IBM XL C/C++ for Linux, V13.1

コンパイラー・リファレンス

バージョン 13.1

SA88-5388-00
(英文原典：SC27-4249-00)

IBM XL C/C++ for Linux, V13.1

コンパイラー・リファレンス

バージョン 13.1

SA88-5388-00
(英文原典：SC27-4249-00)

お願い

本書および本書で紹介する製品をご使用になる前に、 663 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM XL C/C++ for Linux, V13.1 (プログラム 5765-J08; 5725-C73)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。正しいレベルの製品をご使用になるようお確かめください。

原典： SC27-4249-00
IBM XL C/C++ for Linux, V13.1
Compiler Reference
Version 13.1
First edition

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

目次

本書について

本書は、IBM® XL C/C++ for Linux, V13.1 コンパイラーに関する参照情報です。本書には C および C++ で作成されたアプリケーションのコンパイルおよびリンクについての情報が記載されていますが、本書は主に、コンパイラー・コマンド行オプション、プラグマ・ディレクティブ、事前定義マクロ、組み込み関数、環境変数、エラー・メッセージ、および戻りコードの参照資料として作成されています。

本書の対象読者

本書の対象読者は、UNIX オペレーティング・システムの XL C/C++ コンパイラーまたは他のコマンド行コンパイラーに習熟している C または C++ の開発経験者です。C または C++ プログラミング言語を熟知しており、オペレーティング・システムのコマンドについても基礎的な知識があることを前提としています。本書は参照ガイドとして作成されていますが、XL C/C++ の経験がないプログラマーでも本書を使用することにより、XL C/C++ コンパイラーに固有の機能やフィーチャーに関する情報を見つけることができます。

本書の使用方法

特に明記されていない限り、本書のテキストはすべて C 言語と C++ 言語の両方に関連します。言語間で違いがある場合は、 x ページの『規則』で説明されているとおり、テキストとアイコンの限定により違いが示されています。

このマニュアル全体を通して、**xlc** および **xlc++** コマンド呼び出しは、コンパイラーのアクションを記述するため使用されます。ただし、特定環境に必要な場合は、他の形式のコンパイラー呼び出しコマンドに置き換えることができ、特に指定されていない限り、コンパイラー・オプションの使用法はそのままとなります。

本書ではコンパイラー環境の構成、および XL C/C++ コンパイラーを使用した C または C++ アプリケーションのコンパイルおよびリンクに関するトピックを扱っていますが、以下のトピックは含まれていません。

- コンパイラーのインストール: XL C/C++ のインストールに関する情報については、「*XL C/C++ インストール・ガイド*」を参照してください。
- C または C++ プログラミング言語: C または C++ プログラミング言語の構文、セマンティクス、および IBM のインプリメンテーションについては、「*XL C/C++ ランゲージ・リファレンス*」を参照してください。
- プログラミング・トピック: プログラムの移植性と最適化にフォーカスし、XL C/C++ を使用したアプリケーションの開発に関する詳細については、「*XL C/C++ 最適化およびプログラミング・ガイド*」を参照してください。

本書の構成

1 ページの『第 1 章 アプリケーションのコンパイルおよびリンク』では、コンパイラー、プリプロセッサー、およびリンカーなどの呼び出しを含むコンパイル・タスク、入力ファイルおよび出力ファイルのタイプ、組み込みファイルのパス名およびディレクトリーの検索シーケンスを設定するためのさまざまな方法、コンパイラー・オプションの指定および矛盾するコンパイラー・オプションの解決のためのさまざまな方法、コンパイラー・ユーティリティー**gxlc** および **gxlc++** を使用した GNU C/C++ コンパイラー・オプションの再利用方法、およびコンパイラー・リストとメッセージに関連するトピックが説明されています。

27 ページの『第 2 章 コンパイラーのデフォルトの構成』 では、環境変数の設定などのデフォルト・コンパイルの設定、構成ファイルのカスタマイズ、および **gxlc** と **gxlc++** オプション・マッピングのカスタマイズに関するトピックが説明されています。

81 ページの『第 4 章 コンパイラー・オプション参照』では最初に機能カテゴリーごとのオプションの要約が記載されています。これによって、機能ごとにオプションを検索したりそのオプションにリンクすることができます。また、アルファベット順にソートされたそれぞれのコンパイラー・オプションの説明が記載されています。

403 ページの『第 5 章 コンパイラー・プラグマ参照』では最初に機能カテゴリーごとのプラグマ・ディレクティブの要約が記載されています。これによって、機能ごとにプラグマを検索したりそのプラグマにリンクすることができます。アルファベット順にソートされたプラグマと OpenMP ディレクティブのそれぞれの説明が記載されています。

481 ページの『第 6 章 コンパイラーの事前定義マクロ』ではカテゴリーごとのコンパイラー・マクロのリストが記載されています。

499 ページの『第 7 章 コンパイラー組み込み関数』では、機能ごとにカテゴリー化された Power® アーキテクチャーの XL C/C++ 組み込み関数がそれぞれ説明されています。

規則

活字の規則

以下の表では、IBM XL C/C++ for Linux, V13.1 の資料で使用されている活字の規則について説明します。

表 1. 活字の規則

書体	意味	例
太字	小文字のコマンド、実行可能ファイル名、コンパイラー・オプション、およびディレクティブ。	コンパイラーには、さまざまな C/C++ 言語レベルおよびコンパイル環境をサポートするために、**xlc** と **xlC (xlc++)** という基本呼び出しコマンドとその他のいくつかのコンパイラー呼び出しコマンドが備わっています。

表 1. 活字の規則 (続き)

書体	意味	例
イタリック	パラメーターまたは変数。実際の名前と値はユーザーによって提供されます。イタリックは新規用語の導入にも使用されます。	要求された *size* よりも大きいものを戻す場合には、 *size* パラメーターの更新を確認してください。
下線	コンパイラー・オプションまたはディレクティブのパラメーターのデフォルト設定。	nomaf | maf
モノスペース	プログラミング・キーワードおよびライブラリー関数、コンパイラー・ビルトイン、プログラム・コードの例、コマンド・ストリング、またはユーザー定義の名前。	myprogram.c をコンパイルおよび最適化するには、`xlc myprogram.c -O3` と入力します。

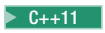
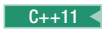
限定を示すエレメント (アイコン)

本書に記述されているフィーチャーの大半は、C と C++ 言語の両方に適用されます。あるフィーチャーが 1 つの言語に限定される場合、あるいは言語間で機能が異なる場合の言語エレメントの説明では、以下のように、アイコンを使用してテキストのセグメントを説明します。

表 2. 限定を示すエレメント

修飾子/アイコン	意味
C のみ、または C のみの始まり C C C のみの終わり	このテキストは C 言語のみでサポートされているフィーチャーを記述しています。または、C 言語に特定の振る舞いを記述しています。
C++ のみ、または C++ のみの始まり C++ C++ C++ のみの終わり	このテキストは C ++ 言語のみでサポートされているフィーチャーを記述しています。または、C++ 言語に特定の振る舞いを記述しています。
IBM の拡張機能、または IBM の拡張機能の始まり IBM IBM IBM の拡張機能の終わり	テキストは、標準の言語仕様に対する IBM 拡張機能であるフィーチャーを説明します。

表 2. 限定を示すエレメント (続き)

修飾子/アイコン	意味
C11、または C11 の始まり   C11 の終わり	このテキストは、C11 の一部として標準 C に導入されるフィーチャーを記述しています。
C++11、または C++11 の始まり   C++11 の終わり	このテキストは、C++11 の一部として標準 C++ に導入されるフィーチャーを記述しています。

構文図

本書中では、ダイアグラムは XL C/C++ 構文を図示します。このセクションは、これらのダイアグラムの解釈と使用に役立ちます。

- 構文図は線のパスに沿って、左から右、上から下へと読んでいきます。

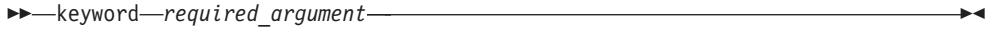
  ▸▸── 記号は、コマンド、ディレクティブ、またはステートメントの開始を示します。

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

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

  ──▸◂ 記号は、コマンド、ディレクティブ、またはステートメントの終了を示します。

  完結したコマンド、ディレクティブ、またはステートメント以外の構文単位の図であるフラグメントは、|── 記号で始まり ──| 記号で終わります。
- 必須項目は、次のように横線 (メインパス) 上に表示されます。



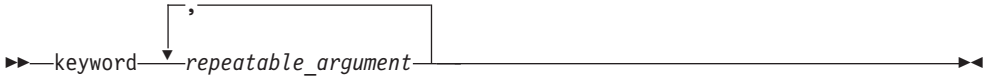
- オプショナル項目は、次のようにメインパスの下側に表示されます。

- 2 つ以上の項目から選択できる場合は、縦に重ねて表示されます。

  項目の中から 1 つを*選択しなければならない* 場合は、スタックの 1 つの項目がメインパスに表示されます。

項目の 1 つを選択することがオプショナルの場合は、スタック全体がメインパスの下に表示されます。

- 主線の上にある左に戻る矢印 (反復矢印) は、スタックされた項目から複数個選択できること、あるいは単一の項目を繰り返すことができることを示します。区切り文字も示されます (それがブランク以外の場合)。

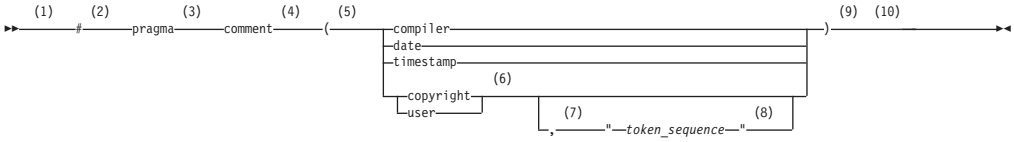


- デフォルトの項目はメインパスの上に表示されます。

- キーワードは、イタリックでない文字で示され、示されているとおりに入力する必要があります。
- 変数は、イタリック体の小文字で示されます。変数は、ユーザー指定の名前や値を表します。
- 句読記号、括弧、算術演算子、または他のそのような記号が表示されている場合は、構文の一部として入力する必要があります。

構文図の例

以下の構文図の例は、**#pragma comment** ディレクティブの構文を示したものです。



注:

1 これが構文図の始まりです。

2 記号 # が最初に示される必要があります。

3 キーワード pragma が # 記号に続いて示される必要があります。

4 プラグマの名前 comment が、キーワード pragma に続いて示される必要があります。

5 左括弧を指定する必要があります。

6 コメント・タイプは、示されているタイプ compiler、date、timestamp、copyright、または user の 1 つとしてのみ入力する必要があります。

7 コメント・タイプ `copyright` または `user` とオプショナル文字ストリングとの間にコンマを 1 つ入れる必要があります。

8 コンマの後に文字ストリングが続いている必要があります。文字ストリングは二重引用符で囲む必要があります。

9 右括弧が必要です。

10 これが、構文図の終わりです。

以下の **#pragma comment** ディレクティブの例は、上記の図に従って、構文的に正しいものです。

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

本書の例

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

インストール情報の例は、*例* または*基本例* としてラベル付けられています。*基本例* は、基本インストールまたはデフォルト・インストール時に実行する手順の説明用です。例はほとんど変更せずに、または全く変更せずに使用できます。

関連情報

以下のセクションでは、XL C/C++ に関連した情報を説明します。

IBM XL C/C++ 情報

XL C/C++ は、以下の形式で製品資料を提供しています。

- README ファイル

  README ファイルには、製品情報に対する変更と訂正も含め、最新の情報が含まれています。 README ファイルは、デフォルトでは XL C/C++ ディレクトリーと、インストール CD のルート・ディレクトリーにあります。

- インストール可能な man ページ

  man ページは製品に準備されているコンパイラー呼び出しとすべてのコマンド行ユーティリティーに対して提供されています。 man ページのインストールおよびアクセスについての指示は、「*IBM XL C/C++ for Linux, V13.1 インストール・ガイド*」に記載されています。

- インフォメーション・センター

  検索機能が完備された HTML ベースの資料は、次の Web サイトで参照できます。http://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.0/com.ibm.compilers.linux.doc/welcome.html

- PDF 文書

  PDF 文書は、デフォルトでは /opt/ibm/xlC/13.1.0/doc/*LANG*/pdf/ ディレクトリーにあります。ここで *LANG* は en_US、zh_CN、 または ja_JP です。 PDF ファ

イルは、以下の Web サイト http://www.ibm.com/support/docview.wss?uid=swg27036675でも入手できます。

以下のファイルは、XL C/C++ 製品資料のフル・セットを構成しています。

表 3. XL C/C++ PDF ファイル

文書タイトル	PDF ファイル名	説明
IBM XL C/C++ for Linux, V13.1 インストール・ガイド, SA88-5404-00	install.pdf	XL C/C++ のインストール方法と基本的なコンパイルおよびプログラム実行のための環境の構成方法に関する情報が含まれています。
IBM XL C/C++ for Linux, V13.1 はじめに, SA88-5392-00	getstart.pdf	XL C/C++ 製品の概要と、環境のセットアップと構成、プログラムのコンパイルとリンク、およびコンパイル・エラーのトラブルシューティングに関する情報が含まれています。
IBM XL C/C++ for Linux, V13.1 コンパイラー・リファレンス, SA88-5388-00	compiler.pdf	さまざまなコンパイラー・オプション、プラグマ、マクロ、環境変数、および組み込み関数 (並列処理に使用されるものを含む) についての情報が含まれます。
IBM XL C/C++ for Linux, V13.1 ランゲージ・リファレンス, SA88-5396-00	langref.pdf	移植性および一般的規格への準拠についての言語拡張機能も含め、 IBM によってサポートされる C および C++ プログラミング言語に関する情報が記載されています。
IBM XL C/C++ for Linux, V13.1 最適化およびプログラミング・ガイド, SA88-5402-00	proguide.pdf	アプリケーションの移植、 Fortran コードによる言語間呼び出し、ライブラリー開発、アプリケーションの最適化および並列処理、および XL C/C++ 高性能ライブラリーなどの高度なプログラミング上のトピックに関する情報が記載されています。

PDF ファイルを読むには、Adobe Reader を使用します。Adobe Reader をお持ちでない場合は、Adobe の Web サイト (http://www.adobe.com) からダウンロードできます (ライセンス条項に従う必要があります)。

IBM Redbooks® 資料、ホワイト・ペーパー、チュートリアル、資料の正誤表、その他の記事など、XL C/C++ に関連する詳細は、次の Web サイトから入手できます。

http://www.ibm.com/support/docview.wss?uid=swg27036675

注: 資料の正誤表は、インフォメーション・センターの英語版にのみ反映されます。

パフォーマンス、生産性、および移植性の向上に関する情報は、 C/C++ café (http://www.ibm.com/software/rational/cafe/community/ccpp) を参照してください。

標準および仕様

XL C/C++ は、以下の標準および仕様をサポートするように設計されています。本情報に含まれているいくつかの機能に関する正確な定義については、これらの標準を参照できます。

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*、別名 *C89*。
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*、別名 *C99*。
- *Information Technology - Programming languages - C, ISO/IEC 9899:2011*、別名 *C11*。(部分サポート)
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*、別名 *C++98*。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*、別名 *標準 C++*。
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*、別名 *C++11*。(部分サポート)
- *Information Technology - Programming languages - Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. このドラフトの技術レポートは、C 標準委員会によって承認されており、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf で入手可能です。
- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. このドラフトの技術レポートは、C 標準委員会に提出されてあり、 http://www.open-std.org/JTC1/SC22/WG21/www/docs/papers/2005/n1836.pdf で入手可能です。
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. ベクトル処理テクノロジーをサポートするための、このベクトル・データ型の仕様はサイト http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf で使用可能です。
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- http://www.openmp.org で使用可能な「*OpenMP Application Program Interface Version 4.0 (部分サポート)*」。

その他の IBM 情報

- *ESSL for AIX® V5.1/ESSL for Linux on POWER V5.1 Guide and Reference* は、Web ページ Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL で入手できます。

その他の情報

- *Using the GNU Compiler Collection* は http://gcc.gnu.org/onlinedocs で入手できます。

テクニカル・サポート

追加のテクニカル・サポートは、XL C/C++ のサポート・ページ (http://www.ibm.com/support/entry/portal/overview/software/rational/xl_c~c++_for_linux) から利用できます。このページには、Technote や他のサポート情報の豊富なセレクションへの検索機能を備えたポータルがあります。

必要なものが見つからない場合には、compinfo@ca.ibm.com に E メールを出して問い合わせることができます (英文でのみ対応)。

XL C/C++ に関する最新の情報に関しては、http://www.ibm.com/software/products/us/en/xlcpp-linux/にある製品情報サイトをご覧ください。

第 1 章 アプリケーションのコンパイルおよびリンク

デフォルトでは、XL C/C++ コンパイラーを呼び出すと、以下のすべての変換フェーズが実行されます。

- プログラム・ソースのプリプロセッシング
- オブジェクト・ファイルへのコンパイルおよびアセンブル
- リンクして実行可能ファイルの作成

これらの異なる変換フェーズは、実際には、コンパイラー・コンポーネントと呼ばれる別々の実行可能プログラムによって実行されます。ただし、コンパイラー・オプションを使用して、プリプロセッシングやアセンブルなどの特定のフェーズのみを実行することができます。その後、コンパイラーを再び呼び出して、最終実行可能ファイルへの中間出力処理を再開できます。

以下の節では、ソース・ファイルとライブラリーのプリプロセス、コンパイル、およびリンクを実行する XL C/C++ コンパイラーの呼び出し方法について説明しています。

コンパイラーの呼び出し

異なる形式の XL C/C++ コンパイラー呼び出しコマンドが異なるレベルの C および C++ 言語をサポートしています。ほとんどの場合は、**xlc** コマンドを使用して C ソース・ファイルをコンパイルし、**xlc++** コマンドを使用して C++ ソース・ファイルをコンパイルします。C と C++ 両方のオブジェクト・ファイルがある場合は、**xlc++** を使用してリンクしてください。

特定の環境で必要な場合は、他の形式のコマンドを使用することができます。 2 ページの表 4 では、異なる基本コマンドがそれぞれの特殊型とともにリストされています。 特殊コマンドについては、 2 ページの表 5 で説明します。

注: 各呼び出しコマンドに対し、コンパイラー構成ファイルでデフォルトのオプション設定が定義され、場合によってはマクロが定義されます。特定の呼び出しによって暗黙指定されるデフォルト情報については、ご使用システムの システムの /opt/ibm/xlC/13.1.0/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* ファイル。例えば、/opt/ibm/xlC/13.1.0/etc/xlc.cfg.sles11.gcc432 または /opt/ibm/xlC/13.1.0/etc/xlc.cfg.rhel6.2.gcc446 です。

表 4. コンパイラー呼び出し

基本呼び出し	説明	同等の特殊呼び出し
xlc	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは、すべての ISO C99 標準機能および大半の IBM 言語拡張機能をサポートします。 この呼び出しは、すべてのアプリケーションに使用することをお勧めします。	xlc_r
c99	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは、すべての ISO C99 言語機能をサポートしますが、IBM 言語拡張機能はサポートしません。C99 規格に厳密に準拠する必要がある場合は、この呼び出しを使用してください。	c99_r
c89	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは、すべての ANSI C89 言語機能をサポートしますが、IBM 言語拡張機能はサポートしません。 C89 規格に厳密に準拠する必要がある場合は、この呼び出しを使用してください。	c89_r
cc	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは、ANSI C 以前、および多くの共通言語拡張機能をサポートします。このコマンドを使用すると、標準 C に準拠しないレガシー・コードをコンパイルできます。	cc_r
gxlc	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは多くの共通の GNU C オプションを受け入れて、それらを XL C オプションの同等のものにマップし、**xlc** を呼び出します。詳しくは、『 12 ページの『gxlc および gxlc++ での GNU C/C++ コンパイラー・オプションの再使用』』を参照してください。	
xlc++, xlC	C++ ソース・ファイル用のコンパイラーを呼び出します。いずれかのソース・ファイルが C++ の場合は、適正なランタイム・ライブラリーとリンクするために、この呼び出しを使用する必要があります。 .c サフィックスのファイルは、**-+** コンパイラー・オプションが使用されていないと想定して、C 言語ソース・コードとしてコンパイルされます。	xlc++_r, xlC_r
gxlc++, gxlC	C++ ファイル用のコンパイラーを呼び出します。このコマンドは多くの共通の GNU C/C++ オプションを受け入れて、それらを XL C/C++ オプションの同等のものにマップし、**xlc++** を呼び出します。詳しくは、『 12 ページの『gxlc および gxlc++ での GNU C/C++ コンパイラー・オプションの再使用』』を参照してください。	

表 5. 特殊呼び出しのサフィックス

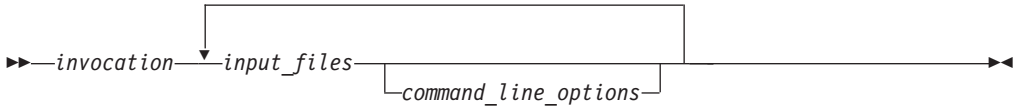
_r サフィックスの呼び出し	**_r** サフィックスを付けた呼び出しはすべて、スレッド・セーフ・コンパイルが可能なため、この形を使用して、マルチスレッドを使用したプログラムのリンクを行うことができます。スレッド化アプリケーションを作成したい場合は、これらのコマンドを使用してください。

関連情報

- 227 ページの『-qlanglvl』

コマンド行構文

コンパイラーは、以下の構文を使用して呼び出します。この構文の *invocation* には、 2 ページの表 4 にリストされている有効な XL C/C++ 呼び出しコマンドを指定します。



コンパイラー呼び出しコマンドのパラメーターには、入力ファイル名、コンパイラー・オプション名、リンカー・オプション名を指定することができます。

多くの場合、プログラムは複数の入力ファイルから構成されます。これらのすべてのソース・ファイルは、コンパイラーを 1 回呼び出すだけで一度にコンパイルすることができます。コンパイラーの 1 回の呼び出しで複数のソース・ファイルをコンパイルすることができますが、1 回の呼び出しに対してコマンド行に指定できるコンパイラー・オプションは 1 セットのみとなります。異なるコマンド行コンパイラー・オプションをそれぞれ指定したい場合は、個別に呼び出す必要があります。

コンパイラー・オプションは、コンパイラー特性の設定、作成されるオブジェクト・コードやコンパイラー出力の指定、いくつかのプリプロセッサー機能の実行など、さまざまな機能を実行します。

デフォルトでは、呼び出しコマンドはコンパイラーとリンカーの*両方を* 呼び出します。呼び出しコマンドは、リンカー・オプションをリンカーに渡します。したがって、呼び出しコマンドでもすべてのリンカー・オプションを受け付けます。リンクしないでコンパイルするには、**-c** コンパイラー・オプションを使用します。 **-c** オプションを指定すると、コンパイルの完了後にコンパイラーが停止します。**-o** オプションを使用して異なるオブジェクト・ファイル名が指定されていない限り、各 *file_name.nnn* 入力ソース・ファイルごとにオブジェクト・ファイル *file_name*.o が出力として作成されます。リンカーは呼び出されません。後でそのオブジェクト・ファイルをリンクするには、**-c** オプションなしでオブジェクト・ファイルを指定して、同じ呼び出しコマンドを使用します。

関連情報
- 『入力ファイルのタイプ』

入力ファイルのタイプ

コンパイラーは、ソース・ファイルを表示される順に処理します。コンパイラーは指定されたソース・ファイルが見つからないと、エラー・メッセージを出し、次に指定されたファイルへ進みます。ただし、リンカーは実行されず、一時オブジェクト・ファイルは除去されます。

コンパイラーはデフォルトで、指定されたすべてのソース・ファイルをプリプロセスしてコンパイルします。通常はこのデフォルトを使用しますが、コンパイラーを使用して、コンパイルなしでソース・ファイルをプリプロセスできます。詳しくは、『 14 ページの『プリプロセッシング』』を参照してください。

以下のタイプのファイルを XL C/C++ コンパイラーに入力することができます。

C および C++ ソース・ファイル

これらは C または C++ ソース・コードを含むファイルです。

C コンパイラーを使用して C 言語ソース・ファイルをコンパイルするには、**-qsourcetype=c** オプションでコンパイルしないのであれば、ソース・ファイルに .c (小文字の c) サフィックスを付ける必要があります。

C++ コンパイラーを使用するには、ソース・ファイルに .C (大文字の C)、.cc、.cp、.cpp、.cxx、.c++ のいずれかのサフィックスを指定する必要があります。ただし、**-+** オプションまたは **-qsourcetype=c++** オプションを使用してコンパイルする場合は、これらのサフィックスを指定する必要はありません。

プリプロセスされたソース・ファイル

プリプロセスされたソース・ファイルは .i サフィックスを含みます (例えば、*file_name*.i)。コンパイラーは、プリプロセス済みのソース・ファイル *file_name*.i を、.c ファイルまたは .C ファイルと同じようにコンパイラーに送り、再度プリプロセスを実行します。プリプロセスされたファイルは、マクロやプリプロセッサー・ディレクティブの検査に役立ちます。

オブジェクト・ファイル

オブジェクト・ファイルは .o サフィックスを含まなければなりません (例えば *file_name*.o)。オブジェクト・ファイル、ライブラリー・ファイル、ストリップされていない実行可能ファイルは、リンカーへの入力として使用できます。コンパイル後、リンカーは実行可能ファイルを作成するために、指定されたすべてのオブジェクト・ファイルをリンクします。

アセンブラー・ファイル

アセンブラー・ファイルは、**-qsourcetype=assembler** オプションでコンパイルしないのであれば、.s のサフィックスを持っていなければなりません (例えば、*file_name*.s)。アセンブラー・ファイルは、オブジェクト・ファイルを作成するためにアセンブルされます。

プリプロセスされていないアセンブラー・ファイル

プリプロセスされていないアセンブラー・ファイルは、.S というサフィックスを指定する必要があります(*file_name*.S など)。ただし、**-qsourcetype=assembler-with-cpp** オプションを使用してコンパイルする場合は、このサフィックスを指定する必要はありません。コンパイラーは、.S 拡張子を含むすべてのソース・ファイルを、プリプロセッシングを必要とするアセンブラー言語のソース・ファイルであるかのようにコンパイルします。

共有ライブラリー・ファイル

共有ライブラリー・ファイルには通常 .a サフィックスが付きます (例えば *file_name*.a) が、.so サフィックスを付けることもできます (例えば、*file_name*.so)。

非ストリップの実行可能ファイル

オペレーティング・システムの **strip** コマンドを使用してストリップされていない 実行可能ファイルおよびリンク・フォーマット (ELF) ファイルは、コンパイラーへの入力として使用できます。

関連情報
- 機能カテゴリー別のオプションの要約: 入力制御

出力ファイルのタイプ

XL C/C++ コンパイラーを呼び出すときに、以下のタイプの出力ファイルを指定することができます。

実行可能ファイル

デフォルトで実行可能ファイルは a.out という名前になります。実行可能ファイルを別の名前にしたい場合は、**-o** *file_name* オプションを呼び出しコマンドに使用してください。このオプションは、*file_name* に指定した名前で実行可能ファイルを作成します。指定する名前には、実行可能ファイルの相対または絶対パス名を使用できます。

オブジェクト・ファイル

-c オプションを指定すると、出力オブジェクト・ファイル *file_name*.o が各入力ファイルに対して生成されます。リンカーは呼び出されず、オブジェクト・ファイルは現行ディレクトリーに入れられます。コンパイルの完了時にすべての処理が停止されます。 **-o** *file_name* オプションを使用して別のサフィックスを指定したり、まったくサフィックスをつけないよう指定しない限り、コンパイラーはオブジェクト・ファイルに .o サフィックスを付けます (例えば *file_name*.o)。

コンパイラーを呼び出すことにより、後でオブジェクト・ファイルをリンクして単一の実行可能ファイルを作成することができます。

共有ライブラリー・ファイル

-qmkshrobj オプションを指定すると、コンパイラーは、すべての入力ファイルに対して単一の共有ライブラリー・ファイルを生成します。**-o** *file_name* オプションを指定して、ファイルに .so サフィックスを付けなければ、コンパイラーは出力ファイルの名前を a.out に設定します。

アセンブラー・ファイル

-S オプションを指定すると、アセンブラー・ファイル *file_name*.s が各入力ファイルに対して生成されます。

次に、アセンブラー・ファイルをオブジェクト・ファイルにアセンブルし、コンパイラーを再び呼び出すことでオブジェクト・ファイルをリンクします。

プリプロセスされたソース・ファイル

-P オプションを指定すると、プリプロセスされたソース・ファイル *file_name*.i が各入力ファイルに対して生成されます。

次に、プリプロセスされたファイルをオブジェクト・ファイルにコンパイルし、コンパイラーを再び呼び出すことでオブジェクト・ファイルをリンクします。

リスト・ファイル

-qlist または **-qsource** などのリスト関連のオプションのいずれかを指定すると、コンパイラー・リスト・ファイル *file_name*.lst が各入力ファイルに対して生成されます。リスト・ファイルは現行ディレクトリーに格納されます。

ターゲット・ファイル

-M オプションまたは **-qmakedep** オプションを指定すると、Make ファイルへの追加に適したターゲット・ファイル *file_name*.d が各入力ファイルに対して生成されます。

関連情報

- 機能カテゴリー別のオプションの要約: 出力制御

コンパイラー・オプションの指定

コンパイラー・オプションは、コンパイラー特性の設定、作成されるオブジェクト・コードやコンパイラー出力の指定、いくつかのプリプロセッサー機能の実行など、さまざまな機能を実行します。コンパイラー・オプションは、次の 1 つ以上の方法で指定することができます。

- コマンド行
- .cfg 拡張子を持つファイルである、カスタム構成ファイル
- ソース・プログラム
- システム環境変数
- Make ファイル

上記の方法で明示的に設定されていないほとんどのコンパイラー・オプションについては、コンパイラーはデフォルトの設定値を想定します。

コンパイラー・オプションを指定した場合、オプションの矛盾や非互換が起きる可能性があります。 XL C/C++ コンパイラーは、それらの矛盾や非互換性の大部分を以下のような一貫性のある方法で解決します。

多くの場合、コンパイラーは以下の順序で、矛盾または非互換のオプションを解決します。

1. ソース・コード内のプラグマ・ステートメントは、コマンド行で指定されたコンパイラー・オプションをオーバーライドする。
2. コマンド行に指定されたコンパイラー・オプションは、構成ファイルで環境変数として指定されたコンパイラー・オプションをオーバーライドする。矛盾したり非互換のコンパイラー・オプションが同じコマンド行コンパイラー呼び出しで指定された場合は、あとから指定されたオプションが優先される。
3. 環境変数として指定されたコンパイラー・オプションは、構成ファイルで指定されたコンパイラー・オプションをオーバーライドする。
4. 構成ファイル内、コマンド行、またはソース・プログラム内で指定されたコンパイラー・オプションは、コンパイラーのデフォルト設定をオーバーライドする。

この優先順序に従わないオプションの矛盾については、10 ページの『矛盾するコンパイラー・オプションの解決』で説明します。

コマンド行でのコンパイラー・オプションの指定

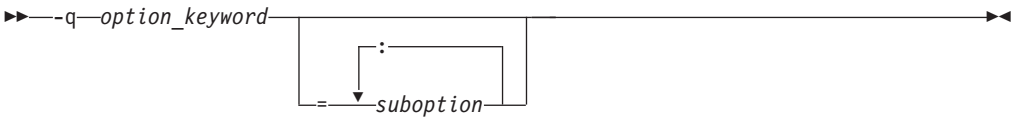
コマンド行で指定されるほとんどのオプションは、オプションのデフォルト設定と、構成ファイルに設定されたオプションをオーバーライドします。 同様に、コマンド行に指定されたほとんどのオプションは順番にプラグマ・ディレクティブによってオーバーライドされ、それによりソース・ファイルの正にその中にコンパイラ

ー・オプションを設定する方法が提供されます。この方式に従わないオプションについては、 10 ページの『矛盾するコンパイラー・オプションの解決』にリストします。

コマンド行オプションには、次の 2 種類があります。

- **-q***option_keyword* (コンパイラー特定)
- フラグ・オプション

-q オプション



-q*option_keyword* 形式のコマンド行オプションは、オン/オフ・スイッチと似ています。ほとんどの **-q** オプションでは、特定のオプションが複数回指定された場合は、そのオプションのうちコマンド行に最後に出現したものがコンパイラーによって使用されるオプションです。例えば、**-qsource** はコンパイラー・リストを作成するためのソース・オプションをオンにし、**-qnosource** はソース・オプションをオフにするためソース・リストは作成されません。 例を以下に示します。

```
xlc -qnosource MyFirstProg.c -qsource MyNewProg.c
```

このように指定すると、`MyNewProg.c` と `MyFirstProg.c` の両方のソース・リストが作成されます。その理由は、最後に指定された **source** オプション (**-qsource**) が優先されるためです。

同一コマンド行に複数の **-q***option_keyword* インスタンスを指定することができますが、これらのインスタンスはブランクで分離する必要があります。オプション・キーワードは、大文字または小文字のいずれかで表示されますが、 **-q** は小文字で指定しなければなりません。 **-q***option_keyword* は、ファイル名の前または後に指定することができます。 例を以下に示します。

```
xlc -qLIST -qfloat=nomaf file.c
xlc file.c -qxref -qsource
```

多くのコンパイラー・オプションは省略することもできます。例えば、**-qopt** を指定すると、**-qoptimize** を指定したことと同等になります。

オプションの中には、サブオプションを持つものがあります。 **-q***option* の次に等号を使用して、これらのサブオプションを指定します。オプションに複数のサブオプションを指定できる場合、コロン (**:**) で、各サブオプションを次のサブオプションから分けなければなりません。 例を以下に示します。

```
xlc -qflag=w:e -qattr=full file.c
```

報告対象となるメッセージの重大度レベルを指定するオプション **-qflag** を使用して、C ソース・ファイル `file.c` をコンパイルします。**-qflag** サブオプション **w** (警告) は、リストで報告される最小レベルの重大度を設定し、サブオプション **e** (エラー) は端末で報告される最小レベルの重大度を設定します。オプション **-qattr** をサブオプション **full** と一緒に指定すると、プログラム内のすべての ID の属性リストが作成されます。

フラグ・オプション

XL C/C++ は、UNIX システムで使用される多くの共通の標準的フラグ・オプションをサポートします。小文字のフラグは、その文字に対応する大文字フラグとは異なります。例えば、**-c** と **-C** は、別々のコンパイラー・オプションです。**-c** は、コンパイラーがプリプロセスとコンパイルのみを行い、リンカーを起動しないことを指定します。一方、**-C** は、ユーザー・コメントの保存を指定する **-P** または **-E** とともに使用することができます。

XL C/C++ は、他のプログラミング・ツールおよびユーティリティー (例えば、**ld** コマンド) に誘導されるフラグもサポートします。コンパイラーはリンク時に、**ld** に誘導されたこれらのフラグを渡します。

フラグ・オプションの中には、フラグの一部を形成する引数を持つものがあります。 例を以下に示します。

```
xlc stem.c -F/home/tools/test3/new.cfg:xlc
```

ここで、new.cfg はカスタム構成ファイルです。

1 つのストリングで引数を取らないフラグを指定することができます。例を以下に示します。

```
xlc -Ocv file.c
```

これは、以下の指定と同じ効果があります。

```
xlc -O -c -v file.c
```

この場合、C ソース・ファイルの file.c を最適化 (**-O**) を使用してコンパイルし、コンパイラーの進行状態 (**-v**) を報告しますが、リンカーを呼び出しません (**-c**)。

引数を取るフラグ・オプションは単一ストリングの一部として指定することができますが、引数を取るフラグは 1 つしか使用できず、そのフラグは最後に指定されるオプションでなければなりません。例えば、他のフラグと一緒に **-o** フラグを (実行可能ファイルの名前を指定するために) 使用できるのは、**-o** オプションとその引数が最後に指定されている場合だけです。 例を以下に示します。

```
xlc -Ovo test test.c
```

これは、以下の指定と同じ効果があります。

```
xlc -O -v -otest test.c
```

ほとんどのフラグ・オプションは 1 文字ですが、中には 2 文字のものもあります。 **-pg** (拡張プロファイル) の指定は、 **-p -g** (プロファイルの **-p**、デバッグ情報の生成の **-g**) を指定することと同じではありませんので注意してください。該当する文字の組み合わせを使用する別のオプションがある場合は、単一ストリングで複数のオプションを指定しないように注意してください。

構成ファイルでのコンパイラー・オプションの指定

デフォルト構成ファイル (/opt/ibm/xlC/13.1.0/etc/xlc.cfg.*$OSRelease.gcc$gccVersion*、例えば、/opt/ibm/xlC/13.1.0/etc/xlc.cfg.sles11.gcc432 または /opt/ibm/xlC/13.1.0/etc/xlc.cfg.rhel6.2.gcc446) はコンパイラーの値とコンパイラー・オプションを定義しま

す。コンパイラーは、C または C++ プログラムをコンパイルするときにこのファイルを参照します。構成ファイルはプレーン・テキスト・ファイルです。特定のコンパイル要件がサポートされるように、このファイルを編集するか、追加のカスタマイズ構成ファイルを作成できます。 詳しくは、『41 ページの『カスタム・コンパイラー構成ファイルの使用』』を参照してください。

プログラム・ソース・ファイルでのコンパイラー・オプションの指定

プラグマ・ディレクティブを使用することにより、プログラム・ソース内にコンパイラー・オプションを指定することができます。プラグマは、コンパイラーに対してのインプリメンテーションを定義した命令です。等価のプラグマ・ディレクティブがあるオプションでは、プラグマの構文の指定方法が複数存在する場合があります。

- **#pragma options** ***option_name*** 構文の使用 — **#pragma options** 構文でコマンド行オプションを使用できます。この構文はオプションと同じ名前を使用し、オプションと同じ構文のサブオプションを使用します。例えば、コマンド行オプションが以下の場合、

  ```
  -qhalt=w
  ```

  プラグマの形式は以下のようになります。

  ```
  #pragma options halt=w
  ```

  個々のオプションの説明で、この形式のプラグマがサポートされているかどうかを示します。詳しくは、『436 ページの『#pragma options』』を参照してください。
- **#pragma** ***name*** 構文を使用する ——一部のオプションには、プラグマ固有の構文を使用して対応するプラグマ・ディレクティブがあります。この場合、追加サブオプションまたはやや異なるサブオプションが含まれることがあります。 101 ページの『個々のオプションの説明』セクションを通して、各オプションの説明は、プラグマのこの形式がサポートされているか、および構文が使用できるかを示します。
- 標準の C99 `_Pragma` 演算子を使用する — 上でリストされたプラグマ・ディレクティブの形式のいずれかをサポートするオプションの場合は、C99 `_Pragma` 演算子構文も C および C++ の両方で使用できます。

プラグマ構文について詳しくは、403 ページの『プラグマ・ディレクティブ構文』を参照してください。

その他のプラグマには、同等のコマンド行オプションはありません。これらについて詳しくは、403 ページの『第 5 章 コンパイラー・プラグマ参照』を参照してください。

プログラム・ソース・ファイル内にプラグマ・ディレクティブを使用して指定されたオプションは、他のプラグマ・ディレクティブを除き、他のすべてのオプション設定をオーバーライドします。同じプラグマ・ディレクティブを複数回指定した場合の効果はさまざまです。特定情報については、それぞれのプラグマの記述を参照してください。

プラグマ設定は組み込みファイルに及ぶことがあります。プラグマ設定の潜在的な副次効果を避けるために、プログラム・ソース内でプラグマで定義された振る舞いが必要なくなった時点で、プラグマ設定のリセットを検討してください。いくつかのプラグマ・オプションでは、それを支援するために、**reset** または **pop** サブオプションが提供されます。これらのサブオプションは、それが適用されるプラグマの詳細記述にリストされています。

矛盾するコンパイラー・オプションの解決

一般に、同じオプションの複数のバリエーションが指定されている場合は (**-qxref** と **-qattr** を除く)、コンパイラーは最後に指定されたオプションの設定を使用します。コマンド行に指定するコンパイラー・オプションは、コンパイラーに処理させる順序で指定しなければなりません。

矛盾するオプションの規則には、 **-I***directory* オプションおよび **-L***directory* オプションの 2 つの例外があります。これらが複数回指定された場合には、その効果が累積されます。

多くの場合、コンパイラーは以下の順序で、矛盾または非互換のオプションを解決します。

1. ソース・コード内のプラグマ・ステートメントは、コマンド行で指定されたコンパイラー・オプションをオーバーライドする。
2. コマンド行に指定されたコンパイラー・オプションは、構成ファイルで環境変数として指定されたコンパイラー・オプションをオーバーライドする。矛盾したり非互換のコンパイラー・オプションがコマンド行に指定されると、コマンド行で後に現れたオプションが優先されます。
3. 環境変数として指定されたコンパイラー・オプションは、構成ファイルで指定されたコンパイラー・オプションをオーバーライドする。
4. 構成ファイルに指定されたコンパイラー・オプションは、コンパイラーのデフォルト設定をオーバーライドする。

すべてのオプション間の矛盾が前記の規則を使用して解決されるとは限りません。以下のテーブルは、例外とコンパイラーによるオプション間の矛盾の処理方法を要約したものです。 コンパイラー・モード間の矛盾を解決する規則、およびアーキテクチャー固有のオプションについては、11 ページの『アーキテクチャー固有のコンパイルでのコンパイラー・オプションの指定』を参照してください。

オプション	矛盾するオプション	解決
-qalias=allptrs	-qalias=noansi	-qalias=noansi
-qalias=typeptr	-qalias=noansi	-qalias=noansi
-qhalt	**-qhalt** によって複数の重大度が指定されている	指定された中で最低の重大度
-qnoprint	-qxref、-qattr、-qsource、-qlistopt、-qlist	-qnoprint
-qfloat=rsqrt	-qnoignerrno	最後に指定されたオプション
-qxref	-qxref=full	-qxref=full
-qattr	-qattr=full	-qattr=full
-qfloat=hsflt	-qfloat=spnans	-qfloat=hsflt

オプション	矛盾するオプション	解決
-E	-P、-o、-S	-E
-P	-c、-o、-S	-P
-#	-v	-#
-F	-B、-t、-W、-qpath	-B、-t、-W、-qpath
-qpath	-B、-t	-qpath
-S	-c	-S
-qnostdinc	-qc_stdinc、-qcpp_stdinc、-qgcc_c_stdinc、-qgcc_cpp_stdinc	-qnostdinc

アーキテクチャー固有のコンパイルでのコンパイラー・オプションの指定

-q32、**-q64**、**-qarch**、および **-qtune** コンパイラー・オプションを使用して、コンパイラーの出力を以下の項目に適合するよう最適化することができます。

- ターゲット・プロセッサーの考えられる最も広範な選択
- 特定のプロセッサー・アーキテクチャー・ファミリー内のプロセッサーの範囲
- 単一の特定プロセッサー

一般に、オプションは以下のことを行います。

- **-q32** は、32 ビット実行モードを選択する。
- **-q64** は、64 ビット実行モードを選択する。
- **-qarch** は、命令コードの生成対象として汎用ファミリー・プロセッサー・アーキテクチャーを選択する。特定の **-qarch** 設定は、選択された **-qarch** 設定に応じてコンパイラーによって生成されるすべての 命令をサポートするシステム上でのみ実行されるコードを生成します。
- **-qtune** は、コンパイラー出力の最適化対象とする特定のプロセッサーを選択する。 **-qtune** の設定には、 **-qarch** オプションとして指定できるものもあり、その場合は **-qtune** オプションとして再度指定する必要はありません。 **-qtune** オプションは、特定のシステム上で実行されているときにコードのパフォーマンスにのみ影響しますが、コードがどこで実行されているかは判別しません。

コンパイラーは以下の順番でコンパイラー・オプションを評価し、最後に検出された有効なコンパイラー・オプションがコンパイラー・モードを決定します。

1. 内部のデフォルト (32 ビット・モード)
2. 構成ファイルの設定
3. コマンド行コンパイラー・オプション (**-q32**、**-q64**、**-qarch**、**-qtune**)
4. ソース・ファイルのステートメント (**#pragma options tune=***suboption*)

コンパイラーが実際に使用するコンパイル・モードは、 **-q32**、**-q64**、**-qarch**、および **-qtune** コンパイラー・オプションの組み合わせによって決定され、これらは以下の条件に左右されます。

- コンパイラー・モード は、最後に検出された **-q32** または **-q64** コンパイラー・オプションのインスタンスに応じて設定される。

- アーキテクチャー・ターゲット は、指定された **-qarch** の設定値がコンパイラー・モード の設定値と互換性がある場合は、 **-qarch** コンパイラー・オプションの最後に検出されたインスタンスに応じて設定される。 **-qarch** オプションが設定されていない場合は、コンパイラーは有効なコンパイラー・モード設定を基に、**-qarch** を適切なデフォルトに設定します。詳しくは、『 113 ページの『-qarch』』を参照してください。
- アーキテクチャー・ターゲットのチューニングは、**-qtune** の設定値がアーキテクチャー・ターゲット およびコンパイラー・モード の設定と互換性がある場合は、**-qtune** コンパイラー・オプションの最後に検出されたインスタンスに応じて設定される。 **-qtune** オプションが設定されていない場合は、コンパイラーは使用中の **-qarch** の設定に応じてデフォルトの **-qtune** 設定を想定します。 **-qarch** が指定されていない場合は、コンパイラーは **-qtune** を、有効なコンパイラー・モード設定を基にデフォルトで選択された有効な **-qarch** を基にした適切なデフォルトに設定します。

これらのオプションの許容される組み合わせについては、377 ページの『-qtune』を参照してください。

以下のリストで、起こりうるオプションの矛盾とそれらの矛盾のコンパイラーによる解決について説明します。

- **-q32** または **-q64** 設定がユーザーの選択した **-qarch** オプションと非互換である。

  解決: -q32 または **-q64** 設定は **-qarch** オプションをオーバーライドします。コンパイラーは警告メッセージを発行し、**-qarch** をデフォルト設定に設定し、**-qtune** オプションをそのデフォルト値に応じて設定します。
- **-qarch** オプションが、ユーザーが選択した **-qtune** オプションと非互換である。

  解決: コンパイラーは警告メッセージを発行し、**-qtune** を **-qarch** 設定のデフォルトの **-qtune** 値に設定します。
- 選択した **-qarch** または **-qtune** オプションがコンパイラーに認識されない。

  解決: コンパイラーは警告メッセージを発行し、**-qarch** および **-qtune** をデフォルト設定に設定します。コンパイラー・モード (32 ビットまたは 64 ビット) は、**-q32 /-q64** コンパイラー設定によって決定されます。

関連情報
- 113 ページの『-qarch』
- 377 ページの『-qtune』
- 104 ページの『-q32、-q64』

gxlc および gxlc++ での GNU C/C++ コンパイラー・オプションの再使用

gxlc および **gxlc++** ユーティリティーはそれぞれ GNU C または C++ コンパイラー・オプションを受け入れて、同等の XL C/C++ オプションに変換します。どちらのユーティリティーも XL C/C++ オプションを使用して **xlc** または **xlc++** の呼び出しコマンドを作成し、そのコマンドを使用してコンパイラーを呼び出します。これらのユーティリティーは、以前に GNU C/C++ を使用して開発されたアプリケーション用に作成された Make ファイルを再利用するために提供されています。しか

し、XL C/C++ の機能を完全に活用するためには、XL C/C++ 呼び出しコマンドとそれらの関連オプションを使用することをお勧めします。

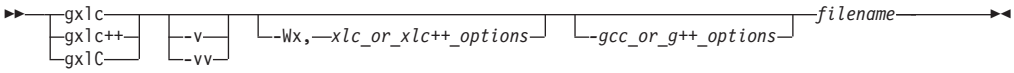
gxlc および **gxlc++** のアクションは構成ファイル /opt/ibm/xlC/13.1.0/etc/gxlc.cfg によって制御されます。XL C または XL C++ の相対オプションを持つ GNU C/C++ オプションはこのファイルに表示されます。すべての GNU オプションが対応する XL C/C++ オプションを持っているわけではありません。 **gxlc** および **gxlc++** は変換されなかった入力オプションについては警告を戻します。

注: SUSE Linux Enterprise Server 11 (SLES11) および Red Hat Enterprise Linux 6 (RHEL6) で **gxlc** または **gxlc++** を使用する場合、ソース・コードは 64 ビット・モードでコンパイルされます。これより低いレベルのオペレーティング・システムでは、ソース・コードは 32 ビット・モードでコンパイルされます。

gxlc および **gxlc++** オプションのマッピングは変更可能です。**gxlc** または **gxlc++** 構成ファイルへの追加またはこのファイルの編集方法について詳しくは、46 ページの『gxlc または gxlc++ オプション・マッピングの構成』を参照してください。

gxlc または gxlc++ 構文

以下の図は gxlc または gxlc++ 構文を示しています。



ここで、

filename
: コンパイルするファイルの名前です。

-v
: XL C/C++ を呼び出すために使用したコマンドを検査します。ユーティリティーは、作成した XL C/C++ 呼び出しコマンドを表示してから、このコマンドを使用してコンパイラーを呼び出します。

-vv
: シミュレーションを実行します。ユーティリティーは、作成した XL C/C++ 呼び出しコマンドを表示しますが、コンパイラーを呼び出しません。

-Wx,*xlc_or_xlc++_ options*
: 指定された XL C/C++ オプションを直接 **xlc** または **xlc++** 呼び出しコマンドに送信します。ユーティリティーは、特定オプションの変換を試行せずに、作成中の XL C/C++ 呼び出しに追加します。このオプションはユーティリティーのパフォーマンスを改善するために、既知の XL C/C++ オプションと共に使用してください。複数の *xlc_or_xlc++_ options* は、コンマで区切られます。

-*gcc_or_g++_options*
: XL C/C++ オプションに変換された GNU C/C++ オプションユーティリティーは、変換できないオプションに対して警告を出します。現在 **gxlc** または **gxlc++** によって認識される GNU C/C++ オプションは構成ファイル gxlc.cfg に入っています。複数の *-gcc_or_g++_options* はスペース文字で区切られます。

例

GCC **-fstrict-aliasing** オプションを使用して Hello World プログラムの C バージョンをコンパイルするには、以下を使用することができます。

```
gxlc -fstrict-aliasing hello.c
```

これは次のように変換されます。

```
xlc -qalias=ansi hello.c
```

その後、このコマンドは XL C コンパイラーを呼び出すために使用されます。

関連情報

- 46 ページの『gxlc または gxlc++ オプション・マッピングの構成』

プリプロセッシング

プリプロセッシングは、通常コンパイラー呼び出しによって開始される変換の第 1 フェーズとして、ソース・ファイルのテキストを操作します。プリプロセッシングで実行される共通タスクは、マクロ置換、条件付きコンパイル・ディレクティブのテスト、およびファイルの組み込みです。

プリプロセッサーは、コンパイルを行わずにテキストを処理するために独立して呼び出すことができます。出力は中間ファイルで、このファイルは以降の変換のための入力にすることができます。コンパイルを行わないプリプロセッシングは、組み込みディレクティブ、条件付きコンパイル・ディレクティブ、および複雑なマクロ展開の結果を確認する方法を提供するため、デバッグ補助機能として役立つ場合があります。

以下のテーブルは、プリプロセッサーの操作を指図するオプションをリストしたものです。

オプション	説明
150 ページの『-E』	ソース・ファイルをプリプロセスし、出力を標準出力に書き込みます。デフォルトで、`#line` ディレクティブが生成されます。
288 ページの『-P』	ソース・ファイルをプリプロセスし、各ソース・ファイルに対してファイル名サフィックス .i を含む中間ファイルを作成します。デフォルトで、`#line` ディレクティブは生成されません。
304 ページの『-qppline』	**-E** オプションおよび **-P** オプションの `#line` ディレクティブの生成をオン/オフに切り替えます。
124 ページの『-C、-C!』	プリプロセスされた出力にコメントを保持します。
141 ページの『-D』	`#define` ディレクティブに定義するように、コマンド行からマクロ名を定義します。
380 ページの『-U』	コンパイラーまたは **-D** オプションによって定義されたマクロ名の定義を解除します。
329 ページの『-qshowmacros』	プリプロセスされた出力にマクロ定義を出す。

組み込みファイルのディレクトリー検索シーケンス

XL C/C++ コンパイラーは、以下のタイプのインクルード・ファイルをサポートしています。

- コンパイラーが提供するヘッダー・ファイル (本書では、*XL C/C++ ヘッダー* として言及されます)
- C および C++ 標準によって操作されるヘッダー・ファイル (本書では、システム・ヘッダーとして言及されます)
- オペレーティング・システムが提供するヘッダー・ファイル (本書では、システム・ヘッダーとしても言及されます)
- ユーザー定義のヘッダー・ファイル

以下いずれの方法でも、すべてのタイプのヘッダー・ファイルを組み込むことができます。

- 組み込みソース・ファイルで標準 `#include <file_name>` プリプロセッサー・ディレクティブを使用する。
- 組み込みソース・ファイルで標準 `#include "file_name"` プリプロセッサー・ディレクティブを使用する。
- **-qinclude** コンパイラー・オプションを使用する。

完全 (絶対) パス名を使用してヘッダー・ファイルを指定するには、組み込むヘッダー・ファイルのタイプにかかわらず、これらの方法を交互に使用できます。ただし、*相対*パス名を使用してヘッダー・ファイルを指定するには、ファイルを組み込む方法に合わせて、ファイルを見つけるための異なるディレクトリー検索順序がコンパイラーで使用されます。

さらに、**-qidirfirst** および **-qstdinc** コンパイラー・オプションがこの検索順序に影響を与えることがあります。以下は、ファイルの組み込み方法および有効なコンパイラー・オプションによって異なる、コンパイラーがヘッダー・ファイルを検索する順序の要約です。

1. **-qinclude** のみでヘッダー・ファイルを組み込む場合: コンパイラーは、コンパイラーが呼び出された現行 (作業中の) ディレクトリーを検索します。[1]
2. **-qinclude** または `#include "file_name"` でヘッダー・ファイルを組み込む場合: コンパイラーは、組み込みファイルが格納されるディレクトリーを検索します。[1]
3. すべてのヘッダー・ファイル: コンパイラーは **-I** コンパイラー・オプションで指定された各ディレクトリーを、コマンド行に表示される順序で検索します。
4. すべてのファイル : コンパイラーは、標準ディレクトリーでシステム・ヘッダーを検索します。これらのヘッダーのデフォルト・ディレクトリーは、コンパイラーの構成ファイルで指定されています。この場所はインストール中に設定されますが、検索パスを C **-qgcc_c_stdinc** C C++ **-qgcc_cpp_stdinc** C++ オプションで変更することもできます。[2]

注:

1. **-qidirfirst** コンパイラー・オプションが有効であれば、ステップ 1 と 2 の前にステップ 3 を実行できます。

2. **-qnostdinc** コンパイラー・オプションが有効であれば、ステップ 4 と 5 を省略できます。

関連情報
- 186 ページの『-I』
- 139 ページの『-qc_stdinc (C のみ)』
- 140 ページの『-qcpp_stdinc (C++ のみ)』
- 176 ページの『-qgcc_c_stdinc (C のみ)』
- 177 ページの『-qgcc_cpp_stdinc (C++ のみ)』
- 188 ページの『-qidirfirst』
- 191 ページの『-qinclude』
- 351 ページの『-qstdinc』

リンク

リンカーは、指定されたオブジェクト・ファイルをリンクして、1 つの実行可能ファイルを作成します。呼び出しコマンドの 1 つを使用してコンパイラーを呼び出すと、次のいずれかのコンパイラー・オプションを指定した場合を除き、自動的にリンカーが呼び出されます: **-E**、 **-P**、**-c**、**-S**、**-qsyntaxonly** または **-#**。

入力ファイル

オブジェクト・ファイル、非ストリップ実行可能ファイル、およびライブラリー・ファイルは、リンカーの入力データとして使用できます。オブジェクト・ファイルには、`.o` のサフィックスが付加されている必要があります (例えば、*filename*`.o`)。静的ライブラリー・ファイル名には `.a` のサフィックスが付きます (例えば、*filename*`.a`)。動的ライブラリー・ファイル名には通常、`.so` のサフィックスが付きます (例えば、*filename*`.so`)。

出力ファイル

リンカーは、*実行可能ファイル* を生成して、そのファイルを現行ディレクトリーに入れます。実行可能ファイルのデフォルト名は `a.out` です。実行可能ファイルを明示的に命名するには、コンパイラー呼び出しコマンドに **-o** file_name オプションを使用してください。ここで、*file_name* はその実行可能ファイルに指定したい名前です。例えば、`myfile.c` をコンパイルして `myfile` という名前の実行可能ファイルを生成するには、以下のように入力します。

```
xlc myfile.c -o myfile
```

-qmkshrobj オプションを使用して共有ライブラリーを作成する場合、作成される共有オブジェクトのデフォルト名は `a.out` となります。**-o** オプションを使用すると、ファイルの名前を変更し、サフィックス `.so` を付加できます。

ld コマンドにより、明示的にリンカーを呼び出すことができます。ただし、コンパイラー呼び出しコマンドは幾つかのリンカー・オプションを設定し、デフォルトでいくつかの標準ファイルを実行可能出力にリンクします。ほとんどの場合、オブジェクト・ファイルをリンクするには、コンパイラー呼び出しコマンドの 1 つを使用することをお勧めします。リンクに使用できるオプションの詳細なリストについては、 98 ページの『リンク』を参照してください。

注: 非デフォルト・リンカーを使用する場合は、以下のオプションのいずれかを使用できます。

- **-t** および **-B** を使用して、非デフォルト・リンカーを指定します。例えば、以下のようにします。

  ```
  -tl -Blinker_path
  ```

- コンパイラーの構成ファイルを、非デフォルト・リンカーを使用するようにカスタマイズします。構成ファイルのカスタマイズ方法について詳しくは、『カスタム・コンパイラー構成ファイルの使用』および『カスタム構成ファイルの作成』を参照してください。

関連情報

- 277 ページの『-qmkshrobj』

リンクの順序

コンパイラーは、次の順序でライブラリーをリンクします。

1. システム開始ライブラリー
2. ユーザー .o ファイルおよびライブラリー
3. XL C/C++ ライブラリー
4. C++ 標準ライブラリー
5. C 標準ライブラリー

関連情報

- 98 ページの『リンク』
- 『再配布可能ライブラリー』

再配布可能ライブラリー

XL C/C++ を使用してアプリケーションを作成すると、以下の 1 つ以上の再配布可能ライブラリーを使用する可能性があります。アプリケーションを出荷するときには、そのアプリケーションのユーザーがライブラリーの含まれたパッケージを持っていることを確認してください。必要なライブラリーがユーザーに使用可能であることを確認するには、以下のいずれかを実行する必要があります。

- 再配布可能なライブラリーを含むパッケージ をアプリケーションと共に出荷することができます。パッケージは、インストール CD の images/rpms ディレクトリーに格納されます。
- ユーザーは、再配布可能なライブラリーを持つパッケージを、以下の XL C/C++ サポートの Web サイトからダウンロードできます。

  http://www.ibm.com/support/entry/portal/overview/software/rational/xl_c~c++_for_linux

これらのパッケージの配布に関連するライセンス要件については、CD にある LicenseAgreement.pdf ファイルを参照してください。

表 6. 再配布可能ライブラリー

パッケージ名	ライブラリー (およびデフォルトのインストール・パス)	説明
libxlc-devel	/opt/ibm/xlC/13.1.0/lib/libxl.a /opt/ibm/xlC/13.1.0/lib/libxlopt.a /opt/ibm/xlC/13.1.0/lib/libxlopt.a /opt/ibm/xlC/13.1.0/lib64/libxlopt.a	XL C/C++ コンパイラー・ライブラリー
vacpp.rte	/opt/ibmcmp/vac/13.1/lib/libibmc++.so.1 /opt/ibmcmp/vac/13.1/lib64/libibmc++.so.1	XL C++ ランタイム・ライブラリー
xlsmp.rte	/opt/ibmcmp/lib/libxlomp_ser.so.1 /opt/ibmcmp/lib/libxlsmp.so.1 /opt/ibmcmp/lib64/libxlomp_ser.so.1 /opt/ibmcmp/lib64/libxlsmp.so.1	SMP (OMP) ランタイム・ライブラリー
xlsmp.msg.rte	/opt/ibmcmp/msg/en_US/smprt.cat /opt/ibmcmp/msg/en_US.utf8/smprt.cat	SMP メッセージ・カタログ (英語)
	/opt/ibmcmp/msg/ja_JP/smprt.cat /opt/ibmcmp/msg/ja_JP.eucjp/smprt.cat /opt/ibmcmp/msg/ja_JP.utf8/smprt.cat	SMP メッセージ・カタログ (日本語)
	/opt/ibmcmp/msg/zh_CN/smprt.cat /opt/ibmcmp/msg/zh_CN.gb18030/smprt.cat /opt/ibmcmp/msg/zh_CN.gb2312/smprt.cat /opt/ibmcmp/msg/zh_CN.gbk/smprt.cat /opt/ibmcmp/msg/zh_CN.utf8/smprt.cat	SMP メッセージ・カタログ (中国語)

以前のバージョンとの互換性

このセクションでは、以前のバージョンとの互換性に関する問題と、その回避策について説明します。

コンパイラー・オプションの互換性の問題

IBM XL C/C++ for Linux, V13.1では、threadprivate データ、つまり OpenMP threadprivate 変数のインプリメンテーションが改良されました。ランタイム実装の代わりに、オペレーティング・システムのスレッド・ローカル・ストレージが使用されます。この新しい実装によって、一部のアプリケーションではパフォーマンスが向上する場合があります。

11.1 より前のレベルでコンパイルしたオブジェクト・ファイル `.o` と、IBM XL C/C++ for Linux, V13.1 でコンパイルしたオブジェクト・ファイルを混在させ、新旧両方のオブジェクト・ファイルで同じ OpenMP threadprivate 変数を参照する計画

の場合、異なるインプリメンテーションによって非互換性の問題が起きることがあります。リンク・エラー、コンパイル時エラー、あるいはそれ以外の未定義の動作が発生する場合があります。以前のバージョンとの互換性をサポートするために、**-qsmp=noostls** サブオプションを使用して古い実装に切り替えることができます。デフォルトのサブオプション **-qsmp=ostls** でプログラム全体を再コンパイルすることで、新しい実装の利点を得られます。

11.1 より前のレベルでコンパイルしたオブジェクト・ファイルに古い実装が含まれているかどうかがわからない場合は、**readelf -s** コマンドを使用して、**-qsmp=noostls** サブオプションの使用が必要かどうかを判別できます。以下のコードは、**readelf -s** コマンドの使用例です。

```
> readelf -s oldfiles.o
...
._xlGetThStorageBlock U           -
._xlGetThValue        U           -
...
```

上記の例で `_xlGetThStorageBlock` または `_xlGetThValue` が見つかった場合は、オブジェクト・ファイルに古い実装が含まれています。その場合は、**-qsmp=noostls** を使用する必要があります。それ以外の場合は、デフォルトのサブオプション **-qsmp=ostls** を使用します。

コンパイラーのメッセージおよびリスト

以下のセクションでは、コンパイル後にコンパイラーが報告を行うときのさまざま方法について説明します。

-
- 『コンパイラー・メッセージ』
- 22 ページの『コンパイラー戻りコード』
- 22 ページの『コンパイラー・リスト』
- 25 ページの『メッセージ・カタログ・エラー』
- 26 ページの『コンパイル中のページ・スペース・エラー』

コンパイラー・メッセージ

コンパイラーは、C または C++ のソース・プログラムをコンパイル中にプログラミング・エラーを検出した場合、診断メッセージを標準エラー・デバイスへ発行するか、**-qsource** オプションを指定してコンパイルしている場合はリスト・ファイルに書き込みます。これらの診断メッセージは、C または C++ 言語に固有のものです。

C コンパイラー・オプション **-qsrcmsg** を指定しており、エラーが特定のコード行に該当する場合、再構成されたソース行または一部のソース行がエラー・メッセージと共に組み込まれます。再構成されたソース行とは、すべてのマクロが展開された、プリプロセス済みのソース行です。

-qflag オプションまたは **-w** オプションを使用すると、重大度に応じて出される診断メッセージを制御することができます。プログラム内に潜在する問題についての追加の通知メッセージを得るには、**-qinfo** オプションを使用します。

関連情報

- 340 ページの『-qsource』
- 344 ページの『-qsrcmsg (C のみ)』
- 158 ページの『-qflag』
- 393 ページの『-w』
- 193 ページの『-qinfo』

コンパイラー・メッセージ・フォーマット

診断メッセージのフォーマットは以下のとおりです。

"*file*", line *line_number.column_number*: 15*dd-number* (*severity*) *text*.

ここで、

file
エラーのある C または C++ ソース・ファイルの名前です。

line_number
エラーが検出されたソース・コードの行番号です。

column_number
エラーが検出されたソース・コードの列番号です。

15 コンパイラーの製品 ID です。

dd このメッセージを発行したコンパイラー・コンポーネントを示す 2 桁のコードです。*dd* は、以下の値のいずれかになります。

00 - メッセージを生成または最適化するコード

01 - コンパイラー・サービス・メッセージ

05 - C コンパイラーに特定のコンパイラー・サービス・メッセージ

06 - C コンパイラーに特定のコンパイラー・サービス・メッセージ

40 - C++ コンパイラーに特定のコンパイラー・サービス・メッセージ

86 - プロシージャー間分析 (IPA) に特定のメッセージ

number
メッセージ番号です。

severity
エラーの重大度を表す文字です。これらの説明については、 21 ページの『メッセージ重大度レベルとコンパイラー応答』を参照してください。

text
エラーを記述するメッセージです。

C **-qsrcmsg** でコンパイルすると、診断メッセージのフォーマットは以下のようになります。

x - 15*dd-nnn*(*severity*) *text*.

ここで、*x* はフィンガー行のフィンガーを示す文字です。

メッセージ重大度レベルとコンパイラー応答

XL C/C++ コンパイラーは、診断メッセージにマルチレベル種別スキームを使用します。重大度の各レベルは、コンパイラー応答と関連しています。以下のテーブルは、重大度レベルの省略形とそのレベルに関連したデフォルトのコンパイラー応答を提供します。以下のいずれかのオプションを使用して、デフォルトのコンパイラー応答を調整できます。

- **-qhalt** は、デフォルトより低い重大度レベルでコンパイル・フェーズを一時停止します。
- **-qmaxerr** は、特定の重大度レベルのエラーが特定の数に到達すると同時に、コンパイル・フェーズを一時停止します。
- **-qhaltonmsg** は、特定のエラーが検出されると同時に、コンパイル・フェーズを一時停止します。

表 7. コンパイラーのメッセージ重大度レベル

文字	重大度	コンパイラー応答
I	通知	コンパイルは継続し、オブジェクト・コードが生成されます。このメッセージは、コンパイル中に検出された条件を報告します。
W	警告	コンパイルは継続し、オブジェクト・コードが生成されます。メッセージは、有効ではあるが、おそらく意図されたものではない条件を報告します。
C E	エラー	コンパイルは継続し、オブジェクト・コードが生成されます。コンパイラーが訂正できるエラー条件が存在しますが、プログラムが予期される結果を生み出さない可能性があります。
S	重大エラー	コンパイルは継続しますが、オブジェクト・コードは生成されません。コンパイラーが訂正できないエラー条件が存在します。 • メッセージがリソースの限界 (例えばファイル・システムまたはページング・スペースがいっぱいになった) を示している場合は、リソースを追加して再コンパイルしてください。 • メッセージが別のコンパイラー・オプションの必要性を示している場合は、それらを使用して再コンパイルしてください。 • 重大エラーの前に報告されたその他のエラーがないか検査して訂正してください。 • メッセージが内部のコンパイル時エラーを示している場合は、このメッセージを IBM サービス担当員に報告してください。
C U	回復不能エラー	コンパイラーは停止します。内部のコンパイル時エラーが発生しました。メッセージを IBM サービス担当員に報告してください。

関連情報

- 179 ページの『-qhalt』
- 271 ページの『-qmaxerr』
- 181 ページの『-qhaltonmsg』

- 機能カテゴリー別のオプションの要約: リストとメッセージ

コンパイラー戻りコード

コンパイルの終了時、コンパイラーは以下のいずれかの条件のときに戻りコードをゼロに設定します。

- メッセージが発行されない。
- 診断されたすべてのエラーの中で最高の重大度レベルが、**-qhalt** コンパイラー・オプションの設定よりも小さく、さらにエラーの数が **-qmaxerr** コンパイラー・オプションで設定された限界値に達していない。
- **-qhaltonmsg** コンパイラー・オプションによって指定されたメッセージが発行されない。

それ以外の場合、コンパイラーは以下の値の 1 つに戻りコードを設定します。

戻りコード	エラー・タイプ
1	**-qhalt** コンパイラー・オプションの設定よりも高い重大度レベルを持つエラーが検出された。
40	オプション・エラーまたは回復不能エラーが検出された。
41	構成ファイル・エラーが検出された。
249	指定ファイルがないというエラーが検出された。
250	メモリー不足のエラーが検出された。コンパイラーが、使用するためのメモリーをこれ以上割り振れません。
251	シグナル受信エラーが検出された。つまり、回復不能エラーまたは割り込みシグナルが発生しました。
252	ファイルが見つからないエラーが検出された。
253	入出力エラーが検出された。ファイルの読み取りや書き込みができない。
254	fork エラーが検出された。新規プロセスを作成できません。
255	プロセスの実行中にエラーが検出された。

注: ランタイム・エラーの戻りコードも表示される可能性があります。

gxlc および gxlc++ の戻りコード

gxlc および gxlc++ は他の呼び出しコマンドと同様に、リスト、コンパイルに関連した診断メッセージ、失敗した GNU オプションの変換に関連した警告、および戻りコードといった出力を戻します。gxlc または gxlc++ が正常にコンパイラーを呼び出すことができないと、次のいずれかの値に戻りコードを設定します。

40 gxlc または gxlc++ オプション・エラーまたは回復不能エラーが検出された。

255 プロセスの実行中にエラーが検出された。

コンパイラー・リスト

リストは、特定のコンパイルに関する情報が含まれたコンパイラー出力ファイル (サフィックス .lst を持つ) です。デバッグ援助機能、コンパイラー・リストは、コンパイルで発生した問題を判別するのに有用です。例えば、コンパイル中に出されたすべての診断メッセージはこのリストに書き込まれます。

リストの作成時に、以下のオプションを使用してコンパイルすると、異なるタイプの情報が参照できます。

- -qsource
- -qlistopt
- -qattr
- -qxref
- -qlist
- -qreport

リスト情報はセクション別に編成されています。リストにはヘッダー・セクションと、有効な他のオプションに応じて、他のセクションの組み合わせが含まれています。これらのセクションの内容は下記の通りです。

ヘッダー・セクション
コンパイラーの名前、バージョン、リリース、ソース・ファイル名、およびコンパイルの日時をリストします。

ソース・セクション
-qsource オプションを使用すると、入力ソース・コードを行番号と共にリストします。 行にエラーがある場合は、関連付けられたエラー・メッセージが、ソース行の後に表示されます。マクロを含む行にはマクロ展開を示す追加行があります。このセクションはデフォルトでメインのソース・ファイルをリストします。すべてのヘッダー・ファイルも展開するには、**-qshowinc** オプションを使用します。

オプション・セクション
コンパイル中に有効だったオプションをリストします。デフォルトでは、指定されたオプションをリストします。すべてのオプションを取得するには、**-qlistopt** オプションを指定します。

属性と相互参照リスト・セクション
-qattr または **-qxref** オプションを使用すると、コンパイル単位で使用される変数に関する情報 (型、ストレージ期間、範囲、およびその変数がどこに定義されていてどこで参照されているかなど) を提供します。これらの各オプションは、コンパイルで使用された ID に関するさまざまな情報を提供します。

ファイル・テーブル・セクション
それぞれのメイン・ソース・ファイルと組み込みファイルのファイル名と数をリストします。各ファイルにはファイル番号が関連付けられています (メイン・ソース・ファイルから開始され、このファイルにはファイル番号 0 が割り当てられます)。リストは各ファイルごとに、そのファイルがどのファイルのどの行から組み込まれたものであるかを示します。**-qshowinc** オプションも有効な場合は、ソース・セクションの各ソース行には、その行がどのファイルから来ているものであるかを示すファイル番号が入ります。

PDF レポート・セクション
-qreport オプションを **-qpdf2** オプションと一緒に使用した場合、このセクションには以下の情報が含まれます。

ループの繰り返しカウント
特定の入力データ・セットについての最も頻繁なループの繰り返し

カウントと、平均的な繰り返しカウントが、プログラム内の大部分のループについて計算されます。この情報は、プログラムを最適化レベル **-O5** でコンパイルした場合にのみ入手できます。

ブロックおよび呼び出しカウント

このセクションは、プログラムの*呼び出し構造* と、呼び出された各関数の実行カウントを対象としています。また、このセクションには各関数の*ブロック情報* も含まれています。非ユーザー定義関数の場合は、実行カウントだけが示されます。合計ブロックと呼び出し範囲、および実行カウントの降順で示したユーザー関数リストが、このレポート・セクションの末尾に出力されます。さらに、リスト・ファイル内の疑似コードの各ブロックでは、冒頭にブロック・カウント情報が出力されます。

キャッシュ・ミス

このセクションは、単一テーブルに出力されます。ここでは、特定の関数のキャッシュ・ミス の数と、関数に関する追加情報、例えば、`Cache Level`、`Cache Miss Ratio`、`Line Number`、`File Name`、および `Memory Reference` などが報告されます。

注: このレポートを取得するには、**-qpdf1=level=2** オプションを使用する必要があります。
また、環境変数 **PDF_PM_EVENT** を実行時に使用して、プロファイルを作成するキャッシュのレベルを選択することもできます。

プロファイル・データの関連度 (Relevance of profiling data)

このセクションには、**-qpdf1** フェーズでのプロファイル・データとソース・コードの関連度が示されます。関連度は、0 から 100 の範囲の数値で示されます。値が大きいほどプロファイル・データとソース・コードの関連性が高く、プロファイル・データの使用によるパフォーマンスの向上幅も大きくなります。

欠落しているプロファイル・データ (Missing profiling data)

このセクションには、欠落しているプロファイル・データについての警告メッセージが出力されます。この警告メッセージは、コンパイラーがプロファイル・データを検出しなかった関数それぞれについて出されます。

古いプロファイル・データ (Outdated profiling data)

このセクションに、古いプロファイル・データについての警告メッセージが出力される場合があります。コンパイラーは、**-qpdf1** フェーズの後に変更された関数それぞれについてこの警告メッセージを出します。警告メッセージは、**-qpdf1** フェーズから **-qpdf2** フェーズまでの間に最適化レベルが変更された場合にも出されます。

変換レポート・セクション

-qreport オプションが有効である場合、このセクションにはオリジナル・ソース・コードに対応する疑似コードが表示されます。これにより、**-qhot** または **-qsmp** オプションで生成された並列化とループ変換を見ることができます。レポートのこのセクションには、**-qsmp** および **-qhot=level=2** を指定してコンパイルした場合の、ループ・ネストに関する追加ループ変換および並列化情報も示されます。

また、このセクションでは、特定のループ用に作成されたストリームの数と、コンパイラーによって挿入されたデータ・プリフェッチ命令のロケーションも報告されます。データ・プリフェッチ挿入ロケーションに関する情報を生成するには、最適化レベルの **-qhot**、**-O3 -qhot**、**-O4**、または **-O5** を**-qreport** と一緒に使用します。

データ再編成セクション

-qreport が **-qipa=level=2** または **-O5** と一緒に使用された場合、IPA リンク・パス時のプログラム変数データに関するデータ再編成メッセージを表示します。再編成情報には、以下が含まれます。

- 配列分割
- 配列転置
- メモリー割り振りのマージ
- 配列インターリービング
- 配列合体

コンパイル・エピローグ・セクション

重大度レベル、読み取られたソース行の数、およびコンパイルが成功したかどうかによって、診断メッセージの要約を表示します。

オブジェクト・セクション

-qlist オプションを使用すると、コンパイラーによって生成されたオブジェクト・コードをリストします。このセクションは、コード生成エラーが原因でプログラムが予期した通りに実行されていないという疑いがある場合、実行時の問題を診断するのに有用です。

関連情報

- コマンド行オプションの要約: リストとメッセージ

メッセージ・カタログ・エラー

コンパイラーでユーザー・プログラムをコンパイルするためには、その前にメッセージ・カタログをインストールし、環境変数 *LANG* と *NLSPATH* を、メッセージ・カタログのインストール言語に設定しておかなければなりません。

コンパイル中に以下のメッセージが表示された場合は、適切なメッセージ・カタログをオープンできません。

```
Error occurred while initializing the message system in
file: message_file
```

ここで、*message_file* はコンパイラーがオープンできないメッセージ・カタログの名前です。このメッセージは英語のみで出されます。

その後、メッセージ・カタログと環境変数が適所にあり、正しいことを検証する必要があります。メッセージ・カタログや環境変数が正しくない場合、コンパイルは継続できますが、診断メッセージが抑止され、代わりに以下のメッセージが出されます。

```
No message text for message_number
```

ここで、*message_number* はコンパイラーの内部メッセージ番号です。このメッセージは英語のみで出されます。

コンパイラーをデフォルトの場所にインストールしていると想定した場合、以下のコマンドを使用すると、ご使用システムにどのメッセージ・カタログがインストールされているかを判別し、カタログのすべてのファイル名をリストすることができます。

```
ls /opt/ibm/xlC/13.1.0/msg/$LANG/*.cat
```

ここで *LANG* は、ご使用のシステム上における、システム・ロケールを指定する環境変数です。

LANG が正しく設定されていないと、デフォルトで、コンパイラーは **en_US** 用のメッセージ・カタログを呼び出します。

NLSPATH および *LANG* 環境変数の詳細については、ご使用のオペレーティング・システムの資料を参照してください。

コンパイル中のページ・スペース・エラー

コンパイル中にオペレーティング・システムのページ・スペースが不足すると、コンパイラーは以下のメッセージを発行します。

```
1501-229 Compilation ended due to lack of space.
```

ページ・スペースの問題を最小限に抑えるには、以下のいずれかの処置を行って、プログラムを再コンパイルしてください。

- プログラムを複数のソース・ファイルに分割して、プログラムのサイズを減らす。
- 最適化を行わずにプログラムをコンパイルする。
- システムのページング・スペースについて競合するプロセスの数を減らす。
- システムのページング・スペースを増やす。

ページング・スペースの詳細と、その割り振り方法については、ご使用のオペレーティング・システムの資料を参照してください。

第 2 章 コンパイラーのデフォルトの構成

XL C/C++ でアプリケーションをコンパイルするとき、コンパイラーでは、複数の方法で決定されたデフォルト設定が使用されます。

- 内部的に定義された設定。これらの設定はコンパイラーによって定義済みで、変更はできません。
- システム環境変数によって定義された設定。コンパイラーには特定の環境変数が必要で、その他はオプションです。インストール中に、基本的な環境変数の一部が既に設定されている場合もあります (詳細は、XL C/C++ インストール・ガイドを参照してください)。 『環境変数の設定』では、並列処理に使用した環境変数を含む、コンパイラーのインストール後に設定またはリセットできる必須およびオプションの環境変数の詳細リストが参照できます。
- コンパイラー構成ファイル xlc.cfg で定義された設定。コンパイラーには、構成ファイルで決定される多くの設定が必要です。通常、構成ファイルはインストールの作業中に自動的に生成されます (詳しくは、「XL C/C++ インストール・ガイド」を参照してください)。ただし、インストール後にこのファイルをカスタマイズして、追加のコンパイラー・オプション、デフォルト・オプション設定、ライブラリー検索パス、およびその他の設定を指定できます。構成ファイルのカスタマイズに関する情報については、 41 ページの『カスタム・コンパイラー構成ファイルの使用』を参照してください。
- GCC オプションの構成ファイルによって定義された設定。gxlc または gxlc++ ユーティリティーを使用して GCC オプションをマップすると、デフォルト・オプションのマッピングが /opt/ibm/xlC/13.1.0/etc/gxlc.cfg ファイルで定義されます。ファイルは要件に合わせてカスタマイズできます。詳しくは、『 46 ページの『gxlc または gxlc++ オプション・マッピングの構成』』を参照してください。

環境変数の設定

Bourne シェル、Korn シェル、BASH シェルで環境変数を設定するには、以下のコマンドを使用します。

```
variable=value
export variable
```

ここで、*variable* は環境変数の名前で、*value* はその変数に割り当てる値です。

C シェルで環境変数を設定するには、以下のコマンドを使用します。

```
setenv variable value
```

ここで、*variable* は環境変数の名前で、*value* はその変数に割り当てる値です。

Bourne、Korn、および BASH シェルで、変数をすべてのユーザーがアクセスできるように設定するには、これらのコマンドをファイル */etc/profile* に追加します。特定のユーザーに対してのみ環境変数を設定するには、これらのコマンドをユーザーのホーム・ディレクトリーの .profile ファイルに追加します。C シェルでは、これらのコマンドをファイル */etc/csh.cshrc* に追加します。特定のユーザーに対してのみ環

境変数を設定するには、これらのコマンドをユーザーのホーム・ディレクトリーの .cshrc ファイルに追加します。この環境変数は、そのユーザーがログインするたびに設定されます。

次のセクションでは、XL C/C++ 用に設定できる環境変数およびこれらの環境変数を使用してコンパイルしたアプリケーションについて説明します。

- 『コンパイル時およびリンク時の環境変数』
- 29 ページの『ランタイム環境変数』

コンパイル時およびリンク時の環境変数

以下に示すのは、コードのコンパイル時およびリンク時にコンパイラーが使用する環境変数です。多くは Linux オペレーティング・システムに組み込まれています。デフォルトの en_US 以外のロケールを使用している場合に設定する必要がある *LANG* および *NLSPATH* を除き、これらすべての変数はオプションです。

LANG ご使用のオペレーティング・システムのロケールを指定します。メッセージおよびヘルプ・ファイル用にコンパイラーが使用するデフォルト・ロケールは、米国英語の en_US ですが、コンパイラーはその他のロケールもサポートします。これらのリストについては、以下を参照してください。*ナショナル・ランゲージ・サポート* (「*XL C/C++ インストール・ガイド*」)。別のロケールを使用できるよう *LANG* 環境変数を設定する方法については、ご使用のオペレーティング・システムの資料を参照してください。

LD_RUN_PATH
動的にロードされるライブラリーの検索パスを指定します。これは、リンク時オプションでの **-R** の指定と同じ意味になります。環境変数によって指定された共有ライブラリーの場所は実行可能ファイルに組み込まれるため、ダイナミック・リンカーはアプリケーション実行時にライブラリーを検出できます。この環境変数について詳しくは、ご使用のオペレーティング・システムの資料を参照してください。 314 ページの『-R』も参照してください。

NLSPATH
コンパイラー・メッセージとヘルプ・ファイルを検出するためのディレクトリー検索パスを指定します。この環境変数を設定する必要があるのは、コンパイラー・メッセージおよびヘルプ・ファイルに使用する各国語が英語でない場合のみです。 *NLSPATH* の設定の詳細については、「*XL C/C++ インストール・ガイド*」の『*XL C/C++ エラー・メッセージの使用可能化*』を参照してください。

PATH コンパイラーの実行可能ファイルのディレクトリー検索パスを指定します。実行可能ファイルは、デフォルト・ロケーションにインストールされた場合は、/opt/ibm/xlC/13.1.0/bin/ にあります。詳しくは、「*XL C/C++ インストール・ガイド*」の『*XL C/C++ 呼び出しへのパスが組み込まれるように PATH 環境変数を設定する*』を参照してください。

TMPDIR
コンパイル実行中に一時ファイルが作成されるディレクトリーをオプションで指定します。高レベルの最適化ではページング・ファイルと一時ファイル用に大量のディスク・スペースが必要となる場合があるため、デフォルト・ロケーションの */tmp/* では容量が不足する可能性があります。その場合は、この環境変数を使用して別のディレクトリーを指定することができます。

XLC_USR_CONFIG

コンパイラーが使用するカスタム構成ファイルのロケーションを指定します。ここで指定するファイル名には、絶対パスを追加する必要があります。コンパイラーは、最初にこのファイルの定義を処理してから、デフォルトのシステム構成ファイルまたは **-F** オプションで指定されたカスタマイズ・ファイルの定義を処理します。詳細は、41 ページの『カスタム・コンパイラー構成ファイルの使用』を参照してください。

ランタイム環境変数

以下に示すのは、システム・ローダーあるいはアプリケーションが実行時に使用する環境変数です。これらすべての変数はオプションです。

LD_LIBRARY_PATH

アプリケーションの実行時に、動的にリンクされたライブラリーの代替のディレクトリー検索パスを指定します。アプリケーションが必要とする共有ライブラリーが、リンク時に指定されなかった代替ディレクトリーに移動され、実行可能ファイルを再リンクしない場合は、ダイナミック・リンカーがこれらのライブラリーを実行時に検出するよう環境変数を設定できます。この環境変数について詳しくは、ご使用のオペレーティング・システムの資料を参照してください。

PDFDIR

-qpdf1 オプションでコンパイルしたアプリケーションの実行時にプロファイル情報を保管するディレクトリーをオプションで指定します。デフォルト値が設定解除され、コンパイラーはプロファイル・データ・ファイルを現行作業ディレクトリーに配置します。 PDFDIR 環境変数が設定されているが、指定したディレクトリーが存在しない場合は、コンパイラーは警告メッセージを出します。 **-qpdf2** オプションを指定してプログラムを再コンパイルまたは再リンクした場合、コンパイラーは、このディレクトリーに保存されたデータを使用して、アプリケーションを最適化します。 Profile-Directed Feedback (PDF) を使用する場合は、この変数を絶対パスに設定することをお勧めします。詳しくは、『 292 ページの『-qpdf1、-qpdf2』』を参照してください。

PDF_PM_EVENT

-qpdf1=level=2 を指定してコンパイルされたアプリケーションを実行し、各種レベルのキャッシュ・ミス・プロファイル情報を収集する場合は、PDF_PM_EVENT 環境変数を L1MISS、L2MISS、または L3MISS (使用可能な場合) に適宜設定します。

PDF_BIND_PROCESSOR

プロセスを特定のプロセッサーにバインドする場合は、PDF_BIND_PROCESSOR 環境変数を指定して、プロセス・ツリーを実行可能ファイルから異なるプロセッサーへバインドできます。デフォルトではプロセッサー 0 が設定されます。

PDF_WL_ID

この環境変数は、ユーザー・プログラムの複数のトレーニング実行によって生成される PDF カウンターのセットを区別するため使用されます。それぞれが異なる入力を受け取ります。

デフォルトでは、トレーニング実行の PDF カウンターは、最初のトレーニング実行が PDF カウンターの最初で唯一のセットに追加された後に実行されます。この動作は、各 PDF トレーニング実行の前に PDF_WL_ID 環境変数を設定することによって変更できます。 PDF_WL_ID は、1 から 65535 までの範囲の整数値に設定されます。次いで PDF ランタイム・ライブラリーはこの数を使用して、このトレーニング実行によって生成される PDF カウンターのセットにタグを付けます。すべてのトレーニング実行が完了すると、PDF プロファイル・ファイルには、各セットが ID 番号を持つ、複数のセットの PDF カウンターが入ります。

並列処理のための環境変数

XLSMPOPTS 環境変数はループの並列化を使用してプログラム・ランタイムのオプションを設定します。XLSMPOPTS 環境変数のサブオプションについては、『XLSMPOPTS』を参照してください。

並列化に OpenMPの構造を使用する場合は、34 ページの『OpenMP の環境変数』で説明しているように、OMP 環境変数を使用して実行時オプションを指定することもできます。

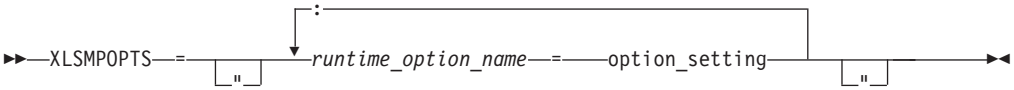
OMP- 環境変数および XLSMPOPTS 環境変数によって指定された実行時オプションが矛盾する場合は、OMP オプションが優先されます。

注: 並列化されたプログラム・コードをコンパイルするときは、スレッド・セーフ・コンパイラー・モードの呼び出しを使用する必要があります。

関連情報
- 459 ページの『並列処理のためのプラグマ・ディレクティブ』
- 645 ページの『並列処理のための組み込み関数』

XLSMPOPTS

並列処理に影響を及ぼす実行時オプションは、XLSMPOPTS 環境変数を使用して指定することができます。この環境変数はアプリケーションを実行する前に設定する必要があり、以下の形式の基本構文を使用します。



オプションの名前および設定は、大文字でも小文字でも指定できます。コロンおよび等号の前後にブランクを追加して、読みやすくすることもできます。ただし、XLSMPOPTS オプション・ストリングに組み込みブランクが含まれる場合は、オプション・ストリング全体を二重引用符 (") で囲む必要があります。

例えば、プログラム・ランタイムで 4 つのスレッドを作成し、チャンク・サイズが 5 の動的スケジューリングを使用するには、XLSMPOPTS 環境変数を以下のように設定します。

```
XLSMPOPTS=PARTHDS=4:SCHEDULE=DYNAMIC=5
```

以下に示すのは、XLSMPOPTS 環境変数に使用可能な実行時オプション設定です。

スケジューリング・オプションは以下のとおりです。

schedule

他のスケジューリング・アルゴリズムがソース・コードに明示的に割り当てられていないループに使用されている、スケジューリング・アルゴリズムの種類およびチャンク・サイズ (*n*) を指定します。

使用するスケジューリングの種類およびチャンク・サイズによって異なる方法で、作業はスレッドに割り当てられます。オーバーヘッドとロード・バランシングのいずれを優先するかを考えてチャンクの細分度を選択する必要があります。このオプションの構文は **schedule=***suboption* で、ここでのサブオプションは以下のように定義されます。

affinity[=*n***]**

最初にループの繰り返しは、**ceiling**(*number_of_iterations*/*number_of_threads*) 回の繰り返しを含む *n* 個の区画に分割されます。各区画は、最初にスレッドに割り当てられてから、それぞれ *n* 回の繰り返しを含むチャンクにさらに分割されます。*n* が指定されていないと、チャンクは **ceiling**(*number_of_iterations_left_in_partition* / 2) 回のループの繰り返しを含みます。

