例では、コンパイラは `prog1.cpp` というファイルをチェックします。

診断情報は、標準エラー出力に出力されます。

```
prompt> icl -Zs prog1.cpp
```

アセンブリ・コード・リストの作成 (-S)

`-S` オプションは、拡張子が `.asm` のアセンブリ・ファイルを生成するために使用します。ソース・ファイルを前処理し、コンパイルした後、コンパイラはアセンブリおよびリンク・フェーズを実行しません。このファイルには、各アセンブリ言語の命令に対応するソース・コードの行とカラムを示すコメントが含まれます。

生成されるファイルに名前を付けるには、`-Fa` オプションを使用します。デフォルトでは、ソース・ファイルの名前に拡張子 `.asm` の付いた名前になります。

例えば、次のコマンドは `file.asm` と `file2.asm` の 2 つのアセンブリ言語ファイルを作成します。これらのファイルは後でアセンブルおよびリンクすることができます。

```
prompt> icl -S file.cpp file2.cpp
```

.asm ファイルをアセンブルして、file.obj と file2.obj を生成します。

```
prompt> icl -c file.asm file2.asm
```

Intel C/C++ コンパイラで作成されたアセンブリ・ファイルは、MASM 6.11 あるいはそれ以降のバージョンでアセンブルできます。MASM 6.11 を使って Intel C/C++ コンパイラで作成したファイルをアセンブルする場合、追加的ないくつかのオプションが必要になることがあります。MASM を呼び出すには、以下のようなコマンド・ラインを使用します。

```
ml -c -coff -Cp file.asm
```

注意：*同じ名前と拡張子を持つ既存のファイルは上書きされます。*

以下に、アセンブリ・ファイルのリストの記述例 (一部抜粋) を示します。

```
$B1$3:      ; 100    ; preds: B1$2 B1$3
      add   eax, 4                        ; 7.5
      jne   $B1$3    ; PROB 95%           ; 8.5
                     ; LOE: eax, ebx, esi, edi, esp
```

- $B1$3 は、ファイルの最初の関数の 3 番目の基本ブロックの先頭を示します。基本ブロックは、最初の命令が実行されるとセット内の以降のすべての命令も実行されるというプロパティを持つ命令の集まりです。
- 基本ブロック・ラベルに続く ; n は、ブロックの実行回数です。この回数は、-Qprof_use オプションを使用している場合にのみ出力されます。これは、機能が組み込まれたプログラムが実行されたときの、ブロックの平均実行回数です。-Qprof_use についての詳細は、第 5 章の「プロファイルによる最適化の基本オプション」を参照してください。
- ; Preds は現在の基本ブロックの前にあるブロックのリストです。現在の基本ブロックに制御を渡すことのできるブロックです。
- 各命令の最後のセミコロン (;) の後の数字 (*x.y*) は、そのアセンブリ言語の命令に対応するソースの行番号とカラムを示します。
- ; Prob 95% はジャンプに割り当てられる可能性を示します。
- ; LOE は、現在の基本ブロックの終了時に残るレジスタのリストです。これらのレジスタには、以降の基本ブロックで使用される値が含まれています。

リンクを抑止する (-c)

-c オプションはリンクを抑止するために使用します。例えば、次のコマンドを実行すると、file.obj および file2.obj ファイルが作成されます。

```
prompt> icl -c file.cpp file2.cpp
```

注：*上記のコマンドは、これらのファイルをリンクしないため、実行可能ファイルは作成されません。*

Microsoft アセンブラによるオブジェクト・コードの生成 (-Quse_asm)

-Quse_asm オプションを指定すると、入力ソース・ファイルからアセンブリ・コードが生成され、Microsoft アセンブラ (MASM) によりオブジェクト・ファイルが生成されます。デフォルトでは、コンパイラはアセンブラを介さずに直接オブジェクト・ファイルを生成します。

例えば、次のコマンドでは、アセンブリ・コードを生成し、アセンブラを呼び出してオブジェクト・ファイルを生成します。

```
prompt> icl -c -Quse_asm file1.cpp
```

注：*MASM は、C++ 用オブジェクト・ファイル上で使用されるすべての機能をサポートするわけではありません。したがって、MASM で* icl *オプションを使用して生成されたすべてのファイルがアセンブル可能とは限りません。詳細については、MASM の解説書を参照してください。*

MASM 6.11 は、最大約 32760 行を処理します。32760 行以上の .asm ファイルをアセンブルしようとすると、致命的エラーとなってアセンブラーは中断します。また、-Quse_asm オプションを使ってコンパイルし、中間アセンブリ・ファイルが 32760 行より多い場合にも、アセンブラーは停止します。

アセンブリ・ファイルは、-S オプションで生成され、-Quse_asm オプションで暗黙的に生成およびアセンブルされます。MASM の制限およびアセンブリ・ファイルの書式のために、Intel C/C++ コンパイラで生成された .asm ファイルをアセンブルできない場合があります。以下の指定した機能はアセンブリ・ファイルの使用に関して互換性がありません。

- using __declspec(thread) シンタックスを使用したスレッド・ローカル・ストレージ
- -Zi、-Z7、または -Zd オプションの使用によるデバッグ・インフォメーションの生成
- #pragma comment およびオブジェクト・ファイルへのデフォルトのライブラリ・ディレクティブの埋め込み。次のオプションが影響を受けます。
  -LD、-LDd、-MD、-MDd、-ML、-MLd、-MT、-MTd、-V
- -Gf または -Gy オプションによる COMDAT の生成。この機能はアセンブリ・ファイルではサポートされていません。
- アセンブリ・ファイル使用時の C++ マルチファイル・プログラムの構築。これは、COMDAT の生成がサポートされていないことによるものです。

リンク

上記のフェーズの制限オプションを何も指定しなかった場合、コードにエラーがなければ、コンパイラはデフォルトで実行可能ファイルを作成します。以下の節では、出力ファイルの名前や、デバッグの準備方法を変更するために使用するオプションについて説明します。

出力ファイルに名前を付ける (-Fe、-Fo、-Fa)

ソース・ファイルのセットをコンパイルおよびリンクする際に、-Fe、-Fo、または -Fa オプションを使用すると、生成されたファイルに対して、コマンド・ラインの最初のソース・ファイルまたはオブジェクト・ファイルの名前に依存しない名前を付けることができます。単一のファイルを処理する場合は、-Fe*name*、-Fo*name*、および -Fa*name* オプションを使用することによって、それぞれ実行可能ファイル、オブジェクト・ファイル、およびアセンブリ・ファイルに任意の名前を付けることができます。オプションと *name* 引数の間にはスペースを入れないでください。これらのオプションを使用して、デフォルトのファイル拡張子 (.asm や .obj) 以外の拡張子を付けることもできます。次の例では、-Fo オプションを使用して、出力オブジェクト・ファイルにデフォルト (x.obj) 以外の名前 file.obj を付けています。-c オプションは、コンパイラにリンクを抑止するよう指示します。

```
prompt> icl -c -Fofile.obj x.cpp
```

次の例のコマンドは、2 つのソース・ファイルをコンパイルおよびリンクした結果として outfile.exe という名前の実行可能ファイルを作成します。

```
prompt> icl -Feoutfile.exe file.cpp file2.cpp
```

1 つ以上のファイルをコンパイルする場合、-Fe、-Fa、および -Fo オプションに引数 *dirname*(すなわち、ディレクトリ名) を指定できます。引数 *dirname* の最後はスラッシュ (/) または円記号 (¥) で、既存のディレクトリ名を指定しなければなりません。この場合、コンパイラはデフォルトの規則に従って、作成した実行可能ファイル、アセンブリ・ファイル、およびオブジェクト・ファイルの名前を付けますが、ファイルは *dir_name* で指定されたディレクトリに配置されます。

次の例では、obj_dir は既存のディレクトリであると仮定します。-Fe オプションにより、コンパイラは実行可能ファイル myprog.exe をカレント・ディレクトリに作成します。-Fo オプションによって、コンパイラはオブジェクト・ファイル a.obj、b.obj、および c.obj を作成し、それらを obj_dir ディレクトリに保存します。

```
prompt> icl -Femyprog.exe -Foobj_dir/ a.cpp b.cpp c.cpp
```

オプションと *dirname* 引数の間にはスペースを入れないでください。-Fe、-Fa、および -Fo の各オプションに異なる名前の引数を指定できます。コンパイラは、コンパイルがリンク・フェーズに進んでも、作成したオブジェクトを削除しません。

デバッグの準備 (-Zi、-Oy、-Oy-)

-Zi オプションは、シンボリック・デバッグをサポートするコードを生成するようコンパイラに指示します。例えば、次のように入力します。

```
prompt> icl -Zi prog1.cpp
```

コンパイラは、アセンブリ・ファイル内のデバッグ情報の生成をサポートしていません。-Zi オプションとともに -Quse_asm を指定した場合、生成されたオブジェクト・ファイルにはデバッグ情報は含まれません。-Zi オプションと -S を指定し、生成されたアセンブリ・ファイルを後でアセンブルした場合、生成されたオブジェクト・ファイルにはデバッグ情報は含まれません。-Zi オプションと -Fa を指定した場合、生成されたオブジェクト・ファイルにはデバッグ情報が含まれますが、アセンブリ・ファイルには含まれません。

また、Intel C/C++ コンパイラでは Microsoft の PDB 書式をデバッグ情報用としては使用しないため、デバッグ情報付きのオブジェクト・ファイルは、Microsoft VisualC++ コンパイラによって生成されるファイルよりも大きくなります。

-Zi オプションを指定した場合、コンパイラは -Od オプションをデフォルトとして使用します。-Zi または -Od オプションを指定すると、-Oy オプションは自動的に無効になります。

-Oy オプションは、デフォルトまたは -O1 や -O2 オプションを指定した場合に有効になり、最適化の汎用レジスタとしてコンパイラが ebp レジスタを使用できるようにします。しかし、ほとんどのデバッガは ebp がスタック・フレーム・ポインタとして使用されることを想定しており、これ以外の場合、スタック・バックトレースを作成できません。-Oy- オプションにより、コンパイラにコードを生成させ、ebp の保守と共にスタック・フレーム・ポインタとして ebp を使用します。最適化をオフにせずこれを行うことができるので、デバッガはスタック・バックトレースを出力することができます。このオプションを使用すると、使用可能な汎用レジスタの数が 1 つ少なくなり、コードの効率が若干低下する場合があります。

最適化およびシンボリック・デバッグのサポート

前項で説明したように、-Zi または -Od オプションを指定すると自動的に -Oy オプションは無効になります。コマンド・ラインで -Zi とともに、-O1、または -O2 最適化オプションを指定している場合、コンパイラでシンボリック・デバッグをサポートするコードを生成できます。しかし、次のような予期しない結果が発生する場合があります。

- -O1、または -O2 オプションと -Zi オプションを指定した場合、最適化の副作用のために、返されるデバッグ情報の一部が正確ではないことがあります。
- -O1、または -O2 オプションを指定した場合、-Oy オプションは無効になりません。したがって、デバッグ情報を生成しながら、フレーム・ポインタを保持したい場合は、コマンド・ラインで明示的に -Oy- オプションを指定して -Oy を無効にしなければなりません。

-Zi オプションと最適化オプションを使用した場合の結果を表 6-2 に示します。

表 6-2 -Zi と最適化オプションを使用した場合の結果

指定するオプション	結果
-Zi	デバッグ情報が作成され、-Od、-Oy は無効
-Zi -O2	デバッグ情報が作成され、-O2 最適化は有効
-Zi -O2 -Oy-	デバッグ情報が作成され、-O2 最適化は有効、-Oy は無効
-Zi -Qip	限定されたデバッグ情報が作成され、-Qip は有効

表 6-2 にリストされている最適化オプションについての詳細は、第 4 章「最適化」の各節を参照してください。

前処理

7

この章では、プリプロセッサの動作を指示するためにコマンド・ラインから使用できるオプションについて説明します。前処理では、マクロの置換、条件付きコンパイル、およびファイルのインクルードなどの作業が実行されます。表 7-1 に前処理オプションの一覧を示します。

表 7-1　前処理を制御するオプション

オプション	説明
-QA*name*[(*tokens*)]	シンボル名と指定された値のシーケンスを関連付けます。`#assert` 前処理ディレクティブと等価です。
-QA-, -u	すべての事前定義マクロ (先頭が `__` のものを除く) および表明を非アクティブにします。
-C	前処理されたソース出力でコメントを保持します。
-D*name*[=*value*]	マクロ名 name を定義し、その名前に指定された値 value を関連付けます。デフォルト (-D*name*) は値が 1 のマクロを定義します。
-E	ソース・モジュールを展開し、結果を標準出力に書き込むようプリプロセッサに指示します。
-EP	`#line` ディレクティブが出力されないことを除けば、-E と同じです。
-P	ソース・モジュールを展開し、結果をカレント・ディレクトリ内のファイルに保存するようプリプロセッサに指示します。
-U*name*	指定したマクロの自動的な定義を抑止します。

前処理されたソース出力内でコメントを保持する (-C)

-C オプションは、前処理されたソース出力内にコメントを保持するために使用します。

前処理のみ行う (-E、-EP、および -P)

ソース・ファイルをコンパイルせずに前処理を行うには、-E または -P オプションのいずれかを使用します。

-E オプションを指定した場合、コンパイラのプリプロセッサはソース・モジュールを展開し、結果を標準出力に書き込みます。前処理されたソースには #line ディレクティブが含まれます。このディレクティブは、コンパイラが次のパスでソース・ファイルと行番号を判断するために使用します。例えば、2 つのソース・ファイルを前処理し、結果を stdout に出力するには、次のコマンドを使用します。

```
prompt> icl -E prog1.cpp prog2.cpp
```

-P オプションを指定した場合、プリプロセッサはソース・モジュールを展開し、結果をカレント・ディレクトリのファイルに保存します。デフォルトの名前を変更する方法はありません。プリプロセッサは、各ソース・ファイルの拡張子を .i に変更した名前を使用します。例えば、次のコマンドは prog1.i と prog2.i という名前の 2 つのファイルを作成します。これらのファイルは別のコンパイルの入力として使用できます。

```
prompt> icl -P prog1.cpp prog2.cpp
```

-EP オプションは、-E または -P と組み合わせて使用できます。このオプションは、出力に #line ディレクティブを含めないようプリプロセッサに指示します。-EP のみ指定した場合は、-E -EP を指定するのと同じです。

注意： *-P オプションを使用する場合、同じ名前と拡張子を持つ既存のファイルは上書きされます。*

マクロを定義する (-QA、-QA-、-u、-D、および -U)

-QA および -D オプションは、前処理で使用される表明およびマクロ名を定義するために使用します。-U オプションは、マクロの自動定義を抑止するようプリプロセッサに指示します。

-QA オプションは、表明を作成するために使用します。このオプションは、#assert プリプロセッサ・ディレクティブと同じ機能を実行します。このオプションの形式は次のとおりです。

-QAname[(value)]

name	表明の識別子を指定します。
value	表明の値を指定します。値を指定する場合は、値を区切る括弧とともに引用符で囲まなければなりません。

例えば、値が orange、banana である識別子 fruit の表明を作成するには、次のコマンドを使用します。

```
prompt> icl -QA"fruit(orange,banana)" prog1.cpp
```

コンパイラは多くの事前定義マクロを提供します。Intel C/C++ コンパイラで使用可能な事前定義マクロのリストについては、「事前定義マクロ」を参照してください。

-QA- または -u を指定すると、2 つの下線で始まる事前定義マクロを除く、すべての事前定義マクロが抑止されます。

-D オプションはマクロを定義するために使用します。このオプションは、#define プリプロセッサ・ディレクティブと同じ機能を実行します。このオプションの形式は次のとおりです。

-Dname[=value]

name	定義するマクロの名前です。
value	*name* が置換される値を指定します。値を入力しなかった場合、*name* は 1 に設定されます。英数字以外を含む値は引用符で囲まなければなりません。

たとえば、size という名前で値が 100 のマクロを定義するには、次のコマンドを使用します。

```
prompt> icl -DSIZE=100 prog1.cpp
```

-Uname オプションは、指定された名前の自動定義を抑止するために使用します。-U オプションは、#undef プリプロセッサ・ディレクティブと同じ機能を実行します。

プリプロセッサ・ディレクティブについての詳細は、『C: A Reference Manual』などの言語リファレンスを参照してください。

事前定義マクロ

Intel C/C++ コンパイラで使用可能な事前定義マクロを表 7-2 に示します。**デフォルト**の欄には、そのマクロがデフォルトで有効 (オン) か無効 (オフ) のいずれであるかが示されています。**無効化**の欄には、このマクロを無効にするオプションがリストされています。「なし」の場合、そのマクロを無効にすることはできません。

表 7-2 事前定義マクロ

マクロ名	デフォルト	無効化	説明 / 使用する場合
__ICL=n	n=400	なし	Win32 システム バージョン 3.0(n=300) 用の Intel C.C++ コンパイラを識別します。
__cplusplus	C++ のみ	なし	C++ ソースをコンパイルするために定義されます。
_WIN32	オン	-u	Win32 アプリケーション用に定義されます。
_MSC_EXTENSIONS	オン	-u	Microsoft Visual C++ 言語の拡張機能を指定します。
_MSC_VER=n	n=1100	-u	Microsoft Visual C++ との互換性のために定義されます。icn.cfg ファイルにあるオプションを使ってインストール・プログラムにより設定されます。n の値は、インストールされている Visual C++ のバージョンによります。 バージョン n= 4.0 1000 4.1 1010 4.2 1020 5.0 1100 6.0 1200

続く

表 7-2 事前定義マクロ (続き)

マクロ名	デフォルト	無効化	説明 / 使用する場合
_M_IX86=*n*	オン , *n*=600	-u	指定したプロセッサ・オプションに基づいて定義されます。 -GBまたは-G3オプションを指定した場合は *n*=300 -G4 オプションを指定した場合は *n*=400 -G5 オプションを指定した場合は *n*=500 -G6 オプションを指定した場合は *n*=600
_DLL	オフ	-u	-MD オプションを指定した場合に定義されます。
_MT	オフ	-u	-MD、-MT、または -LD オプションを指定した場合に定義されます。
_CHAR_UNSIGNED	オフ	-u	-J オプションを指定した場合に定義されます。
_CPPRTTI	オフ	-u	C++ 専用に -GR オプションを指定した場合に定義されます。
_CPPUNWIND	オフ	-u	C++ 専用に -GX オプションを指定した場合に定義されます。

ANSI 標準への準拠で必要な事前定義マクロのリストについては、第 8 章の「標準に準拠の事前定義マクロ」を参照してください。

インクルード・ファイルの依存関係を出力する (-QH)

-QH オプションは、#include ディレクティブでソースにコンパイルされる各ファイルのパス名を出力します。パス名は絶対パスの場合と相対パスの場合があります。コンパイラは依存するファイルを標準出力に表示し、各インクルード・ファイルのパスを個別の行に出力します。前処理が完了すると、コンパイル処理は停止します。

次の例では、ソース・ファイル dtest.cpp は 3 つのファイルをインクルードします。依存するファイルを表示するには、次のように入力します。

```
prompt> icl -QH dtest.cpp
./d1.h
./d2.h
./d3.h
```

makefile の依存関係を出力する (-QM)

-QM オプションは、コンパイルにおいて、各ソース・ファイルの makefile の依存関係の行をリストするために使用します。コンパイラは、各ソース・ファイルの #include 行に基づいて依存関係の行を表示します。例えば、インクルード・ファイルが 1 つだけの単純なプログラムの場合、出力は次のようになります。

```
hello.obj: hello.c
hello.obj: c:/Msdev/Include/stdio.h
```

C/C++ 言語機能

8

この章では、Intel C/C++ コンパイラの C および C++ 言語処理系について説明します。この章では、C または C++ でのプログラミング方法については説明しません。この章で説明するトピックは次のとおりです。

- C の ANSI/ISO 標準への準拠
- C++ 言語をサポートするオプション

C の標準への準拠

Intel C/C++ コンパイラは、厳密な ANSI 準拠および拡張という、2 つの C 言語の方言に対応しています。アプリケーションに最適なスタイルを選択できます。

このコンパイラの C の処理系は、C 言語の ANSI/ISO 標準 (ISO/IEC9899:1990) に準拠しています。標準は、準拠する C コンパイラの処理系が特定の翻訳の制限について最低条件を満たしていなければならないことを規定しています。すべての状況において、このコンパイラはこれらの制限を超えています。テストされた値については、付録 A「コンパイラの制限」を参照してください。このコンパイラは ANSI/ISO 標準の拡張子にも対応しています。

デフォルトの場合、コンパイラは拡張コードが使われていてもそれをエラー・メッセージや警告メッセージで伝えるということはしません。コンパイラが標準に従うようにするには、`-Za` オプションを使用しなければなりません。このオプションについての詳細は、この章の「厳密な ANSI 方言 (-Za)」を参照してください。

C 言語の方言

このコンパイラは、厳密な ANSI と拡張の、2 つの C 言語の方言をサポートしています。デフォルトの場合、コンパイラは拡張方言に対応します。`-Za` オプションを使用して、ANSI/ISO 標準 (ISO/IEC 9899:1990) で定義されている、派生機能を持たない言語を選択できます。

以下の節では、各方言について説明します。C 言語の詳細な説明については、「このマニュアルについて」の節で示されている『C: A Reference Manual』または『C Programming Language』を参照してください。

厳密な ANSI 方言 (-Za)

`-Za` オプションは、厳密な方言を選択するために使用します。このオプションが有効である場合、コンパイラは標準に準拠します。ANSI 標準に厳密に準拠している場合、コンパイラはソース・ファイル内に非標準コードを見つけると、それを警告メッセージやエラー・メッセージで伝えます。

拡張方言 (-Ze)

起動ラインで `-Ze` オプションを入力すると、コンパイラは以下に示す点において、標準で指定されている言語に対応します。

ファイルおよびデータ記憶の拡張

- 入力ファイルにテキストが含まれていなくてもかまわないので、空のファイルをコンパイラの入力として使用できます。
- 構造体の最後のメンバーを不完全な配列型にすることができます。しかし、このメンバーを構造体の唯一のメンバーにすることはできません。唯一のメンバーにすると、構造体のサイズが 0 になります。
- ファイル内有効範囲の配列が、その要素の型として不完全な `struct` または `union` 型を持つことができます。この `struct` または `union` 型は、配列に添え字が付けられる前に (配列がコンパイルに定義されている場合はコンパイル終了までに)、完了しなければなりません。
- `enum` タグ名を定義し、後でソース・ファイルにおいて解釈処理できます。
- 初期化子の式の値が単一で、静的配列、構造体、または共用体を初期化するために使用される場合、中括弧で囲む必要はありません。標準 C では中括弧を必要とします。
- コンパイラは、`int static` の場合と同様に、記憶クラス指定子のリマーク・メッセージを、指定子のリストの最初の位置以外の場所に表示します。

ポインタの拡張

- 初期化子では、整数型が十分な大きさである場合、ポインタの定数値を整数型にキャストできます。
- 整数定数式では、整数定数をポインタ型にキャストし、整数型に戻すことができます。
- コンパイラは、明示的なキャストがなくても、ポインタの整数やその他の非互換ポインタ型への代入に対応します。
- `p` が `field` を含む `struct` または `union` を指していない場合でも、`p->field` 形式でフィールドを選択できます。変数 `p` はポインタでなければなりません。同様に、`x` が `field` を含む構造体や共用体でない場合でも `x.field` を使用できます。変数 `x` は `lvalue` でなければなりません。どちらの場合も、フィールドとしての `field` のすべての定義は、その構造体または共用体内で同じ型オフセットを持っていなければなりません。

型および構文の拡張

- ビット・フィールドに、`enum` 型、または `int` および `unsigned int` 以外の整数型を使用できます。
- `double` の同義語として `long float` を使用できます。
- ディレクティブの前処理の最後に任意のテキストを配置できます。
- 数は `pp-number` 構文ではなく、数の構文にしたがってスキャンされます。したがって、`0x123e+1` は、無効な 1 つのトークンではなく、3 つのトークンとしてスキャンされます。厳密な ANSI 準拠では、コンパイラは `pp-number` 構文を使用します。`pp-number` 構文についての詳細は、標準のテキストを参照してください。

述語の拡張

- `#assert` および `#unassert` マクロを使用して、述語名を定義およびテストできます。

拡張表記に対する警告メッセージ `-Ze` オプションを使用した場合、コンパイラは以下に示す構文およびセマンティクスの拡張表記を見つけると、その存在を警告メッセージで知らせます。この警告を無視すれば、拡張表記はそのまま使われます。

構文の拡張（警告メッセージが出ます）

- `enum` リストの最後に余分なカンマを使用できます。
- 構造体や共用体の閉じる中括弧（`}`）の前の最後のセミコロンを省略できます。
- ラベル定義の直後に右中括弧を使用できます。通常、ラベル定義の後には文が続きます。
- 空宣言 (前に何もないセミコロン) を使用できます。

セマンティクスの拡張（警告メッセージが出ます）

- `unsigned char*` および `char*` など、同じではないが相互に交換可能な型へのポインタ間で、代入およびポインタの差を使用できます。しかし、コンパイラは警告を出します。任意の種類の文字を指すポインタへの文字列定数の代入は、警告なしで使用できます。
- 明示的な型キャストを使用せずに、`>`、`>=`、`<` または `<=` 演算子を使用して、`void` へのポインタと他の型のポインタを比較することができます。標準 C では、`==` または `!=` を使用してこのような比較を行うことができ、警告は出ません。
- `asm` キーワードを使用して、インライン・アセンブリ・コードを挿入できます。厳密な ANSI C 方言では、このような挿入は無効です。
- 古いスタイルのパラメータを使用する関数のパラメータ宣言リストで、独立タグ宣言を使用できます。
- 定数についての符号付き整数演算の折り畳みにおいてオーバーフローが検出された場合、コンパイラは警告を生成します。

標準に準拠の事前定義マクロ

標準では、特定の事前定義マクロをコンパイラとともに提供することを必要としています。表 8-1 に、標準に従ってコンパイラが定義するマクロを示します。

表 8-1 標準に準拠の事前定義マクロ

マクロ	説明
_ _DATE_ _	"Mmm dd yyyy" の形式の文字列リテラルとしてのコンパイルの日付。
_ _FILE_ _	コンパイルされるファイルの名前を表す文字列リテラル。
_ _LINE_ _	10 進定数としての現在の行番号。
_ _STDC_ _	ANSI 準拠方言 (-Za) では定数 1。拡張方言では定義されない。
_ _TIME_ _	"hh:mm:ss" の形式の文字列リテラルとしてのコンパイルの時刻。

コンパイラは、標準で必要とされている事前定義マクロに加えて、いくつかの事前定義マクロを提供しています。これらのマクロのリストについては、このマニュアルの第 7 章の「事前定義マクロ」と、Microsoft の『Visual C++ User's Guide』のプリプロセッサ・カテゴリの項を参照してください。

C++ 標準への準拠

Intel C/C++ コンパイラは、Microsoft の C++ 言語処理系に準拠しています。Microsoft の C++ 言語処理系についての詳細は、Microsoft Vusual C++ のマニュアルを参照してください。

Microsoft との互換性 9

Intel C/C++ コンパイラは、Microsoft Visual C++ の C および C++ 言語拡張機能をサポートしています。この章では、Intel C/C++ コンパイラが Microsoft Visual C++ の拡張機能を解釈する方法の相違点について説明します。以下の点が異なります。

- コンパイラのプラグマ
- Microsoft 互換性オプション (`-Qms`)
- Microsoft バージョン互換性オプション (`-Qvcn`)
- サポートされていないコンパイラ・オプション
- PCH サポートの相違
- コンパイルおよび実行の相違
- 左シフト演算の評価
- フレンド注入の使用
- 名前空間に定義された関数のスコープでの関数宣言
- enum ビットフィールドの符号の有無

コンパイラのプラグマ

Intel C/C++ コンパイラは Microsoft Visual C++ のプラグマをサポートしていますが、以下の制限があります。

- プラグマ `pragma optimize` では、特定の最適化を有効または無効にするリストを使用できます。Intel のコンパイラはこのリストを無視します。このプラグマは、コマンド・ラインのオプションにより指定されているすべての最適化を有効または無効にします。

- 以下のプラグマを使用してもエラーは出ませんが、効果はありません。

```
pragma auto_inlinepragma component
pragma functionpragma include_alias
pragma inline_depthpragma inline_recursion
pragma intrinsicpragma setlocale
pragma warning
```

プラグマについての詳細は、Microsoft Visual C++ に付属のオンライン・ヘルプまたはマニュアルを参照してください。

Microsoft 互換性オプション (-Qms)

まれに、Microsoft のコンパイラがエラーを出さずにソース・コードをコンパイルできても、Intel のコンパイラで同じソース・コードをコンパイルするとエラーが生成されることがあります。サードパーティから配布されたファイル (インクルード・ファイルや C++ クラス・ライブラリ・ファイルなど) でこのようなエラーが発生した場合は、ソース・ファイル内のエラーを訂正する規則的な方法はありません。このような場合は、`-Qms` オプションを指定すると、Intel のコンパイラで正常にコンパイルできることがあります。

Microsoft バージョン互換性オプション (-Qvcn)

Intel C/C++ コンパイラを Microsoft Visual C++ バージョン 4.2 またはそれ以上のバージョンといっしょに使用できます。言語機能には、Visual C++ バージョン 5.0 以降でしか使用できないものと、バージョン 6.0 でしか使用できないものがあります。-Qvcn オプションは、使用中の Visual C++ のバージョンをコンパイラに通知します。表 9-1 には、各バージョンに有効な `-Qvcn` オプションを記載しています。

表 9-1 Microsoft バージョン互換性オプション

オプション	互換性対象
`-Qvc4`	Visual C++ バージョン 4.x
`-Qvc5`	Visual C++ バージョン 5.0
`-Qvc6`	Visual C++ バージョン 6.0

注：*Intel C/C++ コンパイラのインストール・プログラムでは、必要な場合、お使いの Visual C/C++ のバージョンに対応した* `-Qvcn` *オプションを、自動的に Intel C/C++ コンパイラの構成ファイル (*`icl.cfg`*) に追加します。したがって、このオプションを使用する必要は通常ありません。*

未サポートのコンパイラ・オプション

Intel C/C++ コンパイラは、Intel 固有のオプションとともに、Microsoft Visual C++ コンパイラと同じオプションをほとんどサポートしています。しかし、Microsoft Visual C++ コンパイラのオプションの一部は、Intel C/C++ コンパイラではサポートされていません。サポートされていないオプションのほとんどは、開発には役に立ちますが、動作するアプリケーションを構築するには必要ないものです。サポートされていないオプションを表 9-2 に示します。

表 9-2 未サポートの Microsoft Visual C++ コンパイラ・オプション

オプション	説明
`-Fd`	指定されたソース・ファイルのデバッグ情報として使用される PDB ファイルの名前。
`-GD`	Windows DLL 用に最適化します。
`-Gi`	インクリメンタル・コンパイルを有効にします。
`-GI`	編集を行いデバッグを続行します (`-Zi` オプションと同じ効果)。
`-Gm`	最小の再構築を有効にします。
`-Oa`	別名がないものと仮定します。
`-Ow`	関数間の別名がないものと仮定します。
`-WX`	警告をエラーとして処理します。
`-Yd`	各オブジェクトにデバッグ情報を配置します (Microsoft PCH 固有のオプション)。
`-Zg`	関数プロトタイプを生成します。
`-Zm`	コンパイラのメモリ割り当ての制限を設定します。

Intel C/C++ コンパイラは、これらの多くのオプションについてサポートされていないというリマークを出しますが、`-Fd`、`-Gi`、`-Gm`、`-Yd`、および `-Zm` については、リマークを出さずに無視します。

PCH サポートの相違

Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラとでは、プリコンパイルされたヘッダ (PCH) ファイルのサポートに相違があります。次のような点が異なります。

- Intel C/C++ コンパイラによって生成された PCH 情報は、Microsoft Visual C++ コンパイラにより生成された PCH 情報と互換性がありません。
- Intel C/C++ コンパイラは、同一翻訳単位での PCH の生成および使用をサポートしていません。
- Intel C/C++ コンパイラは、主翻訳単位の宣言の位置を越えて PCH 情報を生成しません。Microsoft Visual C++ コンパイラは、同じ状況でこの位置を越える PCH 情報を生成します。

コンパイルおよび実行の相違

Intel C/C++ コンパイラは Microsoft Visual C++ コンパイラと互換性がありますが、正常なコンパイルを妨げる可能性のある相違点がいくつかあります。また、ソース・ファイルによっては、Intel C/C++ コンパイラで生成されたコードの動作に互換性がない場合もあります。ほとんどの場合、ソース・ファイルを修正すると、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラの両方で正常にコンパイルできます。このような相違について以下に説明します。

インライン・アセンブリのターゲット・ラベル

Intel C/C++ コンパイラでコンパイルした場合、インライン・アセンブリのターゲット・ラベルである `goto` 文には大文字と小文字の区別があります。Microsoft Visual C++ コンパイラでは、この区別はありません。たとえば、次のようなコードをコンパイルした場合、Intel C++ コンパイラではエラーとなります。

```
int func(int x)
{
    goto LAB2;
    /*   error: label  "LAB2" was  referenced
     *   but not defined
     */
    __asm lab2: mov x, 1
    return x;
}
```

しかし、Microsoft Visual C++ コンパイラでは、上記のコードが使用できます。Intel C/C++ コンパイラの暫定処理としては、インライン・アセンブリで定義したラベルを `goto` 文で参照する際に、名前と大文字、小文字をラベル定義とラベル参照で一致させる必要があります。

プリプロセッサ・マクロの展開

Intel C/C++ コンパイラは、`#include` ディレクティブ内で使用されるプリプロセッサ・マクロの展開方法が Microsoft Visual C++ コンパイラと異なります。コードが、マクロをパラメータとして、トークン連結演算子を使用する別のマクロに渡すことがあります。この場合、パラメータとして使用されるマクロは連結前に展開されません。これを次の例で示します。

```
#define D   var
#define Decl(d)   int my ## d ## ;
Decl(D)
```

どちらのコンパイラも上記のソースを前処理して、次のコードを生成します。

```
int myD;
```

`#include` ディレクティブの中で似たようなマクロが使用される場合、Intel C/C++ コンパイラは上記と同様の動作をします。しかし、Microsoft Visual C++ コンパイラの場合は、`#include` ディレクティブ全体をスキャンする前処理も行うので、結果が異なります。次の例では、`#include` ディレクティブの中で使用された場合、`D` および `F` マクロは Intel C/C++ コンパイラでは展開されません。

```
#define  D    sys
#define  F    stat.h

#define INC(d,f)   < ## d ## / ## f ## >

#include INC(D,F)
// Visual C++ コンパイラは次の様に展開します：
//# include <sys/stat.h>
// Intel C/C++ コンパイラは次の様に展開します：
// #include <D/F>
```

Intel C/C++ コンパイラは、D/F という名前のファイルを見つけられない場合、このコードに対してエラーを出します。Microsoft Visual C++ コンパイラは、D マクロおよび F マクロを展開するので、このコードを使用できます。次のようにコードを変更すると同じ結果が得られ、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラの両方で動作します。

```
#define  D    sys
#define  F    stat.h

#define CAT5(a,b,c,d,e)   a ## b ## c ## d ## e
#define INC(d,f)          CAT5(<,d,/,f,>)

#include INC(D,F)
```

左シフト演算の評価

Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラでは、左シフト演算において、右辺のオペランドかシフト数がビット表現されている、左辺のオペランドのサイズ以上である場合、評価方法が異なります。ANSI C 標準では、このような左シフト演算の動作は未定義であり、プログラムでこのような演算から特定の動作を想定してはなりません。この相違は、シフト演算の両方のオペランドがともに定数である場合にのみ生じます。次の例では、Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラでの相違を示しています。

```
int x;
int y = 1;

void func()
{
 x = 1 << 32;
    // Visual C++ Compiler generates code to set x=0
    // Intel C/C++ Compiler generates code to set x=1
 y = y << 32;
    // Visual C++ Compiler generates code to set x=1
    // Intel C/C++ Compiler generates code to set x=1
}
```

フレンド注入の使用：推奨されません

クラスが namespace のメンバーで、まだ宣言されていない関数のフレンド宣言を含んでいることがあります。このような場合、Intel C/C++ コンパイラでは、たとえ、その namespace の定義にクラスの定義が語彙的に含まれていなくても、そのクラスが属する namespace に関数宣言を注入します。一方、Microsoft Visual C++ のバージョン 4.2 では、クラスの定義が語彙的に含まれているスコープだけに関数宣言を「注入」します。この両者の違いにより、別の関数にフレンド関係が生じて、アクセス・エラーが発生する場合があります。

フレンド「注入」は、C++ 標準化委員会による言語機能から最近削除されたため、コードの解釈は Microsoft Visual C++ バージョン 4.2 の特定の動作に依存しています。一方、Microsoft Visual C++ バージョン 5.0 では、この段落の最初に説明したようなソースを受け入れないため、Intel C/C++ コンパイラと同義のエラー診断メッセージを発行します。このような動作を避けるにためにも、フレンド注入を使用しないことをお勧めします。

名前空間に定義された関数のスコープでの宣言

C++ 言語の仕様に従えば、関数定義の中に関数宣言が検出されると、参照された関数は、その関数が保持されている namespace の別のメンバーとして受け取られます。これは、保持されている関数の定義が namespace の定義内に語彙的に含まれているかどうかには関係ありません。Microsoft Visual C++ コンパイラの場合は、参照された関数を大域関数 (どの namespace にも含まれない）として受け取ります。大域的なスコープまたは namespace のスコープで宣言されている関数は、Intel C/C++ コンパイラでも Microsoft Visual C++ コンパイラでも、同じように解釈されます。

enum ビットフィールドの符号の有無

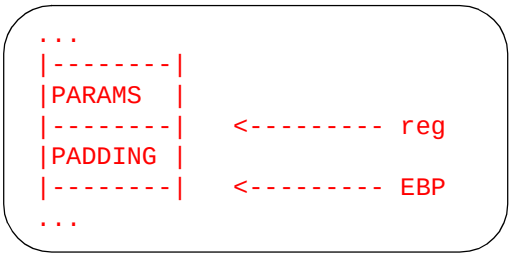
Intel C/C++ コンパイラと Microsoft Visual C++ コンパイラとでは、enum タイプで宣言されるビット・フィールドの符号の有無の判定方法が異なります。Microsoft Visual C++ では、enum タイプの値がビットフィールドで表現できないものがあったとしても、常に enum ビットフィールドを符号付きとみなします。一方、Intel C/C++ コンパイラでは、enum タイプに負の値を持つ enum 定数が 1 つも含まれていなければ、enum ビットフィールドを符号なしとみなします。いかなる場合も、Intel C/C++ コンパイラでは、ビット数が不足して enum タイプのすべての値を表現できないようなビットでビットフィールドが宣言されている場合には、警告を出します。

MSVC++ 4.2 でアライメント後のスタック・フレームを持つ関数をデバッグする

MSVC++ 4.2 のツールを使用して、Intel C/C++ コンパイラでコンパイルされたアライメント後のフレームを持つ関数をデバッグすると、Visual C++ デバッガでは無効なパラメータ値が表示されます。これは、MSVC++ 4.2 のリンカが、アライメント後のフレームのパラメータ表現に必要なデバッグ・レコードをサポートしていないためです。

通常、関数に入った時点では、EBP レジスタはパラメータ・ブロックの最下部を指しています。デバッガは、この EBP レジスタにオフセットを足した値を使用して、関数パラメータにアクセスします。しかし、アライメント後のフレームでは、フレームをアライメント処理するために、EBP レジスタが不定バイト数だけ減らされ、別のレジスタがパラメータのポインタとして使用されます。図 9-1 を参照してください。

図 9-1 スタック内の不定オフセットを簡易に表現したもの



Visual C++ バージョン 4.2 でサポートされている Microsoft のデバッグ形式では、パラメータ変数のアクセスに使用される基本レジスタには柔軟性がありません。そのため、Visual C++ バージョン 4.2 のデバッガでは、常に EBP を使ってパラメータにアクセスします。したがって、正しいアドレスにパラメータが見つからないため、パラメータに関して不適切な値が表示されることになります。

アライメント後の関数のパラメータを表示するには、アセンブリ・レベルのデバッグ処理を行って、パラメータの実際の位置を探します。Intel C/C++ コンパイラのバージョン 4.0 では、EBX レジスタを使用して、アライメント後のフレームを持つ関数のパラメータにアクセスします。このような関数では、EBX にオフセットを足した値で示されるアドレスにパラメータが存在しています。パラメータ・ブロック内には、__m128 パラメータをアライメント処理するためにパディングが生じるため、正しいオフセットを特定するには、逆アセンブルしたコードで対象とするパラメータの参照箇所を調べる方が簡単です。

Visual C++ バージョン 5.0 のツールで Intel C/C++ コンパイラを使用する場合には、この問題は影響ありません。Visual C++ バージョン 5.0 で Intel C/C++ コンパイラを使用するには、Visual C++ 5.0 をインストールしてから、Intel C/C++ コンパイラをもう一度インストールしなおす必要があります。アライメント後のスタック・フレーム機能の詳細については、第 13 章の「スタック・フレーム・アライメントのオプション」の表から、-Qsfalign オプションの説明を参照してください。

診断情報 10

この章では、コンパイラが生成するさまざまなメッセージについて説明します。これらのメッセージには、リマーク、警告、またはエラーのサインオン・メッセージや診断メッセージが含まれます。コンパイラは、常に、診断メッセージを、エラーのあるソース行とともに、標準出力します。

この章では、診断メッセージの重要度を制御する方法についても説明します。

サインオン・メッセージを無効にする (-nologo)

コンパイラは起動されるたびに次のメッセージを表示します。

```
Intel C/C++ Compiler Version x.y.z ID
Copyright (C) years Intel Corporation.  All rights reserved.
```

ID このコンパイラの固有な識別番号。

x.y.z コンパイラのバージョンを識別します。

years ソフトウェアの著作権が承認された年です。

このメッセージを抑止したい場合は、`-nologo` オプションを使用します。

icl のオプションのリストを出力する (-?、-help)

コンパイラに対して `-help` (または `-?`) を指定することによって、役に立つ `icl` のオプションのリストを表示できます。このリストを出力するには、次のコマンドを使用します。

```
prompt> icl -help
```

または

```
prompt> icl -?
```

診断メッセージ

診断メッセージはソース・テキストについての構文および意味上の情報を提供します。構文の情報には構文エラーや非標準 C または C++ の使用などが含まれます。意味上の情報には到達不能コードなどが含まれます。

診断メッセージは、コマンド・ライン診断、リマーク・メッセージ、警告メッセージ、エラー・メッセージ、または致命的なエラー・メッセージのいずれかです。

コマンド・ライン診断　このメッセージは、コマンド・ライン・オプションや引数をレポートします。コマンド・ライン・エラー・メッセージは、次の形式で標準出力に表示されます。