スレッドが解放されると、このスレッドに最初に割り当てられていた区画から次のチャンクが取得されます。その区画の中にチャンクがなくなると、スレッドは、別のスレッドに最初に割り当てられた区画から使用可能な次のチャンクを取得します。

最初にスリープ・スレッドに割り当てられている区画での作業は、アクティブなスレッドによって完了します。

類縁性スケジューリング・タイプは、OpenMP API 標準の一部ではありません。

注: このサブオプションは非推奨であり、将来のリリースでは除去される可能性があります。類似の機能には、**guided** サブオプションを使用できます。

dynamic[=*n***]**

ループの繰り返しは、それぞれ *n* 回の連続する繰り返しを含むチャンクに分割されます。最後のチャンクに含まれる繰り返しは、*n* 回よりも少ない繰り返しになる可能性があります。 *n* が指定されていない場合、デフォルトのチャンク・サイズは 1 です。

各スレッドには、最初に 1 つのチャンクが割り当てられています。割り当てられたチャンクをスレッドが処理し終わると、残りのチャンクは「先着順」でスレッドに割り当てられます。

guided[=*n***]**

ループの繰り返しは、チャンクの最小サイズである *n* ループの繰り返しに達するまで、徐々により小さなチャンクに分割されます。 *n* が指定されなかった場合、*n* のデフォルト値は 1 回の繰り返しです。

アクティブ・スレッドには、チャンクが「先着順実行」の基準で割り当てられます。最初のチャンクには **ceiling**(*number_of_iterations*/*number_of_threads*) 繰り返しが含まれます。それ以降のチャンクは、

ceiling(*number_of_iterations_left / number_of_threads*) 繰り返しを含みます。最後のチャンクに含まれる繰り返しは、n 回よりも少ない繰り返しになる可能性があります。

static[=*n*]

ループの繰り返しが、それぞれ *n* 回の繰り返しを含むチャンクに分割されます。各スレッドには、「ラウンドロビン」方式でチャンクが割り当てられます。これをブロック巡回スケジューリング と言います。 *n* の値が 1 である場合は、*n* の値が 1 のスケジューリング・タイプは、特に巡回スケジューリングと呼ばれます。

n を指定しない場合、チャンクには **floor**(*number_of_iterations*/*number_of_threads*) 回の繰り返しが含まれます。最初の **remainder**(*number_of_iterations*/*number_of_threads*) チャンクには複数の繰り返しがあります。各スレッドには、これらのチャンクの 1 つが割り当てられます。これをブロック・スケジューリング と言います。

スレッドは、スリープ状態で作業を割り当てられている場合、その作業を完了できるようスリープ状態から解除されます。

n 1 以上の値の整数代入式でなければなりません。

サブオプションなしで **schedule** を指定すると、**schedule=runtime** を指定した場合と同じになります。

並列環境のオプションは以下のとおりです。

parthds=*num*

必要な並列スレッドの数 (*num*)。通常この数は、システムで使用可能なプロセッサーの数と同じです。

アプリケーションによっては、使用可能なプロセッサーの最大数を超えてスレッドを使用することはできません。その他のアプリケーションでは、存在するプロセッサーの数より多くのスレッドを使用するとパフォーマンスが大幅に改善されます。このオプションにより、プログラムの実行に使用されるユーザー・スレッドの数を完全に制御することができます。

num のデフォルト値は、システムで使用可能なプロセッサーの数です。

usrthds=*num*

コードによる明示的なスレッド作成時に、明示的に作成されることを期待する最大スレッド数 (*num*) を指定します。 *num* のデフォルト値は 0 です。

stack=*num*

スレッドのスタックに必要な最大量のスペースをバイト単位 (*num*) で指定します。*num* のデフォルト値は 4194304 です。

使用可能な上限を超えないように *num* を設定します。*num* には、システム・リソースによって課される制限またはスタック・サイズ ulimit のいずれか小さい方を超えない数値を指定できます。上限を超えるアプリケーションは、セグメンテーション障害の原因になる場合があります。

stackcheck[=*num*]

-qsmp=stackcheck が有効なとき、実行時に、スレーブ・スレッドの有無のチェックをスタック・オーバーフローができるようにします。*num* は、スタックのサイズ (バイト単位) です。これは非ゼロの正数でなければなりま

せん。残りのスタック・サイズがこの値より小さい場合は、ランタイム警告メッセージが発行されます。 *num* の値を指定しないと、デフォルト値は 4096 バイトになります。このオプションが有効になるのは、コンパイル時に **-qsmp=stackcheck** も指定されていた場合です。詳しくは、『 335 ページの『-qsmp』』を参照してください。

startproc=*cpu_id*
スレッドのバインディングを使用可能に設定し、最初のスレッドのバインド先となる *cpu_id* を指定します。提供された値が使用可能プロセッサーの範囲外の場合、警告メッセージを発行し、スレッドはバインドされません。

procs=cpu_id*[,cpu_id,...]*
スレッドのバインディングを使用可能に設定し、スレッドのバインド先となる *cpu_id* のリストを指定します。

stride=*num*
後続のスレッドのバインド先となる *cpu_id* を決定するために使用する増分を指定します。*num* は 1 以上でなければなりません。提供された値によってスレッドが使用可能プロセッサーの範囲外の CPU にバインドされる場合は、警告メッセージが発行され、スレッドはバインドされません。

パフォーマンス・チューニング・オプションは以下のとおりです。

spins=*num*
譲歩するまでのループ・スピンまたは繰り返しの数を示します。

スレッドは、作業を完了すると、新しい作業を探して短いループの実行を続行します。各作業待ち状態中に、作業キューの完全スキャンが 1 回実行されます。拡張作業待ち状態によって特定のアプリケーションの応答性を高くすることができますが、定期的にスキャンして他のアプリケーションからの要求に譲歩するようにスレッドに指示しない限り、システム全体としての応答が低下する場合もあります。

spins および **yields** の両方を 0 に設定することによって、ベンチマークを目的として完全な作業待ち状態にすることができます。

num のデフォルト値は 100 です。

yields=*num*
スリープするまでの譲歩の数を示します。

スレッドがスリープすると、実行する処理があることが別のスレッドによって示されるまで完全に実行が中断されます。これによりシステムの使用効率は向上しますが、アプリケーションに余分なシステム・オーバーヘッドが加わります。

num のデフォルト値は 100 です。

delays=*num*
作業キューの各スキャンの間の、処理なしの遅延時間の長さを示します。遅延の各単位は、メモリーへのアクセスがない遅延ループを一度実行することによって行われます。

num のデフォルト値は 500 です。

動的プロファイルのオプションは以下のとおりです。

profilefreq=*n*

並列実行または順次実行の適切性を判断するために、動的プロファイラーがループに戻る頻度を指定します。ランタイム・ライブラリーは動的プロファイルを使用して、自動的に並列化されるループのパフォーマンスを動的に調整します。動的プロファイルはループの実行時間に関する情報を収集して、次回ループを実行するときに順次実行すべきか並列に実行すべきかを判別します。 実行時間のしきい値は、以下で説明する **parthreshold** および **seqthreshold** 動的プロファイル・オプションによって設定します。

このオプションに使用できるのは 0 から 32 までの数値です。*num* が 0 の場合は、プロファイルはすべてオフになり、プロファイルのために起こるオーバーヘッドはなくなります。 *num* が 0 より大きい場合は、ループを *num* 回実行するごとに 1 回、ループの実行時間がモニターされます。 *num* のデフォルトは 16 です。*num* の値が 32 を超えている場合は、32 に変更されます。

動的プロファイルは、ユーザー指定の並列ループには適用できないのでご注意ください。

parthreshold=*num*

各ループが順次実行しなければならない時間をミリ秒で指定します。*num* を 0 に設定すると、コンパイラーによって並列化されたすべてのループが並列で実行されます。 デフォルト設定は 0.2 ミリ秒です。この場合、ループの並列実行に必要な時間が 0.2 ミリ秒より短いと、ループは順次実行に変更されることを意味します。

通常、*num* は並列化オーバーヘッドと同等に設定されています。並列化ループでの計算が非常に小規模で、これらのループの実行に必要な時間が主に並列化のセットアップに費やされるのであれば、これらのループを順次実行したほうがパフォーマンスは向上します。

seqthreshold=*num*

前に動的プロファイラーによって順次化されたループを並列ループに戻す場合の目安時間をミリ秒で指定します。デフォルト設定は 5 ミリ秒です。この場合、ループの順次実行に必要な時間が 5 ミリ秒を超えると、ループは並列実行に変更されることを意味します。

seqthreshold は、**parthreshold** の正反対の機能を持ちます。

OpenMP の環境変数

並列処理に影響を与える OpenMP 実行時オプションは OMP 環境変数を指定することにより設定されます。これらの環境変数は以下の形式の構文を使用します。

OMP 環境変数が明示的に設定されていない場合は、そのデフォルト設定が使用されます。

OpenMP 仕様の情報については、www.openmp.org/specs を参照してください。

OMP_DISPLAY_ENV: OMP_DISPLAY_ENV 環境変数は、環境変数に関連付けられた内部制御変数 (ICV) の値、およびランタイム・ライブラリーに関するビルド固有の情報を表示します。

OMP_DISPLAY_ENV は、次の場合に役立ちます。

- ランタイム・ライブラリーが OpenMP プログラムに静的にリンクされている場合、OMP_DISPLAY_ENV を使用して、リンク時に使用されるライブラリーのバージョンを確認できます。
- ランタイム・ライブラリーが OpenMP プログラムに動的にリンクされている場合、OMP_DISPLAY_ENV を使用して、ランタイム時に使用されるライブラリーのバージョンを確認できます。
- OMP_DISPLAY_ENV を使用して、ランタイム環境の現在の設定を確認できます。

デフォルトでは、何の情報も表示されません。

この環境変数の構文は以下のとおりです。

注: 値 TRUE、FALSE、および VERBOSE には大/小文字の区別はありません。

TRUE
_OPENMP マクロによって定義された OpenMP のバージョン番号、および OpenMP 環境変数の ICV の初期値を表示します。

FALSE
情報を表示しないようランタイム環境に指示します。

VERBOSE
ビルド固有の情報、OpenMP 環境変数に関連付けられている ICV 値、および XLSMPOPTS 環境変数の設定を表示します。

例

export OMP_DISPLAY_ENV=TRUE コマンドを入力すると、次の例のような出力が得られます。

```
OPENMP DISPLAY ENVIRONMENT BEGIN

   OMP_DISPLAY_ENV='TRUE'


  _OPENMP='201106'

   OMP_DYNAMIC='FALSE'

   OMP_MAX_ACTIVE_LEVELS='5'

   OMP_NESTED='FALSE'

   OMP_NUM_THREADS='64'

   OMP_PLACES='{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},
   {14,15},{16,17},{18,19},{20,21},{22,23},{24,25},{26,27},
   {28,29},{30,31},{32,33},{34,35},{36,37},{38,39},{40,41},
   {42,43},{44,45},{46,47},{48,49},{50,51},{52,53},{54,55},
   {56,57},{58,59},{60,61},{62,63}'
```

```
  OMP_PROC_BIND='FALSE'
  OMP_SCHEDULE='STATIC,0'
  OMP_STACKSIZE='4194304'
  OMP_THREAD_LIMIT='64'
  OMP_WAIT_POLICY='PASSIVE'
 OPENMP DISPLAY ENVIRONMENT END
```

関連情報

30 ページの『XLSMPOPTS』

OMP_DYNAMIC: OMP_DYNAMIC 環境変数は、並列領域の実行に使用可能なスレッドの数の動的調整を使用可能または使用不可にします。

これを TRUE に設定した場合、並列領域の実行に使用可能なスレッドの数を実行時に調整して、システム・リソースを最大限に活用できます。詳しくは、『30 ページの『XLSMPOPTS』』にある **profilefreq=***num* の説明を参照してください。

FALSE に設定されている場合、動的調整は使用不可になります。

デフォルト設定は TRUE です。

OMP_MAX_ACTIVE_LEVELS:
OMP_MAX_ACTIVE_LEVELS を使用して、*max-active-levels-var* 内部制御変数を設定します。これは、アクティブなネストされた並列領域の最大数を制御します。

n ネストされたアクティブな並列領域の最大数。正のスカラー整数でなければなりません。指定できる最大値は 5 です。

ネストされた並列処理が無効なプログラムでは、初期値は 1 になります。ネストされた並列処理が有効なプログラムでは、初期値は 1 より大きくなります。関数 **omp_get_max_active_levels** を使用すると、実行時にこの値を取得できます。

OMP_NESTED: OMP_NESTED=TRUE|FALSE 環境変数はネストされた並列処理を使用可能に設定したり、使用不可に設定したりします。

OMP_NESTED 設定は、**omp_set_nested** ランタイム・ライブラリー関数を呼び出すことによってオーバーライドできます。

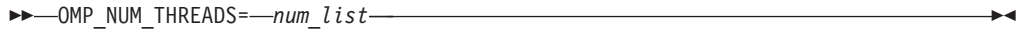
ネストされた並列性が使用不可になっている場合は、ネストされた並列領域は直列化されて、現行スレッドで実行されます。

OMP_NESTED のデフォルト値は FALSE です。

注: 並列領域内およびそのネストした並列領域内のスレッド数が、使用可能なプロセッサー数を超えると、プログラムで性能低下が発生する可能性があります。

OMP_NUM_THREADS: OMP_NUM_THREADS 環境変数は、並列領域で使用するスレッドの数を指定します。

この環境変数の構文は以下のとおりです。



num_list
コンマで区切られた、1 つ以上の正の整数値のリスト。

OMP_NUM_THREADS を設定しなかった場合は、使用可能なプロセッサーの数がデフォルト値になり、最初に検出された並列構文に対して新規チームが形成されます。ネストされた並列処理が使用不可になっている場合、すべてのネストされた並列構文は、単一のスレッドによって実行されます。

num_list に単一の値が入っており、スレッド数の動的調整が使用可能であり (**OMP_DYNAMIC** が true に設定されている)、**num_threads** 節のない並列構文が検出された場合は、その値が、検出された並列構文に新規チームを形成するために使用可能なスレッドの最大数になります。

num_list に単一の値が入っており、スレッド数の動的調整が使用可能ではなく (**OMP_DYNAMIC** が false に設定されている)、**num_threads** 節のない並列構文が検出された場合は、その値が、検出された並列構文に新規チームを形成するために使用可能なスレッドの正確な数になります。

num_list に複数の値が入っており、スレッド数の動的調整が使用可能であり (**OMP_DYNAMIC** が true に設定されている)、**num_threads** 節のない並列構文が検出された場合は、最初の値が、検出された並列構文に新規チームを形成するために使用可能なスレッドの最大数になります。検出された構文に入った後に最初の値が削除され、残りの値が新しい *num_list* を形成します。次に、検出された並列構文内にある近接ネストされた並列構文について、新しい *num_list* が同様に使用されます。

num_list に複数の値が入っており、スレッド数の動的調整が使用可能ではなく (**OMP_DYNAMIC** が false に設定されている)、**num_threads** 節のない並列構文が検出された場合は、最初の値が、検出された並列構文に新規チームを形成するために使用可能なスレッドの正確な数になります。検出された構文に入った後に最初の値が削除され、残りの値が新しい *num_list* を形成します。次に、検出された並列構文内にある近接ネストされた並列構文について、新しい *num_list* が同様に使用されます。

注: 並列領域の数が *num_list* の値の数以上である場合、**omp_get_max_threads** 関数は、並列領域の *num_list* の最後の値を返します。

要求されたスレッドの数が、使用可能なシステム・リソースを超過した場合、プログラムは停止します。

omp_set_num_threads 関数は、*num_list* の最初の値を設定します。
omp_get_max_threads 関数は、*num_list* の最初の値を返します。

特定の並列領域のスレッド数を、異なる設定を使用して複数回指定した場合、コンパイラーは以下の優先順位を使用して、有効になる設定を決定します。

1. **num_threads** 節を使用して設定されたスレッド数は、**omp_set_num_threads** 関数を使用して設定されたものより優先されます。
2. **omp_set_num_threads** 関数を使用して設定されたスレッド数は、OMP_NUM_THREADS 環境変数を使用して設定されたものより優先されます。
3. OMP_NUM_THREADS 環境変数を使用して設定されたスレッド数は、XLSMPOPTS 環境変数の **PARTHDS** サブオプションを使用して設定されたものより優先されます。

例

```
export OMP_NUM_THREADS=3,4,5
export OMP_DYNAMIC=false

// omp_get_max_threads() returns 3

#pragma omp parallel
{
// Three threads running the parallel region
// omp_get_max_threads() returns 4

  #pragma omp parallel if(0)
  {
  // One thread running the parallel region
  // omp_get_max_threads() returns 5

    #pragma omp parallel
    {
    // Five threads running the parallel region
    // omp_get_max_threads() returns 5
    }
  }
}
```

OMP_PROC_BIND: **OMP_PROC_BIND** 環境変数は、OpenMP スレッドをプロセッサー間で移動させることができるかどうかを制御します。

OMP_PROC_BIND 構文

TRUE
スレッドをプロセッサーにバインドします。

FALSE
スレッドをプロセッサー間で移動させることを許可し。

OMP_PROC_BIND 環境変数と **XLSMPOPTS** 環境変数は、以下の規則に従って相互に対話します。

表 8. スレッド・バインド規則の要約

OMP_PROC_BIND 設定	**XLSMPOPTS 設定**	**スレッド・バインド結果**
OMP_PROC_BIND の設定なし	**XLSMPOPTS** の設定なし	スレッドが見つかりません。
	XLSMPOPTS が **startproc/stride** または **procs** に設定されている	スレッドは、**XLSMPOPTS** の設定に従ってバインドされます。
	XLSMPOPTS の設定が無効である	スレッドが見つかりません。
OMP_PROC_BIND = TRUE	**XLSMPOPTS** の設定なし	スレッドが見つかります。
	XLSMPOPTS が **startproc/stride** または **procs** に設定されている	スレッドは、**XLSMPOPTS**[1] の設定に従ってバインドされます。
	XLSMPOPTS の設定が無効である	スレッドが見つかります。
OMP_PROC_BIND = FALSE	**XLSMPOPTS** の設定なし	スレッドが見つかりません。
	XLSMPOPTS が設定されている (**startproc/stride** または **procs**)	
	XLSMPOPTS の設定が無効である	

注:

1. **procs** を設定しており、指定した CPU ID の数が、プログラムによって使用されるスレッド数より少ない場合には、残りのスレッドも、リストされた CPU ID にバインドされます。ただしこの場合、特定の順序にはなりません。 **XLSMPOPTS=startproc** を使用すると、**startproc** で指定した値が CPU 数より少なく、**stride** での指定値を適用すると、使用可能な場所の範囲を超えてスレッドが CPU にバインドされてしまう場合は、バインドされるスレッドとバインドされないスレッドが発生します。

OMP_PROC_BIND 環境変数は、OpenMP スレッドをマイグレーションできるかどうかを制御するための移植可能な手段です。**XLSMPOPTS** 環境変数の **startproc/stride** または **procs** サブオプションにより、プロセッサーへの OpenMP スレッドのバインドを詳細に制御できます (これらのサブオプションはいずれも IBM 拡張です)。アプリケーションの移植性を重視する場合は、**OMP_PROC_BIND** 環境変数のみを使用してスレッドのバインディングを制御してください。

OMP_SCHEDULE: OMP_SCHEDULE 環境変数は、**OpenMP schedule** 節でランタイム・スケジュール・タイプに明示的に割り当てられたループに対して使用されるスケジュール・タイプを指定します。

例を以下に示します。

```
OMP_SCHEDULE="guided, 4"
```

スケジュール・タイプの有効なオプションは、以下のとおりです。

- auto
- dynamic[, *n*]
- guided[, *n*]
- static[, *n*]

n を使用してチャンク・サイズを指定する場合、*n* の値は正の整数でなければなりません。

デフォルトのスケジュール・タイプは **auto** です。

関連資料:

654 ページの『omp_set_schedule』

654 ページの『omp_get_schedule』

OMP_STACKSIZE:

OMP_STACKSIZE 環境変数は、OpenMP ランタイムによって作成されたスレッドのスタック・サイズを示します。OMP_STACKSIZE は、*stacksize-var* 内部制御変数の値を設定します。OMP_STACKSIZE は、マスター・スレッドのスタック・サイズを制御しません。構文は次のとおりです。

デフォルトでは、サイズ値はキロバイトで表されます。また、サイズをバイト、キロバイト、メガバイト、またはギガバイトで示す場合は、それぞれサフィックスとして B、K、M、または G を使用することができます。サイズ値とサフィックスの間とその両側には、空白を入れてもかまいません。例えば、以下の 2 つの例はいずれも 10 メガバイトのスタック・サイズを示しています。

```
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
```

OMP_STACKSIZE が設定されない場合、*stacksize-var* 内部制御変数の初期値はデフォルト値に設定されます。 32 ビット・モードのデフォルト値は 256M です。64 ビット・モードの場合、デフォルトはシステム・リソースによって課せられる制限を最大値とします。コンパイラーが指定されたスタック・サイズを使用できない場合、または OMP_STACKSIZE が正しいフォーマットに従っていない場合、コンパイラーはこの環境変数をデフォルト値に設定します。 XLSMPOPTS 環境変数の STACK サブオプションと OMP_STACKSIZE 環境変数が指定されている場合は、OMP_STACKSIZE 環境変数が優先されます。

OMP_THREAD_LIMIT:

OMP_THREAD_LIMIT 環境変数は、プログラム全体で使用する OpenMP スレッド数を設定します。構文は次のとおりです。

n プログラム全体で使用する OpenMP スレッド数。正のスカラー整数でなければなりません。

OMP_THREAD_LIMIT の値は正整数です。ネストされた並列処理が使用可能な場合、**OMP_THREAD_LIMIT** に指定した値が並列領域の動作に影響を与える可能性があります。例えば、**OMP_THREAD_LIMIT** の値がプログラムで必要なスレッド数より大幅に小さい場合 (例: **OMP_THREAD_LIMIT**=1)、並列領域は並列ではなく順次実行されます。

OMP_THREAD_LIMIT 環境変数を設定せず、**OMP_NUM_THREADS** 環境変数を単一の値に設定した場合、**OMP_THREAD_LIMIT** のデフォルト値は、**OMP_NUM_THREADS** の値と使用可能なプロセッサーの数のうち、いずれか大きい値になります。

OMP_THREAD_LIMIT 環境変数を設定せず、**OMP_NUM_THREADS** 環境変数をリストに設定した場合、**OMP_THREAD_LIMIT** のデフォルト値は、リストされたすべての数値の積と使用可能なプロセッサーの数のうち、いずれか大きい値になります。

OMP_THREAD_LIMIT 環境変数も **OMP_NUM_THREADS** 環境変数も設定しない場合、**OMP_THREAD_LIMIT** のデフォルト値は使用可能なプロセッサーの数になります。

OMP_WAIT_POLICY:
OMP_WAIT_POLICY 環境変数は、プログラムの実行時における待機スレッドの優先動作に関して、コンパイラーにヒントを与えます。 OMP_WAIT_POLICY 環境変数では、*wait-policy-var* 内部制御変数の値を設定します。

構文は次のとおりです。

OMP_WAIT_POLICY のデフォルト値は PASSIVE です。

待機スレッドを通常はアクティブにする場合は、ACTIVE を使用します。 ACTIVE では、スレッドは待機中、可能な場合にプロセッサー・サイクルを消費します。

待機スレッドを通常はパッシブにする場合は、PASSIVE を使用します。これは、スレッドが待機中にプロセッサー・サイクルを消費しない設定が優先されるということです。例えば、待機スレッドがスリープするか、プロセッサーを他のスレッドに提供することを優先させる場合です。

注: OMP_WAIT_POLICY 環境変数が設定され、XLSMPOPTS 環境変数の SPINS、YIELDS、または DELAYS サブオプションが指定されている場合は、OMP_WAIT_POLICY が優先されます。

カスタム・コンパイラー構成ファイルの使用

XL C/C++ コンパイラーは、デフォルト構成ファイル /opt/ibm/xlC/13.1.0/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* を生成します。例えば、/opt/ibm/xlC/13.1.0/etc/xlc.cfg.sles11.gcc432 または /opt/ibm/xlC/13.1.0/etc/xlc.cfg.rhel6.2.gcc446 です。(インストール実行中に構成ファイルの生成に使用できる多様なツールについて詳しくは、「*XL C/C++ インストール・ガイド*」を参照してください。) 構成ファイルは、コンパイラーが呼び出し時に使用する情報を指定します。

単一ユーザー・システムで稼働している場合や、コンパイル・スクリプトや Make ファイルを持つコンパイル環境がすでにある場合は、デフォルトの構成ファイルをそのままにしておくことができます。

いくつかのコンパイラー・オプション・セットからユーザーが選択できるようにしたい場合は、特定の必要に合わせたカスタム構成ファイルを使用することもできます。例えば、**xlc** コンパイラー呼び出しコマンドを使用したコンパイルができるよう、**-qlist** をデフォルトで有効にすることもできます。これにより、**xlc** コマンドでコンパイラーを呼び出すたびに **-qnolist** が自動的に有効になるため、コンパイル時に毎回コマンド行でこのオプションを指定する必要がなくなります。

構成ファイルをカスタマイズする方法はいくつかあります。

- デフォルトの構成ファイルを直接編集する。 この場合、カスタマイズされたオプションは、すべてのコンパイルですべてのユーザーに対して適用されます。このオプションの欠点は、コンパイラーを更新するたびに提供される新規のデフォルト構成ファイルにカスタマイズ内容を再適用しなければならない点です。
- デフォルトの構成ファイルを、コンパイル時に **-F** オプションで指定するカスタマイズ・コピーの基盤として使用する。この場合、カスタム・ファイルは、コンパイル単位のデフォルト・ファイルをオーバーライドします。

  注: このオプションでは、コンパイラーにサービスを適用後、カスタマイズを再適用する必要があります。
- XLC_USR_CONFIG 環境変数を使用して、コンパイル時に指定するカスタムまたはユーザー定義の構成ファイルを作成する。この場合、カスタムのユーザー定義ファイルは、デフォルトの構成ファイルをオーバーライドするのではなく補完します。これらのファイルはコンパイル単位またはグローバル単位で指定することもできます。このオプションの利点は、更新中に新規のシステム構成ファイルをインストールする際、既存のカスタム構成ファイルを変更する必要がない点です。 カスタム、およびユーザー定義の構成ファイルの作成手順を以下に示します。

関連情報:

- 156 ページの『-F』
- 28 ページの『コンパイル時およびリンク時の環境変数』

カスタム構成ファイルの作成

XLC_USR_CONFIG 環境変数を使用して、コンパイラーにカスタムのユーザー定義構成ファイルを使用するよう指示すると、コンパイラーはそのユーザー定義構成ファイルの設定を検討および処理してから、デフォルトのシステム構成ファイルの設定を確認します。

カスタムのユーザー定義構成ファイルを作成するには、**use** 属性の複数レベルを指定するスタンザを追加します。ユーザー定義の構成ファイルは、システム構成ファイルで指定された定義だけでなく、同じファイル内のいずれかで指定された定義も参照できます。特定のコンパイルの場合、コンパイラーは特定スタンザの検索をユーザー定義構成ファイルの先頭から開始してから、use 属性で指定された他のスタンザ (システム構成ファイルで指定されたスタンザを含む) を検索します。

use 属性で指定されたスタンザの名前が現在処理中のスタンザの名前と異なる場合、use スタンザの検索はユーザー定義構成ファイルの先頭から開始します。これは、次の例に示すスタンザ A、C、および D の場合です。例での 2 つの B スタン

ザのように、**use** 属性のスタンザの名前が現在処理中のスタンザの名前と同じ場合、**use** スタンザの検索は現行スタンザのその位置から開始します。

以下の例で、**use** 属性の複数レベルを使用する方法を示します。この例では、**options** 属性を使用して **use** 属性の機能方法を説明していますが、**libraries** などの他の属性を使用することもできます。

```
A: use =DEFLT
    options=<set of options A>
B: use =B
    options=<set of options B1>
B: use =D
    options=<set of options B2>
C: use =A
    options=<set of options C>
D: use =A
    options=<set of options D>
DEFLT:
    options=<set of options Z>
```

図 1. サンプル構成ファイル

この例では以下のとおりになります。

- スタンザ A ではオプション・セット *A* および *Z* が使用される
- スタンザ B ではオプション・セット *B1*、*B2*、*D*、*A*、および *Z* が使用される
- スタンザ C ではオプション・セット *C*、*A*、および *Z* が使用される
- スタンザ D ではオプション・セット *D*、*A*、および *Z* が使用される

属性は、スタンザと同じ順序で処理されます。オプションが指定される順序は、オプション解決にとって重要です。通常、あるオプションが複数回指定されている場合、そのオプションの最後に指定されたインスタンスが優先されます。

デフォルトで、構成ファイルのスタンザで定義された値は、前に処理されたスタンザで指定された値のリストに追加されます。例えば、XLC_USR_CONFIG 環境変数が ~/userconfig1 にあるユーザー定義のカスタム構成ファイルを指すように設定されているとします。以下の例で示す、ユーザー定義の構成ファイルおよびデフォルトの構成ファイルの場合、コンパイラーは、ユーザー定義構成ファイルの **xlc** スタンザを参照し、この構成ファイルで指定されたオプション・セットを *A1*、*A*、*D*、および *C* の順序で使用します。

```
xlc:  use=xlc
    options= <A1>

DEFLT: use=DEFLT
    options=<D>
```

図 2. カスタムのユーザー定義構成ファイル ~/userconfig1

```
xlc:  use=DEFLT
    options=<A>

DEFLT:
    options=<C>
```

図 3. デフォルトの構成ファイル xlc.cfg

属性値のデフォルト順序のオーバーライド

属性値のデフォルト順序をオーバーライドするには、構成ファイルの属性の代入演算子 (=) を変更します。

表 9. 代入演算子および属性の順序

代入演算子	説明
-=	デフォルトの検索順序によって決定される値の前に、以下の値を付加します。
:=	デフォルトの検索順序によって決定される値を、以下の値で置き換えます。
+=	デフォルトの検索順序によって決定される値の後に、以下の値を付加します。

例えば、XLC_USR_CONFIG 環境変数が ~/userconfig2 にあるカスタムのユーザー定義構成ファイルを指すように設定されているとします。

カスタムのユーザー定義構成ファイル
~/userconfig2

```
xlc_prepend: use=xlc
             options-=<B1>
xlc_replace: use=xlc
             options:=<B2>
xlc_append: use=xlc
             options+=<B3>

DEFLT: use=DEFLT
       options=<D>
```

デフォルトの構成ファイル xlc.cfg

```
xlc: use=DEFLT
     options=<B>

DEFLT:
     options=<C>
```

上記の構成ファイル内のスタンザは、以下のオプション・セットを以下の順序で使用します。

1. スタンザ xlc は *B*、*D*、および *C* を使用する
2. スタンザ xlc_prepend は *B1*、*B*、*D*、*C* を使用する
3. スタンザ xlc_replace は *B2* を使用する
4. スタンザ xlc_append は *B*、*D*、*C* および *B3* を使用する

属性を 2 回以上指定する場合に、代入演算子を使用することもできます。例を以下に示します。

```
xlc:
    use=xlc
    options-=-Isome_include_path
    options+=some options
```

図 4. 追加の代入演算の使用

カスタム構成ファイルのスタンザの例

```
DEFLT: use=DEFLT
    options = -g
```

この例では、**-g** オプションをすべてのコンパイルで使用することを指定しています。

`xlc: use=xlc options+=-qlist`
`xlc_r: use=xlc_r`
`options+=-qlist`
	この例は、**xlc** および **xlc_r** コマンドによって呼び出されたすべてのコンパイルで、**-qlist** が使用されることを指定します。このように **-qlist** を使用すると、システム構成ファイルで指定された **-qlist** のデフォルト設定がオーバーライドされます。

`DEFLT: use=DEFLT` `libraries=-L/home/user/lib,-lmylib`	この例では、すべてのコンパイルが /home/user/lib/libmylib.a にリンクされることを指定しています。

Advance Toolchain との IBM XL C/C++ for Linux, V13.1 の併用

IBM XL C/C++ for Linux, V13.1 は、オープン・ソース開発ツールとランタイム・ライブラリーのセットである、IBM Advance Toolchain 7.0 をサポートします。IBM Advance Toolchain 7.0 を使用することで、最新の POWER® ハードウェア機構 (特に、チューニングされたライブラリー) を Linux 上で一層活用できます。Advance Toolchain 7.0 について詳しくは、『IBM Advance Toolchain for PowerLinux™ Documentation』を参照してください。

IBM XL C/C++ for Linux, V13.1 を Advance Toolchain と共に使用するには、以下の手順を実行します。

1. at7.0 rpm パッケージをデフォルトのインストール場所にインストールします。手順については、『IBM Advance Toolchain for PowerLinux Documentation』を参照してください。
2. **new_install** ユーティリティーを実行して、**vac.at.cfg** 構成ファイルを作成します。 **vac.at.cfg** 構成ファイルで、XL C/C++ コンパイラー以外の他のすべてのエンティティーは、Advance Toolchain のエンティティーに送られます。このエンティティーには、リンカー、ヘッダー、およびランタイム・ライブラリーが含まれます。
   - コンパイラーをデフォルトの場所にインストールした場合は、以下のコマンドを発行します。

     ```
     new_install -at
     ```

   - コンパイラーをデフォルト以外のインストール (NDI) の場所にインストールした場合は、以下のコマンドを発行します。

     ```
     new_insatll -at -prefix $ndi_path
     ```

     ここで *$ndi_path* は、コンパイラーのインストール先のディレクトリーです。
3. XL コンパイラーを Advance Toolchain サポートと共に呼び出します。
   - コンパイラーをデフォルトの場所にインストールした場合は、C/C++ コンパイラーとして以下のコマンドを発行します。

     ```
     /opt/ibm/xlC/13.1.0/bin/xlc_at
     /opt/ibm/xlC/13.1.0/bin/xlC_at
     ```

   - コンパイラーを NDI の場所にインストールした場合は、以下のコマンドを発行します。

     ```
     $ndi_path/xlC/13.1.0/bin/xlc_at
     $ndi_path/xlC/13.1.0/bin/xlC_at
     ```

注: XL コンパイラーを Advance Toolchain サポートと共に使用してアプリケーションを作成する場合、アプリケーションは Advance Toolchain のランタイム・ライブラリーに依存するため、Advance Toolchain 環境下でしか実行できなくなります。アプリケーションを他のマシン上で実行するためにコピーする場合は、Advance Toolchain、または少なくとも Advance Toolchain のランタイム・ライブラリーが、それらのマシン上で使用可能であることを確認してください。

gxlc または gxlc++ オプション・マッピングの構成

gxlc および**gxlc++** ユーティリティーは構成ファイル /opt/ibm/xlC/13.1.0/etc/gxlc.cf を使用して GNU C および C++ オプションを XL C/C++ オプションに変換します。gxlc.cfg の各項目は、ユーティリティーがどのように GNU C または C++ オプションを XL C/C++ オプションにマップし、それを処理するかを記述しています。

1 つの項目は、処理命令のフラグのストリング、GNU C/C++ オプションのストリング、および XL C/C++ オプションのストリングで構成されています。この 3 つのフィールドは空白文字で区切られている必要があります。項目に最初の 2 つのフィールドだけが含まれていて、XL C/C++ オプションのストリングが抜けている場合は、2 番目のフィールドの GNU C オプションが **gxlc** または **gxlc++** によって認識され、無音で無視されます。

構成ファイルにコメントを挿入するためには、# 文字が使用されます。コメントはそのコメント独自の行、または項目の最後に入れることができます。

以下の構文が gxlc.cfg 内の項目に使用されます。

```
abcd    "gcc_or_g++_option"    "xlc_or_xlc++_option"
```

ここで、

a
: **no-** をプレフィックスとして追加することによりオプションを使用不可にできます。この値は yes を表す **y** か、no を表す **n** のいずれかになります。例えば、フラグが **y** に設定されている場合は、**finline** は **fno-inline** として使用不可にすることができ、項目は以下のようになります。

  ```
  ynn*        "-finline"                  "-qinline"
  ```

  -fno-inline が指定されていると、ユーティリティーはそれを **-qnoinline** に変換します。

b
: XL C/C++ オプションに関連した値があることをユーティリティーに通知します。この値は yes を表す **y** か、no を表す **n** のいずれかになります。例えば、オプション **-fmyvalue=*n*** が **-qmyvalue=*n*** にマップされている場合は、フラグは y に設定され、項目は以下のようになります。

  ```
  nyn*        "-fmyvalue"             "-qmyvalue"
  ```

  そして、ユーティリティーはこれらのオプションの値を予期します。

c
: オプションの処理を制御します。値は、以下のいずれかになります。

  n
  : *gcc_or_g++_option* フィールドにリストされたオプションを処理するようユーティリティーに命令します。

  i
  : *gcc_or_g++_option* フィールドにリストされたオプションを無視す

るようユーティリティーに命令します。ユーティリティーは、これが実行されたことを示すメッセージを生成し、指定されたオプションの処理を継続します。

e *gcc_or_g++_option* フィールドにリストされたオプションを検出した場合、処理を停止するようユーティリティーに命令します。ユーティリティーはエラー・メッセージも生成します。

例えば、GCC オプション `-I-` がサポートされていないため、**gxlc** または **gxlc++** に無視されなければならないとします。この場合、フラグは `i` に設定され、項目は以下のようになります。

```
nni*        "-I-"
```

ユーティリティーは、このオプションを入力として検出すると、処理を行わず警告を生成します。

d **gxlc** または **gxlc++** はコンパイラーのタイプを基にオプションを組み込んだり無視することができます。値は、以下のいずれかになります。

c C 専用のオプションを変換するようユーティリティーに命令します。

x C++ 専用のオプションを変換するようユーティリティーに命令します。

***** C および C++ 用のオプションを変換するよう **gxlc** または **gxlc++** に命令します。

例えば、**-fwritable-strings** は両方のコンパイラーによってサポートされており、**-qnoro** にマップされます。項目は以下のようになります。

```
nnn*        "-fwritable-strings"        "-qnoro"
```

"*gcc_or_g++_option*"
GNU C/C++ オプションを表すストリングです。このフィールドは必須であり、二重引用符で囲む必要があります。

"*xlc__or_xlc++_option*"
XL C/C++ オプションを表すストリングです。このフィールドはオプショナルであり、指定する場合は、二重引用符で囲む必要があります。ブランクのままにすると、ユーティリティーは、その項目の *gcc_or_g++_option* を無視します。

ある範囲のオプションをマップする項目を作成することが可能です。これはアスタリスク (*) とワイルドカードを使用して実行できます。例えば、GCC **-D** オプションにはユーザー定義名が必須で、オプションの値を取ることができます。以下のオプションのシリーズを持つことが可能です。

```
-DCOUNT1=100
-DCOUNT2=200
-DCOUNT3=300
-DCOUNT4=400
```

このオプションの各バージョンごとに項目を作成する代わりに、以下のような単一項目を作成することができます。

```
nnn*        "-D*"                        "-D*"
```

ここで、アスタリスクは **-D** オプションに続く任意のストリングによって置換されます。

逆に、アスタリスクを使用してある範囲のオプションを除外することができます。例えば、**gxlc** または **gxlc++** にすべての **-std** オプションを無視させたい場合は、以下のような項目を作成することができます。

```
nni*       "-std*"
```

アスタリスクがオプション定義の中で使用された場合、オプション・フラグ *a* および *b* はこれらの項目には適用されません。

文字 **%** は GNU C/C++ オプションと共に使用して、そのオプションに関連パラメーターがあることを示します。これを使用すると、**gxlc** または **gxlc++** は、無視されるオプションと関連したパラメーターを確実に無視します。例えば、**-isystem** オプションはサポートされておらず、パラメーターを使用しています。両方ともアプリケーションによって無視されなければなりません。この場合、項目は以下のようになります。

```
nni*        "-isystem %"
```

GNU C および C++ と XL C/C++ オプションのマッピングの完全リストについては、以下を参照してください。

http://www.ibm.com/support/docview.wss?uid=swg27039015

関連情報

- GNU コンパイラー・コレクションのオンライン文書 (http://gcc.gnu.org/onlinedocs/)

第 3 章 コンパイラー使用状況トラッキングおよびレポート作成

使用状況トラッキングおよびレポート作成機能を使用して、組織内のどのユーザーがコンパイラーを使用しているか、およびコンパイラーを同時に使用しているユーザーの数を記録し、分析することができます。この情報は、組織のコンパイラーの使用がコンパイラー・ライセンス資格の数を超えているかどうかを判別するのに役立ちます。

この機能を使用するには、以下の手順に従ってください。

1. この機能の処理を理解してください。詳しくは、『『使用状況トラッキングおよびレポート作成について』』を参照してください。
2. ユーザー組織内でのコンパイラーの使用方法を調査して、それに従ってコンパイラーの使用量をトラッキングする方法を判断してください。詳しくは、『 58 ページの『この機能の使用準備』』を参照してください。
3. 使用状況トラッキングを構成して、使用可能にしてください。詳しくは、『 65 ページの『使用状況トラッキングの構成』』を参照してください。
4. 使用状況レポート作成ツールを使用して使用状況レポートを作成するか、または使用量ファイルを整理してください。詳しくは、『 74 ページの『使用状況レポートの生成』』または『 77 ページの『使用量ファイルの整理』』を参照してください。

使用状況トラッキングおよびレポート作成について

使用状況トラッキングおよびレポート作成機能は、組織のコンパイラーの使用が、コンパイラー・ライセンス資格の数を超えているかどうかを検出するメカニズムを備えています。このセクションでは、この機能を紹介し、その処理方法を説明し、さらに標準的な使用シナリオを具体的に説明します。

概説

使用状況トラッキングを使用可能に設定している場合、すべてのコンパイラー呼び出しはファイルに記録されます。このファイルは使用量ファイルと呼ばれ、`.cuf` サフィックスが付きます。次に使用状況レポート作成ツールを使用して、これらの使用量ファイルの 1 つ以上からレポートを生成できます。オプションでその使用量ファイルを整理することができます。

使用状況トラッキングおよびレポート作成機能は、組織においてコンパイラーがどのように使用されるかに基づいて、さまざまな方法で使用することができます。50 ページの『4 つの使用シナリオ』 には、この機能の一般的な使用シナリオが示されています。

以下のセクションでは、使用状況トラッキング機能の構成と使用状況レポート作成ツールの使用方法を説明します。

使用状況トラッキング

使用状況トラッキング構成ファイル urtxlc_cpp1301linux.cfg は、デフォルトのコンパイラー・インストールに含まれます。このファイルを使用して、使用状況トラッキングを使用可能に設定し、トラッキングのさまざまな側面を制御することができます。

シンボリック・リンク urt_client.cfg も、デフォルトのコンパイラー・インストールに含まれます。これは、使用状況トラッキング構成ファイルのロケーションを指定します。使用状況トラッキング構成ファイルを別のロケーションに置く場合、それに応じてシンボリック・リンクを変更できます。

詳しくは、『 65 ページの『使用状況トラッキングの構成』』を参照してください。

注: 使用状況トラッキングは、デフォルトで使用不可に設定されています。

使用状況レポート作成ツール

使用状況レポート作成ツールによって、使用量ファイルの情報を基に、コンパイラー使用状況レポートが生成されます。このツールを使用して、オプションで使用量ファイルを整理することもできます。詳しくは、74 ページの『使用状況レポートの生成』および 77 ページの『使用量ファイルの整理』を参照してください。

4 つの使用シナリオ

このセクションでは、コンパイラー使用状況の管理、コンパイラー使用状況情報の記録、およびこの情報からのレポートの生成を行うために考えられる 4 つのシナリオについて説明しています。

以下のシナリオでは、ユーザー組織でコンパイラーを使用する典型的方法をいくつか説明し、またこの機能を使用して各ケースのコンパイラー使用状況をトラッキングする方法を具体的に説明します。

注: 実際の使用法は、これらのシナリオに制限されません。

『シナリオ: 1 台のマシン、1 つの共有 .cuf ファイル』

52 ページの『シナリオ: 1 台のマシン、複数の .cuf ファイル』

54 ページの『シナリオ: 複数のマシン、1 つの共有 .cuf ファイル』

56 ページの『シナリオ: 複数のマシン、複数の .cuf ファイル』

シナリオ: 1 台のマシン、1 つの共有 .cuf ファイル

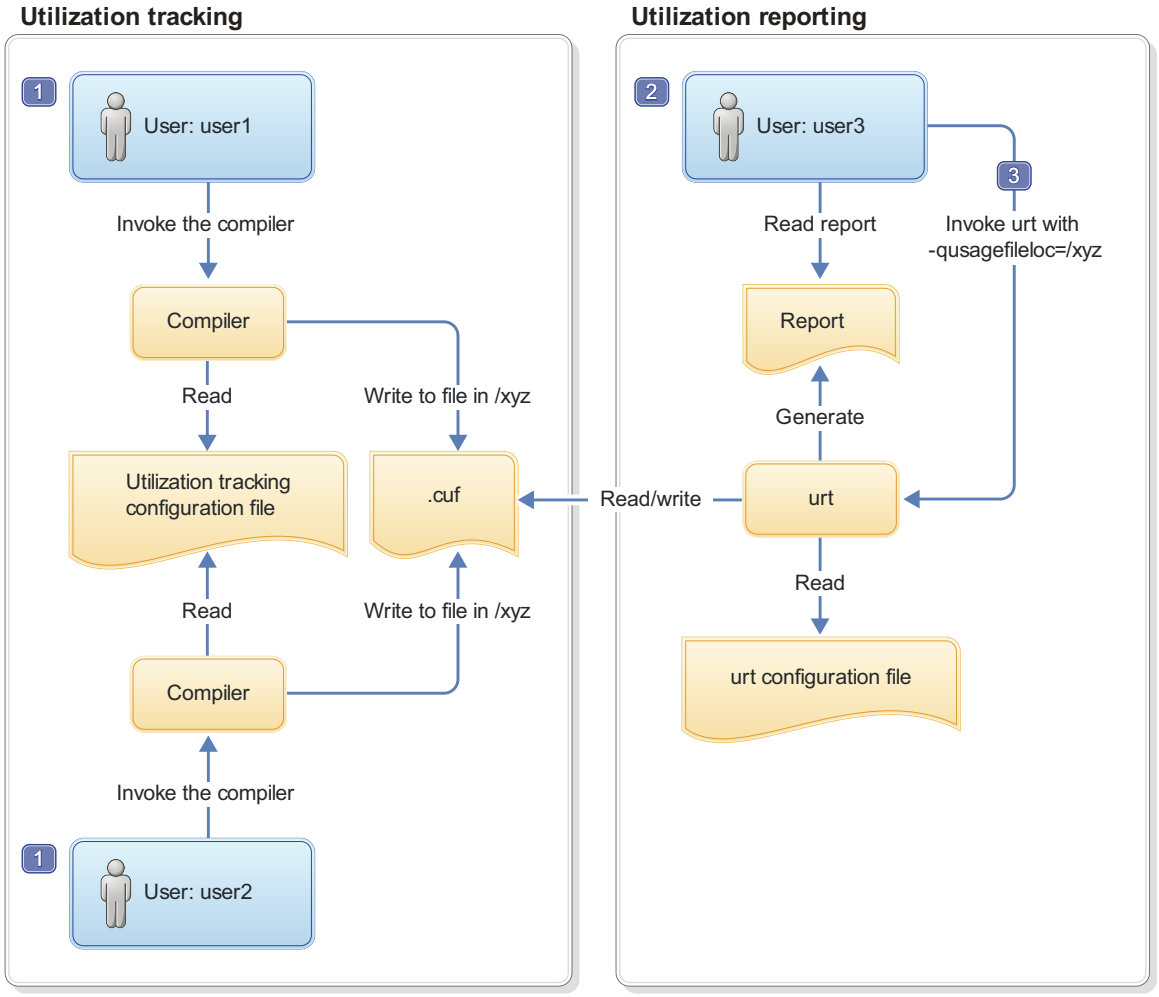
このシナリオでは、すべてのコンパイルが 1 台のマシンで実施され、すべてのユーザーが 1 つの .cuf ファイルを共有する環境について説明します。

このシナリオのアプローチの利点は、使用状況レポート作成ツールが 1 つの .cuf ファイルにアクセスするだけですむため、レポートの生成と使用量ファイルの整理が簡潔になることです。欠点は、このファイルにすべてのコンパイラー・ユーザーからのアクセスが集中するという点です。このファイルは大きくなる可能性があるため、パフォーマンスに影響することがあります。また、共有 .cuf ファイルを作

成し、すべてのコンパイラー・ユーザーに書き込み権限を与えるための設定作業も必要になります。 61 ページの『使用量ファイルの数』セクションで、すべてのコンパイラー・ユーザーに対して単一の使用量ファイルを使用する場合の情報を詳しく説明しています。

このシナリオでは、コンパイラー・ユーザーが同じマシン上でコンパイラーを実行し、使用状況情報は、共有 `.cuf` ファイルに記録されます。コンパイラーの使用状況トラッキング構成ファイルは、`.cuf` ファイルのロケーションを指すように変更されます。コンパイラーは、呼び出されると、使用状況情報をそのファイルに書き込みます。この後、使用状況レポート作成ツールを使用して、ファイルから使用状況情報を取り出し、使用状況レポートを生成できます。

次の図は、このシナリオを示しています。



1. user1 と user2 のどちらにも、`/xyz` 内の `.cuf` ファイルに対する書き込み権限が必要です。
2. user3 には、`/xyz` 内の `.cuf` ファイルに対して、使用状況レポートを生成するための読み取り権限と `.cuf` ファイルを整理するための書き込み権限が必要です。
3. クーロン・ジョブは、定期的に **urt** を自動実行するように作成できます。

図 5. コンパイラー・ユーザーが単一のマシンと共有 .cuf ファイルを使用する場合

この図は、以下の点を示しています。

1. user1 と user2 は、同じ使用状況トラッキング構成ファイルを使用します。この構成ファイルにより、トラッキング機能が一元的に管理されます。共有 .cuf ファイルを保持するため、共通ロケーション /xyz が作成されます。
2. user1 と user2 がコンパイラーを呼び出すと、共通ディレクトリー /xyz 下の .cuf ファイルに使用状況情報が記録されます。
3. user3 が -qusagefileloc=/xyz を指定して **urt** を呼び出し、使用状況レポートを生成します。

注: 使用状況レポート作成ツールは使用量ファイルを整理することができるため、このツールを定期的に実行すると、これらのファイルが大きくなりすぎるのを防止できます。

シナリオ: 1 台のマシン、複数の .cuf ファイル

このシナリオでは、すべてのコンパイルが 1 台のマシンで実施され、すべてのユーザーが独自の .cuf ファイルを持つ環境について説明します。

このシナリオのアプローチには、以下の利点があります。

- コンパイラー・ユーザーは、単一の .cuf ファイルへのアクセスで競合する必要はありません。結果として、パフォーマンスが向上することがあります。
- すべてのコンパイラー・ユーザーに対して、単一の共通ロケーションへの書き込みアクセスをセットアップする必要はありません。ユーザーは、自身のホーム・ディレクトリーに対する書き込み権限を既に持っています。

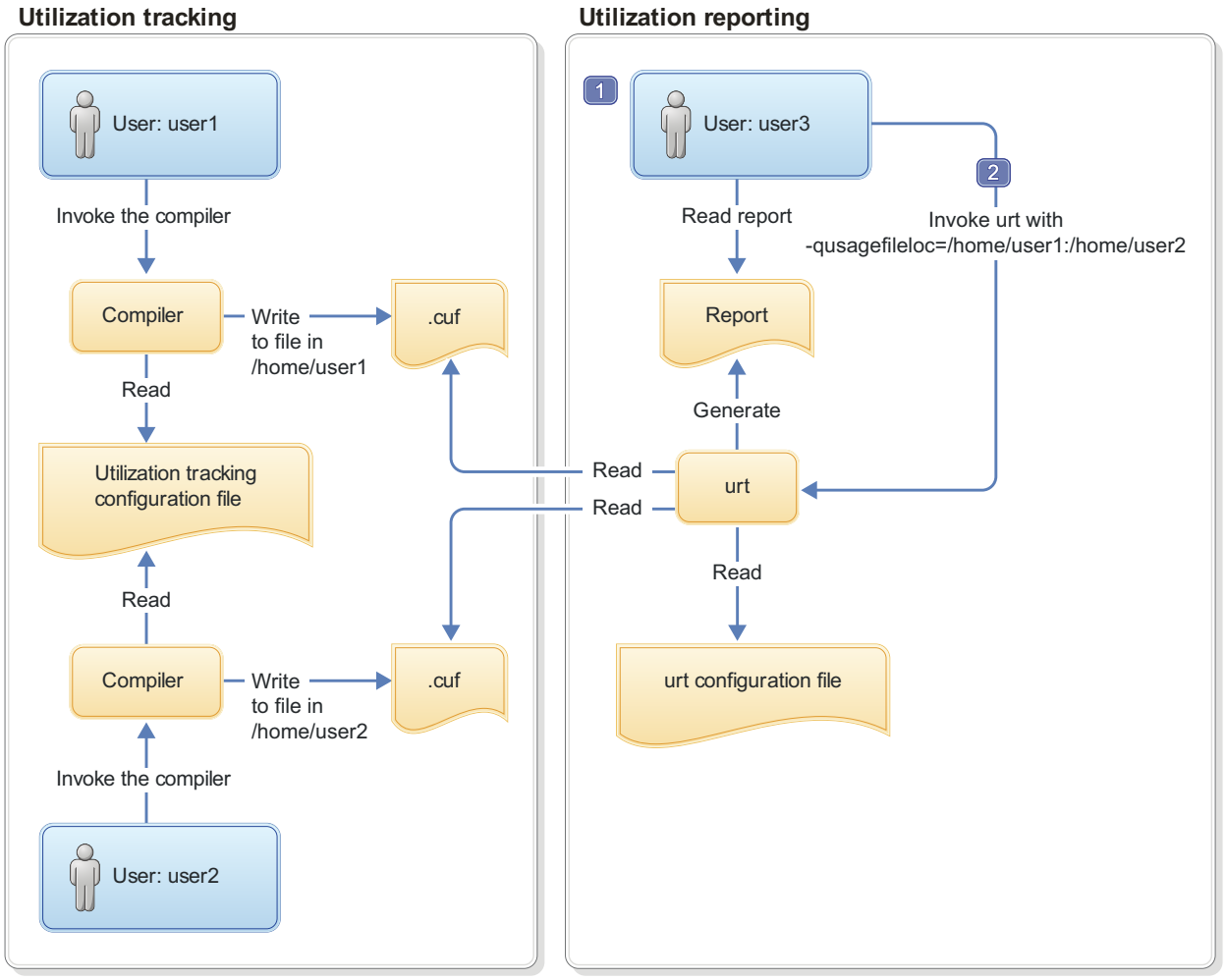
ただし、個々のユーザーのホーム・ディレクトリーに自動的に作成される複数の .cuf ファイルを使用すると、以下の問題が起きることがあります。

- コンパイラー・ユーザーは、そのファイルが作成されたこと、またはそのファイルを見たときにそれが何であるかが分からない可能性があります。この場合、ユーザーがそのファイルを削除することがあるかもしれません。
- 一部のユーザーのホーム・ディレクトリーが、リモート・システムからマウントされたファイル・システムに含まれている可能性があります。これにより使用状況トラッキングがリモート・ファイルを使用することになり、パフォーマンスに影響する可能性があります。
- コンパイラー・ユーザーは、.cuf ファイルがホーム・ディレクトリーでスペースをとることを望まない場合があります。

個々のユーザーのホーム・ディレクトリーを使用する代わりに、個々のユーザーの .cuf ファイルを共通の場所に作成することができます。60 ページの『使用量ファイルのロケーション』セクションで、これらのファイルを共通の場所に作成する方法について詳しく説明しています。

このシナリオでは、2 人のコンパイラー・ユーザーが同じマシン上でコンパイラーを実行し、独自の .cuf ファイルを持ちます。コンパイラーは、呼び出されると、個々のユーザーに .cuf ファイルを自動的に作成し、使用状況情報をそのファイルに書き込みます。この後、使用状況レポート作成ツールを使用して、.cuf ファイルから使用状況情報を取り出し、使用状況レポートを生成できます。

次の図は、このシナリオを示しています。



1. user3 には、/home/user1 および /home/user2 内の .cuf ファイルに対して、使用状況レポートを生成するための読み取り権限と、使用量ファイルを整理するための書き込み権限が必要です。
2. クーロン・ジョブは、定期的に **urt** を自動実行するように作成できます。

図 6. コンパイラー・ユーザーが 1 台のマシンを別々の .cuf ファイルで使用する場合

このダイアグラムは、以下の時点を反映しています。

1. user1 と user2 は、同じ使用状況トラッキング構成ファイルを使用します。この構成ファイルにより、トラッキング機能が一元的に管理されます。
2. user1 と user2 がコンパイラーを呼び出すと、それぞれのホーム・ディレクトリー /home/user1 と /home/user2 下の 2 つの .cuf ファイルに使用状況情報が記録されます。
3. user3 が -qusagefileloc=/home/user1:/home/user2 を指定して **urt** を呼び出し、使用状況レポートを生成します。

   注: 使用量ファイルを含むホーム・ディレクトリーを見つける必要がある場合は、 **urt** を以下のように呼び出してください。

   ```
   urt -qusagefileloc=/home -qmaxsubdirs=1
   ```

   この場合、**urt** は、/home ディレクトリーですべての .cuf ファイルを検索します。

シナリオ: 複数のマシン、1 つの共有 .cuf ファイル

このシナリオでは、コンパイルが複数のマシンで実施され、すべてのユーザーが単一の .cuf ファイルを共有する環境について説明します。

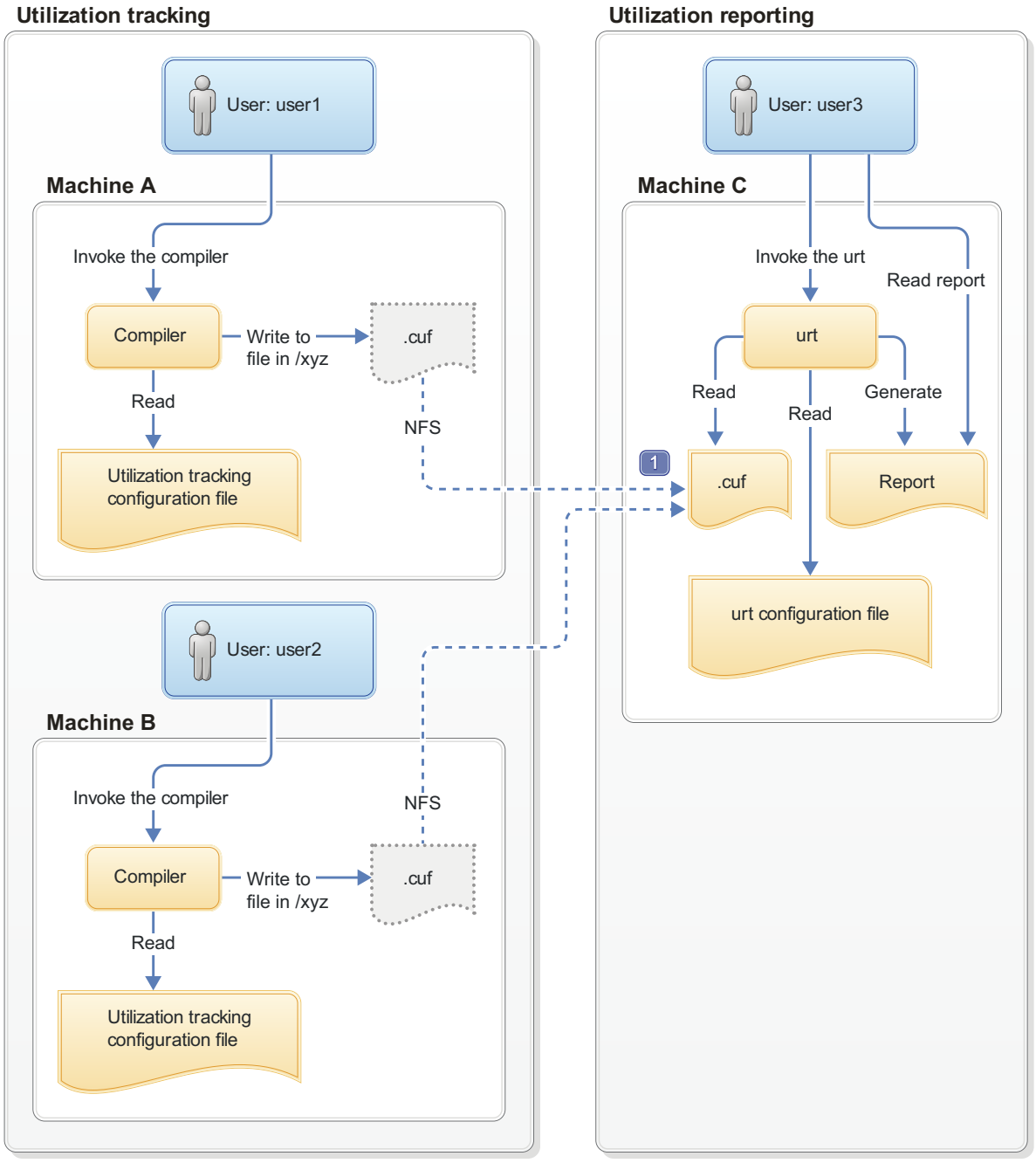
このシナリオのアプローチの利点は、1 つの .cuf ファイルを使用することにより、レポートの生成と使用量ファイルの整理が簡潔になることです。セクション 61 ページの『使用量ファイルの数』で、すべてのコンパイラー・ユーザーに対して単一の使用量ファイルを使用する場合の情報を詳しく説明しています。.cuf ファイルは、使用状況レポート作成ツールがインストールされているマシンに既に存在します。 .cuf ファイルを整理するために、ファイルをこのマシンにコピーし、ツールを複数のマシンにインストールする必要はありません。

このアプローチには以下の欠点があります。

- コンパイラー・ユーザーは 1 つの使用量ファイルへのアクセスを競合しなければなりません。このファイルは大きくなる可能性があるため、パフォーマンスに影響することがあります。
- 一部のセットアップ作業では、共有 .cuf ファイルを作成し、すべてのコンパイラー・ユーザーにネットワーク・ファイル・システムへの書き込み権限を与える必要があります。
- コンパイラーと .cuf ファイルが別々のマシン上に存在するため、プロセス全体の効率は、使用しているネットワーク・ファイル・システムの速度と信頼性に依存します。例えば、複数のユーザーによる同時アクセスに必要なファイル・ロッキングのサポートは、ファイル・システムによって優劣があります。

このシナリオでは、2 人のコンパイラー・ユーザーが別々のマシン上でコンパイラーを実行し、NFS、DFS、または AFS™ などのネットワーク・ファイル・システム上で、1 つの共有 .cuf ファイルを使用します。コンパイラーは、呼び出されると、使用状況情報をそのファイルに書き込みます。この後、使用状況レポート作成ツールを使用して、ファイルから使用状況情報を取り出し、使用状況レポートを生成できます。

次の図は、このシナリオを示しています。



1. Machine A および Machine B 上で、Machine C へのマウント・ポイント `/xyz` が作成されます。すべてのコンパイラー使用状況が、`.cuf` ファイルに記録され、使用状況レポートがそこから生成されます。

図 7. コンパイラー・ユーザーが複数のマシンと、共有 .cuf ファイルを使用する場合

このダイアグラムは、以下の時点を反映しています。

1. 使用状況トラッキングが、Machine A と Machine B でそれぞれ構成されます。

注:

- 個々のマシンには独自の構成ファイルがありますが、ファイルの内容は同じでなければなりません。

- 使用状況トラッキング機能を中央で管理すると、構成に必要な手間が減り、起こりうるエラーを排除することができます。 59 ページの『中央構成』セクションで、異なるマシンを使用するコンパイラー・ユーザーによって共有される共通構成ファイルの使用方法について詳しく説明しています。

2. `.cuf` ファイルを中央管理するために、ネットワーク・ファイル・システムがセットアップされます。user1 と user2 が Machine A と Machine B からコンパイラーを呼び出すと、両方のコンパイラーの使用状況情報が Machine C 上の `.cuf` ファイルに書き込まれます。
3. user3 が **urt** を呼び出して、Machine C 上の `.cuf` ファイルから使用状況レポートを生成します。

注: 使用状況レポート作成ツールを使用して使用量ファイルを定期的に整理し、それらのファイルが大きくなりすぎるのを防止できます。

シナリオ: 複数のマシン、複数の .cuf ファイル

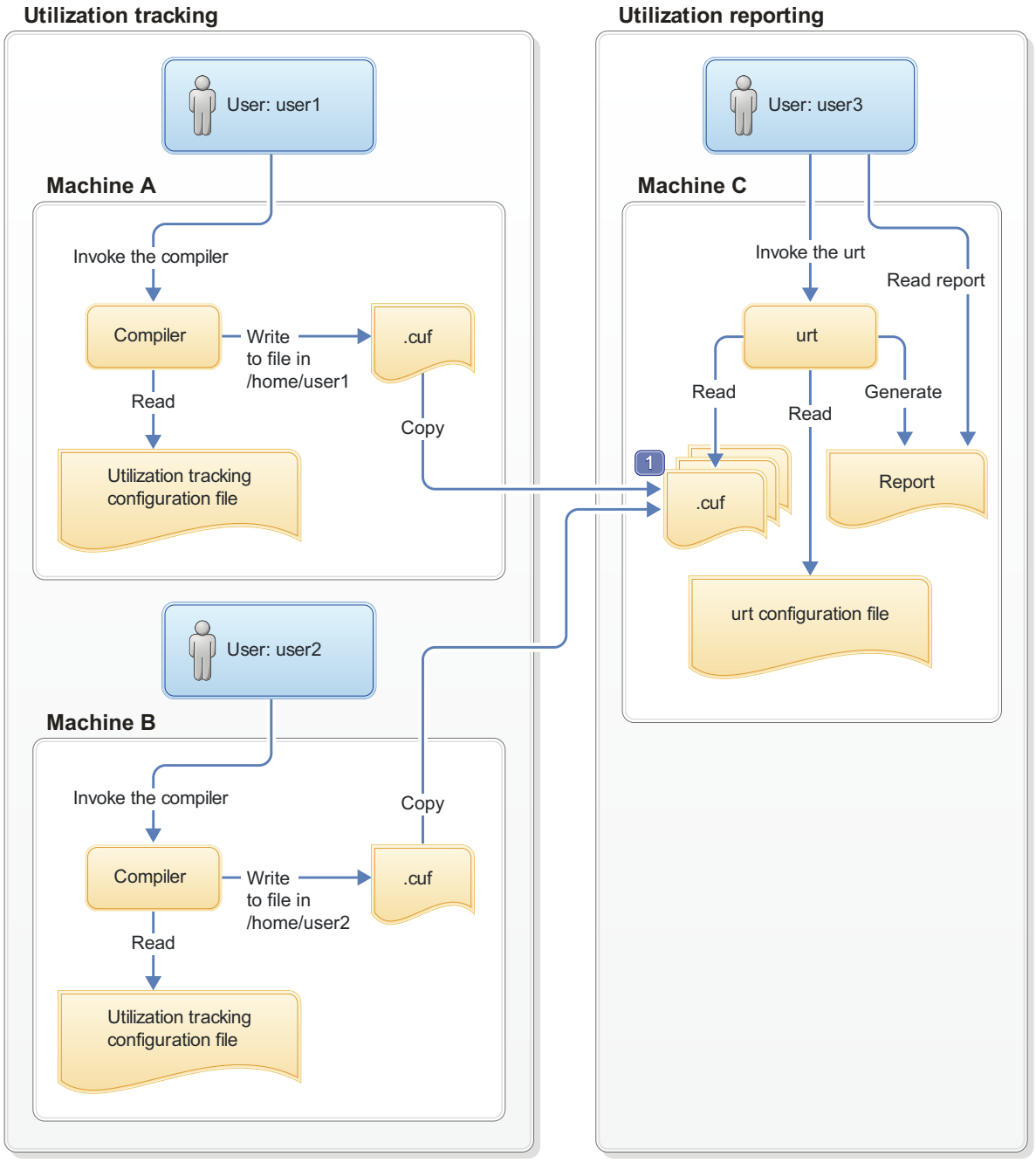
このシナリオでは、コンパイルが複数のマシンで実施され、すべてのユーザーが独自の使用量ファイルを持つ環境について説明します。

このシナリオでは、2 人のコンパイラー・ユーザーが別々のマシン上でコンパイラーを実行し、それぞれ独自の `.cuf` ファイルを持ちます。コンパイラーは、呼び出されると、使用状況情報をそのファイルに書き込みます。この後、使用状況レポート作成ツールを使用して、ファイルから使用状況情報を取り出し、使用状況レポートを生成できます。このツールは、コンパイラーがインストールされているマシン上でも、また別のマシン上でも実行できます。

注: 使用状況レポート作成ツールは、すべての `.cuf` ファイルに読み取りアクセスできることが必要です。
この例では、以下の方法のいずれかを使用して、ファイルをアクセス可能にすることができます。

- NFS、DFS、または AFS などのネットワーク・ファイル・システムを使用。
- 元のロケーションから使用状況レポート作成ツールを実行する予定のマシンにファイルをコピーする。ファイルのコピーには、**ftp**、**rcp**、**rsync**、またはその他の任意のリモート・コピー・コマンドを使用できます。

次の図は、このシナリオを示しています。



1. user3 が、.cuf ファイルを Machine C にコピーします。クーロン・ジョブは、定期的にファイルを自動コピーするように作成できます。

図 8. コンパイラー・ユーザーが複数のマシンを複数の .cuf ファイルで使用する場合

このダイアグラムは、以下の時点を反映しています。

1. 使用状況トラッキングが、Machine A と Machine B でそれぞれ構成されます。

注:

- 個々のマシンには独自の構成ファイルがありますが、ファイルの内容は同じでなければなりません。

- 使用状況トラッキング機能を中央で管理すると、構成に必要な手間が減り、起こりうるエラーを排除することができます。 59 ページの『中央構成』セクションで、異なるマシンを使用するコンパイラー・ユーザーによって共有される共通構成ファイルの使用方法について詳しく説明しています。

2. user1 と user2 がコンパイラーを呼び出すと、それぞれのホーム・ディレクトリー /home/user1 と /home/user2 下の 2 つの .cuf ファイルに使用状況情報が記録されます。

   注: これらの .cuf ファイルは、/var/tmp などの別の共通の場所にも作成できます。 60 ページの『使用量ファイルのロケーション』セクションで、これらのファイルを共通の場所に作成する方法について詳しく説明しています。
3. user3 が、2 つの .cuf ファイルを Machine A と Machine B から Machine C にコピーします。
4. user3 が **urt** を呼び出して、Machine C 上の .cuf ファイルから使用状況レポートを生成します。

関連情報

- 『この機能の使用準備』
- 65 ページの『使用状況トラッキングの構成』
- 74 ページの『使用状況レポートの生成』
- 77 ページの『使用量ファイルの整理』

この機能の使用準備

組織内で使用状況トラッキングを使用可能にする前に、組織におけるコンパイラーの使用方法に関連するいくつかの要因を考慮する必要があります。

以下のセクションでは、それらの考慮事項について詳細に説明します。

時刻の同期

複数のマシン上でコンパイラーの使用状況をトラッキングする場合には、それらのマシン間の時刻の同期を考慮しなければなりません。

使用状況レポート作成ツールが生成する使用状況レポートには、コンパイラー呼び出しの開始時刻と終了時刻がリストされます。また、レポートでは、どの呼び出しが同時に行われたかも判断されます。これらのマシン間で時刻が同期していない場合には、この情報の信頼性と有効性に影響が生じます。

異なるマシン間で時刻を同期できない場合には、**-qadjusttime** オプションを使用して、記録された時刻を調整するよう使用状況レポート作成ツールに指示することができます。

ライセンス・タイプおよびユーザー情報

この機能の使用を開始する前に、組織のライセンス資格の数とタイプが必要です。

必要なライセンスおよびユーザー情報は以下のとおりです。

- このコンパイラーに対して所有する同時ユーザー・ライセンスの数。この情報は、使用状況トラッキング構成ファイルの **-qmaxconcurrentusers** 項目に必要です。
- このコンパイラーの許可ユーザー・ライセンスを所有するユーザー。この情報は、使用状況トラッキング構成ファイルの **-qexemptconcurrentusers** 項目に使用されます。
- 複数のアカウントでコンパイラーを使用するユーザー。この情報は、使用状況レポート作成ツールの **-qsameuser** オプションに使用されます。

注: 許可ユーザー・ライセンスを所有するユーザーと、複数のアカウントでコンパイラーを使用するユーザーを指定することは必須ではありませんが、これらのユーザーを指定すると使用状況レポート作成ツールで生成される使用状況レポートの精度を上げることができます。詳しくは、『 60 ページの『同時ユーザーに関する考慮事項』』を参照してください。

中央構成

すべてのコンパイラー・ユーザーについて同じ使用状況トラッキングを構成することは、それにより使用状況トラッキングの正確性が確保され、かつ構成と保守にかかる労力を最小限に抑えることができるため、非常に重要です。このことは、すべてのユーザーが同じ使用状況トラッキング構成ファイルを確実に使用するようにして実現できます。

コンパイラーのインストール済み環境が 1 つだけの場合は、使用状況トラッキング構成ファイルを直接編集できます。すべてのコンパイラー・ユーザーは、その構成ファイルを自動的に使用することになります。

コンパイラーのインストール済み環境が複数ある場合、単一の使用状況トラッキング構成ファイルを維持し、すべてのインストール済み環境からこのファイルを参照する必要があります。使用状況トラッキングを使用可能または使用不可に設定するなど、使用状況トラッキング構成ファイルに対して行った変更はすべて、ユーザーがコンパイラーを呼び出す際に、すべてのコンパイラーのインストール済み環境に自動的に適用されます。それぞれのインストール済み環境には、`urt_client.cfg/opt/ibm/xlC/13.1.0/urt` の下にあります。構成ファイルのこの共有インスタンスを指すようにシンボリック・リンクを変更してください。

コンパイラーが複数のマシンにインストールされている場合に使用状況トラッキング構成ファイルを個々のマシンのコンパイラーで使用するには、 NFS、DFS、または AFS などのネットワーク・ファイル・システム上に配置する必要があります。

注: すべてのコンパイラー・ユーザーに単一の使用状況トラッキング構成ファイルを使用することができない場合、個々のコンパイラーのインストール済み環境について、すべての使用状況トラッキング構成ファイルに一貫性があることを確認する必要があります。同じコンパイラーで別々の構成を使用する方法はサポートされていません。

同時ユーザーに関する考慮事項

コンパイラーの呼び出しは、呼び出しの開始時刻と終了時刻がオーバーラップすると、同時であるとみなされます。このセクションでは、使用状況レポート作成ツールが同時ユーザーをカウントする方法と、使用状況レポートの正確性を向上させる方法について説明します。

使用状況レポート作成ツールが同時ユーザーの数をカウントする際には、このツールは使用量ファイルに取り込まれたユーザー・アカウント情報を参照します。アカウント情報は、ユーザー名、ユーザー ID、およびホスト名から構成されます。デフォルトでは、このアカウント情報の固有の組み合わせごとに、異なるユーザーとしてみなされ、カウントされます。ただし、以下のユーザーによるコンパイラーの呼び出しは同時ユーザーのカウントに含めてはいけません。

- 許可ユーザー・ライセンスを所有するユーザーは、コンパイラーを使用する際に同時ユーザー・ライセンスをまったく使わないため、免除ユーザーとみなされます。
- 複数のアカウントを所有するユーザー。これらのアカウントは同一ユーザーに属すため、それらのアカウントを使用してログオン中のコンパイラー呼び出しは、単一ユーザーによる使用としてカウントされます。

使用状況レポート作成ツールは、免除ユーザーと複数のアカウントを持つユーザーに関する情報が提供されると、上記の状況に正しく対処することができます。以下にその情報をどのように提供できるかを示します。

- **-qexemptconcurrentusers** 項目を使用状況トラッキング構成ファイルに指定します。この項目により、許可ユーザー・ライセンスを所有するユーザーが指定されます。
- **-qsameuser urt** コマンド行オプションを指定します。このオプションにより、複数のアカウントを持つユーザーが指定されます。

注:

- 同時ユーザーの数が **-qexemptconcurrentusers** または **-qsameuser** で調整されると、使用状況レポート作成ツールは、同時使用量情報が調整されたことを示すメッセージを生成します。
- 同時ユーザーの数は、すべての同時呼び出しが免除ユーザーにより呼び出された場合、ゼロになる可能性があります。このツールは、この情報を含むメッセージも生成します。

使用量ファイルの考慮事項

使用量 (`.cuf`) ファイルはコンパイラーの使用量情報を保管するために使用されます。このセクションには、これらのファイルの生成方法と使用方法を決める際に役立つ情報が記載されます。

使用量ファイルのロケーション

使用量ファイルは各ユーザーのホーム・ディレクトリーに作成するか、すべてのユーザーの中央の場所に作成することができます。

使用状況トラッキングが有効な場合にコンパイラー・ユーザーがプログラムをコンパイルすると、 `.cuf` ファイルが存在しない場合には、このファイルがユーザーの

ホーム・ディレクトリーに自動的に作成されます。これは、ユーザーは自分のホーム・ディレクトリーへの書き込み権限を既に所有しており、追加のセットアップが不要になるため、使用状況トラッキング機能をテストする場合に便利です。ただし、この方法には以下の問題が発生することがあります。

- コンパイラー・ユーザーは、そのファイルが作成されたこと、またはそのファイルを見たときにそれが何であるかが分からない可能性があります。この場合、ユーザーがそのファイルを削除することがあるかもしれません。
- 一部のユーザーのホーム・ディレクトリーが、リモート・システムからマウントされたファイル・システムに含まれている可能性があります。これにより使用状況トラッキングがリモート・ファイルを使用することになり、パフォーマンスに影響する可能性があります。
- コンパイラー・ユーザーは、使用量ファイルがホーム・ディレクトリーでスペースをとることを望まない場合があります。

有効な代替手段は、使用量ファイルを作成できる中央の場所をセットアップし、コンパイラー・ユーザーと使用状況レポート作成ツールのユーザーの両方に対して、その場所への適正なアクセス権限を付与する方法です。このセットアップを行うには、other/world 権限またはグループ権限を使用します。

例えば、中央の場所のディレクトリー名が /var/tmp/track_compiler_use である場合、使用状況トラッキング構成ファイルの **-qusagefileloc** 項目を以下のように変更できます。

```
-qusagefileloc=/var/tmp/track_compiler_use/$LOGNAME.cuf
```

これにより user1.cuf や user2.cuf などの .cuf ファイルが、各ユーザーごとに指定されたロケーションに作成されます。この中央の場所にある .cuf ファイルから使用状況レポートを生成するための、使用状況レポート作成ツールの実行がより簡単になります。必要な操作は、ロケーションのパス /var/tmp/track_compiler_use を使用状況レポート作成ツールに渡すことのみです。これでツールはそのロケーションのすべての .cuf ファイルを読み取ることができます。

コンパイラー・ユーザーが複数のマシンでコンパイラーを実行する場合、*$HOSTNAME* を **-qusagefileloc** 項目に追加して、ファイル名が衝突しないようにする必要があります。例えば、**-qusagefileloc** 項目は、以下のように指定できます。

```
-qusagefileloc=/var/tmp/track_compiler_use/$HOSTNAME_$LOGNAME.cuf
```

これにより、各ユーザーごとに .cuf ファイルが作成され、その .cuf ファイルの名前に、コンパイラーが使用されるホストの名前も含まれて、host1_user1.cuf などとなります。

使用量ファイルの数

各コンパイラー・ユーザーに対して 1 つの (同一の) 使用量ファイルまたは個別の使用量ファイルを使用できます。

各コンパイラー・ユーザーに対して個別の使用量ファイルを使用する

個別の使用量ファイルを使用する利点は以下のとおりです。

- コンパイラー・ユーザーは、共有ファイルへのアクセスについて競合せずに独自の使用量ファイルにアクセスし、また個別の使用量ファイルは通常、より小さくなるため、パフォーマンスが向上する可能性があります。
- あるユーザーの使用量ファイルは、そのユーザーがコンパイラーを使用してプログラムをコンパイルする際に自動作成することができます。ユーザーごとに事前に使用量ファイルを明示的に作成する必要はありません。 詳しくは、『 60 ページの『使用量ファイルのロケーション』』を参照してください。
- 通常は、使用状況レポートの生成時に、すべてのコンパイラー・ユーザーが含まれます。ただし、一部のユーザーを除外したい場合には、使用状況レポート作成ツールを呼び出す際に、除外するユーザーの使用量ファイルを除くだけですみます。例えば、許可ユーザー・ライセンスを所有するユーザーを除外することができます。

この方法の欠点は、ユーザーごとに個別の使用量ファイルを維持しなければならない場合があることです。

すべてのコンパイラー・ユーザーに対して単一の使用量ファイルを使用する

すべてのユーザーに対して 1 つの共有使用量ファイルを使用する場合、複数ファイルではなく単一ファイルを維持するだけですむという利点があります。ただし、単一の使用量ファイルを使用すると、上記のサブセクションで説明した複数の使用量ファイルを使用した場合の柔軟性とパフォーマンスの向上の可能性が失われます。

コンパイラーには、空の使用量ファイル `urtstub.cuf` が `/opt/ibm/xlC/13.1.0/urt` ディレクトリーに用意されています。 すべてのコンパイラー・ユーザーが書き込み権限を持つディレクトリーにこの空の使用量ファイルをコピーすることにより、すべてのコンパイラー・ユーザーに対する 1 つの使用量ファイルを作成することができます。この場合、使用状況トラッキング構成ファイルの **-qusagefileloc** 項目をこの使用量ファイルの場所を指すように変更する必要があります。

複数マシン上の使用量ファイル

コンパイラーを複数のマシンで使用する場合、使用量ファイルを使用状況レポート作成ツールで使用可能にする方法を決定する必要があります。

使用量ファイルを使用状況レポート作成ツールで利用できるようにし、使用状況レポートの生成および使用量ファイルの整理を行うには、さまざまな方法があります。使用量ファイルを複数のマシンで管理するには、以下のいずれかの方法を選択します。

- コンパイラーが使用されるマシンから、使用状況レポート作成ツールがインストールされているマシンに使用量ファイルをコピーします。任意のリモート・コピー・コマンド (**ftp**、**rcp**、**scp**、**rsync** など) を使用できます。この場合、使用量ファイルは、コンパイラー側 (使用状況トラッキング) と使用状況レポート作成ツール側 (使用状況レポートの生成) の両方からローカルでアクセスされます。ファイルをローカルでアクセスすると、最高のパフォーマンスが得られます。
- 分散ファイル・システムを使用して、コンパイラーが使用されるマシンからファイル・システムをエクスポートし、使用状況レポート作成ツールがインストールされているマシンで、そのファイル・システムをマウントします。使用状況レポート作成ツールを実行すると、マウントしたファイル・システムを介して使用量ファイルにリモートでアクセスできます。

- また、使用状況レポート作成ツールがインストールされているマシンからファイル・システムをエクスポートし、コンパイラーが使用される各マシンでそのファイル・システムをマウントして、これをコンパイラーが使用状況を記録する使用量ファイルのロケーションとして使用することもできます。この方法では、コンパイラーはリモートの使用量ファイルに使用状況を記録し、使用状況レポート作成ツールは使用量ファイルをローカルで読み取ります。

  注: これによりコンパイラーのパフォーマンスが低下する場合は、最初の 2 つの方法のいずれかを検討してください。

使用量ファイルのサイズ

使用量ファイルのサイズは、特にすべてのコンパイラー・ユーザーで共有ファイルを使用する場合、急速に大きくなる可能性があることを考慮する必要があります。使用量ファイルが大きくなりすぎると、使用状況トラッキングのパフォーマンスに影響する場合があります。

使用量ファイルが急速に大きくならないようにするには、使用状況レポートを生成する際に、オプションで使用量ファイルを整理することができます。このファイルは、クーロンを使用して定期的に整理することもできます。

ファイルの整理について詳しくは、『 77 ページの『使用量ファイルの整理』』を参照してください。

定期的な使用状況の検査

使用状況レポート作成ツールを定期的に実行して、コンパイラーの使用量が、購入した同時ユーザー・ライセンス数に準拠しているかどうかを検証することができます。クーロン・ジョブを作成して、これを自動的に実行することができます。

使用状況レポート作成ツールを実行するマシンに使用量ファイルをコピーする必要がある場合は、コピー作業もクーロン・ジョブにより自動化できます。

このツールを定期的に実行する別の理由は、使用量ファイルを整理してこのファイルのサイズを制御するためです。

注: 使用量ファイルに対する読み取りアクセスと書き込みアクセスの競合を削減するために、使用状況レポート作成ツールの実行および使用量ファイルのコピーは、コンパイラーが使用されていないときに行うようにしてください。

使用状況トラッキングのテスト

組織内の全ユーザーのコンパイラー使用量のトラッキングを開始する前に、ユーザー数を制限するか別個のコンパイラー・インストール済み環境を使用して、この機能をテストすることができます。このテスト中に、組織にとって最適なセットアップを判断するために、異なる構成を試すことができます。

ユーザー数を制限したテスト

ユーザー数を制限してコンパイラー使用状況トラッキングを使用可能にする場合には、別個の使用状況トラッキング構成ファイルを使用して、これらのユーザーのみにこのファイルを使用するよう依頼することができます。同じインストール済み環

境の他のユーザーは、使用状況トラッキングが使用不可に設定されたデフォルトの使用状況トラッキング構成ファイルを使用します。したがって、コンパイラーの使用は記録されません。

デフォルトのコンパイラー構成ファイル xlc.cfg.*$OSRelease.gcc$gccVersion*。例えば、xlc.cfg.sles11.gcc432 または xlc.cfgrhel6.4.gcc447 には、*xlurt_cfg_path* と *xlurt_cfg_name* の 2 つの項目が含まれており、それらの項目で使用状況トラッキング構成ファイルの場所を指定します。指定されたユーザーが別個の使用状況トラッキング構成ファイルを使用するようにするには、以下のタスクを実行する必要があります。

1. 別個のコンパイラー構成ファイルまたはスタンザを作成し、その中で、使用する使用状況トラッキング構成ファイルのロケーションを *xlurt_cfg_path* 項目と *xlurt_cfg_name* 項目で指定します。
2. これらのユーザーに対して、以下のコンパイラー・オプションまたは環境変数を使用して、コンパイラーに別個のコンパイラー構成ファイルまたはスタンザを使用するよう指示します。これによってユーザーは別個の使用状況トラッキング構成ファイルを使用することができます。
   - **-F** オプション
   - XLC_USR_CONFIG 環境変数

例 1

デフォルトの構成ファイルおよび新しいスタンザ *xlc_urt* を使用してプログラム myprogram.c をコンパイルする場合、以下の 2 つのステップに従います。

1. 対応する xlc.cfg.61 または xlc.cfg.71 コンパイラー構成ファイル内にスタンザを作成します。例を以下に示します。

```
xlc_urt: use        =  DEFLT
         xlurt_cfg_path=$location_of_separate_utilization_conf_file
         xlurt_cfg_name=$name_of_separate_utilization_conf_file
         crt        = /lib/crt0.o
         mcrt       = /lib/mcrt0.o
         gcrt       = /lib/gcrt0.o
         libraries  = -L/opt/ibm/xlC/13.1.0/lib,-lxlopt,-lxl,-lc
         proflibs   = -L/lib/profiled,-L/usr/lib/profiled
         options    = -qlanglvl=extc99,-qcpluscmt,-qkeyword=inline,-qalias=ansi
```

2. 以下のコマンドを使用して myprogram.c をコンパイルします。

```
xlc myprogram.c -F:xlc_urt
```

例 2

新規作成されたコンパイラー構成ファイル myconfig.cfg を使用してプログラム myprogram.c をコンパイルする場合、以下の 2 つのステップに従います。

1. *xlurt_cfg_path* および *xlurt_cfg_name* エントリーを、適宜別の使用状況トラッキング構成ファイルの場所と名前に設定します。例を以下に示します。

```
DEFLT_C:
        use           =DEFLT
        xlurt_cfg_path=$location_of_separate_utilization_conf_file
        xlurt_cfg_name=$name_of_separate_utilization_conf_file

DEFLT_CPP:
```

```
use           =DEFLT
xlurt_cfg_path=$location_of_separate_utilization_conf_file
xlurt_cfg_name=$name_of_separate_utilization_conf_file
```

2. 以下のいずれかのコマンドを使用して myprogram.c をコンパイルします。

```
export XLC_USR_CONFIG="$location_of_newly_created_configuration_file/myconfig.cfg"
xlc myprogram.c
```

または

```
xlc myprogram.c -F$location_of_newly_created_configuration_file/myconfig.cfg
```

注: このアプローチは、使用状況トラッキング機能のテスト専用です。すべてのコンパイラー呼び出しが **-F** オプションまたは **XLC_USR_CONFIG** 環境変数を設定した状態で行われるようにできない場合は、このアプローチは、組織内のすべてのコンパイラー使用状況のトラッキングには使用しないでください。

別のコンパイラー・インストールでのテスト

使用状況トラッキングのテスト用に、コンパイラーの別個のインスタンスをインストールすることができます。このケースでは、そのインストール内の使用状況トラッキング構成ファイルを直接変更して、使用状況トラッキングを使用可能にし、構成することができます。テストに参加するコンパイラー・ユーザーは、トラッキングのために何からのタスクを実行する必要はありません。

ユーザー組織に対する最高の使用状況トラッキング構成が見つかった場合には、それを組織内のすべてのコンパイラー・ユーザーから使用可能にすることができます。

関連情報

- 『使用状況トラッキングの構成』
- -F

使用状況トラッキングの構成

使用状況トラッキング構成ファイルを使用すると、使用状況トラッキング機能を使用可能にして、構成することができます。

構成ファイルのデフォルトの場所は /opt/ibm/xlC/13.1.0/urt で、ファイル名は urtxlc_cpp1301linux.cfg です。

コンパイラーはシンボリック・リンクを使用して、使用状況トラッキング構成ファイルの場所を指定します。シンボリック・リンクは /opt/ibm/xlC/13.1.0/urt にもあり、名前は urt_client.cfg です。以下の状態では、シンボリック・リンクの変更が必要な場合があります。

- 使用状況トラッキング構成ファイルを別の場所で使用する場合、その場所を指すようにシンボリック・リンクを変更してください。
- 同じコンパイラーの複数のインストール済み環境があり、単一の使用状況トラッキング構成ファイルを使用する予定がある場合、インストール済み環境ごとにその場所を指すようにシンボリック・リンクを変更してください。詳しくは、『59 ページの『中央構成』』を参照してください。