`icl: コマンド・ラインのエラー： メッセージ`

メッセージ　　エラーの説明です。

コマンド・ライン警告メッセージは、次の形式で表示されます

`icl: コマンド・ラインの警告： メッセージ`

言語診断　このメッセージはソース・ファイルの処理中にレポートされた診断を示します。この診断の形式は次のとおりです。

ファイル名（行番号）： タイプ [#]nn: メッセージ

ファイル名　　現在処理されているソース・ファイルの名前を示します。

行番号　　コンパイラがその状況を検出したソース行を示します。

タイプ　　診断メッセージの重要度 (警告、注意、エラー、または致命的エラー) を示します。

[#]nn　　エラー(または警告)メッセージに関連付けられている番号です。ハード・エラーや致命的なエラー・メッセージには番号は付けられません。

メッセージ　　診断の説明です。

警告メッセージの例を次に示します。

```
tantst.cpp(3): 警告 #328: ローカル変数 "increment" は使用されません。
```

コンパイラは内部エラー・メッセージについても標準エラーに表示できます。コンパイル時に内部エラーが発生した場合は、Intel のテクニカル・サポートに問い合わせてください。内部エラー・メッセージの形式は次のとおりです。

```
FATAL COMPILER ERROR: message
```

リマーク・メッセージ このメッセージは、一般的ではあるが、C または C++ の規約に従わない使用方法をレポートします。コンパイラは、`-w` オプションでレベル 4 を指定しない限り、出力または表示されません。`-w` オプションについては、この章の「警告メッセージを抑止するまたはリマークを有効にする (-w、-Wn)」を参照してください。リマークによって翻訳やリンクが停止することはありません。リマークは出力ファイルにも影響しません。代表的なリマーク・メッセージを次に示します。

```
関数が暗黙のうちに宣告されました。
型識別子は、この宣言では意味がありません。
制御式が定数です。
```

警告メッセージ このメッセージは、コンパイル中のプログラムにおいて、規約に従ってはいるが問題の発生するおそれのある言語の使用をレポートします。デフォルトの場合、コンパイラは警告を表示します。診断レベルを `0` に設定することにより、警告メッセージを抑止できます。警告によって翻訳やリンクが停止することはありません。警告は出力ファイルにも影響しません。代表的な警告メッセージを次に示します。

```
宣言は何も宣言していません。
符号なし整数とゼロの比較には意味がありません。
"==" の代わりに "=" を使用しています。
```

エラー・メッセージ このメッセージは、C または C++ の構文上または意味上の使用の誤りをレポートします。コンパイラは常にエラー・メッセージを表示します。エラーが発生すると、エラーを含むモジュールのオブジェクト・コードは作成されず、リンクも行われません。しかし、他にエラーがないかスキャンするために構文解析は続行されます。代表的なエラー・メッセージには、次のようなものがあります。

```
閉じる引用符がありません。
式は算術型を持たなければなりません。
";" が必要です。
```

致命的なエラー このメッセージは環境の問題を示します。致命的なエラー状態が発生すると、翻訳、アセンブリ、およびリンクは停止します。致命的なエラーによってコンパイルが終了した場合は、コンパイラは標準出力に終了メッセージを表示します。致命的なエラー・メッセージの代表例を次に示します。

メモリ外です。

一時的なファイル < ファイル名 > をオープンすることができませんでした。

ソースファイル < ファイル名 > をオープンすることができませんでしした。

lint のコメントで警告メッセージを抑止する

UNIX の `lint` プログラムは、バグ、移植不能、または不用と思われる C または C++ プログラムの機能を検出しようとします。コンパイラは lint 固有の 3 つのコメント、`/*ARGSUSED*/`、`/*NOTREACHED*/`、`/*VARARGS*/` を認識します。`lint` プログラムと同様に、ソースの特定のポイントにこれらのコメントを記述すると、特定の状態についての警告が抑止されます。

警告メッセージを抑止するまたはリマークを有効にする (-w、-W*n*)

`-w` または `-Wn` オプションは、前処理およびコンパイルにおいて、警告メッセージを抑止したり、リマークを有効にするために使用します。このオプションは、以下の引数のいずれかを付けて入力できます。

`-W0,-w`	エラー・メッセージだけを表示します。
`-W1,-W2,-W3`	警告およびエラー・メッセージを表示します。コンパイラはデフォルトでこのレベルを使用します。
`-W4`	リマーク、警告、およびエラー・メッセージを表示します。

コンパイルにおいて、K&R C の構成要素など、コード内の既知の良性の特性については、警告を抑止できます。例えば、次のコマンドは、`newprog.cpp` をコンパイルし、コンパイラのエラーを表示しますが、警告は表示しません。

```
prompt> icl -W0 newprog.cpp
```

診断の重要度を指定する (-Qwd、-Qwr、-Qww、-Qwe)

コンパイラによって返される診断の重要度を指定できます。コンパイラは 2 つのタイプの診断を返します。

ハード・エラー	完全に誤りであるか、疑わしいコードについて発行される診断です。ハード・エラーの重要度を設定することはできません。ハード・エラーの場合、メッセージ番号は出力されません。リマークおよび警告はハード・エラーとはみなされません。
ソフト診断	上記以外のすべての診断 (リマークおよび警告を含む) です。ソフト診断の場合、メッセージ番号は常に出力されます。ソフト診断の重要度は、以下のオプションによって設定可能です。

以下の説明で、*tag* は診断に関連付けられた番号を示します。カンマで区切って、複数のタグを指定することもできます。

`-Qwd[tag,...]` *tag* に対応するソフト診断を無効にします。

`-Qwr[tag,...]` *tag* に対応するソフト診断の重要度を無効してリマークにします。

`-Qww[tag,...]` *tag* に対応するソフト診断の重要度を無効して警告にします。

`-Qwe[tag,...]` *tag* に対応するソフト診断の重要度を無効してエラーにします。

例えば、次のコマンド・ラインは、ファイル `a.cpp` のコンパイル時にソフト診断 `68` を無効にします。

```
prompt> icl -Qwd68 -c a.cpp
```

次のコマンド・ラインは、ファイル `a.cpp` のコンパイル時に、ソフト診断 `68` および `152` の重要度をリマークに変更します。

```
prompt> icl -Qwr68,152 -c a.cpp
```

例 : 以下の行を含むファイル `x.cpp` があるとします。

```
/*
/* This is a comment
*/

extern i;
```

警告を有効にして (デフォルト) このコードをコンパイルすると、コンパイラから以下の応答があります。

```
x.cpp(2): 警告 #9: ネストされたコメントは認められません。
   /* This is a comment
   ^
x.cpp(5): 警告 #260: 明示的な型がありません ("int"  とみなします )。
   extern i;
   ^
```

オプション -Qwd9 (警告番号 9 を無効にする) を指定してこのコードをコンパイルすると、コンパイラからの応答は次のようになります。

```
x.cpp(5): 警告 #260: 明示的な型がありません ("int" とみなします )。
   extern i;
   ^
```

レポートされるエラーの数を制限する (-Qw*num*)

-Qwnum オプションは、コンパイラが異常終了するまでに表示されるエラー・メッセージの数を設定するために使用します。デフォルトの場合、100 を超えるエラーが表示されると、コンパイラは異常終了します。

-Qw*num*	コンパイルが異常終了するまでに表示されるエラー診断の数を *num* に制限します。リマークおよび警告はこの制限のカウントに含まれません。

例えば、次のコマンド・ラインは a.cpp のコンパイル時に 50 を超えるエラー・メッセージが表示されると、コンパイルが異常終了することを指定しています。

```
prompt> icl -Qwn50 -c a.cpp
```

コンパイルについてのその他の情報

コンパイル・システムは、このマニュアルで説明していないコンパイル・システムの構成要素やプロセスに対しても、さまざまな説明のメッセージを表示します。

ライブラリ 11

Intel C/C++ コンパイラでは、Microsoft Visual C++ Version 4.0 またはそれ以降のバージョンに含まれる、標準ランタイム・ライブラリをすべて使用できます。リンカを制御するか、この章で説明するオプションを使用して、アプリケーションで使用するライブラリを指定できます。

ライブラリの管理

環境変数 LIB には、Microsoft リンカがライブラリ (.lib) ファイルを検索するディレクトリがセミコロンで区切られて指定されています。リンカが他のライブラリも検索するようにしたい場合は、その名前をコマンド・ライン、応答ファイル、または icl.cfg ファイルに追加できます。どちらの場合も、追加したライブラリ名は、Intel のライブラリ (libm.lib や libm_chk.lib) 、およびドライバが常に指定する Microsoft により提供されたデフォルト・ライブラリ (oldnames.lib や libc.lib) よりも前にリンカに渡されます。応答ファイルや設定ファイル (icl.cfg) へのライブラリ名の追加については、第 3 章の「応答ファイル」および「設定ファイル」を参照してください。

コマンド・ラインでライブラリ名を指定するには、あらかじめそのライブラリのパスを環境変数 LIB に追加しておかなければなりません。次に、file.cpp をコンパイルし、ライブラリ mylib.lib にリンクするには、次のコマンドを入力します。

```
prompt> icl file.cpp mylib.lib
```

コンパイラ・ドライバ `icl.exe` は、次の順序でファイル名を Microsoft リンカに渡します。

1. オブジェクト・ファイル
2. コマンド・ライン、応答ファイル、または設定ファイルで指定されたオブジェクトやライブラリ
3. デフォルトで `libm.lib` ライブラリ (または、`-QIfdiv` オプションを指定した場合は `libm_chk.lib`)
4. Microsoft が提供するライブラリ `libc.lib` または `oldnames.lib`

デフォルトのライブラリ

Intel C/C++ コンパイラでは、Microsoft Visual C++ Version 4.x またはそれ以上のバージョンに含まれる、標準ランタイム・ライブラリをすべて使用できます。このコンパイラは、デフォルトで多くの標準 C、C++、および数値演算ライブラリ関数を自動的に展開します。詳細については、第 4 章の「ライブラリ関数のインライン展開 (-Oi、-Oi-)」を参照してください。

ライブラリ・ファイル

コンパイラは以下のサポート・ライブラリを使用するよう自動的にリンカに指示します。

`libm_chk.lib` または `libm.lib`
: `libm.lib` ファイルには数値演算ライブラリが含まれています。`libm_chk.lib` には、数値演算ライブラリと、Pentium プロセッサの特定のステップに関する浮動小数点除算ソフトウェア・パッチのサポート・ルーチンが含まれています。これらのライブラリについての詳細は、「浮動小数点除算チェックの有効化 (-QIfdiv)」を参照してください。

以下のライブラリは、Microsoft Visual C++ Version 4.0 以降で提供されます。

`libc.lib`
: 標準 C ランタイム・ライブラリ

`oldnames.lib`
: Microsoft のライブラリでは通常下線で始まる非 ANSI 関数の別名が含まれています。

プログラムを標準ライブラリ以外のライブラリや追加ライブラリにリンクしたい場合は、icl コマンド・ラインの最後に指定します。例えば、hello.cpp をコンパイルして、mylib.lib にリンクするには、次のコマンドを使用します。

```
prompt> icl -Fehello.exe hello.cpp mylib.lib
```

mylib.lib ライブラリは、LINK リンカのコマンド・ラインでは、libc.lib ライブラリよりも前に指定されます。

数値演算ライブラリ

このコンパイラ・パッケージには、標準 C ランタイム・ライブラリの数値演算関数を最適化した Intel 数値演算ライブラリ libm.lib が含まれています。このライブラリの関数は、Pentium プロセッサ上のプログラム実行速度が最適化されています。

最適化された数値演算ライブラリを使用するために、セットアップ・プロセスでは、LIB 変数で定義されたライブラリ検索パスに指定されているディレクトリのいずれかに、libm.lib を配置します。Intel では、パスで指定された最初のディレクトリに libm.lib を置くことを推奨しています。

浮動小数点除算チェックの有効化 (-QIfdiv)

-QIfdiv オプションは、Pentium プロセッサの特定のステップに存在する浮動小数点除算の欠陥に対応するソフトウェア・パッチを有効にします。このパッチによって、浮動小数点除算の精度が保証されます。

注： *-G3 または -G5 オプションのいずれかを指定すると、このオプション (-QIfdiv) はデフォルトで無効になります。*

-QIfdiv オプションが有効である場合、コンパイラは Intel が提供する特殊なライブラリ libm_chk.lib を使用して、プログラムをリンクします。libm_chk.lib ライブラリには、浮動小数点除算ソフトウェア・パッチのサポート・ルーチンと、影響を受ける数値演算ライブラリ関数が含まれます。

-QIfdiv- オプションは、他のオプションに関係なく、浮動小数点除算の欠陥に対するソフトウェア・パッチを無効にします。-QIfdiv- を指定すると、コンパイラは浮動小数点除算および影響を受ける組み込み関数について、単純なハードウェア命令を使用します。-QIfdiv- オプションを指定した場合は、コンパイラは libm.lib でリンクします。同様に、特殊な最適化された数値演算ライブラリを使用しない場合も、-QIfdiv- を指定しなければなりません。このオプションは、-G4 または -G6 オプションのいずれかを指定すると、デフォルトで指定されます。

特定の命令の不正なデコードを回避する (-QI0f)

一部の命令は 2 バイトの命令コードを持ち、最初のバイトが 0f の場合があります。まれにですが、Pentium プロセッサがこのような命令を不正にデコードする場合があります。-QI0f オプションを指定すると、このような命令の不正なデコードを回避できます。Intel C/C++ コンパイラはこのような命令が生成されるのを回避できます。

コンパイラ生成コードの制御

12

この章では、プログラムの動作に影響を与えずに、Intel コンパイラにより生成されるコードを制御するオプションおよび機能について説明します。

- 「構造体タグのアライメントを指定する (-Zp)」では、ソース・ファイルにプラグマを追加せずに、コマンド・ラインから構造体および共用体の整列を設定する方法について説明します。
- 「ゼロに初期化される変数を割り当てる (-Qnobss_init)」では、コンパイラがゼロに初期化される変数を BSS セクションに配置できるようにする代わりに、このような変数を DATA セクションに配置する方法を説明します。

構造体タグのアライメントを指定する (-Zp)

構造体や共用体のアライメント境界を指定する方法は 2 つあります。ソース・ファイルに pack プラグマを配置する方法と、コマンド・ラインでアライメント・オプションを入力する方法です。どちらの方法でも、構造体タグのアライメント境界が変更されます。

-Zp オプションは、構造体宣言のアライメント境界を指定するために使用します。一般的に、境界を小さくするとデータ・セクションが小さくなり、境界を大きくすると実行速度が高速になります。

-Zp オプションの形式は次のとおりです。

-Zp*number*

number	以下のいずれかの値で指示される境界整列の制約です。	
	1	1 バイト
	2	2 バイト
	4	4 バイト
	8	8 バイト
	16	16 バイト

例えば、ファイル prog1.e 内のすべての構造体および共用体について、2 バイトの境界整列の制約を指定するには、次のコマンドを使用します。

```
prompt> icl -Zp2 prog1.e
```

ゼロに初期化される変数を割り当てる (-Qnobss_init)

-Qnobss_init オプションは、明示的にゼロに初期化される変数を _DATA セクションに配置するために使用します。デフォルトの場合、明示的にゼロに初期化される変数は _BSS セクションに配置されますが、一部のプログラムはこれらの変数が _DATA セクションにあることを必要とします。

MMX® テクノロジとストリーミングSIMD拡張命令のサポート

13

Pentium III プロセッサ、および MMX テクノロジを搭載する Pentium プロセッサや Pentium II プロセッサには、高度なマルチメディア・アプリケーションを開発するための特性が備わっています。ストリーミング SIMD 拡張命令および MMX テクノロジの組み込み関数は、プロセッサのマルチメディア機能を利用するためのコーディング拡張命令です。この章では、これらの強力かつ高速な命令を使ったコーディング方法と、それらの効果的な使用法について説明します

MMX テクノロジの組み込み関数

MMX テクノロジの組み込み関数を使ったコーディングの利点は、C の関数呼び出しの構文と、ハードウェア・レジスタの代わりとなる C の変数を使用できることにあります。これにより、レジスタの管理やアセンブリのプログラミングが不要になります。さらに、コンパイラによって命令のスケジューリングが最適化されるため、実行可能プログラムの動作も高速になります。

組み込み関数は、新しい `__m64` データ型に基づいて、`mmx` レジスタの特定の内容を表現します。

しかし、`__m64` データ型は ANSI C の基本データ型ではありません。したがって、次のような使用上の制限があります。

- 返り値として、あるいは、パラメータとしてアサインメント (代入式) の左側だけに `__m64` データ型が使えます。また、これはその他の数値演算式 ("+" や ">>"、など) といっしょに使用することはできません。
- バイト要素や構造体へアクセスする際の共用体のように、`__m64` オブジェクトを集合体として使用します。`__m64` オブジェクトのアドレスが取り込まれます

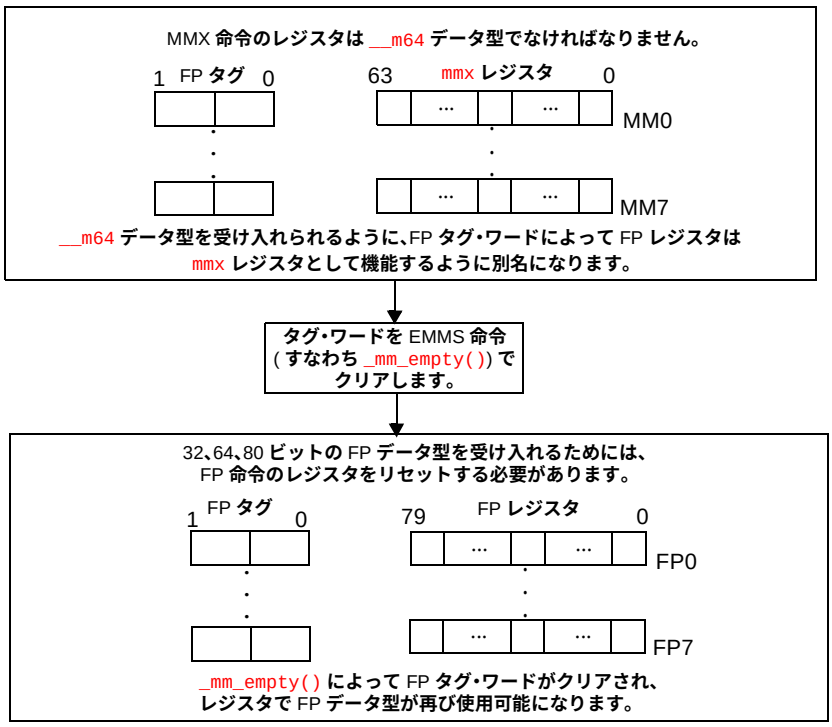
- __m64 データは、このガイドに記載されている MMX 組み込み関数とのみ使用します。

ハードウェア命令の詳細については、『インテル・アーキテクチャ MMX テクノロジ・プログラマーズ・リファレンス・マニュアル』を参照してください。データ型の詳細については、『インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル、中巻：命令セット・リファレンス・マニュアル』を参照してください。

EMMS 命令：必要性と使用法

EMMS の使用は、新しい荷物を積み込むためにコンテナを空にする作業に似ています。例えば、MMX 命令では、__m64 データ型を使用できるように自動的にレジスタ内の FP(浮動小数点) タグ・ワードを有効にします。これにより、FP レジスタ・セットはリセットされ、mmx レジスタ・セットとして別名が付けられます。FP レジスタ・セットを再び有効にするには、EMMS 命令、すなわち組み込み関数 _mm_empty() を使用して、レジスタの状態をリセットします。

図 13-1　MMX® 命令の後で EMMS のリセットが必要な理由



注意：*MMX 命令を実行した後、FP 命令のタグ・ワードをリセットしないと、プログラムの実行で予期せぬ結果が生じたり、実行パフォーマンスが低下する可能性があります。*

EMMS 命令の使用に関するガイドライン

以下のガイドラインを参考にして、EMMS をいつ使用すべきかを判断してください。

- *次の命令が FP の場合* — 次の命令が FP の場合は、MMX 命令の後に組み込み関数 `_mm_empty()` を使用します。例えば、float 型、double 型、long double 型の計算を行う前などに使用します。
- *既にレジスタが空の場合は空にしない* — 次の命令に MMX レジスタが使用される場合、`_mm_empty()` は効果のない (no-op) 演算を実行します。
- *グループ命令* — FP 命令を使用するレジスタと MMX 命令を使用するレジスタには、異なる関数を使用します。こうすると、クリティカル・ループの本体で EMMS 命令が不要になります。
- *実行時初期設定* — `__m64` データ型および FP データ型の実行時初期設定の際に `_mm_empty()` を使用します。これにより、次のデータ型に変更するまでの間に、レジスタが必ずリセットされます。コーディングの使用法については、例 13-1 を参照してください。

例 13-1 初期設定コードにおける EMMS の正しい使い方

間違った使い方	正しい使い方
`__m64 x = _m_paddd(y, z);` `float f = init();`	`__m64 x = _m_paddd(y, z);` `float f = (_mm_empty(), init());`

さらに、Intel C/C++ コンパイラでコードに MMX 命令が生成されるのはどのような場合かについて、すべて知っておく必要があります。MMX 命令が生成されるのは、次のような場合です。

- MMX 組み込み関数を使用する場合
- ストリーミング SIMD 拡張命令を (MMX データを使用している組み込み関数に) 使用する場合
- インライン・アセンブリで MMX 命令を使用する場合
- `__m64` データ型変数を参照する場合

EMMS に関する詳細な資料については、以下のウェブ・サイトを閲覧してください。

`http://www.intel.co.jp/jp/developer/index.htm` (日本語)
`http://developer.intel.com` (英語)

MMX テクノロジの組み込み関数グループ

MMX テクノロジの組み込み関数は、次のグループに分類されます。

- 汎用サポート組み込み関数
- パック化算術組み込み関数
- シフト組み込み関数
- 論理組み込み関数
- 比較組み込み関数

上記の各組み込み関数の構文は、次のように定義されます。

```
data_type intrinsic_name (parameters)                INST
```

ただし、

data_type 組み込み関数から返されるデータ型で、`void`、`int`、`__m64` のいずれかです。`void` を返す組み込み関数は `_mm_empty` だけで、`int` を返す組み込み関数は `_mm_cvtsi64_si32` だけです。他の MMX テクノロジの組み込み関数は、すべて `_m64` データ型を返します。

intrinsic_name 組み込み関数の名前です。組み込み関数は、実際の命令をインライン展開する代わりに、C/C++ コードで使用できる関数のように動作します。以降に説明する各組み込み関数の項には、組み込み関数名、代替名、およびそれらに対応する命令が一覧で示されています。表の後の説明に記されている名前には、機能をより簡単に示す新しいニーモニックも記載されています。代替名はオリジナルの組み込み関数名で、すでにこれらの関数名を使用している場合は、そのまま使用できます。

INST 組み込み関数によって使用されるアセンブリ命令の名前です。この命令はコードの構文として使用してはいけません。組み込み関数の構文のこの位置に入力して、その組み込み関数に対応付けるアセンブリ命令を指定します。

汎用サポート組み込み関数

次の表 13-1 は、汎用サポート組み込み関数の一覧です。それぞれの組み込み関数については、表の後に説明があります。

表 13-1 汎用サポート組み込み関数

組み込み関数名	代替名	対応する命令
_mm_empty	_m_empty	EMMS
_mm_cvtsi32_si64	_m_from_int	MOVD
_mm_cvtsi64_si32	_m_to_int	MOVD
_mm_packs_pi16	_m_packsswb	PACKSSWB
_mm_packs_pi32	_m_packssdw	PACKSSDW
_mm_packs_pu16	_m_packuswb	PACKUSWB
_mm_unpackhi_pi8	_m_punpckhbw	PUNPCKHBW
_mm_unpackhi_pi16	_m_punpckhwd	PUNPCKHWD
_mm_unpackhi_pi32	_m_punpckhdq	PUNPCKHDQ
_mm_unpacklo_pi8	_m_punpcklbw	PUNPCKLBW
_mm_unpacklo_pi16	_m_punpcklwd	PUNPCKLWD
_mm_unpacklo_pi32	_m_punpckldq	PUNPCKLDQ

`void _mm_empty (void)` — EMMS

MMX レジスタの状態を空にします。
「EMMS 命令：必要性と使用法」を参照してください。

`__m64 _mm_cvtsi32_si64 (int i)` — MOVD

整数オブジェクト *i* を 64 ビット __m64 オブジェクトに変換します。整数値は（32 ビット）64 ビットにゼロ拡張されます。

`int _mm_cvtsi64_si32 (__m64 m)` — MOVD

__m64 オブジェクト *m* の下位 32 ビットを整数に変換します。

`__m64 _mm_packs_pi16 (__m64 m1, __m64 m2)` — PACKSSWB

m1 内の 4 つの 16 ビット値を、符号付き飽和させて戻し値下位 4 つの 8 ビット値にパックします。また、m2 内の 4 つの 16 ビット値を、符号付き飽和させて戻し値上位 4 つの 8 ビット値にパックします。

`__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)` PACKSSDW

m1 内の 2 つの 32 ビット値を、符号付き飽和させて戻し値下位 2 つの 16 ビット値にパックします。また、m2 内の 2 つの 32 ビット値を、符号付き飽和させて戻し値上位 2 つの 16 ビット値にパックします。

`__m64 _mm_packs_pu16 (__m64 m1, __m64 m2)` PACKUSWB

m1 内の 4 つの 16 ビット値を、符号なし飽和させて戻し値下位 4 つの 8 ビット値にパックします。また、m2 内の 4 つの 16 ビット値を、符号なし飽和させて戻し値上位 4 つの 8 ビット値にパックします。

`__m64 _mm_unpackhi_pi8 (__m64 m1, __m64 m2)` PUNPCKHBW

m1 の上位半分の 4 つの 8 ビット値と m2 の上位半分の 4 つの値とを取り出し、m1 側が下位となるようにインターリーブします。

`__m64 _mm_unpackhi_pi16 (__m64 m1, __m64 m2)` PUNPCKHWD

m1 の上位半分の 2 つの 16 ビット値と *m2* の上位半分の 2 つの値とを取り出し、m1 側が下位となるようにインターリーブします。

`__m64 _mm_unpackhi_pi32 (__m64 m1, __m64 m2)` PUNPCKHDQ

m1 の上位半分の 32 ビット値と *m2* の上位半分の 32 ビット値とを取り出し、m1 側が下位となるようにインターリーブします。

`__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)` PUNPCKLBW

m1 の下位半分の 4 つの 8 ビット値と *m2* の下位半分の 4 つの値とを取り出し、*m1* 側が下位となるようにインターリーブします。

`__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)` PUNPCKLWD

m1 の下位半分の 2 つの 16 ビット値と *m2* の下位半分の 2 つの値とを取り出し、m1 側が下位となるようにインターリーブします。

`__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)` PUNPCKLDQ

m1 の下位半分の 32 ビット値と *m2* の下位半分の 32 ビット値とを取り出し、m1 側が下位となるようにインターリーブします。

パック化算術組み込み関数

次の表 13-2 は、パック化算術組み込み関数の一覧です。それぞれの組み込み関数については、表の後に説明があります。

表 13-2 パック化算術組み込み関数

組み込み関数名	代替名	対応する命令
_mm_add_pi8	_m_paddb	PADDB
_mm_add_pi16	_m_paddw	PADDW
_mm_add_pi32	_m_paddd	PADDD
_mm_adds_pi8	_m_paddsb	PADDSB
_mm_adds_pi16	_m_paddsw	PADDSW
_mm_adds_pu8	_m_paddusb	PADDUSB
_mm_adds_pu16	_m_paddusw	PADDUSW
_mm_sub_pi8	_m_psubb	PSUBB
_mm_sub_pi16	_m_psubw	PSUBW
_mm_sub_pi32	_m_psubd	PSUBD
_mm_subs_pi8	_m_psubsb	PSUBSB
_mm_subs_pi16	_m_psubsw	PSUBSW
_mm_subs_pu8	_m_psubusb	PSUBUSB
_mm_subs_pu16	_m_psubusw	PSUBUSW
_mm_madd_pi16	_m_pmaddwd	PMADDWD
_mm_mulhi_pi16	_m_pmulhw	PMULHW
_mm_mullo_pi16	_m_pmullw	PMULLW

```
__m64 _mm_add_pi8 (__m64 m1, __m64 m2)                PADDB
```

m1 の 8 つの 8 ビット値を、*m2* の 8 つの 8 ビット値に加えます。

```
__m64 _mm_add_pi16 (__m64 m1, __m64 m2)               PADDW
```

m1 の 4 つの 16 ビット値を、*m2* の 4 つの 16 ビット値に加えます。

`__m64 _mm_add_pi32 (__m64 m1, __m64 m2)` PADDD

m1 の 2 つの 32 ビット値を、*m2* の 2 つの 32 ビット値に加えます。

`__m64 _mm_adds_pi8 (__m64 m1, __m64 m2)` PADDSB

m1 の 8 つの符号付き 8 ビット値を、*m2* の 8 つの符号付き 8 ビット値に加え、飽和させます。

`__m64 _mm_adds_pi16 (__m64 m1, __m64 m2)` PADDSW

m1 の 4 つの符号付き 16 ビット値を、*m2* の 4 つの符号付き 16 ビット値に加え、飽和させます。

`__m64 _mm_adds_pu8 (__m64 m1, __m64 m2)` PADDUSB

m1 の 8 つの符号なし 8 ビット値を、*m2* の 8 つの符号なし 8 ビット値に加え、飽和させます。

`__m64 _mm_adds_pu16 (__m64 m1, __m64 m2)` PADDUSW

m1 の 4 つの符号なし 16 ビット値を、*m2* の 4 つの符号なし 16 ビット値に加え、飽和させます。

`__m64 _mm_sub_pi8 (__m64 m1, __m64 m2)` PSUBB

m1 の 8 つの 8 ビット値から、*m2* の 8 つの 8 ビット値を引きます。

`__m64 _mm_sub_pi16 (__m64 m1, __m64 m2)` PSUBW

m1 の 4 つの 16 ビット値から、*m2* の 4 つの 16 ビット値を引きます。

`__m64 _mm_sub_pi32 (__m64 m1, __m64 m2)` PSUBD

m1 の 2 つの 32 ビット値から、*m2* の 2 つの 32 ビット値を引きます。

`__m64 _mm_subs_pi8 (__m64 m1, __m64 m2)` PSUBSB

m1 の 8 つの符号付き 8 ビット値から、*m2* の 8 つの符号付き 8 ビット値を引き、飽和させます。

`__m64 _mm_subs_pi16 (__m64 m1, __m64 m2)` PSUBSW

m1 の 4 つの符号付き 16 ビット値から、*m2* の 4 つの符号付き 16 ビット値を引き、飽和させます。

`__m64 _mm_subs_pu8 (__m64 m1, __m64 m2)` PSUBUSB

m1 の 8 つの符号なし 8 ビット値から、*m2* の 8 つの符号なし 8 ビット値を引き、飽和させます。

`__m64 _mm_subs_pu16 (__m64 m1, __m64 m2)` PSUBUSW

m1 の 4 つの符号なし 16 ビット値から、*m2* の 4 つの符号なし 16 ビット値を引き、飽和させます。

`__m64 _mm_madd_pi16 (__m64 m1, __m64 m2)` PMADDWD

m1 の 4 つの 16 ビット値に *m2* の 4 つの 16 ビット値を乗じて 4 つの 32 ビットの中間結果を生成し、ペアごとに合計して 2 つの 32 ビット値を生成します。

`__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)` PMULHW

m1 の 4 つの 16 ビット値に *m2* の 4 つの 16 ビット値を乗じて、この 4 つの結果の上位 16 ビットを生成します。

`__m64 _mm_mullo_pi16 (__m64 m1, __m64 m2)` PMULLW

m1 の 4 つの 16 ビット値に *m2* の 4 つの 16 ビット値を乗じて、この 4 つの結果の下位 16 ビットを生成します。

シフト組み込み関数

次の表 13-3 は、シフト組み込み関数の一覧です。それぞれの組み込み関数については、表の後に説明があります。

表 13-3 シフト組み込み関数

組み込み関数名	代替名	対応する命令
_mm_sll_pi16	_m_psllw	PSLLW
_mm_slli_pi16	_m_psllwi	PSLLW
_mm_sll_pi32	_m_pslld	PSLLD
_mm_slli_pi32	_m_pslldi	PSLLD

表 13-3 シフト組み込み関数 (続き)

組み込み関数名	代替名	対応する命令
_mm_sll_si64	_m_psllq	PSLLQ
_mm_slli_si64	_m_psllqi	PSLLQ
_mm_sra_pi16	_m_psraw	PSRAW
_mm_srai_pi16	_m_psrawi	PSRAW
_mm_sra_pi32	_m_psrad	PSRAD
_mm_srai_pi32	_m_psradi	PSRADI
_mm_srl_pi16	_m_psrlw	PSRLW
_mm_srli_pi16	_m_psrlwi	PSRLW
_mm_srl_pi32	_m_psrld	PSRLD
_mm_srli_pi32	_m_psrldi	PSRLD
_mm_srl_si64	_m_psrlq	PSRLQ
_mm_srli_si64	_m_psrlqi	PSRLQ

`__m64 _mm_sll_pi16 (__m64 m, __m64 count)` PSLLW

m の 4 つの 16 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ左にシフトします。

`__m64 _mm_slli_pi16 (__m64 m, int count)` PSLLW

m の 4 つの 16 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ左にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_sll_pi32 (__m64 m, __m64 count)` PSLLD

m の 2 つの 32 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ左にシフトします。

`__m64 _mm_slli_pi32 (__m64 m, int count)` PSLLD

m の 2 つの 32 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ左にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_sll_si64 (__m64 m, __m64 count)` PSLLQ

m の 64 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ左にシフトします。

`__m64 _mm_slli_si64 (__m64 m, int count)` PSLLQ

m の 64 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ左にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_sra_pi16 (__m64 m, __m64 count)` PSRAW

m の 4 つの 16 ビット値を、符号ビットを詰めながら *count* で指定されたビット数だけ右にシフトします。

`__m64 _mm_srai_pi16 (__m64 m, int count)` PSRAW

m の 4 つの 16 ビット値を、符号ビットを詰めながら *count* で指定されたビット数だけ右にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_sra_pi32 (__m64 m, __m64 count)` PSRAD

m の 2 つの 32 ビット値を、符号ビットを詰めながら *count* で指定されたビット数だけ右にシフトします。

`__m64 _mm_srai_pi32 (__m64 m, int count)` PSRADI

m の 2 つの 32 ビット値を、符号ビットを詰めながら *count* で指定されたビット数だけ右にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_srl_pi16 (__m64 m, __m64 count)` PSRLW

m の 4 つの 16 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ右にシフトします。

`__m64 _mm_srli_pi16 (__m64 m, int count)` PSRLW

m の 4 つの 16 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ右にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_srl_pi32 (__m64 m, __m64 count)` PSRLD

m の 2 つの 32 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ右にシフトします。

`__m64 _mm_srli_pi32 (__m64 m, int count)` PSRLD

m の 2 つの 32 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ右にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

`__m64 _mm_srl_si64 (__m64 m, __m64 count)` PSRLQ

m の 64 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ右にシフトします。

`__m64 _mm_srli_si64 (__m64 m, int count)` PSRLQ

m の 64 ビット値を、ゼロを詰めながら *count* で指定されたビット数だけ右にシフトします。最適なパフォーマンスを得るために、*count* は定数でなければなりません。

論理組み込み関数

次の表 13-4 は、論理組み込み関数の一覧です。それぞれの組み込み関数については、表の後に説明があります。

表 13-4 論理組み込み関数

組み込み関数名	代替名	対応する命令
_mm_and_si64	_m_pand	PAND
_mm_andnot_si64	_m_pandn	PANDN
_mm_or_si64	_m_por	POR
_mm_xor_si64	_m_pxor	PXOR

`__m64 _mm_and_si64 (__m64 m1, __m64 m2)` PAND

m1 の 64 ビット値と *m2* の 64 ビット値のビット単位の論理積(AND)を実行します。

`__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)` PANDN

m1 の 64 ビット値の論理否定(NOT)を実行し、その結果を *m2* の 64 ビット値とのビット単位の論理積 (AND) で使用します。

`__m64 _mm_or_si64 (__m64 m1, __m64 m2)` POR

m1 の 64 ビット値と *m2* の 64 ビット値のビット単位の論理和(OR)を実行します。

```
__m64 _mm_xor_si64 (__m64 m1, __m64 m2)          PXOR
```

*m1*の64ビット値と*m2*の64ビット値のビット単位の排他的論理和(XOR)を実行します。

比較組み込み関数

次の表 13-5 は、比較組み込み関数の一覧です。それぞれの組み込み関数については、表の後に説明があります。

表 13-5 比較組み込み関数

組み込み関数名	代替名	対応する命令
_mm_cmpeq_pi8	_m_pcmpeqb	PCMPEQB
_mm_cmpeq_pi16	_m_pcmpeqw	PCMPEQW
_mm_cmpeq_pi32	_m_pcmpeqd	PCMPEQD
_mm_cmpgt_pi8	_m_pcmpgtb	PCMPGTB
_mm_cmpgt_pi16	_m_pcmpgtw	PCMPGTW
_mm_cmpgt_pi32	_m_pcmpgtd	PCMPGTD

```
__m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)          PCMPEQB
```

*m1*の個々の8ビット値が*m2*の個々の8ビット値に等しい場合、結果の個々の 8 ビット値をすべて 1 に設定します。それ以外の場合はすべて 0 に設定します。

```
__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)         PCMPEQW
```

*m1*の個々の16ビット値が*m2*の個々の16ビット値に等しい場合、結果の個々の 16 ビット値をすべて 1 に設定します。それ以外の場合はすべて 0 に設定します。

```
__m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)         PCMPEQD
```

*m1*の個々の32ビット値が*m2*の個々の32ビット値に等しい場合、結果の個々の 32 ビット値をすべて 1 に設定します。それ以外の場合はすべて 0 に設定します。

```
__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)          PCMPGTB
```

m1 の個々の 8 ビット値が *m2* の個々の 8 ビット値よりも大きい場合、結果の個々の 8 ビット値をすべて 1 に設定します。それ以外の場合はすべて 0 に設定します。

`__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)` PCMPGTW

m1 の個々の 16 ビット値が m2 の個々の 16 ビット値よりも大きい場合、結果の個々の 16 ビット値をすべて 1 に設定します。それ以外の場合はすべて 0 に設定します。

`__m64 _mm_cmpgt_pi32 (__m64 m1, __m64__m64 m2)` PCMPGTD

m1 の個々の 32 ビット値が *m2* の個々の 32 ビット値よりも大きい場合、結果の個々の 32 ビット値をすべて 1 に設定します。それ以外の場合はすべて 0 に設定します。

プロセッサ・ディスパッチ機能のサポート

`__declspec(cpu_specific)` および `__declspec(cpu_dispatch)` を使用すると、コンパイラが動作しているプロセッサを自動的に判断して、関数を適切にインプリメントしたコードを生成するよう、コンパイラに指示することができます。これにより、例えば、MMX 命令をサポートしているプロセッサで動作しているときには MMX 命令を利用し、MMX 命令をサポートしていない旧式のプロセッサでも正しく動作するコードを作成できるようになります。これらの拡張属性の構文は、次のようになります。

`cpu_specific(`*`cpuid`*`)`

`cpu_dispatch(`*`cpuid-list`*`)`

cupid は次のいずれかです。

```
generic
pentium
pentium_pro
pentium_mmx
pentium_ii
pentium_iii
pentium_iii_no_xmm_regs
```

cupid-list は次のいずれかです。

cpuid
cpuid-list , cpuid

cupid の名前では大文字 / 小文字を区別しません。これらの属性は、関数の定義のためだけに指定します。__declspec(cpu_dispatch) で宣言される関数の本体は空でなければならず、スタブとして参照されます。

関数 f が __declspec(cpu_specific(*p*)) として定義されている場合は、cpu_dispatch スタブもプログラムのどこかに *f* のものとして記述されていなければなりません。さらに、このとき、*p* がそのスタブの *cupid-list* に含まれていなければなりません。そうしないと、cpu_specific 定義を呼び出すことができないばかりか、この状態に対するエラーも通知されません。

関数 *f* の cpu_dispatch スタブに *cpuid p* が含まれている場合は、*cpuid p* が含まれる関数 *f* の cpu_specific 定義がプログラムのどこかに記述されていなければなりません。そうしないと、未解決の外部エラーが通知されます。cpu_specific 関数定義は、cpu_specific 関数が static で宣言されていない限り、対応する cpu_dispatch スタブと同じ翻訳ユニットに記述する必要はありません。inline 属性は、すべての cpu_specific 関数および cpu_dispatch 関数に対して無効になります。

cpu_dispatch スタブがコンパイルされると、その本体には、プログラムが実行されるプロセッサを判断してから、そのプロセッサ上で実行される (*cpu-list* の定義による) 使用可能な「最善の」cpu_specific のインプリメントにディスパッチするコードが入ります。cpu_specific 関数がコンパイルされると、コマンド・ライン・オプションの設定に関係なく、指定されたプロセッサに適した最適化とコード生成オプションが使用されます。

以下に、これらの機能の使用例を示します。

```
#include <mmintrin.h>

/* array_sum(r, a, b, l) adds two arrays of unsigned
short (a and b),of length l, and stores the result in r.
*/

__declspec(cpu_specific(Pentium))
void array_sum(unsigned short *result,
unsigned short const *a,
```

```
unsigned short const *b,
size_t length)

{
/* The implementation specific to the Pentium processor
uses no special architectural features. */

for (; length > 0; length--)
    *result++ = *a++ + *b++;
}
__declspec(cpu_specific(Pentium_MMX))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
/* The implementation for a Pentium processor with MMX
technology uses uses an MMX instruction intrinsic to add
four elements at a time, to reduce the number of loop
iterations by 3/4. */

__m64 *mmx_result = (__m64 *)result;
__m64 const *mmx_a = (__m64 const *)a;
__m64 const *mmx_b = (__m64 const *)b;
for (; length > 3; length -= 4)
    *mmx_result++ = _m_paddw(*mmx_a++, *mmx_b++);

/* If the size of all arrays passed to this routine is
known to be a multiple of four, the following code
(which takes care of excess elements) is not necessary.
*/

result = (unsigned short *)mmx_result;
a = (unsigned short const *)mmx_a;
b = (unsigned short const *)mmx_b;
for (; length > 0; length--)
    *result++ = *a++ + *b++;
```

```
}

__declspec(cpu_dispatch(Pentium, Pentium_MMX))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
/* An empty function body, which informs the compiler
that it should generate a dispatch function for the
CPU-specific implementations listed in the cpu_dispatch
clause. */
}
```