注: PTF 更新のインストールでは、使用状況トラッキング構成ファイルは上書きされません。

使用状況トラッキング構成ファイルの項目を使用すると、コンパイラー使用量をトラッキングする方法を構成することができます。このファイルの特定の項目の詳細と、それらの変更方法については、『『使用状況トラッキング構成ファイルの項目の編集』』を参照してください。

使用状況トラッキング構成ファイルの項目の編集

使用状況トラッキング構成ファイルの項目を編集することにより、使用状況トラッキングのさまざまな側面を構成することができます。

項目は 2 つのカテゴリーに分かれます。

1. *製品情報* カテゴリーの項目ではコンパイラーが識別されます。これらの項目は変更しないでください。
2. *トラッキング構成* カテゴリーの項目は、この製品に使用状況トラッキングを構成するために使用できます。これらの項目への変更は、次のコンパイラー呼び出し時に使用量ファイルに反映されます。この場合、コンパイラーから新規構成値が使用量ファイルに保存されたことを示すメッセージが出されます。使用量ファイルからレポートを生成する際に、この新規の値が使用されます。

以下の規則は項目を変更する際に適用されます。

- 使用量ファイルを変更した場合には、常に以下の項目がこのファイルに書き込まれ、これらの項目は、使用状況レポート作成ツールが使用量ファイルから次にレポートを生成する際に使用されます。以下の構成項目は、すべてのコンパイラー・ユーザーで同じでなければなりません。
  - **-qmaxconcurrentusers**
  - **-qexemptconcurrentusers**
  - **-qqualhostname**
- **-qqualhostname** が変更されると、既存の使用量ファイルを廃棄し、新規使用量ファイルで再度使用状況のトラッキングを開始する必要があります。それ以外の場合、呼び出しには修飾ホスト名で記録されるものと、非修飾ホスト名で記録されるものがあります。

注:

- これらの項目はコンパイラー・オプションではありません。使用状況トラッキング構成ファイルでのみ使用されます。
- **-qexemptconcurrentusers** 項目が、使用状況トラッキング構成ファイルで複数回指定された場合、指定されたすべてのインスタンスが使用されます。その他の項目が複数回指定された場合、最後の項目の値は前の項目の値をオーバーライドします。
- 複数の使用状況トラッキング構成ファイルを使用する際に、上記の項目にユーザーごとに異なる値を指定した場合、コンパイラーはメッセージを生成します。項目を一貫性のある状態を保持するように変更するか、すべてのコンパイラー・ユーザーが単一の使用状況構成ファイルを使用するようにしてください。

製品情報

-qprodId=*product_identifier_string*
固有の製品 ID ストリングを示します。

-qprodVer=*product_version*
製品のバージョンを示します。

-qprodRel=*product_release*
製品リリースを示します。

-qprodName=*product_name*
製品名を示します。

-qconcurrentusagescope=prod | ver | rel
同時ユーザーがカウントされるレベルを指定します。ユーザーの数には制限があります。サブオプションは以下のとおりです。

- **prod** は製品レベルを示します。
- **ver** はバージョン・レベルを示します。
- **rel** はリリース・レベルを示します。

デフォルトは **-qconcurrentusagescope=prod** です。

トラッキング構成

-qmaxconcurrentusers=*number*

同時ユーザーの最大数を指定します。これは、製品用に購入した同時ユーザー・ライセンスの数です。使用状況レポート作成ツールは、使用量ファイルからレポートを生成する際に、組織内のコンパイラー使用量がこの同時ユーザーの最大数を超えているかどうかを判別します。

注: 購入した同時ユーザー・ライセンスの実際の数を反映させるには、この項目を更新する必要があります。

デフォルト: 0

-qexemptconcurrentusers ="*user_account_info_1* **[|** *user_account_info_2* **| ... |** *user_account_info_n***]"**

許可ユーザー・ライセンスを所有する免除ユーザーを指定します。免除ユーザーは、組織で使用できる同時ユーザー・ライセンスの数に影響を及ぼさずに、必要とするだけのコンパイラーの同時呼び出しを行うことができます。使用状況レポート作成ツールは、使用状況レポートを生成する際に、同時ユーザーのカウントにこのようなユーザーを含めません。

user_account_info は、以下の項目の任意の組み合わせにできます。

- name(*user_name*)
- uid(*user_ID*)
- host(*host_name*)

情報が指定された基準に一致するユーザーは免除ユーザーとみなされます。例えば、*user1@host1* および *user2@host1* が免除ユーザーであることを示すには、以下の形式のいずれかでこの項目を指定できます。

- `-qexemptconcurrentusers="name(user1)host(host1)"`
  `-qexemptconcurrentusers="name(user2)host(host1)"`

- `-qexemptconcurrentusers="name(user1)host(host1) | name(user2)host(host1)"`

user_name、*user_ID*、および *host_name* の場合、ユーザー名、ユーザー ID、またはホスト名のリストも使用できます。この場合、リスト項目を括弧内でスペースで区切ってください。例を以下に示します。

```
-qexemptconcurrentusers="name(user1 user2)host(host1)"
```

これは前の例と同等です。

注: この項目を指定しても、ユーザーはコンパイラー使用状況トラッキングからは免除されません。ユーザーが免除されるのは、同時ユーザーとしてカウントされることのみです。使用状況トラッキング・パフォーマンスを最適化するには、指定された値のフォーマットは、レポートが作成されるまで検証されません。同時ユーザーのカウントについて詳しくは、『 60 ページの『同時ユーザーに関する考慮事項』』を参照してください。

-qqualhostname | -qnoqualhostname

使用量ファイルに取り込まれ、次にコンパイラー使用状況レポートにリストされるホスト名が、ドメイン名で修飾されるかどうかを指定します。

組織内でのすべてのコンパイラー使用が単一ドメイン内のマシン上で行われる場合、 **-qnoqualhostname** を使用してドメイン名の修飾を抑制することにより、使用量ファイルのサイズを減らすことができます。

デフォルトは **-qqualhostname** で、ホスト名がドメイン名で修飾されることを意味します。

-qenabletracking | -qnoenabletracking

使用状況トラッキングを使用可能または使用不可に設定します。

デフォルトは **-qnoenabletracking** で、使用状況トラッキングを使用不可にすることを意味します。

-qusagefileloc=*directory_or_ file_name*

使用量ファイルの場所を指定します。

デフォルトでは、各ユーザーごとにそのホーム・ディレクトリーに `.cuf` ファイルが自動的に作成されます。ユーザーごとにファイルを作成できる中央の場所をセットアップすることができます。詳しくは、『 60 ページの『使用量ファイルのロケーション』』を参照してください。

以下の規則はこの項目を指定する際に適用されます。

- ファイル名を指定する場合は、`.cuf` 拡張子を付ける必要があります。ファイルがシンボリック・リンクである場合、`.cuf` 拡張子の付いたファイルを示す必要があります。指定されたファイルが存在しない場合、まだ存在していない親ディレクトリーと共に `.cuf` ファイルが作成されます。
- ディレクトリーを指定する場合は、そのディレクトリー内に `.cuf` 拡張子が付いたファイルがちょうど 1 つ存在する必要があります。この場合には、新しいファイルは作成されません。
- 指定されたディレクトリーのパスは、相対パスまたは絶対パスにできます。相対パスは、コンパイラー・ユーザーの現行作業ディレクトリーからの相対です。

注: ファイルを使用または作成するための十分な権限を持っていないなどの理由でコンパイラー・ユーザーがファイルにアクセスできない場合、コンパイラーはメッセージを生成し、コンパイルが続行します。

このオプションには、以下の変数を使用できます。

- ユーザーのホーム・ディレクトリーを表す *$HOME*。これにより、個々のユーザーが、ホーム・ディレクトリーまたはホーム・ディレクトリーのサブディレクトリーに `.cuf` ファイルを所有することができます。
- ユーザーのログイン・ユーザー名を表す *$USER* または *$LOGNAME*。この変数を使用すると、個々のユーザーに `.cuf` ファイルを作成し、ユーザーのログイン名を `.cuf` ファイルの名前または親ディレクトリーの名前に含めることができます。
- コンパイラーが稼働するホストの名前を表す *$HOSTNAME*。これは異なるホストにまたがって使用状況をトラッキングする際に有効です。

-qfileaccessmaxwait=*number_of_milliseconds*

使用量ファイルへのアクセスを待機する最大ミリ秒数を指定します。

注: この項目は、システムに極端な負荷がかかっており、使用量ファイルへのアクセスが遅延する異常な状況に対処するために使用されます。

デフォルトは 3000 ミリ秒です。

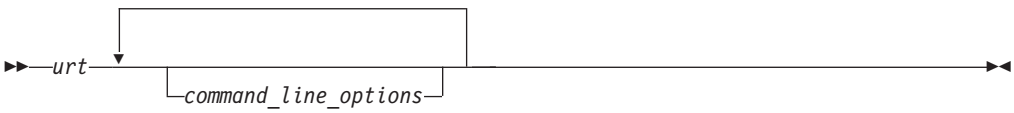
注:

- これらの項目はコンパイラー・オプションではありません。使用状況トラッキング構成ファイルでのみ使用されます。
- **-qexemptconcurrentusers** 項目が、使用状況トラッキング構成ファイルで複数回指定された場合、指定されたすべてのインスタンスが使用されます。その他の項目が複数回指定された場合、最後の項目の値は前の項目の値をオーバーライドします。

使用状況レポート作成ツールについて

使用状況レポート作成ツールを使用して、1 つ以上の使用量ファイル内の情報からコンパイラー使用状況レポートを生成できます。さらに、レポートの生成時に、オプションでそのレポートを整理することができます。

このツールは `/opt/ibmurt/1.2/bin` ディレクトリーにあります。ツールの呼び出しには **urt** コマンドを使用できます。**urt** コマンドの構文は、次のとおりです。



生成されたレポートは、標準出力に表示されます。レポートを保持する場合には、出力をファイルに送ることができます。

コマンド行オプションにより、使用状況レポートの生成方法を制御します。このオプションについて詳しくは、『使用状況レポート作成ツールのコマンド行オプション』を参照してください。

デフォルトの構成ファイル ibmurt.cfg は、/opt/ibmurt/1.2/config ディレクトリーに提供されています。このファイル内の項目はコマンド行オプションの形式を取り、同じ効果を持ちます。また、追加の構成ファイルを作成し、**-qconfigfile** オプションを使用してその名前を指定することもできます。

オプションは、次の 1 つ以上の場所で指定することができます。

- デフォルトの構成ファイル
- **-qconfigfile** に指定された追加構成ファイル
- コマンド行

使用状況レポート作成ツールは、コマンド行のオプションを使用する前に、デフォルトの構成ファイルのオプションを使用します。このツールは、コマンド行で **-qconfigfile** オプションを見つけた場合には、指定された構成ファイルのオプションを読み取り、それらのオプションを **-qconfigfile** オプションが使用される場所のコマンド行に配置します。

あるオプションを複数指定すると、ツールが最後に検出した指定が有効になります。**-qconfigfile** と **-qsameuser** は例外です。これらの 2 つのオプションについては、すべての指定が有効になります。

使用状況レポート作成ツールのコマンド行オプション

使用状況レポート作成ツールのコマンド行オプションは、コンパイラー使用状況レポートの生成を制御します。

これらのコマンド行オプションを使用して、コンパイラー使用状況レポートの詳細を変更してください。

-qreporttype=detail | maxconcurrent

生成する使用状況レポートのタイプを指定します。

- **detail** を指定すると、コンパイラーのすべての呼び出しが使用状況レポートにリストされます。このサブオプションでは、詳細なレポートが生成され、許可される最大同時ユーザー数を超えた呼び出しがリストされます。
- **maxconcurrent** を指定すると、許可される最大同時ユーザー数を超えたコンパイラー呼び出しだけがリストされます。このサブオプションでは、許容される同時ユーザーの最大数以内の呼び出しをリストしない簡潔なレポートが生成されます。

注: 許可される最大同時ユーザー数は、使用状況トラッキング構成ファイルの **-qmaxconcurrentusers** 項目で指定します。

デフォルト: **-qreporttype=maxconcurrent**。

-qrptmaxrecords=*num* **| nomax**

製品ごとにレポート内にリストする最大レコード数を指定します。*num* は、正整数でなければなりません。

デフォルト: **-qrptmaxrecords=nomax**。すべてのレコードがリストされることを意味します。

-qusagefileloc=*directory_or_file_name*

レポートの生成または整理に関して、使用量ファイルの場所を指定します。ディレクトリー名またはファイル名のリスト、またはその両方をコロンで区切ったリストにすることができます。

このオプションの指定時には、以下の規則が適用されます。

- 1 つ以上のディレクトリーを指定した場合は、それらのディレクトリーにある、.cuf 拡張子が付いたすべてのファイルが使用されます。また、**-qmaxsubdirs** オプションでサブディレクトリーも検索できます。
- 指定されたディレクトリーのパスは、相対パスまたは絶対パスにできます。相対パスは、コンパイラー・ユーザーの現行作業ディレクトリーからの相対です。
- シンボリック・リンク (symlink) には .cuf 拡張子は必要ありませんが、指されるファイルは拡張子を持つ必要があります。

注:

- 使用状況トラッキング構成ファイル内の **-qusagefileloc** 項目は、コンパイラー使用状況の記録にどの使用量ファイルを使用するかを指示します。**-qusagefileloc** オプションは、使用状況レポート作成ツールに対して、それれらの使用量ファイルの存在する場所を指示します。

デフォルトは *.:$HOME* で、使用状況レポート作成ツールが、現行作業ディレクトリーとホーム・ディレクトリーで使用量ファイルを検索することを意味しています。

-qmaxsubdirs=*num* **| nomax**

サブディレクトリーで使用量ファイルを検索するかどうか、および何段階のサブディレクトリーを検索するかを指定します。*num* は、負でない整数でなければなりません。

nomax を指定すると、すべてのサブディレクトリーが検索されます。0 を指定すると、サブディレクトリーは検索されません。

デフォルト: 0。

-qconfigfile=*file_path*

使用するユーザー定義の構成ファイルを指定します。

使用状況レポート作成ツールでの構成ファイルの使用方法について詳しくは、『 69 ページの『使用状況レポート作成ツールについて』』を参照してください。

注: このオプションを複数回指定すると、指定されたすべてのインスタンスが使用されます。

-qsameuser=*user_account_info*

同じコンパイラー・ユーザーに所属する異なるユーザー・アカウントを指定します。1 人のユーザーが複数のユーザーとして報告されないように、そのユーザーが複数のユーザー ID またはマシンからコンパイラーにアクセスする際にこの

オプションを使用します。これらの異なるアカウントによるコンパイラーの呼び出しは、複数の異なるユーザーではなく単一ユーザーとしてカウントされます。

user_account_info は、以下の項目の任意の組み合わせにできます。

- name(*user_name*)
- uid(*user_ID*)
- host(*host_name*)

これらの規則を使用状況レポート作成ツールに渡すには 2 つの方法があります。同じユーザーで共有される *user_name*、*user_ID* または *host_name* の特定のリストを提供するか、より汎用的な (=) 構文を使用することができます。

例えば、*user1* と *user2* が両方とも *host1* マシン上でコンパイラーを使用する同じ人に属するユーザー名であることを示すためには、次のようにこれらのユーザー名とホスト名を明示的に指定する構文を使用します。

```
-qsameuser="name(user1)host(host1) | name(user2)host(host1)"
```

または

```
-qsameuser="name(user1 user2)host(host1)"
```

上記の例は、両方とも特定のユーザー名とホスト名を使用して同じユーザーに属するアカウントを示していますが、その方法には若干の違いがあります。最初の例では、このユーザーに属する異なるユーザー・アカウントを分けるために縦線を使用していますが、 2 番目の例では、同じホスト情報を 2 度繰り返す代わりに、括弧で囲んだユーザー名のリストを使用しています。両方の例は同じアカウント情報を伝えていますが、2 番目の例がより簡潔です。

より汎用的な (=) 構文の例として、以下のように同じユーザー名とユーザー ID を持つすべてのユーザー・アカウントが同じユーザーに属することを示すことができます。

```
-qsameuser="name(=)uid(=)"
```

このオプションでは、前の例で指定したように、特定のユーザー名またはユーザー ID を指定しません。同じユーザー名とユーザー ID を持つユーザー・アカウントは、特定のユーザー名、ユーザー ID、ホスト名の指定内容に関わらず、同じユーザーに属するとみなされます。これにより、組織において特定のアカウントではなくすべてのアカウントに適用する汎用的な規則が確立します。

このオプションの指定時には、以下の規則が適用されます。

- **-qsameuser** オプションのインスタンスごとに、リストまたは汎用的な (=) 構文のいずれかを使用する必要があります。これらをオプションの同じインスタンスで組み合わせることはできませんが、 **-qsameuser** オプションの複数のインスタンスを使用して、レポートを詳細化することができます。
- 使用状況レポート作成ツールは、**-qsameuser** オプション値が指定された順序に基づいてユーザー情報の突き合わせを行います。一致を検出すると、同じユーザー情報をそれ以降のオプションに対して突き合わせるのを停止します。

  以下の例で、その違いを説明します。

  – 以下のように **-qsameuser** オプションを指定する場合を考えます。

    ```
    -qsameuser="name(user1)" -qsameuser="uid(=)"
    ```

-qsameuser オプションをこの順序で指定することは、ユーザー名 *user1* を持つユーザー・アカウントが最初のオプションに一致し、 2 番目のオプションに対しては評価されないことを意味しています。ユーザー・アカウント *user1* と *user2* は、同じ *uid* を持つ場合にも、同じユーザーとは解釈されません。

– 以下のように **-qsameuser** オプションを指定する場合を考えます。

```
-qsameuser="uid(=)" -qsameuser="name(user1)"
```

-qsameuser オプションをこの順序で指定することは、同じ *uid* を持つユーザー・アカウントは常に同じユーザーと解釈され、さらにユーザー名 *user1* を持つどのユーザー・アカウントも、 *uid* で一致しなくても同じユーザーに属すると解釈されなければならないことを意味しています。

注: このオプションを指定しても、ユーザー情報は使用状況レポートにリストされます。同時ユーザーについて詳しくは、 60 ページの『同時ユーザーに関する考慮事項』を参照してください。

-qadjusttime=*time_adjustments*

指定されたマシンに関して、使用量ファイルに記録される時刻を調整します。 *time_adjustments* は、*machine name* + | - *number of seconds* の形式を持ち、コロンで区切られた項目のリストです。

このオプションの使用時には、以下の規則が適用されます。

- *machine name* + *number of seconds* の形の項目を指定すると、指定したマシンに関して記録されたすべての呼び出しの開始時刻および終了時刻に、指定した秒数が加算されます。
- *machine name* - *number of seconds* の形の項目を指定すると、指定したマシンに関して記録されたすべての呼び出しの開始時刻および終了時刻から、指定した秒数が減算されます。

例を以下に示します。

```
-qadjusttime="hostA+5:hostB-3"
```

hostA の呼び出しの開始時刻および終了時刻に 5 秒が加算され、hostB の呼び出しの開始時刻および終了時刻から 3 秒が減算されます。

このオプションは、 2 台以上のマシンの使用状況情報が使用量ファイル内に存在し、時刻がこれらのマシン間で同期されていない場合にのみ使用してください。このオプションで指定された調整により、同期の欠落が補正されます。

注:

- 指定した調整は、**urt** コマンドの現行の実行にのみ使用されます。このオプションを指定しても、使用量ファイルに記録される呼び出し情報は変化しません。
- このオプションで、同じマシン名を複数回指定しないでください。

-qusagefilemaxage=*number_of_days* | **nomax**

指定した日数より古いすべての呼び出しを削除することにより、使用量ファイルを整理します。

-qusagefileloc オプションで指定されたすべての使用量ファイルが整理されます。使用状況レポートには、整理されたレコード数を示す情報が含まれます。

デフォルト: **-qusagefilemaxage=nomax**。整理を行わないことを意味します。

-qusagefilemaxsize=*number_of_MB* | **nomax**

指定したサイズ未満に保つように使用量ファイルを整理します。ファイルは、最も古い呼び出しを削除することにより整理されます。

-qusagefileloc オプションで指定されたすべての使用量ファイルが整理されます。使用状況レポートには、整理されたレコード数を示す情報が含まれます。

デフォルト: **-qusagefilemaxsize=nomax**。整理を行わないことを意味します。

-qtimesort=ascend | **descend**

使用状況レポートをソートする年代順の順序を指定します。

- **ascend** を指定すると、新しい情報が古い情報の後にリストされます。
- **descend** を指定すると、最新情報がレポートの先頭になります。

デフォルト: **-qtimesort=ascend**。

使用状況レポートの生成

使用状況レポート作成ツールを使用すると、使用量ファイルに格納されている使用量情報に基づいてコンパイラー使用状況レポートを生成できます。

レポートを生成するには、コマンド行オプションまたは **urt** 構成ファイルを使用して、レポートの生成方法を指定します。これらのオプションの詳細については、70 ページの『使用状況レポート作成ツールのコマンド行オプション』を参照してください。

注:

- クーロン・サービスをセットアップして、使用状況レポート作成ツールを定期的に実行できます。ツールがレポート生成に使用する使用量ファイルをツールを実行するマシンにコピーする必要がある場合は、このコピー作業もクーロンにより自動化できます。
- 使用量ファイルへの読み取りアクセスと書き込みアクセスの競合を削減するために、コンパイラーの使用時にツールの実行および使用量ファイルのコピーを行わないでください。

生成されたレポートは標準出力に表示されます。レポートを保持する場合には、出力をファイルに送ることができます。

使用状況レポートの生成後、使用状況レポート作成ツールは以下の終了コードを使用して、コンパイラー・ライセンスの準拠状況を示します。

- 終了コード ="1"。

  使用状況レポート作成ツールは、使用状況トラッキング構成ファイルの **-qmaxconcurrentusers** 項目に指定されている同時ユーザー・ライセンス資格の数を超過していることを検出しました。追加の同時ユーザー・ライセンスを購入するには、生成されたレポートの詳細を確認して IBM 担当員にお問い合わせください。

- 終了コード ="0"。

  使用されているコンパイラーの数は、指定された同時ユーザー・ライセンス資格の数以内です。

urt コマンドの詳細については、 69 ページの『使用状況レポート作成ツールについて』を参照してください。

使用状況レポートについて

使用状況レポート作成ツールが生成するレポートを使用して、組織内のコンパイラー使用量を分析することができます。

このレポートには、以下の情報をリストした REPORT SUMMARY セクションがあります。

1. レポートが生成された日時。
2. レポートを生成するために使用された .cuf ファイル、または使用されたすべての .cuf ファイルのリスト。
3. **urt** コマンドに渡されたオプションと、指定されなかったオプションのデフォルト値。
4. ERROR、WARNING、または INFO として分類できるメッセージ。出力される可能性のあるメッセージについて詳しくは、 79 ページの『使用状況トラッキングおよびレポート作成からの診断メッセージ』を参照してください。

要約セクションの後に、コンパイラー・バージョンごとに REPORT DETAILS セクションがあります。このセクションでは、使用量ファイルに記録されたすべてのコンパイラー呼び出しをリストします。これらのセクションの内容は、指定したレポートのタイプによって異なります。レポート・タイプについて詳しくは、**-qreporttype** を参照してください。

以下に、2 つの異なるレポート・タイプで生成したサンプルのレポートを示します。

サンプル 1: **-qreporttype=detail** で生成したサンプルのレポート

```
REPORT SUMMARY
--------------

DATE: 12/18/13      TIME: 01:30:24

OPTIONS USED (* indicates that a default value was used):

reporttype=detail
maxsubdirs=0
configfile="/opt/ibmurt/1.2/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax


FILES USED:

/home/testrun/ibmxlcompiler.cuf
```

```
REPORT DETAILS
--------------

USAGE INFORMATION FOR PRODUCT: IBM XL C/C++ for Linux 13.1

Max. Concurrent Users Exceeded? : *** YES ***

Max. Concurrent Users Allowed: 1          Max. Concurrent Users Recorded: 5

Exempt Users:

Product invocations:

Start Time         End Time           User                Number of Concurrent Users
------------------ ------------------ ------------------  --------------------------
12/17/13 16:56:44  12/17/13 16:57:13  user1@host1.ibm.com 1
12/18/13 00:58:29  12/18/13 00:58:32  user2@host2.ibm.com 1
12/18/13 01:16:01  12/18/13 01:16:02  user3@host3.ibm.com 5 ( exceeds max. allowed)
12/18/13 01:16:02  12/18/13 01:16:26  user2@host2.ibm.com 5 ( exceeds max. allowed)
12/18/13 01:16:08  12/18/13 01:16:08  user3@host2.ibm.com 5 ( exceeds max. allowed)
12/18/13 01:16:12  12/18/13 01:16:12  user2@host1.ibm.com 5 ( exceeds max. allowed)
12/18/13 01:16:24  12/18/13 01:16:28  user1@host2.ibm.com 5 ( exceeds max. allowed)
12/18/13 01:26:11  12/18/13 01:27:46  user3@host3.ibm.com 2 ( exceeds max. allowed)
12/18/13 01:26:27  12/18/13 01:27:46  user1@host1.ibm.com 2 ( exceeds max. allowed)
12/18/13 01:29:59  12/18/13 01:30:00  user2@host1.ibm.com 1
12/18/13 01:30:00  12/18/13 01:30:00  user2@host2.ibm.com 3 ( exceeds max. allowed)
12/18/13 01:30:14  12/18/13 01:30:15  user3@host1.ibm.com 3 ( exceeds max. allowed)
12/18/13 01:30:14  12/18/13 01:30:14  user2@host2.ibm.com 3 ( exceeds max. allowed)
```

サンプル 2: **-qreporttype=maxconcurrent** で生成したサンプルのレポート

```
REPORT SUMMARY
--------------

DATE: 12/18/13      TIME: 01:32:53

OPTIONS USED (* indicates that a default value was used):

reporttype=maxconcurrent
maxsubdirs=0
configfile="/opt/ibmurt/1.2/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax


FILES USED:

/home/testrun/ibmxlcompiler.cuf

REPORT DETAILS
--------------

USAGE INFORMATION FOR PRODUCT: IBM XL C/C++ for Linux 13.1

Max. Concurrent Users Exceeded? : *** YES ***

Max. Concurrent Users Allowed: 1          Max. Concurrent Users Recorded: 5

Exempt Users:

Dates and times where usage exceeded the maximum allowed:
```

```
Date          Time        Number of Concurrent Users    Users
------------  ----        --------------------------    -----
12/18/13      01:16:01    5                             user3@host3.ibm.com
                                                        user2@host2.ibm.com
                                                        user3@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user1@host2.ibm.com
12/18/13      01:16:02    5                             user3@host3.ibm.com
                                                        user2@host2.ibm.com
                                                        user3@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user1@host2.ibm.com
12/18/13      01:16:08    5                             user3@host3.ibm.com
                                                        user2@host2.ibm.com
                                                        user3@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user1@host2.ibm.com
12/18/13      01:16:12    5                             user3@host3.ibm.com
                                                        user2@host2.ibm.com
                                                        user3@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user1@host2.ibm.com
12/18/13      01:16:24    5                             user3@host3.ibm.com
                                                        user2@host2.ibm.com
                                                        user3@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user1@host2.ibm.com
12/18/13      01:26:11    2                             user3@host3.ibm.com
                                                        user1@host1.ibm.com
12/18/13      01:26:27    2                             user3@host3.ibm.com
                                                        user1@host1.ibm.com
12/18/13      01:30:00    3                             user2@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user3@host1.ibm.com
12/18/13      01:30:14    3                             user2@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user3@host1.ibm.com
12/18/13      01:30:14    3                             user2@host2.ibm.com
                                                        user2@host1.ibm.com
                                                        user3@host1.ibm.com
```

注: 終了時刻が記録されない可能性のある状況が存在します。これには、以下の場合が含まれます。

- 例えばコンパイル中の停電など、コンパイラーの大きな障害。
- レポートの生成時や使用量ファイルのコピー時に、呼び出しが終了しなかった。
- 呼び出しの終了時刻が記録される前のいずれかの時点で、使用量ファイルに書き込む権限が取り消された。

終了時刻が記録されていない呼び出しは、同時ユーザーのカウントに含まれません。

使用量ファイルの整理

使用量ファイルは、コンパイラーを呼び出すごとに増大します。これらの使用量ファイルは、使用状況レポート作成ツールで整理することができます。

使用状況レポートの生成時に、以下の 2 つのオプションを指定して、使用量ファイルを任意に整理することができます。

- **-qusagefilemaxage**: 指定した日数より古い呼び出しを削除します。例えば、30 日より古い使用量ファイルのすべてのエントリーを削除するには、次のコマンドを使用します。

  ```
  urt -qusagefilemaxage=30
  ```

- **-qusagefilemaxsize**: 最も古い呼び出しを削除して、使用量ファイルを指定されたサイズ未満に保持します。例えば、最も古い呼び出しを削除して、使用量ファイルを 30 MB 未満に保持するには、次のコマンドを使用します。

  ```
  urt -qusagefilemaxsize=30
  ```

使用量ファイルを整理すると、整理されたレコード数を示す情報メッセージが使用状況レポートに入ります。ファイルを整理した後、生成されたレポートを保持したい場合には、出力をファイルにリダイレクトすることができます。

使用量ファイルのサイズを制御するには、使用量ファイルを定期的に整理します。クーロン・ジョブを作成して、これを自動的に実行することができます。

使用量ファイルがあるマシンごとに使用状況レポート作成ツールをインストールしていない場合は、以下のオプションがあります。

- 使用量ファイルが存在するマシンごとにファイル・システムをエクスポートして、これを使用状況レポート作成ツールがインストールされているマシンにマウントします。次にツールを実行して、マウントされたネットワーク・ファイル・システム上の使用量ファイルを整理します。
- 使用状況レポート作成ツールがインストールされているマシンに使用量ファイルをコピーしてから、元のファイルを削除し、新規の使用量ファイルを使用して、この時点以降のコンパイラー呼び出しをすべて取り込みます。このアプローチを用いると、使用状況レポート作成ツールが使用量ファイルにリモートにアクセスせず、使用量ファイルの整理に時間を費やさないため、レポート生成のスピードも上がる場合があります。

使用量ファイルを整理すると、特に使用量ファイルの数が多い場合やサイズが大きい場合に、使用状況レポート生成処理の速度が低下することがあります。レポートの生成ごとに毎回ファイルを整理したくない場合には、**-qusagefilemaxage** オプションと **-qusagefilemaxsize** オプションの値を以下のように設定します。

- 毎日レポートを生成する場合には、以下の 2 つのオプションに非常に大きな値を設定して、整理が行われないようにします。この場合、デフォルト値の **nomax** を使用できます。
- この 2 つのオプションに適切な値を設定し、別のクーロン・ジョブを使用することで、使用量ファイルの整理が週単位で行われるようにします。

注: 使用量ファイルに対する読み取りアクセスと書き込みアクセスの競合を削減するために、コンパイラーの使用中は、使用状況レポート作成ツールの実行および使用量ファイルのコピーを行わないでください。

使用状況トラッキングおよびレポート作成からの診断メッセージ

コンパイラーは、使用状況トラッキングの問題を示す診断メッセージを生成します。これらのメッセージは、関連する問題の修正に役立ちます。

例を以下に示します。

```
許可が不十分のため、使用状況トラッキング構成ファイルを読み取ることが
できませんでした。
(Utilization tracking configuration file could not be read due to
insufficient permissions.)
```

このメッセージは、使用状況トラッキング構成ファイルに対する読み取り権限が必要なことを示しています。

使用状況レポート作成ツールを使用して使用状況レポートを作成した場合、または使用量ファイルを整理した場合には、やはり診断メッセージが生成されます。例を以下に示します。

```
認識されないオプション -qmaxsubdir。(Unrecognized option -qmaxsubdir.)
```

このメッセージは、誤ったオプションが指定されたことを示しています。

注: 考えられるエラー、警告、または情報メッセージは、ツールによって生成されるコンパイラー使用状況レポートにも含まれます。

第 4 章 コンパイラー・オプション参照

以下のセクションには、機能カテゴリーごとに XL C/C++ で使用可能なコンパイラー・オプションの要約、および個々のオプションの詳細な説明が記載されています。

関連情報
- 6 ページの『コンパイラー・オプションの指定』
- 12 ページの『gxlc および gxlc++ での GNU C/C++ コンパイラー・オプションの再使用』

機能カテゴリー別コンパイラー・オプションの要約

Linux プラットフォームで使用可能な XL C/C++ オプションは、以下のカテゴリーにグループ化されています。オプションが同等のプラグマ・ディレクティブをサポートする場合は、これが示されます。リストにあるオプションの詳細を参照するには、そのオプションの詳しい説明を参照してください。
- 『出力制御』
- 82 ページの『入力制御』
- 83 ページの『言語エレメント制御』
- 85 ページの『テンプレート制御 (C++ のみ)』
- 86 ページの『浮動小数点および整数制御』
- 89 ページの『エラー・チェックおよびデバッグ』
- 92 ページの『リスト、メッセージ、およびコンパイラー情報』
- 94 ページの『最適化およびチューニング』
- 87 ページの『オブジェクト・コード制御』
- 98 ページの『リンク』
- 99 ページの『移植性とマイグレーション』
- 99 ページの『コンパイラーのカスタマイズ』

出力制御

このカテゴリーのオプションは、コンパイラーが作成するファイル出力のタイプ、および出力のロケーションを制御します。以下は、呼び出されるコンパイラー・コンポーネント、実行される (またはされない) プリプロセッシング、コンパイル、およびリンクのステップ、そして生成される出力の種類を判別する基本オプションです。

表 10. コンパイラー出力オプション

オプション名	同等のプラグマ名	説明
123 ページの『-c』	なし。	コンパイラーに、ソース・ファイルをコンパイルまたはアセンブルすることのみを指示し、リンクすることは指示しない。このオプションでは、出力は各ソース・ファイルの .o ファイルです。

表 10. コンパイラー出力オプション (続き)

オプション名	同等のプラグマ名	説明
124 ページの『-C、-C!』	なし。	**-E** または **-P** オプションと使用すると、プリプロセスされた出力内のコメントを保存または除去する。
150 ページの『-E』	なし。	コンパイラー呼び出しに指定されたソース・ファイルをプリプロセスし、コンパイルせずに出力を標準出力に書き込む。
268 ページの『-qmakedep, -M』	なし。	各ソース・ファイルに対して **make** ツールによって使用される従属関係ファイルを生成する。
275 ページの『-MF』	なし。	**-qmakedep** または **-M** オプションによって生成される従属関係出力ファイルの名前または場所を指定する。
277 ページの『-qmkshrobj』	なし。	生成されたオブジェクト・ファイルから共有オブジェクトを作成する。
278 ページの『-o』	なし。	出力オブジェクト、アセンブラー、または実行可能ファイルの名前を指定する。
288 ページの『-P』	なし。	コンパイラー呼び出しに指定されたソース・ファイルをプリプロセスし、コンパイルせずにそれぞれの入力ファイルごとにプリプロセスされた出力ファイルを作成する。
324 ページの『-S』	なし。	ソース・ファイルごとにアセンブラー言語ファイルを生成する。
329 ページの『-qshowmacros』	なし。	プリプロセスされた出力にマクロ定義を出す。
372 ページの『-qtimestamps』	なし。	暗黙的なタイム・スタンプがオブジェクト・ファイルに挿入されるようにするかどうかを制御する。

入力制御

このカテゴリーのオプションは、ソース・ファイルのタイプおよびロケーションを指定します。

表 11. コンパイラー入力オプション

オプション名	同等のプラグマ名	説明
103 ページの『-+ (正符号) (C++ のみ)』	なし。	ファイルを C++ 言語ファイルとしてコンパイルする。
132 ページの『-qcinc (C++ のみ)』	なし。	`extern "C" { }` ラッパーを指定したディレクトリー内の組み込みファイルのコンテンツのまわりに配置する。
186 ページの『-I』	なし。	ディレクトリーを組み込みファイルの検索パスに追加する。
188 ページの『-qidirfirst』	#pragma options idirfirst	他のディレクトリーを検索する*前* または後 に **-I** オプションによって指定されたディレクトリーで、コンパイラーがユーザー組み込みファイルを検索するかどうかを指定する。
191 ページの『-qinclude』	なし。	ソース・ファイルの `#include` 文で指定されているかのようにコンパイル単位に組み込まれる追加のヘッダー・ファイルを指定する。
342 ページの『-qsourcetype』	なし。	実際のファイル名サフィックスに関係なく、すべての認識されるソース・ファイルを指定されたソース・タイプとして扱うようコンパイラーに命令する。
351 ページの『-qstdinc』	#pragma options stdinc	標準組み込みディレクトリーがシステムおよびユーザー・ヘッダー・ファイルの検索パスに組み込まれているかどうかを指定する。

言語エレメント制御

このカテゴリーのオプションによって、ソース・コードの特性を指定することができます。また、このオプションを使用すると言語制限を強制または緩和したり、言語拡張を使用可能または使用不可にしたりすることができます。

表 12. 言語エレメント制御オプション

オプション名	同等のプラグマ名	説明
112 ページの『-qaltivec』	なし。	Vector データ型およびオペレーターに対するコンパイラー・サポートを使用可能にする。
116 ページの『-qasm』	なし。	コードの解釈およびその後のコードの生成を制御して、アセンブラー言語を拡張します。

表 12. 言語エレメント制御オプション (続き)

オプション名	同等のプラグマ名	説明
137 ページの『-qcpluscmt (C のみ)』	なし。	C ソース・ファイルでの C++ 形式のコメントの認識を使用可能にする。
141 ページの『-D』	なし。	マクロ `#define` プリプロセッサー・ディレクティブ内と同様に定義する。
146 ページの『-qdigraph』	#pragma options digraph	一部のキーボードにない文字を表すため、連字キーの組み合わせ C++ およびオペレーター・キーワード C++ の認識を使用可能にする。連字キーの組み合わせには、`<:`、`<%` などが含まれます。 C++ オペレーター・キーワードには、`and`、`or` などが含まれます。 C++
147 ページの『-qdollar』	#pragma options dollar	ID の名前にドル記号 ($) シンボルを使用できるようにする。
190 ページの『-qignprag』	#pragma options ignprag	特定のプラグマ・ステートメントを無視するようにコンパイラーに命令する。
222 ページの『-qkeyword』	なし。	指定された名前がプログラム・ソースに現れたときに、それをキーワードとして処理するかまたは ID として処理するかを制御する。
227 ページの『-qlanglvl』	C #pragma options langlvl、#pragma langlvl	ソース・コードおよびコンパイラー・オプションが固有の言語標準、または標準のサブセットまたはスーパーセットに準拠しているかを検査するかどうかを判別する。
267 ページの『-qlonglong』	#pragma options long long	プログラムで IBM `long long` 整数型を許可する。
273 ページの『-qmbcs, -qdbcs』	#pragma options mbcs、#pragma options dbcs	ソース・コード内で、マルチバイト文字セット (MBCS) およびユニコード文字のサポートを使用可能にする。
346 ページの『-qstaticinline (C++ のみ)』	なし。	インライン関数を静的または外部リンケージとして扱うかを制御する。
363 ページの『-qtabsize』	なし。	エラー・メッセージで列番号を報告するために、デフォルトのタブの長さを設定する。

表 12. 言語エレメント制御オプション (続き)

オプション名	同等のプラグマ名	説明
376 ページの『-qtrigraph』	なし。	キーボードにない文字を表すために、3 文字表記キーの組み合わせの認識を使用可能にする。
380 ページの『-U』	なし。	コンパイラーまたは **-D** コンパイラー・オプションによって定義されたマクロ名の定義を解除する。
386 ページの『-qutf』	なし。	UTF リテラル構文の認識を使用可能にする。

テンプレート制御 (C++ のみ)

以下のオプションを使用すると、C++ コンパイラーがテンプレートを処理する方法を制御することができます。

表 13. C++ テンプレート・オプション

オプション名	同等のプラグマ名	説明
365 ページの『-qtempinc (C++ のみ)』	なし。	テンプレート関数およびクラス宣言用に別個のテンプレート組み込みファイルを生成し、オプションで指定できるディレクトリーにこれらのファイルを入れます。
366 ページの『-qtemplatedepth (C++ のみ)』	なし。	再帰的にインスタンス化され、コンパイラーによって処理されるテンプレートの特殊化の最大数を指定する。
367 ページの『-qtemplaterecompile (C++ のみ)』	なし。	**-qtemplateregistry** コンパイラー・オプションを使用してコンパイルされたコンパイル単位間の依存性の管理に役立つ。
368 ページの『-qtemplateregistry (C++ のみ)』	なし。	ソース内で検出されるすべてのテンプレートについてそのレコードを保守し、テンプレートごとに一度だけインスタンス化が行われることを保証する。
370 ページの『-qtempmax (C++ のみ)』	なし。	**-qtempinc** オプションにより各ヘッダー・ファイルごとに生成されるテンプレート組み込みファイルの最大数を指定する。

表 13. C++ テンプレート・オプション (続き)

オプション名	同等のプラグマ名	説明
374 ページの『-qtmplinst (C++ のみ)』	なし。	テンプレートの暗黙のインスタンス生成を管理する。
375 ページの『-qtmplparse (C++ のみ)』	なし。	構文解析とセマンティック検査がテンプレート定義に適用されるかどうかを制御する。
C++11 -qlanglvl=[no]externtemplate	なし。	テンプレートの特殊化またはそのメンバーの暗黙のインスタンス生成を抑止する。
-qlanglvl=[no]gnu_externtemplate	なし。	テンプレートの特殊化またはそのメンバーの暗黙のインスタンス生成を抑止する。このオプションは、XL C/C++ V13.1 では推奨されません。代わりに、オプション **-qlanglvl=[no]externtemplate** を使用できます。
C++11 -qlanglvl=[no]variadic[templates]	なし。	任意の数 (ゼロを含む) のパラメーターを持つことができるクラスまたは関数テンプレートを定義する。

浮動小数点および整数制御

アプリケーションがどのように計算を行うかを詳細に指定すると、システムの浮動小数点のパフォーマンスや精度 (丸めの送信方法など) をより効率的に活用することができます。IEEE 浮動小数点の仕様に順守しすぎると、アプリケーションのパフォーマンスに影響を及ぼす可能性があるので注意してください。以下の表のオプションを使用して、浮動小数点パフォーマンスと IEEE 標準への順守間のトレードオフを制御することができます。

表 14. 浮動小数点と整数の制御

オプション名	同等のプラグマ名	説明
123 ページの『-qbitfields』	なし。	ビット・フィールドを符号付きにするか符号なしにするかを指定する。
128 ページの『-qchars』	#pragma options chars、#pragma chars	char 型のすべての変数を符号付きまたは符号なしのいずれかとして処理するかを判別する。
152 ページの『-qenum』	#pragma options enum、#pragma enum	列挙が占有するストレージの量を指定する。
160 ページの『-qfloat』	#pragma options float	浮動小数点の計算を高速化したり、精度を上げるためのさまざまなストラテジーを選択する。

表 14. 浮動小数点と整数の制御 (続き)

オプション名	同等のプラグマ名	説明
256 ページの『-qldbl128』	#pragma options ldbl128	long double 型のサイズを 64 ビットから 128 ビットに増加させる。
267 ページの『-qlonglit』	なし。	64 ビット・モードで整数リテラルの暗黙の型を決める際に、コンパイラーは l または L 接尾部が接尾部のない整数リテラルまたは接尾部が u か U のみである整数リテラルに追加されたかのように動作する。
400 ページの『-y』	なし。	コンパイル時に定数浮動小数点式を評価する場合にコンパイラーが使用する丸めモードを指定する。

オブジェクト・コード制御

これらのオプションは、オブジェクト・コードの特性、前処理されたコード、またはコンパイラーが生成したその他の出力に影響を与えます。

表 15. オブジェクト・コード制御オプション

オプション名	同等のプラグマ名	説明
104 ページの『-q32、-q64』	なし。	32 ビットまたは 64 ビットのコンパイラー・モードを選択します。
111 ページの『-qalloca、-ma (C のみ)』	#pragma alloca	alloca.h ヘッダーを含まないソース・コードから呼び出されるとき、システム関数 alloca のインライン定義を提供する。
133 ページの『-qcommon』	なし。	未初期化グローバル変数が割り振られる場所を制御する。
151 ページの『-qeh (C++ のみ)』	なし。	例外処理がコンパイルされているモジュールで使用可能であるかどうかを制御する。
206 ページの『-qinlglue』	#pragma options inlglue	**-O2** 以上の最適化で使用すると、アプリケーション内の外部関数呼び出しを最適化するグルー・コードをインライン化する。
220 ページの『-qkeepinlines (C++ のみ)』	なし。	参照されていない extern インライン関数の定義を保持または破棄する。
303 ページの『-qpic』	なし。	共有ライブラリーでの使用に適した位置独立コードを生成する。

表 15. オブジェクト・コード制御オプション (続き)

オプション名	同等のプラグマ名	説明
304 ページの『-qppline』	なし。	**-E** または **-P** オプションと一緒に使用すると、`#line` ディレクティブの生成を使用可能または使用不可にする。
309 ページの『-qpriority (C++ のみ)』	#pragma options priority、#pragma priority	静的オブジェクトの初期化の優先順位を指定する。
313 ページの『-qproto (C のみ)』	#pragma options proto	浮動小数点引数をプロトタイプ化されていない関数へ渡すためのリンケージ規約を指定する。
314 ページの『-r』	なし。	別の ID コマンド呼び出しでの入力ファイルとして使用する実行不能出力ファイルを生成する。このファイルには未解決のシンボルも含むことができる。
317 ページの『-qreserved_reg』	なし。	スタック・ポインター、フレーム・ポインター、またはその他の固定の役割を除き、指定されたレジスターのリストがコンパイル中に使用できないことを示します。
320 ページの『-qro』	#pragma options ro、 #pragma strings	ストリング・リテラルのストレージ・タイプを指定する。
321 ページの『-qroconst』	#pragma options roconst	定数値のストレージ・ロケーションを指定する。
322 ページの『-qrtti (C++ のみ)』	なし。	`typeid` 演算子および `dynamic_cast` 演算子による例外処理および使用のための実行時型識別 (RTTI) 情報を生成する。
323 ページの『-s』	なし。	出力ファイルからシンボル・テーブル、行番号情報、および再配置情報をストリップする。
325 ページの『-qsaveopt』	なし。	ソース・ファイルのコンパイルに使用するコマンド行オプション、構成ファイルで指定されたユーザーの構成ファイル名とオプション、コンパイル中に呼び出される各コンパイラー・コンポーネントのバージョンとレベル、およびその他の情報を対応するオブジェクト・ファイルに保管する。

表 15. オブジェクト・コード制御オプション (続き)

オプション名	同等のプラグマ名	説明
345 ページの『-qstackprotect』	なし。	スタックを上書きするか破損する悪質なコードまたはプログラム・エラーに対する保護を提供する。
350 ページの『-qstatsym』	なし。	永続的なストレージ・クラスを持つ、ユーザー定義の非外部名 (初期化される静的変数または初期化されない静的変数など) をオブジェクト・ファイルのシンボル・テーブルに追加する。
364 ページの『-qtbtable』	#pragma options tbtable	オブジェクト・ファイルに組み込まれるデバッグ・トレースバック情報の量を制御する。
371 ページの『-qthreaded』	なし。	スレッド・セーフ・コードを生成する必要があるかどうかをコンパイラーに指示する。
372 ページの『-qtls』	なし。	スレッド・ローカル・ストレージが割り振られる変数を指定する __thread ストレージ・クラス指定子の認識を使用可能にし、使用するスレッド・ローカル・ストレージ・モデルを指定する。
391 ページの『-qvrsave』	#pragma altivec_vrsave	VRSAVE レジスターを保守するための関数のプロローグとエピローグのコードを使用可能にする。
398 ページの『-qxcall』	なし。	コンパイル内の静的関数を外部関数であるかのように扱うためのコードを生成する。

エラー・チェックおよびデバッグ

このカテゴリーのオプションを使用すると、ご使用のソース・コードで問題を検出して訂正することができます。場合によっては、これらのオプションを使用することでオブジェクト・コードが変更されたり、コンパイル時間が長くなったり、アプリケーションの実行速度を落とすことがあるランタイム検査を導入することがあります。オプションの説明には、余分な検査によってどれくらいの影響をパフォーマンスが受ける可能性があるかが示されています。

ご使用のアプリケーションの振る舞いおよびパフォーマンスに関して受け取る情報の量とタイプを制御するには、 92 ページの『リスト、メッセージ、およびコンパイラー情報』のオプションを参照してください。

最適化されたコードのデバッグについて詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」を参照してください。

表 16. エラー・チェックおよびデバッグ・オプション

オプション名	同等のプラグマ名	説明
103 ページの『-# (ポンド記号)』	なし。	実際にコンパイラー・コンポーネントを呼び出さずに、コマンド行に指定されたコンパイルのステップをプレビューする。
129 ページの『-qcheck』	#pragma options check	特定タイプのランタイム検査を実行するコードを生成する。
144 ページの『-qdbgfmt』	なし	オブジェクト・ファイル内のデバッグ情報のフォーマットを指定する。
165 ページの『-qflttrap』	#pragma options flttrap	実行時に検出する浮動小数点例外のタイプを決定する。
168 ページの『-qformat』	なし。	ストリング入力および出力フォーマットの指定で考えられる問題について警告する。
169 ページの『-qfullpath』	#pragma options fullpath	このオプションは、**-g** オプションまたは **-qlinedebug** オプションと使用した場合、デバッグ情報付きでコンパイルされたオブジェクト・ファイルのソース・ファイルおよび組み込みファイルのフルパス名または絶対パス名を記録して、デバッグ・ツールが正確にソース・ファイルの場所を検索できるようにする。
170 ページの『-qfunctrace』	なし。	コンパイル単位で指定された関数の入り口点および出口点をトレースするトレース・ルーチンを呼び出す。
173 ページの『-g』	なし。	シンボリック・デバッガーが使用するデバッグ情報を生成し、選択したソース・ロケーションでのデバッグ・セッションでプログラムの状態を使用できるようにします。
179 ページの『-qhalt』	#pragma options halt	コンパイル時のメッセージの最大重大度が指定した重大度と同じかそれを超える場合は、オブジェクト・ソース・ファイル、実行可能ソース・ファイル、またはアセンブラー・ソース・ファイルのいずれかを作成する前に、コンパイルを停止する。

表 16. エラー・チェックおよびデバッグ・オプション (続き)

オプション名	同等のプラグマ名	説明
181 ページの『-qhaltonmsg』	なし。	指定されたエラー・メッセージが生成された場合、オブジェクト・ファイル、実行可能ファイル、またはアセンブラー・ソース・ファイルを作成せずにコンパイルを停止する。
193 ページの『-qinfo』	#pragma options info、#pragma info	通知メッセージのグループを作成または抑制する。
204 ページの『-qinitauto』	#pragma options initauto	デバッグのために、未初期化の自動変数を特定の値に初期化する。
221 ページの『-qkeepparm』	なし。	**-O2** 以上の最適化で使用した場合に、プロシージャー・パラメーターをスタックに保管するかどうかを指定します。
260 ページの『-qlinedebug』	なし。	デバッガー用に行番号およびソース・ファイル名の情報のみを生成する。
271 ページの『-qmaxerr』	なし。	指定した重大度レベル以上のエラー・メッセージの数が指定した数に到達すると、コンパイルを停止する。
284 ページの『-qoptdebug』	なし。	高水準の最適化で使用されると、デバッガーが読み取ることができる最適化済みの疑似コードを含むファイルを作成する。
359 ページの『-qsymtab (C のみ)』	なし。	シンボル・テーブルに表示する情報を決定する。
361 ページの『-qsyntaxonly (C のみ)』	なし。	オブジェクト・ファイルを生成せずに構文検査を実行する。
395 ページの『-qwarn0x (C++11)』	なし。	C++98 標準から C++11 標準へのマイグレーションによって発生したプログラム内の相違点について、ユーザーにメッセージで通知するかどうかを制御する。
397 ページの『-qwarn64』	なし。	32 ビットと 64 ビットのコンパイラー・モード間で発生する可能性のあるデータ変換問題の検査を使用可能にする。

リスト、メッセージ、およびコンパイラー情報

このカテゴリーのオプションを使用すると、コンパイラー・メッセージをいつ、どのように表示するかを制御できるのみでなく、リスト・ファイルをも制御することができます。以下のオプションを 89 ページの『エラー・チェックおよびデバッグ』で説明されているオプションと一緒に使用して、エラーおよび予期しない振る舞いを検査するときに提供される、ご使用のアプリケーションに関する概要をより堅固なものにすることができます。

表 17. リスト・オプションとメッセージ・オプション

オプション名	同等のプラグマ名	説明
120 ページの『-qattr』	#pragma options attr	リストの属性および相互参照セクションの属性コンポーネントを組み込むコンパイラー・リストを作成する。
148 ページの『-qdump_class_hierarchy (C++ のみ)』	なし。	それぞれのクラス・オブジェクトの階層と仮想関数テーブルのレイアウトの表記をファイルにダンプする。
158 ページの『-qflag』	#pragma options flag、C++ 448 ページの『#pragma report (C++ のみ)』	診断メッセージを指定した重大度レベルまたはそれより高い重大度レベルのものに制限する。
183 ページの『-qhelp』	なし。	コンパイラーの man ページを表示する。
261 ページの『-qlist』	#pragma options list	オブジェクト・リストを含むコンパイラー・リスト・ファイルを生成する。
262 ページの『-qlistfmt』	なし。	最適化の機会の検出を支援するために、XML または HTML レポートを作成する。
266 ページの『-qlistopt』	なし。	コンパイラー呼び出し時に有効なすべてのオプションを含むコンパイラー・リスト・ファイルを作成する。
301 ページの『-qphsinfo』	なし。	各コンパイル・フェーズで標準出力にかかった時間を報告する。
308 ページの『-qprint』	なし。	リストを使用可能にするか、または抑制する。
315 ページの『-qreport』	なし。	コードのセクションをどのように最適化したかを示すリスト・ファイルを作成する。

表 17. リスト・オプションとメッセージ・オプション (続き)

オプション名	同等のプラグマ名	説明
328 ページの『-qshowinc』	#pragma options showinc	**-qsource** オプションと使用してリスト・ファイルを生成すると、リスト・ファイルのソース・セクションにユーザーまたはシステムのヘッダー・ファイルを選択的に示す。
333 ページの『-qskipsrc』	なし。	**-qsource** オプションを使用してリスト・ファイルを生成する場合、**-qskipsrc** を使用して、コンパイラーによってスキップされたソース・ステートメントをリスト・ファイルのソース・セクションに表示するかどうかを決定できる。あるいは、**-qskipsrc=hide** オプションを使用して、コンパイラーによってスキップされたソース・ステートメントを隠蔽します。
340 ページの『-qsource』	#pragma options source	リストのソース・セクションを含むコンパイラー・リスト・ファイルを作成し、エラー・メッセージの印刷時にソース情報を追加して提供する。
344 ページの『-qsrcmsg (C のみ)』	なし。	対応するソース・コード行をコンパイラーが生成した診断メッセージに追加する。
357 ページの『-qsuppress』	なし。	特定の通知メッセージ、または警告メッセージが生成された場合、表示されたりリスト・ファイルに追加されるのを防ぐ。
387 ページの『-v, -V』	なし。	呼び出されているプログラムと各プログラムに指定されているオプションの名前を戻すことによって、コンパイルの進行を報告する。
387 ページの『-qversion』	なし。	呼び出しているコンパイラーのバージョンおよびリリースを表示する。
393 ページの『-w』	なし。	通知メッセージ、言語レベル・メッセージ、および警告メッセージを抑制する。

表 17. リスト・オプションとメッセージ・オプション (続き)

オプション名	同等のプラグマ名	説明
399 ページの『-qxref』	#pragma options xref	リストの属性および相互参照の相互参照コンポーネントを含むコンパイラー・リストを作成する。

最適化およびチューニング

このカテゴリーのオプションを使用すると、最適化およびチューニング・プロセスを制御することができます。これによって、実行時にアプリケーションのパフォーマンスを向上させることができます。

すべてのオプションがすべてのアプリケーションに利益を及ぼすわけではありません。トレードオフは、コンパイル時間の増加、デバッグ機能の低下、最適化によって提供される向上との兼ね合いの中で発生することがあります。

Optimize、**-qcompact**、または **-qstrict** などのこれらのオプションの一部を、**option_override** プラグマによって制御することもできます。

最適化とチューニング・プロセスおよび最適化に適したソース・コードの作成について詳しくは、このセクションのオプションの説明の他に、「*XL C/C++ 最適化およびプログラミング・ガイド*」を参照してください。

表 18. 最適化オプションおよびチューニング・オプション

オプション名	同等のプラグマ名	説明
106 ページの『-qaggrcopy』	なし。	構造体および共用体の破壊コピー操作を使用可能にする。
106 ページの『-qalias』	なし。	プログラムは、特定のカテゴリーの別名割り当てを含んでいるか、または C/C++ 標準別名割り当て規則に準拠していないかどうかを示す。複数の異なる名前が同じストレージ・ロケーションの別名となっている可能性がある場合、コンパイラーは最適化の一部のスコープを制限します。.
113 ページの『-qarch』	なし。	コード (命令) の生成対象の汎用プロセッサー・アーキテクチャーを指定する。
125 ページの『-qcache』	なし。	特定の実行マシンに対して、キャッシュ構成を指定する。
134 ページの『-qcompact』	#pragma options compact	コード・サイズを増やす最適化を回避する。

表 18. 最適化オプションおよびチューニング・オプション (続き)

オプション名	同等のプラグマ名	説明
142 ページの『-qdataimported、-qdatalocal、-qtocdata』	なし。	64 ビット・コンパイルでローカルまたはインポートとしてデータにマークを付ける。
147 ページの『-qdirectstorage』	なし。	特定のコンパイル単位がライトスルーを使用可能にしたストレージまたはキャッシュ禁止ストレージを参照する可能性があることをコンパイラーに通知する。
157 ページの『-qfdpr』	なし。	オブジェクト・ファイルに IBM Feedback Directed Program Restructuring (FDPR®) パフォーマンス・チューニング・ユーティリティーが結果の実行可能ファイルを最適化するために必要とする情報を提供する。
183 ページの『-qhot』	#pragma nosimd、 #pragma novector	最適化中に上位ループ分析および変換 (HOT) を実行する。
189 ページの『-qignerrno』	#pragma options ignerrno	システム呼び出しが `errno` を変更しないかのようにコンパイラーに最適化の実行を許可する。
211 ページの『-qipa』	なし。	プロシージャー間分析 (IPA) と呼ばれる最適化のクラスを使用可能にしたり、カスタマイズする。
217 ページの『-qisolated_call』	#pragma options isolated_call、#pragma isolated_call	パラメーターに暗黙指定されるもの以外の副次作用がない関数をソース・ファイルで指定する。
258 ページの『-qlibansi』	#pragma options libansi	ANSI C ライブラリー関数の名前が付いたすべての関数が実際はシステム関数であると見なす。
259 ページの『-qlibmpi』	なし。	Message Passing Interface (MPI) 名を持つすべての関数が事実上 MPI 関数であり、異なるセマンティクスを持つユーザー関数でないことを表明する。
272 ページの『-qmaxmem』	#pragma options maxmem	メモリーを消費する、指定したキロバイト数になるまでの特定の最適化の実行中に、コンパイラーによって割り振られるメモリーの量を制限する。

表 18. 最適化オプションおよびチューニング・オプション (続き)

オプション名	同等のプラグマ名	説明
276 ページの『-qminimaltoc』	なし。	64 ビット・コンパイル・モードで実行可能ファイルのためにコンパイラーが作成する目次 (TOC) の生成を制御する。
279 ページの『-O、-qoptimize』	#pragma options optimize	コンパイル中にコードを最適化するかどうかを指定し、最適化する場合は、レベルを指定する。
287 ページの『-p、-pg、-qprofile』	なし。	コンパイラーが作成するオブジェクト・ファイルをプロファイル用に準備する。
292 ページの『-qpdf1、-qpdf2』	なし。	*profile-directed feedback* (PDF) を介して最適化を調整する。サンプル・プログラムの実行から得られた結果を使用して、条件付き分岐の付近や頻繁に実行されるコード・セクションでの最適化を改善します。
305 ページの『-qprefetch』	なし。	コード・パフォーマンスを改善する機会がある場所に、プリフェッチ命令を自動的に挿入する。
310 ページの『-qprocimported、-qproclocal、-qprocunknown』	#pragma options procimported、 #pragma options proclocal、 #pragma options procunkown	64 ビットのコンパイルで、関数にローカル、インポート、または不明のマークを付ける。
207 ページの『-qinline』	なし。	パフォーマンスを改善するために、関数への呼び出しを生成する代わりに、それらの関数のインライン化を試行する。
319 ページの『-qrestrict (C のみ)』	なし。	このオプションを指定することは、指定された関数内でポインター・パラメーターに `restrict` キーワードを追加することと同等ですが、ソース・ファイルを変更する必要がない点が異なる。
330 ページの『-qshowpdf』	なし。	コンパイル・ステップとリンク・ステップで **-qpdf1** および最小最適化レベル **-O2** と共に使用すると、アプリケーション内のすべてのプロシージャーについて、追加プロファイル情報を含む PDF マップ・ファイルを作成する。

表 18. 最適化オプションおよびチューニング・オプション (続き)

オプション名	同等のプラグマ名	説明
331 ページの『-qsimd』	#pragma nosimd	ベクトル命令をサポートするプロセッサーでコンパイラーがベクトル命令を自動的に利用できるかどうかを制御する。
334 ページの『-qsmallstack』	なし。	スタック・フレームのサイズを削減する。
335 ページの『-qsmp』	なし。	プログラム・コードの並列化を使用可能にする。
352 ページの『-qstrict』	#pragma options strict	デフォルトの最適化レベル **-O3** 以上、およびオプションの **-O2** で実行された最適化が、プログラムのセマンティクスを変更しないことを確認する。
357 ページの『-qstrict_induction』	なし。	コンパイラーが帰納 (ループ・カウンター) 変数の最適化を実行しないようにする。これらの最適化は、帰納変数を含んだ整数オーバーフロー操作がある場合は安全ではない (ご使用のプログラムのセマンティクスを変更する) 可能性があります。
377 ページの『-qtune』	#pragma options tune	特定のハードウェア・アーキテクチャーで最も効率よく実行できるように、命令選択、スケジューリング、およびその他のアーキテクチャー依存のパフォーマンスの強化をチューニングする。ターゲット SMT モードの指定で、そのモードで最高のパフォーマンスが得られる最適化を指示できるようにする。
381 ページの『-qunroll』	#pragma options unroll、#pragma unroll	パフォーマンスを改善するために、ループのアンロールを制御する。
384 ページの『-qunwind』	なし。	スタックに保管されたレジスターを検索するコードによって呼び出しスタックをアンワインドできるかどうかを指定する。

表 18. 最適化オプションおよびチューニング・オプション (続き)

オプション名	同等のプラグマ名	説明
389 ページの『-qvisibility』	#pragma GCC visibility push、#pragma GCC visibility pop	オブジェクト・ファイル内の外部リンケージ・エンティティーに visibility 属性を指定する。外部リンケージ・エンティティーがプラグマ・ディレクティブ、明示的に指定された属性、または伝搬規則から visibility 属性を取得しない場合、それらのエンティティーは、**-qvisibility** オプションによって指定された visibility 属性を持ちます。

リンク

リンクは自動的に発生しますが、このカテゴリーでのオプションを使用すると、入力と出力をリンカーに送信して、リンカーによるオブジェクト・ファイルの処理方法を制御することができます。

表 19. リンク・オプション

オプション名	同等のプラグマ名	説明
138 ページの『-qcrt』	なし。	システム・スタートアップ・ファイルをリンクするかどうかを指定する。
149 ページの『-e』	なし。	**-qmkshrobj** と一緒に使用された場合、共有オブジェクトのエントリー・ポイントを指定する。
226 ページの『-L』	なし。	**-l** オプションによって指定されたライブラリー・ファイルのディレクトリー・パスをリンク時に検索する。
225 ページの『-l』	なし。	指定されたライブラリー・ファイル lib*key*.so があるかどうかを検索してから、動的リンクの場合は lib *key*.a、静的リンクの場合は単に lib*key*.a のみがあるかどうかを検索する。
257 ページの『-qlib』	なし。	標準システム・ライブラリーおよび XL C/C++ ライブラリーがリンクされるかどうかを指定する。
314 ページの『-R』	なし。	プログラムの実行時に要求された共有ライブラリーがこれらのディレクトリーで検索されるように、リンク時に、共有ライブラリーの検索パスを実行可能ファイルに書き込む。

表 19. リンク・オプション (続き)

オプション名	同等のプラグマ名	説明
347 ページの『-qstaticlink』	なし。	静的または共有のランタイム・ライブラリーをアプリケーションにリンクするかどうかを制御する。

移植性とマイグレーション

このカテゴリーのオプションは、過去、現在、そして未来のハードウェア、オペレーティング・システム、およびコンパイラーでのアプリケーション動作互換性を保守し、最小の変更でご使用のアプリケーションを XL コンパイラーに移行するのに役立ちます。

表 20. 移植性とマイグレーション・オプション

オプション名	同等のプラグマ名	説明
105 ページの『-qabi_version (C++ のみ)』	なし。	コンパイル中に使用される C++ アプリケーション・バイナリー・インターフェース (ABI) バージョンのバージョンを指定する。このオプションは、異なるレベルの GNU C++ との互換性のために提供されています。
109 ページの『-qalign』	#pragma options align、#pragma align	ストレージ内でデータ・オブジェクトの位置合わせを指定する。これによって、誤って配置されたデータが原因で起こるパフォーマンス上の問題を回避できます。
178 ページの『-qgenproto (C のみ)』	なし。	K&R 関数定義または空の括弧付きの関数定義からプロトタイプ宣言を作成し、それらを標準出力に表示する。
289 ページの『-qpack_semantic』	なし。	**#pragma pack** ディレクティブの構文およびセマンティクスを制御する。
385 ページの『-qupconv (C のみ)』	#pragma options upconv	整数拡張が実行されるときに符号なしの指定が保存されるかどうかを指定する。

コンパイラーのカスタマイズ

このカテゴリーのオプションを使用すると、コンパイラー・コンポーネント、構成ファイル、標準 include ディレクトリー、および内部コンパイラー操作のための代替の場所を指定することができます。これらのオプションは、特殊なインストール済み環境、テスト・シナリオ、および追加コマンド行オプションの指定で有用です。

表 21. コンパイラー・カスタマイズ・オプション

オプション名	同等のプラグマ名	説明
118 ページの『-qasm_as』	なし。	asm アセンブリー・ステートメントでアセンブラー・コードを処理するために、アセンブラーの呼び出しに使用されるパスとフラグを指定する。
121 ページの『-B』	なし。	アセンブラー、C プリプロセッサー、およびリンカーなどの XL C/C++ コンポーネントの代替パス名を指定します。
135 ページの『-qcomplexgccincl』	#pragma complexgcc	選択した組み込みファイルのみに対して、複素数データ型に GCC パラメーター受け渡し規則を使用するかどうかを指定します (**-qfloat=complexgcc** を使用可能にすることと同等です)。
139 ページの『-qc_stdinc (C のみ)』	なし。	XL C ヘッダー・ファイルの標準検索ロケーションを変更する。
140 ページの『-qcpp_stdinc (C++ のみ)』	なし。	XL C++ ヘッダー・ファイルの標準検索ロケーションを変更する。
156 ページの『-F』	なし。	コンパイラーの代替構成ファイルまたはスタンザを指定する。
176 ページの『-qgcc_c_stdinc (C のみ)』	なし。	GNU C システム・ヘッダー・ファイルの標準検索ロケーションを変更する。
177 ページの『-qgcc_cpp_stdinc (C++ のみ)』	なし。	GNU C++ システム・ヘッダー・ファイルの標準検索ロケーションを変更する。
291 ページの『-qpath』	なし。	コンパイラー、アセンブラー、リンカー、およびプリプロセッサーなどの XL C/C++ コンポーネントの代替パス名を指定する。
285 ページの『-qoptfile』	なし。	コンパイルで使用する追加コマンド行オプションのリストが入ったファイルを指定する。
343 ページの『-qspill』	#pragma options spill	レジスター・スピル・スペース (溢れたレジスターをストレージに退避させるために最適化プログラムが使用する内部プログラム・ストレージ域) のサイズをバイト単位で指定する。

表 21. コンパイラー・カスタマイズ・オプション (続き)

オプション名	同等のプラグマ名	説明
362 ページの『-t』	なし。	**-B** オプションで指定したプレフィックスを、指定したコンポーネントに適用する。
394 ページの『-W』	なし。	リストされたオプションをコンパイル中に実行されるコンポーネントに渡す。

推奨されないオプション

コンパイラーは、以下の表にリストされたオプションも受け入れています。アスタリスクが付いていないオプションは、同じ機能を提供する他のオプションまたは環境変数に置き換えられています。アスタリスクが付いているオプションは、廃止されている、または予期しない結果を引き起こす可能性があり、以前に資料で説明されたように実行される保証はありません。慎重に使用してください。

表 22. 推奨されないオプション

オプション名	置き換えオプション
-Q	-qinline
-qarch = ppc | ppc64 | ppcgr | ppc64gr | ppc64grsq	-qarch=pwr5
-qbigdata*	
-qenablevmx	-qsimd
-qhot=simd | nosimd	-qsimd
-qinfo=private	-qreport
-qinfo=reduction	-qreport
-qipa=clonearch | noclonearch	-qtune
-qipa=cloneproc | nocloneproc	-qtune
-qipa=inline | noinline	-qipa -qinline | -qipa -qnoinline
-qipa=pdfname	-qpdf1=pdfname, -qpdf2=pdfname
-qlanglvl=[no]gnu_externtemplate	-qlanglvl=[no]externtemplate
-qsmp= nested_par | nonested_par	36 ページの『OMP_NESTED』 環境変数または 659 ページの『omp_set_nested』 関数

個々のオプションの説明

この節では、XL C/C++ で使用可能な個々のコンパイラー・オプションについて説明します。

オプションにはそれぞれ、以下の情報が提供されています。

カテゴリー

オプションが属する機能カテゴリーはここに一覧表示されます。

プラグマ同等物
多くのコンパイラー・オプションでは、同等のプラグマ・ディレクティブを使用してオプションの機能をソース・コード内に適用し、オプションの適用範囲を単一のソース・ファイルまたはコードの選択したセクションに制限することができます。このことは、オプションによってディレクティブの **#pragma options** *option_name* または **#pragma** ***name*** 形式がサポートされている場所で指示されています。

目的 このセクションでは、オプション (およびそれと同等のプラグマ) の効果とその使用目的が簡単に説明されています。

構文 このセクションでは、オプションの構文が提供されており、同等の **#pragma** ***name*** がサポートされている箇所では、プラグマの特定の構文が提供されています。プラグマの **#pragma options** *option_name* 形式の構文は提供されていません。これは、通常オプションの構文と同じであるためです。任意のプラグマの C99 スタイル `_Pragma` 演算子形式も使用することができます。ただし、この構文はオプションの説明には記載されていません。プラグマ構文について詳しくは、403 ページの『プラグマ・ディレクティブ構文』を参照してください。

デフォルト
多くの場合、デフォルトのオプション設定は、構文図に明確に示されます。ただし、多くのオプションでは、効果のある他のコンパイラー・オプションに応じて、複数のデフォルト設定があります。このセクションでは、適用する場合のさまざまなデフォルト設定を示します。

パラメーター
このセクションでは、適用可能である場合の、オプションとプラグマの同等物に使用可能なサブオプションが説明されています。コマンド行オプションまたはプラグマ・ディレクティブに固有のサブオプションの場合、これは説明に示されています。

使用法 このセクションでは、オプションを使用するときに注意すべき規則および使用上の考慮事項が説明されています。オプションの適用可能性における制限、プラグマ・ディレクティブの有効な配置、複数のオプション指定の優先順位規則などが説明されています。

事前定義マクロ
多くのコンパイラー・オプションでは、保護されている (つまり、ユーザーによって定義解除または再定義できない) マクロが設定されます。適用可能な場合は、オプションによって事前定義されるマクロ、およびマクロが定義される値がこのセクションにリストされています。これらのマクロ (およびオプションの設定から独立して定義される他のマクロ) の参照リストは、481 ページの『第 6 章 コンパイラーの事前定義マクロ』にリストされています。

例 コマンド行構文およびプラグマ・ディレクティブの使用例がこのセクションの適切な箇所で提供されています。

-+ (正符号) (C++ のみ)

カテゴリー

入力制御

プラグマ同等物

なし。

目的

ファイルを C++ 言語ファイルとしてコンパイルする。

このオプションは **-qsourcetype=c++** オプションと同等です。

構文

使用法

-+ を使用して .a、.o、.so、.S または .s 以外のサフィックスを持つファイルをコンパイルすることができます。**-+** オプションを使用しない場合、ファイルを C++ ファイルとしてコンパイルするためには、.C (大文字の C)、.cc、.cp、.cpp、.cxx、または .c++ のサフィックスが必要です。 .c (小文字の c) のサフィックスの付いたファイルを **-+** を指定せずにコンパイルするとファイルは C 言語ファイルとしてコンパイルされます。

-+ オプションは、**-qsourcetype** オプションと一緒に使用しないでください。

事前定義マクロ

なし。

例

ファイル `myprogram.cplspls` を C++ ソース・ファイルとしてコンパイルするには、以下のように入力します。

```
xlc++ -+ myprogram.cplspls
```

関連情報

- 342 ページの『-qsourcetype』

-# (ポンド記号)

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

実際にコンパイラー・コンポーネントを呼び出さずに、コマンド行に指定されたコンパイルのステップをプレビューする。

このオプションが使用可能である場合、情報は標準出力に書き込まれ、呼び出されるプリプロセッサー、コンパイラー、およびリンカー内のプログラムの名前とそれぞれのプログラムに指定されるデフォルト・オプションが表示されます。プリプロセッサー、コンパイラー、およびリンカーは呼び出されません。

構文

使用法

このコマンドを使用して、特定のコンパイルで呼び出されるコマンドとファイルを判別することができます。これにより、ソース・コードのコンパイル、および既存のファイル (.lst ファイルなど) の上書きのオーバーヘッドが回避されます。

このオプションは **-v** と同じ情報を表示しますが、コンパイラーは呼び出しません。**-#** オプションは、**-v** オプションをオーバーライドします。

事前定義マクロ

なし。

例

ソース・ファイル `myprogram.c` のコンパイルのステップをプレビューするには、以下のように入力します。

```
xlc myprogram.c -#
```

関連情報

- 387 ページの『-v, -V』

-q32、-q64

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

32 ビットまたは 64 ビットのコンパイラー・モードを選択します。

-q32 および **-q64** オプションを、**-qarch** および **-qtune** コンパイラー・オプションと共に使用して、コンパイラーの出力が使用されるアーキテクチャーに合わせて、コンパイラーの出力を最適化します。

構文

デフォルト

-q32

事前定義マクロ

-q64 が有効な場合は、__64BIT__ が 1 に定義されます。それ以外の場合は未定義です。

例

myprogram.c からコンパイルされた実行可能プログラム testing を、64 ビットの Power アーキテクチャーのコンピューターで実行するよう指定するには、以下のように入力します。

```
xlc -o testing myprogram.c -q64 -qarch=ppc
```

関連情報

- アーキテクチャー固有のコンパイルでのコンパイラー・オプションの指定
- 113 ページの『-qarch』
- 377 ページの『-qtune』

-qabi_version (C++ のみ)

カテゴリー

移植性とマイグレーション

プラグマ同等物

なし。

目的

コンパイル中に使用される C++ アプリケーション・バイナリー・インターフェース (ABI) バージョンのバージョンを指定する。このオプションは、異なるレベルの GNU C++ との互換性のために提供されています。

構文

デフォルト

-qabi_version=2

パラメーター

1 GNU C++ 3.2 と同じ C++ ABI の振る舞いを指定します。

2 GNU C++ 3.4 と同じ C++ ABI の振る舞いを指定します。

事前定義マクロ

なし。

-qaggrcopy

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

構造体および共用体の破壊コピー操作を使用可能にする。

構文

デフォルト

-qaggrcopy=nooverlap

パラメーター

overlap | nooverlap

nooverlap は構造体および共用体のソースおよび宛先の割り当てがオーバーラップしないと見なすため、コンパイラーでより速いコードが生成されます。

overlap はこれらの最適化を禁止します。

事前定義マクロ

なし。

-qalias

カテゴリー

最適化およびチューニング

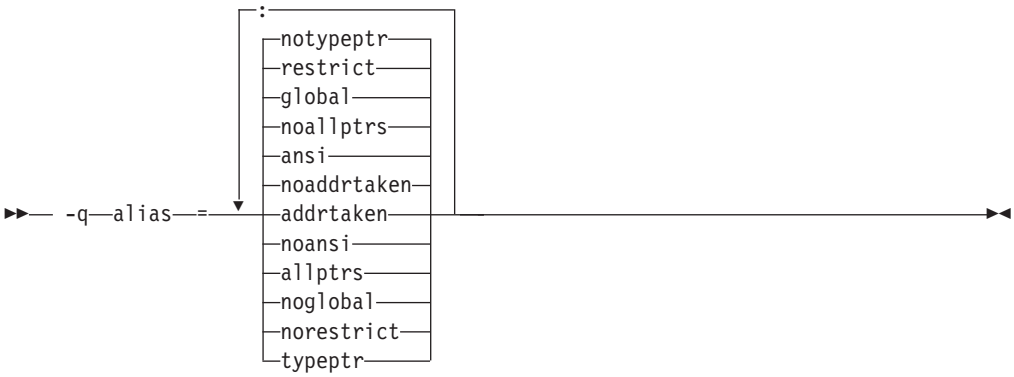
プラグマ同等物

なし

目的

プログラムは、特定のカテゴリーの別名割り当てを含んでいるか、または C/C++ 標準別名割り当て規則に準拠していないかどうかを示す。複数の異なる名前が同じストレージ・ロケーションの別名となっている可能性がある場合、コンパイラーは最適化の一部のスコープを制限します。

構文



デフォルト

- C++ -qalias=noaddrtaken:noallptrs:ansi:global:restrict:notypeptr
- C **cc** を除くすべての呼び出しコマンドでは **-qalias=noaddrtaken:noallptrs:ansi:global:restrict:notypeptr** です。**cc** 呼び出しコマンドでは **-qalias=noaddrtaken:noallptrs:noansi:global:restrict:notypeptr** です。

パラメーター

addrtaken | noaddrtaken

addrtaken が有効である場合、変数はアドレスが取得されない限り、ポインターから切り離されています。アドレスがコンパイル単位に記録されていない 変数のクラスは、ポインターを介した間接アクセスから切り離されていると見なされます。

noaddrtaken が指定されている場合、コンパイラーは有効な別名割り当て規則に従って別名割り当てを生成します。

allptrs | noallptrs

allptrs が有効な場合、ポインターに別名が割り当てられることはありません (これは **-qalias=typeptr** を暗黙指定します)。 **allptrs** を指定すると、2 つのポインターが同じ保管場所をポイントしないことをコンパイラーに表明することになります。これらのサブオプションが有効であるのは、**ansi** も有効である場合のみです。

ansi | noansi

ansi が有効である場合、最適化中に型ベースの別名割り当てが使用されます。これは、データ・オブジェクトへのアクセスに安全に使用できる左辺値に制限を加えます。このサブオプションは、最適化オプションを指定しない限り無効です。型ベースの別名割り当て規則に従っていない型に対しては、`may_alias` 属性を指定できます。

noansi が有効である場合、最適化プログラムはワーストケースの別名割り当てを想定します。これは、型に関係なく、特定の型のポインターが外部オブジェクト、またはアドレスが既に取得されているオブジェクトを指すことができると想定します。

global | noglobal

global が有効だと、コンパイル単位をまたがった IPA リンク時の最適化実行中に、型ベースの別名割り当て規則が使用可能になります。 **-qalias=global** を有効にするには、**-qipa** と **-qalias=ansi** の両方が有効になっている必要があります。 **noglobal** を指定すると、型ベースの別名割り当て規則が無効になります。

-qalias=global を使用すると、より高い最適化レベルにおけるパフォーマンスが向上し、またリンク時におけるパフォーマンスも向上します。 **-qalias=global** を使用した場合、サブオプションがパフォーマンスに与える効果が最大化されるように、できる限り同じバージョンのコンパイラーでアプリケーションをコンパイルすることをお勧めします。

restrict | norestrict

restrict が有効な場合、`restrict` キーワードで修飾されたポインターの最適化が使用可能です。**norestrict** を使用すると、`restrict` で修飾されたポインターの最適化が無効になります。

-qalias=restrict は、他の **-qalias** サブオプションから独立しています。**-qalias=restrict** オプションを使用すると通常、`restrict` で修飾されたポインターを使用するコードのパフォーマンスが向上します。ただし、**-qalias=restrict** では制限付きポインターを正しく使用する必要があります。そうでなければ、コンパイル時および実行時に障害が発生する可能性があります。 **norestrict** を使用すると、V9.0 より前のバージョンのコンパイラーでコンパイルされたコードとの互換性を維持できます。

typeptr | notypeptr

typeptr が有効である場合、異なる型へのポインターには別名は割り当てられません。 **typeptr** サブオプションが有効であるのは、**ansi** も有効である場合のみです。 **typeptr** は、**ansi** よりも制限度が高いです。 **typeptr** が有効である場合、ポインターは同じタイプまたは互換タイプのオブジェクトのみを指すことができ、`char*` の間接参照によって他の型に別名を付けることはできません。

C++ **-qalias=typeptr** を C++ 標準ライブラリーを含むプログラムで指定する場合、未定義の結果となる可能性があります。 C++

使用法

-qalias は、コンパイル中のコードに関するアサーションをコンパイラーに対して行います。コードに関するアサーションが FALSE の場合、アプリケーションの実行時に、コンパイラーによって生成されるコードが予測不能の振る舞いをする可能性があります。

以下は、型ベースの別名割り当ての対象となりません。

- signed 型および unsigned 型。例えば、`signed int` へのポインターは `unsigned int` を指すことができます。
- 文字ポインター型はどの型も指すことができます。

- volatile または const として限定された型。例えば、const int へのポインターは int を指すことができます。

事前定義マクロ

なし。

例

myprogram.c をコンパイルするときにワーストケースの別名割り当ての想定を指定するには、以下のように入力します。

```
xlc myprogram.c -O -qalias=noansi
```

関連情報

- 211 ページの『-qipa』
- -qinfo=als
- 415 ページの『#pragma disjoint』
- 「*XL C/C++ ランゲージ・リファレンス*」の『*型ベースの別名割り当て*』
- 「*XL C/C++ ランゲージ・リファレンス*」の『*may_alias 型属性 (IBM 拡張)*』
- 「*XL C/C++ ランゲージ・リファレンス*」の*restrict 型修飾子*
- 319 ページの『-qrestrict (C のみ)』

-qalign

カテゴリー

移植性とマイグレーション

プラグマ同等物

#pragma options align、#pragma align

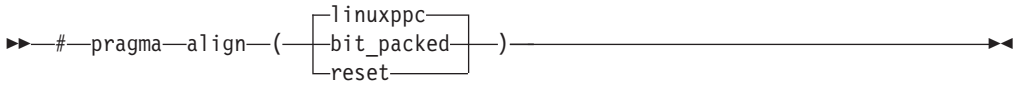
目的

ストレージ内でデータ・オブジェクトの位置合わせを指定する。これによって、誤って配置されたデータが原因で起こるパフォーマンス上の問題を回避できます。

構文

オプション構文

プラグマ構文



デフォルト

-qalign=linuxppc

パラメーター

bit_packed
ビット・フィールド・データは、バイト境界に関係なくビット単位に基づいてパックされます。

linuxppc
GNU C/C++ 位置合わせ規則を使用して、GNU C/C++ オブジェクトとのバイナリー互換性を保持します。

reset (プラグマのみ)
現行のプラグマ設定を破棄し、前のプラグマ・ディレクティブによって指定された設定に戻してください。前のプラグマ・ディレクティブを指定しないと、コマンド行またはデフォルト・オプションの設定に戻ります。

使用法

コマンド行で **-qalign** オプションを複数回使用した場合は、最後に指定した位置合わせ規則がファイルに適用されます。

プラグマ・ディレクティブは、プログラム・ソース・コードの指定されたセクションに対する **-qalign** コンパイラー・オプション設定をオーバーライドします。プラグマは、特定のプラグマ・ディレクティブの後に表示されるすべての集合体定義に影響を与えます。つまり、プラグマがネストされた集合体の中に格納される場合、そのプラグマはその後に続く定義に適用されるのみで、ネストに含まれる他の定義には適用されません。 宣言されたすべての集合体変数には、変数の宣言に先行するプラグマに関わらず、集合体が*定義* されたポイントで適用される位置合わせ規則が使用されます。以下の例を参照してください。

注: -qalign を使用した場合は、すべてのシステム・ヘッダーも **-qalign** でコンパイルされます。使用にあたっての考慮事項、およびオプションとプラグマ・パラメーターの詳細説明については、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『位置合わせデータ』を参照してください。

事前定義マクロ

なし。

例

以下の例は、オプションとプラグマの相互作用を示したものです。コマンド `xlcfile2.c` でコンパイルしたと想定し、以下の例では、プラグマがどのように 1 つの集合体の*定義* のみに影響を与え、それに続くその集合体型の変数の宣言には影響を与えないことを示します。

```
/* file2.c  The default alignment rule is in effect */

typedef struct A A2;

#pragma options align=bit_packed /*  bit_packed alignment rules are now in effect */
struct A {
```

```
int a;
char c;
}; #pragma options align=reset /*  Default alignment rules are in effect again */

struct A A1;  /*  A1 and A3 are aligned using bit_packed alignment rules since   */
A2 A3;        /*  this rule applied when struct A was defined      */
```

コマンド xlc file.c -qalign=bit_packed でコンパイルしたと想定し、以下の例では、ネストされた集合体定義に組み込まれたプラグマが、それに続く定義のみにどのように影響を与えるかを示します。

```
/* file2.c  The default alignment rule in effect is bit_packed  */

struct A {
int a;
#pragma options align=linuxppc /* Applies to B; A is unaffected  */
   struct B {
   char c;
   double d;
            } BB;   /*  BB uses linuxppc alignment rules  */
} AA;               /*  AA uses bit_packed alignment rules  /*
```

関連情報

- 440 ページの『#pragma pack』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『位置合わせデータ』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『__align 指定子』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『aligned 変数属性』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『packed 変数属性』

-qalloca、-ma (C のみ)

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma alloca

目的

alloca.h ヘッダーを含まないソース・コードから呼び出されるとき、システム関数 alloca のインライン定義を提供する。

関数 void* alloca(size_t *size*) は、標準ライブラリー関数の malloc のように、メモリーを動的に割り振ります。コンパイラーは以下いずれかの場合に、システム alloca 関数への呼び出しを、インライン組み込み関数 __alloca に自動的に置き換えます。

- ヘッダー・ファイル file alloca.h を組み込む場合
- **-Dalloca=__alloca** でコンパイルする場合
- フォーム __alloca を使用して組み込み関数を直接呼び出す場合

上記のいずれの方法も使用しない場合、**-qalloca** オプションと **-ma** オプション、および **#pragma alloca** ディレクティブが、C のみで同じ機能を果たします。

構文

オプション構文

プラグマ構文

デフォルト

適用されません。

使用法

alloca への呼び出しが __alloca に置き換えられたことを確認するために上記のいずれの方法も使用しない場合、alloca は組み込み関数ではなく、ユーザー定義の ID として処理されます。

#pragma alloca は一度指定されるとファイル全体に適用され、無効にできません。**#pragma alloca** を指定せずにコンパイルしたい関数がソース・ファイルに含まれている場合は、それらの関数を別のファイルに移してください。

alloca の代わりに C99 可変長配列を使用することもできます。

事前定義マクロ

なし。

例

関数 alloca への呼び出しがインラインとして扱われるように myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qalloca
```

関連情報

- 141 ページの『-D』
- 639 ページの『__alignx』

-qaltivec

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

Vector データ型およびオペレーターに対するコンパイラー・サポートを使用可能にする。

ベクトル・データ型に関する完全な資料については、「*XL C/C++ ランゲージ・リファレンス*」を参照してください。

構文

デフォルト

-qnoaltivec

使用法

このオプションは、**-qarch** がベクトル命令をサポートするアーキテクチャーになるように設定または暗黙指定されている場合にのみ有効です。それ以外の場合、コンパイラーは **-qaltivec** を無視し、警告メッセージを発行します。

事前定義マクロ

-qaltivec が有効な場合、__ALTIVEC__ は 1 に、__VEC__ は 10205 に定義されています。それ以外の場合は未定義です。

例

ベクトル・プログラミングのコンパイラー・サポートを使用可能にするには、以下のコマンドを入力します。

```
xlc myprogram.c -qarch=ppc64v -qaltivec
```

関連情報

- 『-qarch』
- 331 ページの『-qsimd』
- 「*AltiVec Technology Programming Interface Manual*」(http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手可能)

-qarch

カテゴリー

最適化およびチューニング

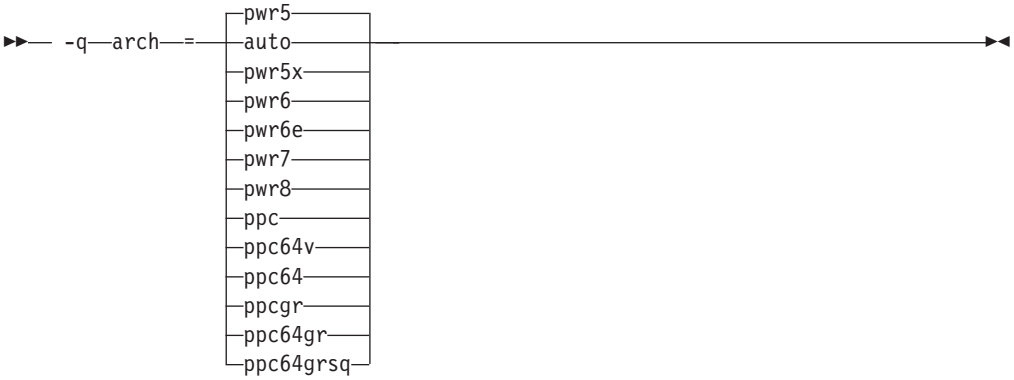
プラグマ同等物

なし。

目的

コード (命令) の生成対象の汎用プロセッサー・アーキテクチャーを指定する。

構文



デフォルト

- **-qarch=pwr5**。
- **-O4** または **-O5** が有効な場合は **-qarch=auto**。

パラメーター

auto
コンパイルを実行しているマシンの特定のアーキテクチャーを自動的に検出します。ランタイム環境はコンパイル環境と同じであると見なされます。 **-O4** オプションまたは **-O5** オプションが設定されている、または暗黙指定されている場合、このオプションは暗黙指定されます。

pwr5
POWER5、POWER5+、POWER6®、POWER7®、または POWER7+™、または POWER8™ 970 ハードウェア・プラットフォームで実行する命令を含んだオブジェクト・コードを作成します。

pwr5x
POWER5+、POWER6、POWER7、POWER7+、または POWER8 ハードウェア・プラットフォームで実行する命令を含んだオブジェクト・コードを作成します。

pwr6
POWER6、POWER7、POWER7+ 、または POWER8 ハードウェア・プラットフォーム POWER6、POWER7、POWER7+または POWER8 アーキテクチャー・モードで実行する命令を含んだオブジェクト・コードを作成します。

pwr6e
POWER6 エンハンスト・モードで実行する POWER6 ハードウェア・プラットフォームで実行する命令を含んだオブジェクト・コードを作成します。

pwr7
POWER7、POWER7+、または POWER8 ハードウェア・プラットフォームで実行する命令を含んだオブジェクト・コードを作成します。

pwr8
POWER8 ハードウェア・プラットフォームで実行する命令を含んだオブジェクト・コードを作成します。

ppc
これは推奨されないサブオプションです。まだ受け入れられますが、**-qarch=pwr5** にサイレント・アップグレードされます。

ppc64
これは推奨されないサブオプションです。まだ受け入れられますが、**-qarch=pwr5** にサイレント・アップグレードされます。

ppcgr
これは推奨されないサブオプションです。まだ受け入れられますが、**-qarch=pwr5** にサイレント・アップグレードされます。

ppc64gr
これは推奨されないサブオプションです。まだ受け入れられますが、**-qarch=pwr5** にサイレント・アップグレードされます。

ppc64grsq
これは推奨されないサブオプションです。まだ受け入れられますが、**-qarch=pwr5** にサイレント・アップグレードされます。

ppc64v
ベクトル・プロセッサー (POWER6 など) での汎用 PowerPC® チップ用の命令を生成します。32 ビットまたは 64 ビット・モードで有効です。

使用法

すべての PowerPC マシンは、共通の命令セットを共有しますが、指定されたプロセッサーまたはプロセッサー・ファミリーに固有の追加の命令を含むことも可能です。**-qarch** オプションを使用して、コンパイルに特定のアーキテクチャーをターゲットにすると、他のアーキテクチャーでは実行しないが、選択したアーキテクチャーでは最高のパフォーマンスを提供するコードが生成されます。特定のアーキテクチャーで最高のパフォーマンスを得たい場合、他のアーキテクチャーでプログラムを使用しないのであれば、適切なアーキテクチャー・オプションを使用してください。 複数のアーキテクチャーで実行できるコードを生成する場合は、アーキテクチャーのグループをサポートとする **-qarch** サブオプションを指定します。表 23 では、異なるプロセッサー・アーキテクチャーによってサポートされる機能およびそれらの代表的な **-qarch** サブオプションを示します。

表 23. プロセッサー・アーキテクチャーでのサポート機能

アーキテクチャー	グラフィックス・サポート	平方根サポート	64 ビット・サポート	ベクトル処理のサポート
pwr5	yes	yes	yes	no
pwr5x	yes	yes	yes	no
ppc	yes	yes	yes	no
ppc64	yes	yes	yes	no
ppc64gr	yes	yes	yes	no
ppc64grsq	yes	yes	yes	no
ppc64v	yes	yes	yes	VMX
pwr6	yes	yes	yes	VMX
pwr6e	yes	yes	yes	VMX
pwr7	yes	yes	yes	VMX、VSX
pwr8	yes	yes	yes	VMX、VSX

注: Vector Multimedia Extension (VMX) および Vector Scalar Extension (VSX) はベクトル処理用のプロセッサー命令です。

どの **-qarch** 設定についても、コンパイラーは特定のマッチングする **-qtune** 設定をデフォルトとして使用します。それによりさらにパフォーマンスが改善されます。また、**-qarch** をグループ引数と一緒に指定する場合、**-qtune** を **auto** として指定するか、グループに特定のアーキテクチャーを提供することができます。**-qarch** および **-qtune** を一緒に使用することについては、377 ページの『-qtune』を参照してください。

指定されたアプリケーション・プログラムに対しては、それぞれのソース・ファイルをコンパイルするときに必ず同じ **-qarch** 設定を指定してください。

事前定義マクロ

-qarch サブオプションによって事前定義されるマクロのリストについては、488 ページの『アーキテクチャー設定に関連したマクロ』を参照してください。

例

`myprogram.c` からコンパイルした実行可能プログラム `testing` を、VSX 命令サポートがあるコンピューター上で実行するように指定するには、以下のように入力します。

```
xlc -o testing myprogram.c  -qarch=pwr7
```

関連情報

- -qprefetch
- -qfloat
- 377 ページの『-qtune』
- 11 ページの『アーキテクチャー固有のコンパイルでのコンパイラー・オプションの指定』
- 104 ページの『-q32、-q64』
- 488 ページの『アーキテクチャー設定に関連したマクロ』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』

-qasm

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

コードの解釈およびその後のコードの生成を制御して、アセンブラー言語を拡張します。

-qasm が有効な場合、コンパイラーはソース・コードのアセンブリー・ステートメントに対してコードを生成します。サブオプションは、アセンブリー・ステートメントの内容の解釈に使用される構文を指定します。

注: このコマンドが効果を持つためには、システム・アセンブラー・プログラムが使用可能になっている必要があります。

構文

-qasm 構文 — C

-qasm 構文 — C++

デフォルト

-qasm=gcc

パラメーター

gcc
アセンブリー・ステートメントの拡張 GCC 構文とセマンティクスを認識するようにコンパイラーに命令します。

C++ **stdcpp**
今後使用するために予約済みです。

サブオプションなしで **-qasm** を指定した場合は、デフォルトを指定した場合と同等です。

使用法

C トークン `asm` は C 言語のキーワードではありません。したがって、言語レベル **stdc89** と **stdc99** (それぞれ C89 および C99 標準への厳密な準拠を強制する) では、アセンブリー・コードを生成するソースをコンパイルするためにオプション **-qkeyword=asm** も指定されている必要があります。他のすべての言語レベルでは、オプション **-qnokeyword=asm** が有効になっていない限り、トークン `asm` はキーワードとして扱われます。C では、コンパイラー特定のバリアント `__asm` と `__asm__` はすべての言語レベルでキーワードとなっており、使用不可にすることはできません。

C++ トークン asm、__asm、および __asm__ はすべての言語レベルでキーワードです。**-qnokeyword=*token*** のサブオプションは、これらのそれぞれの予約語を使用不可にするために使用することができます。

インラインの asm ステートメントの構文およびセマンティクスに関する詳細については、 「*XL C/C++* ランゲージ・リファレンス」の 『インライン・アセンブリー・ステートメント』を参照してください。

事前定義マクロ

- C **stdc89 | stdc99** を除くすべての言語レベルで asm がキーワードとして認識され、アセンブラー・コードが生成された場合、または **-qkeyword=asm** が有効である場合、および **-qasm[=gcc]** が有効である場合に、__IBM_GCC_ASM が 1 に事前定義されます。**stdc89 | stdc99** を除くすべての言語レベルで asm がキーワードとして認識されるが、アセンブラー・コードが生成されない場合、または **-qkeyword=asm** が有効である場合、および **-qnoasm** が有効である場合に、0 に事前定義されます。 **stdc89 | stdc99** 言語レベルまたは **-qnokeyword=asm** が有効な場合には未定義です。
- C++ すべての言語レベルで asm がキーワードとして認識され、アセンブラー・コードが生成され、かつ **-qasm[=gcc]** が有効である場合に、__IBM_GCC_ASM が 1 事前定義されます。 すべての言語レベルで asm がキーワードとして認識されるが、アセンブラー・コードが生成されず、**-qnoasm[=gcc]** が有効な場合に、0 が事前定義されます。**-qnoasm=stdcpp** が有効な場合には未定義です。**-qnoasm=stdcpp** が有効な場合は __IBM_STDCPP_ASM が 0 に定義されます。それ以外の場合は未定義です。

例

以下のコード・スニペットは、インライン・ステートメントの asm 構文向け GCC 規則の例です。

```
int a, b, c;
int main() {
    asm("add %0, %1, %2" : "=r"(a) : "r"(b), "r"(c) );
}
```

関連情報

- 『-qasm_as』
- 227 ページの『-qlanglvl』
- 222 ページの『-qkeyword』
- 「*XL C/C++* ランゲージ・リファレンス」の 『インライン・アセンブリー・ステートメント』
- 『言語拡張機能のキーワード』

-qasm_as

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

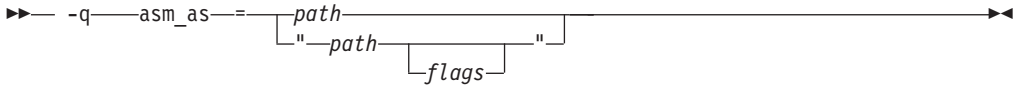
なし。

目的

asm アセンブリー・ステートメントでアセンブラー・コードを処理するために、アセンブラーの呼び出しに使用されるパスとフラグを指定する。

通常、コンパイラーはアセンブラーの場所を構成ファイルから読み取ります。このオプションを使用すると、代替アセンブラー・プログラムとそのアセンブラーに渡すためのフラグを指定できます。

構文



デフォルト

コンパイラーはデフォルトにより、コンパイラー構成ファイルの **as** コマンドに対して定義されたアセンブラー・プログラムを呼び出します。

パラメーター

path
使用されるアセンブラーの絶対パス名です。

flags
スペースで区切られた、アセンブリー・ステートメントの作成のためにアセンブラーに渡すオプションのリストです。 スペースが存在する場合は引用符を使用する必要があります。

事前定義マクロ

なし。

例

myprogram.c でインライン・アセンブラー・コードが検出されたときに、/bin/as にあるアセンブラー・プログラムを使用するようコンパイラーに命令するには、以下のように指定してください。

```
xlc myprogram.c -qasm_as=/bin/as
```

myprogram.c のインライン・アセンブラー・コードを処理するよう、/bin/as にあるアセンブラーに追加オプションを渡すようコンパイラーに命令するには、以下のように指定してください。

```
xlc myprogram.c -qasm_as="/bin/as -a64 -l a.lst"
```

関連情報

- 116 ページの『-qasm』

-qassert

カテゴリー

最適化およびチューニング

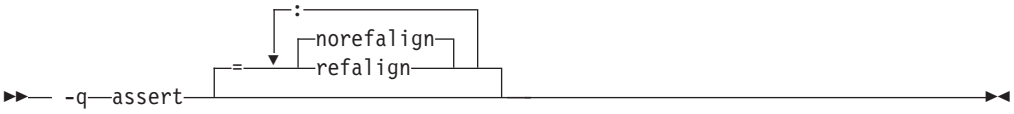
プラグマ同等物

なし

目的

最適化の微調整に役立つファイルの特性に関する情報を提供する。

構文



デフォルト

-qassert=norefalign

パラメーター

refalign | **norefalign**
コンパイル単位内のすべてのポインターが、ポインター型の長さに応じて自然に位置合わせされるデータのみを指すことを指定します。このアサーションを使用した場合、コンパイラーは、より効率的なコードを生成する可能性があります。このアサーションが特に有用なのは、SIMD アーキテクチャーをターゲットとして、**-qhot=level=0** または **-qhot=level=1** を **-qsimd=auto** と一緒に指定した場合です。

-qattr

カテゴリー

リスト、メッセージ、およびコンパイラー情報

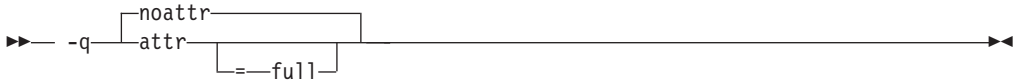
プラグマ同等物

C #pragma options [no]attr C

目的

リストの属性および相互参照セクションの属性コンポーネントを組み込むコンパイラー・リストを作成する。

構文



デフォルト

-qnoattr

パラメーター

full
プログラムにある ID をすべて報告します。サブオプションなしで **attr** を指定すると、使用された ID のみが報告されます。

使用法

-qattr は **-qattr=full** の後に指定すると無効になり、完全リストが作成されます。

このオプションは、**-qxref** も指定しない限り、相互参照リストを生成しません。

-qnoprint オプションは、このオプションをオーバーライドします。

注: -qattr を指定しても `#define` ディレクティブはリストされません。代わりに 329 ページの『-qshowmacros』 を使用できます。

事前定義マクロ

なし。

例

プログラム `myprogram.c` をコンパイルしてすべての ID のコンパイラー・リストを作成するには、以下のように入力します。

```
xlc myprogram.c -qxref -qattr=full
```

関連情報

- 329 ページの『-qshowmacros』
- 308 ページの『-qprint』
- 399 ページの『-qxref』

-B

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

なし。

目的

アセンブラー、C プリプロセッサー、およびリンカーなどの XL C/C++ コンポーネントの代替パス名を指定します。

XL C/C++ 実行可能ファイルの一部またはすべてについて複数のレベルを保持し、どれを使用するかを指定できるようにする場合、このオプションを使用することができます。しかし同じ機能として、代わりに **-qpath** オプションを使用することをお勧めします。

構文

デフォルト

コンパイラー実行可能ファイルのデフォルトのパスは、コンパイラー構成ファイルで定義されています。

パラメーター

prefix
-t オプションで指定できるプログラムのパス名の一部を定義します。スラッシュ (/) を追加しなければなりません。 *prefix* なしで **-B** オプションを指定すると、デフォルトのプレフィックスは /lib/o になります。

使用法

-t オプションは、**-B** プレフィックス名の追加先のプログラムを指定します。これらのリストについては、 362 ページの『-t』を参照してください。 **-t***programs* なしで **-B** オプションを使用すると、指定したプレフィックスがすべてのコンパイラー実行可能ファイルに適用されます。

-B および **-t** オプションは、**-F** オプションをオーバーライドします。

事前定義マクロ

なし。

例

この例では、前のレベルのコンパイラー・コンポーネントがデフォルトのインストール・ディレクトリーにインストールされています。アップグレードしたプロダクトを、全員に使用可能にする前にテストするには、システム管理者はディレクトリー /home/jim に最新のインストール・イメージを復元してから、以下のようなコマンドで試します。

```
xlc -tcbI -B/home/jim/opt/ibm/xlC/13.1.0/bin/ test_suite.c
```

アップグレードが許容基準を満たしていれば、システム管理者はそれをデフォルトのインストール・ディレクトリーにインストールします。

関連情報

- 291 ページの『-qpath』
- 362 ページの『-t』
- 1 ページの『コンパイラーの呼び出し』

-qbitfields

カテゴリー

浮動小数点および整数のコントロール

プラグマ同等物

なし。

目的

ビット・フィールドを符号付きにするか符号なしにするかを指定する。

構文

デフォルト

-qbitfields=signed

パラメーター

signed
ビット・フィールドは符号付きです。

unsigned
ビット・フィールドは符号なしです。

事前定義マクロ

なし。

-c

カテゴリー

出力制御

プラグマ同等物

なし。

目的

コンパイラーに、ソース・ファイルをコンパイルまたはアセンブルすることのみを指示し、リンクすることは指示しない。このオプションでは、出力は各ソース・ファイルの .o ファイルです。

構文

デフォルト

コンパイラーはデフォルトにより、リンカーを呼び出して、オブジェクト・ファイルを最終実行可能ファイルにリンクします。

使用法

このオプションが有効であれば、コンパイラーは *file_name*.c、*file_name*.i、*file_name*.C、*file_name*.cpp、または *file_name*.s などの有効なそれぞれのソース・ファイルに対して、出力オブジェクト・ファイル *file_name*.o を作成します。オブジェクト・ファイルの明示名を指定するには、**-o** オプションを使用できます。

-c オプションは、**-E**、**-P**、または **-qsyntaxonly** オプションが指定されている場合にオーバーライドされます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルしてオブジェクト・ファイル `myprogram.o` を作成するが、実行可能ファイルを作成しない場合は、以下のコマンドを入力します。

```
xlc myprogram.c -c
```

`myprogram.c` をコンパイルしてオブジェクト・ファイル `new.o` を作成するが、実行可能ファイルを作成しない場合は、以下のコマンドを入力します。

```
xlc myprogram.c -c -o new.o
```

関連情報

- 150 ページの『-E』
- 278 ページの『-o』
- 288 ページの『-P』
- 361 ページの『-qsyntaxonly (C のみ)』

-C、-C!

カテゴリー

出力制御

プラグマ同等物

なし。

目的

-E または **-P** オプションと使用すると、プリプロセスされた出力内のコメントを保存または除去する。

-C が有効な場合、コメントは保持されます。 **-C!** が有効な場合、コメントは除去されます。

構文

デフォルト

-C

使用法

-C オプションは、**-E** または **-P** オプションを指定しないと無効になります。 **-E** を指定した場合は、継続シーケンスが出力で保持されます。**-P** を指定した場合は、継続シーケンスが出力から除去され、連結された出力行が形成されます。

-C! オプションを使用すると、デフォルトの Make ファイルまたは構成ファイルで指定された **-C** オプションをオーバーライドできます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、コメントを含んだプリプロセスされたプログラム・テキストを含むファイル `myprogram.i` を生成するには、以下のように入力します。

```
xlc myprogram.c -P -C
```

関連情報

- 150 ページの『-E』
- 288 ページの『-P』

-qcache

カテゴリー

最適化およびチューニング

プラグマ同等物

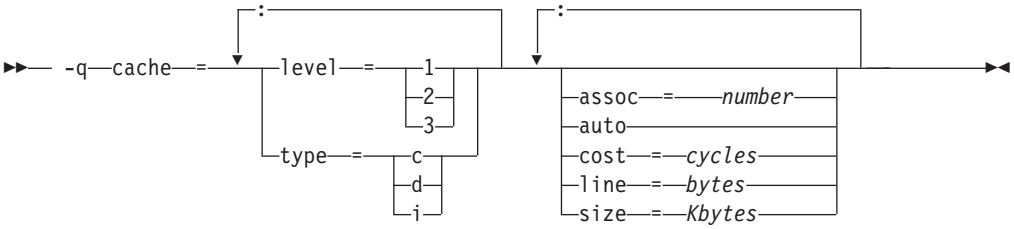
なし。

目的

特定の実行マシンに対して、キャッシュ構成を指定する。

プログラムの実行システムの型がわかっていて、システムにデフォルトのケースとは異なる構成の命令またはデータ・キャッシュがある場合は、このオプションを使用して、正確なキャッシュ特性を指定します。コンパイラーはこの情報を使用して、キャッシュ関連の最適化の利点を計算します。

構文



デフォルト

-qtune オプションの設定によって自動的に決定します。

パラメーター

assoc
キャッシュのセット結合順序を指定します。

number
これは以下のいずれかです。

0 直接マップされたキャッシュ

1 完全結合のキャッシュ

N>1 N-way 設定の結合キャッシュ

auto
コンパイル・マシンの特定のキャッシュ構成を自動的に検出します。これは、ランタイム環境がコンパイル環境と同じであることを前提としています。

cost
キャッシュ・ミスの結果生ずるパフォーマンス・ペナルティーを指定します。

cycles

level
影響を受けたキャッシュのレベルを指定します。 マシンに複数のレベルのキャッシュがある場合は、別々の **-qcache** オプションを使用します。

level
これは以下のいずれかです。

1 基本キャッシュ

2 レベル 2 のキャッシュか、レベル 2 のキャッシュがない場合は、テーブル・ルックアサイド・バッファー (TLB)

3 TLB

line
キャッシュの行サイズを指定します。

bytes
キャッシュ・ラインのバイト数を示す整数です。

size
キャッシュの合計サイズを指定します。

Kbytes
合計キャッシュのキロバイト数を示す整数です。

type
指定された *cache_type* に設定を適用することを指定します。

cache_type
これは以下のいずれかです。

c 結合されたデータおよび命令キャッシュ

d データ・キャッシュ

i 命令キャッシュ

使用法

-qtune 設定は、最も一般的なコンパイルに最適なデフォルト **-qcache** 設定を判別します。これらのデフォルト設定は、**-qcache** を使用してオーバーライドすることができます。しかし、間違った値をキャッシュ構成に指定したり、異なる構成のマシン上でプログラムを実行した場合、そのプログラムは正常に稼働しますが、若干遅くなる可能性があります。

-qcache サブオプションを指定するときには、以下のガイドラインを使用してください。

- できるだけ多くの構成パラメーターの情報を指定する。
- ターゲットの実行システムに複数のレベルのキャッシュがある場合は、別々の **-qcache** オプションを使用して各キャッシュ・レベルを記述する。
- ターゲットの実行マシン上のキャッシュの正確なサイズがわからない場合は、見積もったキャッシュ・サイズの小さい方を指定する。実際のキャッシュ・サイズより大きなサイズを指定したためにキャッシュの欠落やページ不在を経験するよりも、キャッシュ・メモリーをいくらか未使用で残しておくことをお勧めします。
- データ・キャッシュは、命令キャッシュよりもプログラム・パフォーマンスにより大きな影響を及ぼす。異なるキャッシュ構成を試してみる時間的余裕があまりない場合は、まずデータ・キャッシュに最適な構成指定を判別します。
- 間違った値をキャッシュ構成に指定したり、異なる構成のマシン上でプログラムを実行した場合は、プログラムのパフォーマンスが低下することがあるが、プログラム出力は予期通りとなる。
- **-O4** および **-O5** 最適化オプションは、コンパイル・マシンのキャッシュ特性を自動的に選択する。 **-qcache** オプションを **-O4** または **-O5** オプションと一緒に指定すると、最後に指定されたオプションが優先される。

- **-qcache=auto** が指定されていない限り、**-qcache** オプションを使用する場合、**type** サブオプションと **level** サブオプションの両方を指定する必要があります。そうしないと、警告メッセージが出されます。

事前定義マクロ

なし。

例

結合された命令とデータ・レベル 1 のキャッシュ (キャッシュは 2-way アソシエイティブで、サイズは 8 KB で、64 バイトのキャッシュ行を持っている) を使用してシステムのパフォーマンスを調整するには、以下のように入力します。

```
xlc -O4 -qcache=type=c:level=1:size=8:line=64:assoc=2 file.c
```

関連情報

- 125 ページの『-qcache』
- 279 ページの『-O、-qoptimize』
- 377 ページの『-qtune』
- 211 ページの『-qipa』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』

-qchars

カテゴリー

浮動小数点と整数の制御

プラグマ同等物

#pragma options chars、#pragma chars

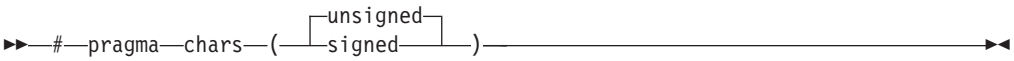
目的

char 型のすべての変数を符号付きまたは符号なしのいずれかとして処理するかを判別する。

構文

オプション構文

プラグマ構文



デフォルト

-qchars=unsigned

パラメーター

unsigned
char 型の変数は、unsigned char として処理されます。

signed
char 型の変数は、signed char として処理されます。

使用法

このオプションまたはプラグマの設定に関係なく、char 型は、型の互換性検査 または C++ 多重定義 の目的で、unsigned char 型および signed char 型とは異なると見なされます。

プラグマは、どのソース・ステートメントよりも前に指定されていなければなりません。 ソース・ファイルにプラグマが複数回指定されている場合は、1 番目のプラグマが優先されます。プラグマは、指定するとファイル全体に適用されて無効にできません。**#pragma chars** を指定せずにコンパイルする関数がソース・ファイルに含まれている場合は、それらの関数を別のファイルに移してください。

事前定義マクロ

- **signed** が有効な場合は、_CHAR_SIGNED および __CHAR_SIGNED__ が 1 に定義されます。それ以外の場合は未定義です。
- **unsigned** が有効な場合は、_CHAR_UNSIGNED および _CHAR_UNSIGNED__ が 1 に定義されます。それ以外の場合は未定義です。

例

myprogram.c をコンパイルするときに、すべての char 型を signed 型として扱うには、以下のように入力してください。

```
xlc myprogram.c -qchars=signed
```

-qcheck

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

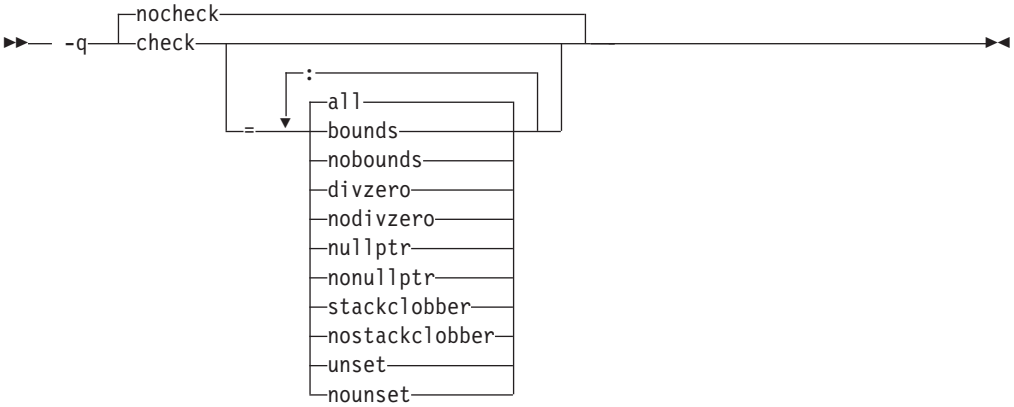
#pragma options [no]check

目的

特定タイプのランタイム検査を実行するコードを生成する。

違反が検出された場合は、SIGTRAP シグナルをプロセスに送信することによって実行時エラーを報告します。ランタイム検査により、アプリケーション実行の速度が落ちる場合もあります。

構文



デフォルト

-qnocheck

パラメーター

all
すべてのサブオプションを使用可能にします。

bounds | nobounds
サイズが既知のオブジェクト内の添え字を指定するときに、アドレスのランタイム検査を行います。指標が検査され、結果がオブジェクトのストレージの境界内のアドレスになることが確認されます。アドレスがオブジェクトの境界内にない場合は、トラップが起こります。

このサブオプションは可変長配列へのアクセスには無効です。

divzero | nodivzero
整数除算のランタイム検査を行います。 ゼロ除算を試行すると、トラップが発生します。

nullptr | nonullptr
ストレージの参照に使用するポインター変数に含まれるアドレスのランタイム検査を行います。 アドレスは、使用する位置で検査されます。値が 512 より小さい場合は、トラップが起こります。

stackclobber | nostackclobber
ユーザー・プログラムの保存域内の不揮発性レジスターのスタック破損を検出します。このタイプの破損は、スタックの保存域内の不揮発性レジスターが変更されている場合にのみ発生します。

-qstackprotect オプションとこのサブオプションの両方がオンの場合、このサブオプションが最初にスタック破損を検出します。

unset | nounset
使用する自動変数を、設定する前にチェックします。使用する前に自動変数が設定されていない場合は、トラップが実行時に行われます。

-qinitauto オプションは、自動変数を初期化します。その結果、**-qinitauto** オプションは、**-qcheck=unset** オプションから未初期化の変数を非表示にします。

-qcheck オプションをサブオプションなしで指定することは、**-qcheck=all** を指定することと同等です。

使用法

-qcheck オプションは、複数回指定することができます。サブオプションの設定は累積されますが、後の方サブオプションによって前の方サブオプションがオーバーライドされます。

all サブオプションは、フィルターとして他の 1 つ以上のオプションの **no...** 形式と共に使用することができます。例えば、以下を使用すると、

```
xlc myprogram.c -qcheck=all:nonullptr
```

ストレージの参照に使用するポインター変数に含まれるアドレスを除いて、すべての検査が行われます。**no...** 形式のサブオプションと共に **all** を使用する場合は、**all** は最初のサブオプションでなければなりません。

事前定義マクロ

なし。

例

以下のコードの例で、**-qcheck=nullptr : bounds** の影響を示します。

```
void func1(int* p) {
  *p = 42;             /* Traps if p is a null pointer */
}

void func2(int i) {
  int array[10];
  array[i] = 42;     /* Traps if i is outside range 0 - 9 */
}
```

以下のコードの例で、**-qcheck=divzero** の影響を示します。

```
void func3(int a, int b) {
  a / b;            /* Traps if b=0  */
}
```

以下のコードの例で、**-qcheck=stackclobber** の影響を示します。

```
void func4(char *p, int off, int value) {
  *(p+off)=value;
}

int foo() {
  int i;
  char boo[9];
  i=24;
  func4(boo, i, 66);
  /* Traps here */
  return 0;
}

int main() {
  foo();
}
```

注: 最適化レベルが異なると、オフセットは変わることがあります。-O2 以下の最適化レベルが有効な場合、func4 は foo の保存域を強制削除します。これは、*(p+off) が保存域内にあるためです。

関数 factorial で、result は n<=1 の場合は初期化されません。factorial.c 内で未初期化変数を検出するには、以下のコマンドを入力します。

```
xlc -g -O -qcheck=unset factorial.c
```

factorial.c には、以下のコードが含まれます。

```
int factorial(int n) {
  int result;

  if (n > 1) {
    result = n * factorial(n - 1);
  }

  return result; /* line 8 */
}

int main() {
  int x = factorial(1);
  return x;
}
```

コンパイラーはコンパイル時に以下の通知メッセージを発行し、トラップは実行時に行 8 で行われます。

```
1500-099: (I) "factorial.c", line 8: "result" might be used before it is set.
```

注: noopt で **-qcheck=unset** を設定すると、コンパイラーはコンパイル時に通知メッセージを発行しません。

-qcinc (C++ のみ)

カテゴリー

入力制御

プラグマ同等物

なし。

目的

extern "C" { } ラッパーを指定したディレクトリー内の組み込みファイルのコンテンツのまわりに配置する。

構文

デフォルト

-qnocinc

パラメーター

directory_path
extern "C" リンケージ指定子で折り返される組み込みファイルが格納されるディレクトリーです。

事前定義マクロ

なし。

例

アプリケーション myprogram.C にヘッダー・ファイル foo.h が組み込まれており、このファイルがディレクトリー /usr/tmp にあって、以下のコードが含まれていると想定します。

```
int foo();
```

以下のように入力してアプリケーションをコンパイルすると、

```
xlc++ myprogram.C -qcinc=/usr/tmp
```

以下のようにヘッダー・ファイル foo.h がアプリケーションに組み込まれます。

```
extern "C" {
int foo();
}
```

-qcommon

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

未初期化グローバル変数が割り振られる場所を制御する。

-qcommon が有効な場合、未初期化のグローバル変数はオブジェクト・ファイルの共通セクションに割り振られます。**-qnocommon** が有効な場合、未初期化のグローバル変数は 0 に初期設定され、オブジェクト・ファイルのデータ・セクションに割り振られます。

構文

デフォルト

- C **-qcommon** (**-qmkshrobj** が指定されている場合を除く)、**-qmkshrobj** が指定されている場合は **-qnocommon**。

- C++ **-qnocommon**

使用法

このオプションは、静的変数または自動変数、あるいは構造の宣言または共用体のメンバーに影響を与えません。

このオプションは、commonlnocommon および section 変数属性によってオーバーライドされます。 「*XL C/C++ ランゲージ・リファレンス*」の『common および nocommon 変数属性』、および『section 変数属性』を参照してください。

事前定義マクロ

なし。

例

以下の宣言では、a と b はグローバル変数です。

```
int a, b;
```

-qcommon でコンパイルを実行すると、以下のアセンブリー・コードと同等のものが作成されます。

```
.comm _a,4
.comm _b,4
```

-qnocommon でコンパイルを実行すると、以下のアセンブリー・コードと同等のものが作成されます。

```
        .globl _a
.data
.zerofill __DATA, __common, _a, 4, 2
        .globl _b
.data
.zerofill __DATA, __common, _b, 4, 2
```

関連情報

- 277 ページの『-qmkshrobj』
- 「*XL C/C++ ランゲージ・リファレンス*」の『common および nocommon 変数属性』
- 「*XL C/C++ ランゲージ・リファレンス*」の『section 変数属性』

-qcompact

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options [no]compact

目的

コード・サイズを増やす最適化を回避する。

構文

デフォルト

-qnocompact

使用法

コード・サイズは一般に、インライン化やループのアンロールなど、インラインのコードを複製または展開する最適化を禁止することによって縮小されます。実行時間は増加する可能性があります。

このオプションは、**-O2** またはそれ以上の最適化レベルで指定された場合にのみ有効です。

事前定義マクロ

-qcompact および最適化レベルが有効な場合は、__OPTIMIZE_SIZE__ が 1 に事前定義されます。それ以外の場合は定義が解除されます。

例

可能なときにコード・サイズを縮小するようコンパイラーに命令して、myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -O -qcompact
```

-qcomplexgccincl

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

なし。

目的

選択した組み込みファイルのみに対して、複素数データ型に GCC パラメーター受け渡し規則を使用するかどうかを指定します (**-qfloat=complexgcc** を使用可能にすることと同等です)。

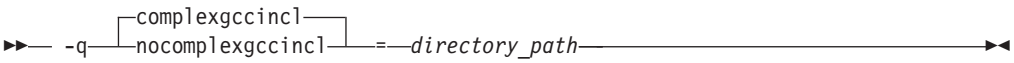
-qcomplexgccincl が有効な場合、コンパイラーは、指定されたディレクトリーに格納されるファイルのまわりの **#pragma complexgcc(on)** ディレクティブと **#pragma complexgcc(pop)** ディレクティブを内部でラップします。 **-qnocomplexgccincl** が有効な場合、指定されたディレクトリーで検出された組み込みファイルは、これらのディレクティブによってラップされません。

このプラグマ・ディレクティブを使用すると、選択されたファイルまたはコード・セクションの複素数データ型に対して GCC パラメーター引き渡し規則を使用可能

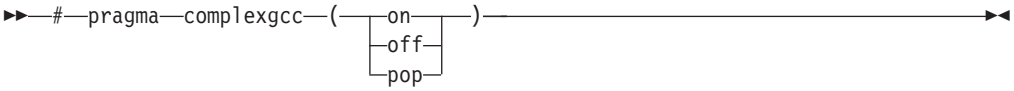
または使用不可に設定することもできます。

構文

オプション構文



プラグマ構文



デフォルト

デフォルトで、標準ディレクトリーで XL C/C++ および GCC ヘッダー・ファイルとして検出されるファイルは、**#pragma complexgcc** ディレクティブでラップされます。これらのリストについては、 15 ページの『組み込みファイルのディレクトリー検索シーケンス』を参照してください。

パラメーター

directory_path **(オプションのみ)**
: **#pragma complexgcc** ディレクティブでラップされる組み込みファイルを含むディレクトリーを検出できます。*directory_path* を指定しないと、コンパイラーは上記でリストしたデフォルトのディレクトリーを想定します。

on (プラグマのみ)
: 後に続くコードに対して **-qfloat=gccomplex** を設定します。これは、複合型パラメーターの受け渡しに対して、汎用レジスターによって GCC 規則を使用するようコンパイラーに命令します。

off (プラグマのみ)
: 後に続くコードに対して **-qfloat=nogccomplex** を設定します。これは、複合型パラメーターの受け渡しに対して、浮動小数点レジスターによって Linux 規則を使用するようにコンパイラーに命令します。

pop (プラグマのみ)
: 現行のプラグマ設定を破棄し、前のプラグマ・ディレクティブによって指定された設定に戻してください。前のプラグマ・ディレクティブを指定しないと、コマンド行またはデフォルト・オプションの設定に戻ります。

使用法

このプラグマの現行設定は、この設定が有効な間に宣言または定義された関数だけに影響します。その他の関数には影響しません。

関数に対するポインターから関数を呼び出すと、有効な **-qfloat=[no]complexgcc** コマンド行オプションによって設定された規則が常に使用されます。複合値を渡す関

数を値またはリターン複合値でミックス・アンド・マッチさせると、エラーが発生します。例えば以下のコードが **-qfloat=nocomplexgcc** でコンパイルされるとします。

```
#pragma complexgcc(on)
void p (_Complex double x) {}

#pragma complexgcc(pop)
typedef void (*fcnptr) (_Complex double);

int main() {
    fcnptr ptr = p; /* error: function pointer is -qfloat=nocomplexgcc;
                              function is -qfloat=complexgcc */
}
```

事前定義マクロ

なし。

関連情報

- 160 ページの『-qfloat』

-qcpluscmt (C のみ)

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

C ソース・ファイルでの C++ 形式のコメントの認識を使用可能にする。

構文

デフォルト

- **xlc** または**c99**、および関連した呼び出しが使用されている場合、あるいは **stdc99** | **extc99** 言語レベルが有効になっている場合は、**-qcpluscmt** です。
- その他のすべての呼び出しコマンドおよび言語レベルでは、**-qnocpluscmt** です。

事前定義マクロ

-qcpluscmt が有効な場合は、__C99_CPLUSCMT が 1 に事前定義されます。それ以外の場合は未定義です。

例

C++ のコメントがコメントとして認識されるように `myprogram.c` をコンパイルするには、次のように入力します。

```
xlc myprogram.c -qcpluscmt
```

// コメントは C89 の一部ではありません。以下の有効な C89 プログラムの結果は誤りになります。

```
main() {
  int i = 2;
  printf("%i¥n", i //* 2 */
                 + 1);
}
```

正しい答えは 2 (2 / 1) です。 **-qcpluscmt** が有効であると (これはデフォルト設定)、結果は 3 (2 + 1) になります。

関連情報

- 124 ページの『-C、-C!』
- 227 ページの『-qlanglvl』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『コメント』

-qcrt

カテゴリー

リンク

プラグマ同等物

なし。

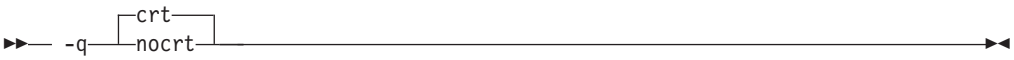
目的

システム・スタートアップ・ファイルをリンクするかどうかを指定する。

-qcrt が有効だと、システム開始のルーチンが自動的にリンクされます。**-qnocrt** が有効だと、リンク時にシステム・スタートアップ・ファイルは使用されません。**-l** フラグを使用してコマンド行で指定したファイルのみがリンクされます。

このオプションをシステム・プログラミングで使用すると、オペレーティング・システムが提供する開始ルーチンの自動リンクが無効になります。

構文



デフォルト

-qcrt

事前定義マクロ

なし。

関連情報

- 257 ページの『-qlib』

-qc_stdinc (C のみ)

カテゴリー

コンパイラーのカスタマイズ

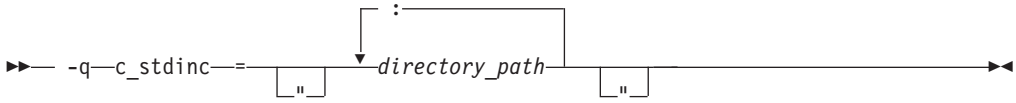
プラグマ同等物

なし。

目的

XL C ヘッダー・ファイルの標準検索ロケーションを変更する。

構文



デフォルト

デフォルトでコンパイラーは、構成ファイルで指定されたディレクトリーで XL C ヘッダー・ファイル (これは通常 /opt/ibm/xlC/13.1.0/include/) を検索します。

パラメーター

directory_path
コンパイラーが、XL C ヘッダー・ファイルを検索するディレクトリーのパスです。 *directory_path* は、相対パスまたは絶対パスにすることができます。 パスは、引用符で囲むとコマンド行で分割されることがありません。

使用法

このオプションを使用すると、特定コンパイルの検索パスを変更できます。 XL C ヘッダーのデフォルト検索パスを永久的に変更するには、構成ファイルを使用します。詳しくは、15 ページの『組み込みファイルのディレクトリー検索シーケンス』を参照してください。

このオプションが複数回指定されている場合、コンパイラーは最後に指定されたオプションのみを使用します。

このオプションは、**-qnostdinc** オプションが有効である場合には無視されます。

事前定義マクロ

なし。

例

`mypath/headers1` および `mypath/headers2` を使用して XL C ヘッダーのデフォルト検索パスをオーバーライドするには、以下を入力します。

```
xlc myprogram.c -qc_stdinc=mypath/headers1:mypath/headers2
```

関連情報

- 176 ページの『-qgcc_c_stdinc (C のみ)』
- 351 ページの『-qstdinc』
- 191 ページの『-qinclude』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』

-qcpp_stdinc (C++ のみ)

カテゴリー

コンパイラーのカスタマイズ

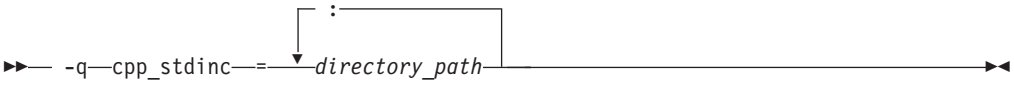
プラグマ同等物

なし。

目的

XL C++ ヘッダー・ファイルの標準検索ロケーションを変更する。

構文



デフォルト

デフォルトでコンパイラーは、構成ファイルで指定されたディレクトリーで XL C++ ヘッダー・ファイル (これは通常 /opt/ibm/xlC/13.1.0/include/) を検索します。

パラメーター

directory_path
: コンパイラーが、XL C++ ヘッダー・ファイルを検索するディレクトリー・パスです。 *directory_path* は、相対パスまたは絶対パスにすることができます。パスは、引用符で囲むとコマンド行で分割されることがありません。

使用法

このオプションを使用すると、特定コンパイルの検索パスを変更できます。 XL C++ ヘッダーのデフォルト検索パスを永久的に変更するには、構成ファイルを使用します。詳しくは、 15 ページの『組み込みファイルのディレクトリー検索シーケンス』を参照してください。

このオプションが複数回指定されている場合、コンパイラーは最後に指定されたオプションのみを使用します。

このオプションは、**-qnostdinc** オプションが有効である場合には無視されます。

事前定義マクロ

なし。

例

mypath/headers1 および mypath/headers2 を使用して XL C++ ヘッダーのデフォルト検索パスをオーバーライドするには、以下を入力します。

```
xlc++ myprogram.C -qcpp_stdinc=mypath/headers1:mypath/headers2
```

関連情報

- 177 ページの『-qgcc_cpp_stdinc (C++ のみ)』
- 351 ページの『-qstdinc』
- 191 ページの『-qinclude』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』

-D

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

マクロ #define プリプロセッサー・ディレクティブ内と同様に定義する。

構文

デフォルト

適用されません。

パラメーター

name
定義するマクロです。-D*name* は #define *name* と同等です。例えば、-DCOUNT は #define COUNT と同等です。

definition
name に割り当てる値です。 -D*name=definition* は、 #define *name definition* と同等です。例えば、-DCOUNT=100 は #define COUNT 100 と同等です。

使用法

#define ディレクティブを使用して、既に **-D** オプションによって定義されているマクロ名を定義すると、エラー条件の結果になります。

-D オプションによって定義されるマクロの定義解除に使用される **-U***name* オプションは、**-D***name* オプションより高い優先順位を持ちます。

事前定義マクロ

コンパイラー構成ファイルは、**-D** オプションを使用して、特定の呼び出しコマンドに対する複数のマクロ名を事前定義します。詳しくは、ご使用システムの構成ファイルを参照してください。

例

`myprogram.c` で、`COUNT` という名前のインスタンスをすべて 100 に置換するには、以下のように入力します。

```
xlc myprogram.c -DCOUNT=100
```

関連情報

- 380 ページの『-U』
- 481 ページの『第 6 章 コンパイラーの事前定義マクロ』

-qdataimported、-qdatalocal、-qtocdata

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

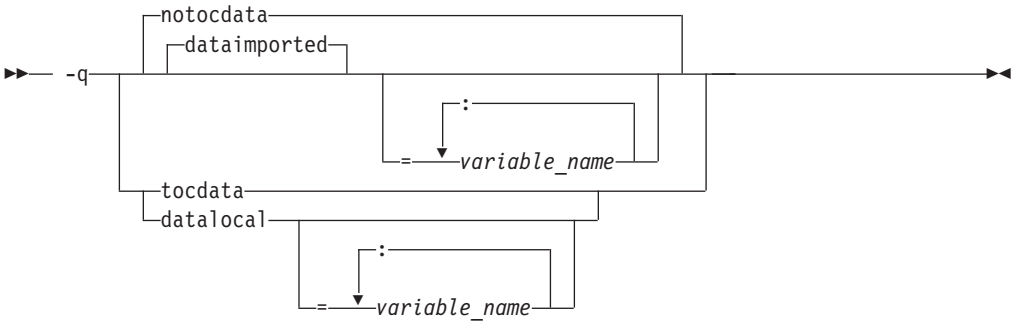
目的

64 ビット・コンパイルでローカルまたはインポートとしてデータにマークを付ける。

ローカル変数は、それらを使用する関数に静的にバインドされます。**-qdatalocal** オプションを使用すると、コンパイラーがローカルと想定する変数を指定できます。あるいは、**-qtocdata** オプションを使用して、すべての変数をローカルと想定するようにコンパイラーに命令できます。

インポートされる変数は、ライブラリーの共有部分と動的にバインドされます。**-qdataimported** オプションを使用すると、コンパイラーがインポート済みと想定する変数を指定できます。あるいは、**-qnotocdata** オプションを使用して、すべての変数をインポート済みと想定するようにコンパイラーに命令できます。

構文



デフォルト

-qdataimported または **-qnotocdata**。すべての変数がインポートされたとコンパイラーは想定します。

パラメーター

variable_name

コンパイラーがローカルまたはインポート済みと想定する変数の名前 (指定したオプションによって異なる)。

C++ 名前は、マングル名を使用して指定する必要があります。C++ マングル名を取得するには、**-c** コンパイラー・オプションを使用してオブジェクト・ファイルのみにソースをコンパイルし、その結果生じるオブジェクト・ファイルで **nm** オペレーティング・システム・コマンドを使用します。(名前マングリングを回避するための、宣言に対する `extern "C"` リンケージ指定子の使用について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『名前マングリング』も参照してください。)

variable_name なしで **-qdataimported** を指定するのは、**-qnotocdata** を指定するのと同等です。この場合は、すべての変数がインポートされたものと想定されます。*variable_name* なしで **-qdatalocal** を指定するのは、**-qtocdata** を指定するのと同等です。この場合は、すべての変数がローカルと想定されます。

使用法

これらのオプションは 64 ビット・コンパイルにのみ適用されます。

ローカルのマークが付けられた変数がインポートされた場合、不正なコードが生成され、パフォーマンスが低下することがあります。

変数なしでこれらのオプションのいずれかを指定すると、最後に指定したオプションが使用されます。同じ変数名を複数のオプションで指定すると、最後のオプションが使用されます。

事前定義マクロ

なし。

関連情報

- 310 ページの『-qprocimported、-qproclocal、-qprocunknown』

-qdbgfmt

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

オブジェクト・ファイル内のデバッグ情報のフォーマットを指定する。

DWARF は、プログラム内のデバッグ情報のフォーマットを定義する標準です。この拡張可能でコンパクトな標準は、幅広いオペレーティング・システムで使用されます。

構文

デフォルト

-qdbgfmt=dwarf

パラメーター

dwarf
DWARF 3 フォーマットでデバッグ情報を生成します。

dwarf4
DWARF 4 フォーマットでデバッグ情報を生成します。

注: -qdbgfmt=dwarf または **-qdbgfmt=dwarf4** を指定してビルドされたプログラムをデバッグするには、**dbx** などの DWARF 対応デバッガーが必要です。

使用法

-qdbgfmt は、**173 ページの『-g』** などのデバッグ・オプションを暗黙指定しません。例を以下に示します。

- DWARF 3 フォーマットでデバッグ情報を生成するには、**-g -qdbgfmt=dwarf** を使用します。
- DWARF 4 フォーマットでデバッグ情報を生成するには、**-g -qdbgfmt=dwarf4** を使用します。

-qdbgfmt は、**173 ページの『-g』** 情報のサブセットを生成する **260 ページの『-qlinedebug』** にも適用されます。

関連情報

- 173 ページの『-g』
- 260 ページの『-qlinedebug』

-qdbxextra (C のみ)

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

#pragma options dbxextra

目的

-g オプションと一緒に使用すると、参照されていない typedef 宣言、構造体、共用体、および enum 型定義にデバッグ情報を生成されるように指定します。

オブジェクト・ファイルおよび実行可能ファイルのサイズを最小にするため、コンパイラーはプログラムによって参照される typedef 宣言、struct 型定義、union 型定義、および enum 型定義の情報のみを組み込みます。 **-qdbxextra** オプションを指定すると、デバッグ情報がオブジェクト・ファイルのシンボル・テーブルに組み込まれます。このオプションは **-qsymtab=unref** オプションと同等です。

構文

デフォルト

-qnodbxextra: 参照されない typedef 宣言、struct 型定義、union 型定義、および enum 型定義はオブジェクト・ファイルのシンボル・テーブルに組み込まれません。

使用法

-qdbxextra を使用すると、オブジェクトおよび実行可能ファイルのサイズが増加する場合があります。

事前定義マクロ

なし。

例

myprogram.c をコンパイルして、参照されていない typedef、構造体、共用体、および列挙型の宣言をシンボル・テーブルに組み込んでデバッガーで使用できるようにするには、以下のように入力します。

```
xlc myprogram.c -g -qdbxextra
```

関連情報

- 169 ページの『-qfullpath』
- 260 ページの『-qlinedebug』
- 173 ページの『-g』
- 436 ページの『#pragma options』
- 359 ページの『-qsymtab (C のみ)』

-qdigraph

カテゴリー

言語エレメント制御

プラグマ同等物

#pragma options [no]digraph

目的

一部のキーボードにない文字を表すため、連字キーの組み合わせ C++ およびオペレーター・キーワード C++ の認識を使用可能にする。連字キーの組み合わせには、<:、<% などが含まれます。 C++ オペレーター・キーワードには、and、or などが含まれます。 C++

構文

デフォルト

- C **extc89 | extended | extc99 | stdc99** 言語レベルが有効な場合は **-qdigraph** です。その他のすべて言語レベルでは、**-qnodigraph** です。
- C++ **-qdigraph**

使用法

連字とは、一部のキーボードで使用できない文字を指定できるキーワードまたはキーの組み合わせです。ダイグラフの詳細については、「*XL C/C++ ランゲージ・リファレンス*」の 『ダイグラフ文字』を参照してください。

事前定義マクロ

-qdigraph が有効な場合は、__DIGRAPHS__ が 1 に事前定義されます。それ以外の場合は未定義です。

例

プログラムをコンパイルするときに連字の文字シーケンスを使用できないようにするには、以下のように入力します。

```
xlc myprogram.c -qnodigraph
```

関連情報

- 227 ページの『-qlanglvl』
- 376 ページの『-qtrigraph』

-qdirectstorage

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

特定のコンパイル単位がライトスルーを使用可能にしたストレージまたはキャッシュ禁止ストレージを参照する可能性があることをコンパイラーに通知する。

構文

デフォルト

-qnodirectstorage

使用法

このオプションは慎重に使用してください。これは、メモリーとキャッシュ・ブロックの機能、およびアプリケーションを最適なパフォーマンスにチューニングする方法を知っているプログラマーによる使用を目的としているからです。アプリケーションがすべてのインプリメンテーションで正しく実行されることを保証するには、別々の命令とデータ・キャッシュが存在しており、それに応じてアプリケーションをプログラミングしていることを想定する必要があります。

-qdollar

カテゴリー

言語エレメント制御

プラグマ同等物

#pragma options [no]dollar

目的

ID の名前にドル記号 ($) シンボルを使用できるようにする。

dollar が有効な場合、ID の中のドル記号 $ は基本文字として扱われます。

構文

デフォルト

-qnodollar

使用法

nodollar と **ucs** の言語レベルが両方有効な場合は、ドル記号は外字として扱われ、¥u0024 に変換されます。

事前定義マクロ

なし。

例

$ をプログラム中の ID で使用できるように myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qdollar
```

関連情報

- 227 ページの『-qlanglvl』

-qdump_class_hierarchy (C++ のみ)

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

それぞれのクラス・オブジェクトの階層と仮想関数テーブルのレイアウトの表記をファイルにダンプする。

構文

デフォルト

適用されません。

使用法

出力ファイル名は、サフィックス .class が付加されたソース・ファイル名で構成されます。

事前定義マクロ

なし。

例

myprogram.c をコンパイルして、クラス階層情報を含む myprogram.C.class というファイルを生成するには、以下のように入力します。

```
xlc++ myprogram.C -qdump_class_hierarchy
```

-e

カテゴリー

リンク

プラグマ同等物

なし。

目的

-qmkshrobj と一緒に使用された場合、共有オブジェクトのエントリー・ポイントを指定する。

構文

デフォルト

なし。

パラメーター

name
共有実行可能ファイルのエントリー・ポイントの名前です。

使用法

-e オプションは、**-qmkshrobj** オプションと一緒にのみ指定します。詳しくは、**-qmkshrobj** の説明を参照してください。

注: オブジェクト・ファイルをリンクする場合は、**-e** オプションを使用しないでください。実行可能出力のデフォルト・エントリー・ポイントは、__start です。このラベルを **-e** フラグで変更すると、エラーが生成される可能性があります。

事前定義マクロ

なし。

関連情報

- 277 ページの『-qmkshrobj』

-E

カテゴリー

出力制御

プラグマ同等物

なし。

目的

コンパイラー呼び出しに指定されたソース・ファイルをプリプロセスし、コンパイルせずに出力を標準出力に書き込む。

構文

デフォルト

デフォルトで、ソース・ファイルをプリプロセス、コンパイル、リンクすることで実行可能ファイルが作成されます。

使用法

認識されないファイル名サフィックスを持つソース・ファイルは、C ファイルとして扱われてプリプロセスされます。

-qnoppline が指定されていなければ、トークンのソース位置を保持するために `#line` ディレクティブが生成されます。継続シーケンスが保持されます。

-C が指定されていない限り、プリプロセスされた出力では、コメントは単一のシングル・スペース文字で置換されます。複数のソース行にわたるコメントに対して、改行および `#line` ディレクティブが出されます。

-E オプションは、**-P**、**-o**、および **-qsyntaxonly** オプションをオーバーライドします。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルし、プリプロセスされたソースを標準出力に送るには、以下のように入力します。

```
xlc myprogram.c -E
```

`myprogram.c` が以下のようなコード・フラグメントを持っている場合:

```
#define SUM(x,y) (x + y)
int a ;
#define mm 1   /* This is a comment in a
                  preprocessor directive */
```

```
int b ;        /* This is another comment across
                  two lines */
int c ;
               /* Another comment */
c = SUM(a, /* Comment in a macro function argument*/
        b) ;
```

出力は以下のようになります。

```
#line 2 "myprogram.c"
int a ;
#line 5
int b ;

int c ;

c = a + b ;
```

関連情報

- 304 ページの『-qppline』
- 124 ページの『-C、-C!』
- 288 ページの『-P』
- 361 ページの『-qsyntaxonly (C のみ)』

-qeh (C++ のみ)

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

例外処理がコンパイルされているモジュールで使用可能であるかどうかを制御する。

-qeh が有効な場合、例外処理は使用可能になります。プログラムが C++ 構造化例外処理を使用しない場合は、 **-qnoeh** を指定してコンパイルすることで、アプリケーションに必要のないコードを生成しないようにしてください。

構文

デフォルト

-qeh

使用法

-qeh を指定すると、**-qrtti** も暗黙指定されます。 **-qeh** が **-qnortti** とともに指定されている場合、RTTI 情報は必要なものとして生成されます。

事前定義マクロ

-qeh が有効な場合は、__EXCEPTIONS が 1 に事前定義されます。それ以外の場合は未定義です。

関連情報

- 322 ページの『-qrtti (C++ のみ)』

-qenum

カテゴリー

浮動小数点および整数のコントロール

プラグマ同等物

#pragma options enum、#pragma enum

目的

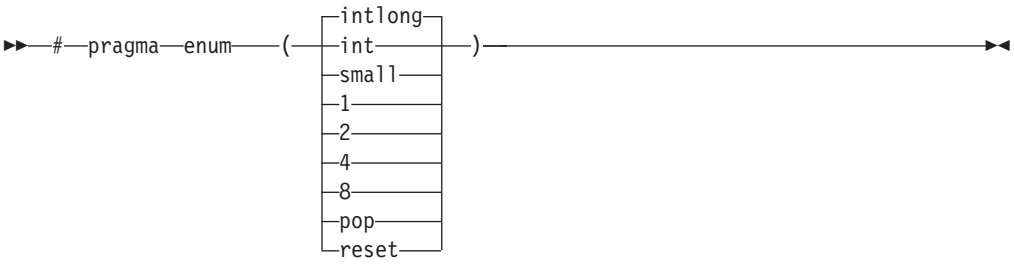
列挙が占有するストレージの量を指定する。

C++11 **-qenum** オプションは、固定の基本型がないスコープなしの列挙にのみ影響します。固定の基本型がある列挙の場合、**-qenum** オプションは無視されます。 C++11

構文

オプション構文

プラグマ構文



デフォルト

-qenum=intlong

パラメーター

1 列挙型がストレージの 1 バイトを占有し、列挙型の値の範囲が signed char の限度内の場合は signed char 型、それ以外の場合は unsigned char 型であると指定します。

2 列挙型がストレージの 2 バイトを占有し、列挙型の値の範囲が signed short の限度内の場合は short 型、それ以外の場合は unsigned short 型であると指定します。 C 値は signed int の範囲を超えることはできません。

4 列挙型がストレージの 4 バイトを占有し、列挙型の値の範囲が signed int の限度内の場合は int 型で、そうでない場合は unsigned int 型であると指定します。

8 列挙が 8 バイトのストレージを占有することを指定します。32 ビットのコンパイル・モードでは、列挙型の値の範囲が signed long long の限度内の場合は long long 型、そうでない場合は unsigned long long 型となります。64 ビットのコンパイル・モードでは、列挙型の値の範囲が signed long の限度内の場合は long 型、そうでない場合は unsigned long 型となります。

int

C 列挙型が 4 バイトのストレージを占め、int 型であると指定します。

C++ 列挙型がストレージの 4 バイトを占有し、列挙型の値の範囲が signed int の限度内の場合は int 型で、そうでない場合は unsigned int 型であると指定します。

intlong

列挙型の値の範囲をいずれかの int または unsigned int で表現できない場合に、**8** サブオプションのように、列挙型が 8 バイトのストレージを占有することを指定します。それ以外の場合、列挙型は **4** サブオプションのように 4 バイトのストレージを占有します。

small

列挙型が、正確に列挙型の値の範囲を表すことができる最少量のスペース (ストレージの 1、2、4、または 8 バイト) を占有することを指定します。符号は、値の範囲に負の値が含まれていなければ、unsigned です。8 バイトの enum の結果の場合、使用される実際の列挙型はコンパイル・モードに依存します。

pop | reset (プラグマのみ)

現行のプラグマ設定を破棄し、前のプラグマ・ディレクティブによって指定された設定に戻してください。前のプラグマ・ディレクティブを指定しないと、コマンド行またはデフォルト・オプションの設定に戻ります。

使用法

以下のテーブルは、事前定義の型を選択するための優先順位を示したものです。また、このテーブルは、事前定義の型、対応する事前定義の型の enum 定数の最大範囲、およびその事前定義の型に必要なストレージの量 (つまり sizeof 演算子が最小サイズの enum に適用されたときに生み出す値) も示します。特に明記されていない限り、すべての型は signed です。

表 24. 列挙のサイズおよび型

	enum=1		enum=2		enum=4		enum=8			
							32 ビットのコンパイル・モード		64 ビットのコンパイル・モード	
範囲	var	const	var	const	var	const	var	const	var	const
0..127	signed char	int	short	int	int	int	long long	long long	long	long
-128..127	signed char	int	short	int	int	int	long long	long long	long	long
0..255	unsigned char	int	short	int	int	int	long long	long long	long	long
0..32767	ERROR[1]	int	short	int	int	int	long long	long long	long	long
-32768..32767	ERROR[1]	int	short	int	int	int	long long	long long	long	long
0..65535	ERROR[1]	int	unsigned short	int	int	int	long long	long long	long	long
0..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
-(2147483647+1) ..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
0..4294967295	ERROR[1]	unsigned int[2]	ERROR[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	long long	long long	long	long
0..(2^{63}-1)	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long long[2]	long long[2]	long[2]	long[2]
-2^{63}..(2^{63}-1)	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long long[2]	long long[2]	long[2]	long[2]
0..2^{64}	ERROR[1]	unsigned long[2]	ERROR[1]	unsigned long[2]	ERROR[1]	unsigned long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

	enum=int		enum=intlong				enum=small			
			32 ビットのコンパイル・モード		64 ビットのコンパイル・モード		32 ビットのコンパイル・モード		64 ビットのコンパイル・モード	
範囲	var	const	var	const	var	const	var	const	var	const
0..127	int	int	int	int	int	int	unsigned char	int	unsigned char	int
-128..127	int	int	int	int	int	int	signed char	int	signed char	int
0..255	int	int	int	int	int	int	unsigned char	int	unsigned char	int
0..32767	int	int	int	int	int	int	unsigned short	int	unsigned short	int
-32768..32767	int	int	int	int	int	int	short	int	short	int
0..65535	int	int	int	int	int	int	unsigned short	int	unsigned short	int
0..2147483647	int	int	int	int	int	int	unsigned int	unsigned int	unsigned int	unsigned int
-(2147483647+1) ..2147483647	int	int	int	int	int	int	int	int	int	int
0..4294967295	unsigned int[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]

0..(2^{63}-1)	ERR[2]	ERR[2]	long long[2]	long long[2]	long[2]	long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]
-2^{63}..(2^{63}-1)	ERR[2]	ERR[2]	long long[2]	long long[2]	long[2]	long[2]	long long[2]	long long[2]	long[2]	long[2]
0..2^{64}	ERR[2]	ERR[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

注:

- これらの列挙型は **-qenum=1|2|4|** C **int** C 設定には大きすぎます。重大エラーが発行され、コンパイルが停止されます。この条件を訂正するには、列挙型の範囲を縮小するか、もっと大きな **-qenum** 設定を選択するか、動的 **-qenum** 設定 (**small** または **intlong** など) を選択してください。
- C 列挙型は、C アプリケーションを ISO C 1989 および ISO C 1999 標準にコンパイルする場合、`int` の範囲を超えることはできません。 **stdc89** | **stdc99** 言語レベルが有効である場合、列挙型の値が `int` の範囲を超えており、有効になっている **-qenum** オプションがその値をサポートしていると、コンパイラーは以下のように動作します。
  - **-qenum=int** が有効な場合は、重大エラー・メッセージが発行されて、コンパイルが停止されます。
  - **-qenum** の他のすべての設定の場合は、通知メッセージが発行されて、コンパイルは継続されます。
- `-qenum=8` の場合、この表を見ると、2^{32}-1 までのすべての範囲の列挙子の値について、基本型が `long long` (32 ビット・コンパイル・モード) および `long` (64 ビット・コンパイル・モード) であることがわかります。これは、列挙子の値が `int` または `unsigned int` に収まる場合、基本型は `int` よりも大きくなってはならないという、標準の規則に違反しています。
- `-qenum=small` の場合、この表を見ると、0 から 2147483647 の範囲の列挙子 (`65535` より大きな値を持つ列挙子が少なくとも 1 つある) について、基本型が `unsigned int` であり、`int` にプロモートできないことがわかります。これは、`int` へのプロモーションが可能である (0 から 2147483647 の範囲の列挙値を保持できるため)、という標準の規則に違反しています。

#pragma enum ディレクティブは、それに続く `enum` 変数より前に指定してください。宣言内で使用されるすべてのディレクティブは無視され、警告を伴う診断が下されるためです。

ソース・ファイルに入れる **#pragma enum** ディレクティブごとに、そのファイルの終わりの前に、対応する **#pragma enum=reset** を入れることをお勧めします。これによって、1 つのファイルが、そのファイルを格納する別のファイルの設定を変更してしまうことを回避できます。

例

以下のフラグメントを **enum=small** オプションを指定してコンパイルした場合:

```
enum e_tag {a, b, c} e_var;
```

列挙型定数の範囲は 0 から 2 までになります。この範囲は、上記テーブルに記述されているすべての範囲に収まります。 優先順位に基づいて、コンパイラーは、事前定義型 unsigned char を使用します。

以下のフラグメントを **enum=small** オプションを指定してコンパイルした場合:

```
enum e_tag {a=-129, b, c} e_var;
```

列挙型定数の範囲は -129 から -127 までになります。この範囲は、short (signed short) と int (signed int) の範囲の間だけになります。short (signed short) はより小さいので、enum を表すために使用されます。

以下のコード・セグメントには警告が生成され、 2 番目に現れる **enum** プラグマは無視されます。

```
#pragma enum(small)
enum e_tag {
  a,
  b,
  #pragma enum(int) /* error: cannot be within a declaration */
  c
} e_var;
#pragma enum(reset)
#pragma enum(reset) /* second reset isn't required */
```

事前定義マクロ

なし。

-F

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

なし。

目的

コンパイラーの代替構成ファイルまたはスタンザを指定する。

構文

デフォルト

デフォルトで、コンパイラーはインストール時に構成された構成ファイルを使用し、そのファイルで定義されたスタンザを現在使用中の呼び出しコマンドに対して使用します。

パラメーター

file_path
使用される代替のコンパイラー構成ファイルの絶対パス名です。

stanza
コンパイルに使用する構成ファイル・スタンザの名前です。これは、使用中の呼び出しコマンドに関係なくコンパイラーに *stanza* の記入項目を使用するよう指示します。例えば、**xlc** を使用してコンパイルする場合に、**c99** スタンザを指定すると、コンパイラーは **c99** スタンザに指定されたすべての設定を使用します。

使用法

-F オプションで指定したすべてのファイル名やスタンザは、システム構成ファイルで指定されたデフォルトをオーバーライドします。 XLC_USR_CONFIG 環境変数でカスタム構成ファイルを指定した場合、そのファイルは、-F オプションで指定されたファイルより先に処理されます。

-B、**-t**、および **-W** オプションは、 **-F** オプションをオーバーライドします。

事前定義マクロ

なし。

例

デフォルトの構成ファイルに追加した `debug` というスタンザを使用して `myprogram.c` をコンパイルするには、以下のように入力してください。

```
xlc myprogram.c -F:debug
```

`/usr/tmp/myconfig.cfg` という構成ファイルを使用して `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -F/usr/tmp/myconfig.cfg
```

`/usr/tmp/myconfig.cfg` という構成ファイルで作成したスタンザ `c99` を使用して `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -F/usr/tmp/myconfig.cfg:xlf95c99
```

関連情報

- 41 ページの『カスタム・コンパイラー構成ファイルの使用』
- 121 ページの『-B』
- 362 ページの『-t』
- 394 ページの『-W』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』
- 28 ページの『コンパイル時およびリンク時の環境変数』

-qfdpr

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

オブジェクト・ファイルに IBM Feedback Directed Program Restructuring (FDPR) パフォーマンス・チューニング・ユーティリティーが結果の実行可能ファイルを最適化するために必要とする情報を提供する。

-qfdpr が有効な場合は、最適化データがオブジェクト・ファイルに保管されます。

構文

デフォルト

-qnofdpr

使用法

最高の結果を得るために、プログラム内ですべてのオブジェクト・ファイルに **-qfdpr** を使用します。FDPR は、静的にリンクされていても、ライブラリー・コードではなく **-qfdpr** でコンパイルされたファイルでのみ最適化を実行します。

FDPR ユーティリティーが実行する最適化は、**-qpdf** オプションが実行する最適化と似たものです。

FDPR パフォーマンス調整ユーティリティーには独自の制限がいくつかあるため、このユーティリティーを使用しても、どのプログラムの実行時間も必ず短縮されるとは限りません。また、オリジナル・プログラムとまったく同じ結果が得られる実行可能プログラムが必ず生成されるとも限りません。

事前定義マクロ

なし。

例

FDPR ユーティリティーが要求するデータを組み込むように `myprogram.c` をコンパイルするには、以下を入力します。

```
xlc myprogram.c -qfdpr
```

関連情報

- 292 ページの『-qpdf1、-qpdf2』

-qflag

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

#pragma options flag、 448 ページの『#pragma report (C++ のみ)』

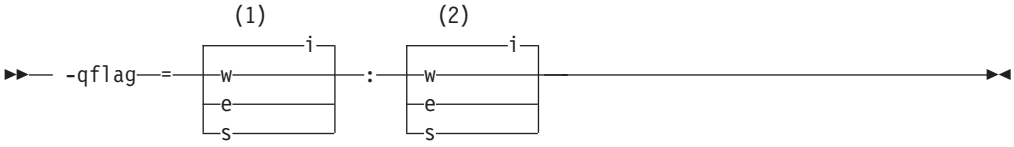
目的

診断メッセージを指定した重大度レベルまたはそれより高い重大度レベルのものに制限する。

メッセージが標準出力に書き込まれますが、オプションで、リスト・ファイルが生成されていればそれに書き込むこともできます。

構文

-qflag 構文 - C

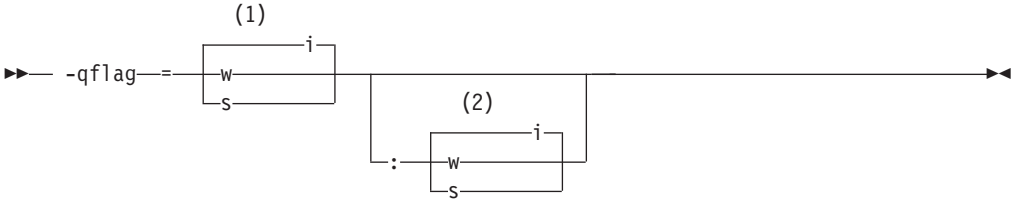


注:

1 リストで報告する最小の重大度レベルのメッセージ

2 端末で報告する最小の重大度レベルのメッセージ

-qflag 構文 - C++



注:

1 リストで報告する最小の重大度レベルのメッセージ

2 端末で報告する最小の重大度レベルのメッセージ

デフォルト

-qflag=i : i で、すべてのコンパイラー・メッセージが表示されます。

パラメーター

i 警告、エラー、および通知メッセージなど、すべての診断メッセージの表示を指定します。通知メッセージ (I) の重大度は最小です。

w 警告 (W) およびすべてのタイプのエラー・メッセージの表示を指定します。

C **e**

エラー (E)、重大エラー (S)、回復不能エラー (U) メッセージのみの表示を指定します。

s C 重大エラー (S) および回復不能エラー (U) メッセージのみの表示を指定します。C++ 重大エラー (S) メッセージのみの表示を指定します。

使用法

C リスト報告と端末報告の両方に、最小のメッセージ重大度レベルを指定してください。

C++ リスト報告に、最小のメッセージ重大度レベルを指定してください。 端末に対してサブオプションを指定しないと、コンパイラーはリストに対して指定した場合と同じ重大度を想定します。

-qflag を使用しても、**-qinfo** オプションによって制御される通知メッセージのクラスを使用可能にできません。詳しくは、**-qinfo** を参照してください。

-qhaltonmsg オプションは、**-qflag** オプションよりも優先されます。 **-qhaltonmsg** と **-qflag** の両方が指定された場合は、**-qflag** によって選択されていないメッセージも表示されます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、生成されたすべてのメッセージをリストに表示し、エラーおよび重大度の高いレベルのメッセージ (エラー修正を補助する関連した通知メッセージ付き) だけをワークステーションに表示するには、以下のように入力します。

```
xlc myprogram.c -qflag=i:e
```

関連情報

- 193 ページの『-qinfo』
- 181 ページの『-qhaltonmsg』
- 393 ページの『-w』
- 19 ページの『コンパイラー・メッセージ』

-qfloat

カテゴリー

浮動小数点および整数のコントロール

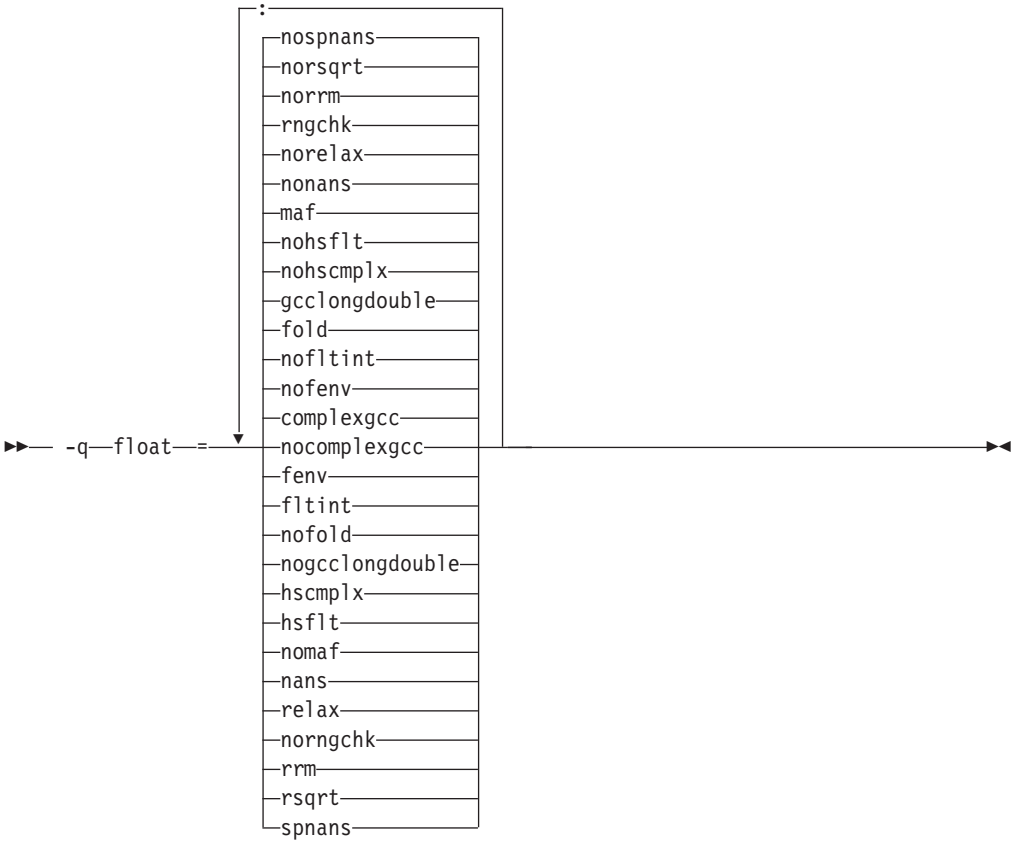
プラグマ同等物

#pragma options float

目的

浮動小数点の計算を高速化したり、精度を上げるためのさまざまなストラテジーを選択する。

構文



デフォルト

- **-qfloat=nocomplexgcc** (64 ビット・モードが使用可能な場合)
- **-qfloat=complexgcc:nofenv:nofltint:fold:gcclongdouble:nohscmplx:nohsflt:**

  maf:nonans:norelax:rngchk:norrm:norsqrt:nospnans
- **-qfloat=fltint:rsqrt:norngchk** (**-qnostrict**、**-qstrict=nooperationprecision:noexceptions**、または **-O3** 以上の最適化レベルが有効な場合)

パラメーター

complexgcc | nocomplexgcc

複素数を引き渡す、または戻すために GCC 規則を使用できるかどうかを指定します。**complexgcc** は、GCC にコンパイルされたコードとの互換性を保持します。 このサブオプションは、複素数型のサポートが有効でない場合には効果がありません。詳しくは、 227 ページの『-qlanglvl』を参照してください。

fenv | nofenv

コードが、ハードウェア環境に依存するかどうか、この依存関係によって予期せぬ結果が生じる可能性がある最適化を抑制するかどうかを指定します。

特定の浮動小数点操作は、浮動小数点状況および制御レジスター (FPSCR) に依存して、例えば、丸めモードを制御またはアンダーフローを検出します。特に、多くのコンパイラー組み込み関数は、FPSCR から値を直接読み取ります。

nofenv が有効なとき、コンパイラーは、プログラムがハードウェア環境に依存していないこと、浮動小数点操作のシーケンスを変更するアグレッシブなコンパイラー最適化が許可されていることを前提とします。**fenv** が有効なとき、そのような最適化は抑制されます。

予期せぬ動作を起こす可能性がある最適化から保護するために、ハードウェア浮動小数点環境を読み取る、または設定する記述を含むコードに **fenv** を使用してください。

ソース・コード内に指定される任意のディレクティブ (標準 C FENV_ACCESS プラグマなど) は、オプション設定より優先されます。

fltint | nofltint

ライブラリー関数の呼び出しではなく、コードのインライン・シーケンスを使用することによって浮動小数点と整数との間の変換をスピードアップします。**nofltint** が有効なときに呼び出されるライブラリー関数は、整数の表示可能範囲外の浮動小数点値をチェックし、範囲外の浮動小数点値を渡す場合に、最小または最大の表示可能整数を戻します。

-qarch を、浮動小数点から整数に変換する命令を備えたプロセッサーに設定した場合は、**[no]fltint** 設定に関係なく、その命令が使用されます。また、この変換は 64 ビット・モードのすべての Power プロセッサーにも適用されます。

-O3 以上の最適化レベルでコンパイルする場合、**fltint** は自動的に使用可能になります。使用不可にするには、**-qstrict**、**-qstrict=operationprecision**、または **-qstrict=exceptions** も指定します。

fold | nofold

コンパイル時に浮動小数点の定数式を評価します。これは、実行時に評価する場合とは多少異なる結果を出す場合があります。 **nofold** が指定されていても、コンパイラーは常に仕様ステートメント内の定数式を評価します。

gcclongdouble | nogcclongdouble

コンパイラーが、128 ビットの long double 操作で GCC 提供または IBM 提供のライブラリー関数を使用するかどうかを指定します。

gcclongdouble は、数値計算で GCC とのバイナリー互換性を確実にします。この互換性がご使用のアプリケーションで重要でない場合は、パフォーマンスを高めるために **nogcclongdouble** を使用してください。 このサブオプションは、128 ビットの long double 型が **-qldbl128** で使用可能な場合のみ有効です。

注: 結果を、**nogcclongdouble** でコンパイルされたモジュールから **gcclongdouble** でコンパイルされたモジュールに引き渡すと、異なる数字結果が作成されます (Inf、 NaN および他のまれなケースなど)。そのような非互換を回避するために、コンパイラーは、IBM long double 型を GCC long double 型に変換する組み込み関数を提供します (詳細は、 508 ページの『2 進浮動小数点組み込み関数』 を参照)。

hscmplx | nohscmplx
複素数の除算および複素数の絶対値を含む演算のスピードアップを行います。 **hsflt** サブオプションの最適化のサブセットを提供するこのサブオプションは、複素数計算では優先されます。

hsflt | nohsflt
単精度式の丸めを防止して、浮動小数点部を除数の逆数を掛ける乗算と置き換えることによって、計算をスピードアップします。また、浮動小数点と整数間の変換に対して、**fltint** サブオプションと同じ手法を使用します。 **hsflt** は、**hscmplx** を暗黙に示します。

hsflt サブオプションは、**nans** および **spnans** サブオプションをオーバーライドします。

注: 浮動小数点計算で特性が分っている場合は、複素数の除算および浮動小数点の変換を行うアプリケーションで **-qfloat=hsflt** を使用します。特に、浮動小数点結果はすべて、単精度表示の定義範囲内になければなりません。このオプションは、警告なしで予期せぬ結果を起こすおそれがあるので、注意して使用してください。複素数計算では、**hscmplx** サブオプション (上述) を使用することをお勧めします。これは、**hsflt** の予期しない結果を起こさずに同じ程度のスピードアップをはかることができます。

maf | nomaf
適切な箇所で浮動小数点乗算・加算命令を使用することによって、より高速でより正確に浮動小数点計算を行います。 この結果は、コンパイル時に行われる類似の計算の結果または他のタイプのコンピューターでの結果と正確に同じにならない場合があります。 負のゼロの結果が作成される可能性があります。このサブオプションは、浮動小数点の中間結果の精度に影響を与える可能性があります。**-qfloat=nomaf** が指定されると、乗加算命令が正確さを必要としない場合、乗加算命令は生成されません。

nans | nonans
-qflttrap=invalid:enable オプションを使用してシグナル NaN (非数字) 値を含む例外条件の検出および処理を行うことを可能にします。シグナル NaN 値は、他の浮動小数点演算からは出てこないため、このサブオプションは、プログラムがこの値を明示的に作成する場合にのみ使用してください。

relax | norelax
通常、一部の浮動小数点の算術演算 (右側にゼロを含む加算や減算など) を除去することによって、速度を速めるために IEEE 適合を少し緩和します。**-qstrict=noieeefp** または **-qfloat=relax** が指定されている場合は、これらの変更が認められます。

rngchk | norngchk
-O3 以上の最適化レベルで **-qstrict** を指定しないと、範囲のチェックが、ソフトウェアの割り算演算とインライン化された平方根演算の入力引数で実行されるかどうかを制御します。**norngchk** の指定は、コンパイラーに範囲検査をスキップするように指示し、ループ内で除算演算および平方根演算を繰り返し行う状況でのパフォーマンスを向上させることができます。

norngchk を有効にして、以下の制限を適用します。

- 割り算演算の被除数は、+/-INF にしないでください。

- 割り算演算の除数は、0.0、+/- INF、または非正規数にしないでください。
- 被除数の商と除数は、+/-INF にしないでください。
- 平方根演算の入力は、INF にしないでください。

これらの条件が満たされない場合、不正な結果が生じる可能性があります。例えば、割り算演算の除数が 0.0 または非正規化数 (倍精度の場合は、絶対値 < 2^{-1022}、単精度の場合は、絶対値 < 2^{-126}) の場合、INF ではなく NaN になります。除数が +/- INF のとき、0.0 の代わりに NaN になります。入力が、sqrt 演算で +INF になる場合、INF ではなく NaN となります。

norngchk は、**-qnostrict** が有効なときのみ許可されます。**-qstrict**、**-qstrict=infinities**、**-qstrict=operationprecision**、または **-qstrict=exceptions** が有効なときは、**norngchk** は無視されます。

rrm | **norrm**
実行時に丸めモードがデフォルト (最も近い値に丸める) にならなければならない浮動小数点の最適化を、浮動小数点の丸めモードが変更されるか、実行時に最も近い値に丸めないようにコンパイラーに通知することで、防ぎます。プログラムが実行時の丸めモードを変更する場合は、**rrm** を使用してください。そうでない場合、プログラムは、間違った計算結果を出してしまいます。

rsqrt | **norsqrt**
平方根の結果で割る除算を、平方根の逆数を掛ける乗算と置き換えることによって、一部の計算をスピードアップします。

また、**-qignerrno** が指定されない場合は、**rsqrt** は有効ではありません。`errno` は、`sqrt` 機能呼び出しでは設定*されません*。

-O3 以上の最適化レベルでコンパイルする場合、**rsqrt** は自動的に使用可能になります。使用不可にするには、**-qstrict**、**-qstrict=nans**、**-qstrict=infinities**、**-qstrict=zerosigns**、または **-qstrict=exceptions** も指定します。

spnans | **nospnans**
単精度から倍精度への変換でシグナル NaN を検出するための追加の命令を生成します。

注: -qfloat サブオプションとそれらに対応する **-qstrict** の関係について詳しくは、352 ページの『-qstrict』を参照してください。

使用法

デフォルト設定以外の **-qfloat** サブオプションを使用すると、特定のサブオプションのすべての必須条件が満たされない場合に浮動小数点計算で正しくない結果を生じる可能性があります。この理由から、IEEE 浮動小数点値を含んだ浮動小数点計算の経験があり、プログラムにエラーを取り込む可能性を正しく評価できる場合にのみこのオプションを使用してください。も参照詳しくは、『浮動小数点演算の処理』(「*XL C/C++ 最適化およびプログラミング・ガイド*」)を参照してください。

-qstrict | **-qnostrict** および **float** サブオプションが競合する場合は、後に指定された設定が使用されます。

事前定義マクロ

例

コンパイル時に定数浮動小数点式が評価され、乗加算命令が生成されないように、myprogram.c をコンパイルするには、以下を入力します。

```
xlc myprogram.c -qfloat=fold:nomaf
```

関連情報

- 113 ページの『-qarch』
- 135 ページの『-qcomplexgccincl』
- 『-qflttrap』
- 256 ページの『-qldbl128』
- 352 ページの『-qstrict』

-qflttrap

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

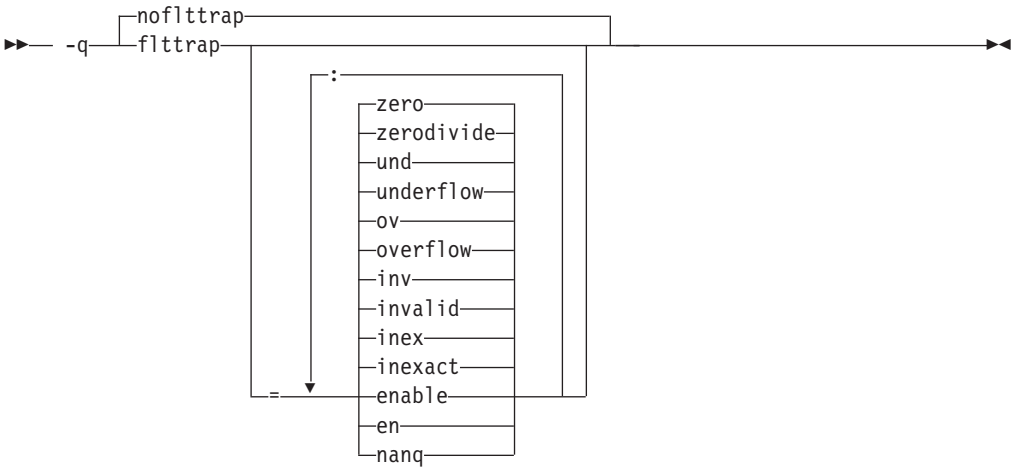
#pragma options [no]flttrap

目的

実行時に検出する浮動小数点例外のタイプを決定する。

該当する例外が発生すると、プログラムは **SIGFPE** シグナルを受信します。

構文



デフォルト

-qnoflttrap

パラメーター

enable、**en**
指定された例外 (**overflow**、**underflow**、**zerodivide**、**invalid**、または **inexact**) が発生した場合に、トラップを挿入します。ソース・コードを変更しないで例外トラッピングをオンにしたい場合は、このサブオプションを指定する必要があります。指定した例外のいずれかが発生した場合、SIGTRAP シグナルまたは SIGFPE シグナルが例外の正確なロケーションと共にプロセスに送信されます。

inexact、**inex**
浮動小数点の非整数演算の検出が有効になります。浮動小数点の非整数演算が発生した場合は、非整数演算例外の状況フラグが浮動小数点の状況および制御レジスター (FPSCR) で設定されます。

invalid、**inv**
浮動小数点の無効演算の検出が有効になります。浮動小数点の無効演算が発生した場合は、無効の演算例外の状況フラグが FPSCR で設定されます。

nanq
値が NaN である場合をトラップするために、各浮動小数点操作 (代入など) の前後、および浮動小数点結果を返す関数の各呼び出し後に NaNQ (Not a Number Quiet) 例外および NaNS (Not a Number Signalling) 例外を検出するコードを生成します。トラッピング・コードは、**enable** サブオプションが指定されているかどうかに関わらず生成されます。

overflow、**ov**
浮動小数点のオーバーフローの検出が有効になります。浮動小数点のオーバーフローが発生した場合は、オーバーフロー例外の状況フラグが FPSCR で設定されます。

underflow、**und**
浮動小数点のアンダーフローの検出が有効になります。浮動小数点のアンダーフローが発生した場合は、アンダーフロー例外の状況フラグが FPSCR で設定されます。

zerodivide、**zero**
浮動小数点のゼロ除算の検出が有効になります。浮動小数点のゼロ除算が発生した場合は、ゼロ除算の状況フラグが FPSCR で設定されます。

使用法

サブオプションを指定せずに **-qfltrap** オプションを指定することは、**-qfltrap=overflow:underflow:zerodivide:invalid:inexact** を指定することと同等です。

例外はハードウェアによって検出されますが、トラッピングは有効ではありません。

-qfltrap で `main` プログラムをコンパイルするごとに、**enable** サブオプションを使用することをお勧めします。これによってコンパイラーは、コードに適切な浮動小数点例外のライブラリー関数への呼び出しを組み込まずに、浮動小数点例外トラッピングを自動的に有効にするコードを生成します。

サブオプションの有無に関わらず **-qflttrap** を 2 回以上指定すると、サブオプションなしの **-qflttrap** は無視されます。

-qflttrap オプションは IPA とリンク中に認識されます。リンク・ステップでオプションを指定するとコンパイル時の設定値をオーバーライドします。

プログラムにシグナル NaN が含まれる場合は、すべての例外をトラップするために、**-qflttrap** と共に **-qfloat=nans** オプションを使用してください。

-qflttrap オプションが最適化オプションと共に指定されている場合、コンパイラーは以下の例のように振る舞います。

- **-O2** の場合:
  - 1/0 は、**div0** 例外を生成し、結果は無限大になります。
  - 0/0 は、無効演算を生成します。
- **-O3** 以上の場合:
  - 1/0 は、**div0** 例外を生成し、結果は無限大になります。
  - 0/0 は、直前の除算の結果によって乗算されたゼロを戻します。

注: 実行された変換および一部のベクトル命令の例外処理サポートが原因で、**-qsimd=auto** を使用すると、例外が catch されるロケーションが変更されるか、またはコンパイラーが例外の catch に失敗することもあります。

事前定義マクロ

なし。

例

```
#include <stdio.h>

int main()
{
  float x, y, z;
  x = 5.0;
  y = 0.0;
  z = x / y;
  printf("%f", z);
}
```

以下のコマンドを使用してこのプログラムをコンパイルした場合は、プログラムは除算が実行されると停止します。