ストリーミング SIMD 拡張命令組み込み関数

この節では、Intel C/C++ コンパイラでストリーミング SIMD 拡張命令をサポートする C/C++ 言語レベルの機能を説明します。この節では、組み込み関数の以下に機能について説明します。

- 組み込み関数 API
- `__m128` データ型
- データ・アライメント・サポート
- アセンブリ言語サポート

この説明では、ストリーミング SIMD 拡張命令用の組み込み関数のすべてのリストを示しています。

組み込み関数 API

ストリーミング SIMD 拡張命令用の組み込み関数を使用して、アセンブリ言語を使用しないで新しい命令によって提供される機能を利用することができます。整数 MMX 命令用の組み込み関数とまったく同様に、新しい命令セットの計算命令およびデータ操作命令ごとに、それを直接にインプリメントする対応した C 組み込み関数があります。128 ビット・レジスタを表す新しい C データ型をそれらの組み込み関数のオペランドとして使用します。組み込み関数を使用して、アルゴリズムの基となるインプリメンテーション (命令選択) を指定し、命令スケジューリングおよびレジスタ割り当てをコンパイラに託すことができます。

__m128 データ型

__m128 データ型を使用して、4 つのパックド単精度浮動小数点値または 1 つのスカラー単精度数である xmm レジスタの内容を表現します。__m128 データ型は、基本 ANSI C データ型ではなく、そのため、その使用にはいくつかの制約があります。

- __m128 は、代入の左側に、リターン値として、またはパラメータとしてだけ使用します。"+" や ">>" など他の算術式でこのデータを使用してはなりません。
- __m128 をリテラルで初期設定してはなりません。つまり、128 ビット定数を表現する方法はありません。
- __m128 オブジェクトは、共用体 (たとえば、浮動型要素をアクセスするため) や構造体など集合体で使用します。__m128 オブジェクトのアドレスを処理することができます。
- __m128 データは、このユーザーズ・ガイドに説明されている組み込み関数だけで使用します。

コンパイラは、__m128 ローカル・データをスタック上の 16 バイト境界にアライメントを合わせます。グローバル __m128 データも 16 バイトにアラインされます。(浮動型配列のアライメントを合わせるには、次の節で説明するアライメント declspec を使用することができます。) 新しい命令セットはストリーミング SIMD 拡張命令レジスタをパックド・データまたはスカラー・データのどちらを使用するかに関係なく同じ方法で扱うので、スカラー・データを表す __m32 データ型はありません。スカラー演算では、__m128 オブジェクトおよび組み込み関数の「スカラー」形式を使用する必要があります。コンパイラおよびプロセッサは、これらの演算を 32 ビット・メモリ参照を使用して行います。

ストリーミング SIMD 拡張命令組み込み関数の規約

ストリーミング SIMD 拡張命令用の組み込み関数のリストは、以降の段落で説明するいくつかの規約を使用しています。

サフィックス ps および ss は、「パックド単精度」操作および「スカラー単精度」操作の指定に使用します。パックド浮動型は、右から左の順番に [z, y, x, w] で表され、最下位ワード (一番右) がスカラー操作に使用されます。メモリ・ストレージがこれをどのように反映するかを説明するため、次の例を考えてみます。下の操作

```
float a[4] = { 1.0, 2.0, 3.0, 4.0 };
__m128 t = _mm_load_ps(a);
```

は、次のように同じ結果を生成します。

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

言い換えると、

```
t = [ 4.0, 3.0, 2.0, 1.0 ]
```

ここで、「スカラー」要素は `1.0` です。

浮動小数点組み込み関数

以降の節は、浮動小数点組み込み関数を操作の性質でグループに分けて一覧しています。各組み込み関数のエントリは、インフォーマル疑似コードを持ち、その後には命令の名前が大文字であります。たとえば、ADDS はこの節に一覧されている最初の命令の名前です。この名前は、`__m128 _mm_add_ss(__m128 a, __m128 b)` の組み込み関数に対応します。

変数 `r` は、一般的には組み込み関数のリターン値に使用されています。変数名の後ろにある数字は、パックド・オブジェクトの要素を示しています。たとえば、`r0` は `r` の最下位ワードです。いくつかの組み込み関数は、複数の命令を必要とします。

これらの組み込み関数を使用してプログラムを作成するときは、ストリーミング SIMD 拡張命令によって提供されるハードウェア機能に精通している必要があります。留意する必要がある 3 つの重要な問題を下に示します。

- `_mm_loadr_ps` や `_mm_cmpgt_ss` など一定の組み込み関数は、命令セットによって直接にはサポートされていません。これらの組み込み関数はプログラミングする際には便利ですが、複数のマシン言語命令で構成されていることに注意してください。
- `__m128` オブジェクトとしてロードまたはストアされる浮動小数点データは、一般的には 16 バイトにアライメントが合っていなければなりません。
- 一部の組み込み関数は、命令の性質のため、それらの引数が即値、つまり、定数整数 (リテラル) であることを必要とします。

- 2 つの NaN(非数) 引数に対して行われる算術演算の結果は定義されていません。したがって、NaN 引数を使用する FP 演算は、対応するアセンブリ命令で予想される動作とは一致しません。

算術演算

`__m128 _mm_add_ss(__m128 a, __m128 b)` ADDSS

a および *b* の下位 SP FP(単精度浮動小数点) 値を加算します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := a0 + b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_add_ps(__m128 a, __m128 b)` ADDPS

a および *b* の 4 つの SP FP 値を加算します。

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

`__m128 _mm_sub_ss(__m128 a, __m128 b)` SUBSS

a および *b* の下位 SP FP 値を減算します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := a0 - b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_sub_ps(__m128 a, __m128 b)` SUBPS

a および *b* の 4 つの SP FP 値を減算します。

```
r0 := a0 - b0
r1 := a1 – b1
r2 := a2 – b2
r3 := a3 – b3
```

`__m128 _mm_mul_ss(__m128 a, __m128 b)` MULSS

a および *b* の下位 SP FP 値を乗算します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := a0 * b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_mul_ps(__m128 a, __m128 b)` MULPS

a および *b* の 4 つの SP FP 値を乗算します。

```
r0 := a0 * b0
r1 := a1 * b1
r2 := a2 * b2
r3 := a3 * b3
```

`__m128 _mm_div_ss(__m128 a, __m128 b)` DIVSS

a および *b* の下位 SP FP 値を除算します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := a0 / b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_div_ps(__m128 a, __m128 b)` DIVPS

a および *b* の 4 つの SP FP 値を除算します。

```
r0 := a0 / b0
r1 := a1 / b1
r2 := a2 / b2
r3 := a3 / b3
```

`__m128 _mm_sqrt_ss(__m128 a)` SQRTSS

a の下位 SP FP 値の平方根を計算します。上位 3 SP FP 値は、そのまま渡されます。

```
r0 := sqrt(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_sqrt_ps(__m128 a)` SQRTPS

a の 4 つの SP FP 値の平方根を計算します。

```
r0 := sqrt(a0)
```

```
r1 := sqrt(a1)
r2 := sqrt(a2)
r3 := sqrt(a3)
```

`__m128 _mm_rcp_ss(__m128 a)` RCPSS

a の下位 SP FP 値の逆数の概数を計算します。上位 3 SP FP 値は、そのまま渡されます。

```
r0 := recip(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_rcp_ps(__m128 a)` RCPPS

a の 4 つの SP FP 値の逆数の概数を計算します。

```
r0 := recip(a0)
r1 := recip(a1)
r2 := recip(a2)
r3 := recip(a3)
```

`__m128 _mm_rsqrt_ss(__m128 a)` RSQRTSS

a の下位 SP FP 値の平方根の逆数の概数を計算します。上位 3 SP FP 値は、そのまま渡されます。

```
r0 := recip(sqrt(a0))
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_rsqrt_ps(__m128 a)` RSQRTPS

a の 4 つの SP FP 値の平方根の逆数の概数を計算します。

```
r0 := recip(sqrt(a0))
r1 := recip(sqrt(a1))
r2 := recip(sqrt(a2))
r3 := recip(sqrt(a3))
```

`__m128 _mm_min_ss(__m128 a, __m128 b)` MINSS

a および *b* の下位 SP FP 値の最小を計算します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := min(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_min_ps(__m128 a, __m128 b)                MINPS
```

a および *b* の 4 つの SP FP 値の最小を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m128 _mm_max_ss(__m128 a, __m128 b)                MAXSS
```

a および *b* の下位 SP FP 値の最大を計算します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := max(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_max_ps(__m128 a, __m128 b)                MAXPS
```

a および *b* の 4 つの SP FP 値の最大を計算します。

```
r0 := max(a0, b0)
r1 := max(a1, b1)
r2 := max(a2, b2)
r3 := max(a3, b3)
```

論理演算

```
__m128 _mm_and_ps(__m128 a, __m128 b)                ANDPS
```

a および *b* の 4 つの SP FP 値のビット AND を計算します。

```
r0 := a0 & b0
r1 := a1 & b1
r2 := a2 & b2
r3 := a3 & b3
```

```
__m128 _mm_andnot_ps(__m128 a, __m128 b)             ANDNPS
```

a および *b* の 4 つの SP FP 値のビット AND NOT を計算します。

```
r0 := ~a0 & b0
r1 := ~a1 & b1
r2 := ~a2 & b2
r3 := ~a3 & b3
```

`__m128 _mm_or_ps(__m128 a, __m128 b)` ORPS

a および *b* の 4 つの SP FP 値のビット OR を計算します。

```
r0 := a0 | b0
r1 := a1 | b1
r2 := a2 | b2
r3 := a3 | b3
```

`__m128 _mm_xor_ps(__m128 a, __m128 b)` XORPS

a および *b* の 4 つの SP FP 値のビット EXOR(排他的論理和) を計算します。

```
r0 := a0 ^ b0
r1 := a1 ^ b1
r2 := a2 ^ b2
r3 := a3 ^ b3
```

比較

各比較組み込み関数は、*a* および *b* の比較を行ないます。パックド形式では、*a* および *b* の 4 つの SP FP 値が比較され、128 ビット・マスクが返されます。スカラー形式では、*a* および *b* の下位 SP FP 値が比較され、32 ビット・マスクが返されます。上位 3 SP FP 値は、*a* からそのまま渡されます。マスクは、要素ごとに、比較が真である場合は `0xffffffff` に設定され、比較が偽である場合は `0x0` に設定されます。

命令の上付き文字 'r' は、オペランドが他の命令の反転であることを示しています。

`__m128 _mm_cmpeq_ss(__m128 a, __m128 b)` CMPEQSS

等しいか比較します。

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
```

```
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpeq_ps(__m128 a, __m128 b)          CMPEQPS
```

等しいか比較します。

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmplt_ss(__m128 a, __m128 b)          CMPLTSS
```

より小さいか比較します。

```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmplt_ps(__m128 a, __m128 b)          CMPLTPS
```

より小さいか比較します。

```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffff : 0x0
r2 := (a2 < b2) ? 0xffffffff : 0x0
r3 := (a3 < b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmple_ss(__m128 a, __m128 b)          CMPLESS
```

より小さいかまたは等しいか比較します。

```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmple_ps(__m128 a, __m128 b)          CMPLEPS
```

より小さいかまたは等しいか比較します。

```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffff : 0x0
r2 := (a2 <= b2) ? 0xffffffff : 0x0
r3 := (a3 <= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpgt_ss(__m128 a, __m128 b)        CMPLTSSr
```
より大きいか比較します。
```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpgt_ps(__m128 a, __m128 b)        CMPLTPSr
```
より大きいか比較します。
```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffff : 0x0
r2 := (a2 > b2) ? 0xffffffff : 0x0
r3 := (a3 > b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpge_ss(__m128 a, __m128 b)        CMPLESSr
```
より大きいかまたは等しいか比較します。
```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpge_ps(__m128 a, __m128 b)        CMPLEPSr
```
より大きいかまたは等しいか比較します。
```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffff : 0x0
r2 := (a2 >= b2) ? 0xffffffff : 0x0
r3 := (a3 >= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpneq_ss(__m128 a, __m128 b)       CMPNEQSS
```
等しくないか比較します。
```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpneq_ps(__m128 a, __m128 b)       CMPNEQPS
```
等しくないか比較します。
```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffff : 0x0
```

```
r2 := (a2 != b2) ? 0xffffffff : 0x0
r3 := (a3 != b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)` CMPNLTSS

より小さくないか比較します。

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)` CMPNLTPS

より小さくないか比較します。

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffff : 0x0
r2 := !(a2 < b2) ? 0xffffffff : 0x0
r3 := !(a3 < b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpnle_ss(__m128 a, __m128 b)` CMPNLESS

より小さくないかまたは等しいか比較します。

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpnle_ps(__m128 a, __m128 b)` CMPNLEPS

より小さくないかまたは等しいか比較します。

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffff : 0x0
r2 := !(a2 <= b2) ? 0xffffffff : 0x0
r3 := !(a3 <= b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpngt_ss(__m128 a, __m128 b)` CMPNLTSS[r]

より大きくないか比較します。

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpngt_ps(__m128 a, __m128 b)        CMPNLTPS[r]
```
より大きくないか比較します。
```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffff : 0x0
r2 := !(a2 > b2) ? 0xffffffff : 0x0
r3 := !(a3 > b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnge_ss(__m128 a, __m128 b)        CMPNLESS[r]
```
より大きくないかまたは等しいか比較します。
```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnge_ps(__m128 a, __m128 b)        CMPNLEPS[r]
```
より大きくないかまたは等しいか比較します。
```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffff : 0x0
r2 := !(a2 >= b2) ? 0xffffffff : 0x0
r3 := !(a3 >= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpord_ss(__m128 a, __m128 b)         CMPORDSS
```
順序付けされるか比較します。
```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpord_ps(__m128 a, __m128 b)         CMPORDPS
```
順序付けされるか比較します。
```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := (a1 ord? b1) ? 0xffffffff : 0x0
r2 := (a2 ord? b2) ? 0xffffffff : 0x0
r3 := (a3 ord? b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpunord_ss(__m128 a, __m128 b)   CMPUNORDSS
```

順序付けされないか比較します。

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpunord_ps(__m128 a, __m128 b)   CMPUNORDPS
```

順序付けされないか比較します。

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := (a1 unord? b1) ? 0xffffffff : 0x0
r2 := (a2 unord? b2) ? 0xffffffff : 0x0
r3 := (a3 unord? b3) ? 0xffffffff : 0x0
```

```
int _mm_comieq_ss (__m128 a, __m128 b)           COMISS
```

a が *b* に等しいか *a* および *b* の下位 SP FP 値を比較します。*a* と *b* が等しければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_comilt_ss (__m128 a, __m128 b)           COMISS
```

a が *b* より小さいか *a* および *b* の下位 SP FP 値を比較します。*a* が *b* より小さければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_comile_ss (__m128 a, __m128 b)           COMISS
```

a が *b* より小さいかまたは等しいか *a* および *b* の下位 SP FP 値を比較します。*a* が *b* より小さいかまたは等しければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_comigt_ss (__m128 a, __m128 b)           COMISS
```

a が *b* より大きいか *a* および *b* の下位 SP FP 値を比較します。*a* が *b* より大きければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_comige_ss (__m128 a, __m128 b)                COMISS
```

a が b より大きいかまたは等しいか a および b の下位 SP FP 値を比較します。a が b より大きいがまたは等しければ、1 が返されます。そうでなければ、0 が返されます

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_comineq_ss (__m128 a, __m128 b)               COMISS
```

a が b と等しくないか a および b の下位 SP FP 値を比較します。a と b が等しくなければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 != b0) ? 0x1 : 0x0
```

```
int _mm_ucomieq_ss (__m128 a, __m128 b)              UCOMISS
```

a が b と等しいか a および b の下位 SP FP 値を比較します。a と b が等しければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_ucomilt_ss (__m128 a, __m128 b)              UCOMISS
```

a が b より小さいか a および b の下位 SP FP 値を比較します。a が b より小さければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_ucomile_ss (__m128 a, __m128 b)              UCOMISS
```

a が b より小さいかまたは等しいか a および b の下位 SP FP 値を比較します。a が b より小さいかまたは等しければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_ucomigt_ss (__m128 a, __m128 b)              UCOMISS
```

a が b より大きいか a および b の下位 SP FP 値を比較します。a が b より大きければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_ucomige_ss (__m128 a, __m128 b)          UCOMISS
```

a が *b* より大きいかまたは等しいか *a* および *b* の下位 SP FP 値を比較します。*a* が *b* より大きいかまたは等しければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_ucomineq_ss (__m128 a, __m128 b)         UCOMISS
```

a が *b* と等しくないか *a* および *b* の下位 SP FP 値を比較します。*a* と *b* が等しくなければ、1 が返されます。そうでなければ、0 が返されます。

```
r := (a0 != b0) ? 0x1 : 0x0
```

変換操作

次の表 13-6 は、変換操作の一覧です。それぞれの組み込み関数の説明、および最新のニーモニック命名規則については、表の後の記述を参照してください。以前に代替名で組み込み関数を使用していた場合には、そのまま使用することができます。

表 13-6　　変換操作

組み込み関数名	代替名	対応する命令
_mm_cvtss_si32	_mm_cvt_ss2si	CVTSS2SI
_mm_cvtps_pi32	_mm_cvt_ps2pi	CVTPS2PI
_mm_cvttss_si32	_mm_cvtt_ss2si	CVTTSS2SI
_mm_cvttps_pi32	_mm_cvtt_ps2pi	CVTTPS2PI
_mm_cvtsi32_ss	_mm_cvt_si2ss	CVTSI2SS
_mm_cvtpi32_ps	_mm_cvt_pi2ps	CVTTPS2PI

```
int _mm_cvtss_si32(__m128 a)                     CVTSS2SI
```

現在の丸めモードに従って *a* の下位 SP FP 値を 32 ビット整数に変換します。

```
r := (int)a0
```

```
__m64 _mm_cvtps_pi32(__m128 a)                   CVTPS2PI
```

現在の丸めモードに従って *a* の下位 2 SP FP 値を 2 つの 32 ビット整数に変換し、整数をパックド形式で返します。

```
r0 := (int)a0
```

```
r1 := (int)a1
```

```
int _mm_cvttss_si32(__m128 a)                    CVTTSS2SI
```

a の下位 SP FP 値を切り詰めて 32 ビット整数に変換します。

```
r := (int)a0
```

```
__m64 _mm_cvttps_pi32(__m128 a)                  CVTTPS2PI
```

a の下位 2 SP FP 値を切り詰めて 2 つの 32 ビット整数に変換し、整数をパックド形式で返します。

```
r0 := (int)a0
r1 := (int)a1
```

```
__m128 _mm_cvtsi32_ss(__m128 a, int b)           CVTSI2SS
```

32 ビット整数値 b を SP FP 値に変換します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := (float)b
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cvtpi32_ps(__m128 a, __m64 b)         CVTPI2PS
```

b にあるパックド形式の 2 つの 32 ビット整数値を 2 つの SP FP 値に変換します。上位 2 SP FP 値は、*a* からそのまま渡されます。

```
r0 := (float)b0
r1 := (float)b1
r2 := a2
r3 := a3
```

```
__m128 _mm_cvtpi16_ps(__m64 a)                   ( 合成 )
```

a にある 4 つの 16 ビット符号付き整数値を 4 つの単精度 FP 値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__m128 _mm_cvtpu16_ps(__m64 a)                   ( 合成 )
```

a にある 4 つの 16 ビット符号なし整数値を 4 つの単精度 FP 値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__m128 _mm_cvtpi8_ps(__m64 a)
```
(合成)

a にある下位 4 つの 8 ビット符号付き整数値を 4 つの単精度 FP 値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__m128 _mm_cvtpu8_ps(__m64 a)
```
(合成)

a にある下位 4 つの 8 ビット符号なし整数値を 4 つの単精度 FP 値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__m128 _mm_cvtpi32x2_ps(__m64 a, __m64 b)
```
(合成)

aおよびbにある2つの32ビット符号付き整数値を4つの単精度FP値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)b0
r3 := (float)b1
```

```
__m64 _mm_cvtps_pi16(__m128 a)
```
(合成)

a にある 4 つの単精度 FP 値を 4 つの符号付き 16 ビット整数値に変換します。

```
r0 := (short)a0
r1 := (short)a1
```

```
r2 := (short)a2
r3 := (short)a3
```

```
__m64 _mm_cvtps_pi8(__m128 a)                                    ( 合成 )
```

aにある4つの単精度FP値を結果の下位4つの8ビット符号付き整数値に変換します。

```
r0 := (char)a0
r1 := (char)a1
r2 := (char)a2
r3 := (char)a3
```

その他

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, int i) SHUFPS
```

マスク *i* に基づいて 4 つの特定の SP FP 値を *a* および *b* から選択します。マスクは、即値でなければなりません。シャッフルのセマンティクスの説明については、この節の最後にある「シャッフルのためのマクロ関数」を参照してください。

```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)      UNPCKHPS
```

a および *b* から上位 2 SP FP 値を選択してインタリーブします。

```
r0 := a2
r1 := b2
r2 := a3
r3 := b3
```

```
__m128 _mm_unpacklo_ps(__m128 a, __m128 b)      UNPCKLPS
```

a および *b* から下位 2 SP FP 値を選択してインタリーブします。

```
r0 := a0
r1 := b0
r2 := a1
r3 := b1
```

```
__m128 _mm_loadh_pi(__m128 a, __m64 *p) MOVHPS reg, mem
```