```
xlc -qflttrap=zerodivide:enable divide_by_zero.c
```

zerodivide サブオプションが、保護対象とする例外のタイプを識別します。**enable** サブオプションを指定した場合は、例外が発生すると **SIGFPE** シグナルが生成されます。

関連情報

- 160 ページの『-qfloat』
- 113 ページの『-qarch』

-qformat

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

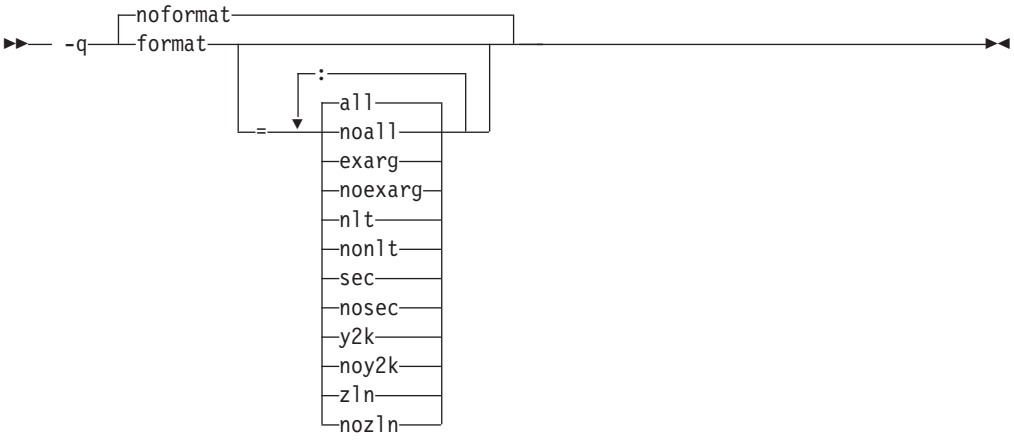
なし。

目的

ストリング入力および出力フォーマットの指定で考えられる問題について警告する。

診断される関数は printf、scanf、strftime、strfmon ファミリー関数とフォーマット属性でマークが付けられた関数です。

構文



デフォルト

-qnoformat

パラメーター

all | noall
すべてのフォーマット診断メッセージを有効または無効にします。

exarg | noexarg
printf および scanf スタイル関数呼び出しに余分な引数が指定された場合、警告を出します。

nlt | nonlt
フォーマット・ストリングがストリング・リテラルでない場合、フォーマット関数が va_list としてそのフォーマット引数を取らない場合を除き、警告を出します。

sec | nosec
フォーマット関数の使用において考えられるセキュリティー上の問題について警告を出します。

y2k | noy2k
2 桁の年を作成する strftime フォーマットについて警告を出します。

zln | nozln
ゼロの長さのフォーマットについて警告を出します。

サブオプションなしで **-qformat** を指定することは、**-qformat=all** を指定することと同等です。

-qnoformat は **-qformat=noall** と同等です。

事前定義マクロ

なし。

例

すべてのフォーマット・ストリング診断を使用可能にするには、次のいずれかを入力してください。

```
xlc myprogram.c -qformat=all
xlc myprogram.c -qformat
```

y2k の日付の診断を除き、すべてのフォーマット検査を使用可能にするには、以下のように入力してください。

```
xlc myprogram.c -qformat=all:noy2k
```

-qfullpath

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

#pragma options [no]fullpath

目的

このオプションは、**-g** オプションまたは **-qlinedebug** オプションと使用した場合、デバッグ情報付きでコンパイルされたオブジェクト・ファイルのソース・ファイルおよび組み込みファイルのフルパス名または絶対パス名を記録して、デバッグ・ツールが正確にソース・ファイルの場所を検索できるようにする。

fullpath が有効な場合、ソース・ファイルの絶対 (フル) パス名が保持されます。
nofullpath が有効な場合、ソース・ファイルの相対パス名が保持されます。

構文

デフォルト

-qnofullpath

使用法

実行可能ファイルを別のディレクトリーに移動すると、デバッガーに検索パスを指定しない限り、デバッガーはファイルを検出することができなくなります。 **fullpath** を使用すると、デバッガーによってファイルが正しく検出されるようになります。

事前定義マクロ

なし。

関連情報

- 260 ページの『-qlinedebug』
- 173 ページの『-g』

-qfunctrace

カテゴリー

エラー・チェックおよびデバッグ

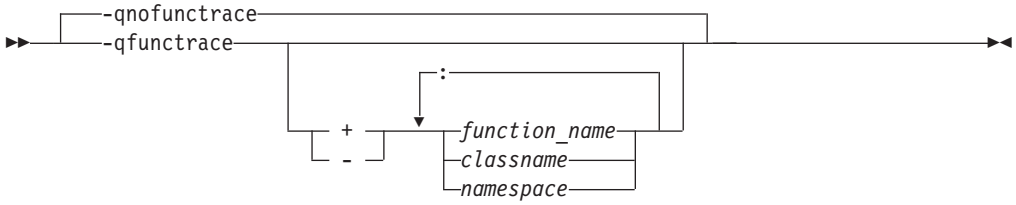
プラグマ同等物

なし。

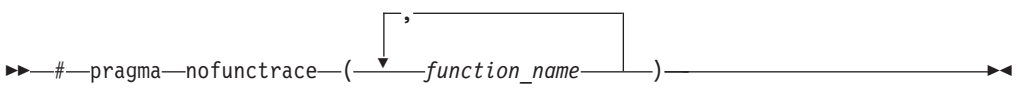
目的

コンパイル単位で指定された関数の入り口点および出口点をトレースするトレース・ルーチンを呼び出す。

構文



プラグマ構文



デフォルト

-qnofunctrace

注: C++ プログラムで **-qfunctrace** が指定された場合は、*std* 名前空間の関数はデフォルトではトレースされません。

パラメーター

+ *function_name*、*classes*、または *namespace*、およびそのすべての内部関数をトレースするようにコンパイラーに指示します。

- *function_name*、*classes*、*namespace*、およびその内部関数のいずれもトレースしないようにコンパイラーに指示します。

function_name
トレースする関数の名前を示します。

classname
トレースするクラスの名前を示します。

namespace
トレースする名前空間を示します。

F2008 F2008

使用法

-qfunctrace は、プログラムのすべての関数に対するトレースを有効にします。**-qnofunctrace** は、**-qfunctrace** で有効にされたトレースを無効にします。

-qfunctrace+ および **-qfunctrace-** サブオプションは、関数の特定のリストに対するトレースを有効にし、**-qnofunctrace** の影響を受けません。この関数のリストは累積リストです。

このオプションは、定義したトレース関数への呼び出しを挿入します。これらの関数は、リンク・ステップで指定する必要があります。トレース関数のインターフェースと、これらの呼び出しのタイミングの詳細については、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『コード内のトレース関数』セクションを参照してください。

+ または **-** を使用して、コンパイラーでトレースを行う関数、クラス名、または名前空間を示します。例えば、関数 `x` をトレースする場合は、`-qfunctrace+x` を使用します。関数のリストをトレースするには、コロン **:** を使用してこれらを分離する必要があります。

1 行に 2 つのコロン (`::`) を使用するとスコープ修飾子と見なされ、C++ の修飾名を示す場合に使用できます。例えば、`-qfunctrace+A::B:C` を使用すると、修飾子 `A::B` または `C` で始まる関数をトレースできます。

コード内で関数をトレースする場合は、以下の C 関数プロトタイプを使用して、コードに関数のトレースを書き込むことができます。

- `void __func_trace_enter(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` は、入り口点のトレース・ルーチンの定義に使用します。
- `void __func_trace_exit(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` は、出口点のトレース・ルーチンの定義に使用します。
- `void __func_trace_catch(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` は、キャッチ・トレース・ルーチンの定義に使用します。

コードに上記の関数プロトタイプを書き込む場合、関数を定義する必要があります。

これらの関数プロトタイプおよびその呼び出すタイミングの詳細については、「*XL C/C++ 最適化およびプログラミング・ガイド*」の**『コード内のトレース関数』**セクションを参照してください。

注:

- **+** と **-** は一度に 1 つのみ使用できます。同じ **-qfunctrace** 呼び出しにこれらを両方一緒に使用しないでください。
- インライン関数の定義はトレースされます。インライン化されてトレースされないのはその呼び出しのみです。

事前定義マクロ

なし。

例

`x`、`y`、および `z` 関数をトレースするには、`-qfunctrace+x:y:z` を使用します。

`x` を除くすべての関数をトレースするには、`-qfunctrace -qfunctrace-x` を使用します。

-qfunctrace+ および **-qfunctrace-** サブオプションは、所定のリストの累積関数のトレースのみを有効/無効にします。関数、クラス、および名前空間を使用する場合、最も詳細に指定されたオプションが有効になります。以下に例のリストを示します。

- `-qfunctrace+x -qfunctrace+y` または `-qfunctrace+x -qnofunctrace -qfunctrace+y` は、`x` および `y` のみのトレースを有効にします。
- `-qfunctrace-x -qfunctrace` または `-qfunctrace -qfunctrace-x` は、コンパイル単位で `x` を除くすべての関数をトレースします。
- `-qfunctrace -qfunctrace+x` は、すべての関数をトレースします。
- `-qfunctrace+y -qnofunctrace` は、`y` のみをトレースします。
- `functionX` が `classX` のメンバー関数の場合は、`-qfunctrace-classX::functionX -qfunctrace+classX` または `-qfunctrace+classX -qfunctrace-classX::functionX` は、`classX` のすべてのメンバー関数をトレースしますが、`functionX` はトレースしません。これは、`classX::functionX` は `classX` より詳細な指定であるためです。より詳細に指定されたオプションの方が、指定が詳細ではないオプションより優先されます。

- `-qfunctrace+MyClass` は、`MyClass` のすべてのメンバー関数をトレースします。
- `-qfunctrace+std::vector` は、`std::vector` のすべてのインスタンス化をトレースします。
- `-qfunctrace+ABC -qfunctrace-ABC::foo` は、`foo` を除く名前空間 `ABC` で定義されているすべての関数をトレースします。

関連情報

- **#pragma nofunctrace** については、435 ページの『#pragma nofunctrace』を参照してください。
- 関数のトレース・ルーチンをコードにインプリメントする方法、その詳細な例、およびその使用規則のリストについては、「*XL C/C++ 最適化およびプログラミング・ガイド*」の**『コード内のトレース関数』**を参照してください。

-g

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

シンボリック・デバッガーが使用するデバッグ情報を生成し、選択したソース・ロケーションでのデバッグ・セッションでプログラムの状態を使用できるようにします。

プログラム状態とは、プログラムの実行中の特定のポイントにおけるユーザー変数の値のことです。

多様な **-g** レベルを使用することで、デバッグ機能とコンパイラー最適化の間のバランスを取ることができます。**-g** レベルを高くすると、デバッグのサポートが充実しますが、実行時または場合によってはコンパイル時のパフォーマンスが低下します。**-g** レベルを低くすると、実行時のパフォーマンスは向上しますが、デバッグ・セッションで一部の機能を使用できなくなります。

-O2 最適化レベルが有効な場合は、デバッグ機能が完全にサポートされます。

注: -O2 より高いレベルの最適化が有効な場合は、デバッグ機能が制限されます。

構文

デフォルト

-g0 と同等)

パラメーター

-g

- 最適化が無効な場合 (**-qnoopt**)、**-g** は **-g9** と同等です。
- **-O2** 最適化レベルが有効な場合、**-g** は **-g2** と同等です。

-g0 デバッグ情報を生成しません。プログラム状態は保存されません。

-g1 行番号およびソース・ファイル名に関する最小限の読み取り専用のデバッグ情報を生成します。プログラム状態は保存されません。このオプションは、**-qlinedebug** と同じです。

-g2 行番号、ソース・ファイル名、および変数に関する読み取り専用のデバッグ情報を生成します。

-O2 最適化レベルが有効な場合、プログラム状態は保持されません。

-g3、-g4

行番号、ソース・ファイル名、および変数に関する読み取り専用のデバッグ情報を生成します。

-O2 最適化レベルが有効な場合は、以下のようになります。

- プログラム状態は保存されません。
- デバッガーでは、関数のパラメーター値を各関数の先頭で使用できます。

-g5、-g6、-g7

行番号、ソース・ファイル名、および変数に関する読み取り専用のデバッグ情報を生成します。

-O2 最適化レベルが有効な場合は、以下のようになります。

- デバッガーでは、`if` 構文、ループ構文、関数定義、および関数呼び出しでプログラム状態を使用できます。詳しくは、『 175 ページの『使用法』』を参照してください。
- デバッガーでは、関数のパラメーター値を各関数の先頭で使用できます。

-g8 行番号、ソース・ファイル名、および変数に関する読み取り専用のデバッグ情報を生成します。

-O2 最適化レベルが有効な場合は、以下のようになります。

- デバッガーでは、すべての実行可能ステートメントの先頭でプログラム状態を使用できます。
- デバッガーでは、関数のパラメーター値を各関数の先頭で使用できます。

-g9 行番号、ソース・ファイル名、および変数に関するデバッグ情報を生成します。変数の値をデバッガーで変更できます。

-O2 最適化レベルが有効な場合は、以下のようになります。

- デバッガーでは、すべての実行可能ステートメントの先頭でプログラム状態を使用できます。
- デバッガーでは、関数のパラメーター値を各関数の先頭で使用できます。

使用法

最適化が無効なときに **-g2** 以上のレベルを指定すると、常にデバッグ情報が使用可能になります。**-O2** 最適化レベルが有効なときに **-g5** 以上のレベルを指定すると、選択したソース・ロケーションでデバッグ情報が使用可能になります。

-O2 とともに **-g8** または **-g9** を指定した場合は、実行可能ステートメントが存在するすべてのソース行でデバッグ情報が使用可能になります。

-O2 とともに **-g5**、**-g6**、または **-g7** を指定した場合は、以下の言語構造体でデバッグ情報が使用可能になります。

- `if` 構文

  すべての `if` ステートメント (`if` キーワードが指定されている行) の先頭でデバッグ情報が使用可能です。また、`if` 構文の直後に存在する次の実行可能ステートメントの先頭でも使用可能です。

- ループ構成体

  すべての `do` ステートメント、`for` ステートメント、または `while` ステートメント (`do` キーワード、`for` キーワード、または `while` キーワードが指定されている行) の先頭でデバッグ情報が使用可能です。また、`do` 構文、`for` 構文、または `while` 構文の直後に存在する次の実行可能ステートメントの先頭でも使用可能です。

- 関数定義

  関数の本体に存在する最初の実行可能ステートメントでデバッグ情報が使用可能です。

- 関数呼び出し

  ユーザー定義関数を呼び出す各ステートメントの先頭でデバッグ情報が使用可能です。また、関数呼び出しを含むステートメントの直後に存在する次の実行可能ステートメントの先頭でも使用可能です。

例

`myprogram.c` をコンパイルし、デバッグ用に `testing` という実行可能プログラムを生成するには、以下のコマンドを使用します。

```
xlc myprogram.c -o testing -g
```

以下のコマンドは、特定の **-g** レベルを **-O2** と併用して myprogram.c をコンパイルし、デバッグ情報を生成します。

```
xlc myprogram.c -O2 -g8
```

関連情報

- 427 ページの『#pragma ibm snapshot』
- 260 ページの『-qlinedebug』
- 169 ページの『-qfullpath』
- 279 ページの『-O、-qoptimize』
- 221 ページの『-qkeepparm』

-qgcc_c_stdinc (C のみ)

カテゴリー

コンパイラーのカスタマイズ

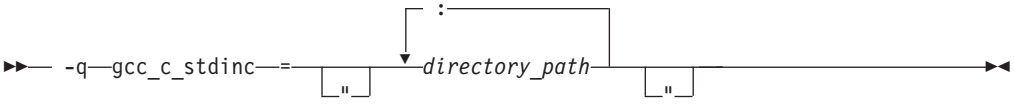
プラグマ同等物

なし。

目的

GNU C システム・ヘッダー・ファイルの標準検索ロケーションを変更する。

構文



デフォルト

コンパイラーはデフォルトにより、コンパイラー構成ファイルで指定されたディレクトリーを検索します。

パラメーター

directory_path
コンパイラーが、GNU C ヘッダー・ファイルを検索するディレクトリーのパスです。パスは、引用符で囲むとコマンド行で分割されることがありません。

使用法

このオプションを使用すると、特定コンパイルの検索パスを変更できます。 GNU C ヘッダーのデフォルト検索パスを永久的に変更するには、構成ファイルを使用します。詳しくは、 15 ページの『組み込みファイルのディレクトリー検索シーケンス』を参照してください。

このオプションが複数回指定されている場合、コンパイラーは最後に指定されたオプションのみを使用します。

このオプションは、**-qnostdinc** オプションが有効である場合には無視されます。

事前定義マクロ

なし。

例

`mypath/headers1` および `mypath/headers2` を使用して GNU C ヘッダーのデフォルト検索パスをオーバーライドするには、以下を入力します。

```
xlc myprogram.c -qgcc_c_stdinc=mypath/headers1:mypath:headers2
```

関連情報

- 139 ページの『-qc_stdinc (C のみ)』
- 351 ページの『-qstdinc』
- 191 ページの『-qinclude』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』

-qgcc_cpp_stdinc (C++ のみ)

カテゴリー

コンパイラーのカスタマイズ

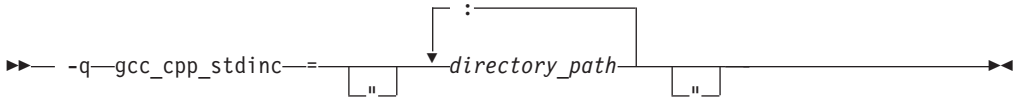
プラグマ同等物

なし

目的

GNU C++ システム・ヘッダー・ファイルの標準検索ロケーションを変更する。

構文



デフォルト

コンパイラーはデフォルトにより、コンパイラー構成ファイルで指定されたディレクトリーを検索します。

パラメーター

directory_path
コンパイラーが、GNU C++ ヘッダー・ファイルを検索するディレクトリーのパスです。パスは、引用符で囲むとコマンド行で分割されることがありません。

使用法

このオプションを使用すると、特定コンパイルの検索パスを変更できます。 GNU C++ ヘッダーのデフォルト検索パスを永久的に変更するには、構成ファイルを使用します。詳しくは、 15 ページの『組み込みファイルのディレクトリー検索シーケンス』を参照してください。

このオプションが複数回指定されている場合、コンパイラーは最後に指定されたオプションのみを使用します。

このオプションは、**-qnostdinc** オプションが有効である場合には無視されます。

事前定義マクロ

なし。

例

`mypath/headers1` および `mypath/headers2` を使用して GNU C++ ヘッダーのデフォルト検索パスをオーバーライドするには、以下を入力します。

```
xlc++ myprogram.C -qgcc_cpp_stdinc=mypath/headers1:mypath:headers2
```

関連情報

- 140 ページの『-qcpp_stdinc (C++ のみ)』
- 351 ページの『-qstdinc』
- 191 ページの『-qinclude』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』

-qgenproto (C のみ)

カテゴリー

移植性とマイグレーション

プラグマ同等物

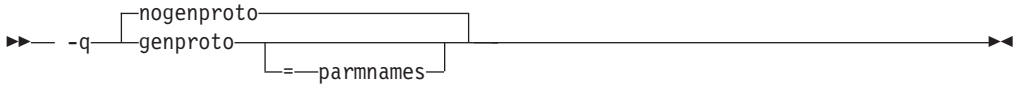
なし。

目的

K&R 関数定義または空の括弧付きの関数定義からプロトタイプ宣言を作成し、それらを標準出力に表示する。

コンパイラーは、K&R 関数定義、または空の括弧付きの関数宣言子を含む定義を受け入れコンパイルします。しかしこれらの関数定義は、C 標準では廃止されたものとして認識されます (**-qinfo=obs** オプションを有効にすると、コンパイラーはこれらの定義を診断します)。 **-qgenproto** が有効な場合、コンパイラーは対応するプロトタイプ宣言を生成し、標準出力で表示します。このオプションを使用すると、廃止された関数定義を識別できるようになり、それと同等のプロトタイプを自動的に取得します。

構文



デフォルト

-qnogenproto

パラメーター

parmnames
パラメーター名は、プロトタイプに組み込まれます。このサブオプションを指定しないと、パラメーター名はプロトタイプに組み込まれません。

事前定義マクロ

なし。

例

以下の関数定義を **- qgenproto** でコンパイルします。

```
int foo(a, b)   // K&R function
  int a, b;
{
}

int faa(int i) { } // prototyped function

main() {  // missing void parameter
}
```

以下の出力が表示されます。

```
int foo(int, int);
int main(void);
```

-qgenproto=parmnames を指定すると、以下のように表示されます。

```
int foo(int a, int b);
int main(void);
```

-qhalt

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

#pragma options halt

目的

コンパイル時のメッセージの最大重大度が指定した重大度と同じかそれを超える場合は、オブジェクト・ソース・ファイル、実行可能ソース・ファイル、またはアセンブラー・ソース・ファイルのいずれかを作成する前に、コンパイルを停止する。

構文

-qhalt 構文 — C

-qhalt 構文 — C++

デフォルト

-qhalt=s

パラメーター

i 警告、エラー、および情報のすべてのタイプのエラーによってコンパイルが停止することを指定します。情報診断 (I) の重大度は最小です。

w 警告 (W) および情報のすべてのタイプのエラーによってコンパイルが停止することを指定します。

C **e**
エラー (E)、重大エラー (S)、および回復不能エラー (U) によってコンパイルが停止することを指定します。

S C 重大エラー (S) および回復不能エラー (U) によってコンパイルが停止することを指定します。 C++ 重大エラー (S) によってコンパイルが停止することを指定します。

使用法

halt オプションの結果としてコンパイラーが停止したときは、コンパイラーの戻りコードはゼロ以外です。戻りコードのリストについては、 22 ページの『コンパイラー戻りコード』を参照してください。

-qhalt を複数回指定した場合は、最低の重大度レベルが使用されます。

診断メッセージは、**-qflag** オプションで制御できます。

-qmaxerr オプションを使用すると、同じ重大度のエラー数に基づいてコンパイルを停止するようコンパイラーに命令することもできます。このオプションは **-qhalt** をオーバーライドします。

-qhaltonmsg オプションを使用しても、エラー・メッセージ数に応じてコンパイルを停止できます。

事前定義マクロ

なし。

例

警告以上のレベルのメッセージが出された場合にコンパイルが停止するよう myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qhalt=w
```

関連情報

- 『-qhaltonmsg』
- 158 ページの『-qflag』
- 271 ページの『-qmaxerr』

-qhaltonmsg

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

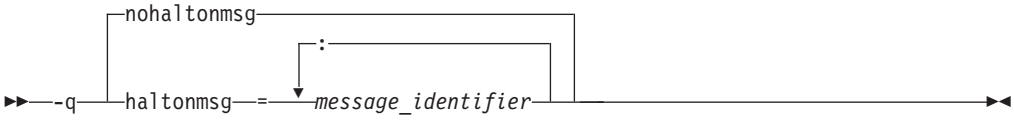
なし。

目的

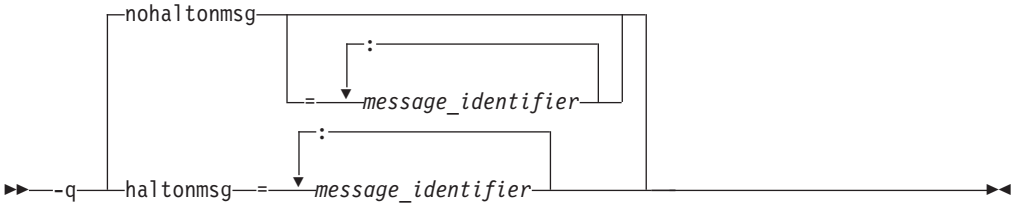
指定されたエラー・メッセージが生成された場合、オブジェクト・ファイル、実行可能ファイル、またはアセンブラー・ソース・ファイルを作成せずにコンパイルを停止する。

構文

-qhaltonmsg 構文 - C



-qhaltonmsg 構文 - C++



デフォルト

-qnohaltonmsg

パラメーター

message_identifier
メッセージ ID です。メッセージ ID は以下のフォーマットでなければなりません。

15*dd-number*

ここで、

15 コンパイラーの製品 ID です。

dd このメッセージを作成するコンパイラー・コンポーネントを示す 2 桁のコードです。これらのコードの説明については、20 ページの『コンパイラー・メッセージ・フォーマット』を参照してください。

number
メッセージ番号です。

使用法

-qhaltonmsg オプションの結果としてコンパイラーが停止した場合、コンパイラーの戻りコードはゼロ以外です。 **-qhaltonmsg** によって指定されたメッセージの重大度レベルは、元の重大度レベルが S より低い場合は S に変更されます。

重大度レベルが S のメッセージが発行されている場合、**-qnohaltonmsg** を指定してコンパイルを再開することはできません。

C **-qnohaltonmsg** コンパイラー・オプションは、**-qhaltonmsg** の直前の設定を取り消します。 C

C++ メッセージ ID と共に **-qnohaltonmsg** を指定した場合は、同じメッセージ ID を持つそれより前の **-qhaltonmsg** インスタンスが無効になります。特定のメッセージ ID を使用せずに **-qnohaltonmsg** を指定した場合は、それより前のすべての **-qhaltonmsg** インスタンスが無効になります。

以下のオプションのうち 2 つまたは 3 つを指定した場合は、最後に指定されたオプションが優先されます。

-qhaltonmsg=*message_identifier*

-qnohaltonmsg=*message_identifier*

-qnohaltonmsg

C++

-qhaltonmsg は、**-qsuppress** および **-qflag** よりも優先されます。

事前定義マクロ

なし。

関連情報

- 158 ページの『-qflag』
- 179 ページの『-qhalt』
- 19 ページの『コンパイラー・メッセージ』
- 357 ページの『-qsuppress』

-qhelp

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

コンパイラーの man ページを表示する。

構文

デフォルト

デフォルトでは、入力ファイルを指定せずにコンパイラーを呼び出した場合、man ページは表示されず、診断メッセージが出されます。

使用法

-qhelp オプションを指定すると、入力ファイルを指定したかどうかに関係なく、コンパイラーの man ページが表示され、コンパイルは停止します。**-qhelp** および **-qversion** オプションを同時に指定した場合、コンパイラーのバージョン情報のみが表示されます。

事前定義マクロ

なし。

関連情報

- -qversion

-qhot

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma novector

目的

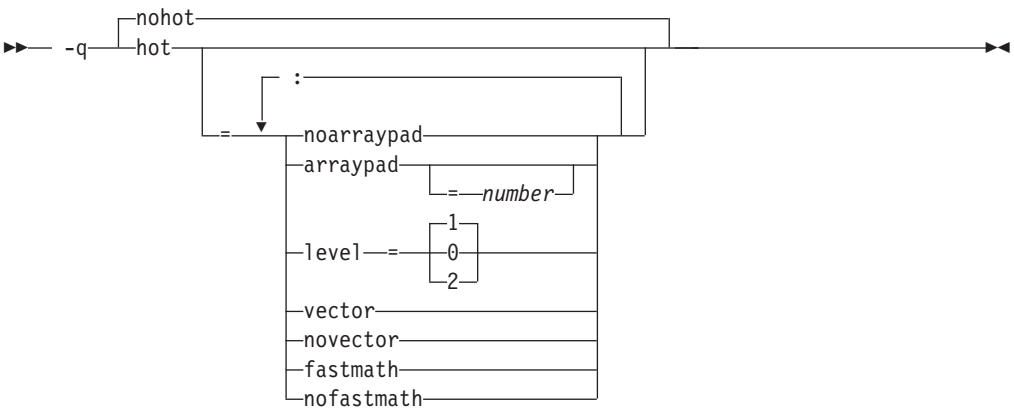
最適化中に上位ループ分析および変換 (HOT) を実行する。

-qhot コンパイラー・オプションは、ループと配列言語を最適化するためのチューニングを助ける強力な代替手段です。このコンパイラー・オプションは、指定されたサブオプションに関係なく、常にループの最適化を試行します。

プラグマ・ディレクティブを使用して、選択したセクションのコードの変換を使用不可にすることができます。

構文

オプション 構文



プラグマ構文

デフォルト

- **-qnohot**
- **-O3** が有効な場合、**-qhot=noarraypad:level=0:novector:fastmath**。
- **-qsmp**、**-O4**、または **-O5** が有効な場合、**-qhot=noarraypad:level=1:vector:fastmath**。
- **-qhot** をサブオプションなしで指定することは、**-qhot=noarraypad:level=1:vector:fastmath** と等価です。

パラメーター

arraypad（オプションのみ）| noarraypad（オプションのみ）

コンパイラーは配列処理ループの効率を高められそうな配列次元を増やすことができます。(キャッシュ・アーキテクチャーのインプリメンテーションのため、2 の累乗である配列次元がキャッシュの使用効率の低下を招く可能性があります。) **-qhot=arraypad** を指定して、ソースに 2 の累乗である次元を持つ大きな配列が含まれる場合は、配列処理プログラムを遅らせるキャッシュのミスとページ障害を削減することができます。これは特に、最初の次元が 2 の累乗である場合に効果的です。このサブオプションを *number* なしで使用する場合、コンパイラーは、有効であると推測した配列を埋め込んだり、選択した量だけを埋め込んだりします。すべての配列で必ずしも埋め込みが行われるわけではなく、また異なる配列ごとに異なる分量で埋め込みが行われます。 *number* を指定すると、コンパイラーは、コード内にすべての配列を埋め込みます。

注: arraypad の使用は危険です。埋め込みが起きた場合、再シェーピングや等価性のチェックを行わないため、コードを中断してしまう可能性があります。

number **(オプションのみ)**
それぞれの配列ごとにソース内に埋め込まれる要素の数を表す正整数値。埋め込み数は、正の整数値でなければなりません。埋め込み値は、最大配列エレメント・サイズの倍数 (通常、4、8、または 16) であることをお勧めします。

level=0 (オプションのみ)
上位変換のサブセットを実行し、デフォルトを **novector:noarraypad:fastmath** に設定します。

level=1 (オプションのみ)
上位変換のデフォルト設定を実行します。

level=2 (オプションのみ)
上位変換のデフォルト・セットと、より積極的なループ変換をいくつか実行します。**-qhot=level=2** は、**-qsmp** と一緒に使用する必要があります。このオプションは、積極的なループ分析と変換を行い、キャッシュの再利用状況を向上させ、ループの並列化の機会を増やします。

vector (オプションのみ) | novector
-qnostrictおよび **-qignerrno**、または **-O3** 以上の最適化レベルを指定したとき、**vector** は、コンパイラーに、配列の連続的要素においてループ内で実行されるある種の操作 (例えば、平方根、逆数平方根) を libxlopt の Mathematical Acceleration Subsystem (MASS) ライブラリーのルーチンへの呼び出しに変換させます。**vector** サブオプションは単精度および倍精度の浮動小数点の数学をサポートし、数学的処理の要求が大きいアプリケーションに役立ちます。

novector は、ループ配列演算の MASS ライブラリー・ルーチン呼び出しへの変換を使用不可にします。

ベクトル化はプログラムの結果の精度に影響を与えることがあるため、**-O4** 以上を使用する場合は、精度の変更が受諾不能ならば、**-qhot=novector** を指定してください。

fastmath (オプションのみ) | nofastmath (オプションのみ)
このサブオプションを使用すると、数学関数の高速スカラー・バージョンとデフォルトのバージョンのいずれかを使用するよう、アプリケーションをチューニングできます。

C/C++ の場合は、このサブオプションを **-qignerrno** とともに使用する必要があります (**-qignerrno** がまだ他のオプションによって使用可能になっていない場合)。

-qhot=fastmath を指定すると、**-qstrict=nolibrary** が使用可能に設定されている場合にのみ、数学ルーチンを、XLOPT ライブラリーから取得可能な数学ルーチンに置き換えることができます。**-qhot=nofastmath** は、XLOPT ライブラリーによる数学ルーチンの置き換えを使用不可にします。**-qhot=fastmath** は、**-qhot** が指定されている場合、ホット・レベルに関係なくデフォルトで使用可能になります。

使用法

コマンド行で **-qhot** を指定したとき、最適化レベルが指定されない場合は、コンパイラーは **-O2** と見なします。

-qsmp、**-O4** または **-O5** を使用して、デフォルトの **level** 設定 **1** を指定する場合、他のオプションの後に **-qhot=level=0** または **-qhot=level=2** を指定してください。

プラグマ・ディレクティブは、`while`、`do while`、および `for` ループにのみ適用します。これらのループの直後にはディレクティブが配置されます。これらは、指定されたループ内でネストされる可能性のある他のループには影響を与えません。

-qreport オプションを **-qhot** オプションまたは **-qhot** を暗黙指定するいずれかの最適化オプションと一緒に使用して、ループがどのように変換されたかを示す疑似 C レポートを生成できます。ループ変換は、オプション **-qreport** または **-qlistfmt** も指定されている場合は、リスト・レポートに組み込まれます。 このリスト・ファイルの `LOOP TRANSFORMATION SECTION` には、データ・プリフェッチ挿入ロケーションに関する情報も入っています。さらに、**-qprefetch=assistthread** を使用してプリフェッチ支援スレッドを生成する場合、リスト・ファイルの `LOOP TRANSFORMATION SECTION` に「データ・プリフェッチの支援スレッドが生成されました。」というメッセージも表示されます。**-qprefetch=assistthread** を指定すると、コンパイラーは最適化レベル**-O3** **-qhot** 以上で積極的なデータ・プリフェッチを生成します。詳しくは、『 315 ページの『-qreport』』を参照してください。

事前定義マクロ

なし。

関連情報

- 113 ページの『-qarch』
- 331 ページの『-qsimd』
- 305 ページの『-qprefetch』
- 315 ページの『-qreport』
- 279 ページの『-O、-qoptimize』
- 352 ページの『-qstrict』
- 335 ページの『-qsmp』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『*Mathematical Acceleration Subsystem (MASS) の使用*』

-I

カテゴリー

入力制御

プラグマ同等物

なし。

目的

ディレクトリーを組み込みファイルの検索パスに追加する。

構文

デフォルト

デフォルトの検索パスの説明については、 15 ページの『組み込みファイルのディレクトリー検索シーケンス』を参照してください。

パラメーター

directory_path
コンパイラーがヘッダー・ファイルを検索するディレクトリーのパスです。

使用法

-qnostdinc が有効な場合、コンパイラーは **、標準の検索パスではなく、 -I** オプションで指定されたパスのみ でヘッダー・ファイルを検索します。**-qidirfirst** が有効な場合、他のディレクトリーの前に、**-I** オプションで指定されたディレクトリーが検索されます。

構成ファイルとコマンド行の両方に **-I** directory オプションが指定されている場合は、構成ファイに指定されたパスが最初に検索されます。**-I** directory オプションは、コマンド行に複数回指定することができます。複数の **-I** オプションを指定した場合は、ディレクトリーは、コマンド行に指定された順序で検索されます。

-I オプションは、絶対パス名を使用して組み込まれたファイルに対して影響を与えません。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルし、`/usr/tmp` の次に `/oldstuff/history` で組み込みファイルを検索するには、以下のように入力してください。

```
xlc myprogram.c -I/usr/tmp -I/oldstuff/history
```

関連情報

- 188 ページの『-qidirfirst』
- 351 ページの『-qstdinc』
- 191 ページの『-qinclude』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』

-qidirfirst

カテゴリー

入力制御

プラグマ同等物

#pragma options [no]idirfirst

目的

他のディレクトリーを検索する*前* または*後* に **-I** オプションによって指定されたディレクトリーで、コンパイラーがユーザー組み込みファイルを検索するかどうかを指定する。

-qidirfirst が有効な場合、コンパイラーは、他のディレクトリーの前に、**-I** オプションで指定されたディレクトリーを検索します。 **-qnoidirfirst** が有効な場合、コンパイラーは、**-I** オプションで入力されたディレクトリーを検索する前に、a) **-qinclude** オプションで入力されたソース・ファイルが格納されるディレクトリー、および b) 組み込みファイルが格納されたディレクトリー、を検索します。

構文

デフォルト

-qnoidirfirst

使用法

このオプションが影響を与えるのは、`#include "file_name"` ディレクティブまたは **-qinclude** オプションで組み込まれたファイルのみです。**-qidirfirst** は **-qnostdinc** オプションから独立しており、XL C/C++ またはシステム・ヘッダー・ファイルの検索順序に影響を与えません。 (ヘッダー・ファイルの検索順序は、『 15 ページの『組み込みファイルのディレクトリー検索シーケンス』』を参照してください。) このオプションは、絶対パス名を使用して組み込まれたファイルに対しても影響を与えません。

最後の有効なプラグマ・ディレクティブは、後続のプラグマで置き換えられるまで有効のままです。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、(ソース・ファイルが常駐する) 現行ディレクトリーより先に `/usr/tmp/myinclude` で組み込みファイルを検索するには、以下のように入力します。

```
xlc myprogram.c -I/usr/tmp/myinclude -qidirfirst
```

関連情報

- 186 ページの『-I』
- 191 ページの『-qinclude』
- 351 ページの『-qstdinc』
- 139 ページの『-qc_stdinc (C のみ)』
- 140 ページの『-qcpp_stdinc (C++ のみ)』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』

-qignerrno

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options [no]ignerrno

目的

システム呼び出しが errno を変更しないかのようにコンパイラーに最適化の実行を許可する。

例外が発生すると、一部のシステム・ライブラリー関数は errno を設定します。**ignerrno** が有効な場合、errno の設定および後続の副次作用は無視されます。このオプションにより、errno に起きることとは関係なく、コンパイラーは最適化を実行できます。

構文

デフォルト

- -qnoignerrno
- **-qignerrno** (**-O3** 以上の最適化レベルが有効な場合)。

使用法

-O3 以上のレベルと errno を設定する能力が両方必要な場合は、コマンド行で最適化オプションの後 に **-qnoignerrno** を指定してください。

事前定義マクロ

C++ **ignerrno** が有効な場合、__IGNERRNO__ は 1 に定義されます。それ以外の場合は未定義です。

関連情報

- 279 ページの『-O、-qoptimize』

-qignprag

カテゴリー

言語エレメント制御

プラグマ同等物

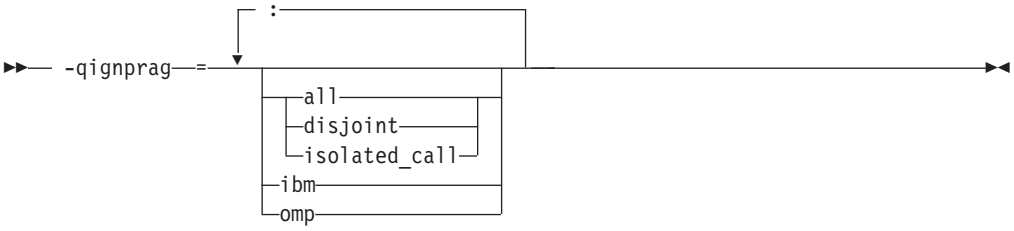
#pragma options ignprag

目的

特定のプラグマ・ステートメントを無視するようにコンパイラーに命令する。

このオプションは、別名割り当てプラグマのエラーの検出に役立ちます。別名割り当てが誤っていると、診断が困難なランタイム・エラーが発生します。ランタイム・エラーが発生しても、**-O** オプションと共に **ignprag** を使用してエラーが消えるときは、別名割り当てプラグマに指定された情報が誤っている可能性があります。

構文



デフォルト

適用されません。

パラメーター

all
ソース・ファイルのすべての **#pragma isolated_call** ディレクティブおよび **#pragma disjoint** ディレクティブを無視します。

disjoint
ソース・ファイルのすべての **#pragma disjoint** ディレクティブを無視します。

ibm
C ソース・ファイル内のすべての **#pragma ibm snapshot** ディレクティブを無視します。

isolated_call
ソース・ファイルのすべての **#pragma isolated_call** ディレクティブを無視します。

omp
#pragma omp parallel、 **#pragma omp critical** など、ソース・ファイルのすべての OpenMP 並列処理ディレクティブを無視します。

事前定義マクロ

なし。

例

myprogram.c をコンパイルして、**#pragma isolated_call** ディレクティブをすべて無視するには、以下のように入力します。

```
xlc myprogram.c -qignprag=isolated_call
```

関連情報

- 415 ページの『#pragma disjoint』
- 217 ページの『-qisolated_call』
- 427 ページの『#pragma ibm snapshot』
- 459 ページの『並列処理のためのプラグマ・ディレクティブ』

-qinclude

カテゴリー

入力制御

プラグマ同等物

なし。

目的

ソース・ファイルの #include 文で指定されているかのようにコンパイル単位に組み込まれる追加のヘッダー・ファイルを指定する。

ヘッダーが挿入されるのは、すべてのコード・ステートメントおよびソース・ファイルの #include プリプロセッサー・ディレクティブで指定されるすべてのヘッダーより前です。このオプションは、サポートされたプラットフォーム間の移植性のために用意されています。

構文

デフォルト

-qnoinclude

パラメーター

file_path
コンパイルするコンパイル単位に組み込まれるヘッダー・ファイルの絶対パス名または相対パス名です。*file_path* が相対パスで指定されると、その検索は 15 ページの『組み込みファイルのディレクトリー検索シーケンス』 で説明されるシーケンスの後に実行されます。

使用法

-qinclude オプションの使用法は、#include ディレクティブの使用法に類似しています。このセクションでは、**-qinclude** と #include の使用における相違点について説明します。

-qinclude オプションが適用されるのは、オプションが指定されたのと同じコンパイルで指定されたファイルのみです。これはリンク・ステップ中に発生するコンパイルや、暗黙コンパイル (オプション **-qtemplateregistry** によって呼び出される)、または **-qtempinc** によって生成されるファイルには渡されません。

このオプションが 1 つの呼び出しに複数回指定されると、コマンド行に出現する順番でヘッダー・ファイルが組み込まれます。同じヘッダー・ファイルがこのオプションで複数回指定されると、そのヘッダー・ファイルは、コマンド行に出現した順に、#include ディレクティブによってソース・ファイルに複数回組み込まれているかのように扱われます。

-qnoinclude を指定すると、以前に指定した -qinclude はすべて無視されます。**-qnoinclude** の後で指定した **-qinclude** のみが有効です。

ソース・ファイルでコメントを除いたステートメントより前に表示されなければならないすべてのプラグマ・ディレクティブが影響を受けます。これらのプラグマの配置を維持する必要がある場合、**-qinclude** を使用してファイルを組み込むことはできません。

-qinclude を他のオプションと併用する場合は、以下の規則が適用されます。

- C++ **-qtemplateregistry** と共に使用された場合、**-qinclude** は、テンプレート・レジストリー・ファイルに、それによって影響を受けるソース・ファイルと共に記録されます。これらのファイルの依存関係によってテンプレート・レジストリーの再コンパイルが開始される場合、**-qinclude** オプションが従属ファイルに渡されます。ただし、これらの従属ファイルがテンプレート・レジストリーに追加されたときに、このオプションがそれらの従属ファイルに対して指定されていた場合に限ります。
- **-qsource** を使用してリスト・ファイルを生成すると、**-qinclude** によって組み込まれるヘッダー・ファイルは、リストのソース・セクションには現れません。これらのヘッダー・ファイルをリストに表示させたい場合は、**-qshowinc=usr** または **-qshowinc=all** を **-qsource** と共に使用してください。
- コンパイラーの起動ディレクトリーが検索された後、**-I** オプションで検索チェーンに追加された検索パスが **-qinclude** によって検索されます。**-I** オプションは、**-qinclude** オプションの前または後に指定できます。
- **-qinclude** で指定されたファイルは、従属関係として **-M** 出力に含められます。しかし、**-qinclude** オプションと #include ディレクティブの従属関係ファイルのパスは異なります。これは、**-qinclude** オプションで指定されたファイルは先に起動パスで検索されますが、#include ディレクティブに記述されたファイルはそのように処理されないためです。

  -qinclude オプションを指定した初回のビルドの結果として従属関係ファイルが作成されているときに、それ以降のビルドで **-qinclude** オプションを指定しなかっ

た場合、そのときまでに **-qinclude** オプションで指定したヘッダー・ファイルが変更されていると、再コンパイルが行われます。

- **-qlistopt** オプションによって生成されるコンパイラー・リスト・ファイルでは、**-qinclude** オプションが使用されるごとに別個のエントリーが OPTION SECTION に出力されます。
- **-qsource** オプションと **-qinclude** オプションの両方を使用した場合、**-qinclude** で指定したヘッダー・ファイルは、#include ディレクティブのようにはプログラム・ソース・リストに出力されません。ただし、ソース・プログラムの #include ディレクティブで指定されたファイルは出力されます。

事前定義マクロ

なし。

例

ファイル test1.h および test2.h をソース・ファイル test.c にインクルードするには、以下のコマンドを入力します。

```
xlc -qinclude=test1.h test.c  -qinclude=test2.h
```

関連情報

- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』

-qinfo

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

#pragma options [no]info、#pragma info

目的

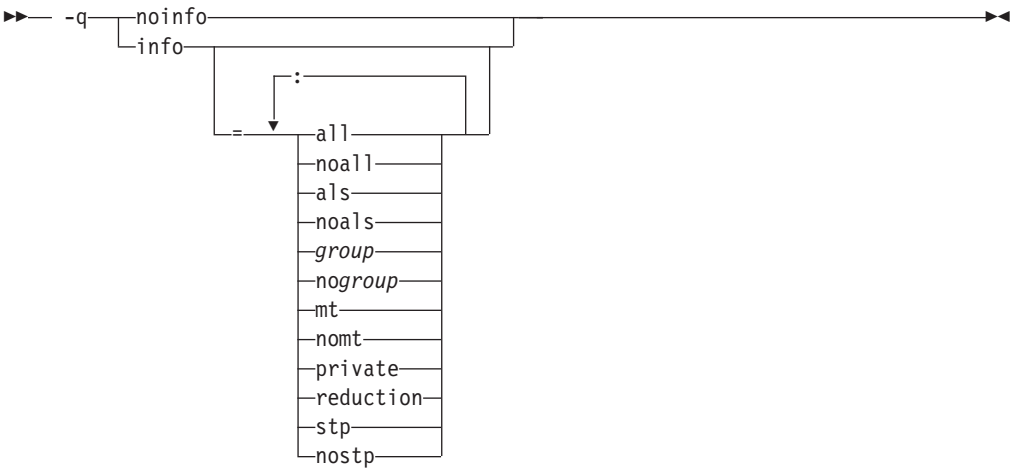
通知メッセージのグループを作成または抑制する。

メッセージが標準出力に書き込まれますが、オプションで、リスト・ファイルが生成されていればそれに書き込むこともできます。コンパイラーは、以下のファイルについてはメッセージを出しません。

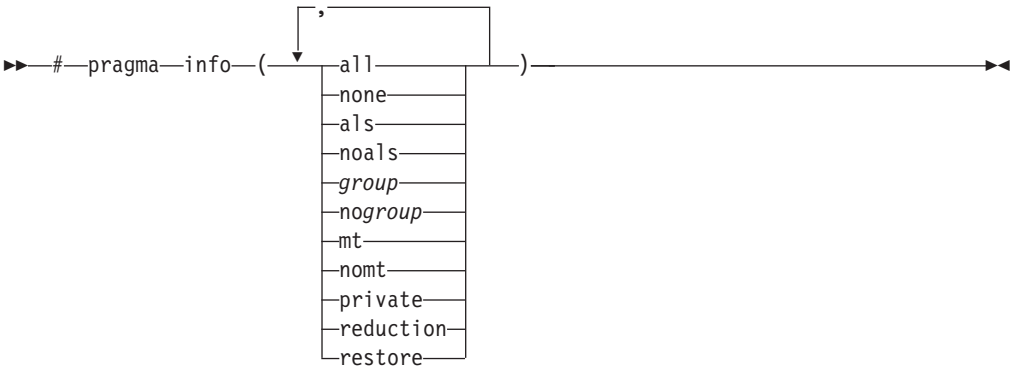
- コンパイラーの標準検索パス内にあるファイルおよびシステム・ヘッダー・ファイル。標準の検索パスは、以下のコンパイラー・オプションの影響を受けます。
  - 351 ページの『-qstdinc』
  - 139 ページの『-qc_stdinc (C のみ)』
  - 140 ページの『-qcpp_stdinc (C++ のみ)』
  - 176 ページの『-qgcc_c_stdinc (C のみ)』
  - 177 ページの『-qgcc_cpp_stdinc (C++ のみ)』
- コンパイラーの標準検索パス内にあるファイルによって最終的に組み込まれるファイルおよびシステム・ヘッダー・ファイル。

構文

オプション構文



プラグマ構文



デフォルト

-qnoinfo

- C -qnoinfo
- C++ -qinfo=lan:trx

パラメーター

all
すべてのグループ (**als**、**mt**、および **ppt** を除く) の診断メッセージを使用可能にします。

noall (オプションのみ)
すべてのグループのすべての診断メッセージを無効にします。

none (プラグマのみ)
すべてのグループのすべての診断メッセージを無効にします。

als
有効な ANSI 別名割り当て規則に対する違反と考えられる箇所の報告を有効にします。

noals
別名割り当て規則に対する違反と考えられる箇所の報告を無効にします。

group | **no***group*
特定のメッセージ・グループを有効または無効にします。ここで、*group* には次の 1 つ以上を使用できます。

group
戻される (抑制される) 通知メッセージのタイプ。

C **c99 | noc99**
C89 言語レベルと C99 言語レベルの間で異なる振る舞いをする可能性のある C コード。

C++ **cls | nocls**
C++ クラス。

cmp | nocmp
符号なしの比較において起こりうる冗長。

cnd | nocnd
条件式において起こりうる冗長または問題。

cns | nocns
定数が関係する演算。

cnv | nocnv
型変換。

dcl | nodcl
宣言の整合性。

eff | noeff
無効なステートメントおよびプラグマ。

enu | noenu
enum 変数の整合性。

ext | noext
未使用の外部定義。

gen | nogen
汎用診断メッセージ。

gnr | nognr
一時変数の生成。

got | nogot
goto 文の使用。

C **ini | noini**
配列を部分的に初期化する配列イニシャライザーを報告します。配列が部分的に初期化される場合、初期化されていないエレメントは、該当する型の値 0 を受け取ります。

lan | nolan
言語レベルの効果。

obs | noobs
廃止されたフィーチャー。

ord | noord
指定されていない評価の順序。

par | nopar
未使用のパラメーター。

por | nopor
移植不能な言語構造体。

ppc | noppc
プリプロセッサーの使用で起こりうる問題。

ppt | noppt
プリプロセッサー・アクションのトレース。

pro | nopro
関数プロトタイプの欠落。

rea | norea
到達できないコード。

ret | noret
戻りステートメントの整合性。

trd | notrd
データまたは精度において考えられる切り捨てまたは欠落。

tru | notru
コンパイラーによる変数名の切り捨て。

trx | notrx
16 進浮動小数点定数の丸め。

uni | nouni
未初期化の変数。**-qinfo=uni** オプションは、コーディング・スタイルを強制します。そのコーディング・スタイルの宣言で変数を初期化する必要があります。

unset | nouset (オプションのみ)
使用する自動変数を、設定する前に検出し、それらにコンパイル時に通知メッセージを使用してフラグを立てます。

-qinfo=unset は、最適化時に入手できる、制御フロー情報などのプログラム情報を使用します。結果として、検出の正確さが最適化レベルにより向上します。例えば、-O0 ではフラグが立てられない未設定変数のいくつかの使用例は、-O2 ではフラグが立てられます。 -O4 や -O5 などの積極的な最適化レベルは、通知メッセージ内の行番号が不正確になる原因となる場合があります。非常にまれなケースとして、これらの最適化レベルがプログラムを大幅に並べ替えて、静的分析から誤った肯定メッセージが出される原因となることがあります。 -O2 最適化レベルを指定すると、最良の結果を得ることができます。

-qinitauto オプションは、自動変数を初期化します。その結果、**-qinitauto** オプションは、使用する変数を、設定する前に **-qinfo=unset** オプションから隠します。

upg | noupg
前のリリースと比較して、現行コンパイラーのリリースの新しい振る舞いを記述するメッセージを生成。

use | nouse
未使用の自動および静的変数。

C++ **vft | novft**
仮想関数テーブルの生成。

mt | nomt

並列コード内の潜在的な同期の問題をレポートします。このサブオプションは、グローバル・スレッド・フラグ・パターンを検出します。このパターンでは、1 つのスレッドが共有揮発性フラグ変数を使用することにより、そのスレッドで計算が完了してその結果をメモリーに格納したことを他のスレッドに通知します。他のスレッドは、このフラグ変数を変更しないループによってフラグ変数を確認します。このフラグ変数の値が変わると、待機スレッドはメモリーからの計算結果にアクセスします。PowerPC ストレージ・モデルでは、このフラグ変数が最初のスレッドに設定される前、およびこのフラグ変数が待機スレッドで確認された後に同期を行う必要があります。同期は同期組み込み関数によって実行できます。

使用する必要がある同期ディレクティブのタイプは、コードによって変わります。通常は、__lwsync 関数でシステム・メモリーへのストレージ・アクセス順序が保持されるため、この関数を使用すれば充分です。ただし、命令プリフェッチによって、このフラグ変数が更新される前に計算結果にアクセスするコードの実行を開始する可能性があるような方法で待機スレッド内のループが作成されている場合、順序を保持するため、__isync などの関数の呼び出しが必要になります。このようなパターンは通常、以下のようになります。

```
gotosleep: sleep(value);
   if (!flag) goto gotosleep;
   // A call to the __isync function is needed here.
   x = shared_computation_result;
```

同期が必要ない一部のパターンは、上述のパターンと同様です。このサブオプションによって生成されるメッセージは、潜在的な同期の問題に関する提案のみです。

-qinfo=mt サブオプションを使用するには、**-qthreaded** オプションを使用可能にして、以下のオプションを少なくとも 1 つ指定する必要があります。

- **-O3**
- **-O4**
- **-O5**
- **-qipa**
- **-qhot**
- **-qsmp**

デフォルト・オプションは **-qinfo=nomt** です。

private
これは推奨されないサブオプションです。これは **-qreport** に置換されます。詳細については、「*XL C/C++ コンパイラー・リファレンス*」の 315 ページの『-qreport』 および 101 ページの『推奨されないオプション』セクションを参照してください。

reduction
これは推奨されないサブオプションです。これは **-qreport** に置換されます。詳細については、「*XL C/C++ コンパイラー・リファレンス*」の 315 ページの『-qreport』 および 101 ページの『推奨されないオプション』セクションを参照してください。

stp | nostp
スタック破損に対する保護がないプロシージャーについて、警告を出します。**-qinfo=stp** は、**-qstackprotect** オプションも使用可能に設定されていなければ効果がありません。他の **-qinfo** オプションと同じように、**-qinfo=stp** は **-qinfo=all / noall** によって使用可能または使用不可に設定されます。**-qinfo=nostp** がデフォルト・オプションです。

restore (プラグマのみ)
現行のプラグマ設定を破棄し、前のプラグマ・ディレクティブによって指定された設定に戻してください。前のプラグマ・ディレクティブを指定しないと、コマンド行またはデフォルト・オプションの設定に戻ります。

使用法

サブオプションなしで **-qinfo** を指定することは、**-qinfo=all** を指定することと同等です。

-qnoinfo を指定することは、**-qinfo=noall** を指定することと同等です。

別名割り当て規則の違反の報告を有効にするときは、以下を考慮してください。

- **-qalias=ansi** を設定してからでないと、別名割り当て規則の違反の報告 (**-qinfo=als**) を実施することはできません。
- **-qinfo=als** が明示的に指定されている場合、どのレベルの最適化またはインライン化でも、**-qinfo=noals** が暗黙指定され、警告が出されます。
- 診断はヒューリスティックに行われるため、false positive (違反でないものが違反と誤判定される場合) が出される場合があります。静的コンパイルでは、ポイント先分析を確定的に評価できません。診断に使用されるポイント先分析は、コンテキストおよびフローに依存しない方法で評価されます。診断のトレースバック・メッセージのシーケンスは、指定された順序で実行された場合に、間接式によって問題のオブジェクトが示される、というものです。その実行シーケンスがアプリケーションで実施されない場合、診断は false positive となります。(実施できる診断のタイプについては、『**例**』セクションを参照してください。)

事前定義マクロ

なし。

例

myprogram.c をコンパイルして、 変換ステートメントと未到達のステートメントを除くすべての項目に関する通知メッセージを作成するには、以下のコマンドを入力します。

```
xlc myprogram.c -qinfo=all -qinfo=nocnv:norea
```

C 以下の例は、コードが **-qinfo=cnd:eff:got:obs:par:pro:rea:ret:uni** でコンパイルされたときに、コンパイラーが検出するコード構文です。

```
#define COND 0

void faa() // Obsolete prototype (-qinfo=obs)
{
   printf("In faa¥n"); // Unprototyped function call (-qinfo=pro)
}

int foo(int i, int k)
{
   int j; // Uninitialized variable (-qinfo=uni)

   switch(i) {
   case 0:
   i++;
   if (COND) // Condition is always false (-qinfo=cnd)
      i--;   // Unreachable statement (-qinfo=rea)
   break;

   case 1:
      break;
      i++;   // Unreachable statement (-qinfo=rea)
   default:
      k = (i) ? (j) ? j : i : 0;
}

   goto L;   // Use of goto statement (-qinfo=got)
   return 3; // Unreachable statement (-qinfo=rea)
L:
   faa(); // faa() does not have a prototype (-qinfo=pro)

// End of the function may be reached without returning a value
// because of there may be a jump to label L (-qinfo=ret)

} //Parameter k is never referenced (-qinfo=ref)

int main(void) {
({ int i = 0; i = i + 1; i; }); // Statement does not have side effects (-qinfo=eff)

 return foo(1,2);
}
```

C++ 以下の例は、有効な **-qinfo=cls:cnd:eff:use** でコードをコンパイルしたときに、コンパイラーが検出するコード構文です。

```
#pragma abc       // pragma not supported (-qinfo=eff or -qinfo=gen)

int bar() __attribute__((xyz));    // attribute not supported (-qinfo=eff)
int j();

class A {
   public:
      A():  x(0), y(0), z(0) { };  // this constructor is in the correct order
                                   // hence, no info message.
      A(int m): y(0), z(0)
```

```
      { x=m; };            // suggest using member initialization list
                              for x (-qinfo=cls)

      A(int m, int n):
      x(0), z(0) { };       // not all data members are initialized
                            // namely, y is not initialized (-qinfo=cls)

      A(int m, int n, int* l):
      x(m), z(l), y(n) { };       // order of class initialization (-qinfo=cls)

    private:
      int x;
      int y;
      int *z;      // suggest having user-defined copy constructor/
                   // assignment operator to handle the pointer data member
                   // (-qinfo=cls)
};

int foo() {
   int j=5;
   j;        // null statement (-qinfo=eff)
             // The user may mean to call j().

return j;

}

void boo() {
   int x;
   int *i = &x;
   float *f;            // f is not used (-qinfo=use)
   f = (float *) i;     // incompatible type (-qinfo=eff)
                        // With ansi aliasing mode, a float pointer
                        // is not supposed to point to an int
}

void cond(int y) {
   const int i=0;
   int j;
   int k=0;

   if (i) {       // condition is always false (-qinfo=cnd)
            j=3;
    }

   if (1) {       // condition is always true  (-qinfo=cnd)
            j=4;
          }

   j=0;
   if (j==0) {     // cond. is always true  (-qinfo=cnd)
       j=5;
   }

   if (y) {
          k+=5
          }

   if (k==5) {     // This case cannot be determined, because k+=5
                   // is in a conditional block.
              j=6;
              }
}
```

以下の例では、MyFunction1 の前の **#pragma info(eff, nouni)** ディレクティブは、無効なステートメントまたはプラグマを識別するメッセージを生成し、未初期化変

数を識別するメッセージを表示しないようコンパイラーに命令します。MyFunction2 の前の **#pragma info(restore)** ディレクティブは、 **#pragma info(eff, nouni)** ディレクティブが指定される前に有効であったメッセージ・オプションを復元するようコンパイラーに命令します。

```
#pragma info(eff, nouni)
int MyFunction1()
{
  .
  .
  .

}

#pragma info(restore)
int MyFunction2()
{
  .
  .
  .

}
```

次の例に、別名割り当て違反に対して有効な診断を示します。

```
t1.c:
int main() {
  short s = 42;
  int *pi = (int*) &s;
  *pi = 63;
  return 0;
}
xlC -qinfo=als t1.c
"t1.c", line 4.3: 1540-0590 (I) Dereference may not conform to the current
                                aliasing rules.
"t1.c", line 4.3: 1540-0591 (I) The dereferenced expression has type "int".
                                "pi" may point to "s" which has incompatible
                                type "short".
"t1.c", line 4.3: 1540-0592 (I) Check assignment at line 3 column 11 of t1.c.
```

次の例では、floatToInt への 2 つの呼び出しが区別されない点で、この分析はコンテキスト依存ではありません。この例では別名割り当て違反はありませんが、診断は引き続き実行されます。

```
t2.c:
int* floatToInt(float *pf) { return (int*)pf; }

int main() {
  int i;
  float f;
   int* pi = floatToInt((float*)*&i));
   floatToInt(&f;)
   return *pi;
}

xlC -qinfo=als t2.c
"t2.c", line 8.10: 1540-0590 (I) Dereference may not conform to the current
                                 aliasing rules.
"t2.c", line 8.10: 1540-0591 (I) The dereferenced expression has type "int".
                                 "pi" may point to "f" which has incompatible
                                 type "float".
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 7 column 14 of t2.c.
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 1 column 37 of t2.c.
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 6 column 11 of t2.c.
```

```
t3.c:
int main() {
  float f;
  int i = 42;
  int *p = (int*) &f;
  p = &i;
  return *p;
}

xlC -qinfo=als t3.c
"t3.c", line 6.10: 1540-0590 (I) Dereference may not conform to the current
        aliasing rules.
"t3.c", line 6.10: 1540-0591 (I) The dereferenced expression has type "int".
        "p" may point to "f", which has incompatible type "float".
"t3.c", line 6.10: 1540-0592 (I) Check assignment at line 4 column 10 of t3.c.
```

sync.c をコンパイルして、並列コード内の潜在的な同期の問題に関する通知メッセージを生成するには、以下のコマンドを入力します。

```
xlc_r -O3 -qinfo=mt sync.c
```

sync.c には以下のコードが含まれているとします。

```
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

volatile int done;         /* shared flag */
volatile int result;       /* shared result */

void *setter(void *id)
{
  sleep(5);
  result = 7;
  /* Need __lwsync(); */
  done = 1;                /* line 13 */
}

void *waiter(void *id)
{
  while (!done)            /* line 18 */
  {
    sleep(1);
  }
  /* need __lwsync(); */
  printf("%d¥n", result);
}

int main()
{
  pthread_t threads[2];
  pthread_attr_t attr;
  int result;

  result = pthread_create(&threads[0], NULL, waiter, NULL);
  if (result != 0) exit(2);
  result = pthread_create(&threads[1], NULL, setter, NULL);
  if (result != 0) exit(3);

  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);

  return 0;
}
```

コンパイラーは、以下の通知メッセージを出します。

```
1586-669 (I) "sync.c", line 18: If this loop is used as a synchronization
point, additional synchronization via a directive or built-in function might
be needed.

1586-670 (I) "sync.c", line 13: If this statement is used as a synchronization
point, additional synchronization via a directive or built-in function might
be needed.
```

C 以下の関数 ini.c は、配列 a[3] を部分的に初期化しました。 ini.c をコンパイルしてこの問題に関する通知メッセージを生成するには、以下のコマンドを入力します。

```
xlc -qinfo=ini ini.c -c
```

ini.c には以下のコードが含まれているとします。

```
int a[3] = {1};
```

コンパイラーは、以下の通知メッセージを出します。

```
"ini.c", line 1.10: 1506-446 (I) Array element(s) [1] ...
[2] will be initialized with a default value of 0.
```

C

以下の関数 factorial.c は、n<1 の場合は result を初期化しません。 **-qnoopt** で **-qinfo=unset** を指定した場合には、この問題は検出されません。 factorial.c をコンパイルして、未初期化変数 result に関する通知メッセージを生成するには、以下のコマンドを入力します。

```
xlc -qinfo=unset -O factorial.c
```

factorial.c には、以下のコードが含まれます。

```
int factorial(int n) {
  int result;

  if (n > 1) {
    result = n * factorial(n - 1);
  }

  return result; /* line 8 */
}

int main() {
  int x = factorial(1);
  return x;
}
```

コンパイラーは、以下の通知メッセージを出します。

```
1500-099: (I) "factorial.c", line 8: "result" might be used before it is set.
```

関連情報

- 158 ページの『-qflag』
- 315 ページの『-qreport』
- 345 ページの『-qstackprotect』
- 530 ページの『同期関数』
- 推奨されないオプションのリストについては、「*XL C/C++ コンパイラー・リファレンス*」の 101 ページの『推奨されないオプション』セクションを参照してください。

- 同期および PowerPC ストレージ・モデルについて詳しくは、http://www.ibm.com/developerworks/systems/articles/powerpc.html の記事を参照してください。

-qinitauto

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

#pragma options [no]initauto

目的

デバッグのために、未初期化の自動変数を特定の値に初期化する。

構文

デフォルト

-qnoinitauto

パラメーター

hex_value
1 から 8 桁の 16 進数。

- ストレージの各バイトを特定の値に初期化するには、 *hex_value* に 1 桁か 2 桁で指定してください。
- ストレージの各ワードを特定の値に初期化するには、 *hex_value* に 3 桁から 8 桁で指定してください。
- 要求される初期化指定子のサイズに対して、最大桁数より少ない桁数が指定された場合は、先行ゼロが想定されます。
- ワード初期化の場合、自動変数の長さが 4 バイトの倍数より小さい場合は、適合するように *hex_value* の左方が切り捨てられます。例えば、自動変数が 1 バイトしかなく、*hex_value* に 5 桁を指定した場合は、コンパイラーは左方の 3 桁を切り捨てて、右方の残りの 2 桁を変数に割り当てます。『例 1』を参照してください。
- 自動変数の長さが *hex_value* より大きい場合、コンパイラーは *hex_value* を繰り返し、それを変数に代入します。『例 1』を参照してください。
- 自動変数が配列である場合は、*hex_value* は、繰り返しパターンの配列のメモリー・ロケーションに (配列の最初のメモリー・ロケーションから順に) コピーされます。『例 2』を参照してください。
- 英字桁の指定は、大文字でも小文字でも構いません。
- *hex_value* には、オプションで 0x のプレフィックスを付けることができます (x の大/小文字は区別しません)。

使用法

-qinitauto オプションには、以下の利点があります。

- *hex_value* をゼロに設定すると、可変修飾でない自動変数はすべて使用前にクリアされます。
- このオプションを使用して、実数型または複素数型の変数をシグナル NaN または静止 NaN に初期化できます。これは、プログラム内で未初期化の変数を検出するのに役立ちます。

このオプションは自動変数の値を初期化するための追加のコードを生成します。これによりプログラムの実行時のパフォーマンスが低下するため、このオプションはデバッグ目的でのみ使用してください。

制約事項:

- 同等なオブジェクト、構造体のコンポーネント、そして配列エレメントは、別々に初期化されることはありません。代わりに、ストレージのシーケンス全体が集合的に初期化されます。
- **-qinitauto=hex_value** オプションは、可変長配列は初期化しません。

事前定義マクロ

- __INITAUTO__ は **-qinitauto** オプションまたはプラグマで指定された *hex_value* に定義されます。それ以外の場合は未定義です。
- __INITAUTO_W__ は、**-qinitauto** オプションまたはプラグマで指定され、かつ 4 回繰り返された *hex_value* に定義されます。それ以外の場合は未定義です。

例

例 1: -qinitauto オプションを使用して、スカラー型の自動変数を初期化します。

```
#include <stdio.h>

int main()
{
  char a;
  short b;
  int c;
  long long int d;

  printf("char a = 0x%X:¥n",(char)a);
  printf("short b = 0x%X:¥n",(short)b);
  printf("int c = 0x%X:¥n",c);
  printf("long long int d = 0x%llX¥n",d);
}
```

例えば、-qinitauto=AABBCCDD を指定してプログラムをコンパイルした場合、結果は以下のようになります。

```
char a = 0xDD
short b = 0xFFFFCCDD
int c = 0xAABBCCDD
long long int d = 0xAABBCCDDAABBCCDD
```

例 2: -qinitauto オプションを使用して、自動配列変数を初期化します。

```
#include <stdio.h>
#define ARRAY_SIZE 5
```

```
int main()
{
  char a[5];
  short b[5];
  int c[5];
  long long int d[5];

  printf("array of char: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)a[i]);
  printf(":¥n");

  printf("array of short: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)b[i]);
  printf(":¥n");

  printf("array of int: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)c[i]);
  printf(":¥n");

  printf("array of long long int: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)d[i]);
  printf(":¥n");
}
```

例えば、-qinitauto=AABBCCDD を指定してプログラムをコンパイルした場合、結果は以下のようになります。

```
array of char: 0xAA 0xBB 0xCC 0xDD 0xAA
array of short: 0xAABB 0xCCDD 0xAABB 0xCCDD 0xAABB
array of int: 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD
array of long long int: 0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD
0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD
```

-qinlglue

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options [no]inlglue

目的

-O2 以上の最適化で使用すると、アプリケーション内の外部関数呼び出しを最適化するグルー・コードをインライン化する。

グルー・コード はリンカーによって生成され、2 つの外部関数の間で制御を渡す目的で使用します。 **-qinlglue** が有効である場合、最適化プログラムは、パフォーマンスを向上させるためにグルー・コードをインライン化します。**-qnoinlglue** が有効である場合、グルー・コードのインライン化は行われません。

構文

デフォルト

- **-qnoinlglue**（**-q32** が有効な場合)
- **-qinlglue**（**-q64** が有効な場合)
- **-qinlglue**（**-qtune=pwr5** 以上、**-qtune=auto**、または **-qtune=balanced** が有効な場合)

使用法

グルー・コードをインライン化すると、コード・サイズが大きくなる場合があります。**-qcompact** を指定すると、コードの増大を防止するために **-qinlglue** 設定はオーバーライドされます。**-qinlglue** を有効にしたい場合は、**-qcompact** を指定しないでください。

-qnoinlglue または **-qcompact** を指定すると、パフォーマンスが低下する場合があります。これらのオプションは注意して使用してください。

-qinlglue オプションは、ポインターを介した関数呼び出しまたは外部コンパイル単位の呼び出しにしか影響を及ぼしません。外部関数への呼び出しの場合は、例えば **-qprocimported** オプションを使用して、その関数がインポートされたものであることを指定します。

事前定義マクロ

なし。

関連情報

- 134 ページの『-qcompact』
- 310 ページの『-qprocimported、-qproclocal、-qprocunknown』
- 377 ページの『-qtune』

-qinline

カテゴリー

最適化およびチューニング

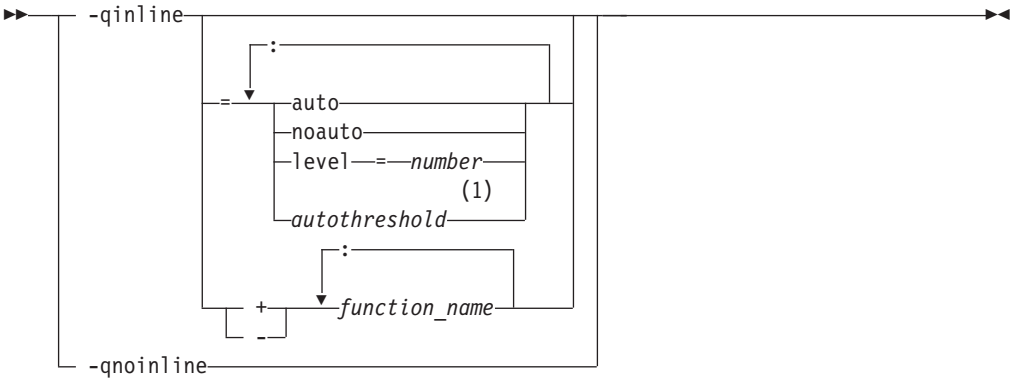
プラグマ同等物

なし。

目的

パフォーマンスを改善するために、関数への呼び出しを生成する代わりに、それらの関数のインライン化を試行する。

構文



注:

1 このサブオプションは、XL C コンパイラーでのみ使用可能です。

デフォルト

-qinline を指定しない場合、デフォルト・オプションは、**-O0** または **-qnoopt** 最適化レベルでは **-qnoinline** であり、**-O2** 以上の最適化レベルでは **-qinline=noauto:level=5** です。

-qinline をサブオプションなしで指定する場合、デフォルト・オプションは **-qinline=auto:level=5** です。

パラメーター

noauto | auto
自動インライン化を有効または無効にします。

level=*number*
インライン化の相対度数を表します。*number* の値は、0 以上 10 以下の正の整数にすることが必要です。*number* のデフォルト値は 5 です。*number* の値が大きくなるほど、コンパイラーはより積極的なインライン化を行います。

C *autothreshold*
関数をインライン化する場合に関数に組み込むことができる実行可能ステートメントの最大数を表します。 *autothreshold* の値は正の整数でなければなりません。*autothreshold* のデフォルト値は 20 です。値 0 を指定した場合、IBM always_inline または __always_inline__ 属性と共に指定された関数、IBM あるいは **-qinline+** の後に指定された関数のみがインライン化されます。以下の例では、3 つの実行可能ステートメントが increment 関数に組み込まれます。

```
int increment(){
  int a, b, i;
  for (i=0; i<10; i++){   // statement 1
    a=i;                  // statement 2
    b=i;                  // statement 3
  }
}
```

C

function_name

function_name が **-qinline+** オプションの後に指定されている場合、指定された関数はインライン化する必要があります。*function_name* が **-qinline-** オプションの後に指定されている場合、指定された関数はインライン化してはなりません。C++ *function_name* は、関数のマングル名でなければなりません。マングル関数名は、リスト・ファイル内にあります。C++

使用法

C++ **-qinline** C++ または **-qinline** を、C++ **-O** C++ 、**-O2**、**-O3**、**-O4**、または **-O5** のいずれかの最適化レベルと共に指定して、関数のインライン化を有効にできます。これには、`inline` 指定子で宣言されている C++ またはクラス宣言内で定義されている C++ 関数が含まれます。

-qinline が有効な場合、コンパイラーは、特定の関数をインライン化することによってパフォーマンスを向上可能かどうかを判別します。つまり、関数がインライン化に適切であるかどうかは、インライン化される呼び出し数の限度、および結果として生じるコード・サイズの増加量の限度という 2 つの要因に左右されます。したがって、関数のインライン化を使用可能にしても、関数がインライン化されるとは保証されません。

インライン化によって必ずしもランタイムのパフォーマンスが向上するわけではないため、使用するコードでこのオプションの効果を検査する必要があります。再帰的または相互に再帰的な関数はインライン化しないでください。

-qinline+<*function_name*> または **-qinline-**<*function_name*> オプションを使用すると、関数をインライン化する必要があるか、またはインライン化するべきでないかを指定できます。

IBM **-qinline-**<*function_name*> オプションは、`always_inline` または `__always_inline__` 属性よりも優先順位は高くなります。関数に `always_inline` または `__always_inline__` 属性と、**-qinline-**<*function_name*> オプションの両方を指定した場合も、関数はインライン化されません。IBM

-qnoinline を指定すると、**-qipa** オプションを指定した高水準最適化プログラムによって実行されたインライン化などのすべてのインライン化およびインラインとして明示的に宣言された関数が使用不可になります。ただし、**-qnoinline** オプションは、以下の関数のインライン化には影響しません。

- IBM `always_inline` または `__always_inline__` 属性を使用して指定された関数 IBM
- **-qinline+<function_name>** オプションを使用して指定された関数。

デバッグ情報を生成する目的で **-g** オプションを指定すると、**-qinline** のインライン化効果が抑制される場合があります。

コード・サイズを増加させる最適化を避ける目的で **-qcompact** オプションを指定すると、**-qinline** のインライン化効果が抑制される場合があります。

注:

- **-qinline** は、**-Q** およびそのサブオプションに代わるものです。

- **-Q**、**-Q!**、**-Q=***threshold*、 **-Q+***name*、および **-Q-***name* は、すべて推奨されないオプションとサブオプションです。
- **-qipa=inline** およびそれに関連したすべてのサブオプションは推奨されません。それらのすべてに代わるものは、**-qinline** です。

事前定義マクロ

なし。

例

例 1

関数が一切インライン化されないように myprogram.c をコンパイルするには、以下のコマンドを使用します。

```
xlc myprogram.c -O2 -qnoinline
```

ただし、myprogram.c の一部の関数が IBM always_inline または __always_inline__ attribute IBM 属性と共に指定されている場合、**-qnoinline** オプションはこれらの関数に対しては効果がなく、関数は引き続きインライン化されます。

自動インライン化を使用可能にしたい場合、auto サブオプションを使用します。

```
-O2 -qinline=auto
```

6 から 10 までのインライン化レベルを指定して、より積極的な自動インライン化を実現できます。例を以下に示します。

```
-O2 -qinline=auto:level=7
```

自動インライン化が既にデフォルトで使用可能に設定されており、インライン化レベル 7 を指定したい場合、次のように入力します。

```
-O2 -qinline=level=7
```

例 2

C myprogram.c に salary、taxes、expenses、および benefits 関数 がある場合、以下のコマンドを使用して myprogram.c をコンパイルし、これらの関数をインライン化します。

```
xlc myprogram.c -O2 -qinline+salary:taxes:expenses:benefits
```

関数 salary、taxes、expenses、および benefits をインライン化せずに myprogram.c をコンパイルしたい場合、以下のコマンドを入力します。

```
xlc myprogram.c -O2 -qinline-salary:taxes:expenses:benefits
```

自動インライン化を使用不可にし、**-qininline+** オプションを指定することで、特定の関数をインライン化することもできます。例えば、次のとおりです。

```
-O2 -qinline=noauto -qinline+salary:taxes:benefits
```

この場合、関数 salary、taxes、および benefits がインライン化されます。さらに、 IBM always_inline または __always_inline__ 属性IBM と共に指定

されているか、または inline 指定子を使用して宣言された関数もインライン化されます。他の関数はインライン化されません。

+ と - のサブオプションを互いに混用したり、他の **-qinline** サブオプションと混用したりすることはできません。例えば、以下のオプションは、無効なサブオプションの組み合わせです。

```
-qinline+increase-decrease    // invalid
-qinline=level=5+increase     // invalid
```

しかし、**-qinline** を別々に使用することはできます。例を以下に示します。

```
-qinline+increase -qinline-decrease -qinline=noauto:level=5
```

C

C++ では、**-qinline+** および **-qinline-** オプションは、オプションの後に実際の関数名ではなくマングル関数名を指定する必要がある点を除いて、例 2 と同じ方法で使用できます。 C++

関連情報

- 173 ページの『-g』
- 『-qipa』
- 279 ページの『-O、-qoptimize』
- 261 ページの『-qlist』
- 22 ページの『コンパイラー・リスト』
- 「*XL C/C++* ランゲージ・リファレンス」の『inline 関数指定子』
- 「*XL C/C++* ランゲージ・リファレンス」の『名前マングリング (C++ のみ)』
- 「*XL C/C++* ランゲージ・リファレンス」の 『always_inline 関数属性 (IBM 拡張)』
- 推奨されないコンパイラー・オプションのリストについては、『推奨されないオプション』を参照してください。

-qipa

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

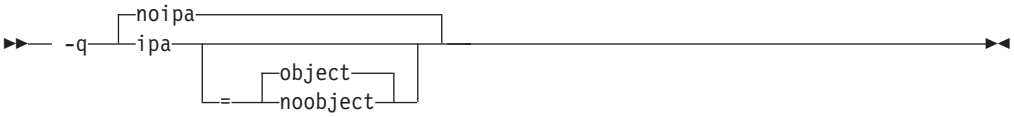
プロシージャー間分析 (IPA) と呼ばれる最適化のクラスを使用可能にしたり、カスタマイズする。

IPA は 2 ステップのプロセスです。コンパイル中に実行される最初のステップは、初期分析の実行およびプロシージャー間の分析情報のオブジェクト・ファイルへの格納で構成されます。リンク実行中に行われる 2 つ目のステップは、アプリケーション全体の完全な再コンパイルを促し、プログラム全体を最適化します。

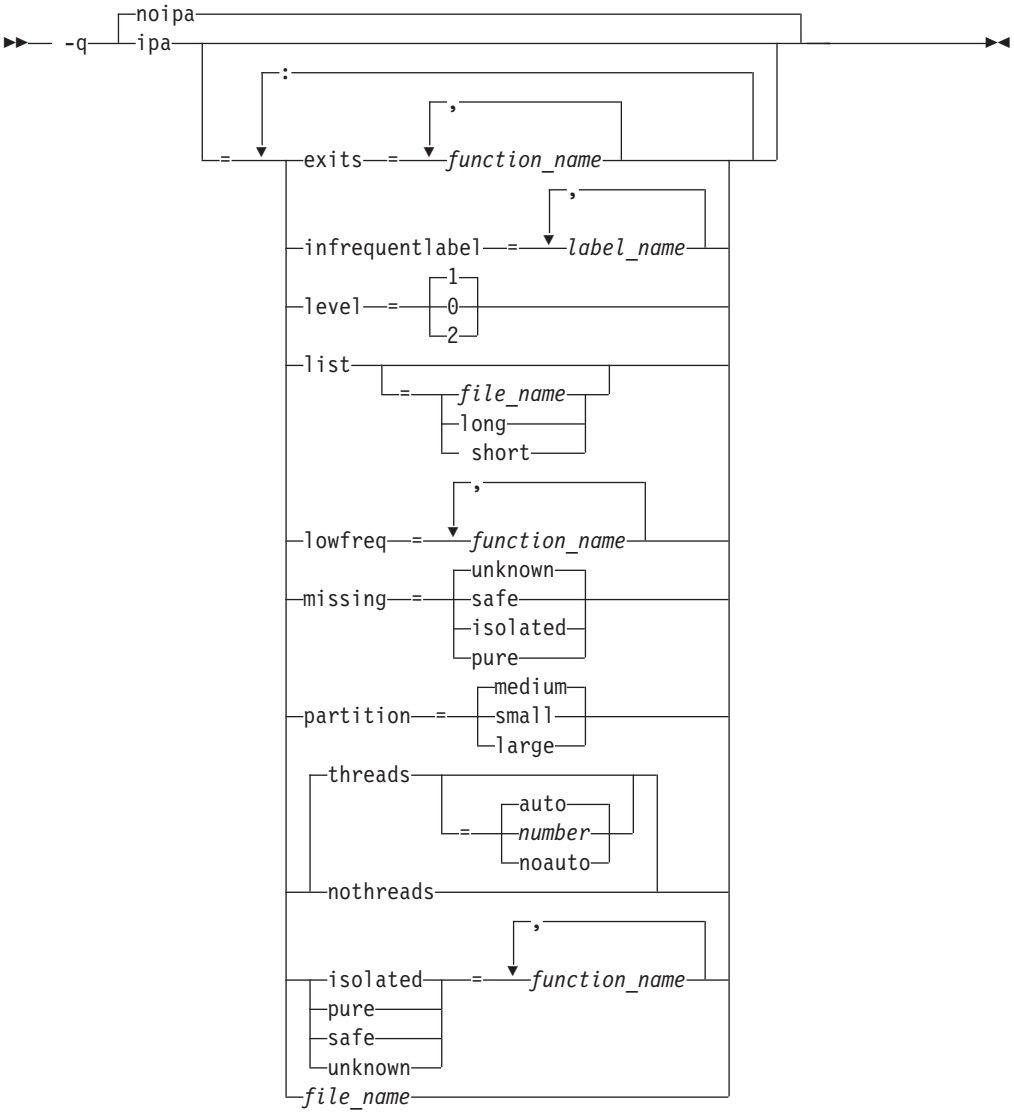
コンパイル・ステップまたはリンク・ステップ、あるいは両ステップの実行中に **-qipa** を使用できます。単一のコンパイラー呼び出しでコンパイルおよびリンクする場合、リンク時のサブオプションのみが関連性を持ちます。別のコンパイラー呼び出しでコンパイルおよびリンクする場合、コンパイル・ステップ実行中にはコンパイル時のサブオプションのみが関連性を持ち、リンク・ステップ実行中にはリンク時のサブオプションのみが関連性を持ちます。

構文

-qipa コンパイル時構文



-qipa リンク時構文



デフォルト

- **-qnoipa**

パラメーター

以下のパラメーターを指定できるのは、個別のコンパイル・ステップでのみです。

object | noobject
標準のオブジェクト・コードを出力オブジェクト・ファイルに入れるかどうかを指定します。

noobject を指定すると、最初の IPA フェーズ実行中にオブジェクト・コードが生成されないため、全体のコンパイル時間を大幅に短縮することができます。**noobject** で **-S** を指定する場合、**noobject** は無視されます。

コンパイルもリンクも同じステップで実行し、 **-S** もリスト・オプションもまったく指定されない場合は、 **-qipa=noobject** が暗黙指定されます。

コンパイル・ステップでサブオプションなしで **-qipa** を指定することは、**-qipa=object** と同等です。

以下のパラメーターは、同じコンパイラー呼び出しでの結合されたコンパイルおよびリンクのステップで、あるいは個別のリンク・ステップでのみ指定できます。

clonearch | noclonearch
このサブオプションは、もうサポートされていません。**-qtune=balanced** の使用を検討してください。

cloneproc | nocloneproc
このサブオプションは、もうサポートされていません。**-qtune=balanced** の使用を検討してください。

exits
プログラム出口を表す関数の名前を指定します。プログラム出口とは、決して戻ることができず、また IPA パス 1 でコンパイルされた関数を呼び出すことができない呼び出しのことです。呼び出しはプログラムに戻らないため、コンパイラーはこれらの関数への呼び出しを最適化できます (例えば、保存/復元シーケンスを除去して)。これらの関数は、**-qipa** でコンパイルされたプログラムの他の部分を呼び出すことはできません。

infrequentlabel
プログラムの実行中にまれに呼び出される可能性の高いユーザー定義ラベルを指定します。

label_name
ラベルの名前、またはコンマ区切りされたラベルのリストです。

isolated
-qipa でコンパイルされないコンマ区切りされた関数リストを指定します。*isolated* または呼び出しチェーン内の関数として指定する関数は、グローバル変数を直接参照できません。

level
プロシージャー間分析の最適化レベルを指定します。有効なサブオプションは以下のとおりです。

0 最小限のプロシージャー間分析および最適化しか行いません。

1 インライン化、限定された別名分析、および限定された呼び出し位置の調整を使用可能にします。

2 完全なプロシージャー間のデータ・フローおよび別名分析を実行します。

レベルを指定しないと、デフォルト値は 1 になります。

データ再編成情報を生成するには、最適化レベル **-qipa=level=2** または **-O5** を **-qreport** と一緒に指定します。IPA リンク・フェーズで、プログラム変数データのデータ再編成メッセージがリスト・ファイルのデータ再編成セクションに生成されます。再編成には、配列分割、配列転置、メモリー割り振りのマージ、配列インターリービング、および配列合体が含まれます。

list

リンク・フェーズ中にリスト・ファイルを生成するように指定します。リスト・ファイルには、 IPA によって実行される変換および分析をはじめ、区画ごとのオプションのオブジェクト・リストに関する情報が入ります。

list_file_name を指定しないと、リスト・ファイル名はデフォルトで a.lst になります。リスト・ファイルを生成する他のオプションと一緒に **-qipa=list** を指定すると、IPA はすべての既存 a.lst ファイルを上書きする a.lst ファイルを生成します。a.c という名前のソース・ファイルがあれば、IPA リストは通常のコンパイラー・リスト a.lst を上書きします。**-qipa=list=***list_file_name* サブオプションを使用すると、代わりのリスト・ファイル名を指定できます。

その他のサブオプションは、以下のサブオプションのいずれかです。

short リスト・ファイルの必要情報が少なくなります。 リストのオブジェクト・ファイル・マップ、ソース・ファイル・マップ、およびグローバル・シンボル・マップの各セクションが生成されます。

long リスト・ファイルの必要情報が多くなります。 **short** サブオプションで生成されるすべてのセクションに加えて、オブジェクト解決警告、オブジェクト参照マップ、インライナー・レポート、および区画マップのセクションが生成されます。

lowfreq

まれにしか呼び出されないと考えられる関数を指定します。これらは一般的に、エラー処理、トレース、または初期化の関数です。これらの関数の呼び出しの最適化を軽くすることによって、コンパイラーが、プログラムのその他の部分の実行を高速化できる場合があります。

missing

-qipa でコンパイルされず、**unknown**、**safe**、**isolated**、および **pure** サブオプションのいずれによっても明示的に指定されていない関数のプロシージャー間の振る舞いを指定します。

有効なサブオプションは、以下のサブオプションのいずれかです。

safe 欠落関数が、直接呼び出しか関数ポインターのいずれによっても可視の (欠落していない) 関数を間接的に呼び出さないことを指定します。

isolated
欠落関数が、可視の関数にアクセスできるグローバル変数を直接参照しないことを指定します。共有ライブラリーからバインドされた関数は*分離された* ものと見なされます。

pure 欠落関数は、*安全 (safe)* かつ*分離されて* おり、可視の関数へアクセスできるストレージを間接的に変更しないことを指定します。また、*純粋な (pure)* 関数には、監視できる内部状態はありません。

unknown
欠落関数が*安全 (safe)*、*分離された*、または*純粋な (pure)* ものとして認識されないことを指定します。このサブオプションは、欠落関数の呼び出しに対するプロシージャー間の最適化の量を大幅に制限します。

デフォルトでは **unknown** と想定されます。

partition
受け渡し 2 の間に IPA によって作成された各プログラム区画のサイズを指定します。有効なサブオプションは、以下のサブオプションのいずれかです。

- **small**
- **medium**
- **large**

より大きな区画には、より多くの関数を組み込めるため、プロシージャー間分析が向上しますが、最適化するにはさらに大きなストレージが必要になります。ページングのためにコンパイルに時間がかかりすぎる場合は、区画サイズを縮小してください。

pure
-qipa でコンパイルされない*純粋な (pure)* 関数を指定します。*純粋 (pure)* として指定された関数は、すべて*分離された* および*安全な (safe)* 関数でなければならず、内部状態を変更したり、副次作用を持つことはできません (副次作用とは呼び出し元から可視のデータを変更する可能性があることと定義されています)。

safe
-qipa を使用してコンパイルされず、プログラムの他の部分を呼び出さない*安全な (safe)* 関数を指定します。安全な関数は、グローバル変数を変更できますが、 **-qipa** でコンパイルされた関数を呼び出すことはできません。

threads | nothreads
パス 2 中の IPA 最適化プロセスの一部を並列スレッドで実行します。これによって、マルチプロセッサー・システムのコンパイル・プロセスを高速化できます。 **threads** サブオプションの有効なサブオプションは、以下のサブオプションのいずれかです。

auto | noauto
auto が有効な場合、コンパイラーは、マシン負荷に基づいてヒューリスティックに複数のスレッドを選択します。 **noauto** が有効な場合、コンパイラーは、マシン・プロセッサー 1 つにつき 1 つのスレッドを作成します。

number
特定数のスレッドを使用するようコンパイラーに命令します。 *number* 1 か

ら 32 767 までの範囲の整数値に設定できます。ただし、*number* は事実上、システムで使用可能なプロセッサーの数に限定されます。

サブオプションなしの **threads** は **-qipa=threads=auto** を暗黙指定します。

unknown

-qipa でコンパイルされない*不明な (unknown)* 関数を指定します。*不明 (unknown)* として指定された関数は、**-qipa** でコンパイルされたプログラムの他の部分を呼び出したり、グローバル変数を変更できます。

file_name

特別な形式のサブオプション情報を含むファイルの名前を指定します。

ファイル・フォーマットは以下のとおりです。

```
# ... comment
attribute{, attribute} = name{, name}
missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
list [ = file-name | short | long ]
level = 0 | 1 | 2
partition = small | medium | large
```

ここで、*attribute* は以下のいずれかです。

- exits
- lowfreq
- unknown
- safe
- isolated
- pure

注:

- **-qipa=inline** およびそれに関連したすべてのサブオプションは推奨されません。それらのすべてに代わるものは、**-qinline** です。詳しくは、207 ページの『-qinline』、および 101 ページの『推奨されないオプション』を参照してください。
- コンパイラーのバージョン 9.0 のリリースでは、**pdfname** サブオプションは推奨されません。新しいアプリケーションでは **-qpdf1=*pdfname*** または **-qpdf2=*pdfname*** を使用してください。詳しくは、『 292 ページの『-qpdf1、-qpdf2』』を参照してください。

使用法

-qipa を指定すると、最適化レベルが **-O2** に自動的に設定されます。さらにパフォーマンスを高めるために、**-qinline** オプションも指定できます。 **-qipa** オプションは、最適化の実行中に検査される領域、単一関数から複数関数s (ソース・ファイルが異なる可能性あり) へのインライン化とそれらの間のリンクを継承します。

-qipa でリンクする際に使用されるオブジェクト・ファイルのいずれかが **-qipa=noobject** オプションで作成された場合は、エントリー・ポイント (実行可能プログラムのメインプログラム、またはライブラリー用にエクスポートされた任意の関数) を含むファイルをすべて **-qipa** でコンパイルしなければなりません。

異なるリリースのコンパイラーで作成されたオブジェクトをリンクすることはできますが、少なくとも、リンクするオブジェクトの作成に使用した新しい方のコンパイラーと同じリリース・レベルのリンカーを使用しなければなりません。

明らかに参照されていたり、ソース・コードで設定されているシンボルは、IPA によって最適化される可能性があり、**debug** または **nm** 出力へと見失う可能性があります。 **-g** コンパイラーとともに IPA を使用すると、その結果、通常、非ステップ可能出力となります。

-# を使用して **-qipa** を指定すると、コンパイラーは IPA リンク・ステップに続くリンカー情報を表示しません。

-qipa の使用に関する推奨手順については、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』を参照してください。

事前定義マクロ

なし。

例

以下の例で、ファイル・セットをプロシージャー間分析でコンパイルする方法を示します。

```
xlc -c *.c -qipa
xlc -o product *.o -qipa
```

同じファイルのセットをコンパイルして、2 番目のコンパイルの最適化と最初のコンパイル・ステップの速度を改善する方法を以下に示します。ルーチン・セット `user_trace1`、`user_trace2`、および `user_trace3` が存在すると想定します。これらはほとんど実行されることがなく、`user_abort` ルーチンがプログラムを終了します。

```
xlc -c *.c -qipa=noobject
xlc -c *.o -qipa=lowfreq=user_trace[123]:exit=user_abort
```

関連情報

- 207 ページの『-qinline』
- 『-qisolated_call』
- 259 ページの『-qlibmpi』
- 418 ページの『#pragma execution_frequency』
- -qpdf1、-qpdf2
- 324 ページの『-S』
- 推奨されないオプション
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』
- ランタイム環境変数

-qisolated_call

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options isolated_call、#pragma isolated_call

目的

パラメーターに暗黙指定されるもの以外の副次作用がない関数をソース・ファイルで指定する。

基本的には、ランタイム環境の状態における変更は、以下のような副次作用と見なされます。

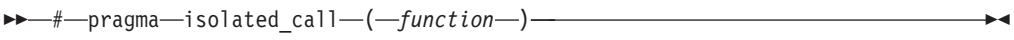
- 揮発性オブジェクトにアクセスする場合
- 外部オブジェクトを変更する場合
- 静的オブジェクトを変更する場合
- ファイルを変更する場合
- 別のプロセスまたはスレッドにより変更されるファイルにアクセスする場合
- 戻る前に解放されない限り、動的オブジェクトを割り当てる場合
- 同じ呼び出し中に割り当てられていない限り、動的オブジェクトを解放する場合
- 丸めモードまたは例外処理などの、システム状態を変更する場合
- 上記のいずれかを行う関数を呼び出す場合

関数を isolated としてマーク付けすると、呼び出された関数によって外部変数および静的変数を変更できないことと、必要に応じてストレージへの不正な参照を呼び出し関数から削除できることが最適化プログラムに通知されます。命令はより自由にリオーダーでき、その結果、パイプラインの遅延が少なくなり、プロセッサーの実行が速くなります。同じパラメーターを指定している同じ関数に対する複数の呼び出しを結合することが可能で、結果が不要であれば、呼び出しを削除することができて、呼び出し順序を変更することができます。

構文

オプション構文

プラグマ構文



デフォルト

適用されません。

パラメーター

function
副次作用がない関数、あるいは副次作用がある関数や処理に依存しない関数の名

前です。*function* は 1 次式であり、ID 、演算子関数、変換関数、または修飾名としても使用できます。ID は、型関数または typedef 関数でなければなりません。 C++ 名前が多重定義関数を参照する場合、この関数のすべての可変部は、分離された (isolated) 呼び出しとしてマークされています。

使用法

オプションまたはプラグマで指定された関数に許可される唯一の副次作用は、その関数に渡されるポインター引数 (つまり参照による呼び出し) によってポイントされるストレージを変更することです。 この関数は、不揮発性外部オブジェクトを検査することが許可され、ランタイム環境の不揮発性状態に依存する結果を戻します。自身を呼び出す関数、またはローカル静的ストレージに依存するなどのその他の副次作用の原因になる関数は指定しないでください。関数が副次作用を持っていないと誤って識別されると、結果のプログラムの振る舞いは予期しないものとなり、誤った結果が生み出される可能性があります。

#pragma options isolated_call ディレクティブは、すべてのステートメントの前の、ソース・ファイルの先頭に指定する必要があります。**#pragma isolated_call** ディレクティブは、プラグマで指定された関数への呼び出しの前後ならば、ソース・ファイル内のどこにでも指定できます。

-qignprag コンパイラー・オプションを指定すると、別名割り当てプラグマが無視されます。**-qignprag** を使用すると、**#pragma isolated_call** ディレクティブを含むアプリケーションをデバッグできます。

事前定義マクロ

なし。

例

関数 myfunction(int) と classfunction(double) に副次作用がないことを指定して、myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

以下の例では、**#pragma isolated_call** ディレクティブ (addmult 関数に対して) の使用方法を示します。また、このディレクティブを使用しない場合も示します (same 関数および check 関数に対して)。

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This function is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
  int rslt;

  rslt = op1*op2 + op2;
  return rslt;
}

/* The function 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called. */
```

```
int same(double op1, double op2) {
  static double delta = 1.0;
  double temp;

  temp = (op1-op2)/op1;
  if (fabs(temp) < delta)
    return 1;
  else {
    delta = delta / 2;
    return 0;
  }
}

/* The function 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output. */
int check(int op1, int op2) {
  if (op1 < op2)
    return -1;
  if (op1 > op2)
    return 1;
  printf("Operands are the same.¥n");
  return 0;
}
```

関連情報

- 190 ページの『-qignprag』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『const 関数属性』 および 『pure 関数属性』

-qkeepinlines (C++ のみ)

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

参照されていない `extern` インライン関数の定義を保持または破棄する。

-qnokeepinlines が有効な場合、コンパイラーは、参照されない外部インライン関数の定義を破棄します。**-qkeepinlines** が有効な場合、コンパイラーは、参照されない外部インライン関数の定義を保持します。

構文

デフォルト

-qnokeepinlines

パラメーター

exports
エクスポート・リストに存在するインライン関数をコンパイラーが破棄しないようにします。

使用法

-qnokeepinlines はオブジェクト・ファイルのサイズを縮小します。 **-qkeepinlines** は、v5.0.2.1 更新レベルより前の VisualAge® C++ コンパイラーと同じ振る舞いを提供するため、共有ライブラリーと旧リリースのコンパイラーで作成されたオブジェクト・ファイルとの間の互換性を保てます。

旧バージョンのコンパイラーを使用して作成されたシンボルとその定義のリストをコンパイラーで保持する場合、**-qkeepinlines=exports** を使用することで、プログラム・ファンクションのインライン化中にそれらのシンボルとその定義をコンパイラーが破棄しないようにすることができます。ただし、エクスポート・ファイルを指定していない場合、またはエクスポート・ファイルにシンボルが含まれていない場合、コンパイラーは **-qnokeepinlines** と同じオブジェクト・ファイルを生成します。

-qkeepinlines=exports を使用してプログラムをコンパイルする場合、**-bE** または **-bexport** オプションのいずれかを使用して、以下の例に示すように、エクスポートするシンボルが含まれるファイルを指定する必要があります。

```
xlC -qmkshrobj -qkeepinlines=exports -bE:file_name source_file
xlC -qmkshrobj -qkeepinlines=exports -bexport:file_name source_file
```

事前定義マクロ

なし。

関連情報

- 277 ページの『-qmkshrobj』
- 共有ライブラリーの作成について詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『共有ライブラリーのコンパイル』のセクションを参照してください。

-qkeepparm

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

-O2 以上の最適化で使用した場合に、プロシージャー・パラメーターをスタックに保管するかどうかを指定します。

関数は通常、その着信パラメーターをスタックの入り口点に保管します。ただし、最適化オプションを使用可能にしてコードをコンパイルすると、コンパイラーはス

タックからこれらのパラメーターを除去した方が最適化の利点があると考えた場合、これらのパラメーターを除去します。 **-qkeepparm** が有効な場合、最適化が有効であってもパラメーターがスタックに保管されます。**-qnokeepparm** が有効な場合、最適化に効果的であれば、パラメーターはスタックから除去されます。

構文

デフォルト

-qnokeepparm

使用法

-qkeepparm を指定すると、着信パラメーターの値をスタックに保存することで、デバッガーなどのツールがこれらの値を使用できます。しかし、これによりアプリケーションのパフォーマンスに悪影響が及ぶことがあります。

事前定義マクロ

なし。

関連情報

- 279 ページの『-O、-qoptimize』

-qkeyword

カテゴリー

言語エレメント制御

プラグマ同等物

なし

目的

指定された名前がプログラム・ソースに現れたときに、それをキーワードとして処理するかまたは ID として処理するかを制御する。

構文

デフォルト

デフォルトでは、C および C++ 言語標準に定義されている組み込みキーワードはすべてキーワードとして予約されています。

使用法

このオプションを使用しても、キーワードを言語に追加することはできません。しかし、**-qnokeyword=***keyword_name* を使用すると、組み込みキーワードを使用不可にでき、**-qkeyword=***keyword_name* を使用すると、これらのキーワードを復元できます。

C++ このオプションは、すべての C++ 組み込みキーワードで使用できます。
C++

C++11 このオプションは、C++11 標準によって導入された以下のキーワードと共に使用できます。

表 25. C++11 標準で導入されたキーワード

キーワード	機能	-qlanglvl サブオプション
constexpr	『一般化された定数式 (C++11)』	**-qlanglvl=[no]constexpr**
decltype	decltype(expression) 型指定子 (C++11)	**-qlanglvl=[no]decltype**
nullptr	nullptr (C++11)	**-qlanglvl=nullptr**
static_assert	『static_assert 宣言 (C++11)』	**-qlanglvl=[no]static_assert**

これらのキーワードは、デフォルトではすべての C++ 言語レベルで予約されています。各機能の **-qlanglvl** サブオプションまたは **-qlanglvl=[no]extended0x** グループ・オプションは、機能を有効または無効にできるのみであり、関連キーワードを有効または無効にすることはできません。 **-qkeyword=***keyword_name* オプションを指定すると、コンパイラーは *keyword_name* (constexpr、decltype、nullptr、または static_assert) をキーワードとして予約します。ただし関連機能が自動的に有効になることはありません。

以下の表は、さまざまなオプション設定が有効である場合のコンパイラーの動作を示しています。

表 26. 異なるオプション設定でのコンパイラーの動作

	-qkeyword=*keyword_name* **が有効な場合**	**-qnokeyword=***keyword_name* **が有効な場合**
-qlanglvl サブオプションまたはグループ・オプションが有効になっている	機能は使用可能です。*keyword_name* はキーワードとして予約されます。	機能は使用可能です。*keyword_name* はキーワードとして予約されます。**-qnokeyword=***keyword_name* オプションが無視されたことを示す警告メッセージが出されます。

表 26. 異なるオプション設定でのコンパイラーの動作 (続き)

	-qkeyword=*keyword_name* **が有効な場合**	**-qnokeyword=***keyword_name* **が有効な場合**
-qlanglvl サブオプションおよびグループ・オプションのどちらも有効になっていない	機能は使用不可です。 *keyword_name* はキーワードとして予約されます。 *keyword_name* キーワードが予約されており、それを使用すべきでないが、機能は使用不可であるというコンテキストでは、エラー・メッセージが出されます。 **-qlanglvl** サブオプションまたはグループ・オプションを使用して機能を使用可能にするか、 **-qnokeyword=***keyword_name* オプションを使用してキーワードを使用不可にすることができます。	機能は使用不可です。 *keyword_name* は ID トークンとして扱われます。 **-qwarn0x** オプションが有効な場合、機能をオンにするとプログラムの妥当性またはセマンティクスが変更されることが予期される場合は、警告メッセージが出されます。

C++11

C このオプションは、以下の C キーワードにも使用できます。

- asm
- inline
- restrict
- typeof

注: asm は、**stdc89** または **stdc99** 言語レベルではキーワードとして予約されません。

C

事前定義マクロ

- C++ __BOOL__ はデフォルトで 1 に定義されています。しかし、**-qnokeyword=bool** が有効な場合は未定義です。
- C **-qkeyword=inline** が有効な場合、__C99_INLINE は 1 に定義されています。
- **-qkeyword=restrict** が有効な場合、__C99_RESTRICT は 1 に定義されています。
- C **-qkeyword=asm** が有効な場合、__IBM_GCC_ASM は 1 に定義されています。 (C++ では、これはデフォルトで定義されています。)
- **-qkeyword=typeof** が有効な場合、__IBM__TYPEOF__ は 1 に定義されています。

例

C++ 以下の呼び出しを使用すると bool を復元できます。

```
xlc++ -qkeyword=bool
```

C++

C 以下の呼び出しを使用すると typeof を復元できます。

```
xlc -qkeyword=typeof
```

C

関連情報

- 395 ページの『-qwarn0x (C++11)』

-l

カテゴリー

リンク

プラグマ同等物

なし。

目的

指定されたライブラリー・ファイル lib*key*.so があるかどうかを検索してから、動的リンクの場合は lib *key*.a、静的リンクの場合は単に lib*key*.a のみがあるかどうかを検索する。

構文

デフォルト

コンパイラーのデフォルトでは、幾つかのコンパイラー・ランタイム・ライブラリーだけを検索します。デフォルトの構成ファイルは **-l** コンパイラー・オプションを使用してデフォルトのライブラリー名を指定し、**-L** コンパイラー・オプションを使用してライブラリーのデフォルト検索パスを指定します。

C および C++ ランタイム・ライブラリーは自動的に追加されます。

パラメーター

key
lib 文字を除去したライブラリー名です。

使用法

デフォルトの検索パスに入っていないライブラリーに対する追加の検索パス情報も提供する必要があります。検索パスは、**-L** オプションを使用して変更できます。

-l オプションは累積されます。コマンド行で **-l** オプションが後に現れた場合は、先に出現した **-l** によって指定されたライブラリーのリストを置換するのではなく、そのリストへの追加を行います。ライブラリーはコマンド行に現れた順番で検索されるため、ライブラリーを指定する順序はアプリケーションのシンボル解決に影響する可能性があります。

詳しくは、オペレーティング・システムの **ld** に関する資料を参照してください。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、`/usr/mylibdir` ディレクトリーにあるライブラリー `mylibrary (libmylibrary.a)` でリンクするには、以下のように入力します。

```
xlc myprogram.c -lmylibrary -L/usr/mylibdir
```

関連情報

- 『-L』
- 8 ページの『構成ファイルでのコンパイラー・オプションの指定』

-L

カテゴリー

リンク

プラグマ同等物

なし。

目的

-l オプションによって指定されたライブラリー・ファイルのディレクトリー・パスをリンク時に検索する。

構文

デフォルト

デフォルトでは、標準のディレクトリーしか検索されません。デフォルトで設定されるディレクトリーについては、コンパイラー構成ファイルを参照してください。

パラメーター

directory_path
ライブラリー・ファイルを検索するディレクトリーのパスです。

使用法

-L コンパイラー・オプションで指定されたパスは、リンク時のみに検索されます。実行時に検索するパスを指定するには、**-R** オプションを使用します。

-L*directory* オプションが構成ファイルとコマンド行の両方に指定されている場合は、構成ファイルに指定された検索パスがリンク時に最初に検索されます。

詳しくは、オペレーティング・システムの **ld** に関する資料を参照してください。

事前定義マクロ

なし。

例

`/usr/tmp/old` ディレクトリーで `libspfiles.a` ライブラリーが検索されるよう `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -lspfiles -L/usr/tmp/old
```

関連情報

- 225 ページの『-l』
- 314 ページの『-R』

-qlanglvl

このトピックには、以下の情報が含まれています。

- 『カテゴリー』
- 『プラグマ同等物』
- 『目的』
- 228 ページの『構文』
- 228 ページの『デフォルト』
- 231 ページの『C 言語プログラム向けのパラメーター』
- 233 ページの『C++ 言語プログラム向けのパラメーター』
- 249 ページの『使用法』
- 249 ページの『事前定義マクロ』

カテゴリー

言語エレメント制御

プラグマ同等物

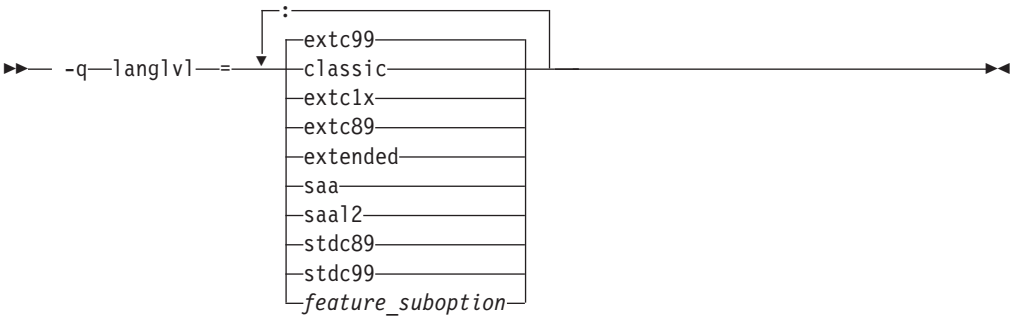
C #pragma options langlvl、#pragma langlvl

目的

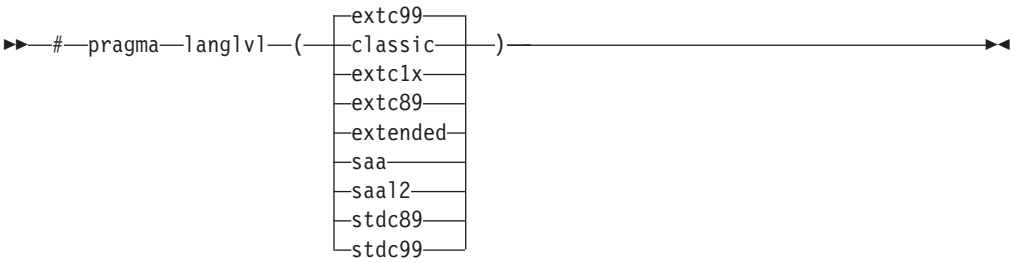
ソース・コードおよびコンパイラー・オプションが固有の言語標準、または標準のサブセットまたはスーパーセットに準拠しているかを検査するかどうかを判別する。

構文

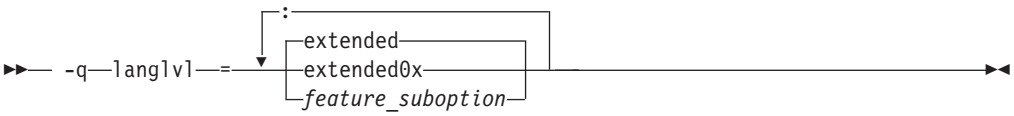
-qlanglvl 構文 — C



#pragma langlvl 構文 — C のみ



-qlanglvl 構文 — C++



デフォルト

- C コンパイラーの呼び出しに使用されるコマンドに合わせて、以下のようにデフォルトが設定されます。
  - **xlc** および関連する呼び出しコマンドの場合は **-qlanglvl=extc99:ucs**
  - **cc** および関連する呼び出しコマンドの場合は **-qlanglvl=extended:noucs**
  - **c89** および関連する呼び出しコマンドの場合は **-qlanglvl=stdc89:noucs**
  - **c99** および関連する呼び出しコマンドの場合は **-qlanglvl=stdc99:ucs**
- C++ コンパイラーの呼び出しに使用されるコマンドに合わせて、以下のようにデフォルトが設定されます。
  - **xlC** または **xlc++** および関連する呼び出しコマンドの場合は **-qlanglvl=extended**
  - 機能関連のサブオプション、およびさまざまな言語レベル (**compat366**、**extended** (C++)、および **extended0x**) でのサブオプションのデフォルトの設定は、『 229 ページの表 27』にリストされています。デフォルト設定「On」

は、そのサブオプションが使用可能に設定されていることを意味します。一方、デフォルト設定「Off」は、そのサブオプションが使用不可に設定されていることを意味します。

表 27. さまざまな言語レベルでのサブオプションのデフォルト設定

オプション	言語レベル		
	compat366	**extended (C++)**	**extended0x**
-qlanglvl=anonstruct | noanonstruct	Off	On	On
-qlanglvl=anonunion | noanonunion	On	On	On
-qlanglvl=ansifor | noansifor	Off	On	On
-qlanglvl=ansisinit | noansisinit	On	On	On
C++11 -qlanglvl=autotypededuction| noautotypededuction	Off	Off	On
-qlanglvl=c1xnoreturn | noc1xnoreturn	Off	On	On
-qlanglvl=c99__func__ | noc99__func__	Off	On	On
-qlanglvl=c99complex | noc99complex	Off	Off	Off
-qlanglvl=c99complexheader | noc99complexheader	Off	Off	Off
-qlanglvl=c99compoundliteral | noc99compoundliteral	Off	On	On
-qlanglvl=c99hexfloat | noc99hexfloat	Off	On	On
C++11 -qlanglvl=c99longlong | noc99longlong	Off	Off	On
C++11 -qlanglvl=c99preprocessor | noc99preprocessor	Off	Off	On
-qlanglvl=c99vla | noc99vla	Off	On	On
IBM -qlanglvl=compatrvaluebinding | nocompatrvaluebinding	Off	Off	Off
-qlanglvl=complexinit | nocomplexinit	Off	On	On
C++11 -qlanglvl=constexpr | noconstexpr	Off	Off	On
C++11 -qlanglvl=decltype | nodecltype	Off	Off	On
C++11 -qlanglvl=defaultanddelete | nodefaultanddelete	Off	Off	On
C++11 -qlanglvl=delegatingctors | nodelegatingctors	Off	Off	On
-qlanglvl=dependentbaselookup | nodependentbaselookup	On	On	Off
-qlanglvl=emptystruct | noemptystruct	On	On	On
C++11 -qlanglvl=explicitconversionoperators | noexplicitconversionoperators	Off	Off	On
C++11 -qlanglvl=extendedfriend | noextendedfriend	Off	Off	On

表 27. さまざまな言語レベルでのサブオプションのデフォルト設定 (続き)

オプション	言語レベル		
	compat366	**extended (C++)**	**extended0x**
C++11 IBM -qlanglvl=extendedintegersafe | noextendedintegersafe	Off	Off	Off
C++11 -qlanglvl=externtemplate | noexterntemplate	Off	On	On
-qlanglvl=FileScopeConstExternLinkage | noFileScopeConstExternLinkage	Off	Off	Off
C++11 -qlanglvl=inlinenamespace | noinlinenamespace	Off	Off	On
-qlanglvl=gnu_assert | nognu_assert	Off	On	On
-qlanglvl=gnu_complex | nognu_complex	Off	Off	Off
-qlanglvl=gnu_computedgoto | nognu_computedgoto	Off	On	On
-qlanglvl=gnu_explicitregvar | nognu_explicitregvar	Off	On	On
-qlanglvl=gnu_externtemplate | nognu_externtemplate	Off	On	On
-qlanglvl=gnu_labelvalue | nognu_labelvalue	Off	On	On
-qlanglvl=gnu_locallabel | nognu_locallabel	Off	On	On
-qlanglvl=gnu_include_next | nognu_include_next	On	On	On
-qlanglvl=gnu_membernamereuse | nognu_membernamereuse	Off	On	On
-qlanglvl=gnu_suffixij | nognu_suffixij	Off	On	On
-qlanglvl=gnu_varargmacros | nognu_varargmacros	Off	On	On
-qlanglvl=gnu_warning | nognu_warning	Off	On	On
-qlanglvl=illptom | noillptom	On	On	On
-qlanglvl=implicitint | noimplicitint	On	On	On
-qlanglvl=newexcp | nonewexcp	Off	Off	Off
C++11 -qlanglvl=nullptr | nonullptr	Off	Off	On
-qlanglvl=offsetnonpod | nooffsetnonpod	On	On	On
-qlanglvl=olddigraph | noolddigraph	Off	Off	Off
-qlanglvl=oldfriend | nooldfriend	On	On	Off
-qlanglvl=oldmath | nooldmath	On	Off	Off
-qlanglvl=oldtempacc | nooldtempacc	On	On	On
-qlanglvl=oldtmplalign | nooldtmplalign	On	Off	Off
-qlanglvl=oldtmplspec | nooldtmplspec	On	On	On
-qlanglvl=redefmac | noredefmac	Off	On	On
C++11 -qlanglvl=referencecollapsing | noreferencecollapsing	Off	Off	On
C++11 -qlanglvl=rightanglebracket | norightanglebracket	Off	Off	On

表 27. さまざまな言語レベルでのサブオプションのデフォルト設定 (続き)

オプション	言語レベル		
	compat366	**extended (C++)**	**extended0x**
C++11 -qlanglvl=rvaluereferences | norvaluereferences	Off	Off	On
C++11 -qlanglvl=scopedenum | noscopedenum	Off	Off	On
C++11 -qlanglvl=static_assert | nostatic_assert	Off	Off	On
IBM -qlanglvl=tempsaslocals | notempsaslocals	Off	Off	Off
IBM -qlanglvl=textafterendif | notextafterendif	Off	Off	Off
-qlanglvl=trailenum | notrailenum	On	On	On
-qlanglvl=typedefclass | notypedefclass	On	On	On
-qlanglvl=noucs | nonoucs	Off	Off	Off
-qlanglvl=varargmacros | novarargmacros	Off	On	On
C++11 -qlanglvl=variadic[templates] | novariadic[templates]	Off	Off	On
-qlanglvl=zeroextarray | nozeroextarray	Off	On	On

C 言語プログラム向けのパラメーター

C 以下に示すのは、C 言語プログラム向けの **-qlanglvl/#pragma langlvl** パラメーターです。

classic
非標準プログラムのコンパイルを許可し、K&R レベルのプリプロセッサーに厳密に準拠します。

詳しくは、『 249 ページの『classic 言語レベルとその他すべての標準ベースの言語レベルの相違点』』を参照してください。

C11

extc1x
コンパイルは C11 標準に基づき、現在サポートされている C11 の機能と、インプリメンテーション固有の言語拡張をすべて呼び出します。これらの C11 機能について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『C11 との互換性のための拡張機能』を参照してください。

注: IBM は、C11 (承認前は C1X と呼ばれていました) の選択された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルのインプリメンテーションは、標準に対する IBM の解釈に基づきます。新しい C11 標準ライブラリーのサポートを含め、C11 のすべての機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による C11 の機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースと

の互換性を維持するための試みは行いません。

C11

extc89
コンパイルは、ANSI C89 標準に準拠し、インプリメンテーション固有の言語拡張を受け入れます。

extc99
コンパイルは、ISO C99 標準に準拠し、インプリメンテーション固有の言語拡張を受け入れます。

extended
RT コンパイラーおよび **classic** との互換性を提供します。この言語レベルは C89 に基づいています。

saa
現行の SAA C CPI 言語定義に準拠するコンパイル。これは現在 SAA C レベル 2 です。

saa2
SAA C Level 2 CPI 言語定義に準拠するコンパイルですが、いくつかの例外があります。

stdc89
ANSI C89 標準に厳格に準拠するコンパイルで、ISO C90 とも呼ばれます。

stdc99
ISO C99 標準に厳格に準拠するコンパイルです。

個々の C 機能の **-qlanglvl** サブオプション・パラメーターを以下にリストします。

feature_suboption
構文図の中で、*feature_suboption* は C オプションをコロンで区切ったリストを表しています。それらは、以下のオプションのいずれかです。

注: 1 つの C 機能に複数の **-qlanglvl** グループ・オプションおよびサブオプションが指定されている場合は、最後の 1 つが有効になります。

IBM **textafterendif | notextafterendif**
`#endif` または `#else` の後に追加テキストを許可するコンパイラーから IBM XL C/C++ コンパイラーにコードを移植する場合に出力される警告メッセージを抑止するかどうかを指定します。デフォルト・オプションは **-qlanglvl=notextafterendif** であり、これは `#else` または `#endif` の後に余分なテキストがある場合にメッセージを出力することを示します。ただし、言語レベルが **classic** の場合、デフォルト・オプションは **-qlanglvl=textafterendif** です。これは、この言語レベルでは、`#else` または `#endif` の後の追加テキストが既に許可されていて、メッセージが生成されないためです。 IBM

ucs | noucs (オプションのみ)
ユニコード文字が、プログラム・ソース・コードの ID、ストリング・リテラル、および文字リテラルで許可されるかどうかを制御します。このサブオプションは、**stdc99** または **extc99** が有効な場合にデフォルトで使用可能です。ユニコード文字セットの詳細については、 「*XL C/C++ ランゲージ・リファレンス*」の『ユニコード規格』を参照してください。

以下の **-qlanglvl** サブオプションは、C コンパイラーによって受け入れられますが、無視されます。これらのサブオプションが暗黙指定する関数を使用可能にするには、**extended | extc99 | extc89** を使用してください。他の言語レベルの場合は、これらのサブオプションによって暗黙指定される関数は使用不可になります。

[no]gnu_assert
GNU C 移植性オプション。

[no]gnu_explicitregvar
GNU C 移植性オプション。

[no]gnu_include_next
GNU C 移植性オプション。

[no]gnu_locallabel
GNU C 移植性オプション。

[no]gnu_warning
GNU C 移植性オプション。

C++ 言語プログラム向けのパラメーター

C++ 以下に示すのは、対応する C++ 言語レベルの **-qlanglvl** グループ・オプション・パラメーターです。

extended
コンパイルは ISO C++ 標準に基づきますが、拡張言語フィーチャーの適応については若干の違いがあります。

C++11 **extended0x**
コンパイルは C++11 標準に基づき、ほとんどの C++ 機能および現在サポートされるすべての C++11 機能を呼び出します。 229 ページの表 27 に、サポートされる機能の詳細を示します。 C++11 機能について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『C++11 の互換性の拡張機能』を参照してください。

注: IBM は、C++11 (承認前は C++0x と呼ばれていました) の選択された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルのインプリメンテーションは、標準に対する IBM の解釈に基づきます。新しい C++11 標準ライブラリーのサポートを含め、C++11 のすべての機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による C++11 の新機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは行いません。

C++11

以下に、個々の C++ 機能の **-qlanglvl** サブオプション・パラメーターを示します。

feature_suboption
構文図の中で、*feature_suboption* は残りの C++ オプションをコロンで区切ったリストを表しています。それらは、以下のいずれかです。

注: 1 つの C++ 機能に複数の **-qlanglvl** グループ・オプションおよびサブオプションが指定されている場合は、最後の 1 つが有効になります。

anonstruct | noanonstruct

無名の構造体およびクラスのサポートを使用可能または使用不可にします。無名の構造体は、以下のコード・フラグメントにあるように、たいていの場合は共用体で使用されます。

```
union U {
   struct {
      int i:16;
      int j:16;
   };
   int k;
} u;
// ...
u.j=3;
```

デフォルトの **-qlanglvl=anonstruct** が有効である場合、無名の構造体がサポートされます。

これは C++ 標準の拡張で、Microsoft Visual C++ との互換性を持つ振る舞いを提供します。標準 C++ に準拠させる場合は、**-qlanglvl=noanonstruct** を指定してください。

anonunion | noanonunion

無名の共用体で許可されるメンバーを制御します。デフォルトの **-qlanglvl=anonunion** が有効な場合、無名の共用体は、標準 C++ が無名の共用体以外で許可するすべての型のメンバーを持つことができます。例えば、structure、typedef、および enumeration などの非データ・メンバーは許可されます。メンバー関数、仮想関数、または単純ではないデフォルト・コンストラクター、コピー・コンストラクター、またはデストラクターを持つクラスのオブジェクトは、このオプションの設定にかかわらず共用体のメンバーにはなれません。

これは標準 C++ の拡張で、VisualAge C++ の前のバージョンと先行商品、および Microsoft Visual C++ との互換性を持つように設計された振る舞いを提供します。標準 C++ に準拠させる場合は、**-qlanglvl=noanonunion** を指定してください。

ansifor | noansifor

C++ 標準に定義されているスコープ規則を、for ループ初期設定ステートメントで宣言されている名前に適用するかどうかを制御します。デフォルトの **-qlanglvl=ansifor** が有効な場合、標準 C++ の規則が使用され、以下のコードが名前ルックアップ・エラーを発生させます。

```
{
   //...
   for (int i=1; i<5; i++) {
      cout << i * 2 << endl;
   }
   i = 10;  // error
}
```

エラーの理由は、i あるいは for ループ初期設定ステートメント内で宣言されたいずれかの名前が、 for ステートメント内でのみ可視であるからです。エラーを訂正するには、i をループの外で宣言するか、あるいは **noansifor** を設定します。

-qlanglvl=noansifor が有効な場合、古い言語の振る舞いが使用されます。VisualAge C++ の前のバージョンと先行商品、および Microsoft Visual C++ との互換性を持たせるには、**-qlanglvl=noansifor** を指定します。

ansisinit | noansisinit
標準 C++ 規則が、グローバル・オブジェクトと静的オブジェクトの静的デストラクターの処理に適用できるかを制御します。デフォルトの **-qlanglvl=ansisinit** が有効な場合、標準規則が使用されます。

-qlanglvl=noansisinit が有効な場合、古い言語の振る舞いが使用されます。VisualAge C++ の前バージョンおよび先行製品との互換性を持たせるには **-qlanglvl=noansisinit** を指定します。

C++11 **autotypededuction | noautotypededuction**
auto 型推定機能を使用可能にするかどうかを制御します。**-qlanglvl=autotypededuction** オプションを指定した場合、auto 型推定機能は使用可能に設定されるので、変数の宣言時に型を指定する必要がなくなります。代わりに、コンパイラーが auto 変数の型を、その初期化指定子の式から推定します。

-qlanglvl=autotypededuction オプションを使用して、後置戻り型機能を制御することもできます。この機能は、以下の型のテンプレートおよび関数を宣言する場合に役に立ちます。

- 関数引数の型に応じた戻りの型を持つ関数テンプレートまたはクラス・テンプレートのメンバー関数
- 複雑な戻りの型を持つ関数またはクラスのメンバー関数
- 完全な転送関数

-qlanglvl=autotypededuction オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して auto 型推定機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noautotypededuction** です。

c1xnoreturn | noc1xnoreturn

`_Noreturn` 関数指定子のサポートを使用可能または使用不可にします。

-qlanglvl=c1xnoreturn オプションはグループ・オプション **-qlanglvl=extended** および **-qlanglvl=extended0x** に含まれているため、これらのグループ・オプションを使用して `_Noreturn` 関数指定子を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noc1xnoreturn** です。

c99__func__ | noc99__func__
C99 `__func__` ID のサポートを使用可能または使用不可にします。この機能の詳細については、「*XL C/C++ ランゲージ・リファレンス*」の『func_predefined ID』を参照してください。

c99complex | noc99complex
C99 複素数データ型と関連キーワードを使用可能または使用不可にします。

c99compoundliteral | noc99compoundliteral
C99 複合リテラルのサポートを使用可能または使用不可にします。

c99hexfloat | noc99hexfloat
C99 型の 16 進数浮動小数点定数のサポートを使用可能または使用不可にします。

C++11 **c99longlong | noc99longlong**
C99 long long 機能を使用可能にするかどうかを制御します。
-qlanglvl=c99longlong オプションを指定すると、C++ コンパイラーによって C99 long long 機能を使用することができます。これによって、C と C++ 言語間のソース互換性が向上します。

-qlanglvl=c99longlong オプションは、**-qlonglong** オプションと競合します。この 2 つのオプションを両方とも指定した場合、**-qlonglong** オプションは無視されます。**-qlonglong** オプションについて詳しくは、267 ページの『-qlonglong』を参照してください。

-qlanglvl=c99longlong オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して、C99 long long 機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noc99longlong** です。

C++11 **c99preprocessor | noc99preprocessor**
C++11 で採用された C99 プリプロセッサー機能を使用可能にするかどうかを制御します。**-qlanglvl=c99preprocessor** が有効である場合、C99 および C++11 コンパイラーによって、より一般的に使用されるプロセッサー・インターフェースを使用することができます。これにより、C ソース・ファイルを C++ コンパイラーに簡単に移植でき、プリプロセッサーの互換性の問題を回避できます。

デフォルト・オプションは **-qlanglvl=noc99preprocessor** です。

注: -qlanglvl=c99preprocessor を指定すると、暗黙的に **-qlanglvl=varargmacros** が設定されます。また、**-qlanglvl=noc99preprocessor** を指定すると、暗黙的に **-qlanglvl=novarargmacros** が設定されます。

-qlanglvl=c99preprocessor オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して C99 プリプロセッサー機能を使用可能にすることもできます。

c99vla | noc99vla
C99 型の可変長配列のサポートを使用可能または使用不可にします。

IBM **compatrvaluebinding | nocompatrvaluebinding**
C++ 標準 (2003) は、右辺値は不揮発性の const 左辺値参照にのみバインドできることを示しています。非準拠コンパイラーでは、非 const または揮発性の左辺値参照を右辺値にバインドすることが許可される場合があります。コードを IBM XL C/C++ コンパイラーに移植する場合は、このオプションを指定して、非 const または揮発性の左辺値参照を、初期化指定子が必要ではないユーザー定義型の右辺値にバインドすることを許可するようにコンパイラーに指示できます。 IBM

C++11 **-qlanglvl=compatrvaluebinding** と **-qlanglvl=rvaluereferences** の両方のオプションが有効になっている場合、コンパイラーはエラー・メッセージを出します。 C++11

C++ **complexinit | nocomplexinit**

C++ コンパイラーが、C99 の複素数型の場合に C11 形式の初期化を使用するかどうかを制御します。

-qlanglvl=complexinit オプションはグループ・オプション **-qlanglvl=extended** および **-qlanglvl=extended0x** に含まれているため、これらのグループ・オプションを使用して複素数型の初期化を有効にすることができます。この場合は、**-qlanglvl=c99complexheader** を指定して、コンパイラーが正しいヘッダー・ファイルを使用できるようにしてください。

デフォルト・オプションは **-qlanglvl=complexinit** です。 C++

C++11 **constexpr | noconstexpr**

一般化された定数式の機能を使用可能にするかどうかを制御します。**-qlanglvl=constexpr** オプションを指定すると、定数式の中で記述可能な式がコンパイラーによって拡張されます。定数式は、コンパイル時に評価できる式です。

-qlanglvl=constexpr オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して、一般化された定数式の機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noconstexpr** です。

C++11 **decltype | nodecltype**

decltype 機能を使用可能にするかどうかを制御します。この機能を使用すると、型に依存する可能性がある式の結果の型に基づいた型を取得できます。この機能を使用可能にするために、**-qlanglvl=decltype** オプションを指定できます。

-qlanglvl=decltype オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれているので、このグループ・オプションを使用して decltype 機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=nodecltype** です。

C++11 **defaultanddelete | nodefaultanddelete**

デフォルト関数および削除済み関数のフィーチャーを使用可能にするかどうかを制御します。このフィーチャーを使用すると、コンパイラーによって生成された実装が含まれる明示的なデフォルト関数を定義して、より高い効率を実現できます。また、コンパイラーによって使用が無効にされている削除済み関数を定義して、不要な関数の呼び出しを避けることもできます。この機能を使用可能にするために、**-qlanglvl=defaultanddelete** オプションを指定できます。

-qlanglvl=defaultanddelete オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれているので、このグループ・オプションを使用してこのフィーチャーを使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=nodefaultanddelete** です。

C++11 **delegatingctors | nodelegatingctors**

委任コンストラクター機能を使用可能にするかどうかを制御します。この機能を使用すると、共通の初期化と事後の初期化を 1 つのコンストラクターにまとめることができます。これによりプログラムが読みやすくなり、保守も容易になります。この機能を使用可能にするために、**-qlanglvl=delegatingctors** オプションを指定できます。

-qlanglvl=delegatingctors オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれているので、このグループ・オプションを使用して委任コンストラクター機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=nodelegatingctors** です。

DependentBaseLookup | noDependentBaseLookup

C++ 標準の Technical Corrigendum 1 (TC1) で定義された従属型のテンプレート基底クラスに対する名前ルックアップ規則が適用するかどうかを制御します。TC1 に準拠するには、**-qlanglvl=noDependentBaseLookup** を指定します。**-qlanglvl=noDependentBaseLookup** が有効である場合、テンプレート・クラス内の修飾されていない名前は、基底クラスがテンプレート・パラメーターに従属していると、その基底クラス内で解決されません。それらの名前を名前ルックアップで検出するためには、それらの名前を基底クラス名で修飾する必要があります。デフォルトの **-qlanglvl=DependentBaseLookup** が有効である場合、以前の XL C++ コンパイラーの動作がそのまま残されます。

注: C++11 言語レベルでのデフォルトのオプションは **-qlanglvl=noDependentBaseLookup** です。

以下の例は、**-qlanglvl=noDependentBaseLookup** を使用するとコンパイルされないコードを示しています。

```
struct base
{
          int baseName;
};

template <class B> struct derived : public B
{
   void func()
    {
    int i = baseName;     // this name will not be found in the base class
    };
};

int main(void)
{
   derived<base> x;
   x.func();
   return 0;
}
```

以下の例は、**-qlanglvl=nodependentbaselookup** を使用しても使用しなくてもコンパイルされるコードを示します。

```
struct base
{
          int baseName;
};

template <class B> struct derived : public B
{
   void func()
    {
    int i = B::baseName;   // qualified name will be found in the base class
    };
};

int main(void)
{
```

```
    derived<base> x;
    x.func();
    return 0;
}
```

empty_struct | noempty_struct
このオプションは、コンパイラーが構造体における空メンバー宣言を容認するよう指示します。構造体内での空のメンバー宣言は、許されません。例えば、**-qlanglvl=noemptystruct** が有効なとき、以下の例はコンパイラーによって拒否されます。

```
struct S {
      ;  // this line is ill-formed
};
```

デフォルトは **-qlanglvl=noemptystruct** です。

C++11 **explicitconversionoperators | noexplicitconversionoperators**
明示的な型変換演算子機能を使用可能にするかどうかを制御します。**-qlanglvl=explicitconversionoperators** オプションを指定すると、explicit 関数指定子をユーザー定義の型変換関数の定義に適用できます。これにより、ユーザー定義型変換関数による意図しない暗黙的な型変換を抑制できます。

-qlanglvl=explicitconversionoperators オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して、明示的な型変換演算子機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noexplicitconversionoperators** です。

C++11 **extendedfriend | noextendedfriend**
拡張フレンド宣言機能を使用可能にするかどうかを制御します。**-qlanglvl=extendedfriend** オプションを指定すると、フレンド宣言を統御する規則が以下のように緩和されます。

- テンプレート・パラメーター、typedef 名、および基本型をフレンドとして宣言できます。
- C++11 内でフレンド宣言のコンテキストにおけるクラス・キーが不要になります。

-qlanglvl=extendedfriend オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれているので、このグループ・オプションを使用して拡張フレンド宣言機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noextendedfriend** です。

注: **-qlanglvl=extendedfriend** は、**-qlanglvl=oldfriend** オプションとは両立しません。 **-qlanglvl=extendedfriend** が有効である場合、**-qlanglvl=oldfriend** オプションは無視され、**-qlanglvl=[no]oldfriend** の設定は **-qlanglvl=nooldfriend** になります。

C++11 IBM **extendedintegersafe | noextendedintegersafe**
このオプションを使用すると、u または U を含んだ接尾部を持たない 10 進整数リテラルを long long int 型によって表すことができない場合に、unsigned long long int 型を使用してそのリテラルを表すかどうかを決めることができます。

このオプションは、**-qlanglvl=c99longlong** オプションを指定した場合にのみ効果があり、それ以外の場合、コンパイラーはオプションが無視されることを示す警告メッセージを発行します。**-qlanglvl=c99longlong** と **-qlanglvl=extendedintegersafe** の両方のオプションを指定した場合、`u` または `U` を含んだ接尾部がない 10 進整数リテラルを `long long int` 型によって表すことができなければ、コンパイラーはリテラルの値が範囲外であることを述べたエラー・メッセージを発行します。

デフォルト・オプションは、すべての言語レベルで **-qlanglvl=noextendedintegersafe** です。

C++11 **externtemplate | noexterntemplate**

明示的インスタンス生成宣言機能を使用可能にするかどうかを制御します。この機能を使用すると、テンプレートの特殊化またはそのメンバーの暗黙のインスタンス生成を抑止できます。この機能を使用可能にするために、**-qlanglvl=externtemplate** オプションを指定することができます。これがデフォルト・オプションになります。

-qlanglvl=externtemplate オプションは **-qlanglvl=extended** および **-qlanglvl=extended0x** のグループ・オプションに含まれているので、これら 2 つのグループ・オプションを使用して、この機能を使用可能にすることができます。

以下の表に、**-qlanglvl=externtemplate** オプションとの相互作用があるオプションをリストします。

*表 28. **-qlanglvl=externtemplate** との相互作用があるオプション*

オプション	説明
-qtemplateregistry、-qtempinc	明示的インスタンス生成宣言は、有効のままです。明示的インスタンス生成宣言の対象である特殊化が参照されても、その特殊化が変換単位内の明示的インストール生成定義の対象でない場合は、その変換単位から、またはその変換単位のために、インスタンスが生成されることはありません。

以下の表に、**-qlanglvl=externtemplate** オプションとの相互作用がある IBM 言語拡張をリストします。

*表 29. **-qlanglvl=externtemplate** との相互作用がある IBM 言語拡張*

IBM 言語拡張	説明
#pragma instantiate	このプラグマは、意味としては明示的インスタンス生成定義と同じものです。
#pragma do_not_instantiate	このプラグマは、C++11 標準に導入された、明示的インスタンス生成宣言の機能のサブセットを提供します。これは、後方互換性の目的でのみ提供されています。新しいアプリケーションでは、明示的インスタンス生成宣言を使用することができます。
#pragma hashome、#pragma ishome	このプラグマを使用すると、特殊化の明示的インスタンス生成宣言とは関係なく、クラス・テンプレート特殊化の仮想関数テーブル (VFT) が生成されます。

-qlanglvl=[no]externtemplate オプションは、非推奨の **-qlanglvl=[no]gnu_externtemplate** オプションと置き換わります。アプリケーションでは、**-qlanglvl=[no]externtemplate** オプションを使用してください。

FileScopeConstExternLinkage | noFileScopeConstExternLinkage
静的または外部キーワードが指定されない場合に、定数変数のファイル・スコープが内部または外部のリンケージを持つかどうかを制御します。

-qlanglvl=FileScopeConstExternLinkage が有効である場合、すべてのファイル・スコープ定数変数は、外部から可視としてマーク付けされます。それ以外の場合、すべてのファイル・スコープ定数変数は、静的としてマーク付けされます。

デフォルトは **-qlanglvl=noFileScopeConstExternLinkage** です。

gnu_assert | nognu_assert
以下の GNU C システム識別アサーションのサポートを使用可能または使用不可にします。
- #assert
- #unassert
- #cpu
- #machine
- #system

gnu_complex | nognu_complex
GNU 複素数データ型と関連キーワードを有効または無効にします。

gnu_computedgoto | nognu_computedgoto
計算された goto ステートメントのサポートを使用可能または使用不可にします。

gnu_externtemplate | nognu_externtemplate
extern テンプレートのインスタンス化を使用可能または使用不可にします。この機能の詳細については、 「*XL C/C++ ランゲージ・リファレンス*」の『明示的インスタンス化』を参照してください。

注: オプション **-qlanglvl=[no]gnu_externtemplate** は XL C/C++ V13.1 では推奨されません。代わりに、オプション **-qlanglvl=[no]externtemplate** を使用できます。

gnu_include_next | nognu_include_next
GNU C #include_next プリプロセッサー・ディレクティブのサポートを使用可能または使用不可にします。

gnu_labelvalue | nognu_labelvalue
値としてのラベルのサポートを使用可能または使用不可にします。

gnu_locallabel | nognu_locallabel
ローカルで宣言されたラベルのサポートを使用可能または使用不可にします。

gnu_membernamereuse | nognu_membernamereuse
メンバー・リスト内のテンプレート名を typedef として再使用することを有効または無効にします。

gnu_suffixij | nognu_suffixij
GNU 型の複素数のサポートを使用可能または使用不可にします。**-qlanglvl=gnu_suffixij** が有効である場合、複素数は i/I または j/J のサフィックスで終了できます。

gnu_varargmacros | nognu_varargmacros
変数引数を持つ GNU 型のマクロのサポートを使用可能または使用不可にします。

この機能の詳細については、「*XL C/C++ ランゲージ・リファレンス*」の『Variadic マクロ拡張』を参照してください。

gnu_warning | nognu_warning
GNU C `#warning` プリプロセッサー・ディレクティブのサポートを使用可能または使用不可にします。

illptom | noillptom
メンバーへのポインター形成に使用できる式を制御します。デフォルトの **-qlanglvl=illptom** が有効な場合、XL C++ コンパイラーは、共通に使用されるが、C++ 標準に準拠していない幾つかの形式を受け入れることができます。例えば、以下のコードは、関数のメンバー `p` へのポインターを定義し、それを古い形式で `C::func` のアドレスへ初期設定します。

```
struct C {
void func(int);
};

void (C::*p) (int) = C::func;
```

これは標準 C++ の拡張で、VisualAge C++ の前のバージョンと先行商品、および Microsoft Visual C++ との互換性を持つように設計されている振る舞いを提供します。

標準 C++ に準拠させる場合は、**-qlanglvl=noillptom** を指定してください。上記のコード例では、`&` 演算子を使用するよう変更しなければなりません。

```
struct C {
void func(int);
};

void (C::*p) (int) = &C::func;
```

implicitint | noimplicitint
コンパイラーが欠落しているか部分的に指定されている型を暗黙指定の `int` として受け入れるかどうかを制御します。デフォルトの **-qlanglvl=implicitint** が有効な場合は、名前空間スコープでの関数宣言、またはメンバー・リスト内の関数宣言は、`int` を戻すよう暗黙的に宣言されます。また、型を完全に指定しない宣言指定子のシーケンスは、いずれも、整数型を暗黙的に指定します。あたかも `int` 指定子が存在しているかのような効果があります。

以下の指定子は、型を完全に指定しません。

- auto
- const
- extern
- extern "*literal*"
- inline
- mutable

- friend
- register
- static
- typedef
- virtual
- volatile
- プラットフォーム固有の型

C++11 C++11 では、auto をストレージ・クラス指定子として使用することはなくなりました。C++11 では、キーワード auto は型指定子として使用されます。コンパイラーは、auto 変数の型を、その初期化指定子の式から推定します。詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の、『auto 型指定子 (C++11)』を参照してください。 C++11

例えば、関数 MyFunction の戻りの型は、以下のコードで省略されたため、int です。

```
MyFunction()
{
   return 0;
}
```

型が指定されている状態は、いずれも、このサブオプションの影響を受けるので、注意してください。これには、例えば、テンプレート型およびパラメーター型、例外指定、式における型 (casts、dynamic_cast、new など)、および変換関数の型が含まれます。

これは標準 C++ の拡張で、VisualAge C++ の前のバージョンと先行製品、および Microsoft Visual C++ との互換性を持つように設計された振る舞いを提供します。

標準 C++ に準拠させる場合は、**-qlanglvl=noimplicitint** を指定してください。例えば、上記の関数宣言は以下のように変更する必要があります。

```
int MyFunction()
{
   return 0;
}
```

C++11 **inlinenamespace | noinlinenamespace**

インライン名前空間定義を使用可能にするかどうかを制御します。この定義は、初期の inline キーワードが前に指定されている名前空間定義です。そのように定義された名前空間は、インライン名前空間です。**-qlanglvl=inlinenamespace** オプションを指定すると、インライン名前空間のメンバーを、それを含んでいる名前空間のメンバーでもあるかのように定義および特殊化できます。

-qlanglvl=inlinenamespace オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれているので、このグループ・オプションを使用してインライン名前空間定義機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noinlinenamespace** です。

C++11 **nullptr | nonullptr**

nullptr 機能を使用可能にするかどうかを制御します。nullptr 値を持つ NULL ポインターは、任意のポインター型、pointer-to-member 型、またはブール型に変換できます。 nullptr 定数は、多重定義関数の整数 0 からは区別できます。

-qlanglvl=nullptr オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれています。このグループ・オプションを使用して、`nullptr` キーワード機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=nonullptr** です。

offsetnonpod | nooffsetnonpod

`offsetof` マクロをデータのみではないクラスへ適用できるかどうかを制御します。 C++ プログラマーはよくデータだけのクラスを「Plain Old Data」(POD) クラスと呼びます。デフォルトの **-qlanglvl=offsetnonpod** が有効である場合、以下のいずれか 1 つを含むクラスへ `offsetof` を適用できます。

- 暗黙的に宣言されているか、または C++11 明示的にデフォルトに設定された C++11 コンストラクターまたはデストラクター
- 暗黙的に宣言されているか、または C++11 明示的にデフォルトに設定された C++11 割り当て演算子
- private または protected 非静的データ・メンバー
- 基底クラス
- 仮想関数
- メンバーへの型ポインターの非静的データ・メンバー
- 非データ・メンバーを持つ構造体または共用体
- 参照

これは標準 C++ の拡張で、 VisualAge C++ for OS/2 3.0、VisualAge for C++ for Windows バージョン 3.5、および Microsoft Visual C++ と互換性を持つように設計された振る舞いを提供します。標準 C++ に準拠させる場合は、**-qlanglvl=nooffsetnonpod** を指定してください。

olddigraph | noolddigraph

古い型のダイグラフのサポートを使用可能または使用不可にします。デフォルトの **-qlanglvl=olddigraph** が有効である場合、古い形式のダイグラフはサポートされません。**-qlanglvl=olddigraph** が有効な場合は、以下のダイグラフがサポートされます。

ダイグラフ
結果の文字

%% # (ポンド記号)

%%%%
(ダブル・ポンド記号、プリプロセッサー・マクロの連結演算子として使用される)

標準 C++ および VisualAge C++ の前のバージョンおよび先行製品によってサポートされている拡張 C++ 言語レベルと互換性を持たせるには **-qlanglvl=noolddigraph** を指定します。

このサブオプションが有効なのは、**-qdigraphs** が有効な場合のみです。

oldfriend | nooldfriend

詳述されたクラス名なしでクラスを指定するフレンド宣言を C++ エラーとして取り扱うかどうかを制御します。デフォルトの **-qlanglvl=oldfriend** が有効な場合、キーワード `class` を含むクラス名を詳述せずにフレンド・クラスを宣言できます。例えば、下記のステートメントは、クラス `IFont` がフレンド・クラスになるように宣言します。

```
friend IFont;
```

これは標準 C++ の拡張で、VisualAge C++ の前のバージョンと先行製品、および Microsoft Visual C++ との互換性を持つように設計された振る舞いを提供します。

標準 C++ に準拠させる場合は、**-qlanglvl=nooldfriend** を指定してください。上記の宣言の例では、以下のように変更しなければなりません。

```
friend class IFont;
```

注: -qlanglvl=oldfriend は **-qlanglvl=extendedfriend** オプションとは両立しません。 **-qlanglvl=extendedfriend** が有効である場合、**-qlanglvl=oldfriend** オプションは無視され、**-qlanglvl=[no]oldfriend** の設定は **-qlanglvl=nooldfriend** になります。

oldtempacc | nooldtempacc

一時オブジェクトの作成が回避される場合でも、一時オブジェクトの作成のためのコピー・コンストラクターへのアクセスを常に検査するかどうかを制御します。デフォルトの **-qlanglvl=oldtempacc** が有効である場合、アクセスの確認がサポートされます。

これは標準 C++ の拡張で、 VisualAge C++ for OS/2 3.0、 VisualAge for C++ for Windows バージョン 3.5、および Microsoft Visual C++ と互換性を持つように設計された振る舞いを提供します。標準 C++ に準拠する場合は、**-qlanglvl=nooldtempacc** を指定してください。例えば、以下のコードの `throw` ステートメントは、コピー・コンストラクターがクラス C の protected メンバーであるため、エラーの原因となります。

```
class C {
public:
   C(char *);
protected:
   C(const C&);
};

C func() {return C("test");} // return copy of C object

void f()
{
// catch and throw both make implicit copies of
// the throw object
   throw C("error");    // throw a copy of a C object
   const C& r = func(); // use the copy of a C object
//                              created by func()
}
```

上記のコード例では、3 個所でコピー・コンストラクター `C(const C&)` の誤った形式が使用されています。

oldtmplalign | nooldtmplalign

ネスト化されたテンプレートに対して指定された位置合わせ規則が無視されるかどうかを制御します。デフォルトの **-qlanglvl=nooldtmplalign** が有効な場合、これらの位置合わせ規則は無視されません。例えば、次のテンプレートでは、`A<char>::B` のサイズは **-qlanglvl=nooldtmplalign** の場合は 5、**-qlanglvl=oldtmplalign** の場合は 8 となります。

```
template <class T>
struct A {
#pragma options align=packed
```

```
 struct B {
  T m;
  int m2;
 };
#pragma options align=reset
};
```

VisualAge for C++ V4.0 および先行製品との互換性を持たせるには、**-qlanglvl=oldtmplalign** を指定します。

oldtmplspec | nooldtmplspec

C++ 標準に準拠していないテンプレートの特殊化を許可するかどうかを制御します。デフォルトの **-qlanglvl=oldtmplspec** が有効な場合、以下の例でテンプレート・クラス ribbon を char 型に特殊化するように、テンプレート・クラスを明示的に特殊化できます。

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

これは標準 C++ の拡張で、 VisualAge C++ for OS/2 3.0、VisualAge for C++ for Windows、V3.5、および Microsoft Visual C++ との互換性を持つように設計された振る舞いを提供します。

標準 C++ に準拠させる場合は、**-qlanglvl=nooldtmplspec** を指定してください。上記の例では、テンプレートの特殊化は以下のように変更する必要があります。

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

redefmac | noredefmac

事前の #undef または undefine() ステートメントなしでマクロを再定義できるかどうかを制御します。

C++11 **referencecollapsing | noreferencecollapsing**

参照の縮約 (reference collapsing) 機能を使用可能にするかどうかを制御します。この機能を使用可能にするためには、**-qlanglvl=referencecollapsing** オプションを指定します。

-qlanglvl=referencecollapsing オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して参照の縮約 (reference collapsing) 機能を使用可能にすることもできます。

-qlanglvl=rvaluereferences オプションが有効になっていれば、**-qlanglvl=referencecollapsing** オプションが有効になっていない場合でも、コンパイラーは **-qlanglvl=referencecollapsing** オプションが指定されている場合と同様に動作します。

デフォルト・オプションは **-qlanglvl=noreferencecollapsing** です。

C++11 **rightanglebracket | norightanglebracket**

右不等号括弧機能を使用可能にするかどうかを制御します。**-qlanglvl=rightanglebracket** オプションを指定すると、この機能を使用可能にすることができます。

-qlanglvl=rightanglebracket オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して右不等号括弧機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=norightanglebracket** です。

C++11 **rvaluereferences | norvaluereferences**

右辺値参照機能を使用可能にするかどうかを制御します。この機能を使用可能にするには、**-qlanglvl=rvaluereferences** オプションを指定します。

-qlanglvl=rvaluereferences オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して右辺値参照機能を使用可能にすることもできます。

-qlanglvl=compatrvaluebinding と **-qlanglvl=rvaluereferences** の両方のオプションが有効になっている場合は、コンパイラーはエラー・メッセージを出します。

デフォルト・オプションは **-qlanglvl=norvaluereferences** です。

C++11 **scopedenum | noscopedenum**

スコープ付き列挙型機能を使用可能にするかどうかを制御します。この機能を使用可能にするために、**-qlanglvl=scopedenum** オプションを指定できます。

-qlanglvl=scopedenum オプションはグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用してスコープ付き列挙型機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=noscopedenum** です。

C++11 **static_assert | nostatic_assert**

静的アサーション機能を使用可能にするかどうかを制御します。**-qlanglvl=static_assert** が有効である場合は、この機能を使用して、障害時に重大エラー・メッセージを発行するコンパイル時アサーションを生成できます。

-qlanglvl=static_assert はグループ・オプション **-qlanglvl=extended0x** に含まれているため、このグループ・オプションを使用して静的アサーション機能を使用可能にすることもできます。

デフォルトは **-qlanglvl=nostatic_assert** です。

IBM **tempsaslocals | notempsaslocals**

C++ 言語標準では、『Temporary Object [class.temporary]』セクションで一時オブジェクトの存続期間について説明されています。一時オブジェクトの遅い破棄を実装するコンパイラーからアプリケーションを移植する場合は、C++ 一時オブジェクトの存続期間を、C++ 言語標準で指定されているよりも長くしなければならない可能性があります。このオプションは、一時ファイルの存続期間を延長して、マイグレーションの問題を低減します。

詳しくは、『 252 ページの『C++ 一時オブジェクトの存続期間を延長するための**-qlanglvl=tempsaslocals** オプションの使用』』を参照してください。

IBM **textafterendif | notextafterendif**

`#endif` または `#else` の後に追加テキストを許可するコンパイラーから IBM XL C/C++ コンパイラーにコードを移植する場合に出される警告メッセージを抑止するかどうかを指定します。デフォルト・オプションは **-qlanglvl=notextafterendif** であり、これは `#else` または `#endif` の後に余分なテキストがある場合にメッセージを出力することを示します。

trailenum | notrailenum

enum 宣言で末尾のコンマを許可するかどうかを制御します。デフォルトの **-qlanglvl=trailenum** が有効な場合、列挙子リストの最後に 1 つ以上の末尾のコンマを使用できます。例えば、以下の enum 宣言はこの拡張を使用します。

```
enum grain { wheat, barley, rye,, };
```

これは C++ 標準の拡張で、Microsoft Visual C++ との互換性を提供します。

標準 C++ に準拠させる場合は、**qlanglvl=notrailenum** を指定してください。

typedefclass | notypedefclass

typedef 名をクラス名を指定するはずの個所に指定できるかどうかを制御します。デフォルトの **-qlanglvl=typedefclass** が有効な場合、標準 C++ 規則が適用され、typedef 名をクラス名を指定するはずの個所に指定できません。
VisualAge for C++ の前のバージョンおよび先行製品との互換性を持たせるために、基本指定子リストおよびコンストラクター初期化指定子リストで typedef 名の使用を許可するには、**-qlanglvl=typedefclass** を指定します。

ucs | noucs

ユニコード文字が、プログラム・ソース・コードの ID、ストリング・リテラル、および文字リテラルで許可されるかどうかを制御します。ユニコード文字セットの詳細については、 「*XL C/C++ ランゲージ・リファレンス*」の『ユニコード規格』を参照してください。

varargmacros | novarargmacros

関数に似たマクロでの C99 型の変数引数リストのサポートを使用可能または使用不可にします。

注: -qlanglvl=c99preprocessor を指定すると、暗黙的に **-qlanglvl=varargmacros** が設定されます。逆に、**-qlanglvl=noc99preprocessor** を指定した場合も、暗黙的に **-qlanglvl=novarargmacros** が設定されます。
この機能の詳細については、「*XL C/C++ ランゲージ・リファレンス*」の『関数類似マクロ』を参照してください。

C++11 **variadic[templates] | novariadic[templates]**

可変数引数テンプレート機能を使用可能にするかどうかを制御します。この機能を使用すると、任意の数 (ゼロを含む) のパラメーターを持つクラスおよび関数テンプレートを定義できます。この機能を使用可能にするために、**-qlanglvl=variadic[templates]** オプションを指定できます。大括弧に囲まれているワード *templates* は、オプションです。**-qlanglvl=variadic** オプションだけを指定した場合、コンパイラーは **-qlanglvl=variadictemplates** オプションが指定されたものと見なします。

-qlanglvl=variadic[templates] オプションは、グループ・オプション **-qlanglvl=extended0x** に含まれているので、このグループ・オプションを使用して、可変数引数テンプレート機能を使用可能にすることもできます。

デフォルト・オプションは **-qlanglvl=novariadic[templates]** です。

zeroextarray | nozeroextarray

構造体定義で最後の非静的データ・メンバーとしてゼロ・エクステント配列を使用できるかどうかを制御します。 **-qlanglvl=zeroextarray** が有効になっている場合 (デフォルト) は、エレメントがゼロ個の配列を使用できます。以下のステートメントは、ゼロ・エクステント配列 a を宣言しています。

```
struct S1 { char a[0]; };
```

これは C++ 標準の拡張で、Microsoft Visual C++ との互換性を提供します。

標準 C++ または VisualAge C++ の前のバージョンおよび先行製品によってサポートされる ANSI 言語レベルに準拠させるには、**-qlanglvl=nozeroextarray** を指定します。

使用法

C++ 一般的に、オプション・フォーム **no** でサブオプションを指定すると、**-qsuppress** オプションで警告を使用不可にしないかぎり、コンパイラーは、コードでこの機能が使用されるたびに診断して警告を発します。さらに、**-qinfo=por** オプションを使用すると、以下のサブオプションと共に通知メッセージが生成されます。

- [no]c99complex
- [no]gnu_complex

注: C++11 C++11 言語レベルでは、サブオプションの **no** 形式を使用して `decltype` または `static_assert` の C++11 の意味を無効にした場合、ユーザーが `decltype` または `static_assert` の C++11 構文を使用すると、コンパイラーは構文エラーを出しますが、診断メッセージは出しません。

C++11

C このプラグマ・ディレクティブによってコードが移植できなくなるため、プラグマではなくオプションを使用することをお勧めします。それでもプラグマを使用する場合は、ソース・コードのコメント以外のすべての行よりも前に指定してください。また、ディレクティブはプリプロセッサーの振る舞いを動的に変更できるため、プリプロセスのみのオプションでコンパイルすると、通常のコンパイル中に作成される結果と異なる可能性があります。 C

事前定義マクロ

-qlanglvl サブオプションで事前定義されたマクロのリストについては、489 ページの『言語レベルに関連したマクロ』を参照してください。

関連情報

- 357 ページの『-qsuppress』
- 「*XL C/C++ ランゲージ・リファレンス*」の『IBM XL C 言語拡張』、および「*XL C/C++ ランゲージ・リファレンス*」の『IBM XL C++ 言語拡張』

classic 言語レベルとその他すべての標準ベースの言語レベルの相違点

以下で、**classic** 言語レベルとすべてのその他の標準ベースの言語レベルの相違点を概説します。

トークン化

マクロ展開により導入されるトークンは、場合によっては隣接するトークンと結合されることがあります。 歴史的には、これは旧式プリプロセッサーのテキスト・ベ

ースのインプリメンテーションの成果物であり、旧式のインプリメンテーションでは、プリプロセッサーは別個のプログラムで、その出力がコンパイラーに渡されていたためです。

同様の理由から、コメントによってのみ分けられたトークンが、 1 つのトークンを形成するように結合されることもあります。 ここでは、 **classic** モードでコンパイルされたプログラムのトークンを行う方法の要約を示します。

1. ソース・ファイルの該当ポイントで、次のトークンが、トークンを形成することのできる可能性のある文字の最長シーケンスです。 例えば、`i ++ + ++ j` は正しいプログラムになりますが、`i+++++j` は `i ++ ++ + j` としてトークン化されます。
2. 形成されたトークンが ID およびマクロ名である場合は、そのマクロは、その `#define` ディレクティブで指定されたトークンのテキストで置き換えられます。各パラメーターは、対応する引数のテキストで置き換えられます。 コメントは、引数とマクロ・テキストの両方から取り除かれます。
3. スキャンは、元のプログラムの一部であるかのように、マクロが置き換えられたポイントの最初のステップから再開されます。
4. プログラム全体のプリプロセスが終わると、その結果は、最初のステップのようにコンパイラーによって再度スキャンされます。置き換えを行うマクロがないため、2 番目と 3 番目のステップはここでは適用されません。プリプロセス・ディレクティブに似ている最初の 3 ステップによって生成される構造体は、そのようには処理されません。

新しいトークンを形成するため、隣接しているものの事前に分けられているトークンのテキストを結合するのは、 3 番目および 4 番目のステップで行います。

行継続用の ¥ 文字は、ストリング、文字リテラル、およびプリプロセス・ディレクティブでしか受け入れられません。

以下の構造体があるとします。

```
#if 0
  "unterminated
#endif
#define US "Unterminating string
char *s = US terminated now"
```

これは、1 番目が FALSE ブロックの終了しないリテラルで、2 番目がマクロ展開後に完了するため診断メッセージを生成しません。しかし、

```
char *s = US;
```

このとおり指定すると、`US` 内のストリング・リテラルが行の終わりまでに完了しないため、診断メッセージが生成されます。

空の文字リテラルは使用できます。このリテラルの値はゼロです。

プリプロセス・ディレクティブ

行の先頭列に、`#` トークンがなければなりません。`#` の直後に続くトークンは、マクロ展開に使用できます。行は、ディレクティブの名前と、以下の例では `(` が表示されている場合にのみ、¥ を使用して継続することができます。

```
#define f(a,b) a+b
f¥
(1,2)      /* accepted */
```

```
#define f(a,b) a+b
f(¥
1,2)       /* not accepted */
```

¥ に関する規則は、ディレクティブが有効かどうかに関係なく適用されます。例を以下に示します。

```
#¥
define M 1   /* not allowed */
```

```
#def¥
ine M 1      /* not allowed */
```

```
#define¥
M 1          /* allowed */
```

```
#dfine¥
M 1          /* equivalent to #dfine M 1, even
                though #dfine is not valid  */
```

以下に、プリプロセッサー・ディレクティブの相違点を示します。

#ifdef/#ifndef
最初のトークンが ID でないと、診断メッセージは生成されず、条件は FALSE です。

#else 余分なトークンがあると、診断メッセージは生成されません。

#endif 余分なトークンがあると、診断メッセージは生成されません。

#include
< と > は別のトークンです。ヘッダーは、< と > のつづりと、これらの間のトークンを結合することにより、形成されます。そのため、/* と // はコメントとして認識されて (常に除去され)、" と ' が < および > 内のリテラルを開始します。 (C プログラムでは、**-qcpluscmt** を指定すると、C++ 形式のコメント // が認識されます。)

#line 行番号の一部でないすべてのトークンのスペルは、新規ファイル名を形成します。これらのトークンは、ストリング・リテラルである必要はありません。

#error
認識されていません。

#define
有効なマクロ・パラメーター・リストは、コンマで区切られた 0 以上の ID で構成されます。コンマは無視され、パラメーター・リストはコンマが指定されていないかのように構成されます。パラメーター名を固有にする必要はありません。矛盾が存在する場合は、最後に指定された名前が認識されます。

無効パラメーター・リストの場合、警告が発行されます。マクロ名を新しい定義で再定義すると、警告が発行され、その新しい定義が使用されます。

#undef
余分なトークンがあると、診断メッセージは生成されません。

マクロ展開

- マクロ呼び出しの引数の数がパラメーターの数に一致しないと、警告が発行されます。
- 関数に似たマクロのマクロ名の後に (トークンがあると、これは (上記のように) 引数の数が少なすぎるとして処理され、警告が発行されます。
- パラメーターはストリング・リテラルと文字リテラルで置換されます。
- 例:

```
#define M()    1
#define N(a)   (a)
#define O(a,b) ((a) + (b))

M();  /* no error */
N();  /* empty argument */
O();  /* empty first argument
             and too few arguments */
```

テキスト出力

コメントの置き換え用に生成されるテキストはありません。

C++ 一時オブジェクトの存続期間を延長するための -qlanglvl=tempsaslocals オプションの使用

C++ 言語標準では、『*Temporary Object [class.temporary]*』セクションで一時オブジェクトの存続期間について説明されています。一時オブジェクトの遅い破棄を実装するコンパイラーからアプリケーションを移植する場合は、C++ 一時オブジェクトの存続期間を、C++ 言語標準で指定されているよりも長くしなければならない可能性があります。このように、前のコンパイラーの、標準準拠でない動作を厳密に複製することができます。

既に一時オブジェクトの破棄時に開放済みである可能性があるリソースに、プログラムが誤って依存する可能性があります。『 253 ページの『例 1』』を参照してください。そのような場合、誤って一時オブジェクトを本来よりも遅く破棄するコンパイラーが、結果として得られるプログラムを、希望どおりの形で実行することがあります。このような問題は、移植時に、一時オブジェクトのデストラクターを正しく挿入しても、開放されたリソースへのアクセスが無効になる場合に、表面化する可能性があります。

IBM XL C/C++ コンパイラーでは、一時オブジェクトの存続期間を延長することにより、マイグレーションがしやすくなります。この機能は、オプション **-qlanglvl=tempsaslocals** を指定することにより、有効になります。これが有効な場合、一時オブジェクトの存続期間は、それらの一時オブジェクトを最内部の収容字句範囲で宣言されたローカル変数として処理するかのように延長されます。ほとんどの一時オブジェクトは、外側の完全な式が完了するときではなく、それらの囲みスコープを終了するときに破棄されます。『 255 ページの『例 2』』を参照してください。

デフォルト

-qlanglvl=notempsaslocals

コンパイラー・リストは、この機能が有効な場合は **-qlanglvl=tempsaslocals** を出力し、この機能が無効な場合は **-qlanglvl=notempsaslocals** を出力します。

使用法

- if ステートメントの条件ステートメントで構成された一時オブジェクトは、その if ステートメント実行の最後に破棄する必要があります。一時オブジェクトの破棄は、条件ステートメントで宣言された変数のように、else-if または else ブロックの後まで遅延されます。
- switch ステートメントの条件で構成された一時オブジェクトは、その switch ステートメント実行の最後に破棄する必要があります。
- 最内部の囲み字句範囲が switch ステートメントの字句範囲である一時オブジェクトは、標準に準拠した方法で処理する必要があります。
- ループの条件または増分式で構成された一時オブジェクトは、標準に準拠した方法で破棄しなければなりません。
- **-qlanglvl=ansifor** が有効な場合、for-init ステートメントで構成された一時オブジェクトは、for-loop 実行の最後に破棄しなければなりません。**-qlanglvl=noansifor** が有効な場合、for-init ステートメントで構成された一時オブジェクトは、for-loop が含まれている最内部の字句ブロックの実行の最後に破棄しなければなりません。
- 名前空間スコープで構成された一時オブジェクトは、標準に準拠した方法で処理しなければなりません。『 255 ページの『例 3』』を参照してください。
- **-qinfo=por** が有効であり、一時オブジェクトの存続期間は本来であればこの機能によって延長されることになっており、その一時オブジェクトの最内部の収容字句範囲内で一時オブジェクトの構造体の後にラベル定義が続く場合は、標準に準拠した方法でその一時オブジェクトを処理します。『 256 ページの『例 4』』を参照してください。
- **-qinfo=por** が有効であり、一時オブジェクトの存続期間は本来であればこの機能によって延長されることになっており、その一時オブジェクトの最内部の収容字句範囲内で一時オブジェクトの構造体の後に computed goto (計算型 goto) が続く場合は、標準に準拠した方法でその一時オブジェクトを処理します。『 256 ページの『例 5』』を参照してください。

例 1

```
>cat myString.h
#include <string>
#define MY_SL_STD(MY_NAME) ::std::MY_NAME

class MYString
{
public:
  // common constructors
  MYString()                                                {}
  MYString(const MY_SL_STD(string&) data)  : data_(data)    {}
  MYString(const MYString& str)            : data_(str.data_)   {}
  MYString(char c, size_t N)               : data_(N, c)    {}
  MYString(const char* s)                  : data_(s)       {}
  MYString(const char* s, size_t N)        : data_(s,N)     {}

  // constructor explicitly from char
  MYString(char c)                         : data_(1,c)     {}

  ~MYString() {}
```

```
 const char*       data() const { return data_.c_str(); }

  // Type conversion:
 operator const char*() const { return data_.c_str(); }

protected:
   MY_SL_STD(string) data_;
};

>cat myString.C
#include <iostream.h>
#include "mystring.h"

class A
{
public:
  A(const char * str_)
  {
    strcpy(str, str_);
  }
  MYString getStr()
  {
     return str;
  }
  void print()
  {
    cout<<"object A "<< str <<endl;
  }

private:
  char str[2000];
};

void foo(const char* s)
{
  cout<<"foo: "<< s <<endl;
}

int main()
{
  A a("This is a test");
  a.print();
  const char * p = (const char*) a.getStr();
  cout <<"p= " << p <<endl;
  return (0);
}
>xlC myString.C
>a.out
object A This is a test
p=  e@
```

この例では、オブジェクトの char 配列を MYString に変換し、それを getStr から渡します。メソッド A::getStr は A::MYString オブジェクトに変わります。 getStr が使用されるたびにこのプログラムは誤った結果を生成し、これは問題があることを示しています。

getStr() の呼び出しが実行されると、このメソッドは str バッファーを取得し、MYString オブジェクトを構成します。これは、戻される MYString オブジェクトであり、このコードは const char * に変換するための MYString メソッドを呼び出します。このメソッドは、ストリング "This is a test." の一時コピーを指すポインターを戻します。ポインターを受け取って p 変数に格納すると、一時オブジェクトのデストラクターが呼び出されます。この時点で、p によって参照されるメモリー内に "This is a test." が存在しなくなります。文字ストリングは一時コピーであるた

め、このストリングはオブジェクト内の元の位置以外のどこにも存在しません。p を使用すると、ガーベッジにアクセスすることになります。

これらの一時オブジェクトの作成では動的メモリーが使用されるため、実際には、現在も有効でかつ同じ内容を格納しているオブジェクトに、それが使用されているステートメントよりも長く依存することはできません。

例 2

```
#include<cstdio>
struct S {
  S() { printf("S::S() ctor at 0x%lx.¥n", this); }
  S(const S& from) { printf("S::S(const S&) copy ctor at 0x%lx.¥n", this); }
  ~S() { printf("S::~S() dtor at 0x%lx.¥n", this); }
} s1;
void foo(S s) { }
int main() {
  foo(s1);
  printf("hello world.¥n");
  return 0;
}
```

このプログラムの C++ 標準準拠の出力は、次のようになります。

```
 S::S() ctor at 0x20000d7c.
 S::S(const S&) copy ctor at 0x2ff221e0.
 S::~S() dtor at 0x2ff221e0.
 hello world.
 S::~S() dtor at 0x20000d7c.
```

foo の呼び出しのために構成された一時コピーは foo から戻ると破棄されることに注意してください。一時オブジェクトの存続期間が延長されると、このプログラムの出力は、次のようになります。

```
 S::S() ctor at 0x20000d7c.
 S::S(const S&) copy ctor at 0x2ff221e0.
 hello world.
 S::~S() dtor at 0x2ff221e0.
 S::~S() dtor at 0x20000d7c.
```

foo の呼び出しのために構成された一時コピーは、囲みスコープが終了すると破棄されます。したがって、これは "hello world." の出力後に破棄されます。

例 3

```
struct S {
  S(int);
  ~S();
  int i;
} s1(42);

int bar(S s);

int gi = bar(s1); //the temporary for argument s of bar is not affected
                  //because it is constructed during static initialization.
```

この例は、ハードコーディングされたアドレスをリストしますが、これらはプログラムが実行されているシステムによって異なります。

例 4

```
struct S {
  S(int);
  ~S();
  int i;
};

void bar(S s);

int main() {
  S s1(42);
  bar(s1);   // the temporary for argument s of bar is not affected
             // because of the label definition that follows.
bypass:
  s1.i = 42; // s1 should be referenced after call to bar or a temporary may not
             // be constructed.
  return 0;  // the temporary would otherwise be destroyed here.
}
```

例 5

```
struct S {
  ~S();
} s1;

void bar(S s);

void foo(void *p) {
bar(s1);   // the temporary for argument s of bar is not affected
           // because of the computed goto that follows.
goto *p;

}
```

-qldbl128

カテゴリー

浮動小数点および整数のコントロール

プラグマ同等物

#pragma options [no]ldbl128

目的

long double 型のサイズを 64 ビットから 128 ビットに増加させる。

構文

デフォルト

-qldbl128

使用法

#pragma options ディレクティブは、ソース・ファイルの最初の C または C++ ステートメントの前に指定しなければなりません。このオプションはファイル全体に適用されます。

事前定義マクロ

- __LONGDOUBLE128 および __LONG_DOUBLE_128__ は **-qldbl128** が有効である場合、1 と定義されます。それ以外の場合は未定義です。
- **-qnoldbl128** が有効な場合、__LONGDOUBLE64 が 1 に定義されます。**-qldbl128** が有効な場合は未定義です。

例

long double 型が 128 ビットになるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qldbl128
```

-qlib

カテゴリー

リンク

プラグマ同等物

なし。

目的

標準システム・ライブラリーおよび XL C/C++ ライブラリーがリンクされるかどうかを指定する。

-qlib が有効だと、標準システム・ライブラリーおよびコンパイラー・ライブラリーが自動的にリンクされます。**-qnolib** が有効だと、リンク時に標準システム・ライブラリーおよびコンパイラー・ライブラリーは使用されません。**-l** フラグを使用してコマンド行で指定したライブラリーのみがリンクされます。

このオプションをシステム・プログラミングで使用すると、必要ないライブラリーの自動リンクが無効になります。

構文

デフォルト

-qlib

使用法

-qnolib を使用すると、XL C/C++ ライブラリー (コンパイラーのインストール・ディレクトリーの lib/ および lib64/ サブディレクトリーに格納される) だけでなく、システム・ライブラリーを含むすべてのライブラリーがリンクしないことを指定します。**-qnocrt** を指定しないと、システム・スタートアップ・ファイルはリンクされたままになります。

ご使用のプログラムが標準ライブラリーまたはコンパイラー固有のライブラリーで定義されたシンボルのいずれかを参照する場合、リンク・エラーが発生します。**-qnolib** を使用したコンパイル時にこれらの未解決の参照を回避するには、**-l** コマンド・フラグおよびライブラリー名を使用して必要ライブラリーを明示的にリンクします。

事前定義マクロ

なし。

例

コンパイラー・ライブラリー libxlopt.a 以外のすべてのライブラリーにリンクせずに `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qnolib -lxlopt
```

関連情報

- 138 ページの『-qcrt』

-qlibansi

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options [no]libansi

目的

ANSI C ライブラリー関数の名前が付いたすべての関数が実際はシステム関数であると見なす。

libansi が有効な場合、最適化プログラムは、副次作用があるかどうかなど特定の関数の振る舞いについて知るので、より優れたコードを生成することができます。

構文

デフォルト

-qnolibansi

事前定義マクロ

C++ **libansi** が有効な場合、__LIBANSI__ は 1 に定義されます。それ以外の場合は未定義です。

-qlibmpi

カテゴリー

94 ページの『最適化およびチューニング』

プラグマ同等物

なし

目的

Message Passing Interface (MPI) 名を持つすべての関数が事実上 MPI 関数であり、異なるセマンティクスを持つユーザー関数でないことを表明する。

構文

デフォルト

-qnolibmpi

使用法

MPI は、メッセージ引き渡し用のライブラリー・インターフェース仕様です。これは、データが連携操作によって 1 つのプロセスのアドレス・スペースから別のプロセスへ移動するメッセージ引き渡し並列プログラミング・モデルに対応しています。MPI について詳しくは、Message Passing Interface Forum を参照してください。

-qlibmpi を使用すると、コンパイラーは、特定の関数に副次作用があるかどうかなど、関数の振る舞いを認識できるため、より優れたコードを生成できるようになります。

-qlibmpi を使用した場合、コンパイラーは MPI ライブラリー関数の名前が付いたすべての関数を、事実上 MPI 関数と想定します。**-qnolibmpi** では、そのような想定は行われません。

注: 標準のライブラリー関数と互換性のないユーザー独自バージョンのライブラリー関数がアプリケーションに含まれている場合、このオプションは使用できません。

事前定義マクロ

なし。

例

myprogram.c をコンパイルするには、以下のコマンドを入力します。

```
xlc -O5 myprogram.c -qlibmpi
```

関連情報

- Message Passing Interface Forum
- 211 ページの『-qipa』

-qlinedebug

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

デバッガー用に行番号およびソース・ファイル名の情報のみを生成する。

-qlinedebug が有効な場合、コンパイラーは最小限のデバッグ情報しか生成しないため、結果として得られるオブジェクト・サイズは、 **-g** デバッグ・オプションを指定した場合に生成されるオブジェクトよりも小さくなります。デバッガーを使用してソース・コードをステップスルーすることができますが、変数情報を表示したり照会することはできません。 トレースバック・テーブルを生成させると、行番号が組み込まれます。

-qlinedebug は **-g1** と同等です。

構文

デフォルト

-qnolinedebug

使用法

-qlinedebug が有効な場合、関数のインライン化は使用不可になります。

-qlinedebug を **-O** (最適化) オプションとともに使用しないでください。生成される情報が不完全であったり、誤解のもととなる場合があります。

-g オプションは、**-qlinedebug** オプションをオーバーライドします。コマンド行に **-g**with**-qnolinedebug** を指定した場合は、**-qnolinedebug** が無視され、警告が出されます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして実行可能プログラム `testing` を作成し、それをデバッガーを使用してステップスルーできるようにするには、以下のように入力します。

```
xlc myprogram.c -o testing -qlinedebug
```

関連情報

- 173 ページの『-g』
- 279 ページの『-O、-qoptimize』

-qlist

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

#pragma options [no]list

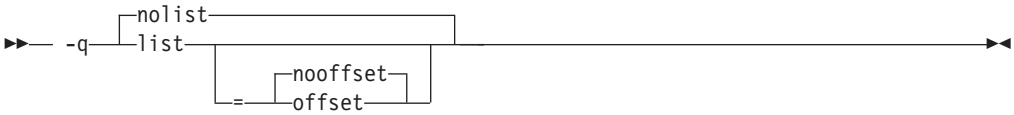
目的

オブジェクト・リストを含むコンパイラー・リスト・ファイルを生成する。

list が有効な場合、コマンド行で指定された各ソース・ファイル名に対して .lst サフィックスが付いたリスト・ファイルが生成されます。リスト・ファイルの内容について詳しくは、22 ページの『コンパイラー・リスト』を参照してください。

生成されたコードのパフォーマンス特性の理解を助けたり、実行上の問題を診断するために、オブジェクト・リストまたはアセンブリー・リストを使用することができます。

構文



デフォルト

-qnolist

パラメーター

offset | nooffset
PDEF ヘッダーのオフセット `00000` を、テキスト領域のはじめからのオフセットに変更します。このオプションを指定すると、.lst ファイルを読み取るすべて

のプログラムが PDEF の値と問題の行を追加することができるため、**offset** または **nooffset** のどちらが指定されていても同じ値になります。 **offset** サブオプションは、コンパイル単位内に複数のプロシージャーがある場合にのみ関係します。

サブオプションなしで **list** を指定することは、**list=nooffset** を使用するのと同等です。

使用法

-qnoprint コンパイラー・オプションは、このオプションをオーバーライドします。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、オブジェクト・リストを含むリスト (.lst) ファイルを作成するには、以下のように入力します。

```
xlc myprogram.c -qlist
```

関連情報

- 266 ページの『-qlistopt』
- 308 ページの『-qprint』
- 340 ページの『-qsource』

-qlistfmt

カテゴリー

リスト、メッセージ、およびコンパイラー情報

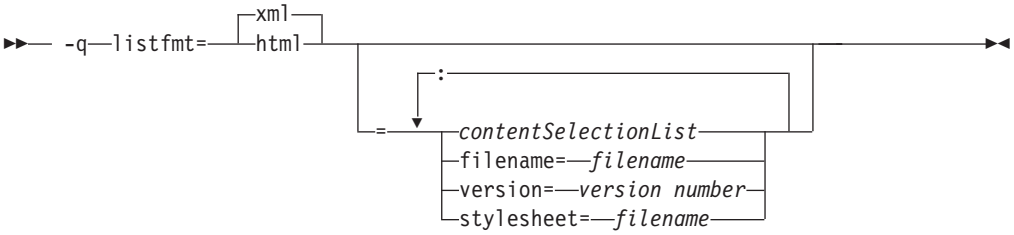
プラグマ同等物

なし。

目的

最適化の機会の検出を支援するために、XML または HTML レポートを作成する。

構文



デフォルト

このオプションは、デフォルトではオフです。*contentSelectionList* オプションがいずれも肯定形式で選択されていない場合は、使用可能なすべてのレポート情報が生成されます。例えば、**-qlistfmt=xml** の指定は、**-qlistfmt=xml=all** と等価です。