上位 2 SP FP 値をアドレス *p* からロードされるデータの 64 ビットで設定します。下位 2 値は、*a* からそのまま渡されます。

```
r0 := a0
r1 := a1
r2 := *p0
r3 := *p1
```

```
void _mm_storeh_pi(__m64 *p, __m128 a)          MOVHPS mem, reg
```

a の上位 2 SP FP 値をアドレス *p* にストアします。

```
*p0 := a2
*p1 := a3
```

```
__m128 _mm_movehl_ps (__m128 a, __m128 b) MOVHLPS
```

b の上位 2 SP FP 値を結果の下位 2 SP FP 値に移動します。
a の上位 2 SP FP 値は結果にそのまま渡されます。

```
r3 := a3
r2 := a2
r1 := b3
r0 := b2
```

```
__m128 _mm_movelh_ps (__m128 a, __m128 b) MOVLHPS
```

b の下位 2 SP FP 値を結果の上位 2 SP FP 値に移動します。*a* の下位 2 SP FP 値は結果にそのまま渡されます。

```
r3 := b1
r2 := b0
r1 := a1
r0 := a0
```

```
__m128 _mm_loadl_pi(__m128 a, __m64 *p) MOVLPS reg, mem
```

下位 2 SP FP 値をアドレス *p* からロードされるデータの 64 ビットで設定します。上位 2 値は、*a* からそのまま渡されます。

```
r0 := *p0
r1 := *p1
r2 := a2
r3 := a3
```

```
void _mm_storel_pi(__m64 *p, __m128 a)  MOVLPS mem, reg
```

a の下位 2 SP FP 値をアドレス *p* にストアします。

```
*p0 := b0
*p1 := b1
```

```
int _mm_movemask_ps(__m128 a)                    MOVMSKPS
```

4 つの SP FP 値の最上位ビットから 4 ビット・マスクを作成します。

```
r := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)
```

```
unsigned int _mm_getcsr(void)                    STMXCSR
```

制御レジスタの内容を返します。

```
void _mm_setcsr(unsigned int i)                  LDMXCSR
```

制御レジスタを指定された値に設定します。

シャッフルのためのマクロ関数

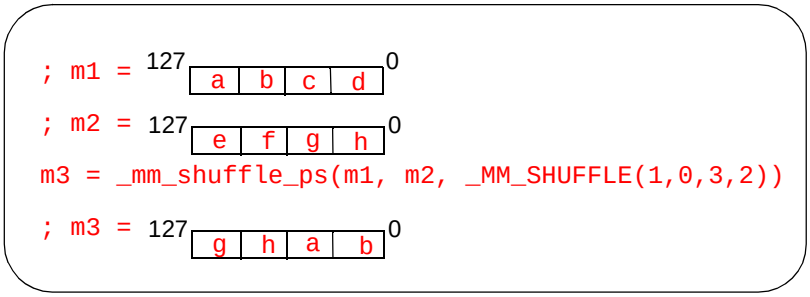
ストリーミング SIMD 拡張命令は、シャッフル操作を記述する定数を作成しやすくする次のマクロ関数を提供しています。このマクロは、4 つの (0 から 3 までの範囲の) 小さい整数を使用し、それらを SHUFPS 命令によって使用される 8 ビット即値に結合します。例 13-2 を参照してください。

例 13-2 シャッフル関数マクロ

```
_MM_SHUFFLE(z,y,x,w)
  次のように展開されます。
  (z<<6) | (y<<4) | (x<<2) | w
```

4 つの整数は、結果のワードに入れる第 1 入力オペランドからの 2 ワードと第 2 入力オペランドからの 2 ワードを選択するためのセレクタとして見ることができます。

例 13-3 ワードの最初の状態とシャッフル関数マクロを使用した結果



制御レジスタの読み取り / 書き込みのためのマクロ関数

以下のマクロ関数を使用すると、制御レジスタに対してビットの読み取りおよび書き込みが可能になります。詳細については、この章で後述の「セット操作」を参照してください。

例外状態マクロ	マクロ引数
_MM_SET_EXCEPTION_STATE(x)	_MM_EXCEPT_INVALID
_MM_GET_EXCEPTION_STATE()	_MM_EXCEPT_DIV_ZERO
マクロ定義	_MM_EXCEPT_DENORM
制御レジスタの最下位から 6 桁目のビットに対して書き込みと読み取りを行います。	_MM_EXCEPT_OVERFLOW
	_MM_EXCEPT_UNDERFLOW
	_MM_EXCEPT_INEXACT

例 13-4 は、ゼロ除算例外状態マクロをテストします。

例 13-4 _MM_EXCEPT_DIV_ZERO を使用した例外状態マクロ

```
if (_MM_GET_EXCEPTION_STATE(x) & _MM_EXCEPT_DIV_ZERO) {
    /* Exception has occurred */
}
```

例外マスク・マクロ	マクロ引数
_MM_SET_EXCEPTION_MASK(x)	_MM_MASK_INVALID

_MM_GET_EXCEPTION_MASK()

マクロ定義

制御レジスタの 7 ～ 12 桁目のビットに対して書き込みと読み取りを行います。
注：6 つの例外マスク・ビットは常に影響を受けます。明示的に設定されていないビットはクリアされます。

_MM_MASK_DIV_ZERO
_MM_MASK_DENORM
_MM_MASK_OVERFLOW
_MM_MASK_UNDERFLOW
_MM_MASK_INEXACT

例 13-5 は、オーバーフロー例外およびアンダーフロー例外をマスクし、これ以外のすべての例外のマスクをはずします。

例 13-5 _MM_MASK_OVERFLOW および _MM_MASK_UNDERFLOW を使用した例外マスク

```
_MM_SET_EXCEPTION_MASK( _MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)
```

丸めモード

_MM_SET_ROUNDING_MODE(x)

_MM_GET_ROUNDING_MODE()

マクロ引数
_MM_ROUND_NEAREST
_MM_ROUND_DOWN
_MM_ROUND_UP
_MM_ROUND_TOWARD_ZERO

マクロ定義

制御レジスタの 13 桁目と 14 桁目のビットに対して書き込みおよび読み取りを行います。

例 13-6 は、ゼロ側に丸める丸めモードをテストします。

例 13-6 _MM_ROUND_TOWARD_ZERO を使用した丸めモード

```
if (_MM_GET_ROUNDING_MODE() == _MM_ROUND_TOWARD_ZERO) {
    /* Rounding mode is round toward zero */
}
```

ゼロフラッシュモード

_MM_SET_FLUSH_ZERO_MODE(x)

_MM_GET_FLUSH_ZERO_MODE()

マクロ引数
_MM_FLUSH_ZERO_ON
_MM_FLUSH_ZERO_OFF

マクロ定義
制御レジスタの 15 桁目のビットに対して書き込みおよび読み取りを行います。

例 13-7 は、ゼロフラッシュモードを無効にします。

例 13-7 _MM_FLUSH_ZERO_OFF を使用したゼロフラッシュモード

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSHZERO_OFF)
```

マトリックス入れ替えのためのマクロ関数

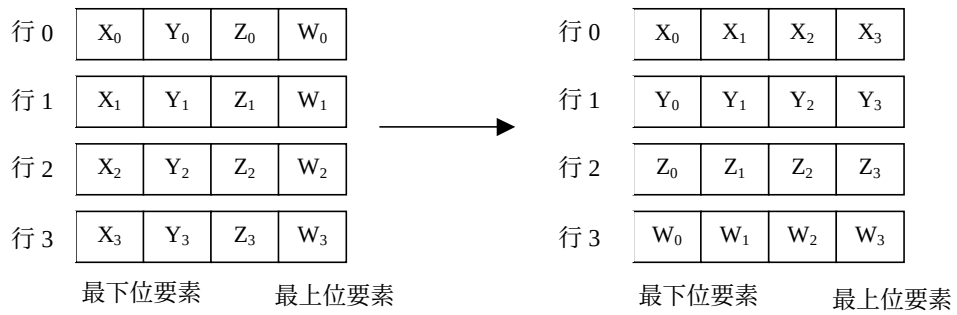
ストリーミング SIMD 拡張命令は、単精度浮動小数点値の 4 × 4 マトリックスを入れ替える次のマクロ関数も提供しています。

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

引数 row0、row1、row2、および row3 は、その要素が 4 × 4 マトリックスの対応する行を形成する __m128 値です。マトリックス入れ替えは、引数 row0、row1、row2、および row3 に返され、その時点では、row0 は元のマトリックスの列 0、row1 は元のマトリックスの列 1 のように保持します。

このマクロの入れ替え機能を図 13-2 に示しています。

図 13-2 _MM_TRANSPOSE4_PS マクロを使用したマトリックス入れ替え



メモリおよび初期設定

この節では、ロード、セット、およびストア操作について説明します。これらの操作を使用して、データのロードおよびメモリへのストアを行なうことかできます。ロード操作とセット操作は、両方とも __m128 データを初期設定するという点では同様です。しかし、セット操作は、浮動型引数を使用して定数で初期設定するためのものであるのに対し、ロード操作は、浮動小数点引数を使用してメモリからデータをロードする命令をまねるためのものです。ストア操作は、初期設定されたデータをアドレスに割り当てます。

ロード操作

```
__m128 _mm_load_ss(float *p)                    MOVSS
```

SP FP 値を下位ワードにロードし、上位 3 ワードをクリアします。

```
r0 := *p
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

```
__m128 _mm_load1_ps(float *p)       MOVSS + シャッフリング
```

または

```
__m128 _mm_load_ps1(float *p)       MOVSS + シャッフリング
```

1 つの SP FP 値をロードし、これを 4 ワードすべてにコピーします。

```
r0 := *p
r1 := *p
r2 := *p
r3 := *p
```

```
__m128 _mm_load_ps(float *p)                    MOVAPS
```

4 つの SP FP 値をロードします。アドレスは、16 バイトにアライメントが合っていなければなりません。

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

```
__m128 _mm_loadu_ps(float *p)                   MOVUPS
```

4 つの SP FP 値をロードします。アドレスは、16 バイトにアライメントが合っている必要はありません。

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

```
__m128 _mm_loadr_ps(float *p)        MOVAPS + シャッフリング
```

4 つの SP FP 値を逆順にロードします。アドレスは、16 バイトにアライメントが合っていなければなりません。

```
r0 := p[3]
r1 := p[2]
r2 := p[1]
r3 := p[0]
```

セット操作

```
__m128 _mm_set_ss(float w)                                    ( 合成 )
```

SP FP 値の下位ワードを *w* に設定し、上位 3 ワードをクリアします。

```
r0 := w
r1 := r2 := r3 := 0.0
```

```
__m128 _mm_set1_ps(float w)                                   ( 合成 )
```

または

```
__m128 _mm_set_ps1(float w)                                   ( 合成 )
```

4 つの SP FP 値を *w* に設定します。

```
r0 := r1 := r2 := r3 := w
```

```
__m128 _mm_set_ps(float z, float y, float x, float w)
                                                              ( 合成 )
```

4 つの SP FP 値を 4 つの入力に設定します。

```
r0 := w
r1 := x
r2 := y
r3 := z
```

```
__m128 _mm_setr_ps(float z, float y, float x, float w)
```
(合成)

4 つの SP FP 値を 4 つの入力に逆順で設定します。

```
r0 := z
r1 := y
r2 := x
r3 := w
```

```
__m128 _mm_setzero_ps(void)
```
(合成)

4 つの SP FP 値をクリアします。

```
r0 := r1 := r2 := r3 := 0.0
```

ストア操作

```
void _mm_store_ss(float *p, __m128 a)
```
MOVSS

下位 SP FP 値をストアします。

```
*p := a0
```

```
void _mm_store1_ps(float *p, __m128 a)
```
MOVSS + シャッフリング

または

```
void _mm_store_ps1(float *p, __m128 a)
```
MOVSS + シャッフリング

下位 SP FP 値を 4 ワードにわたってストアします。

```
p[0] := a0
p[1] := a0
p[2] := a0
p[3] := a0
```

```
void _mm_store_ps(float *p, __m128 a)
```
MOVAPS

4 つの SP FP 値をストアします。アドレスは、16 バイトにアライメントが合っていなければなりません。

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

```
void _mm_storeu_ps(float *p, __m128 a)                    MOVUPS
```

4 つの SP FP 値をストアします。アドレスは、16 バイトにアライメントが合っている必要はありません。

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

```
void _mm_storer_ps(float *p, __m128 a)
                                        MOVAPS + シャッフリング
```

4 つの SP FP 値を逆順でストアします。アドレスは、16 バイトにアライメントが合っていなければなりません。

```
p[0] := a3
p[1] := a2
p[2] := a1
p[3] := a0
```

```
__m128 _mm_move_ss(__m128 a, __m128 b)                    MOVSS
```

下位ワードを *b* の SP FP 値に設定します。上位 3 SP FP 値は、*a* からそのまま渡されます。

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

整数組み込み関数

次の表 13-7 は、整数組み込み関数の一覧です。それぞれの組み込み関数の説明、および最新のニーモニック命名規則については、表の後の記述を参照してください。以前に代替名で組み込み関数を使用していた場合には、そのまま使用することができます。

表 13-7　整数組み込み関数

組み込み関数名	代替名	対応する命令
_mm_extract_pi16	_m_pextrw	PEXTRW
_mm_insert_pi16	_m_pinsrw	PINSRW
_mm_max_pi16	_m_pmaxsw	PMAXSW
_mm_max_pu8	_m_pmaxub	PMAXUB

表 13-7 整数組み込み関数 (続き)

組み込み関数名	代替名	対応する命令
_mm_min_pi16	_m_pminsw	PMINSW
_mm_min_pu8	_m_pminub	PMINUB
_mm_movemask_pi8	_m_pmovmskb	PMOVMSKB
_mm_mulhi_pu16	_m_pmulhuw	PMULHUW
_mm_shuffle_pi16	_m_pshufw	PSHUFW
_mm_maskmove_si64	_m_maskmovq	MASKMOVQ
_mm_avg_pu8	_m_pavgb	PAVGB
_mm_avg_pu16	_m_pavgw	PAVGW
_mm_sad_pu8	_m_psadbw	PSADBW

この組み込み関数を使用する場合は、mmx レジスタのステートをクリアする必要があります。「EMMS 命令：必要性と使用法」を参照してください。

```
int _mm_extract_pi16(__m64 a, int n)                PEXTRW
```

a の 4 ワードのうちの 1 つを抽出します。セレクタ *n* は、即値でなければなりません。

```
r := (n==0) ? a0 : ( (n==1) ? a1 : ( (n==2) ? a2 : a3 ) )
```

```
__m64 _mm_insert_pi16(__m64 a, int d, int n)      PINSRW
```

ワード *d* を *a* の 4 ワードのうちの 1 つに挿入します。セレクタ *n* は、即値でなければなりません。

```
r0 := (n==0) ? d : a0;
r1 := (n==1) ? d : a1;
r2 := (n==2) ? d : a2;
r3 := (n==3) ? d : a3;
```

```
__m64 _mm_max_pi16(__m64 a, __m64 b)                PMAXSW
```

a および *b* にあるワードの要素ごとの最大を求めます。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _mm_max_pu8(__m64 a, __m64 b)                PMAXUB
```

a および *b* にある符号なしバイトの要素ごとの最大を求めます。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
__m64 _mm_min_pi16(__m64 a, __m64 b)               PMINSW
```

a および *b* にあるワードの要素ごとの最小を求めます。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _mm_min_pu8(__m64 a, __m64 b)                PMINUB
```

a および *b* にある符号なしバイトの要素ごとの最小を求めます。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
int _mm_movemask_pi8(__m64 a)                      PMOVMSKB
```

a にあるバイトの最上位ビットから 8 ビット・マスクを作成します。

```
r := sign(a7)<<7 | sign(a6)<<6 | ... | sign(a0)
```

```
__m64 _mm_mulhi_pu16(__m64 a, __m64 b)             PMULHUW
```

a および *b* の符号なしワードを乗算し、32 ビット即値結果の上位 16 ビットを返します。

```
r0 := hiword(a0 * b0)
r1 := hiword(a1 * b1)
r2 := hiword(a2 * b2)
r3 := hiword(a3 * b3)
```

```
__m64 _mm_shuffle_pi16(__m64 a, int n)                PSHUFW
```

a の 4 ワードの組み合わせを返します。セレクタ *n* は、即値でなければなりません。

```
r0 := word (n&0x3) of a
r1 := word ((n>>2)&0x3) of a
r2 := word ((n>>4)&0x3) of a
r3 := word ((n>>6)&0x3) of a
```

```
void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MASKMOVQ
```

d のバイト要素を条件付きでアドレス *p* にストアします。セレクタ *n* の各バイトの最上位ビットが *d* の対応するバイトをストアするかどうかを決定します。

```
if (sign(n0)) p[0] := d0
if (sign(n1)) p[1] := d1
...
if (sign(n7)) p[7] := d7
```

```
__m64 _mm_avg_pu8(__m64 a, __m64 b)                   PAVGB
```

a および *b* にある符号なしバイトの (丸め) 平均を計算します。

```
t = (unsigned short)a0 + (unsigned short)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned short)a7 + (unsigned short)b7
r7 = (unsigned char)((t >> 1) | (t & 0x01))
```

```
__m64 _mm_avg_pu16(__m64 a, __m64 b)                  PAVGW
```

a および *b* にある符号なしワードの (丸め) 平均を計算します。

```
t = (unsigned int)a0 + (unsigned int)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned word)a7 + (unsigned word)b7
r7 = (unsigned short)((t >> 1) | (t & 0x01))
```

```
__m64 _mm_sad_pu8(__m64 a, __m64 b)                    PSADBW
```

a および *b* にある符号なしバイトの絶対差の合計を計算し、下位ワードの値を返します。上位 3 ワードはクリアされます。

```
r0 = abs(a0-b0) + ... + abs(a7-b7)
r1 = r2 = r3 = 0
```

キャッシュ操作サポート

以下の組み込み関数を使用すると、キャッシュを効率よく使用できるようになります。

```
void _mm_prefetch(char *p, int i)                      PREFETCH
```

アドレス *p* からの 1 キャッシュ・ライン分のデータをプロセッサに「より近い」位置のキャッシュ階層にロードします。値 *i* は、プリフェッチ操作のタイプを指定します。つまり、プリフェッチ命令のタイプに応じて、定数 _MM_HINT_T0、_MM_HINT_T1、_MM_HINT_T2、および _MM_HINT_NTA を使用する必要があります。

```
void _mm_stream_pi(__m64 *p, __m64 a)                  MOVNTQ
```

a にあるデータをアドレス *p* にストアします。この場合データはキャッシュに書き込まれていません。この組み込み関数を使用する場合は、mmx レジスタのステートをクリアする必要があります。「EMMS 命令：必要性と使用法」を参照してください。

```
void _mm_stream_ps(float *p, __m128 a)                 MOVNTPS
```

a にあるデータをアドレス *p* にストアします。アドレスは、16 バイトにアライメントが合っていなければなりません。この場合データはキャッシュに書き込まれていません。

```
void _mm_sfence(void)                                  SFENCE
```

すべての先行するストアが後続のストアの前に完了することを保証します。

データ・アライメント

パフォーマンスを向上させるため、ストリーミング SIMD 拡張命令を使用するときは、メモリ操作でデータを 16 バイトにアライメントを合わせる必要があります。特に、アドレスを _mm_load および _mm_store 組み込み関数に渡すときに __m128 オブジェクトのアライメントを合わせなければなりません。浮動小数点型の配列を宣言して型変換によってそれらを __m128 オブジェクトとして扱いたい場合は、浮動小数点型配列のアライメントが正しく合っていることを保証する必要があります。

アライメント・サポート

__declspec(align) を使用して、デフォルトで行なわれる場合よりもより厳格にデータのアライメントを合わせることをコンパイラに指示します。たとえば、型 int のデータ・オブジェクトは、デフォルトでは 4(int のサイズ) の倍数であるバイト・アドレスに割り当てられます。しかし、__declspec(align) を使用すると、代わりに、次の制約とともに 8、16、または 32 の倍数であるアドレスを使用することをコンパイラに指示することができます。

- 32 バイト・アドレスは静的に割り当てなければなりません
- 16 バイト・アドレスは局所的にまたは静的に割り当てることができます

このデータ・アライメント・サポートをキャッシュ・ラインの使用を最適化する際の利点として使用することができます。一般的に一緒に使用される小さいオブジェクトを構造体にまとめて、その構造体をキャッシュ・ラインの先頭に割り当てるようにすると、どれか 1 つのオブジェクトがアクセスされれば同時に各オブジェクトもキャッシュにロードされることを実質的に保証することができ、かなりのパフォーマンスの向上となります。

この拡張属性の構文は、次のとおりです。

align(*n*)

ここで、*n* は 32 以下の 2 の整数累乗です。指定する値は、要求されるアライメントです。

注：*影響を受けるデータ型のアライメントより小さい値を指定しても、効果はありません。言い換えると、データは、その独自のアライメントまたは __declspec(align) で指定されたアライメントのうちの大きいほうにアライメントを合わせられます。*

継続時間が、静的または自動的ストレージに関係なく、個別変数にアライメントを要求することができます。(デフォルトでは、グローバル変数と静的変数は静的ストレージ・デュレーションを持ち、ローカル変数は自動ストレージ・デュレーションを持ちます。) しかし、パラメータあるいは構造体またはクラスのフィールドのアライメントを調整することはできません。しかし、構造体 (または共用体またはクラス) のアライメントを増加させることはでき、その場合には、その型のすべてのオブジェクトが影響を受けます。

例として、関数がローカル変数 i および j を 2 次元配列の添字として使用すると想定します。これらの変数は、次のように宣言することができます。

```
int i, j;
```