パラメーター

以下のリストで **-qlistfmt** のパラメーターを説明します。

xml | html
XML または HTML フォーマットでレポートを生成する必要があることを示します。XML レポートが前に生成されている場合は、**genhtml** コマンドを使用してそのレポートを HTML フォーマットに変換できます。このコマンドについては詳しくは、265 ページの『genhtml コマンド』を参照してください。

contentSelectionList
以下のサブオプションは、レポート内の情報のタイプと量を制限するためのフィルターを提供します。

data | nodata
データ再編成情報を生成します。

inlines | noinlines
インライン化情報を生成します。

pdf | nopdf
Profile-Directed Feedback 情報を生成します。

transforms | notransforms
ループ変換情報を生成します。

all
入手可能なすべてのレポート情報を生成します。

none
レポートを生成しません。

filename
レポート・ファイルの名前を指定します。コンパイル・フェーズで 1 つのファイルが生成され、IPA リンク・フェーズで 1 つのファイルが生成されます。ファイル名を指定しなかった場合は、そのプラットフォームの名前生成規則に整合する方法で、接尾部 `.xml` または `.html` が付いたファイルが生成されます。例えば、`foo.c` をコンパイルする場合、生成された XML ファイルは、コンパイル・ステップの場合は `foo.xml` であり、リンク・ステップの場合は `a.xml` になります。

注: コンパイルとリンクを 1 つのステップで行い、このサブオプションを使用してレポートのファイル名を指定すると、IPA リンク・ステップからの情報は、コンパイル・ステップで生成された情報を上書きします。

`filename` サブオプションを使用して複数のファイルをコンパイルする場合にも同じことが当てはまります。コンパイラーは個々のファイルにレポートを作成するため、最後にコンパイルされたファイルのレポートが前のレポートを上書きします。例を以下に示します。

```
xlc -qlistfmt=xml=all:filename=abc.xml -O3 myfile1.c myfile2.c myfile3.c
```

これは、最後のファイル myfile3.c のコンパイルに基づいた abc.xml レポートのみを生成します。

stylesheet
結果のレポート内に埋め込まれる xml-stylesheet ディレクティブの対象となる、既存の XML スタイルシートの名前を指定します。デフォルトの振る舞いでは、スタイルシートは含まれません。 XL C/C++ に添付されているスタイルシートは、xlstyle.xsl です。このスタイルシートは、XML を XSLT をサポートするブラウザーで表示したときに、読みやすいフォーマットにレンダリングします。

stylesheet サブオプションを指定して作成した XML レポートを表示するには、実際のスタイルシート (xlstyle.xsl) と XML メッセージ・カタログ (XMLMessages-*locale*.xml。*locale* はコンパイル・マシンに設定されているロケールを示します) を **stylesheet** サブオプションで指定したパスに配置する必要があります。例えば、**stylesheet=xlstyle.xsl** を指定して a.xml を生成した場合は、ブラウザーで a.xml を正しく表示するには、xlstyle.xsl および XMLMessages-*locale*.xml を a.xml と同じディレクトリーに配置しておく必要があります。メッセージ・カタログとスタイルシートは /opt/ibm/xlC/13.1.0/listings/ ディレクトリーにインストールされます。

version
出力する内容のメジャー・バージョンを指定します。このレポートの特定のバージョンを必要とするツールを作成した場合は、バージョンを指定してください。IBM XL C/C++ for Linux, V13.1 は、レポートを XML v1.1 で作成します。したがって、それらのレポート使用するツールを作成した場合は、version=v1 を指定してください。

使用法

-qlistfmt オプションによってレポートに生成される情報は、プログラムのコンパイルに使用される最適化オプションによって異なります。

- **-qinline** などのインライン化を使用可能にするオプションとともに使用した場合、レポートには、インライン化された関数と、他の関数がインライン化されなかった理由が示されます。
- ループのアンロールを使用可能にするオプションとともに使用した場合、このレポートにはプログラム・ループを最適化する方法の要約が含まれます。また、このレポートには、特定のループをベクトル化できない理由を示す診断情報も含まれています。また、**-qlistfmt** でループ変換に関する情報を生成するには、以下のオプションの中から少なくとも 1 つを指定する必要があります。
  - **-qsimd=auto**
  - **-qsmp**
  - **-O5**
  - **-qipa=level=2**
- 並列変換を使用可能にするオプションとともに使用した場合、レポートには、並列変換に関する情報が含まれます。また、**-qlistfmt** で、並列変換または並列パフ

ォーマンス・メッセージに関する情報を生成するには、以下のオプションの中から少なくとも 1 つを指定する必要があります。

  - **-qsmp**
  - **-O5**
  - **-qipa=level=2**
- プロファイル作成を使用可能にするオプション **-qpdf** とともに使用した場合、レポートには、呼び出しおよびブロックの数、およびキャッシュ・ミスに関する情報が含まれます。
- **-qipa=level=2** のようなデータ再編成を生成するオプションとともに使用した場合、レポートには、これらの再編成に関する情報が含まれます。

contentSelectionList オプションがいずれも肯定形式で選択されていない場合は、使用可能なすべてのレポート情報が生成されます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、ループの最適化方法を示す XML レポートを生成するには、以下のように入力します。

```
xlc -qhot -O3 -qlistfmt=xml=transforms myprogram.c
```

`myprogram.c` をコンパイルして、インライン化された関数を示す XML レポートを生成するには、以下を入力します。

```
xlc -qinline -qlistfmt=xml=inlines myprogram.c
```

genhtml コマンド

既に生成済みの XML レポートの HTML バージョンを表示するには、**genhtml** ツールを使用できます。

以下のコマンドを使用して、既存の XML レポートを HTML フォーマットで表示します。以下のコマンドは、HTML コンテンツを標準出力に生成します。

```
genhtml xml_file
```

以下のコマンドを使用して、HTML コンテンツを、定義した HTML ファイルに生成します。 Web ブラウザーを使用して、生成された HTML ファイルを表示できます。

```
genhtml xml_file > target_html_file
```

注: HTML ファイル名のサフィックスは、静的 HTML ページの標準に準拠している必要があります (例えば、`.html` や `.htm`)。そうしないと、Web ブラウザーでファイルを開くことができない場合があります。

関連情報

- 315 ページの『-qreport』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『コンパイラー・レポートを使用した最適化の機会の診断』

-qlistopt

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

コンパイラー呼び出し時に有効なすべてのオプションを含むコンパイラー・リスト・ファイルを作成する。

listopt が有効な場合、コマンド行で指定された各ソース・ファイル名に対して a .lst サフィックスを含むリスト・ファイルが生成されます。リストにはコンパイラー・デフォルト、構成ファイル、およびコマンド行設定によって設定された有効オプションが表示されます。リスト・ファイルの内容について詳しくは、 22 ページの『コンパイラー・リスト』を参照してください。

構文

デフォルト

-qnolistopt

使用法

プログラム・ソースにあるプラグマ・ステートメントによるオプション設定は、コンパイラー・リストには示されません。

-qnoprint コンパイラー・オプションは、このオプションをオーバーライドします。

事前定義マクロ

なし。

例

すべての有効オプションを示すリスト (.lst) ファイルが作成されるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qlistopt
```

関連情報

- 261 ページの『-qlist』
- 308 ページの『-qprint』
- 340 ページの『-qsource』

-qlonglit

カテゴリー

浮動小数点および整数のコントロール

プラグマ同等物

なし。

目的

64 ビット・モードで整数リテラルの暗黙の型を決める際に、コンパイラーは `l` または `L` 接尾部が接尾部のない整数リテラルまたは接尾部が `u` か `U` のみである整数リテラルに追加されたかのように動作する。

構文

デフォルト

-qnolonglit

使用法

-qlonglit オプションを指定した後に、`int` または `unsigned int` 型が整数リテラルの暗黙型リストに含まれていると、`int` または `unsigned int` 型は、それぞれ `long int` または `unsigned long int` 型に置き換えられます。整数リテラルについて詳しくは、『整数リテラル』を参照してください。

事前定義マクロ

なし。

例

-qlonglit オプションを指定すると、整数リテラル 0x80000000 は 64 ビット・モードの `long int` 型を持ちます。このオプションを指定しない場合、整数リテラルは 32 ビット・モードと 64 ビット・モードの両方の `unsigned int` 型を持ちます。

-qlonglong

カテゴリー

言語エレメント制御

プラグマ同等物

#pragma options [no]longlong

目的

プログラムで IBM long long 整数型を許可する。

構文

デフォルト

- C **-qlonglong** (xlc、xlc++、xlC、cc および c99 呼び出しコマンドの場合)。**-qnolonglong** (c89 呼び出しコマンドの場合)。
- C++ **-qlonglong**

使用法

C このオプションは、**-qlanglvl=extended** | **stdc89** | **extc89** オプションが有効な場合に、有効になります。このオプションは、**-qlanglvl=stdc99** | **extc99** オプションが有効な場合には無効になります。このオプションで提供される long long サポートは、C99 標準によって必須になっている long long 型のセマンティクスと互換性を持たないからです。

C++ このオプションは、**-qlanglvl=c99longlong** オプションが有効な場合には無効になります。このオプションで提供される long long サポートは、C++11 で採用されているように、C99 標準によって必須となっている long long 型のセマンティクスと互換性を持たないからです。

事前定義マクロ

long long データ型が使用可能な場合、_LONG_LONG は 1 に定義されます。それ以外の場合は未定義です。

例

IBM long long 整数がサポートされるように myprogram.c をコンパイルするには、以下のように入力します。

```
cc myprogram.c -qlonglong
```

関連情報

- 「*IBM XL C/C++ for Linux, V13.1* ランゲージ・リファレンス」の『整数型』

-ma (C のみ)

『 111 ページの『-qalloca、-ma (C のみ)』』を参照してください。

-qmakedep, -M

カテゴリー

出力制御

プラグマ同等物

なし。

目的

各ソース・ファイルに対して **make** ツールによって使用される従属関係ファイルを生成する。

従属関係出力ファイルには .d サフィックス付きの名前が付けられます。

構文

デフォルト

適用されません。

パラメーター

gcc（-qmakedep オプションのみ)
生成された **make** 規則のフォーマットで、GCC フォーマットに一致します。従属関係出力ファイルには、主なソース・ファイルの従属関係をすべてリストした単一ターゲットが組み込まれます。

サブオプションなしで **-qmakedep** を指定するか、または **-M** を指定すると、従属関係出力ファイルは主なソース・ファイルの従属関係それぞれに対して個別に規則を指定します。

使用法

コマンド・ラインで指定された .c、.C、.cpp、または .i サフィックスを含むソース・ファイルごとに、オブジェクト・ファイルと同じ名前の従属関係出力ファイルが生成されます。ただし、その出力ファイルのサフィックスは .d になります。その他のタイプの入力ファイルに対しては、従属関係出力ファイルは作成されません。**-o** オプションを使用してオブジェクト・ファイルの名前を変更する場合、従属関係出力ファイルの名前は **-o** オプションで指定した名前に基づきます。詳しくは、『例』のセクションを参照してください。

これらのオプションを指定して生成された従属関係出力ファイルは、**make** 記述ファイルではありません。これらのファイルを **make** コマンドで使用するには、リンクさせる必要があります。このコマンドについて詳しくは、ご使用のオペレーティング・システムの資料を参照してください。

出力ファイルには入力ファイルのための行とそれぞれの組み込みファイルのための項目があります。この一般形式は次の通りです。

file_name.o:*include_file_name*
file_name.o:*file_name.suffix*

-qmakedep および **-M** は、以下のオプションと共に使用することもできます。

-MF *file_path*
従属関係出力ファイルの名前を設定します。ここで *file_path* は従属関係出力ファイルの部分パスまたは完全なパス、あるいはファイル名です。詳しくは、『275 ページの『-MF』』を参照してください。

組み込みファイルは、#include プリプロセッサー・ディレクティブの検索順序の規則に従ってリストされます。この規則については、15 ページの『組み込みファイルのディレクトリー検索シーケンス』に説明があります。組み込みファイルが検出されない場合、.d ファイルには追加されません。

include ステートメントのないファイルでは、入力ファイル名だけをリストした 1 行が含まれる従属関係出力ファイルが作成されます。

事前定義マクロ

なし。

例

例 1: mysource.c をコンパイルして、mysource.d という名前の従属関係出力ファイルを作成するには、以下を入力します。

```
xlc -c -qmakedep mysource.c
```

例 2: foo_src.c をコンパイルして、mysource.d という名前の従属関係出力ファイルを作成するには、以下を入力します。

```
xlc -c -qmakedep foo_src.c -MF mysource.d
```

例 3: foo_src.c をコンパイルして、deps/ ディレクトリー内に mysource.d という名前の従属関係出力ファイルを作成するには、以下を入力します。

```
xlc -c -qmakedep foo_src.c -MF deps/mysource.d
```

例 4: foo_src.c をコンパイルして、foo_obj.o という名前のオブジェクト・ファイル、および foo_obj. d の場合、以下を入力します。

```
xlc -c -qmakedep foo_src.c -o foo_obj.o
```

例 5: foo_src.c をコンパイルして、foo_obj.o という名前のオブジェクト・ファイル、および mysource.d という名前の従属関係出力ファイルを作成するには、以下を入力します。

```
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.d
```

例 6: foo_src1.c および foo_src2.c をコンパイルして、それぞれ foo_src1.d および foo_src2.d という名前の 2 つの従属関係出力ファイルを c:/tmp/ ディレクトリー内に作成するには、以下を入力します。

```
xlc -c -qmakedep foo_src1.c foo_src2.c -MF c:/tmp/
```

関連情報

- 275 ページの『-MF』
- 278 ページの『-o』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』

-qmaxerr

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

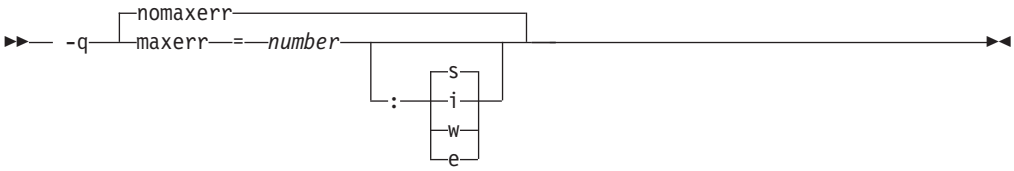
なし。

目的

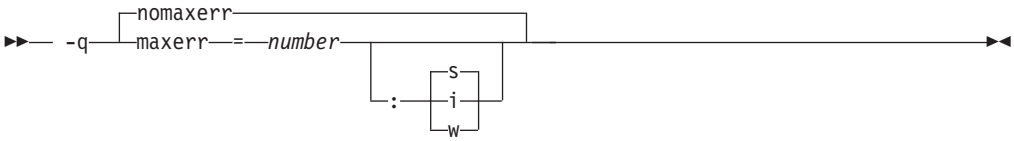
指定した重大度レベル以上のエラー・メッセージの数が指定した数に到達すると、コンパイルを停止する。

構文

-qmaxerr 構文 — C



-qmaxerr 構文 — C++



デフォルト

-qnomaxerr

パラメーター

number
コンパイラーが停止するまでに生成するメッセージの最大数を指定します。*number* は、1 以上の整数値でなければなりません。

i 重大度レベルを通知 (I) 以上にすることを指定します。

w 重大度レベルを警告 (W) 以上にすることを指定します。

C **e**
重大度レベルをエラー (E) 以上にすることを指定します。

s 重大度レベルが重大 (S) であることを指定します。

使用法

-qmaxerr オプションで重大度レベルを指定しない場合は、**-qhalt** オプションによって有効になる重大度が使用されます。それ以外の場合は、**-qmaxerr** と **-qhalt** のうち最後に記述されたオプションによって重大度レベルが指定されます。

診断メッセージは、**-qflag** オプションによって制御できます。

事前定義マクロ

なし。

例

10 件の警告が検出されたら `myprogram.c` のコンパイルを停止するには、以下のコマンドを入力してください。

```
xlc myprogram.c -qmaxerr=10:w
```

現行の **-qhalt** オプション値が **S** (重大) であると想定し、5 件の重大エラーが検出された場合に `myprogram.c` のコンパイルを停止するには、以下のコマンドを入力してください。

```
xlc myprogram.c -qmaxerr=5
```

3 件の通知メッセージを検出した場合に `myprogram.c` のコンパイルを停止するには、以下のコマンドを入力してください。

```
xlc myprogram.c -qmaxerr=3:i
```

または

```
xlc myprogram.c -qmaxerr=3 -qhalt=i
```

関連情報

- 158 ページの『-qflag』
- 179 ページの『-qhalt』
- 21 ページの『メッセージ重大度レベルとコンパイラー応答』

-qmaxmem

カテゴリー

最適化およびチューニング

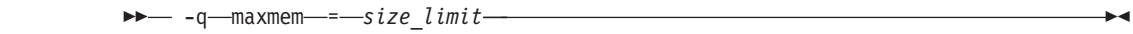
プラグマ同等物

#pragma options maxmem

目的

メモリーを消費する、指定したキロバイト数になるまでの特定の最適化の実行中に、コンパイラーによって割り振られるメモリーの量を制限する。

構文



デフォルト

- **-O2** が有効な場合は **-qmaxmem=8192** です。
- **-qmaxmem=-1** (**-O3** 以上の最適化レベルが有効な場合)

パラメーター

size_limit
最適化プログラムが使用するメモリー量に相当するキロバイト数です。その制限は、コンパイラー全体のものではなく、特定の最適化におけるメモリー量です。全コンパイル処理中に必要とされるテーブルは、この単位によって影響を受けることも組み込まれることもありません。

値を -1 にすると、制限について検査されることなく、最適化ごとに必要な量のメモリーが使用できるようになります。

使用法

制限を低くしても、結果として得られるプログラムが必ずしも遅くなるわけではありません。単に、パフォーマンスを向上させるすべての機会を検出する前にコンパイラーが終了する場合があるだけです。 制限を増加しても、結果として得られるプログラムが必ずしも高速になるわけではありません。単に、パフォーマンスを向上させる機会がある場合にコンパイラーがその機会を検出しやすくなるだけです。

コンパイラーが制限以下のメモリーしか必要としない場合は、大きい制限を設定してもソース・ファイルのコンパイルに悪影響は及びません。 コンパイル中のソース・ファイル、ソース内のサブプログラムのサイズ、マシン構成、およびシステムのワークロードによっては、制限の設定を高くしすぎたり -1 に設定した場合、使用可能なシステム・リソースを超えることがあります。

事前定義マクロ

なし。

例

ローカル・テーブルに指定されるメモリーが 16384 キロバイトになるように myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qmaxmem=16384
```

-qmbcs, -qdbcs

カテゴリー

言語エレメント制御

プラグマ同等物

#pragma options [no]mbcs、#pragma options [no]dbcs

目的

ソース・コード内で、マルチバイト文字セット (MBCS) およびユニコード文字のサポートを使用可能にする。

mbcs または **dbcs** が有効な場合、マルチバイト文字リテラルおよびコメントがコンパイラーによって認識されます。 **nombcs** または **nodbcs** が有効な場合、コンパイラーはすべてのリテラルを 1 バイト・リテラルとして処理します。

構文

デフォルト

-qnombcs、-qnodbcs

使用法

ソース・コードにおけるマルチバイト文字使用の規則については、「*XL C/C++* ランゲージ・リファレンス」の 『マルチバイト文字』を参照してください。

さらに、マルチバイト文字を以下のコンテキストで使用することもできます。

- コマンド行でコンパイラー呼び出しに引数として渡されたファイル名の中。以下に例を示します。

  ```
  xlc /u/myhome/c_programs/kanji_files/multibyte_char.c -omultibyte_char
  ```

- ファイル名の中で、ファイル名を引数として使用するコンパイラー・オプションへのサブオプションとして。
- **-D** オプションを使用したマクロ名の定義の中。以下に例を示します。

  ```
  -DMYMACRO="kpsmultibyte_chardcs"
  -DMYMACRO='multibyte_char'
  ```

リスト・ファイルには適切な国際言語に対応する日時が表示され、ソース・ファイル名のマルチバイト文字も、対応するリスト・ファイルの名前の中に表示されます。 例えば、以下の名前の C ソース・ファイルがあるとします。

multibyte_char.c

この C ソース・ファイルは、以下の名前のリスト・ファイルを作成します。

multibyte_char.lst

事前定義マクロ

なし。

例

マルチバイト文字が含まれる myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qmbcs
```

関連情報

- 141 ページの『-D』

-MF

カテゴリー

出力制御

プラグマ同等物

なし。

目的

-qmakedep または **-M** オプションによって生成される従属関係出力ファイルの名前または場所を指定する。

-qmakedep および **-M** オプションについて詳しくは、 268 ページの『-qmakedep, -M』 を参照してください。

構文

デフォルト

-MF が指定されていない場合、従属関係出力ファイルは、オブジェクト・ファイルと同じ名前に .d サフィックスを付けて、現行作業ディレクトリー内に作成されます。

パラメーター

file_path
ターゲット出力のパスです。*file_path* は完全なディレクトリー・パスまたはファイル名です。 path or file name. *file_path* がディレクトリー名である場合、コンパイラーによって作成される従属関係ファイルは、その指定ディレクトリーに置かれます。ディレクトリーを指定しない場合、従属関係ファイルは現行作業ディレクトリーに格納されます。

使用法

-MF オプションによって指定されたファイルが既に存在する場合は、上書きされます。

複数のソース・ファイルをコンパイルするときに **-MF** オプションに対して単一ファイル名を指定すると、単一の従属関係ファイルのみが生成されます。従属関係ファイルには、コマンド・ラインで最後に指定されたファイルの **make** 規則が含まれます。

事前定義マクロ

なし。

関連情報

- 268 ページの『-qmakedep, -M』
- 278 ページの『-o』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』

-qminimaltoc

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

64 ビット・コンパイル・モードで実行可能ファイルのためにコンパイラーが作成する目次 (TOC) の生成を制御する。

64 ビット・モードでコンパイルされたプログラムは、TOC エントリーが 8192 に制限されています。そのため、64 ビット・モードで大きなプログラムをリンクする場合は、「再配置切り捨て」エラー・メッセージが表示される場合があります。これらのエラー・メッセージの原因は、TOC オーバーフロー条件です。**-qminimaltoc** が有効な場合、コンパイラーは、各オブジェクト・ファイルの別々のデータ・セクションに TOC エントリーを入れることによって、これらの TOC オーバーフロー条件を回避します。

-qminimaltoc を指定すると、コンパイラーは各コンパイル単位に対して TOC エントリーを 1 つのみ作成します。このオプションを指定すると、使用可能な TOC エントリーの使用を最小限にできますが、その使用によってパフォーマンスが影響を受けます。特に頻繁に実行するコードを含むファイルに **-qminimaltoc** オプションを使用する場合は慎重にしてください。

構文

デフォルト

-qnominimaltoc

使用法

このコンパイラー・オプションは 64 ビット・コンパイルにのみ適用されます。

-qminimaltoc でコンパイルすると、多少動作が遅い大きなプログラム・コードが作成されることがあります。しかし、プログラムのコンパイル時に最適化オプション

を指定すると、このような影響を最小限に抑えることができます。

事前定義マクロ

なし。

-qmkshrobj

カテゴリー

出力制御

プラグマ同等物

なし。

目的

生成されたオブジェクト・ファイルから共有オブジェクトを作成する。

リンカーを直接呼び出す代わりに、このオプションをこのトピック内で説明する関連オプションと一緒に使用して、共有オブジェクトを作成します。 このオプションを使用する利点は、(テンプレート・インクルード・ディレクトリーまたはテンプレート・レジストリーを使用した) リンク時 C++ テンプレートのインスタンス生成の自動処理と、(**-O5** で実行されるような) **-qipa** リンク時最適化との互換性があることです。

構文

-qmkshrobj syntax

デフォルト

デフォルトで、出力オブジェクトをランタイム・ライブラリーおよび始動ルーチンとリンクすることで、実行可能ファイルが作成されます。

使用法

--version-script リンカー・オプションを使用してエクスポートするシンボルを指定しない限り、コンパイラーは自動的に共有オブジェクトからすべてのグローバル・シンボルをエクスポートします。また、**-qnoweakexp** オプションを使用することで、弱いシンボルのエクスポートを回避することもできます。 IBM hidden または internal の visibility 属性を持つシンボルは、エクスポートされません。 IBM

-qmkshrobj を指定すると、**-qpic** が暗黙指定されます。

以下の関連オプションを **-qmkshrobj** に使用することもできます。

-o *shared_file*
共有ファイルの情報を保持するファイルの名前です。デフォルトは a.outです。

-e *name*
共有実行可能ファイルの入り口名を *name* に設定します。

注: オプション **-qmkshrobj** と **-qstaticlink** は両立しないため、一緒に指定することはできません。

-qmkshrobj を使用した共有ライブラリーの作成の詳細については、以下を参照してください。「*XL C/C++ 最適化およびプログラミング・ガイド*」の『ライブラリーの構成』。

事前定義マクロ

なし。

例

3 つの小さなオブジェクト・ファイルから共有ライブラリー `big_lib.so` を構成するには、以下のコマンドを入力します。

```
xlc -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

関連情報

- 149 ページの『-e』
- 211 ページの『-qipa』
- 『-o』
- 303 ページの『-qpic』
- 309 ページの『-qpriority (C++ のみ)』
- 389 ページの『-qvisibility』
- 421 ページの『#pragma GCC visibility push、#pragma GCC visibility pop』

-o

カテゴリー

出力制御

プラグマ同等物

なし。

目的

出力オブジェクト、アセンブラー、または実行可能ファイルの名前を指定する。

構文

デフォルト

コンパイルの異なるフェーズで作成されるデフォルトのファイル名およびサフィックスについての詳細は、 5 ページの『出力ファイルのタイプ』を参照してください。

パラメーター

path

ソース・ファイルからコンパイルするオプションを使用する場合、ファイルまたはディレクトリーの名前に *path* を使用できます。 *path* は、相対パスまたは絶対パスの名前にすることができます。 オブジェクト・ファイルからリンクするオプションを使用する場合、ファイル名に *path* を使用しなければなりません。

path が既存ディレクトリーの名前である場合、コンパイラーによって作成されるファイルはそのディレクトリーに置かれます。 *path* が既存ディレクトリーでない場合、*path* はコンパイラーによって作成されたファイルの名前になります。以下の例を参照してください。

C または C++ のソース・ファイル・サフィックス (.C、.c、.cpp、または .i) を持つ `myprog.c` や `myprog.i` などのファイル名は指定できません。指定するとエラーが発生し、コンパイラーもリンカーも呼び出されないためです。

使用法

-c オプションと **-o** を一緒に使用して、 *path* が既存ディレクトリーでない場合は、一度に 1 つのソース・ファイルしかコンパイルできません。この場合、コンパイラー呼び出し時に複数のソース・ファイル名がリストされる場合、コンパイラーは、警告メッセージを出して **-o** を無視します。

-E、**-P**、および **-qsyntaxonly** オプションは、 **-o** オプションをオーバーライドします。

事前定義マクロ

なし。

例

`myaccount` という名前のディレクトリーが存在しないと想定し、結果として `myaccount` という実行可能ファイルが作成されるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -o myaccount
```

`test.c` をオブジェクト・ファイルのみにコンパイルし、そのオブジェクト・ファイルに `new.o` と名付けるには、以下のように入力します。

```
xlc test.c -c -o new.o
```

関連情報

- 123 ページの『-c』
- 150 ページの『-E』
- 288 ページの『-P』
- 361 ページの『-qsyntaxonly (C のみ)』

-O、-qoptimize

カテゴリー

最適化およびチューニング

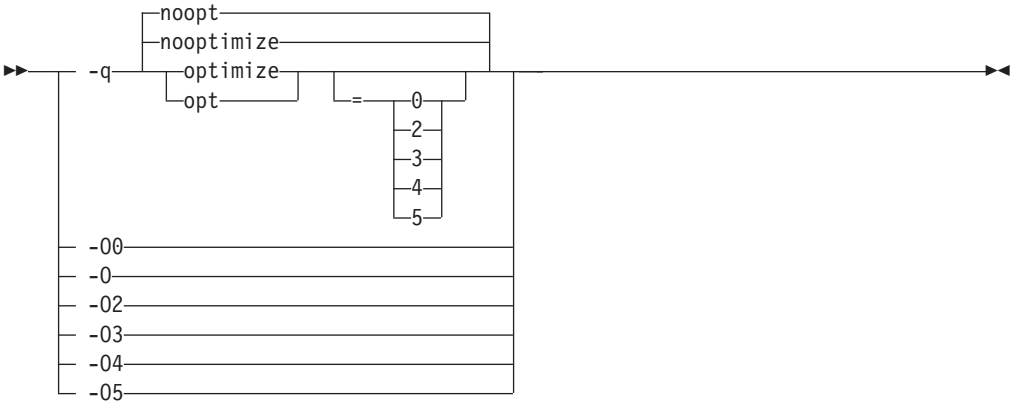
プラグマ同等物

#pragma options [no]optimize

目的

コンパイル中にコードを最適化するかどうかを指定し、最適化する場合は、レベルを指定する。

構文



デフォルト

-qnooptimize または **-O0** または **-qoptimize=0**

パラメーター

-O0 | nooptimize | noopt | optimize|opt=0
定数のフォールディングやローカルの共通副次式の除去など、高速でローカルな最適化のみを実行します。

この設定は、**-qnostrict_induction** を明示的に指定していない限り、**-qstrict_induction** を暗黙指定します。

-O | -O2 | optimize | opt | optimize|opt=2
コンパイル速度とランタイム・パフォーマンスの最良の組み合わせであるとコンパイラー開発者が考えた最適化を実行します。最適化は製品のリリースごとに異なる場合があります。特定のレベルの最適化が必要な場合は、適切な数値を指定してください。

この設定は、**-qstrict_induction** または **-qnostrict** によって明示的に否定されていない限り、**-qstrict** と **-qnostrict_induction** を暗黙指定します。

-O3 | optimize|opt=3
メモリー、コンパイル時間、またはこれら両方を大量に消費する追加の最適化を実行します。 このレベルは、コンパイル・リソースの最小化よりも実行時の向上を図りたい場合に有効です。

-O3 は **-O2** レベルの最適化を適用しますが、時間とメモリーの制限は無制限になります。また **-O3** は、プログラムのセマンティクスを若干変更する可能性の

ある、より高度で積極的な最適化を実行します。コンパイラーは、**-O2** ではこれらの最適化から保護します。**-O3** を指定したときに実行される積極的な最適化は以下の通りです。

1. 積極的なコードの動き、および例外を発生させる可能性がある計算のスケジューリングが許可されます。

   ロードおよび浮動小数点計算は、このカテゴリーに分類されます。 この最適化が積極的なのは、命令がプログラムの実際のセマンティクスに一致していない*可能性*がある ときに命令を実行する 実行パスに、それらの命令を配置することがあるためです。

   例えば、ループ内で不変の浮動小数点計算がループを通るいくつかのパスで見つかっても、すべてのパスを通らない場合は、その計算が例外を発生させる場合があるため、 **-O2** では移動されません。**-O3** では、例外が発生することが確実ではないので、コンパイラーはその計算を移動させます。ロードの動きについても同じことが言えます。ポインターを介したロードが移動されることはありませんが、静的またはスタック基底レジスターを介さないロードは、 **-O3** では移動可能であると見なされます。プログラムに 10 個のエレメントがある静的配列 `a` の宣言を入れて、 `a[60000000003]` をロードすると、セグメント違反が発生する可能性があるため、一般に **-O2** でのロードは絶対に安全であるとはいえません。

   同様の概念がスケジューリングにも適用されます。

   例:

   以下の例において、**-O2** では、 `b+c` の計算は、以下の 2 つの理由でループからは移動されません。

   - 浮動小数点演算であるため危険と見なされる。
   - これは、ループを介したすべてのパス上で発生するものではない。

   -O3 では、コードは移動されます。

   ```
     ...
   int i ;
   float a[100], b, c ;
   for (i = 0 ; i < 100 ; i++)
    {
    if (a[i] < a[i+1])
     a[i] = b + c ;
    }
     ...
   ```

2. IEEE 規則への準拠が緩和されます。

   -O2 では、ある特定の最適化は実行されません。これは、そのような最適化によって、結果がゼロの場合に誤った符号が生成されたり、なんらかのタイプの浮動小数点例外を発生させる可能性のある算術演算が除去されるためです。

   例えば、**X + 0.0** は X には折り畳まれません。これは、IEEE 規則では **-0.0 + 0.0 = 0.0** であり、これは **-X** になるからです。他の例として、最適

化の中には、符号が誤ったゼロの結果を生成する最適化を実行するものもあります。 例えば、**X - Y * Z** の結果は **-0.0** になります。これは、元の計算では **0.0** になります。

ほとんどの場合、結果の相違はアプリケーションにとって重要ではないため、 **-O3** ではこれらの最適化が許可されます。

3. 浮動小数点式が書き換えられる場合があります。

   再配置によって共通副次式を抽出する機会が得られる場合などには、 **a*b*c** などの計算を **a*c*b** に書き換える場合があります。浮動小数点計算の再配置の別の例として、除算を逆数による乗算で置き換えることが挙げられます。

4. **-O3** を指定すると、**-qhot** または **-qhot=level=1** が明示的に指定されている場合を除き、**-qhot=level=0** が暗黙指定されます。

-qfloat=rsqrt は、**-O3** ではデフォルトで設定されます。

-qmaxmem=1 は、 **-O3** ではデフォルトで設定され、最適化を実行するときにコンパイラーが必要なだけ多くのメモリーを使用できるようにします。

組み込み関数は、 `-O3` では **errno** を変更しません。

整数除算命令の最適化は、 **-O3** であっても非常に危険であると見なされます。

optimize オプションを **-qflttrap** オプションと一緒に指定したときのコンパイラーの振る舞いについては、165 ページの『-qflttrap』を参照してください。

-qstrict および **-qstrict_induction** コンパイラー・オプションを使用して、プログラムのセマンティクスを変更する可能性がある **-O3** の影響をオフにすることができます。**-qstrict** を **-O3** と一緒に指定すると、**-O2** で実行されるすべての最適化のほか、ループの最適化も呼び出されます。**-qstrict** コンパイラー・オプションへの参照は、**-O3** オプションの前に指定することも、後に指定することもできます。

-O3 コンパイラー・オプションの後に **-O** オプションを指定すると、**-qignerrno** がオンのままになります。

-O3 と **-qhot=level=1** が有効な場合、コンパイラーはソース・コード内のすべての呼び出しを、同等の MASS ライブラリー関数、および可能であれば、ベクトル・バージョンへの呼び出しを持つ標準の数学ライブラリー関数に置換します。

-O4 | optimize|opt=4

このオプションは **-O3** と同じですが、以下の点が異なります。

- コンパイルを行うマシンのアーキテクチャーに対して、 **-qarch** および **-qtune** オプションを設定する。
- コンパイル・マシンの特性に最も適した **-qcache** オプションを設定する。
- **-qhot** オプションが設定される。
- **-qipa** オプションが設定される。

注: -O、**-qcache**、**-qhot**、**-qipa**、**-qarch**、および **-qtune** オプションを後で設定すると、**-O4** オプションによって暗黙指定された設定がオーバーライドされます。

このオプションは「最後のオプションが選択される」という競合解決規則に従っており、そのため **-O4** によって変更されるオプションを後から変更できます。

例えば、**-O4 -qarch=ppc** を指定すると、積極的に内部手順の最適化を行うことができ、同時にコードの移植性を維持することもできます。

-O5 | optimize|opt=5
このオプションは **-O4** と同じですが、以下の点が異なります。

- **-qipa=level=2** オプションを設定して、完全なプロシージャー間のデータ・フローおよび別名の分析を実行します。

注:

-O、**-qcache**、**-qipa**、**-qarch**、および **-qtune** オプションを後で設定すると、**-O5** オプションによって暗黙指定された設定がオーバーライドされます。

使用法

最適化のレベルを上げても、追加の分析によって最適化の機会がさらに検出されるかどうかに応じて、さらにパフォーマンスが向上する場合と向上しない場合があります。

最適化を伴うコンパイルでは、他のコンパイルよりも多くの時間およびマシンのリソースが必要になる場合があります。

最適化によってステートメントが移動または削除される場合があるため、通常は、デバッグ・プログラムのための **-g** フラグと一緒に最適化を指定しないでください。作成されるデバッグ情報が正確でなくなる場合があります。

-O 以上の最適化を使用すると、**-qtbtable=small** が暗黙に指定されます。生成されるトレースバック・テーブルには、関数名やパラメーター情報は入りません。

事前定義マクロ

- __OPTIMIZE__ は、**-O | O2** が有効な場合は 2、**-O3| O4 | O5** が有効な場合は 3 に事前定義されます。それ以外の場合は定義が解除されます。
- __OPTIMIZE_SIZE__ は、**-O | -O2 | -O3 | -O4 | -O5** および **-qcompact** が有効な場合は 1 に事前定義されます。それ以外の場合は定義が解除されます。

例

`myprogram.c` をコンパイルして最適化するには、以下のように入力します。

```
xlc myprogram.c -O3
```

関連情報

- 183 ページの『-qhot』
- 211 ページの『-qipa』
- 292 ページの『-qpdf1、-qpdf2』
- 352 ページの『-qstrict』
- 364 ページの『-qtbtable』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』

-qoptdebug

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

高水準の最適化で使用されると、デバッガーが読み取ることができる最適化済みの疑似コードを含むファイルを作成する。

.optdbg 拡張子付きの出力は、 **-qoptdebug** を指定してコンパイルされた各ソース・ファイルで作成されます。このファイルに含まれた情報を使用すると、最適化の下でコードが実際に振る舞う方法が理解しやすくなります。

構文

デフォルト

-qnooptdebug

使用法

-qoptdebug は、高水準最適化プログラムを使用可能にするオプション、すなわち **-O3** 以上の最適化レベル、または **-qhot**、**-qsmp**、**-qpdf**、または **-qipa** と一緒に使用するときのみ、有効になります。コンパイルおよびリンクの両方のステップでこのオプションを使用することができます。コンパイル・ステップでこのオプションを指定すると、それぞれのソース・ファイルに 1 つの出力ファイルが生成されます。**-qipa** リンク・ステップでこのオプションを指定すると、単一の出力ファイルが生成されます。

.optdbg ファイルの命名規則は以下のとおりです。

- .optdbg ファイルがコンパイル・ステップで生成される場合、ファイル名は、コンパイル・ステップの出力ファイル名に基づいて命名されます。
- .optdbg ファイルがリンク・ステップで生成される場合、ファイル名は、リンク・ステップの出力ファイル名に基づいて命名されます。

-qoptdebug オプションを **-qipa** と併用して 1 つのステップでコンパイルとリンクを行う場合、.optdbg ファイルはリンク・ステップのみで生成されます。

-g または **-qlinedebug** オプションを使用して、デバッガーが使用できるデバッグ情報を組み込む必要があります。

このオプションの使用例について詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイドXL C/C++ 最適化およびプログラミング・ガイド*」の『最適化プログ

ラムをデバッグするのに役立つ -qoptdebug の使用』を参照してください。

関連情報

- 279 ページの『-O、-qoptimize』
- 183 ページの『-qhot』
- 211 ページの『-qipa』
- 292 ページの『-qpdf1、-qpdf2』
- 335 ページの『-qsmp』
- 173 ページの『-g』
- 260 ページの『-qlinedebug』

-qoptfile

カテゴリー

コンパイラーのカスタマイズ

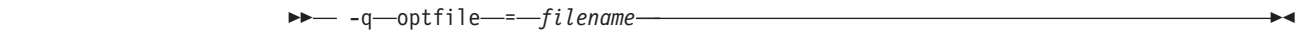
プラグマ同等物

なし。

目的

コンパイルで使用する追加コマンド行オプションのリストが入ったファイルを指定する。

構文



デフォルト

なし。

パラメーター

filename
追加コマンド行オプションのリストが入ったファイルの名前を指定します。*filename* には、相対パスまたは絶対パスを指定することができ、またパスを指定しないことも可能です。これは、1 行に 1 つ以上のコマンド行オプションが含まれたプレーン・テキスト・ファイルです。

使用法

オプション・ファイルのフォーマットは、以下の規則に従います。

- コマンド行の場合と同じ構文を使用して、ファイル内に組み込むオプションを指定します。オプション・ファイルは空白文字で区切られたオプションのリストです。特殊文字 ¥n、¥v、および ¥t は空白文字を示します。 (これらの文字の効果はすべて同じです。)

- 単一引用符または二重引用符のペアに囲まれた文字ストリングは、コンパイラーに 1 つのオプションとして渡されます。
- オプション・ファイルにコメントを含めることができます。コメント行は # 文字から開始され、行末まで続きます。コンパイラーはコメントおよび空の行を無視します。

コンパイラーは処理時に **-qoptfile** オプションをコマンド行から削除し、指定した残りの後続オプションの前に、ファイルに含まれているオプションを連続して挿入します。

-qoptfile オプションは、オプション・ファイル内でも有効です。別のオプション・ファイルが含まれているファイルは、深さ優先方式で処理されます。コンパイラーは、オプション・ファイルの組み込みにおける循環を検出して無視することで、無限ループを回避します。

-qoptfile と **-qsaveopt** を同じコマンド行で指定すると、**-qsaveopt** に対して元のコマンド行が使用されます。各オプション・ファイルの内容を示すために、オプション・ファイルごとに改行が含まれます。ファイルに含まれているオプションは、コンパイルされたオブジェクト・ファイルに保存されます。

事前定義マクロ

なし。

例

これは、オプション・ファイルの指定例です。

```
$ cat options.file
# To perform optimization at -O4 level, and high-order
# loop analysis and transformations during optimization
-O4 -qhot
# To generate position-independent code
-qpic

$ xlC -qlist -qoptfile=options.file -qipa test.c
```

上記の例は、以下の呼び出しと同等です。

```
$ xlC -qlist -O4 -qhot -qpic -qipa test.c
```

これは、循環のある -qoptfile が含まれたオプション・ファイルの指定例です。

```
$ cat options.file2
# To perform optimization at -O4 level, and high-order
# loop analysis and transformations during optimization
-O4 -qhot
# To include the -qoptfile option in the same option file
-qoptfile=options.file2
# To generate position-independent code
-qpic
# To produce a compiler listing file
-qlist

$ xlC -qlist -qoptfile=options.file2 -qipa test.c
```

上記の例は、以下の呼び出しと同等です。

```
$ xlC -qlist -O4 -qhot -qpic -qlist -qipa test.c
```

これは、循環のない -qoptfile が含まれたオプション・ファイルの指定例です。

```
$ cat options.file1
-O4 -qhot
-qoptfile=options.file2
-qfixed

$ cat options.file2
-qfree

$ xlC -qoptfile=options.file1 test.c
```

上記の例は、以下の呼び出しと同等です。

```
$ xlC -O4 -qhot -qfree -qfixed test.c
```

これは、同じコマンド行における -qsaveopt と -qoptfile の指定例です。

```
$ cat options.file3
-O4
-qhot

$ xlC -qsaveopt -qipa -qoptfile=options.file3 test.c -c

$ what test.o
test.o:
opt f xlC -qsaveopt -qipa -qoptfile=options.file3 test.c -c
optfile options.file3 -O4 -qhot
```

関連情報

- 325 ページの『-qsaveopt』

-p、-pg、-qprofile

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

コンパイラーが作成するオブジェクト・ファイルをプロファイル用に準備する。

プロファイル・オプションとコンパイルすると、コンパイラーは、それぞれのルーチンが呼び出される回数を数えるモニター・コードを作成します。コンパイラーは、各サブプログラムの開始ルーチンを開始時にモニター・サブルーチンを呼び出す開始ルーチンに置換します。コンパイルされたプログラムを実行して、それが正常に終了すると、記録された情報は gmon.out ファイルに書き込まれます。**gprof** コマンドを使用してランタイム・プロファイルを生成することができます。

構文

デフォルト

適用されません。

使用法

別々のステップでコンパイルおよびリンクを行うときには、両方のステップでプロファイル・オプションを指定してください。

-qtbtable オプションを設定しない場合は、プロファイル・オプションが完全なトレースバック・テーブルを生成します。

事前定義マクロ

なし。

例

myprogram.c をコンパイルしてプロファイル・データを組み込むには以下のように入力します。

```
xlc myprogram.c -p
```

コンパイルおよび リンクするには、必ずプロファイル・オプションの 1 つを指定してください。例を以下に示します。

```
xlc myprogram.c -p -c
xlc myprogram.o -p -o program
```

関連情報

- 364 ページの『-qtbtable』
- **gprof** コマンドについての詳細は、ご使用のオペレーティング・システムの資料を参照してください。

-P

カテゴリー

出力制御

プラグマ同等物

なし。

目的

コンパイラー呼び出しに指定されたソース・ファイルをプリプロセスし、コンパイルせずにそれぞれの入力ファイルごとにプリプロセスされた出力ファイルを作成する。

プリプロセスされた出力ファイルの名前は、入力ファイルに .i というサフィックスが付いたものになります。

構文

デフォルト

デフォルトでは、ソース・ファイルをプリプロセスし、コンパイルして、リンクすると実行可能ファイルが生成されます。

使用法

認識されないファイル名サフィックスを持つソース・ファイルは、.i サフィックスが付くものを除き、C ファイルとしてプリプロセスされます。

-qppline が指定されない場合は、#line ディレクティブは生成されません。

行継続シーケンスは除去されて、ソース行が連結されます。

-P オプションは、以下の例外を除き、改行文字を含むすべての空白文字を保持します。

- すべてのコメントはシングル・スペースに縮小される (**-C** が指定されていない場合)。
- プリプロセス・ディレクティブの末尾にある改行は保持されない。
- 関数形式のマクロに対する引数の前後にある空白文字は保持されない。

-P オプションは、 **-E** オプションによってオーバーライドされます。**-P** オプションは、**-c**、**-o**、および **-qsyntaxonly** オプションをオーバーライドします。

事前定義マクロ

なし。

関連情報

- 124 ページの『-C、-C!』
- 150 ページの『-E』
- 304 ページの『-qppline』
- 361 ページの『-qsyntaxonly (C のみ)』

-qpack_semantic

カテゴリー

移植性とマイグレーション

プラグマ同等物

なし。

目的

#pragma pack ディレクティブの構文およびセマンティクスを制御する。

構文

デフォルト

-qpack_semantic=ibm

パラメーター

gnu

#pragma pack の GCC 構文およびセマンティクスを使用します。このサブオプションの効果は以下のとおりです。

- 現行のパッキング値は、パッキング・スタックから独立しています。
- 値は、**push** パラメーターによってパック・スタックに配置されるのみです。**push** パラメーターによって指定された値のみを **pop** パラメーターによってスタックから除去することができます。
- **#pragma pack** ディレクティブがネストされた集合体内で指定される場合は、外部に影響を与え、集合体も含みます。

ibm

#pragma pack の IBM 構文およびセマンティクスを使用します。このサブオプションの効果は以下のとおりです。

- 現行のパッキング値は、パッキング・スタックの先頭に自動的に配置されます。
- **push** パラメーターはありません。値は、**pop** パラメーターによってスタックから除去することができます。
- **#pragma pack** ディレクティブがネストされた集合体内で指定される場合は、後続の集合体にのみ影響を与えます。つまり、影響を受ける可能性があるのは内部の集合体のみです。

これらのサブオプションの構文、セマンティクス、および例について詳しくは、440 ページの『#pragma pack』を参照してください。

使用法

GCC でコンパイルされたアプリケーションを移植して、プラグマ・ディレクティブの GCC バージョンとのソース・レベルの互換性を保存する必要がない限りは、このオプションを使用する必要はありません。

事前定義マクロ

なし。

例

例については、『 440 ページの『#pragma pack』』を参照してください。

-qpath

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

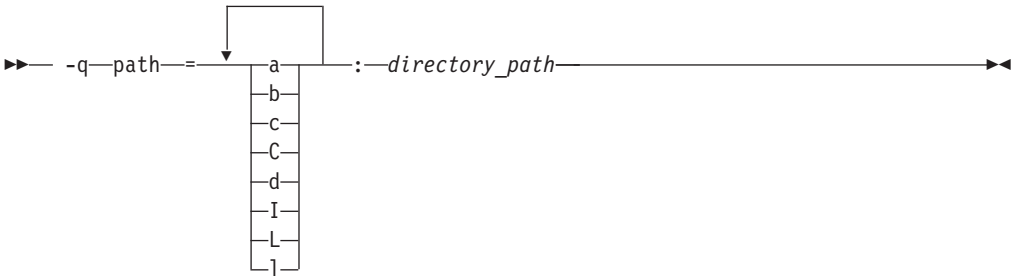
なし。

目的

コンパイラー、アセンブラー、リンカー、およびプリプロセッサーなどの XL C/C++ コンポーネントの代替パス名を指定する。

XL C/C++ コンポーネントの一部またはすべてについて複数のレベルを保持し、どれを使用するかを指定できるようにする場合、このオプションを使用できます。このオプションは、**-B** および **-t** オプションよりも優先されます。

構文



デフォルト

デフォルトで、コンパイラーは構成ファイルで定義されたコンパイラー・コンポーネントのパスを使用します。

パラメーター

directory_path
代替プログラムが配置されているディレクトリーへのパス。

以下の表は、**-qpath** パラメーターとコンポーネント名との対応を示しています。

パラメーター	説明	コンポーネント名
a	アセンブラー	as
b	低水準最適化プログラム	xlCcode
c	コンパイラーのフロントエンド	xlcentry、xlCentry
C++ C	C++ コンパイラーのフロントエンド	xlCentry
d	逆アセンブラー	dis

パラメーター	説明	コンポーネント名
I (大文字の i)	高水準最適化プログラム、コンパイル・ステップ	ipa
L	高水準最適化プログラム、リンク・ステップ	ipa
l (小文字の L)	リンカー	ld

使用法

-qpath オプションは、 **-F**、**-t**、および **-B** オプションをオーバーライドします。

事前定義マクロ

なし。

例

`/lib/tmp/mine/` の代替 **xlc** コンパイラーを使用して `myprogram.c` をコンパイルするには、次のように入力します。

```
xlc myprogram.c -qpath=c:/lib/tmp/mine/
```

`/lib/tmp/mine/` の代替リンカーを使用して `myprogram.c` をコンパイルするには、次のように入力します。

```
xlc myprogram.c -qpath=l:/lib/tmp/mine/
```

関連情報

- 121 ページの『-B』
- 156 ページの『-F』
- 362 ページの『-t』

-qpdf1、-qpdf2

カテゴリー

最適化およびチューニング

プラグマ同等物

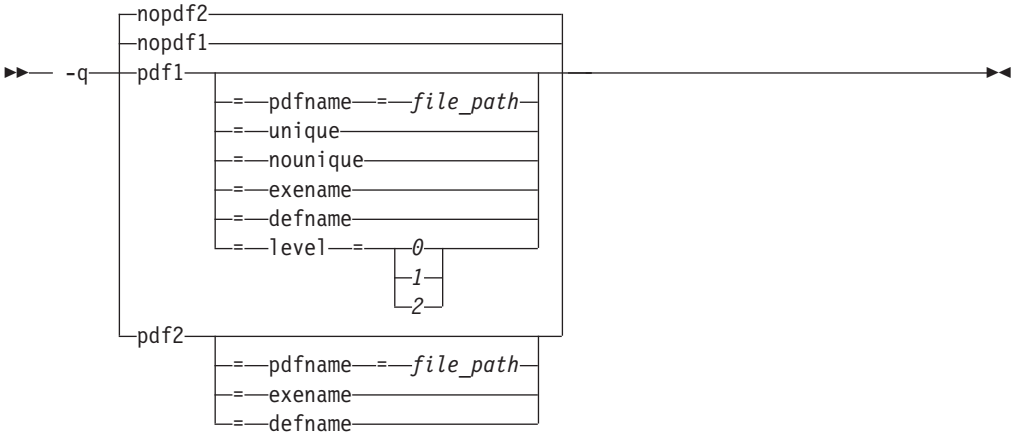
なし。

目的

profile-directed feedback (PDF) を介して最適化を調整する。サンプル・プログラムの実行から得られた結果を使用して、条件付き分岐の付近や頻繁に実行されるコード・セクションでの最適化を改善します。

分岐の使用頻度およびコード・ブロックの実行頻度の分析に基づいて、標準的な使用シナリオについてアプリケーションを最適化します。

構文



デフォルト

-qnopdf1、-qnopdf2

パラメーター

defname
PDF ファイルをデフォルトのファイル名に戻します。

exename
-o オプションによって指定された出力ファイル名に従って、生成される PDF ファイルの名前を指定します。例えば、**-qpdf1=exename -o func func.c** を使用すると、.func_pdf という PDF ファイルを生成できます。

level=0 | 1 | 2
結果のアプリケーションによって生成される、各種レベルのプロファイル情報を指定します。以下の表に、各レベルでサポートされるプロファイル情報のタイプを示します。正符号 (+) は、プロファイル・タイプがサポートされることを示しています。

表 30. 各 -qpdf1 レベルでサポートされるプロファイル・タイプ

	Level		
プロファイル・タイプ	**0**	**1**	**2**
ブロック・カウンター・プロファイル	+	+	+
呼び出しカウンター・プロファイル	+	+	+
単一パス・プロファイル	+	+	
値プロファイル		+	+
マルチパス・プロファイル			+
キャッシュ・ミス・プロファイル			+

- **-qpdf1=level=0** は基本レベルであり、**-qpdf1=level=1** よりファイル・サイズは小さくなり、コンパイル時間は短くなります。
- **-qpdf1=level=1** はデフォルト・レベルです。これは、IBM XL C/C++ for Linux, V11.1 より前のリリースの **-qpdf1** と等価です。
- **-qpdf1=level=2** は、**-qpdf1=level=0** および **-qpdf1=level=1** よりも積極性が高くなります。これは、PDF 機能が使用可能なすべての最適化レベルでサポートされます。

  注:
  - 個々のコンピューターでは、**-qpdf1=level=2** オプションを指定してコンパイルされたアプリケーションは、一度に 1 つしか実行できません。
  - キャッシュ・ミス・プロファイルは **SLES11 SP1** システムで使用可能であり、POWER5 以上のプロセッサーのみで使用できます。
  - キャッシュ・ミス・プロファイル情報にはいくつかのレベルがあります。各種レベルのキャッシュ・ミス・プロファイル情報を収集する場合は、PDF_PM_EVENT 環境変数を L1MISS、L2MISS、または L3MISS (使用可能な場合) に適宜設定します。一度に装備できるキャッシュ・ミス・プロファイル情報は 1 つのレベルのみです。L2 cache-miss はデフォルト・レベルです。
  - キャッシュ・ミス・プロファイルのために、アプリケーションを指定プロセッサーにバインドする場合は、PDF_BIND_PROCESSOR 環境変数を設定します。デフォルトではプロセッサー 0 が設定されます。

pdfname= *file_path*

PDF ファイルおよび任意の既存の PDF マップ・ファイルのディレクトリーと名前を指定します。デフォルトでは、PDFDIR 環境変数が設定されている場合、コンパイラーは PDF ファイルおよび PDF マップ・ファイルを、PDFDIR によって指定されたディレクトリーに配置します。それ以外の場合、PDFDIR 環境変数が設定されていなければ、コンパイラーはこれらのファイルを現行作業ディレクトリーに配置します。 PDFDIR 環境変数が設定されているが、指定したディレクトリーが存在しない場合は、コンパイラーは警告メッセージを出します。 **-qpdf1=unique** オプションが指定されていない場合、PDF マップ・ファイルの名前は、PDF ファイルの名前に従います。例えば、**-qpdf1=pdfname=/home/joe/func** オプションを指定した場合は、生成される PDF ファイルの名前は `func` になり、PDF マップ・ファイルの名前は `func_map` になります。どちらのファイルも `/home/joe` ディレクトリーに配置されます。 **pdfname** サブオプションを使用すると、同じディレクトリーを使用して複数の実行可能アプリケーションを同時に実行できます。これは、動的ライブラリーでの PDF プロセスを使用したチューニングで特に役立ちます。

unique | **nounique**

PDF トレーニング・ステップで複数のプロセスが同じ PDF ファイルに書き込みを行う場合、**-qpdf1=unique** オプションを使用すると、単一の PDF ファイルがロッキングされるのを防ぐことができます。このオプションは、ランタイム時に各プロセスに対して固有の PDF ファイルを作成するかどうかを指定します。PDF ファイル名は *<pdf_file_name>.<pid>* です。*<pdf_file_name>* は、`._pdf` (デフォルト) であるか、または他の **-qpdf1** サブオプションによって指定されます。このサブオプションには、**pdfname**、**exename**、および **defname** が含まれます。 *<pid>* は、PDF トレーニング・ステップで実行中のプロセスの ID で

す。例えば、**-qpdf1=unique:pdfname=abc** オプションを指定し、12345678 と 87654321 の ID を持つ 2 つの PDF トレーニング用プロセスがある場合、2 つの PDF ファイル `abc.12345678` および `abc.87654321` が生成されます。

注:

- **-qpdf1=unique** が指定されている場合、1 つの PDF マップ・ファイルのみが生成されます。 PDF マップ・ファイルのデフォルト名は `._pdf_map` です。
- **-qpdf1=unique** が指定されている場合、プロセス ID がサフィックスとして付いた複数の PDF ファイルが生成されます。 PDF トレーニング・ステップの後で、**mergepdf** プログラムを使用して、これらすべての PDF ファイルを 1 つにマージする必要があります。

使用法

PDF プロセスは、以下の 3 つのステップから成ります。

1. **-qpdf1** オプションおよび最小の最適化レベル **-O2** を指定して、プログラムをコンパイルします。デフォルトの `._pdf_map` という名前の PDF マップ・ファイルと結果のアプリケーションが生成されます。
2. 標準的なデータ・セットを使用して、結果のアプリケーションを実行します。プロファイル情報が、デフォルトで `._pdf` という名前の PDF ファイルに書き込まれます。このステップは、PDF トレーニング・ステップと呼ばれています。
3. **-qpdf2** オプションおよび **-qpdf1** オプションに対して使用した最適化レベルを指定して、プログラムを再コンパイルおよびリンクまたは再リンクします。結果のアプリケーションの実行時に収集されるプロファイル情報に従って、**-qpdf2** プロセスが最適化を微調整します。

注:

- **showpdf** ユーティリティーは PDF マップ・ファイルを使用して、プロファイル情報の一部をテキストまたは XML フォーマットで表示します。詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『showpdf によるプロファイル情報の表示』を参照してください。プロファイル情報を表示する必要がない場合は、**-qpdf1** フェーズで **-qnoshowpdf** オプションを指定して、PDF マップ・ファイルが生成されないようにします。 **-qnoshowpdf** について詳しくは、「*XL C/C++ コンパイラー・リファレンス*」の『**-qshowpdf**』を参照してください。
- オプション **-O4**、**-O5**、または任意のレベルのオプション **-qipa** が有効な場合に、**-qpdf1** または **-qpdf2** オプションをリンク・ステップで指定し、コンパイル・ステップでは指定していないときは、コンパイラーによって警告メッセージが出されます。このメッセージは、プログラムを再コンパイルしてプロファイル情報をすべて取得する必要があることを示します。
- **-qpdf1** フェーズで **-qpdf1=pdfname** オプションを使用した場合は、**-qpdf2** フェーズで **-qpdf2=pdfname** オプションを使用して、コンパイラーが正しい PDF ファイルを認識できるようにする必要があります。この規則は、**-qpdf[1|2]=exename** オプションにも適用されます。

コンパイラーは、**-qpdf2** フェーズで、0 から 100 の範囲の数字が含まれた情報メッセージを出します。 **-qpdf1** フェーズから **-qpdf2** フェーズまでの間にプログラムを変更していない場合、数字は 100 になります。これは、プログラムの最適化にすべてのプロファイル情報が使用できることを意味します。数字が 0 の場合は、プロ

ファイル情報が完全に古くなっていることを意味し、コンパイラーは一切情報を使用することができません。数字が 100 未満の場合は、**-qpdf1** オプションを指定してプログラムを再コンパイルし、プロファイル情報を再生成することを選択できます。

単一パス・プロファイル

単一パス・プロファイルは、**-qpdf1** フェーズのレベル 0 および 1 でサポートされます。プログラムを再コンパイルして、**-qpdf1=level=0** オプションまたは **-qpdf1=level=1** オプションのいずれかを使用した場合、コンパイラーは、新規アプリケーションの生成前に、既存の PDF ファイルおよび既存の PDF マップ・ファイル (存在する場合) を削除します。

マルチパス・プロファイル

マルチパス・プロファイルは、**-qpdf1** フェーズのレベル 2 でサポートされます。結果のアプリケーションをトレーニングするときにプログラムを **-qpdf1=level=2** オプションを指定してコンパイルしたら、プログラムを **-qpdf1=level=2** オプションを使用して再コンパイルすることができます。前に収集されたプロファイル情報を使用して、さらなるインスツルメンテーションがガイドされます。結果のアプリケーションを再度トレーニングすると、プロファイル情報は、デフォルトでは `._pdf.1` という名前の新規プロファイル・ファイルに書き込まれます。このコンパイルと PDF トレーニングを複数回繰り返した場合は、PDF ファイルが最大で 5 回生成されます (`._pdf.1` から `._pdf.5`)。コンパイラーは、すべての PDF ファイル名が使用されたことを検出すると、警告メッセージを出して、最後の PDF ファイル `._pdf.5` を上書きします。 **-qpdf1=level=2** オプションを指定してプログラムをコンパイルしたときに、コンパイラーが PDF ファイルを 1 つも読み取れない場合、コンパイラーは警告メッセージを出して、PDF ファイルが見つからないことを示します。**-qpdf1=level=0** オプションまたは **-qpdf1=level=1** オプションを使用すると初期プロファイル情報を取得でき、**-qpdf1=level=2** オプションを使用するとさらにプロファイル情報を取得できます。

注:

- **-qnoshowpdf** オプションを指定しなかった場合は、デフォルト名 `._pdf_map`、`._pdf.1_map`、...、`._pdf.5_map` を使用して、PDF ファイルに対応する PDF マップ・ファイルも生成されます。
- **-qpdf2=pdfname** オプションを使用して PDF ファイルを指定する場合、`.1` から `.5` の数値のサフィックスで終了しないファイル名を指定してください。これに従わないと、コンパイラーが間違ったファイルを検索してしまいます。例えば、**-qpdf2** フェーズで **-qpdf2=pdfname=func.2** オプションを指定したすると、コンパイラーは `func.2`、`func.2.1`、`func.2.2`、`func.2.3` という名前の PDF ファイルを検索しますが、これらのファイルは存在しない可能性があります。数値のサフィックスを使用せずに **-qpdf2=pdfname=func** オプションを指定した場合、コンパイラーは `func`、`func.1`、`func.2`、`func.3` を検索します。

その他の関係オプション

以下のオプションを **-qpdf1** オプションと共に使用できます。

-qprefetch

-qprefetch=assistthread オプションを実行してデータ・プリフェッチ支援スレッドを生成すると、コンパイラーは delinquent ロード (キャッシュ・ミス頻発ロード) 情報を使用して分析を行い、それらのスレッドを生成します。delinquent ロード (キャッシュ・ミス頻発ロード) 情報は、**-qpdf1=level=2** オプションを使用して、動的プロファイルから収集できます。詳しくは、『-qprefetch』を参照してください。

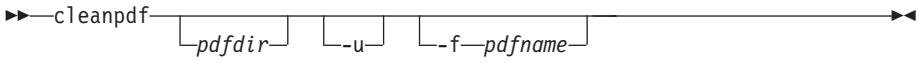
-qshowpdf

追加情報をプロファイル・ファイルに提供します。 詳しくは、『 330 ページの『-qshowpdf』』を参照してください。

PDF を使用する場合の推奨される手順については、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『プロファイル指示フィードバックの使用』を参照してください。

`/opt/ibm/xlC/13.1.0/bin/` にある以下のユーティリティー・プログラムは、プロファイル情報が書き込まれるディレクトリーを管理するために使用できます。

cleanpdf



プロセス ID サフィックスを持つ PDF ファイルを含む、すべての PDF ファイルまたは指定された PDF ファイルを削除します。プログラムを変更して PDF プロセスをもう一度実行する場合、プロファイル情報を除去するとランタイム・オーバーヘッドが減ります。

pdfdir 削除される PDF ファイルが含まれているディレクトリーを指定します。 *pdfdir* が指定されていない場合、ディレクトリーは PDFDIR 環境変数により設定されます。PDFDIR が設定されていない場合、ディレクトリーは現行ディレクトリーです。

-f *pdfname*

削除する PDF ファイルの名前を指定します。指定すると、命名規則 *pdfname.<multiple_pass_profiling_times>* を持つファイルも削除されます (該当する場合)。 *<multiple_pass_profiling_times>* は、1 から 5 の数値サフィックスです。

`-f` *pdfname* が指定されていない場合、`._pdf` と、命名規則 `._pdf.`*<multiple_pass_profiling_times>* を持つファイル (該当する場合) が削除されます。

-u *pdfname* によって指定された PDF ファイルと、以下の命名規則を持つファイルを (該当する場合に) 削除します。

- *pdfname.<pid>*。ここで *<pid>* は、PDF トレーニング・ステップの実行中のプロセスの ID です。
- *pdfname.<multiple_pass_profiling_times>.<pid>*

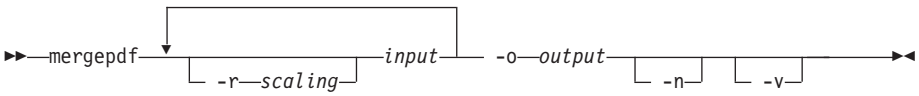
`-f` *pdfname* が指定されていない場合、`._pdf` と、以下の命名規則を持つファイル (該当する場合) が削除されます。

- `._pdf.`*<pid>*

- ._pdf.*<multiple_pass_profiling_times>.<pid>*

cleanpdf を実行するのは、特定のアプリケーションの PDF プロセスが終了した場合のみにしてください。これに従わないと、そのアプリケーションで PDF プロセスを使用して再開する場合に、**-qpdf1** を指定してすべてのファイルをもう一度コンパイルする必要が生じます。

mergepdf



複数の PDF ファイルを単一の PDF ファイルにマージします。

-r *scaling*
PDF ファイルの位取りの比率を指定します。この値はゼロより大でなければならず、整数または浮動小数点のいずれの値にすることもできます。指定されない場合は、1.0 の率が想定されます。

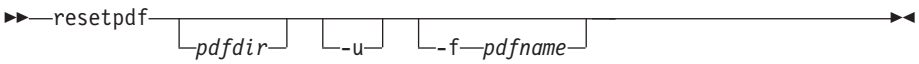
input PDF 入力ファイルの名前、または PDF ファイルが入っているディレクトリーの名前を指定します。

-o *output*
PDF 出力ファイルの名前、またはマージされた出力が書き込まれるディレクトリーの名前を指定します。

-n これを指定すると、PDF ファイルは正規化されません。指定されていない場合は、ユーザー定義の位取り係数を適用する前に、内部で計算された比率を基に **mergepdf** がファイルを正規化します。

-v 冗長モードを指定し、内部およびユーザー指定のスケーリング比率が標準出力に表示されるようにします。

resetpdf



cleanpdf と同じ。

showpdf

PDF ファイルおよび PDF マップ・ファイルに書き込まれるプロファイル情報の一部を表示します。このコマンドを使用するには、まずプログラムをコンパイルし、**-qpdf1** オプションを使用する必要があります。詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『showpdf によるプロファイル情報の表示』を参照してください。

事前定義マクロ

なし。

例

以下の例では、**-qpdf1=level=0** オプションを使用して、実行時のインスツルメンテーション・オーバーヘッドの可能性を低減します。

```
#Compile all the files with -qpdf1=level=0
xlc -qpdf1=level=0 -O3 file1.c file2.c file3.c

#Run with one set of input data
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

以下の例では、**-qpdf1=level=1** オプションを使用します。

```
#Compile all the files with -qpdf1
xlc -qpdf1 -O3 file1.c file2.c file3.c

#Run with one set of input data
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

以下の例では、**-qpdf1=level=2** オプションを使用して、キャッシュ・ミス・プロファイル情報を収集します。

```
#Compile all the files with -qpdf1=level=2
xlc -qpdf1=level=2 -O3 file1.c file2.c file3.c

#Set PM_EVENT=L2MISS to gather L2 cache-miss profiling
#information
export PDF_PM_EVENT=L2MISS

#Run with one set of input data
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

以下の例では、複数の実行で **-qpdf1=level=2** オプションを使用して、各種キャッシュ・レベルでキャッシュ・ミス・プロファイル情報を収集します。

```
#Compile all the files with -qpdf1=level=2
xlc -qpdf1=level=2 -O3 file1.c file2.c file3.c

#Set PM_EVENT=L1MISS to gather L1 cache-miss profiling
#information
export PDF_PM_EVENT=L1MISS

#Run with one set of input data
./a.out < sample.data

#Set PM_EVENT=L2MISS to gather L2 cache-miss profiling
#information
export PDF_PM_EVENT=L2MISS
```

```
#Run with one set of input data
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

以下の例では、マルチパス・プロファイルのプロセスを示します。

```
#Compile all the files with -qpdf1=level=2. The static profiling
#information is recorded in a file named ._pdf_map by default
xlc -qpdf1=level=2 -O3 file1.c file2.c file3.c

#Run with one set of input data, the profiling information
#is recorded in a file named ._pdf by default
./a.out < sample.data

#Recompile all the files with -qpdf1=level=2 again
#The compiler reads the previous profiling information, refines
#instrumentation, and generates a new instrumented
#executable. The static profiling information
#is recorded in ._pdf.1_map
xlc -qpdf1=level=2 -O3 file1.c file2.c file3.c

#Run it again, the profiling information is recorded in
#._pdf.1
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

以下の例では、PDF_BIND_PROCESSOR 環境変数の使用について示します。

```
#Compile all the files with -qpdf1=level=1
xlc -qpdf1=level=1 -O3 file1.c file2.c file3.c

#Set PDF_BIND_PROCESSOR environment variable so that
#all processes for this executable are run on Processor 1
export PDF_BIND_PROCESSOR=1

#Run executable with sample input data
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

以下の例では、**-qpdf[1|2]=exename** オプションの使用について示します。

```
#Compile all the files with -qpdf1=exename
xlc -qpdf1=exename -O3 -o final file1.c file2.c file3.c

#Run executable with sample input data
./final < typical.data

#List the content of the directory
 >ls -lrta

 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
```

```
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
 -rwxr-xr-x 1 user staff 12243 Dec 05 17:00 final
 -rwxr-Sr-- 1 user staff 762 Dec 05 17:03 .final_pdf

#Recompile all the files with -qpdf2=exename
xlc -qpdf2=exename –O3 -o final file1.c file2.c file3.c

#The program is now optimized using PDF information
```

以下の例では、**-qpdf[1|2]=pdfname** オプションの使用について示します。

```
#Compile all the files with -qpdf1=pdfname.The static profiling
#information is recorded in a file named final_map
xlc -qpdf1=pdfname=final -O3 file1.c file2.c file3.c

#Run executable with sample input data.The profiling
#information is recorded in a file named final
./a.out < typical.data

#List the content of the directory
 >ls -lrta

 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
 -rwxr-xr-x 1 user staff 12243 Dec 05 18:30 a.out
 -rwxr-Sr-- 1 user staff 762 Dec 05 18:32 final

#Recompile all the files with -qpdf2=pdfname
xlc -qpdf2=pdfname=final -O3 file1.c file2.c file3.c

#The program is now optimized using PDF information
```

関連情報

- 330 ページの『-qshowpdf』
- 211 ページの『-qipa』
- -qprefetch
- 315 ページの『-qreport』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』
- 29 ページの『ランタイム環境変数』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『プロファイル指示フィードバック』

-qphsinfo

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

各コンパイル・フェーズで標準出力にかかった時間を報告する。

構文

デフォルト

-qnophsinfo

使用法

出力はそれぞれのフェーズごとに *number1/number2* の形式を取ります。ここで、*number1* はコンパイラーによって使用される CPU 時間を表し、*number2* はコンパイル時間と CPU がシステム呼び出しの処理に費やす時間の合計を表します。

`-qphsinfo` によって報告される時間は秒単位です。

事前定義マクロ

なし。

例

C myprogram.c をコンパイルし、コンパイルの各フェーズごとにかかった時間を報告させるには、以下のように入力します。

```
xlc myprogram.c -qphsinfo
```

出力は以下のようになります。

```
C Init    - Phase Ends;   0.010/  0.040
IL Gen    - Phase Ends;   0.040/  0.070
W-TRANS   - Phase Ends;   0.000/  0.010
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
```

-O4 を使用して同じプログラムをコンパイルすると、以下のようになります。

```
C Init    - Phase Ends;   0.010/  0.040
IL Gen    - Phase Ends;   0.060/  0.070
IPA       - Phase Ends;   0.060/  0.070
IPA       - Phase Ends;   0.070/  0.110
W-TRANS   - Phase Ends;   0.060/  0.180
OPTIMIZ   - Phase Ends;   0.010/  0.010
REGALLO   - Phase Ends;   0.010/  0.020
AS        - Phase Ends;   0.000/  0.000
```

C++ myprogram.C をコンパイルし、コンパイルの各フェーズごとにかかった時間を報告させるには、以下のように入力します。

```
xlc++ myprogram.C -qphsinfo
```

出力は以下のようになります。

```
Front End - Phase Ends;   0.004/  0.005
W-TRANS   - Phase Ends;   0.010/  0.010
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
```

-O4 を使用して同じプログラムをコンパイルすると、以下のようになります。

```
Front End - Phase Ends;   0.004/  0.006
IPA       - Phase Ends;   0.040/  0.040
IPA       - Phase Ends;   0.220/  0.280
W-TRANS   - Phase Ends;   0.030/  0.110
OPTIMIZ   - Phase Ends;   0.030/  0.030
REGALLO   - Phase Ends;   0.010/  0.050
AS        - Phase Ends;   0.000/  0.000
```

-qpic

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

共有ライブラリーでの使用に適した位置独立コードを生成する。

構文

デフォルト

- 32 ビット・コンパイル・モードで **-qmkshrobj** オプションが指定されていない場合は **-qnopic**。
- 64 ビット・コンパイル・モードの **-qpic=small**。
- **-qmkshrobj** オプションが指定されている場合は **-qpic=small**。

パラメーター

small
: 32 ビット・モードでのグローバル・オフセット・テーブル (GOT) または 64 ビット・モードでの目次 (TOC) のサイズが 64 Kb 以下であると想定するようコンパイラーに指示します。**-qpic=small** が有効になっている場合、コンパイラーは GOT アクセスまたは TOC アクセスごとに 1 つの命令を生成します。

large
: 64 ビット・モードで TOC のサイズが 64 Kb より大きいと想定するようにコンパイラーに指示します。**-qpic=large** が有効になっている場合は、コンパイラーは TOC アクセスごとに 2 つの命令を生成してアクセス範囲を拡大します。これにより、目次 (TOC) のサイズが 64 Kb を超える場合でも TOC オーバーフロー条件が回避されます。

サブオプションなしで **-qpic** を指定することは、**-qpic=small** と同じです。

使用法

-q64 が有効になっている場合は、**-qpic** は常に使用可能になります。

`-qpic=large -qtls -q64` を指定した場合は、スレッド・ローカル・ストレージ (TLS) シンボルは、**-qpic=large** の影響を受けません。

アプリケーション内のコンパイル単位ごとに、異なる TOC アクセス・オプションを使用できます。

注: TOC サイズが 64K より大きいアプリケーションの場合は、**-qpic=large** を使用すると、パフォーマンスが改善される場合があります。ただし、TOC が 64K よりも小さいアプリケーションの場合は、**-qpic=large** を使用すると、プログラムがスローダウンします。 **-qpic=large** を使用するかどうかを決定するには、まず **-qpic=small** を指定してプログラムをコンパイルします。オーバーフロー・エラー・メッセージが生成された場合は、代わりに **-qpic=large** を使用します。

事前定義マクロ

なし。

例

共有ライブラリー `libmylib.so` をコンパイルするには、以下のコマンドを使用します。

```
xlc mylib.c -qpic=small -c -o mylib.o
xlc -qmkshrobj mylib -o libmylib.so.1
```

関連情報

- 104 ページの『-q32、-q64』
- 277 ページの『-qmkshrobj』

-qppline

カテゴリー

オブジェクト・コード制御

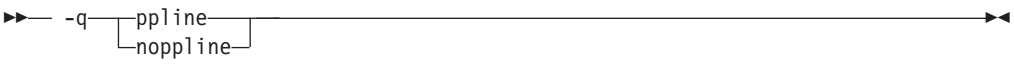
プラグマ同等物

なし。

目的

-E または **-P** オプションと一緒に使用すると、`#line` ディレクティブの生成を使用可能または使用不可にする。

構文



デフォルト

- **-P** が有効な場合は **-qnoppline**
- **-E** が有効な場合は **-qppline**

使用法

-C オプションは、**-E** または **-P** オプションを指定しないと無効になります。 **-E** オプションを指定すると、行ディレクティブが標準出力に書き込まれます。 **-P** オプションを指定すると、行ディレクティブが出力ファイルに書き込まれます。

事前定義マクロ

なし。

例

`myprogram.c` をプリプロセスして出力を `myprogram.i` に書き込み、`#line` ディレクティブを生成するには以下のようにします。

```
xlc myprogram.c -P -qppline
```

関連情報

- 150 ページの『-E』
- 288 ページの『-P』

-qprefetch

カテゴリー

最適化およびチューニング

プラグマ同等物

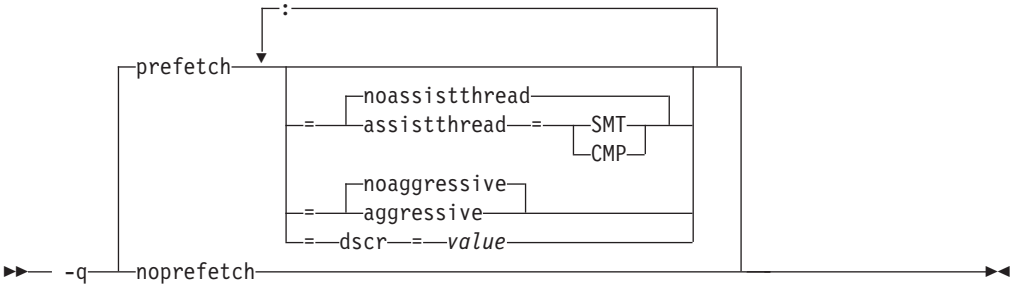
なし。

目的

コード・パフォーマンスを改善する機会がある場所に、プリフェッチ命令を自動的に挿入する。

-qprefetch が有効な場合、コンパイラーはコンパイルされたコードでプリフェッチ命令を挿入する可能性があります。**-qnoprefetch** が有効な場合、プリフェッチ命令はコンパイルされたコードに挿入されません。

構文



デフォルト

-qprefetch=noassistthread:noaggressive:dscr=0

パラメーター

assistthread | noassistthread
キャッシュ・ミス率の高いアプリケーションを使用している場合、**-qprefetch=assistthread** を使用して、データ・プリフェッチの支援スレッドを活用できます。このサブオプションは、支援スレッドを最適化レベル **-O3 -qhot** 以上で活用するようにコンパイラーを誘導します。**-qprefetch=assistthread** を指定しない場合は、**-qprefetch=noassistthread** が暗黙指定されます。

CMP
チップ・マルチプロセッサー (CMP) アーキテクチャーに基づいたシステムの場合は、**-qprefetch=assistthread=cmp** を使用できます。

SMT
同時マルチスレッディング (SMT) アーキテクチャーに基づいたシステムの場合は、**-qprefetch=assistthread=smt** を使用できます。

注: CMP と SMT のどちらも指定しなければ、コンパイラーはシステムのアーキテクチャーに基づいたデフォルト設定を使用します。

aggressive | noaggressive
このサブオプションは、積極的なデータ・プリフェッチを最適化レベル **-O3** 以上で生成するようにコンパイラーを誘導します。**aggressive** を指定しない場合は、暗黙的に **-qprefetch=noaggressive** が使用されます。

dscr
dscr サブオプションに値を指定して、アプリケーションのランタイム・パフォーマンスを向上できます。コンパイラーは、データ・ストリーム制御レジスター (DSCR) を指定の dscr の値に設定し、ハードウェア・プリフェッチ・エンジンを制御します。この値が有効なのは、-qarch=pwr8 が有効であり、最適化レベルが -O2 以上である場合のみです。 dscr のデフォルト値は 0 です。

value

dscr に指定する値は、0 以上で、64 ビット符号なし整数として表現可能でなければなりません。それ以外の場合、コンパイラーは警告メッセージを出して、dscr を 0 に設定します。コンパイラーには 10 進数および 16 進数の両方の数字を指定でき、16 進数の場合にはプレフィックス 0x が必要で

す。値の範囲は、システム・アーキテクチャーによって異なります。詳しくは、POWER Architecture の製品情報を参照してください。複数の dscr 値を指定した場合、最後の 1 つが有効になります。

使用法

-qnoprefetch オプションは、**__prefetch_by_stream** などの組み込み関数がプリフェッチ命令を生成するのを妨げません。

-qprefetch=assistthread を実行すると、コンパイラーは delinquent ロード (キャッシュ・ミス頻発ロード) 情報を使用して分析を行い、プリフェッチ支援スレッドを生成します。delinquent ロード (キャッシュ・ミス頻発ロード) 情報は、組み込み __mem_delay 関数 (const void *delinquent_load_address、const unsigned int delay_cycles) を通じて提供するか、**-qpdf1=level=2** を使用して動的プロファイルから収集することができます。

-qpdf を使用して **-qprefetch=assistthread** を呼び出す場合、以下のように従来の 2 ステップの PDF 呼び出しを使用する必要があります。

1. **-qpdf1=level=2** を実行する
2. **-qpdf2 -qprefetch=assistthread** を実行する

例

以下に、__MEM_DELAY で支援スレッドを使用したコードの生成方法を示します。

初期のコード:

```
int y[64], x[1089], w[1024];

  void foo(void){
    int i, j;
    for (i = 0; i &l; 64; i++) {
      for (j = 0; j < 1024; j++) {

        /* what to prefetch? y[i]; inserted by the user */
        __mem_delay(&y[i], 10);
        y[i] = y[i] + x[i + j] * w[j];
        x[i + j + 1] = y[i] * 2;
    }
  }
}
```

支援スレッド生成コード:

```
void foo@clone(unsigned thread_id, unsigned version)

{ if (!1) goto lab_1;

/* version control to synchronize assist and main thread */
if (version == @2version0) goto lab_5;

goto lab_1;

lab_5:

@CIV1 = 0;

do { /* id=1 guarded */ /* ~2 */

if (!1) goto lab_3;
```

```
@CIV0 = 0;

do { /* id=2 guarded */ /* ~4 */

/* region = 0 */

/* __dcbt call generated to prefetch y[i] access */
__dcbt(((char *)&y + (4)*(@CIV1)))
@CIV0 = @CIV0 + 1;
} while ((unsigned) @CIV0 < 1024u); /* ~4 */

lab_3:
@CIV1 = @CIV1 + 1;
} while ((unsigned) @CIV1 < 64u); /* ~2 */

lab_1:

return;
}
```

関連情報

- 113 ページの『-qarch』
- 183 ページの『-qhot』
- 292 ページの『-qpdf1、-qpdf2』
- 315 ページの『-qreport』
- 644 ページの『__mem_delay』

-qprint

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

リストを使用可能にするか、または抑制する。

-qprint が有効な場合、リストは、リストを作成する他のコンパイラー・オプションによって要求されると、使用可能になります。**-qnoprint** が有効な場合、リスト作成オプションが指定されているかどうかにかかわらず、すべてのリストが抑制されます。

構文

デフォルト

-qprint

使用法

-qnoprint を使用して、すべてのリスト作成オプションおよび同等のプラグマを、それらが指定されている場所にかかわらずオーバーライドすることができます。これらのオプションは以下のとおりです。

- -qattr
- -qlist
- -qlistopt
- -qsource
- -qxref

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルし、幾つかのファイルが **#pragma options source** および類似するディレクティブを持っていたとしても、すべてのリスト表示を抑制するには、以下のように入力します。

```
xlc myprogram.c -qnoprint
```

-qpriority (C++ のみ)

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options priority、#pragma priority

目的

静的オブジェクトの初期化の優先順位を指定する。

C++ 標準では、同じ変換単位内のすべてのグローバル・オブジェクトが上から下へと構成される必要がありますが、別の変換単位で宣言されたオブジェクトの順序付けを強制しません。**-qpriority** オプションおよび **#pragma priority** ディレクティブを使用すると、構成順序を同じロード・モジュール内で宣言されたすべての静的オブジェクトに強制することができます。これらのオブジェクトのデストラクターは、終了時に逆順で実行されます。

構文

オプション構文

プラグマ構文

デフォルト

デフォルトの優先順位は 65535 になります。

パラメーター

number
101 から 65535 までの範囲内の整数リテラル。低い値は優先順位が高いことを示しています。高い値は優先順位が低いことを示しています。 *number* を指定しないと、コンパイラーは 65535 を想定します。

使用法

複数の **#pragma priority** を変換単位内で指定することができます。1 つのプラグマで指定される優先度の値は、このプラグマの後または次のプラグマの前で宣言されるすべてのグローバル・オブジェクトの構造体に適用されます。ただし、標準と整合性をとるために、同じ変換単位内で指定された優先度の値は厳密に増加させる必要があります。同じ優先順位の値を持つオブジェクトは、宣言順に構成されます。

#pragma priority の効果があるのは 1 つのロード・モジュール内においてのみです。そのため、**#pragma priority** を使用して別のロード・モジュール内のオブジェクトの構造順序を制御することはできません。

注: C++ の変数属性 `init_priority` は、クラスの型の共有変数に優先順位を割り当てるためにも使用することができます。詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『init_priority 変数属性』を参照してください。

例

ファイル `myprogram.C` をコンパイルしてオブジェクト・ファイル `myprogram.o` を作成し、そのファイル内のオブジェクトの初期化の優先順位を 2000 にするには、次のように入力します。

```
xlc++ myprogram.C -c -qpriority=2000
```

関連情報

- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『ライブラリーでの静的オブジェクトの初期化』

-qprocimported、-qproclocal、-qprocunknown

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options proclocal、 #pragma options procimported、 #pragma options procunknown

目的

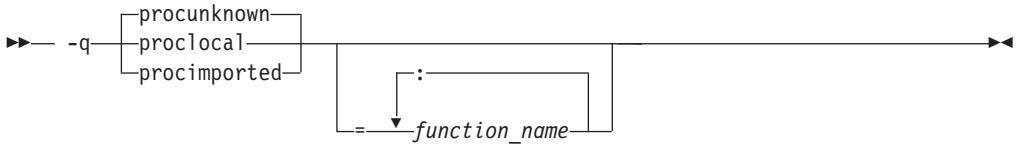
64 ビットのコンパイルで、関数にローカル、インポート、または不明のマークを付ける。

ローカル関数は、それを呼び出す関数と静的にバインドされます。そのような関数に対する呼び出しのために、より小さくて高速なコードが生成されます。**-qproclocal** オプションまたはプラグマを使用して、ローカルであるとコンパイラーが想定できる関数を指定することができます。

インポートされる関数は、ライブラリーの共有部分と動的にバインドされます。インポートされるものとしてマークされた関数の呼び出しのために生成されるコードは、不明としてマークされた関数のために生成されるデフォルトのコード・シーケンスよりサイズが大きくなる場合がありますが、高速になります。 **-qprocimported** オプションまたはプラグマを使用して、インポート済みとコンパイラーが想定できる関数を指定することができます。

不明の関数は、リンク中に、静的または動的のいずれかでバインドされたオブジェクトに解決されます。 **-qprocunkown** オプションまたはプラグマを使用して、不明であるとコンパイラーが想定できる関数を指定することができます。

構文



デフォルト

-qprocunkown: このコンパイラーは、すべての関数の定義が不明であると想定します。

パラメーター

function_name
ローカル、インポート済み、または不明であるとコンパイラーが想定する関数の名前 (指定したオプションによって異なる)。 *function_name* を指定しない場合、コンパイラーはすべての関数がローカル、インポート、または不明であると想定します。

C++ 名前は、マングル名を使用して指定する必要があります。C++ マングル名を取得するには、**-c** コンパイラー・オプションを使用してオブジェクト・ファイルのみにソースをコンパイルし、その結果生じるオブジェクト・ファイルで **nm** オペレーティング・システム・コマンドを使用します。(名前マングリングを回避するための、宣言に対する `extern "C"` リンケージ指定子の使用について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『名前マングリング』も参照してください。)

使用法

このオプションは 64 ビットのコンパイルにのみ適用されます。

ローカルとしてマークされた関数が共有ライブラリー関数に解決される場合は、リンカーは、エラーを検出して警告を出します。インポートとしてマークされた関数が静的にバインドされたオブジェクトに解決される場合は、生成されるコードは、不明の関数のために生成されるデフォルトのコード・シーケンスよりサイズが大きく、実行が遅くなる場合があります。

関数が以下の条件をすべて満たす場合、コンパイラーは、最後の実行可能ファイルにパフォーマンス・ロスが存在する可能性があることを示す警告メッセージを出します。

- ローカル定義を持っている。
- インポートまたは不明としてマークが付いている。
- IBM protected、hidden、または internal の visibility 属性を持っている。 IBM

関数名なしでこれらのオプションを複数指定する場合は、指定した最後のオプションが使用されます。複数のオプションを指定するときに同じ関数名を指定する場合は、最後のオプションが使用されます。

事前定義マクロ

なし。

例

`myprogram.c` をアーカイブ・ライブラリー `oldprogs.a` と共にコンパイルして、以下のように指定するには、

- 関数 `fun` および `sun` をローカルと指定する。
- 関数 `moon` および `stars` をインポートとして指定する。
- 関数 `venus` を不明と指定する。

以下のコマンドを使用します。

```
xlc myprogram.c oldprogs.a -qprolocal=fun(int):sun()
  -qprocimported=moon():stars(float) -qprocunknown=venus()
```

ローカルとマークされた関数が共有ライブラリー関数に解決される次の例が **-qproclocal** でコンパイルされる場合は、以下のようになります。