これらの変数は、一般的には一緒に使用されます。しかし、異なるキャッシュ・ラインに入る可能性があり、そうなるとパフォーマンスに対して有害になるおそれがあります。代わりにそれらを次のように宣言することができます。

```
__declspec(align(8)) struct { int i, j; } sub;
```

コンパイラは、この場合にはそれらが同じキャッシュ・ラインに割り当てられることを保証します。C++ では、(上の例では *sub* として書かれている) 構造体の変数名を省略することができます。しかし、C では必要であり、`i` および `j` への参照を *`sub.i`* および *`sub.j`* として書かなければなりません。

そのような添字ペアで多くの関数を使用する場合は、次の例にあるように、それらに構造体型を宣言して使用するほうが便利です。

```
typedef struct __declspec(align(8)) { int i, j; } Sub;
```

キーワード `struct` の後に `__declspec(align)` を置くことによって、その型のすべてのオブジェクトに対して適当なアライメントを要求しています。しかし、パラメータの割り当ては、`__declspec(align)` によって影響されません。必要な場合には、パラメータの値をローカル変数に適当なアライメントで割り当てることができます。

配列などグローバル変数のアライメントを次のようにして強制することもできます。

```
__declspec(align(16)) float array[1000];
```

動的スタック・フレーム・アライメント

デフォルトでは、8 または 16 バイトのオブジェクトを使用する関数がコンパイラによる動的アライメントの対象となります。(一定のパフォーマンスを保つため) 動的なスタック・アライメントが関数単位のレベルで必要かどうかは、コンパイラによって判断されます。このことは、`double`、`__m64`、および `__m128` ローカル変数への参照が 16 で偶数に除算可能であるアドレスにアライメントを合わせられることを保証します。パラメータのアライメントは、影響を受けません。

スタック・フレーム・アライメントは、表 13-8 に説明しているように -Qsfalign オプションによって制御されます。

表 13-8 スタック・フレーム・アライメントのオプション

オプション	意味
-Qsfalign8	**コンパイラによって適切だと判断された場合に、8 または 16 バイト変数を持つフレームまたは関数のスタックに対しアライメントを行います。**
-Qsfalign16	**コンパイラによって適切だと判断された場合に、16 バイト変数を持つフレームまたは関数のスタックに対しアライメントを行います。**
-Qsfalign-	**すべての関数のスタック・アライメントを無効にします。**

アライメント後のフレームを使用する場合は、インライン展開されたアセンブリ・ブロックの EBX レジスタを変更してはなりません。これは、EBX が引数ブロックの記録に使用されているためです。EBX を使用するたびに保存して復元する場合に限り、変更してもかまいません。Intel C/C++ コンパイラでは、EBX レジスタを使用してこれらのデータ型の変数のアライメントを制御します。そのため、EBX を保存せずに使用した場合は、プログラムの実行結果は保証されません。

インライン・アセンブリ・コードでの EBX の使用に関する補足情報、およびその他の関連情報については、アプリケーション・ノート『AP-833 Data Alignment and Programming Issues with the Intel C/C++ Compiler』(資料番号：243872-001)、および『AP-589 Software Conventions for the Streaming SIMD Extensions』(資料番号：243873-001) を参照してください。

アセンブリ言語サポート

Intel C/C++ コンパイラでは、アセンブリ・ブロックのインライン展開、およびアセンブリ・ファイルの生成をサポートしています。

インライン・アセンブリ

コンパイラは、インライン・アセンブリ (_asm) ブロックですべての MMX およびストリーミング SIMD 拡張命令の使用をサポートしています。コンパイラは、64 ビット・データおよび 128 ビット・データを参照するための新しい構文 NMWORD PTR および XMMWORD PTR を受け付けます。

アセンブリ・ファイルの生成

-S コンパイラ・スイッチを使用して、アセンブリ・リスト出力を生成します。これは、コンパイラが生成したコードを手作業でチューニングするために役立ちます。アセンブリ・ファイルは、iammx.inc および iaxmm.inc MASM インクルード・ファイルで MASM バージョン 6.12 を使用して直接コンパイルすることができます。生成される .asm ファイルには、MMX 組み込み関数またはストリーミング SIMD 拡張命令が使用されるときにそれらへのインクルード・ディレクティブが含まれています。インライン・アセンブリの使用法に関する詳細は、第 6 章の「アセンブリ・コード・リストの作成 (-S)」を参照してください。

コンパイラによるベクトル化のサポートとガイドライン

14

この章では、Intel C/C++ コンパイラに関するベクトル化のガイドラインとオプションについて、例を挙げながら説明します。この章の内容は、次のとおりです。

- ベクトル化の機能と特徴についてのクイック・リファレンス
- ベクトル化を制御するためのコンパイラ・スイッチの説明
- ベクトル化を制御するための C/C++ 言語機能の説明
- ベクトル化のレベルに関する考察と一般的なガイドライン
  - 自動ベクトル化
  - ユーザ介入によるベクトル化
  - ベクトル化なし
- 一般的なベクトル化の問題と解決法に関する例と説明

ベクトライザのクイック・リファレンス

表 14-1 に、Intel C/C++ コンパイラのベクトル化に使用されるコマンド・ライン・スイッチの一覧を示します。

表 14-1 ベクトル化のコマンド・ライン・スイッチ

オプション	説明	デフォルト	参照先
`-Qvec`	ベクトライザを有効にします。	オフ	14-2 ページ
`-Qvec_alignment`	ベクトル化可能なデータに対するデフォルトのアライメントを制御します。	オフ	14-2 ページ
`-Qvec_verbose`	ベクトライザの診断レベルを制御します。	`n=1`	14-3 ページ

続く

表 14-1 ベクトル化のコマンド・ライン・スイッチ (続き)

オプション	説明	デフォルト	参照先
-Qrestrict	restrict 修飾子と一緒に指定して、ポインタの明確化を有効にします。	オフ	14-3 ページ
-Qkscalar	デフォルトの x87 命令でなく、ストリーミング SIMD 拡張命令を使用して、すべての 32 ビット浮動小数点演算を実行します。	オフ	14-3 ページ
-Qvec_emms[-]	mmx 命令レジスタを空にするための EMMS 命令の自動挿入を制御します。	オン	14-3 ページ
-Qvec_no_arg_alias[-]	エントリ時にプロシージャの引数がエイリアシングされないものと仮定します。	オフ	14-4 ページ
-Qvec_no_alias[-]	名前の異なるオブジェクト間でエイリアシングがないものと仮定します。	オフ	14-4 ページ

コマンド・ライン・スイッチのサポート

コンパイラのコマンド・ラインには、以下のベクトライザ・スイッチ・オプションを指定できます。これらを効果的に使用し、予期せぬプログラムの動作を避けるためにも、コンパイラに多数備えられている拡張機能と同様に、スイッチの機能についても正しく理解しておく必要があります。

-Qvec
ベクトライザを有効にします。これ以外のすべてのベクトル化スイッチを使用するには、このオプションをオンにしておかなければなりません。

-Qvec_alignment[n]
ベクトル化可能なデータに対するデフォルトのアライメントを制御します。デフォルトでは、コンパイラは本来のアライメントを想定します。例えば、4 バイトにアライメントされた float 型の配列の場合は、コンパイラは「アライメントに従わない」メモリ演算を実行します。ただし、コンパイラが適当だと判断するアライメントについては、アライメントに従った命令を使ってパフォーマンスを改善します。コンパイラのデフォルトのアライメントを無効にするには、次のサブオプションを使用します。

n=0: デフォルトのアライメント
n=1: アライメントに従わない演算を想定

n=2: アライメントに従った演算を想定

この機能が使用されるのは、一般に、ポインタが含まれるデータの受け渡しを、__declspec(align) を使用せずに行うプログラムにおいてです。第 13 章の「データ・アライメント」を参照してください。

注意：*デフォルトのアライメントを変更する場合は、絶対に安全であることを確認しておく必要があります。そうしないと、コンパイラによって不正なコードが生成されます。*

-Qvec_emms[-]

整数ループがベクトル化されるたびに、コンパイラが EMMS 命令を挿入するかどうかを制御します。第 13 章の「EMMS 命令：必要性と使用法」を参照してください。

注意：*プログラムに MMX® テクノロジが使用されているかどうかが不明な場合は、使用時の安全性を確認してから、-Qvec_emms[-] を使用するようにしてください。*

-Qrestrict restrict キーワードは、別名の仮定条件に明確さに関する柔軟性を持たせるための修飾子で、これによって、ベクトル化や他の最適化オプションをさらに適用させることができます。restrict キーワードは言語拡張機能であるため、これを有効にするには、-Qrestrict オプションを使用しなければなりません。

-Qkscalar ストリーミング SIMD 拡張命令が含まれる 32 ビット・スカラー演算で、32 ビット浮動小数点演算が実行されるように指定します。

-Qvec_verbose[n]

診断メッセージのベクトライザのレベルを制御します。

n=0: 診断情報を表示しない

n=1: ループのベクトル化が成功したことを示す診断メッセージを表示する (デフォルト)

n=2: n=1 のメッセージのほか、ループのベクトル化が失敗したことを示す診断メッセージも表示する

n=3: n=2 のメッセージのほか、確定された依存関係または想定される依存関係についての補足情報を表示する

-Qvec_no_arg_alias[-]

プロシージャ引数が、エントリ時に直接参照によっても間接参照によってもエイリアシングされないものと仮定します。例 14-1 に示すループは、-Qvec_no_arg_alias スイッチを使用しない限り、ベクトル化されません。

例 14-1 -Qvec_no_arg_alias を使用したベクトル化

```
void f(float *out, float *in){
  int i;
  for (i = 0; i < 100; i++)
    out[i] = in[i];
}
```

注意：*プロシージャ引数がエイリアシングされないという判断を誤ると、プログラムが正しく動作しません。-Qvec_no_arg_alias でも -Qvec_no_alias スイッチでも、この点を注意する必要があります。*

-Qvec_no_alias[-]

異なる名前のオブジェクト間では、直接参照によっても間接参照によってもエイリアシングがないものと仮定します。例 14-2 に示すループは、-Qvec_no_alias スイッチを使用しない限り、ベクトル化されません。

例 14-2 -Qvec_no_alias を使用したベクトル化

```
void f(float *p){
int i;
float *out=p;
float *in=p;

for (i = 0; i < 100; i++)
  out[i] = in[i];
}
```

言語サポートとプラグマ

この節では、対象となるハードウェア上の制限や様式上の問題に対してベクトル化を支援するための言語機能について説明します。`declspec(align(n))` 宣言を使用すると、ハードウェアによるアライメント上の制約を解消することができます。また、`restrict` 修飾子やプラグマを使用すると、字句解析上のスコープ、データの相互依存関係、および解決のあいまいさなどによって生じる様式上の問題に対処できます。

表 14-2 言語サポート

オプション	説明	参照先
`declspec(align(n))`	変数 `var-name` を `n` バイト境界にアライメントするようにコンパイラに指示します。	14-6 ページ
`restrict`	別名の仮定条件に明確さに関する柔軟性を持たせ、ベクトル化の対象を広げます。	14-3 ページ
`#pragma ivdep`	想定されるベクトルの依存関係を無視するようにコンパイラに命令します。	14-7 ページ
`#pragma vector {aligned | unaligned}`	ループのベクトル化の方法を指定します。	14-8 ページ
`#pragma novector`	ループがベクトル化されないように指定します。	14-9 ページ

declspec によるアライメント

構文： `__declspec(align(n)) <type> var-name`

定義： 変数 *var-name* を *n* バイト境界にアライメントするようにコンパイラに指示します。このとき、*n* は 2 ～ 32 の数値でなければなりません。第 13 章の「データ・アライメント」を参照してください。

例 14-3 は、`float` 型の配列を 16 バイトにアライメントするようにコンパイラに指示する関数を示しています。

例 14-3　`declspec(align(n))` を使用したアライメント

```
__declspec(align(16)) float x[1024];
```

restrict キーワードによる修飾

構文： `<type> *restrict p`

定義： `restrict` キーワードは修飾子です。例えば、`p` がオブジェクト `x` に対する制限付きポインタであるとすると、`p` の字句解析のスコープに含まれる `x` への参照はすべて、`p` が含まれる式で呼び出さなければなりません。この修飾子を使用すると、別名の仮定条件に明確さに関する柔軟性を持たせ、実行可能なベクトル化や他の最適化オプションの対象範囲を広げることができます。一般に、この機能は、関数の引数が別名でないことを示すために使用されます。

`restrict` キーワードは言語拡張機能であるため、これを有効にするには `-Qrestrict` オプションを使用しなければなりません。その後ろに制限付きポインタを指定しない場合は、コンパイラは `a[i]` が `b[i-1]` にエイリアシングされるものと仮定することになるため、ベクトル化は行われません。

例 14-4 修飾子としての restrict キーワードの使い方

```
void foo (float *restrict a, float *b) {
  for (i=0; i<N; i++)
    a[i] = b[i] * 2.0f;
}
```

プラグマのスコープ

以下に説明するプラグマは、プログラム内の後続のループに対してのみ、ベクトル化を制御します。入れ子になっているループにはプラグマは適用されません。入れ子になったループにプラグマを適用するには、そのループそのもののプラグマをループの前に指定する必要があります。プラグマは、ループ制御文の前にしか置くことができません。ただし、プラグマとその影響を受けるループの間に代入文と関数呼び出しが置かれることもあります。

#pragma ivdep

構文： `#pragma ivdep`

定義： このプラグマは、想定されるベクトルの依存関係を無視するようにコンパイラに指示します。コンパイラは、コードの正当性を保証するために、想定される依存関係を確定した依存関係として扱い、これによってベクトル化を回避します。しかし、このプラグマを使用すると、コンパイラのこの判断を無効にすることができます。このプラグマは、想定されるループの依存関係を無視しても安全であることが確認された場合のみ使用するようにしてください。

例 14-5 に示すループでは、`k` の値がわからないため (`k<0` の可能性もあるため)、`ivdep` プラグマを使用しないとベクトル化されません。

例 14-5　#pragma ivdep を使用したループ

```
#pragma ivdep
  for (i=0; i<m; i++) {
    a[i] = a[i+k] * c;
...
  }
```

#pragma vector

構文：　`#pragma vector {aligned |unaligned}`

定義：　`vector` ループ・プラグマは、ループがベクトル化されることが正当である場合に、有益かどうかの通常のヒューリスティックな判断を無視して、ループをベクトル化するように指定します。このプラグマと `aligned`(または `unaligned`) 修飾子を一緒に使用すると、ループは `aligned`(または `unaligned`) 演算によってベクトル化されます。この場合に指定できるのは、`aligned` または `unaligned` のどちらか 1 つだけです。プラグマを使用してループ内のアライメントを制御すると、`-Qvec_alignment[n]` オプションは無効になります。

例 14-6 に示すループは、ベクトル・プラグマを使用することによって、ストライド -2 での参照であるにもかかわらず、ループがベクトル化されることを示しています。

例 14-6　ストライド -2 の #pragma vector を使用したループ

```
#pragma vector
  for (i=0; i<m; i+=2) {
    a[i] = a[i] * c;
  }
```

例 14-7 に示すループは、通常はコンパイラが安全だと判断できないような方法で配列が宣言されているため、aligned 修飾子を使用することによって、アライメントに従った命令でループがベクトル化されるように要求しています。

例 14-7 #pragma vector aligned を使用したループ

```
void f(float *a) {
#pragma vector aligned
  for (i=0; i<m; i+=2) {
    a[i] = a[i] * c;
  }
}
```

#pragma novector

構文： `#pragma novector`

定義： `novector` ループ・プラグマは、ループのベクトル化が正当で有益であっても、ループがベクトル化されないように指定します。

例 14-8 の `novector` プラグマを見ると、トリップカウントが低すぎてベクトル化を行う価値がないということを、プログラマが認識していることがわかります。これは、(`ub - lb`) の値が小さいことによって示されます。

例 14-8 #pragma novector を使用したトリップカウントの低いループ

```
void f(int lb, int ub) {
...
#pragma novector
  for (j=lb; j<ub; j++) {
  a[j] = a[j] + b[i];
  }
...
}
```

ループ構造のコーディングの背景

ベクトル化を実行するコンパイラの最終目的は、ソース・コードを自動的に並列処理することです。しかし、この最終目的を実現するのは、よく知られるベクトル化コンパイラでも困難でした。自動並列処理を実現することが困難な理由は、主に次の 2 つの要因によります。

1. *様式* — ソース・コードを書くときの様式が、最適化の妨げとなる場合があります。例えば、大域的なポインタの一般的な問題として、これがあるために、2 つのメモリ参照が別個の位置にあることをコンパイラが判断できなくなるという点があります。その結果、特定の順序変更に関する変換が回避されてしまいます。
2. *ハードウェア上の制約* — コンパイラは、根底となるハードウェアから強いられる制約によって制限されます。ストリーミング SIMD 拡張命令の場合、ベクトル・メモリ演算は、16 バイトにアライメントされたメモリ参照を優先させて、ストライド -1 でのアクセスに制限されます。すなわち、理論的にループがベクトル化可能なものとしてコンパイラが認識していても、別種のターゲット・アーキテクチャにベクトル化されない可能性があることを意味します。

ループ構造には、ベクトル化コンパイラによる自動並列処理を妨げる多数の様式上の問題があります。ループ本体内でのキーワード、演算子、データ参照、およびメモリ演算が複雑に合わさって、あいまいさが大きくなります。

しかし、これらの制限を理解し、診断メッセージの解釈方法を知っていれば、プログラムに修正を加えることによって、従来からの難題を解決し、最適なベクトル化を行えるようになります。以下の節では、ループ構造に関するベクトライザの機能と制約について要約します。

プログラミングにおける主要ガイドライン

以下に示すガイドライン、制約事項、関数例を検討して、作成したコードをチェックし、コンパイラが最適にベクトル化する際の障害となるあいまいさを取り除いてください。

- ループ本体に関するガイドライン
  - 一直線に配列されたコード (単一の基本ブロック) を使用します。
  - ベクトル・データだけを使用します。すなわち、代入文の右辺には、配列と不変式を使用します。配列参照は代入文の左辺にも使用できます。
  - 代入文だけを使用します。

- ループ本体での使用を避けるべきもの
  - 関数呼び出し
  - ベクトル化不能な演算
  - 同一ループ内でのベクトル化可能なデータ型の混在
  - 多種類の SIMD 演算
  - 関係演算子または ?: 演算子
  - データに依存したループ終了条件

ループの構成要素

ループには、一般的な for や while-do、または repeat-until など、あるいは goto やラベルを使用できます。ただし、ベクトル化の対象となる入り口と出口が 1 つずつなければなりません。

例 14-9 ループの構成要素の使用例

正しい使い方	間違った使い方

`for (i=0; i<n; i++) {`
`  ...`
`  }`
	`while (i<n) {`
	`  ...`
	`  if (cond) break;`
	`/* 2nd exit */`
	`  …`
	`}`

ループ本体の制御フロー

ループの本体は、一直線に配列されたコード、つまり分岐のないコードでなければなりません。

例 14-10　ループ本体の制御フロー

正しい使い方

```
do {
    /* body is a single basic block */
    a[i] = b[i] * x
    b[i] = c[i] + sqrt(d[i]);
    --count;
  } while (count != 0);
```

間違った使い方

```
while (i<n) {
    /* if-branch inside body of loop */
    a[i] = b[i] * c[i];
    if (a[i] < 0.0) {
    a[i] = 0.0;
    }
}
```

ループの終了条件

ループの終了条件によって、ループを実行する反復回数が決まります。例えば、for ループでは、固定のインデックスによって反復回数が決まります。ループの反復はカウントできるものでなければなりません。すなわち、反復の回数は、次のいずれかで表す必要があります。

- 定数
- 整変数の線形関数
- ループ不変項

終了条件が演算に依存するループは、カウントできません。

例 14-11　　ループの使い方の比較

カウント可能なループによる正しい使い方

```
count = N; /* exit condition specified by "N - lb + 1" */
...
do {
a[i] = b[i] * x
b[i] = c[i] + sqrt(d[i]);
--count;
} while (count != lb); /* 1b is not defined within loop */
```

カウント可能なループによる正しい使い方

```
 /* exit condition is "(n-m+2)/2" */
       i = 0;
       for (l=m; l<n; l+=2) {
       a[i] = b[i] * x
       b[i] = c[i] + sqrt(d[i]);
       ++i;
       }
```

カウント不能なループによる間違った使い方

```
i = 0;
/* iterations dependent on a[i] */
while (a[i] > 0.0) {
      a[i] = b[i] * c[i];
      ++i;
}
```

ストリップマイニングとクリーンアップ

コンパイラは自動的にループをストリップマイニングし、クリーンアップ・ループを生成します。これは、ループのアンロールが不要であることを意味します。

例 14-12 ストリップマイニングとクリーンアップ・ループ

```
i = 0;
while (i<n) {
      a[i] = b[i] + c[i]; /* Code Clean-up Loop. */
      ++i;
}
/* The vectorizer will generate the following two loops. */
i = 0;
while (i < (n - n%4)) {
a[i] = b[i] + c[i]; /* Vector Loop. */
++i;
}
while (i < n) {
      a[i] = b[i] + c[i];/* Scalar clean-up loop. */
      ++i;
}
```

ループ本体の文

ベクトル化可能な演算は、浮動小数点データと整数データとで異なります。

浮動小数点配列の演算

ループ本体の文には、`float` 型の演算 (通常は配列) を使用できます。算術演算は、加算、減算、乗算、除算、および否定に制限されます。なお、浮動小数点からの変換、または浮動小数点への変換はできません。`double` 型の演算もできません。

整数配列の演算

ループ本体の文には、char 型、unsigned char 型、short 型、unsigned short 型、int 型、および unsigned int 型の演算 (これも、通常は配列) を使用できます。算術演算は、加算、減算、ビット単位の論理積 (AND)、論理和 (OR)、および排他的論理和 (XOR) 演算に制限されます。

その他の整数演算

整数の帰納変数を除き、データ型を混在させて使用することはできません。

その他のデータ型

前述の浮動小数点演算と整数演算以外は使用できません。特に、特殊データ型の __m64 および __m128 については、ベクトル化不能なことに注意してください。

関数呼び出しは不可

ループの本体には、関数呼び出しは使用できません。Intel C/C++ コンパイラの現バージョンでは、数値演算用の組み込み関数 sqrt および fabs のベクトル化はサポートしていません。ストリーミング SIMD 拡張命令の組み込み関数 (_mm_add_ps) も使用できません。

ベクトル化可能なデータ参照

いかなるデータ参照においても、配列要素またはポインタの参照として、ベクトル化を妨げるような潜在的な依存関係や別名の制約がないことを確認しなければなりません。わかりやすく言うと、ある反復で使用される式が、その前の反復時に計算された値に依存するものであってはならず、ポインタ変数も別々の場所を指していなければなりません。想定される依存関係を無視するようにコンパイラに指示するには、ivdep プラグマと restrict キーワードを使用します。「データのアライメント」の項に示した例も参照してください。

配列

配列参照がストライド -1(4 バイト) またはループ不変であれば、ループ内のベクトル化可能なデータを配列要素を使用するものとして表すことができます。ストライド -1 での参照は、デフォルトではベクトル化されません。このデフォルトを無効にするには、vector プラグマを使用します。

ポインタ ベクトル化可能なデータはポインタを使って表すこともでき、配列要素を使用する場合と同じ制約が加えられます。すなわち、参照がストライド -1 またはループ不変であることが条件です。ストライド単位なしでの整数の参照はサポートされていません。

不変式 ベクトル化可能なデータには、変数または数値定数として、式の右辺にループ不変の参照を含めることもできます。例 14-13 に示すループは、ベクトル化されます。

例 14-13 ベクトル化可能なループ不変の参照

```
for (i=0; i<n; i++) {
  a[i] = b[i] * 3.14f + c[j];
}
```

データのアライメント

ベクトル化可能な float の型データが確実にアライメントされていれば、コンパイラはアライメントに従った命令を生成します。これは、局所的に宣言されたデータ、およびアライメント declspec で宣言されたデータに該当します。データ・アライメントが確定されない場合は、「declspec によるアライメント」の節で説明したように、プラグマまたはコマンド・ライン・スイッチによって無効にされない限り、アライメントに従わない参照が使用されます。

ベクトル化準拠のコード作成における一般的なエラー

コードをベクトル化可能なものにするには、一般に、ループに何らかの変更を加える必要があります。ただし、ベクトル化を可能にするために必要な部分だけを変更し、その他の部分は変更してはいけません。特に、以下のコマンドについては、変更してはなりません。

- ループのアンロール。これはコンパイラによって自動的に行われます。
- 本体の中で複数の文で構成されている 1 つのループを複数の単文ループに解体する。
- 手動による EMMS の呼び出しの挿入。例えば、組み込み関数 _m_empty を使用して、ベクトル化されるループの後に EMMS を挿入することは、デフォルトによってコンパイラが行うようになっています。このデフォルトを無効にするには、-Qvec_emms を使用します。この章に前述した「-Qvec_emms[-]」の項を参照してください。

プログラミング例

この節では、ベクトル・プログラミングの一般的な問題について、簡単な例を挙げていくつか紹介します。

引数のエイリアシング：ベクトル・コピー

例 14-14 には、ベクトルのコピー操作を行うループが示されていますが、コンパイラが dest[i] と src[i] を確定できないため、ベクトル化は行われません。

例 14-14 相違を確定できないためベクトル化不能なコピー操作

```
void vec_copy(float *dest, float *src, int len){
    int i;
    for (i=0; i<len; i++)
      dest[i] = src[i];
}
```

restrict は、ポインタが別個のオブジェクトを参照していることを示すキーワードです。したがって、このキーワードを使用すれば、コンパイラはベクトル化を行うことができます。

例 14-15 restrict を使用したベクトル化可能な相違の確定

```
void vec_copy(float restrict *dest, float restrict *src, int len){
       int i;
       for (i=0; i<len; i++)
             dest[i] = src[i];
}
```

データのアライメント：2 つの例

データのアライメント例

例 14-16 に示す関数には、アライメントに従わないメモリ命令を使った場合のみベクトル化を行うループが含まれています。これは、コンパイラが確定することも、16 バイトのアライメントを強制することもできない大域変数が使用されているためです。

例 14-16　大域変数のためにアライメントされないループ

```
float z[N], a[N], y[N], x;
void f(void)
{
  for (i=0; i<N; i++) {
    a[i] = a[i] * x + y[i];
   }
}
```

ループを局所的に処理できるようにするには、例 14-17 のように、declspec アライメントを使用します。

例 14-17　declspec によるループのアライメント

```
__declspec(align(16)) float z[N], a[N], y[N], x;
void f(void)
{
  for (i=0; i<N; i++) {
    a[i] = a[i] * x + y[i];
   }
}
```

例 14-18 に示すループも、アライメントに従わないメモリ命令を使った場合のみベクトル化を実行します。この場合、コンパイラは局所配列をアライメントすることはできても、コンパイル時には 1b がわからないため、正しいアライメントを判断することができません。

例 14-18　コンパイル時に未確認変数値があるためにアライメントされないループ

```
void f(int lb){
  float z2[N], a2[N], y2[N], x2;
  ...
  for (i=lb; i<N; i++) {
    a2[i] = a2[i] * x2 + y2[i];
}
   ...
}
```

lb が 4 の倍数だとわかっている場合は、例 14-19 に示すように、#pragma vector aligned を使用してループをアライメントすることができます。

例 14-19　変数を 4 の倍数として代入したことによるアライメント

```
void f(int lb)
{
  float z2[N], a2[N], y2[N], x2;
  ...
  assert(lb%4==0);
  #pragma vector aligned
  for (i=lb; i<N; i++) {
    a2[i] = a2[i] * x2 + y2[i];
  }
  ...
}
```

代入を使用すると、制約条件の 1b が 4 の倍数であるかどうかがチェックされます。

データの依存関係

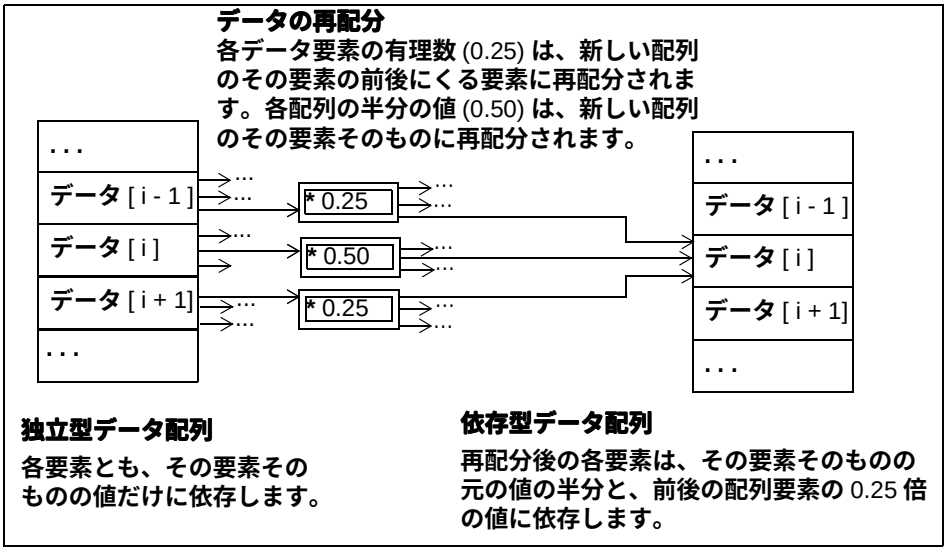
例 14-20 は、データの依存関係を表すコードを示しています。配列の各要素は、それ自体の関数と隣り合う 2 つの要素に変更されます。

例 14-20　データ依存関係を持つループ

```
float data[N];
int i;
  for (i=1; i<N-1; i++) {
    data[i] = data[i-1]*0.25 + data[i]*0.5 + data[i+1]*0.25
}
```

図 14-1 は、例 14-20 の関数が、どのようにデータの依存関係を表しているのかを示したものです。

図 14-1　データの依存関係の再配分



例 14-20 に示すループはベクトル化されません。これは、現在の要素 data[i] への書き込みが直前の要素 data[i-1] の使用に依存しており、この要素 data[i-1] が、その前の反復時に既に書き込まれて変更されているためです。

このことは、例 14-21 に示すように、配列のアクセス・パターンの最初の 2 回の反復を見ればわかります。

例 14-21 データ依存型のベクトル化のパターン

ベクトル化不能なアクセス・パターン

```
i=1:READ data[0]
    READ data[1]
    READ data[2]
  WRITE data[1]

i=2:READ data[1]
    READ data[2]
    READ data[3]
  WRITE data[2]
```

ベクトル化可能なアクセス・パターン

```
i=1,i=2:READ data[0]
        READ data[1]
        READ data[2]
        READ data[1]
        READ data[2]
        READ data[3]
        READ data[2]
      WRITE data[1]
      WRITE data[2]
```

「ベクトル化不能なアクセス・パターン」の例に示すループの通常のシーケンスでは、2 回目の反復時に読み込まれる data[1] の値が、最初の反復時に書き込まれます。

ベクトル化の場合、「ベクトル化可能なアクセス・パターン」の例に示すように、反復は並列的に行われなければなりません。この例では、data[1] の読み込み値は元の値であり、更新後の値ではありません。そこで、例 14-22 に示すように、コピー操作が後に続く一時配列を使用して、ループをベクトル化可能な形式で書き直したいと思う場合が出てきます。

例 14-22　データは独立型であってもベクトル化不能なループ

```
float data[N], newdata[N];
int i;
  for (i=1; i<N-1; i++) {
    newdata[i] = data[i-1] * 0.25 + data[i]*0.5 + data[i+1]*0.25
  }
  for (i=1; i<N-1; i++) {
      data[i] = newdata[i];
  }
```

上の例の 2 つのループはどちらもベクトル化されますが、元のループのセマンティクスを変更しているため、この演算の結果は元の演算と違う結果になります。

ループ交換と添字：マトリックス積

マトリックス積は、一般に、例 14-23 のように記述されます。

例 14-23　一般的なマトリックス積

```
for (i=0; i<N; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

b[k][j] を使用するのはストライド -1 での参照ではないため、通常はベクトル化できません。左辺で c[i][j] を使用することも、ベクトル化が回避される要因となります。左辺には不変の参照を使用できないためです。しかし、ループを交換すると、例 14-24 に示すように、すべての参照がストライド -1 となります。

例 14-24 ストライド -1 でのマトリックス積

```
for (i=0; i<N; i++) {
  for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

参考文献

以下の文献は、ベクトル化の基本的な用語および技術を理解する上で参考になります。

- High Performance Computing』第 2 版、Kevin Dowd 著、O'Reilly and Associates、1998、ISBN 156592312X
- 『インテル・アーキテクチャ最適化マニュアル』インテル株式会社、資料番号 730795J

コンパイラの制限

A

表 A-1 は、コンパイラが処理できる各項目のサイズまたは数です。表に示されている値はすべてテストされた値です。実際の値は示されている値よりも大きい場合があります。

表 A-1　コンパイラの制限

項目	テストされた値
制御構造の入れ子 (ブロックの入れ子)	512
条件付きコンパイルの入れ子	512
宣言子の変更子	512
括弧の入れ子レベル	512
有効文字数、内部識別子	2048
外部識別子名の長さ	64K バイト
外部識別子 / ファイルの数	512
単一ブロックの識別子の数	2048
同時に定義されるマクロの数	128K バイト
関数呼び出しのパラメータの数	512
マクロごとのパラメータの数	128
文字列の文字数	128K バイト
オブジェクトのバイト	512K バイト
インクルード・ファイルの入れ子の深さ	512
switch の case ラベル	32K バイト
1 つの構造体または共用体のメンバー	32K バイト
1 つの列挙内の列挙定数	8192
構造体の入れ子のレベル	320

実験的なパフォーマンス調整

B

この付録では、プロシージャ間の最適化を行いながら、特定のアプリケーションについてコンパイラの最適化を調整する方法について説明します。

注： *ここで説明するサブオプションは、高度な最適化を実験しようとする開発者を対象としています。これらの高度な最適化を行った場合、予期しない結果が生じることがあるので注意してください。したがって、このマニュアルでは、これらの高度な最適化の基本的な機能についてのみ説明しています。*

最適化のキーワード (-Qoption,*c*,*optlist*)

`-Qoption` オプションは適用可能なキーワードとともに入力して、特定のインライン展開およびループの最適化を選択します。このオプションは、`-Qmem` または `-Qip` とともに次のように入力します。

`-Qip [-Qoption,`*`c`*`,`*`optlist`*`]`

c	C コンパイラ・ツールを示す指定子です。
optlist	指定されたレベルに適用可能な最適化のいずれかです。

例えば、次のように入力すると、レジスタの引数の受け渡しは無効になります。

```
prompt> icl -Qip -Qoption,c,ip_args_in_regs=0 a.cpp
```

次の項では、`-Qip` および `-Qoption` とともに使用するキーワードについて定義します。

プロシージャ間の最適化

-Qoption 修飾子を使用せずに -Qip を指定すると、コンパイラは関数をインライン展開し、定数の引数を伝達し、引数をレジスタに渡し、モジュール・レベルの静的変数を監視します。このようなプロシージャ間の最適化を調整するには、以下の -Qoption のキーワードを使用します。

ip_args_in_regs=0	レジスタへの引数の引き渡しを無効にします。デフォルトの場合、外部関数は局所的に呼び出された場合レジスタに引数を渡すことができます。通常、関数のアドレスは渡されず、関数は引数の変数番号を使用しないので、静的関数だけが引数をレジスタに渡すことができます。
ip_inline_max_calls=*N*	インライン・ヒューリスティック (-Qinl_heur) オプションとともに使用します。このオプションは、インライン化する呼び出しサイトのデフォルトの数を変更します。*N* 個の呼び出しサイトがインライン化されるのは、その数の呼び出しサイトがインライン化の最低の基準を満たしている場合だけです。詳細については、「インライン・ヒューリスティックを使用する (-Qinl_heur n)」および「関数のインライン展開の基準」を参照してください。
ip_inline_max_stats=*n*	インライン展開される関数で使用できる中間言語文と式の数を設定します。数値 n は正の整数です。中間言語文の数は、通常、ソース言語の文の実際の数を超えています。デフォルトの最大数は 230 に設定されています。

`ip_inline_max_total_stats=n` インライン展開される各関数について、中間言語の文と式の最大増加数を設定します。数値 n は正の整数です。デフォルトの場合、各関数は最大 2000 の文に展開できます。

複数ファイルの IPO の効果を分析する (-Qipo_c, -Qipo_S)

`-Qipo_c` および `-Qipo_S` オプションは、複数ファイルの IPO の効果を分析したり、完全なプログラムを構成しないモジュール間で複数ファイルの IPO を試す場合に便利です。

`-Qipo_c` オプションは、ファイル間で最適化を行い、オブジェクト・ファイルを生成するために使用します。このオプションは、`-Qipo` で説明した最適化を実行しますが、最適化されたオブジェクト・ファイルを生成し、最後のリンクの前で停止します。このファイルのデフォルトの名前は `ipo_out.obj` です。次の例のように、`-Fe` オプションを使用して他の名前を指定することもできます。

```
prompt> icl -G6 -Qipo_c -Fefilename a.cpp b.cpp c.cpp
```

`-Qipo_S` オプションは、ファイル間で最適化を行い、アセンブリ・ファイルを生成するために使用します。このオプションは、`-Qipo` で説明した最適化を実行しますが、最適化されたアセンブリ・ファイルを生成し、最後のリンクの前で停止します。このファイルのデフォルトの名前は `ipo_out.asm` です。次の例のように、`-Fe` オプションを使用して他の名前を指定することもできます。

```
prompt> icl -G6 -Qipo_S -Fefilename a.cpp b.cpp c.cpp
```

インライン・ヒューリスティックを使用する (-Qinl_heur *n*)

`-Qinl_heur n` オプションは、最も頻繁に使用される呼び出しサイトをインライン化するために使用します。このオプションを指定すると、呼び出しサイトはリーフ・ルーチンとして処理されます。インライン化およびインライン化の最低条件についての詳細は、第 5 章の「ユーザ関数のインライン展開の制御 (-Obn、-Qip_no_inlining)」を参照してください。

`-Qinl_heur n` オプションは、`-Qip` または `-Qipo` オプションとともに使用した場合にのみ有効です。*n* の値は、`0`、`1`、`2`、`3` のいずれかです。

注： *インライン・ヒューリスティック* 1 *から* 3 *はすべてプロファイル・ガイド付きです。プロファイル情報がない場合、コンパイラはデフォルトのインライン・ヒューリスティック (すなわち、n=0) を使用し、警告を出します。プロファイル情報のプロシージャについては、第 5 章の「プロファイルによる最適化 (PGO)：3 つのフェーズ」を参照してください。*

`-Qinl_heur` *n* オプションには、4 つのバリエーションがあります。

`-Qinl_heur 0` デフォルトのインライン・ヒューリスティックです。このヒューリスティックはプロファイル情報の有無に関係なく機能します。

`-Qinl_heur 1` *N* 個の最も頻繁に実行される呼び出しサイトをインライン化します。

`-Qinl_heur 2` *N* 個の最も頻繁に実行される呼び出しサイトをインライン化します。各インライン化の後で、実行回数が概算され、頻繁に実行される呼び出しサイトのリストが更新されます。

`-Qinl_heur 3` 呼び出される側のサイズによって概算された *N* 個の最も頻繁に実行される呼び出しサイトをインライン化します。各インライン化の後で、実行回数が概算され、頻繁に実行される呼び出しサイトのリストが更新されます。

デフォルトでは *N*=100 です。この値は、`-Qoption,c,-ip_inline_max_calls=` *N* オプションによって変更できます。*N* 個の呼び出しサイトがインライン化されるのは、その数の呼び出しサイトがインライン化の最低の基準を満たしている場合だけです。

関数のインライン展開の基準

インライン化の対象とするルーチンは、一定の最低基準を満たしていなければなりません。呼び出しサイト、呼び出し側、および呼び出される側が満足しなければならない基準があります。呼び出しサイトは、インライン化される関数の呼び出しのサイトです。呼び出し側は、呼び出しサイトを含む関数です。呼び出される側は、呼び出されてインライン化される関数です。

呼び出しサイトの最低の基準は次のとおりです。

- 実際の引数の数は、呼び出される側の仮引数の数と一致していなければなりません。

- 戻り値の数は、呼び出される数と一致していなければなりません。
- 実際の引数と仮引数のデータ型は互換性がなければなりません。
- 多言語のインライン化は実行できません。呼び出し側と呼び出される側は同じソース言語で記述されていなければなりません 。

呼び出し側の最低の基準は次のとおりです。

- 呼び出し側にインライン化されるすべての呼び出しサイトから、呼び出し側にインライン化される中間文は最大 2000 です。この値は次のオプションを使用して変更できます。
  `-Qoption,c,-ip_inline_max_total_stats=` *`new value`*
- 関数が `static` として宣言されている場合は呼び出されなければなりません。それ以外の場合は、削除されます。

呼び出される側の最低の基準は次のとおりです。

- 名前によって頻繁に使用されないとはみなされるわけではありません。名前に次のサブ文字列を含むルーチンはインライン展開されません。`abort`、`allca`、`denied`、`err`、`exit`、`fail`、`fatal`、`ault`、`halt`、`init`、`interrupt`、`invalid`、`quit`、`rare`、`stop`、`timeout`、`trace`、`trap`、および `warn` などです。

これらの基準が満たされると、コンパイラは、インライン展開するとプログラムのパフォーマンスに最も効果のあるルーチンを選択します。これは、以下のデフォルトのヒューリスティックを使用して行われます。プロファイルによる最適化を使用する場合は、このほかにも多くのヒューリスティックが使用されます (詳細については第 5 章の「プロファイルによる最適化 (PGO)：3 つのフェーズ」、およびこの章に前述した「インライン・ヒューリスティックを使用する (-Qinl_heur n)」を参照してください)。

- デフォルトのヒューリスティックは、ループ内の呼び出しサイトまたはループを含む関数の呼び出しに重点を置きます。
- プロファイル情報が使用できる場合は、最も頻繁に実行される呼び出しサイトに重点が置かれます。また、デフォルトのインライン・ヒューリスティックでは、中間文が 230 またはオプション `-Qoption,c,-ip_inline_max_stats` で指定された数を超える関数のインライン化を実行できません。
- デフォルトのインライン・ヒューリスティックは、直接的な再帰呼び出しが検出されるとインライン化を停止します。
- デフォルトのヒューリスティックは、インライン化の最低基準を満たす非常に小さな関数を常にインライン化します。デフォルトでは、中間文の数が 15 以下の関数はインライン化されます。この制限は、オプション `-Qoption,c,-ip_inline_min_stats` で変更できます。

索引

記号

数字

A

C

D

E

F

H

I

L

M

N

O

P

お

か

き

け

こ

さ

ひ

ふ

ほ

ま

め

も

ら

り

る