```
int main(void)
{
    printf("Just in function foo1()¥n");
    printf("Just in function foo1()¥n");
}
```

リンカー・エラーが発生します。この問題を訂正するには、呼び出されるルーチンを共有オブジェクトからインポートされたものとして明示的にマークを付けます。この場合、ソース・ファイルを再コンパイルし、`-qproclocal -qprocimported=printf` でコンパイルすることによって `printf` を明示的にインポートとマークします。

関連情報

- 142 ページの『-qdataimported、-qdatalocal、-qtocdata』
- 389 ページの『-qvisibility』
- 421 ページの『#pragma GCC visibility push、#pragma GCC visibility pop』

-qproto (C のみ)

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options [no]proto

目的

浮動小数点引数をプロトタイプ化されていない関数へ渡すためのリンケージ規約を指定する。

proto が有効な場合、関数がプロトタイプ化されていなくても、コンパイラーは、関数呼び出しでの引数の型が、関数定義の対応するパラメーターと同じであると想定します。プロトタイプ化されていない関数は、浮動小数点引数と一緒に呼び出される場合、実際に浮動小数点引数を予期していると表明することによって、コンパイラーは浮動小数点レジスターで浮動小数点引数を排他的に受け渡すことができます。**noproto** が有効な場合、コンパイラーはこのように想定しないため、浮動小数点および汎用レジスターで浮動小数点パラメーターを受け渡す必要があります。

構文

デフォルト

-qnoproto

使用法

コンパイラーがプロトタイプ化されていない関数を許可する場合のみ、このオプションは有効です。つまり、**cc** または **xlc** 呼び出しコマンド、あるいは **-qlanglvl** オプションを使用して、**classic | extended | extc89 | extc99** に設定します。

事前定義マクロ

なし。

例

関数がプロトタイプ化されていない場合にも、コンパイラーが浮動小数点パラメーターの標準リンケージ規約を使用できるように `my_c_program.c` をコンパイルするには、次のように入力します。

```
xlc my_c_program.c -qproto
```

-r

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

別の ID コマンド呼び出しでの入力ファイルとして使用する実行不能出力ファイルを生成する。このファイルには未解決のシンボルも含むことができる。

構文

デフォルト

適用されません。

使用法

このフラグを使用して作成されたファイルは、別のコンパイラー呼び出しまたは ID コマンド呼び出しの入力ファイルとして使用されることが予想されます。

事前定義マクロ

なし。

例

`myprogram.c` および `myprog2.c` を単一のオブジェクト・ファイル `mytest.o` にコンパイルするには、以下のように入力します。

```
xlc myprogram.c myprog2.c -r -o mytest.o
```

-R

カテゴリー

リンク

プラグマ同等物

なし。

目的

プログラムの実行時に要求された共有ライブラリーがこれらのディレクトリーで検索されるように、リンク時に、共有ライブラリーの検索パスを実行可能ファイルに書き込む。

構文

デフォルト

デフォルトでは、標準のディレクトリーしか組み込まれません。デフォルトで設定されるディレクトリーについては、コンパイラー構成ファイルを参照してください。

使用法

構成ファイルとコマンド行の両方に **-R***directory_path* オプションが指定されている場合は、実行時に構成ファイルに指定したパスが最初に検索されます。

-R コンパイラー・オプションは累積です。コマンド行で後に指定された **-R** オプションは、前の **-R** によって指定されたディレクトリー・パスを置換するのではなく、このパスへの追加を行います。

事前定義マクロ

なし。

例

ディレクトリー `/usr/tmp/old` が動的ライブラリー `libspfiles.so` の標準ディレクトリーと共に実行時に検索されるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -lspfiles -R/usr/tmp/old
```

関連情報

- 226 ページの『-L』

-qreport

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

コードのセクションをどのように最適化したかを示すリスト・ファイルを作成する。

コマンド行で指定された各ソース・ファイルごとに、リスト・ファイルが、.lst サフィックス付きで生成されます。自動並列化またはベクトル化を使用可能にするオプションと一緒に使用した場合、リスト・ファイルには疑似 C コード・リストと、プログラム・ループが並列化または最適化される方法の要約が示されます。また、このレポートには、特定のループを並列化またはベクトル化できなかった理由を示す診断情報も含まれています。例えば、**-qreport** を **-qsimd=auto** とともに使用すると、ループのベクトル化を回避できる、スライド 1 以外の参照を示すメッセージが出されます。

またコンパイラーは、ロード・ストリームとストア・ストリームの両方を含む、指定されたループ用に作成されたストリームの数も報告します。 この情報は、リスト・ファイルの `Loop Transformation` セクションに組み込まれます。この情報を使用して、アプリケーション・コードを理解し、プログラム・コードのパフォーマンス・チューニングを行うことができます。例えば、基礎のアーキテクチャーがサポートする数より多いストリームがあるループを配布することができます。POWER5 プロセッサーはロード・ストリーム・プリフェッチをサポートし、POWER6 以上のプロセッサーはロードとストアの両方のストリーム・プリフェッチをサポートします。

構文

デフォルト

-qnoreport

使用法

-qreport でループ変換リストを生成するには、コマンド・ラインで以下のいずれかのオプションも指定する必要があります。

- **-qsimd=auto**
- **-qsmp**
- **-qhot=level=2** および **-qsmp**
- **-O5**
- **-qipa=level=2**

-qreport で PDF 情報をリストで生成するには、以下のオプションをコマンド行に指定する必要があります。

- **-qpdf2 -qreport**

-qreport で並列変換リストまたは並列パフォーマンス・メッセージを生成するには、コマンド・ラインで以下のいずれかのオプションを指定する必要もあります。

- **-qsmp**
- **-O5**
- **-qipa=level=2**

データ再編成情報を生成するには、最適化レベル **-qipa=level=2** または **-O5** を **-qreport** と一緒に指定します。再編成には、配列分割、配列転置、メモリー割り振りのマージ、配列インターリービング、および配列合体が含まれます。

データ・プリフェッチ挿入ロケーションに関する情報を生成するには、最適化レベルの **-qhot**、または **-qhot** を暗黙指定する他のいずれかのオプションを **-qreport** と一緒に使用します。この情報は、リスト・ファイルの LOOP TRANSFORMATION SECTION に表示されます。さらに、**-qprefetch=assistthread** を使用してプリフェッチ支援スレッドを生成する場合、「データ・プリフェッチの支援スレッドが生成されました。」というメッセージも、リスト・ファイルの LOOP TRANSFORMATION SECTION に表示されます。

リスト・ファイルの LOOP TRANSFORMATION SECTION に、ループ・ネストに対して行われた積極的なループ変換と並列化のリストを生成するには、最適化レベル **-qhot=level=2** および **-qsmp** を **-qreport** と一緒に使用します。

疑似 C コード・リストは、コンパイル可能であるというわけではありません。プログラムに疑似 C コードを読み込んだり、名前が疑似 C コード・リストに出ている内部ルーチンを明示的に呼び出したりしないでください。

事前定義マクロ

なし。

例

myprogram.c をコンパイルして、コンパイラー・リストに、ループがどのように最適化されるかを示すレポートが含まれるようにするには、以下を入力します。

```
xlc -qhot -O3 -qreport myprogram.c
```

myprogram.c をコンパイルして、コンパイラー・リストに、並列化されたループがどのように変換されるかを示すレポートも含まれるようにするには、以下を入力します。

```
xlc_r -qhot -qsmp -qreport myprogram.c
```

関連情報

- 183 ページの『-qhot』
- 331 ページの『-qsimd』
- 211 ページの『-qipa』
- 335 ページの『-qsmp』
- 284 ページの『-qoptdebug』
- 305 ページの『-qprefetch』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『最適化プログラムをデバッグするのに役立つ -qoptdebug の使用』

-qreserved_reg

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

スタック・ポインター、フレーム・ポインター、またはその他の固定の役割を除き、指定されたレジスターのリストがコンパイル中に使用できないことを示します。

このオプションは、グローバル・レジスター変数または手書きのアセンブラー・コードを使用する他のモジュールと連動させる必要があるモジュールで使用してください。

構文

デフォルト

適用されません。

パラメーター

register_name
ターゲット・プラットフォーム上で有効なレジスター名。有効なレジスターは以下のとおりです。

r0 から r31
汎用レジスター

f0 から f31
浮動小数点レジスター

v0 から v31
ベクトル・レジスター (選択されたプロセッサー専用)

使用法

-qreserved_reg は累積です。例えば、`-qreserved_reg=r14` と `-qreserved_reg=r15` を指定することは、 `-qreserved_reg=r14:r15` を指定することと同等です。

重複するレジスター名は無視されます。

事前定義マクロ

なし。

例

`myprogram.c` が汎用レジスター r3 および r4 を予約するように指定するには、次のように入力します。

```
xlc myprogram.c -qreserved_reg=r3:r4
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『指定されたレジスターの変数』

-qrestrict (C のみ)

カテゴリー

最適化およびチューニング

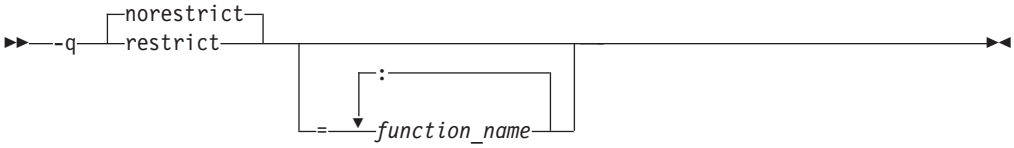
プラグマ同等物

なし。

目的

このオプションを指定することは、指定された関数内でポインター・パラメーターに `restrict` キーワードを追加することと同等ですが、ソース・ファイルを変更する必要がない点が異なる。

構文



デフォルト

-qnorestrict。これは、ソース内に **restrict** 属性を指定した場合を除き、どの関数ポインター・パラメーターも制限されないことを意味します。

使用法

function_name を指定しない場合、すべての関数内のポインター・パラメーターは **restrict** として扱われます。それ以外の場合は、リストされた関数内のポインター・パラメーターのみが **restrict** として扱われます。

function_name は、コロンで区切ったリストです。

このオプションを使用するとアプリケーションのパフォーマンスを改善できますが、このポインター制限を不正に表明すると、コンパイラーが、誤った前提に基づいて正しくないコードを生成する可能性があります。**-qrestrict** を指定しないで再コンパイルするとアプリケーションが正しく動作する場合、アサーションが誤っている可能性があります。その場合は、このオプションを使用しないでください。

注:

- **-qnokeyword=restrict** を使用しても、**-qrestrict** オプションに影響はありません。
- **-qalias=norestrict** と **-qrestrict** オプションの両方を使用する場合、最後に指定したほうが使用されます。

事前定義マクロ

なし。

例

ポインター・アクセスを制限するようにコンパイラーに命令して、myprogram.c をコンパイルするには、以下のように入力します。

```
xlc -qrestrict myprogram.c
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の restrict 型修飾子
- 「*XL C/C++ ランゲージ・リファレンス*」のキーワード
- -qkeyword
- -qalias

-qro

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options ro、#pragma strings

目的

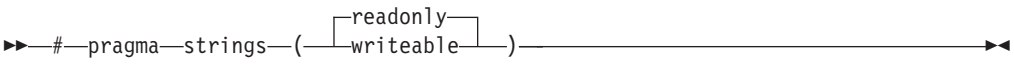
ストリング・リテラルのストレージ・タイプを指定する。

ro または **strings=readonly** が有効な場合、ストリングが読み取り専用ストレージに配置されます。**noro** または **strings=writeable** が有効な場合、ストリングが読み取り/書き込みストレージに配置されます。

構文

オプション構文

プラグマ構文



デフォルト

C ストリングは、**cc** 以外のすべての呼び出しコマンドで読み取り専用です。 **cc** 呼び出しコマンドを使用する場合、ストリングは書き込み可能です。

C++ ストリングは読み取り専用です。

パラメーター

readonly (プラグマのみ)
ストリング・リテラルは読み取り専用メモリー中に配置されます。

writeable (プラグマのみ)
ストリング・リテラルは読み取り/書き込みメモリー中に配置されます。

使用法

ストリング・リテラルを読み取り専用メモリーに配置すると、ランタイム・パフォーマンスが改善され、ストレージを節約できます。ただし、読み取り専用ストリング・リテラルの変更を試行するコードによりメモリー・エラーが生成されることがあります。

プラグマはファイル内ですべてのソース・ステートメントの前になければなりません。

事前定義マクロ

なし。

例

ストレージ・タイプが writable になるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qnoro
```

関連情報

- 320 ページの『-qro』
- 『-qroconst』

-qroconst

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options [no]roconst

目的

定数値のストレージ・ロケーションを指定する。

roconst が有効な場合、定数が読み取り専用ストレージに配置されます。**noroconst** が有効な場合、定数が読み取り/書き込みストレージに配置されます。

構文

デフォルト

- C **cc** とその派生物を除くすべてのコンパイラー呼び出しの **-qroconst**。**cc** 呼び出しとその派生物の **-qnoroconst**。
- C++ **-qroconst**

使用法

定数値を読み取り専用メモリーに配置することにより、ランタイム・パフォーマンスを改善して、ストレージを節約し、共有アクセスを提供することができます。 ただし、読み取り専用の定数値をコードが変更しようとすると、メモリー・エラーが生成されます。

-qroconst オプションのコンテキストでは、「定数値」とは、`const` によって修飾された変数 (`const` によって修飾された文字、整数、浮動小数点、列挙、構造体、共用体、および配列を含む) を指します。以下の構造体は、このオプションの影響は受けません。

- `volatile` で修飾された変数、および `volatile` 変数を含む集合体 (構造体や共用体など)
- ポインター、およびポインター・メンバーを含む複合集合体
- ブロック・スコープを持つ自動型および静的型
- 未初期化の型
- `const` によってすべてのメンバーが修飾された通常の構造体
- アドレスである初期化指定子、または非アドレス値へのキャストである初期化指定子

-qroconst オプションでは、 **-qro** オプションは暗黙指定されません。ストリング・リテラル (**-qro**) と定数値 (**-qroconst**) の両方のストレージ特性を指定したい場合は、これらのオプションを両方とも指定しなければなりません。

事前定義マクロ

なし。

関連情報

- 320 ページの『-qro』

-qrtti (C++ のみ)

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options rtti

目的

`typeid` 演算子および `dynamic_cast` 演算子による例外処理および使用のための実行時型識別 (RTTI) 情報を生成する。

構文

デフォルト

-qrtti

使用法

ランタイム・パフォーマンス改良するためには、**-qnortti** の設定により RTTI 情報の生成を抑止します。

-qrtti コンパイラー・オプションを指定する場合は、以下の影響に注意してください。

- **-qrtti** が指定されているときには、仮想関数テーブルの内容は異なります。
- オブジェクトをまとめてリンクするとき、対応するソース・ファイルはすべて、正しい **-qrtti** オプションを指定してコンパイルしなければなりません。
- ライブラリーを混合オブジェクト (いくつかのオブジェクトには **-qrtti** が指定されており、その他のオブジェクトには **-qnortti** が指定されている) でコンパイルすると、未定義シンボル・エラーとなる場合があります。

事前定義マクロ

- __RTTI_ALL__ は、**-qrtti** が有効である場合は 1 に定義され、それ以外の場合は、定義されません。
- __NO_RTTI__ は、**-qnortti** が有効である場合は 1 に定義され、それ以外の場合は、定義されません。

関連情報

- 151 ページの『-qeh (C++ のみ)』

-s

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

出力ファイルからシンボル・テーブル、行番号情報、および再配置情報をストリップする。

このコマンドは、オペレーティング・システムの **strip** コマンドと同等です。

構文

デフォルト

シンボル・テーブル、行番号情報、および再配置情報が出力ファイルに組み込まれます。

使用法

-s を指定するとスペースの節約になりますが、**-g** などのオプションを使用してデバッグ情報を生成するときには、従来のデバッグ・プログラムの実用性が制限されます。

事前定義マクロ

なし。

関連情報

- 173 ページの『-g』

-S

カテゴリー

出力制御

プラグマ同等物

なし。

目的

ソース・ファイルごとにアセンブラー言語ファイルを生成する。

結果として得られるファイルは .s サフィックスを持ち、アセンブルされてオブジェクト .o ファイルまたは実行可能ファイル (a.out) を作成することができます。

構文

デフォルト

適用されません。

使用法

アセンブラーは、どのコンパイラー呼び出しコマンドを使用しても呼び出すことができます。例を以下に示します。

```
xlc myprogram.s
```

これによってアセンブラーが呼び出され、成功すると、リンカーが呼び出されて実行可能ファイル a.out が作成されます。

-E または **-P** と一緒に **-S** を指定した場合は、 **-E** または **-P** が優先されます。この優先順位は、コマンド行に指定された順序に関係なく保持されます。

ソース・ファイルを 1 つしか提供しない場合に限り、 **-o** オプションを使用して、作成されるファイルの名前を指定することができます。例えば、以下は無効 です。

```
xlc myprogram1.c myprogram2.c -o -S
```

事前定義マクロ

なし。

例

myprogram.c をコンパイルして、アセンブラー言語ファイル myprogram.s を作成するには、以下のように入力します。

```
xlc myprogram.c -S
```

このプログラムをアセンブルしてオブジェクト・ファイル myprogram.o を作成するには、以下のように入力します。

```
xlc myprogram.s -c
```

myprogram.c をコンパイルして、アセンブラー言語ファイル asmprogram.s を作成するには、以下のように入力します。

```
xlc myprogram.c -S -o asmprogram.s
```

関連情報

- 150 ページの『-E』
- 288 ページの『-P』

-qsaveopt

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

ソース・ファイルのコンパイルに使用するコマンド行オプション、構成ファイルで指定されたユーザーの構成ファイル名とオプション、コンパイル中に呼び出される各コンパイラー・コンポーネントのバージョンとレベル、およびその他の情報を対応するオブジェクト・ファイルに保管する。

構文

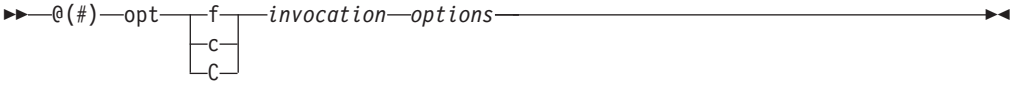
デフォルト

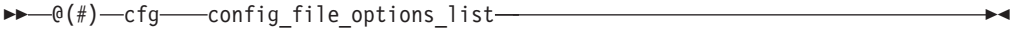
-qnosaveopt

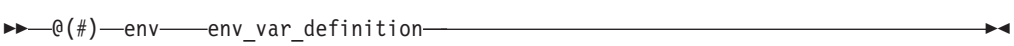
使用法

このオプションは、オブジェクト・ファイル (.o) へコンパイルするときのみ有効です (つまり、**-c** オプションを使用)。各オブジェクトには複数のコンパイル単位が含まれている場合がありますが、コマンド行オプションの 1 つのコピーのみが保存されます。pragma ディレクティブで指定されたコンパイラー・オプションは、無視されます。

以下の形式を使用して、コマンド行のコンパイラー・オプション情報は、ストリングとしてオブジェクト・ファイルにコピーされます。







ここで、

f FORTRAN 言語のコンパイルを指定します。

c C 言語のコンパイルを指定します。

C C++ 言語のコンパイルを指定します。

invocation
コンパイルで使用されるコマンド (例えば、**xlc**) を表示します。

options コマンド行に指定されたコマンド行オプションのリスト。個々のオプションはスペースで区切られています。

config_file_options_list
コンパイルで有効な、すべての構成ファイルにある **options** 属性で指定されたオプションのリスト。スペースで区切られています。

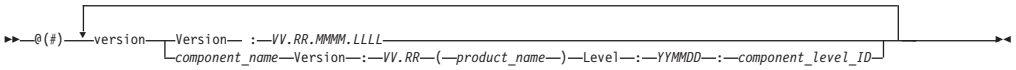
env_var_definition
コンパイラーによって使用される環境変数。現在、**XLC_USR_CONFIG** のみがリストされています。

注: このオプションは常時使用することができますが、環境変数 **XLC_USR_CONFIG**が設定されている場合にのみ、対応する情報が生成されます。

環境変数 **XLC_USR_CONFIG** について詳しくは、『コンパイル時およびリンク時の環境変数』を参照してください。

注: コマンド行オプションのストリングは、64 K バイトを超えると切り捨てられます。

コンパイル中に呼び出される各コンポーネントのバージョンとレベルと、コンパイラー・バージョンおよびリリース情報は、次の形式でオブジェクト・ファイルにも保存されます。



ここで、

V バージョンを表します。

R リリースを表します。

M 変更を表します。

L レベルを表します。

component_name
このコンパイルで呼び出されたコンポーネントを指定します (低水準最適化プログラムなど)。

product_name
コンポーネントが属する製品を示します (例えば、C/C++ または Fortran)。

YYMMDD
インストールされた更新の年、月、日を表します。インストールされた更新が基本レベルである場合、そのレベルは BASE として表示されます。

component_level_ID
インストール済みコンポーネントのレベルに関連付けられている ID を表します。

この情報をオブジェクト・ファイルに書き込まずに、単に標準出力に出力する場合は、**-qversion** オプションを使用します。

事前定義マクロ

なし。

例

t.c を次のコマンドでコンパイルします。

```
xlc t.c -c -qsaveopt -qhot
```

作成された t.o オブジェクト・ファイルで**strings -a** コマンドを発行すると、以下のような情報が作成されます。

```
opt c /opt/ibm/xlC/13.1.0/bin/xlc t.c -c -qsaveopt -qhot
cfg -qlanglvl=extc99 -qcpluscmt -qkeyword=inline -qalias=ansi -qtls -D_CALL_SYSV
 -D__null=0 -D__NO_MATH_INLINES -qtls
version IBM XL C/C++ for Linux, V13.1
version Version: 13.01.0000.0000
version Driver Version: 13.01(C/C++) Level: YYMMDD

version C Front End Version : 13.01(C/C++)  Level: YYMMDD
version High-Level Optimizer Version: 13.01(C/C++) and 15.01(Fortran)  Level: YYMMDD
version Low-Level Optimizer Version: 13.01(C/C++) and 15.01(Fortran) Level: YYMMDD
```

1 行目では、c は、C として使用されたソースを示し、/opt/ibm/xlC/13.1.0/bin/xlc は、使用された呼び出しコマンドを示し、-qhot -qsaveopt は、コンパイル・オプションを示しています。

残りの行は、コンパイル中に呼び出されるそれぞれのコンパイラー・コンポーネント、そのバージョンおよびレベルを一覧表示します。複数の製品で共有されるコンポーネントは、複数のバージョン番号を表示する可能性があります。表示されるレベル番号は、ご使用のシステムにインストールされた更新 によって異なる場合があります。

関連情報

- 387 ページの『-qversion』

-qshowinc

カテゴリー

リスト、メッセージ、およびコンパイラー情報

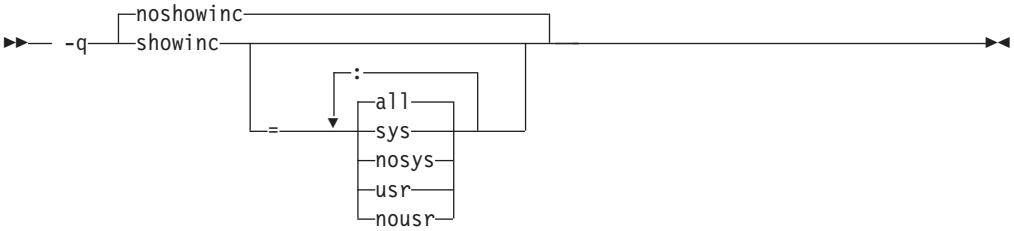
プラグマ同等物

#pragma options [no]showinc

目的

-qsource オプションと使用してリスト・ファイルを生成すると、リスト・ファイルのソース・セクションにユーザーまたはシステムのヘッダー・ファイルを選択的に示す。

構文



デフォルト

-qnoshowinc: ソース・ファイルに組み込まれたヘッダー・ファイルは、ソース・リストに表示されません。

パラメーター

all
ユーザー組み込みファイルとシステム組み込みファイルの両方をプログラム・ソース・リストに表示します。

sys
プログラム・ソース・リスト内のシステム組み込みファイル (`#include <filename>` プリプロセッサー・ディレクティブに組み込まれたファイル) を表示します。

usr
プログラム・ソース・リスト内のユーザー組み込みファイル (`#include "filename"` プリプロセッサー・ディレクティブまたは **-qinclude** と一緒に組み込まれたファイル) を表示します。

サブオプションなしで **showinc** を指定すると **-qshowinc=sys : usr** および **-qshowinc=all** と同等です。**noshowinc** を指定すると **-qshowinc=nosys : nousr** と同等です。

使用法

このオプションは、**-qlist** または **-qsource** コンパイラー・オプションが有効な場合にのみ効果を持ちます。

事前定義マクロ

なし。

例

すべての組み込みファイルがソース・リストに表示されるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qsource -qshowinc
```

関連情報

- 340 ページの『-qsource』

-qshowmacros

カテゴリー

81 ページの『出力制御』

プラグマ同等物

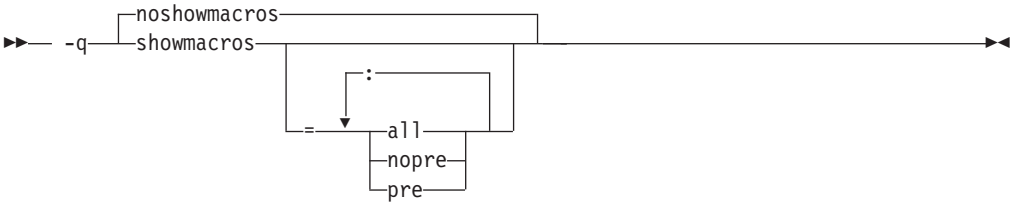
なし

目的

プリプロセスされた出力にマクロ定義を出す。

プリプロセスされた出力にマクロを出すと、コンパイラーで使用可能な機能を判別することができます。マクロ・リストは、複雑なマクロ展開をデバッグするときにも役立つ場合があります。

構文



デフォルト

-qnoshowmacros

パラメーター

all
プリプロセスされた出力にすべてのマクロ定義を出します。これは **-qshowmacros** を指定する場合と同じです。

pre | nopre
pre はプリプロセスされた出力に事前定義されたマクロ定義のみを出します。**nopre** では、これらの定義の付加が抑制されます。

使用法

このオプションを使用するときは、次の点に注意してください。

- このオプションは、**-E** または **-P** オプションなどを使用することによってプリプロセス出力が生成されていない場合は、有効にはなりません。
- マクロが定義され、後で、コンパイルの終了前に定義解除された場合、このマクロはプリプロセスされた出力に含まれません。
- プリプロセッサーによって内部的に定義されたマクロのみが事前定義と見なされ、他のすべてのマクロはユーザー定義と見なされます。

関連情報

- 150 ページの『-E』
- 288 ページの『-P』

-qshowpdf

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

コンパイル・ステップとリンク・ステップで **-qpdf1** および最小最適化レベル **-O2** と共に使用すると、アプリケーション内のすべてのプロシージャーについて、追加プロファイル情報を含む PDF マップ・ファイルを作成する。

構文

デフォルト

-qshowpdf

使用法

標準的なデータを使用してアプリケーションを実行すると、プロファイル情報が Profile-Directed Feedback (PDF) ファイル (デフォルトでは、ファイル名は `._pdf`) に記録されます。

PDF ファイルに加えて、コンパイラーは、**-qpdf1** フェーズで静的情報が入った PDF マップ・ファイルも生成します。これらの 2 つのファイルを使用することで、**showpdf** ユーティリティーを使用して、アプリケーションのプロファイル情報の一部をテキストまたは XML フォーマットで表示できます。 **showpdf** ユーティリティーについて詳しくは、「*XL C/C++ 最適化およびプログラミング・ガイド*」の『showpdf によるプロファイル情報の表示』を参照してください。

プロファイル情報を表示する必要がない場合は、**-qpdf1** フェーズで **-qnoshowpdf** オプションを指定して、PDF マップ・ファイルが生成されないようにします。これにより、コンパイル時間を短縮できます。

事前定義マクロ

なし。

関連情報

- 292 ページの『-qpdf1、-qpdf2』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『アプリケーションの最適化』

-qsimd

カテゴリー

最適化およびチューニング

プラグマ同等物

```
#pragma nosimd
```

目的

ベクトル命令をサポートするプロセッサーでコンパイラーがベクトル命令を自動的に利用できるかどうかを制御する。

これらの命令は、マルチメディア・アプリケーションなどの算法集中型のタスクと共に使用された場合、より高いハイパフォーマンスを提供することができます。

構文

デフォルト

- **-qsimd** を指定しないことは、**-qsimd=noauto** を指定することと同等です。
- **-qsimd** が指定されている場合、デフォルト・オプションは、**O2** 以下の最適化レベルでは **-qsimd=noauto** であり、**O3 -qhot** 以上の最適化レベルでは **-qsimd=auto** です。

使用法

-qsimd=auto オプションは、ベクトル命令をサポートするプロセッサーでベクトル命令の自動生成を行えるようにします。これは、非推奨となった **-qenablevmx** オプションに代わるものです。

-qsimd=auto オプションは、自動 SIMD 化を制御します。これは以前は非推奨の **-qhot=simd** オプションによって実行されていました。 **-qhot=simd** を指定すると、コンパイラーはそれを無視し、警告メッセージを発行しません。

-qsimd=auto が有効な場合、コンパイラーは、連続する配列エレメントのループ内で実行される特定の演算を、ベクトル命令に変換します。これらの命令は一度にいくつかの結果を計算するため、それぞれの結果を順次計算するよりも高速です。このオプションを適用すると、アプリケーションが、重要なイメージ処理要求を持つ場合に役立ちます。

-qsimd=noauto オプションは、ループ配列演算のベクトル命令への変換を使用不可にします。 **-qstrict=ieeefp**、**-qstrict=operationprecision**、および **-qstrict=vectorprecision** を使用すると、さらに精細な制御を行うことができます。詳しくは、『 352 ページの『-qstrict』』を参照してください。

注: ベクトル命令を使用して一度に複数の結果を計算すると、一部のアーキテクチャーで遅延が生じたり、さらには浮動小数点例外の検出ミスが生じたりすることがあります。例外の検出が重要である場合は、**-qsimd=auto** を使用しないでください。

規則

-qsimd オプションを使用すると、以下の規則が適用されます。

- 最適化レベル **-O3 -qhot** 以上を、POWER7 以上のプロセッサーに指定すると、**-qsimd=auto** が暗黙指定されます。 **-qsimd=auto** オプションを有効にするには、**-O3** または **-qhot** オプションの 1 つを有効にする必要があります。
- 非推奨の **-qenablevmx** オプションを指定すると、**-qsimd=auto** を指定した場合と同じ結果となります。このことに対して、コンパイラーは警告を出しません。
- サブオプションを何も指定しないで **-qsimd** を指定すると、**-qsimd=auto** を指定した場合と同じ結果となります。

- **-qsimd=auto** は、simd 命令セットをサポートするプラットフォーム上でのみ有効です。 **-qsimd=auto** は、POWER6 以上のプロセッサー、および simd サポートがある将来のプラットフォームにのみ指定できます。
- このオプションは、ベクトル命令をサポートするターゲット・アーキテクチャーに **-qarch** を設定した場合にのみ使用可能になります。
- IPA を使用可能にして、IPA のコンパイル・ステップで **-qsimd=auto** で指定したにもかかわらず、IPA リンク・ステップで **-qsimd=noauto** を指定した場合、コンパイラーは自動的に IPA リンク・ステップで **-qsimd=auto** を設定します。さらに、**-qarch** に対して、コンパイル時に指定されたアーキテクチャーに一致する適切な値も設定します。同様に、IPA を使用可能にして、IPA コンパイル・ステップでは **-qsimd=noauto** を指定し、IPA リンク・ステップでは **-qsimd=auto** を指定する場合、コンパイラーはコンパイル・ステップで **-qsimd=auto** を自動的に設定します。

事前定義マクロ

なし。

例

以下の例は、`-qsimd=auto` を特定の `for` ループに対して使用不可にする **#pragma nosimd** の使用法を示しています。

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}
```

関連情報

- 113 ページの『-qarch』
- 352 ページの『-qstrict』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『*プロシージャー間分析の使用*』。

-qskipsrc

カテゴリー

92 ページの『リスト、メッセージ、およびコンパイラー情報』

プラグマ同等物

なし。

目的

-qsource オプションを使用してリスト・ファイルを生成する場合、**-qskipsrc** を使用して、コンパイラーによってスキップされたソース・ステートメントをリスト・ファイルのソース・セクションに表示するかどうかを決定できる。あるいは、**-qskipsrc=hide** オプションを使用して、コンパイラーによってスキップされたソース・ステートメントを隠蔽します。

構文

デフォルト

- -qskipsrc=show

パラメーター

show | hide

show が有効である場合、コンパイラーはリスト内にすべてのソース・ステートメントを表示します。その結果、プリプロセス・ディレクティブの true と false の両方のパスが示される結果になります。

逆に、**hide** が有効である場合は、コンパイラーがスキップしたすべてのソース・ステートメントが省略されます。

使用法

一般に、**-qskipsrc** オプションはソース・セクションをリスト・ファイルに組み込むかどうかを制御せず、**-qsource** オプションが有効である場合にだけ、それを制御します。

すべてのソース・ステートメントをリスト内に表示するには (デフォルト・オプション)、以下のようにします。

```
xlc myprogram.c -qsource -qskipsrc=show
```

コンパイラーがスキップしたソース・ステートメントを省略するには、以下のようにします。

```
xlc myprogram.c -qsource -qskipsrc=hide
```

事前定義マクロ

なし。

関連情報

- 340 ページの『-qsource』
- 328 ページの『-qshowinc』
- 344 ページの『-qsrcmsg (C のみ)』

-qsmallstack

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

スタック・フレームのサイズを削減する。

構文

デフォルト

-qnosmallstack

使用法

スレッド化プログラムなど、スタックに大量のデータを割り振るプログラムでは、スタック・オーバーフローが発生する可能性があります。このオプションはオーバーフローを回避するためにスタック・フレームのサイズを縮小することができます。

このオプションは、IPA (**-qipa**、**-O4**、 **-O5** コンパイラー・オプション) とともに使用したときに有効です。

このオプションを指定すると、プログラムのパフォーマンスに逆の影響を与える可能性があります。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルしてスタック・フレームを小さくするには、以下のように入力します。

```
xlc myprogram.c -qipa -qsmallstack
```

関連情報

- 173 ページの『-g』
- 211 ページの『-qipa』
- 279 ページの『-O、-qoptimize』

-qsmp

カテゴリー

最適化およびチューニング

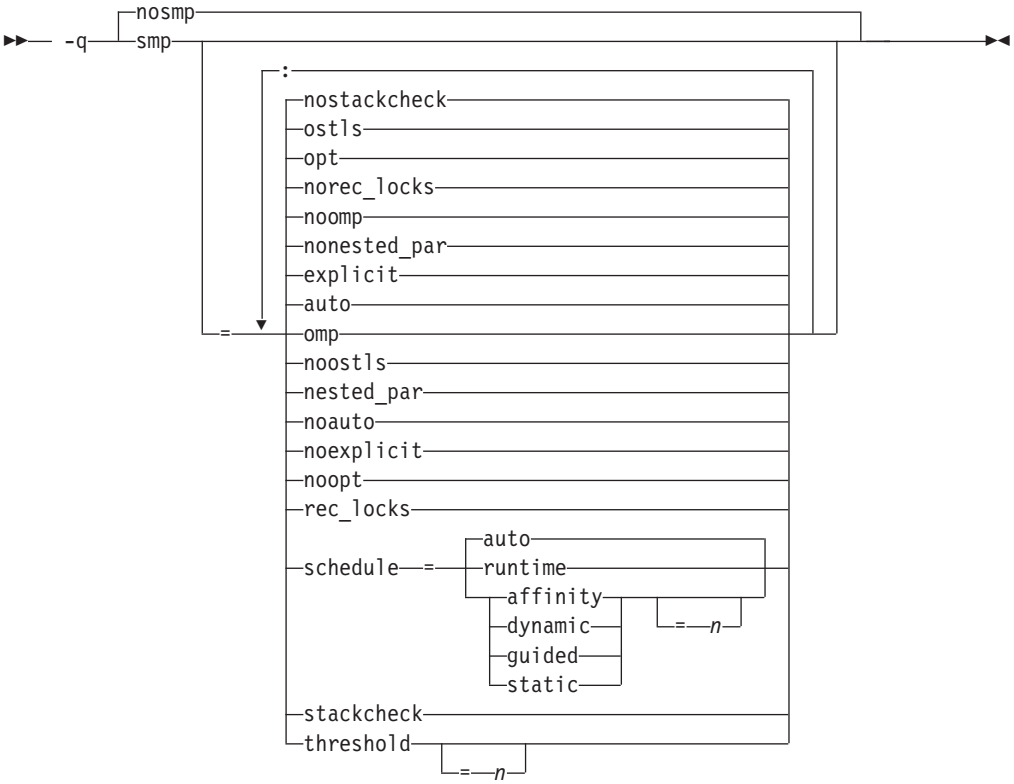
プラグマ同等物

なし。

目的

プログラム・コードの並列化を使用可能にする。

構文



デフォルト

-qnosmp。コードは、単一プロセッサー・マシンに作成されます。

パラメーター

auto | noauto
プログラム・コードの自動並列化および最適化を使用可能または使用不可にします。 **noauto** が有効な場合、OpenMP ディレクティブによって明示的に並列化されたプログラム・コードのみが最適化されます。**-qsmp=omp** または **-qsmp=noopt** を指定すると、**noauto** が暗黙指定されます。

explicit | noexplicit
ループの明示的並列化を制御するディレクティブを使用可能または使用不可にします。

nested_par | nonested_par
デフォルトで、コンパイラーはネストされた並列構造体を直列化します。 **nested_par** が有効なとき、コンパイラーは、所定のネスト並列構造体は並列化します。この場合の並列化の対象には、有効範囲単位内にあるネストされたループ構造体だけでなく、他の並列構造体から (直接であれ間接であれ) 参照されるサブプログラム内の並列構造体も含まれます。このサブオプションは、自動的に

並列化されるループには効果がないことに注意してください。この場合、多くても (有効範囲単位内にある) ループ・ネストに含まれる 1 ループしか並列化されません。

OMP_NESTED 環境変数の omp_set_nested 関数の設定は、**-qsmp = nested_par | nonested_par** オプションの設定をオーバーライドします。

このサブオプションは、注意して使用してください。外部ループ内の使用可能なスレッドの数および作業の量によっては、このオプションが有効な場合でも、内部ループが順次実行される可能性があります。並列化オーバーヘッドは、必ずしもプログラムのパフォーマンス向上と相殺されるとは限りません。

注: -qsmp=nested_par | nonested_par オプションは非推奨であり、将来のリリースで除去される可能性があります。代わりに、OMP_NESTED 環境変数または omp_set_nested 関数を使用してください。

omp | noomp
OpenMP 標準に対する厳格な準拠を強化または緩和します。**noomp** が有効なとき、**auto** が暗黙指定されます。 **omp** が有効なとき、**noauto** は暗黙指定され、OpenMP 並列化ディレクティブのみが認識されます。OpenMP API に準拠しない言語構成要素がコードに含まれている場合、コンパイラーは警告メッセージを発行します。

注: OpenMP 並列化を使用可能にするには、**-qsmp=omp** オプションを使用する必要があります。

opt | noopt
並列化プログラム・コードの最適化を使用可能または使用不可にします。**noopt** が有効なとき、コンパイラーは、コードの並列化に必要な最小の最適化量を実行します。これは、デフォルトで **-qsmp** が **-O2** および **-qhot** オプションを使用可能にし、いくつかの変数をレジスターに移動してデバッガーでアクセス不能にする結果になるので、デバッグに役立ちます。しかし、**-qsmp=noopt** および **-g** オプションを指定すると、これらの変数はまだデバッガーからは使用できます。

ostls| noostls
オペレーティング・システムが提供するスレッド・ローカル・ストレージ (TLS) を、**threadprivate** データに使用できるようにします。**noostls** サブオプションを使用して、非 TLS を **threadprivate** に使用可能にすることができます。**noostls** サブオプションは、以前のバージョンとの互換性を確保するために用意されています。

注: このサブオプションを使用したい場合は、オペレーティング・システムが、OpenMP **threadprivate** データをインプリメントするために TLS をサポートしている必要があります。オペレーティング・システムが TLS をサポートしていない場合は、**noostls** を使用して OS レベル TLS を使用不可にしてください。

rec_locks | norec_locks
再帰的ロックを使用するかどうかを判別します。**rec_locks** が有効なとき、ネストされたクリティカル・セクションはデッドロックを起こしません。 **rec_locks** サブオプションは、OpenMP API との整合性のない CRITICAL 構文の動作を指定します。

schedule
ソース・コードで他のスケジューリング・アルゴリズムが明示的に割り当てられていないループに使用されている、スケジューリング・アルゴリズムの種類、および (**auto** の場合を除き) チャンク・サイズ (*n*) を指定します。 **schedule** サブオプションのサブオプションは、次のとおりです。

affinity[=*n*]
最初にループの繰り返しは、**ceiling**(*number_of_iterations*/*number_of_threads*) 回の繰り返しを含む *n* 個の区画に分割されます。各区画は、最初にスレッドに割り当てられてから、それぞれ *n* 回の繰り返しを含むチャンクにさらに分割されます。*n* が指定されていないと、チャンクは **ceiling**(*number_of_iterations_left_in_partition* / 2) 回のループの繰り返しを含みます。

スレッドが解放されると、このスレッドに最初に割り当てられていた区画から次のチャンクが取得されます。その区画の中にチャンクがなくなると、スレッドは、別のスレッドに最初に割り当てられた区画から使用可能な次のチャンクを取得します。

最初にスリープ・スレッドに割り当てられている区画での作業は、アクティブなスレッドによって完了します。

類縁性スケジューリング・タイプは、OpenMP API 仕様の一部ではありません。

注: これは推奨されないサブオプションです。類似の機能として、**OMP_SCHEDULE** 環境変数に **dynamic** 節を指定して使用できます。

auto
ループの繰り返しのスケジューリングがコンパイラーとランタイム・システムに委任されます。コンパイラーとランタイム・システムは、実行可能なスレッドへの繰り返しのマッピングを任意に選択することができ (考えられるすべての有効なスケジュール・タイプを含みます)、別のループではそれらが異なっていることがあります。チャンク・サイズ (*n*) を指定しないでください。

dynamic[=*n*]
ループの繰り返しは、それぞれ *n* 回の繰り返しを含むチャンクに分割されます。 *n* が指定されない場合、各チャンクには 1 回の繰り返しが含まれます。

アクティブ・スレッドには、これらのチャンクが「先着順実行」の基準で割り当てられます。残りの作業のチャンクは、すべての作業の割り当てが完了するまで、使用可能なスレッドに割り当てられます。

guided[=*n*]
ループの繰り返しは、チャンクの最小サイズである *n* ループの繰り返しに達するまで、徐々により小さなチャンクに分割されます。 *n* が指定されなかった場合、*n* のデフォルト値は 1 回の繰り返しです。

アクティブ・スレッドには、チャンクが「先着順実行」の基準で割り当てられます。最初のチャンクには **ceiling**(*number_of_iterations*/*number_of_threads*) 繰り返しが含まれます。それ以降のチャンクは、**ceiling** (*number_of_iterations_left* / *number_of_threads*) 繰り返しを含みます。

runtime
チャンク入れのアルゴリズムを実行時に決定することを指定します。

static[=*n*]
ループの繰り返しは、それぞれ *n* 回の繰り返しを含むチャンクに分割されます。各スレッドには、「ラウンドロビン」方式でチャンクが割り当てられます。これを*ブロック巡回スケジューリング* と言います。 *n* の値が 1 である場合は、*n* の値が 1 のスケジューリング・タイプは、特に*巡回スケジューリング*と呼ばれます。

n を指定しない場合、チャンクには **floor**(*number_of_iterations*/*number_of_threads*) 回の繰り返しが含まれます。最初の **remainder**(*number_of_iterations*/*number_of_threads*) チャンクには複数の繰り返しがあります。スレッドにはそれぞれ別個のチャンクが割り当てられます。 これを*ブロック・スケジューリング* と言います。

スレッドは、スリープ状態で作業を割り当てられている場合、その作業を完了できるようスリープ状態から解除されます。

n 1 以上の整数の値でなければなりません。

サブオプションなしで **schedule** を指定すると、**schedule=auto** を指定した場合と同じになります。

stackcheck | nostackcheck
実行時にスレーブ・スレッドによってスタック・オーバーフローの有無を確認し、残りのスタック・サイズが XLSMPOPTS 環境変数の **stackcheck** オプションに指定されたバイト数に満たない場合は警告を出すようにコンパイラーに指示します。このサブオプションはデバッグの目的で設けられているもので、**XLSMPOPTS=stackcheck** も設定されている場合にのみ効果があります。詳しくは、30 ページの『XLSMPOPTS』を参照してください。

threshold[=*n*]
-qsmp=auto が有効なとき、行われる自動ループ並列化の程度を制御します。*n* の値は、並列化されるためにループで必要な最小の作業量を表します。現在、「work」の計算の大部分は、ループ内の繰り返し回数で占められています。通常は、*n* に高い値を指定すればするほど、並列化されるループの数は少なくなります。値 0 を指定すると、それが有益であろうとなかろうと、コンパイラーがすべての自動並列可能なループを並列化するよう指示します。値 100 を指定すると、コンパイラーに、有益であると思われる自動並列可能なループだけを並列化するよう指示します。100 より大きい値を指定すると、より多くのループがシリアライズされます。

n 0 以上の正整数になる必要があります。

サブオプションなしで **threshold** を指定すると、プログラムはデフォルト値の 100 を使用します。

サブオプションなしで **-qsmp** を指定すると、以下と等価になります。

```
-qsmp=auto:explicit:opt:noomp:norec_locks:nonested_par:schedule=auto:
nostackcheck:threshold=100:ostls
```

使用法

- **omp** サブオプションを指定すると、**noauto** が暗黙指定されます。**-qsmp=omp:auto** を指定して、OpenMP 準拠アプリケーションに自動並列化を同様に適用します。
- 自動的に、スレッド・セーフ・コンポーネントのすべてとリンクするには、**-qsmp** を **_r**-suffixed 呼び出しコマンドを指定して使用してください。**-qsmp** オプションを **_r** 以外のサフィックス付き呼び出しコマンドと一緒に使用することもできますが、適切なコンポーネントにリンクさせるのはユーザー側の責任で行います。**-qsmp** オプションを使用してプログラム内のソース・ファイルをコンパイルする場合、 **ld** コマンドを使用してリンクしない限り、リンク時に **-qsmp** も一緒に指定してください。
- **-qsmp=opt** オプションで生成されたオブジェクト・ファイルは、 **-qsmp=noopt** で生成されたオブジェクト・ファイルとリンクすることができます。各オブジェクト・ファイル内の変数のデバッガーにおける可視性は、リンクによって影響を受けることはありません。
- **-qnosmp** のデフォルト・オプション設定では、構文検査は実行されますが、並列化ディレクティブのためのコードが生成されないよう指定されています。**-qignprag=omp** を使用して、並列化ディレクティブを完全に無視します。
- **-qsmp** を指定すると、**-O2** が暗黙的に設定されます。 **-qsmp** オプションは **-qnooptimize** をオーバーライドしますが、 **-O3**、**-O4** または **05** はオーバーライドしません。並列化されたプログラム・コードをデバッグするときには、**-qsmp=noopt** を指定して、並列化されたプログラム・コードの最適化を使用不可にすることができます。
- **-qsmp=noopt** サブオプションは、コマンド行上のどこにあっても、パフォーマンス最適化オプションをオーバーライドします (**-qsmp** の前に **-qsmp=noopt** が現れる場合以外は)。例えば、**-qsmp=noopt -O3** は **-qsmp=noopt** と、**-qsmp=noopt -O3 -qsmp** は **-qsmp -O3** と等しいです。

関連情報

- 279 ページの『-O、-qoptimize』
- 371 ページの『-qthreaded』
- 30 ページの『並列処理のための環境変数』
- 459 ページの『並列処理のためのプラグマ・ディレクティブ』
- 645 ページの『並列処理のための組み込み関数』

-qsource

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

#pragma options [no]source

目的

リストのソース・セクションを含むコンパイラー・リスト・ファイルを作成し、エラー・メッセージの印刷時にソース情報を追加して提供する。

構文

デフォルト

-qnosource

使用法

-qsource または **#pragma オプション・ソース**が有効な場合、コマンド・ラインで指定された各ソース・ファイルに対して .lst サフィックスが付いたリスト・ファイルが生成されます。リスト・ファイルの内容について詳しくは、 22 ページの『コンパイラー・リスト』を参照してください。

#pragma options source および **#pragma options nosource** プリプロセッサー・ディレクティブのペアを、ソース・プログラム全体で使用することによって、ソースの一部を選択的に出力することができます。**#pragma options source** と **#pragma options nosource** の間のソースが出力されます。

-qnoprint オプションは、このオプションをオーバーライドします。

事前定義マクロ

なし。

例

myprogram.c ファイルには、次のコードが入っています。

```
#include <stdio.h>
int main()
{
  printf("Hello World");
}
```

myprogram.c ファイルをコンパイルしてソース・コードを含むコンパイラー・リストを作成するには、以下のように入力します。

```
xlc myprogram.c -qsource
```

myprogram.lst ファイルには、myprogram.c ファイルのコードが含まれるソース・セクションがあります。

```
>>>>> SOURCE SECTION <<<<<

1 | # include <stdio.h>
2 |
3 | int main ()
4 | {
5 |   printf("Hello World");
6 | }
```

関連情報

- 261 ページの『-qlist』
- 266 ページの『-qlistopt』

- 308 ページの『-qprint』

-qsourcetype

カテゴリー

入力制御

プラグマ同等物

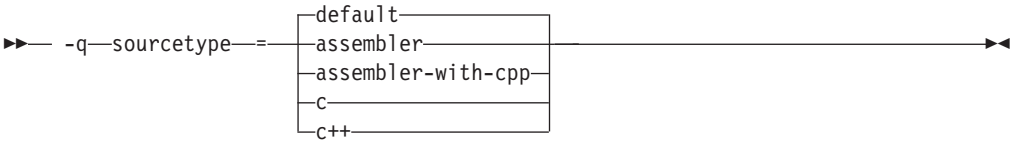
なし。

目的

実際のファイル名サフィックスに関係なく、すべての認識されるソース・ファイルを指定されたソース・タイプとして扱うようコンパイラーに命令する。

通常、コンパイラーはコマンド行に指定されたソース・ファイルのファイル名サフィックスを使用してソース・ファイルのタイプを判別します。例えば、.c のサフィックスは通常、C ソース・コードを暗黙指定し、.C のサフィックスは通常 C++ ソース・コードを暗黙指定します。**-qsourcetype** オプションは、ファイル名サフィックスに依存せずに、オプションによって指定されたソース・タイプを想定するようコンパイラーに命令します。

構文



デフォルト

-qsourcetype=default

パラメーター

assembler
このオプションの後のすべてのソース・ファイルは、アセンブラー言語のソース・ファイルであるかのようにコンパイルされます。

assembler-with-cpp
このオプションの後のすべてのソース・ファイルは、プリプロセッシングが必要なアセンブラー言語のソース・ファイルであるかのようにコンパイルされます。

c このオプションの後のすべてのソース・ファイルは、C 言語のソース・ファイルであるかのようにコンパイルされます。

C++ **c++**
このオプションの後のすべてのソース・ファイルは、C++ 言語のソース・ファイルであるかのようにコンパイルされます。このサブオプションは **-+** と同等です。

default
ソース・ファイルのプログラム言語はそのファイル名のサフィックスによって暗黙指定されます。

使用法

このオプションを使用しない場合、ファイルを C ファイルとしてコンパイルするためには .c のサフィックスが必要で、C++ ファイルとしてコンパイルするためには .C (大文字の C)、.cc、.cp、.cpp、.cxx、または .c++ のサフィックスが必要です。

このオプションはファイル・システムで大/小文字の区別があるかどうかに関係なく適用されます。つまり、 file.c および file.C が同じ物理ファイルを参照する、大/小文字の区別がないファイル・システムにおいてさえ、コンパイラーはなおコマンド行のファイル名引数の大/小文字の違いを認識して、それに応じてソース・タイプを判別します。

このオプションが影響を与えるのは、オプションよりも前ではなく、そのオプションに続くコマンド行で指定されるファイルのみであることに注意してください。そのため、次の例のようになります。

```
xlc goodbye.C -qsourcetype=c hello.C
```

hello.C は C ソース・ファイルとしてコンパイルされますが、goodbye.C は、C++ ファイルとしてコンパイルされます。

-qsourcetype オプションは、**-+** オプションと一緒に使用しないでください。

事前定義マクロ

なし。

例

ソース・ファイル hello.C を C 言語のソース・コードとして扱うには、以下のように入力します。

```
xlc -qsourcetype=c hello.C
```

関連情報

- 103 ページの『-+ (正符号) (C++ のみ)』

-qspill

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

#pragma options [no]spill

目的

レジスター・スピル・スペース (溢れたレジスターをストレージに退避させるために最適化プログラムが使用する内部プログラム・ストレージ域) のサイズをバイト単位で指定する。

構文

デフォルト

-qspill=512

パラメーター

size
レジスター割り振り予備域のバイト数を表す整数。

使用法

プログラムが非常に複雑な場合、あるいは計算が多過ぎてレジスター内に一度に保持できないためにプログラムに一時記憶域が必要な場合は、この領域の増加が必要となることがあります。コンパイラーが予備域を拡大するように要求するメッセージを出さない限り、この予備域は拡大しないでください。 矛盾がある場合は、指定された最大予備域が使用されます。

事前定義マクロ

なし。

例

`myprogram.c` のコンパイル時に警告メッセージを受け取ったものの、900 項目の予備域を指定してコンパイルする場合には、以下のように入力します。

```
xlc myprogram.c -qspill=900
```

-qsrcmsg (C のみ)

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

#pragma options [no]srcmsg

目的

対応するソース・コード行をコンパイラーが生成した診断メッセージに追加する。

nosrcmsg が有効な場合、エラー・メッセージには単にエラーが発生したファイル、行、および列が表示されます。**srcmsg** が有効な場合、コンパイラーは、診断メッセ

ージが参照するソース行またはソース行の一部を再構成し、診断メッセージの前に表示します。エラーがある列位置を指すポインターが表示される場合もあります。

構文

デフォルト

-qnosrcmsg

使用法

srcmsg が有効な場合、再構成されたソース行は、マクロ展開後に表示される行を表します。行は一部しか再構成されない場合もあります。 表示された行の先頭または末尾に文字 "...." がある場合は、ソース行の一部が表示されていないことを示します。

-qnosrcmsg を使用して解析可能な簡潔なメッセージを表示します。

事前定義マクロ

なし。

-qstackprotect

カテゴリー

87 ページの『オブジェクト・コード制御』

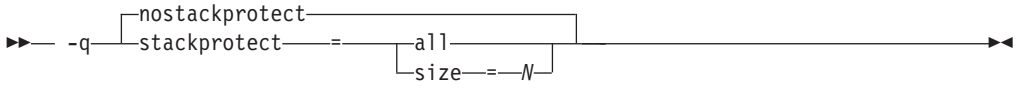
プラグマ同等物

なし。

目的

スタックを上書きするか破損する悪質なコードまたはプログラム・エラーに対する保護を提供する。

構文



デフォルト

- -qnostackprotect

パラメーター

all

all は、すべてのプロシージャーを、ぜい弱オブジェクトの有無にかかわらず保護します。このオプションは、デフォルトでは設定されません。

size=*N*

size=*N* を指定すると、サイズが *N* バイト以上の自動オブジェクトを含むすべてのプロシージャーが保護されます。**-qstackprotect** を使用可能にした場合のデフォルト・サイズは 8 です。

使用法

-qstackprotect は、ぜい弱オブジェクトを持つプロシージャーをスタックの破損から保護する追加コードを生成します。このオプションは、パフォーマンスの低下をもたらす可能性があるため、デフォルトでは使用不可に設定されます。

ぜい弱オブジェクトを持つすべてのプロシージャーを保護するコードを生成するには、以下のようにします。

```
xlc myprogram.c -qstackprotect=all
```

特定のバイト数のオブジェクトを持つプロシージャーを保護するコードを生成するには、以下のようにします。

```
xlc myprogram.c -qstackprotect=size=8
```

事前定義マクロ

なし。

関連情報

- 193 ページの『-qinfo』

-qstaticinline (C++ のみ)

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

インライン関数を静的または外部リンケージとして扱うかを制御する。

-qnostaticinline が有効な場合、コンパイラーはインライン関数を `extern` として扱います。異なるソース・ファイルにいくつ関数の定義が現れるかに関係なく、インライン関数指定子によってマークされた 1 つの関数に対して生成される関数本体は 1 つのみです。 **-qstaticinline** が有効な場合、コンパイラーはインライン関数を静的リンケージを持つものとして扱います。別個の関数本体が、インライン関数指定子によってマークされた同じ関数の異なるソース・ファイルで定義ごとに生成されます。

構文

デフォルト

-qnostaticinline

使用法

-qnostaticinline が有効な場合、本体が生成されない冗長な関数定義はすべてデフォルトで破棄されます。**-qkeepinlines** オプションを使用してこの振る舞いを変更することができます。

事前定義マクロ

なし。

例

-qstaticinline オプションを使用すると、以下の宣言における関数 f は、たとえ明示的に静的であると宣言していなくても、静的関数として扱われます。関数の定義ごとに別の関数本体が作成されます。これによりコード・サイズがかなり大きくなる可能性がある点に注意してください。

```
inline void f() {/*...*/};
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『インライン関数のリンケージ』
- 220 ページの『-qkeepinlines (C++ のみ)』

-qstaticlink

カテゴリー

リンク

プラグマ同等物

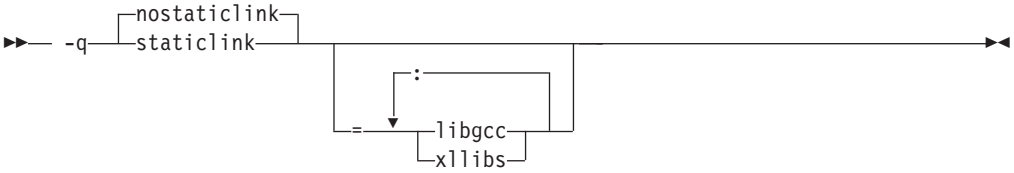
なし。

目的

静的または共有のランタイム・ライブラリーをアプリケーションにリンクするかどうかを制御する。

このオプションは、単独または組み合わせて使用される GNU オプション **-static**、**-static-libgcc**、および **-shared-libgcc** によって暗黙指定されるものと同等のリンク規則を指定する能力を提供します。

構文



デフォルト

-qnostaticlink

パラメーター

libgcc

- **libgcc** を **nostaticlink** と共に指定すると、コンパイラーは共有バージョンの **libgcc** にリンクします。
- **libgcc** を **staticlink** と共に指定すると、コンパイラーは静的バージョンの **libgcc** にリンクします。

このサブオプションは、GNU オプションの **-static-libgcc** および **-shared-libgcc** によって有効にされるものと同等の機能を提供します。

xllibs

- **xllibs** を **nostaticlink** と共に指定すると、コンパイラーは共有バージョンの XL コンパイラー・ライブラリーにリンクします。
- **xllibs** を **staticlink** と共に指定すると、コンパイラーは XL コンパイラー・ライブラリーの静的バージョンにリンクします。

使用法

-qstaticlink をサブオプションなしで指定すると、静的ライブラリーのみがオブジェクト・ファイルとリンクします。

-qnostaticlink をサブオプションなしで指定すると、共有ライブラリーがオブジェクト・ファイルとリンクします。

競合するコンパイラー・オプションは、以下のように解決されます。

- 最初に **-qnostaticlink** をサブオプションなしで指定し、次に **-qstaticlink** をサブオプションありまたはなしで指定すると、**-qnostaticlink** はオーバーライドされます。例えば、**-qnostaticlink -qstaticlink=xllibs** は **-qstaticlink=xllibs** と同等です。
- **-qstaticlink** をサブオプションありまたはなしで指定し、それに続いて **-qnostaticlink** をサブオプションなしで指定すると、**-qnostaticlink** が有効になり、共有ライブラリーがリンクします。または、**-qstaticlink** をサブオプションなしで指定すると、**-qstaticlink** が有効になり、静的ライブラリーのみがオブジェクト・ファイルとリンクします。以下の例を参照してください。

表 31. 競合するコンパイラー・オプションおよび解決の例

オプションの組み合わせ	コンパイラーの動作
-qstaticlink=libgcc -qnostaticlink	共有ライブラリーはリンクされます。

表 31. 競合するコンパイラー・オプションおよび解決の例 (続き)

オプションの組み合わせ	コンパイラーの動作
-qstaticlink -qnostaticlink=libgcc	すべてのライブラリーは静的にリンクされます。コンパイラーは、以下の警告メッセージを出します。 (W) The options -qnostaticlink=libgcc and -qstaticlink are incompatible ((警告) オプション -qnostaticlink=libgcc および -qstaticlink は非互換です). Option -qnostaticlink=libgcc is ignored (オプション -qnostaticlink=libgcc は無視されます).
-qstaticlink -qnostaticlink=libgcc:xllibs	すべてのライブラリーは静的にリンクされます。コンパイラーは、以下の警告メッセージを出します。 (W) The options -qnostaticlink=libgcc and -qstaticlink are incompatible ((警告) オプション -qnostaticlink=libgcc および -qstaticlink は非互換です). オプション -qnostaticlink=libgcc は無視されます。 (W) The options -qnostaticlink=xllibs and -qstaticlink are incompatible ((警告) オプション -qnostaticlink=xllibs および -qstaticlink は非互換です). オプション -qnostaticlink=xllibs は無視されます。
-qstaticlink -qstaticlink=libgcc	すべてのライブラリーは静的にリンクされます。
-qnostaticlink=libgcc -qstaticlink	すべてのライブラリーは静的にリンクされます。

注:

- メッセージ・カタログがシステムにインストールされていないのにランタイム・ライブラリーが静的にリンクすると、メッセージ番号のみのメッセージが出され、メッセージ・テキストは表示されません。
- 共有ライブラリーまたは動的にリンクされたアプリケーションが例外をスローまたははキャッチすると想定されている場合、それを有効な **-qnostaticlink=libgcc** オプションを指定した共有 **libgcc** (デフォルト) を使用してリンクする必要があります。

重要: サード・パーティーのライブラリーまたは製品の使用はいずれも、個々のライセンスの規定に従います。**-qstaticlink** オプションを使用すると、コンパイルしたプログラムに重大な法律上の影響をもたらす可能性があります。このオプションを使用する前に、法律上のアドバイスを求めることを強く推奨します。

以下の表は、共有ライブラリーおよび非共有ライブラリーのリンケージを指定する同等の GNU オプションおよび XL C/C++ オプションを示しています。

表 32. オプション・マッピング: GNU リンカーの制御

GNU オプション	意味	XL C/C++ オプション
-shared	共有オブジェクトをビルドします。	-qmkshrobj 1
-static	静的オブジェクトをビルドして、共有ライブラリーとのリンクを回避します。リンクされるすべてのライブラリーは、静的ライブラリーでなければなりません。	-qstaticlink
-shared-libgcc	libgcc の共有バージョンとのリンク。	-qnostaticlink=libgcc 2

表 32. オプション・マッピング: GNU リンカーの制御 (続き)

GNU オプション	意味	XL C/C++ オプション
-static-libgcc	libgcc の静的バージョンとのリンク。この場合でも共有ライブラリーをリンクできます。	-qstaticlink=libgcc

注:

1 オプション **-qmkshrobj** とオプション **-qstaticlink** は両立しないため、一緒に指定することはできません。

2 これは、SUSE Linux Enterprise Server (SLES) および Red Hat Enterprise Linux (RHEL) でのデフォルト設定です。

事前定義マクロ

なし。

関連情報

- 277 ページの『-qmkshrobj』

-qstatsym

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

永続的なストレージ・クラスを持つ、ユーザー定義の非外部名 (初期化される静的変数または初期化されない静的変数など) をオブジェクト・ファイルのシンボル・テーブルに追加する。

構文

デフォルト

-qnostatsym

使用法

-qnostatsym が指定されている場合、静的変数はシンボル・テーブルに追加されません。ただし、静的関数はシンボル・テーブルに追加されます。

事前定義マクロ

なし。

-qstdinc

カテゴリー

入力制御

プラグマ同等物

#pragma options [no]stdinc

目的

標準組み込みディレクトリーがシステムおよびユーザー・ヘッダー・ファイルの検索パスに組み込まれているかどうかを指定する。

-qstdinc が有効な場合、コンパイラーは以下のディレクトリーでヘッダー・ファイルを検索します。

- C XL C ヘッダー・ファイルの構成ファイルで指定されたディレクトリー (通常は /opt/ibm/xlC/13.1.0/include/)、または **-qc_stdinc** オプションで指定されたディレクトリー
- C++ XL C および C++ ヘッダー・ファイルの構成ファイルで指定されたディレクトリー (通常は /opt/ibm/xlC/13.1.0/include/)、または **-qcpp_stdinc** オプションで指定されたディレクトリー
- システム・ヘッダー・ファイルの構成ファイルで指定されたディレクトリー または **-qgcc_c_stdinc** および **-qgcc_cpp_stdinc** オプションによって指定されたディレクトリー。

-qnostdinc が有効な場合、これらのディレクトリーは検索パスから除外されます。検索されるディレクトリーは以下のとおりです。

- `#include "filename"` ディレクティブを含むソース・ファイルが配置されるディレクトリー
- **-I** オプションによって指定されるディレクトリー
- **-qinclude** オプションによって指定されるディレクトリー

構文

デフォルト

-qstdinc

使用法

ヘッダー・ファイルの検索順序は 15 ページの『組み込みファイルのディレクトリー検索シーケンス』の説明を参照してください。

このオプションが影響を与えるのは相対名と一緒に組み込まれたヘッダー・ファイルの検索パスのみです。フル (絶対) パス名を指定した場合は、このオプションはそのパス名に影響を与えません。

最後の有効なプラグマ・ディレクティブは、後続のプラグマで置き換えられるまで有効のままです。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、(`myprogram.c` を含むディレクトリー以外に) ディレクトリー `/tmp/myfiles` のみで `#include "myinc.h"` ディレクティブと一緒に組み込まれたファイルが検索されるようにするには、次のように入力します。

```
xlc myprogram.c -qnostdinc -I/tmp/myfiles
```

関連情報

- 139 ページの『-qc_stdinc (C のみ)』
- 140 ページの『-qcpp_stdinc (C++ のみ)』
- 176 ページの『-qgcc_c_stdinc (C のみ)』
- 177 ページの『-qgcc_cpp_stdinc (C++ のみ)』
- 186 ページの『-I』
- 15 ページの『組み込みファイルのディレクトリー検索シーケンス』

-qstrict

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options [no]strict

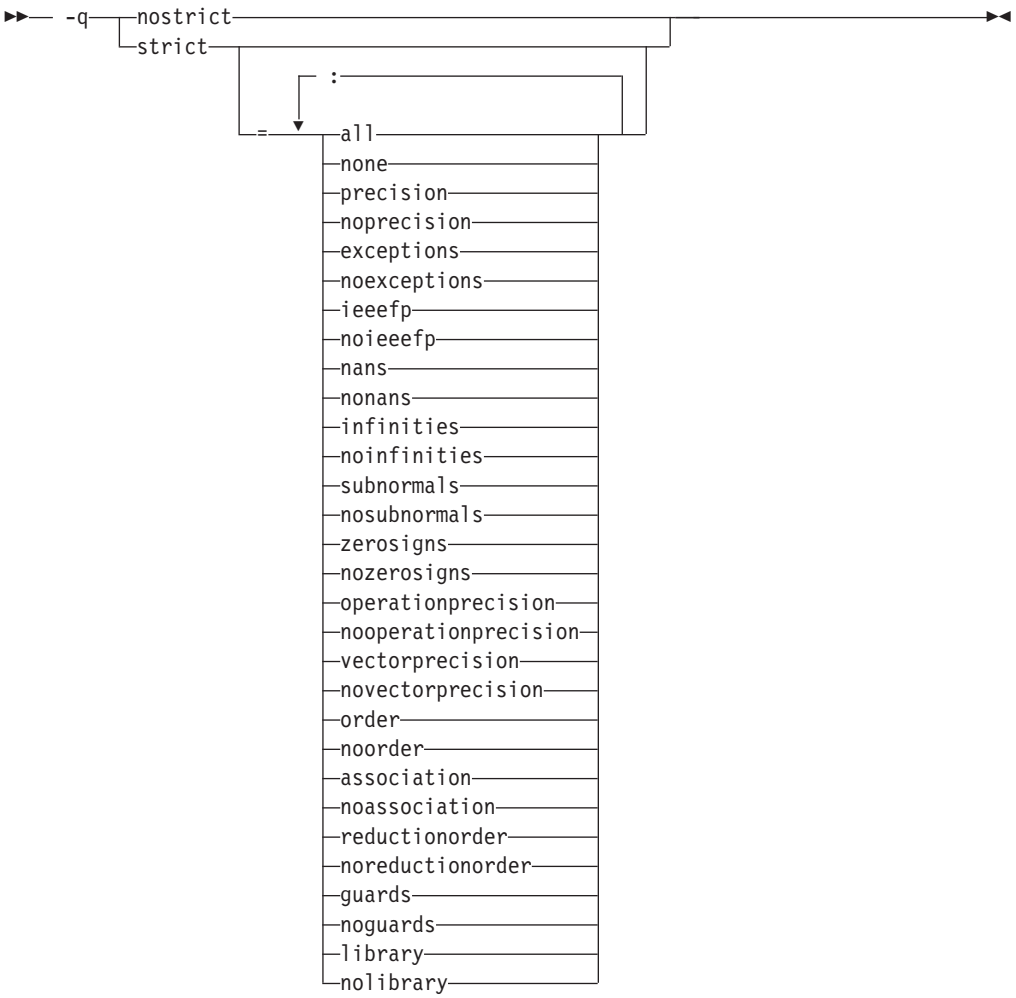
#pragma option_override (*function_name*, "opt (*suboption_list*)")

目的

デフォルトの最適化レベル **-O3** 以上、およびオプションの **-O2** で実行された最適化が、プログラムのセマンティクスを変更しないことを確認する。

このオプションは、最適化されたプログラムのプログラム実行における変更により、最適化されていないプログラムとは異なる結果が生成される状態の場合に使用します。

構文



デフォルト

- **-qnoopt** または **-O0** 最適化レベルが有効な場合は、**-qstrict** または **-qstrict=all** がデフォルトです。
- **-O2** または **-O** 最適化レベルが有効な場合は、**-qstrict** または **-qstrict=all** がデフォルトです。
- **-O3** 以上の最適化レベルが有効な場合は、**-qnostrict** または **-qstrict=none** がデフォルトです。

パラメーター

-qstrict サブオプションには以下が含まれます。

all | none
all は、**ieeefp**、**order**、**library**、**precision**、および **exceptions** サブオプションによって制御されるものを含め、すべてのセマンティクス変更トランスフォーメーションを無効にします。**none** では、これらのトランスフォーメーションが有効になります。

precision | noprecision
precision は、**subnormals**、**operationprecision**、**vectorprecision**、**association**、**reductionorder**、および **library** サブオプションによって制御されるものを含

め、浮動小数点の精度に影響を与える可能性のあるすべてのトランスフォーメーションを無効にします。**noprecision** では、これらのトランスフォーメーションが有効になります。

exceptions | noexceptions
exceptions は、**nans**、**infinities**、**subnormals**、**guards**、および **library** サブオプションによって制御されるものを含め、例外に影響を与える、または例外の影響を受ける可能性のあるすべてのトランスフォーメーションを無効にします。**noexceptions** では、これらのトランスフォーメーションが有効になります。

ieeefp | noieeefp
ieeefp は、**nans**、**infinities**、**subnormals**、**zerosigns**、**vectorprecision**、および **operationprecision** サブオプションによって制御されるものを含め、IEEE 浮動小数点への準拠に影響を与えるトランスフォーメーションを無効にします。**noieeefp** では、これらのトランスフォーメーションが有効になります。

nans | nonans
nans は、IEEE 浮動小数点 NaN (非数字) 値が存在する場合に誤った結果を生成する可能性があるか、またはこの値を不正確に生成する可能性があるトランスフォーメーションを無効にします。**nonans** では、これらのトランスフォーメーションが有効になります。

infinities | noinfinities
infinities は、浮動小数点の無限大が存在する場合に誤った結果を生成する可能性があるか、または浮動小数点の無限大を不正確に生成する可能性のあるトランスフォーメーションを無効にします。**noinfinities** では、これらのトランスフォーメーションが有効になります。

subnormals | nosubnormals
subnormals は、IEEE 浮動小数点のサブノーマル (非正規化数) (以前は「デノーマル」と呼ばれていました) が存在する場合に誤った結果を生成する可能性があるか、またはサブノーマルを不正確に生成する可能性があるトランスフォーメーションを無効にします。**nosubnormals** では、これらのトランスフォーメーションが有効になります。

zerosigns | nozerosigns
zerosigns は、浮動小数点ゼロの符号が正しいかどうかに影響を与える、またはその影響を受ける可能性のあるトランスフォーメーションを無効にします。**nozerosigns** では、これらのトランスフォーメーションが有効になります。

operationprecision | nooperationprecision
operationprecision は、各浮動小数点演算の概算結果を生成するトランスフォーメーションを無効にします。 **nooperationprecision** では、これらのトランスフォーメーションが有効になります。

vectorprecision | novectorprecision
vectorprecision は、ベクトル化した繰り返しによって、ベクトル化しない残りの繰り返しと異なる結果が生成される可能性がある場合、ループでのベクトル化を使用不可にします。**vectorprecision** を使用すると、同一のデータに対する同一の浮動小数点演算のすべてループの繰り返しで、同一の結果が生成されます。

novectorprecision は、さまざまな繰り返しによって、同じ入力から異なる結果が生成される可能性があっても、ベクトル化を使用可能にします。

order | noorder
order は、**association**、**reductionorder**、および **guards** サブオプションによって制御されるものを含め、結果または例外に影響を与える可能性のある、複数の演算間のすべてのコードの再配列を無効にします。**noorder** では、コードの再配列が有効になります。

association | noassociation
association は、1 つの式の中の再配列操作を無効にします。**noassociation** では、再配列操作が有効になります。

reductionorder | noreductionorder
reductionorder は、浮動小数点の縮約の並列化を無効にします。
noreductionorder では、それらの縮約の並列化を有効にします。

guards | noguards
guards は、操作を実行すべきかどうかを制御する保護領域の境界を越えた (すなわち、**if** を過ぎた、またはループ外の、あるいはプログラムを終了させる可能性があるか、例外をスローする可能性がある関数 呼び出しを過ぎた) 移動操作を無効にします。**noguards** は、保護領域の境界を越えた移動操作を有効にします。

library | nolibrary
library は、浮動小数点ライブラリー関数に影響を与えるトランスフォーメーション (例えば、浮動小数点ライブラリー関数を他のライブラリー関数または定数と置き換えるトランスフォーメーション) を無効にします。 **nolibrary** では、これらのトランスフォーメーションが有効になります。

使用法

all、**precision**、**exceptions**、**ieeefp**、および **order** サブオプションとその負の形式は、複数の個別のサブオプションに影響を与えるグループ・サブオプションです。多くの状態において、グループ・サブオプションはトランスフォーメーションに対して十分かつ粒度の細かい制御を行います。グループ・サブオプションは、そのグループの正の形式または形式なしのすべてのサブオプションが指定されている場合と同様に機能します。必要な場合は、1 つのグループ内の個々のサブオプション (例えば、**precision** グループ内の **subnormals** または **operationprecision**) が、そのグループ内の特定のトランスフォーメーションの制御を行います。

-qnostrict または **-qstrict=none** が有効な場合、以下の最適化がオンになります。
- 例外の原因となる可能性のあるコードは、再配置される場合があります。実行時の別の時点でそれに対応する例外が起こる可能性がありますが、まったく起こらない場合もあります。 (コンパイラーは引き続きその状態を最小限にくい止めるように試みます。)
- 浮動小数点演算では、ゼロ値の符号が保存されない場合があります。(ゼロ値の符号が確実に保存されるようにするには、**-qfloat=rrm**、**-qfloat=nomaf**、または **-qfloat=strictnmaf** も指定してください。)
- 浮動小数点式の関連付けがやり直される場合があります。例えば、**(2.0*3.1)*4.2** は **2.0*(3.1*4.2)** になる可能性があります。これは、結果が同一にならない場合でも、その方が速ければ実施されます。
- **-qfloat** オプションの **fltint** および **rsqrt** サブオプションがオンになります。これを再びオフにするには、**-qstrict** オプションか、または **-qfloat** の **nofltint** および

norsqrt サブオプションも使用します。低レベルの最適化を指定するか、最適化をまったく指定しない場合、これらのサブオプションはデフォルトでオフになります。

-qstrict[=suboptions] または **-qnostrict** の組み合わせのさまざまなサブオプションを指定することにより、以下のサブオプションが設定されます。

- **-qstrict** または **-qstrict=all** では **-qfloat=norsqrt:rngchk** が設定されます。**-qnostrict** または **-qstrict=none** では **-qfloat=rsqrt:norngchk** が設定されます。
- **-qstrict=operationprecision** または **-qstrict=exceptions** では **-qfloat=nofltint** が設定されます。 **-qstrict=nooperationprecision** と **-qstrict=noexceptions** の両方を指定すると、**-qfloat=fltint** が設定されます。
- **-qstrict=infinities**、**-qstrict=operationprecision**、または **-qstrict=exceptions** では **-qfloat=norsqrt** が設定されます。
- **-qstrict=noinfinities:nooperationprecision:noexceptions** では **-qfloat=rsqrt** が設定されます。
- **-qstrict=nans**、**-qstrict=infinities**、**-qstrict=zerosigns**、または **-qstrict=exceptions** では **-qfloat=rngchk** が設定されます。すべての **-qstrict=nonans:nozerosigns:noexceptions** または **-qstrict=noinfinities:nozerosigns:noexceptions** を指定するか、あるいはこれらすべてを暗黙指定する任意のグループ・サブオプションを指定すると、**-qfloat=norngchk** が設定されます。

注: -qstrict サブオプションとそれらに対応する **-qfloat** の関係について詳しくは、160 ページの『-qfloat』を参照してください。

これらのいずれかの設定をオーバーライドするには、コマンド行上の **-qstrict** オプションの後に適切な **-qfloat** サブオプションを指定します。

事前定義マクロ

なし。

例

-O3 の積極的な最適化がオフにされ、平方根の結果による除算が逆数による乗算によって置換される (**-qfloat=rsqrt**) ように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -O3 -qstrict -qfloat=rsqrt
```

精度に影響を与えるもの以外のすべてのトランスフォーメーションを有効にするには、以下のように指定します。

```
xlc myprogram.c -qstrict=none:precision
```

NaNs および無限大に関係するものを除くすべてのトランスフォーメーションを無効にするには、以下のように指定します。

```
xlc myprogram.c -qstrict=all:nonans:noinfinities
```

関連情報
- 331 ページの『-qsimd』
- 160 ページの『-qfloat』
- 183 ページの『-qhot』
- 279 ページの『-O、-qoptimize』

-qstrict_induction
カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

コンパイラーが帰納 (ループ・カウンター) 変数の最適化を実行しないようにする。これらの最適化は、帰納変数を含んだ整数オーバーフロー操作がある場合は安全ではない (ご使用のプログラムのセマンティクスを変更する) 可能性があります。

構文

デフォルト
- **-qstrict_induction**
- **-O2** 以上の最適化レベルが有効の場合は **-qnostrict_induction** です。

使用法

-O2 以上の最適化を使用する場合に、ループ帰納変数の切り捨てや符号の拡張子が、結果として変数のオーバーフロー、または循環を起こす場合、プログラムの結果が最適化で変更されるのを阻止する **-qstrict_induction** を指定することができます。ただし、**-qstrict_induction** の使用は、大きな性能低下の原因となることがあるため、一般に推奨されません。

事前定義マクロ

なし。

関連情報
- 279 ページの『-O、-qoptimize』

-qsuppress
カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

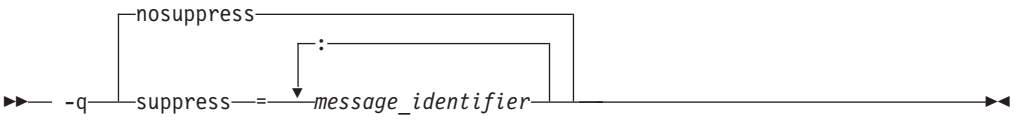
448 ページの『#pragma report (C++ のみ)』

目的

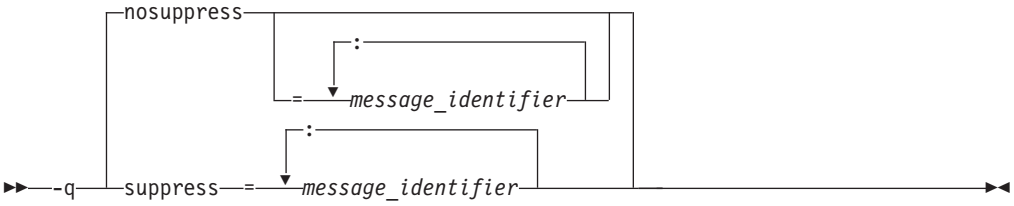
特定の通知メッセージ、または警告メッセージが生成された場合、表示されたりリスト・ファイルに追加されるのを防ぐ。

構文

-qsuppress 構文 — C



-qsuppress 構文 — C++



デフォルト

-qnosuppress: **-qflag** オプションが指定されない限りは、すべての通知メッセージおよび警告メッセージが報告されます。

パラメーター

message_identifier
メッセージ ID です。メッセージ ID は以下のフォーマットでなければなりません。

15*dd-number*

ここで、

15 コンパイラーの製品 ID です。

dd このメッセージを作成するコンパイラー・コンポーネントを示す 2 桁のコードです。これらのコードの説明については、20 ページの『コンパイラー・メッセージ・フォーマット』を参照してください。

number
メッセージ番号です。

使用法

抑制できるのは、通知 (I) メッセージおよび警告 (W) メッセージのみです。 (S) および (U) レベルのメッセージなど、他のタイプのメッセージは抑制できません。

重大エラーに追加情報を提供する通知メッセージと警告メッセージは、このオプションでは使用不可にならないことに注意してください。

すべての通知メッセージおよび警告メッセージを抑制するには、**-w** オプションを使用することができます。

IPA メッセージを抑制するには、コマンド行で **-qipa** の前に **-qsuppress** を入力してください。

-qhaltonmsg コンパイラー・オプションは、**-qsuppress** よりも優先されます。**-qhaltonmsg** と **-qsuppress** の両方が指定された場合は、**-qsuppress** によって抑止されるメッセージも表示されます。

C **-qnosuppress** コンパイラー・オプションは、**-qsuppress** の直前の設定を取り消します。C

C++ 特定のメッセージ ID と共に **-qnosuppress** を指定した場合は、同じメッセージ ID を持つそれより前の **-qsuppress** インスタンスが無効になります。特定のメッセージ ID を使用せずに **-qnosuppress** を指定した場合は、それより前のすべての **-qsuppress** インスタンスが無効になります。

以下のオプションのうち 2 つまたは 3 つを指定した場合は、最後に指定されたオプションが優先されます。

-qsuppress=*message_identifier*

-qnosuppress=*message_identifier*

-qnosuppress

C++

事前定義マクロ

なし。

例

プログラムが、通常、結果として以下を出力する場合、

```
"myprogram.c", line 1.1:1506-224 (I) Incorrect #pragma ignored
```

以下のように指定してコンパイルすることにより、このメッセージを抑制することができます。

```
xlc myprogram.c -qsuppress=1506-224
```

関連情報

- 158 ページの『-qflag』
- 181 ページの『-qhaltonmsg』

-qsymtab (C のみ)

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

シンボル・テーブルに表示する情報を決定する。

構文

デフォルト

静的変数および参照されていない typedef、構造体、共用体、および列挙型の宣言は、オブジェクト・ファイルのシンボル・テーブルに組み込まれていません。

パラメーター

unref
-g オプションと一緒に使用すると、オブジェクト・ファイルのシンボル・テーブルにある参照されていない typedef 宣言、構造体、共用体、および enum 型定義にデバッグ情報を組み込むように指定します。このサブオプションは **-qdbxextra** と同等です。

-qsymtab=unref を使用すると、オブジェクトおよび実行可能ファイルのサイズが大きくなる可能性があります。

static
永続的なストレージ・クラスを持つ、ユーザー定義の非外部名 (初期化される静的変数または初期化されない静的変数など) をオブジェクト・ファイルのシンボル・テーブルに追加する。このサブオプションは **-qstatsym** と同等です。

事前定義マクロ

なし。

例

静的シンボルがシンボル・テーブルに追加されるように myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qsymtab=static
```

myprogram.c をコンパイルして、参照されていない typedef、構造体、共用体、および列挙型の宣言をシンボル・テーブルに組み込んでデバッガーで使用できるようにするには、以下のように入力します。

```
xlc myprogram.c -g -qsymtab=unref
```

関連情報

- 173 ページの『-g』
- 145 ページの『-qdbxextra (C のみ)』
- 350 ページの『-qstatsym』

-qsyntaxonly (C のみ)

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

オブジェクト・ファイルを生成せずに構文検査を実行する。

構文

デフォルト

デフォルトでは、ソース・ファイルをコンパイルしてリンクすると、実行可能ファイルが生成されます。

使用法

-P、**-E**、および **-C** オプションは、 **-qsyntaxonly** オプションをオーバーライドします。これによって、 **-c** および **-o** オプションもオーバーライドされます。

-qsyntaxonly オプションは、オブジェクト・ファイルの生成しか抑制しません。リスト・ファイルなど、他のすべてのファイルは、それらに対応するオプションが設定されている限り作成されます。

事前定義マクロ

なし。

例

オブジェクト・ファイルを生成せずに `myprogram.c` の構文を検査するには、以下のように入力します。

```
xlc myprogram.c -qsyntaxonly
```

関連情報

- 124 ページの『-C、-C!』
- 123 ページの『-c』
- 150 ページの『-E』
- 278 ページの『-o』
- 288 ページの『-P』

-t

カテゴリー

コンパイラーのカスタマイズ

プラグマ同等物

なし。

目的

-B オプションで指定したプレフィックスを、指定したコンポーネントに適用する。

構文

デフォルト

すべてのコンパイラー・コンポーネントのデフォルト・パスは、コンパイラー構成ファイルで定義されます。

パラメーター

以下の表は、**-t** パラメーターとコンポーネント名との対応を示しています。

パラメーター	説明	コンポーネント名
a	アセンブラー	as
b	低水準最適化プログラム	xlCcode
c	コンパイラーのフロントエンド	xlcentry、xlCentry
C++ C	C++ コンパイラーのフロントエンド	xlCentry
d	逆アセンブラー	dis
I (大文字の i)	高水準最適化プログラム、コンパイル・ステップ	ipa
L	高水準最適化プログラム、リンク・ステップ	ipa
l (小文字の L)	リンカー	ld

使用法

このオプションは、**-B***prefix* オプションと一緒に使用します。 **-B** が *prefix* なしで指定される場合、デフォルト・プレフィックスは /lib/o です。 **-B** がまったく指定されないと、標準プログラム名のプレフィックスは /lib/n です。

注: p サブオプションを使用すると、ソース・コードがコンパイル前に別々にプリプロセスされるために、プログラムのコンパイル方法が変わってしまう可能性があることに注意してください。

事前定義マクロ

なし。

例

名前 `/u/newones/compilers/` が、コンパイラーおよびアセンブラー・プログラム名の接頭部として付くように `myprogram.c` をコンパイルするには、以下を入力してください。

```
xlc myprogram.c -B/u/newones/compilers/ -tca
```

関連情報

- 121 ページの『-B』

-qtabsize

カテゴリー

言語エレメント制御

プラグマ同等物

#pragma options tabsize

目的

エラー・メッセージで列番号を報告するために、デフォルトのタブの長さを設定する。

構文

デフォルト

-qtabsize=8

パラメーター

number
ソース・プログラム内のタブを表す文字スペースの数。

使用法

このオプションは、エラーが発生した列番号を示すエラー・メッセージのみに影響します。

事前定義マクロ

なし。

例

コンパイラーがタブの幅を 1 文字であると見なすように myprogram.c をコンパイルするには、次のように入力します。

```
xlc myprogram.c -qtabsize=1
```

この場合、ユーザーは、1 文字位置 (ここでは、タブの長さに関係なく文字とタブはそれぞれ 1 つの文字位置に等しい) が、 1 文字分の列と同等であると考えることができます。

-qtbtable

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma options tbtable

目的

オブジェクト・ファイルに組み込まれるデバッグ・トレースバック情報の量を制御する。

多くのパフォーマンス測定ツールでは、最適化されたコードを適切に分析するために完全なトレースバック・テーブルが必要です。トレースバック・テーブルは、生成されると、オブジェクト・コードの最後にあるテキスト・セグメントに配置され、関数の型、スタック・フレーム、およびレジスター情報など、各関数についての情報を含みます。

構文

デフォルト

- **-qtbtable=full**
- **-O** 以上の最適化が有効の場合は **-qtbtable=small** です。

パラメーター

full
名前およびパラメーターの情報が入った完全なトレースバック・テーブルを生成します。

none
トレースバック・テーブルを生成しません。スタック・フレームをアンワインドすることはできないので、例外処理は使用不可となります。

small
生成されるトレースバック・テーブルには名前やパラメーターの情報は入りませんが、トレースバックとして完全に機能します。 このサブオプションは、プログラム・コードのサイズを削減します。

使用法

このオプションは 64 ビット・コンパイルにのみ適用され、 32 ビット・コンパイルで指定された場合には無視されます。

#pragma options ディレクティブは、コンパイル単位の最初のステートメントより前に指定しなければなりません。

事前定義マクロ

なし。

関連情報

- 173 ページの『-g』

-qtempinc (C++ のみ)

カテゴリー

テンプレート制御

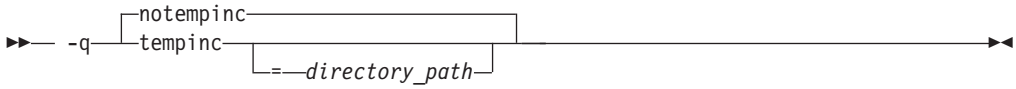
プラグマ同等物

なし。

目的

テンプレート関数およびクラス宣言用に別個のテンプレート組み込みファイルを生成し、オプションで指定できるディレクトリーにこれらのファイルを入れます。

構文



デフォルト

-qnotempinc

パラメーター

directory_path
生成されたテンプレートの組み込みファイルが置かれるディレクトリー。

使用法

-qtempinc コンパイラー・オプションと **-qtemplateregistry** コンパイラー・オプションは、相互に排他的です。**-qtempinc** を指定すると、 **-qnotemplateregistry** が暗黙指定されます。同様に、**-qtemplateregistry** を指定すると、**-qnotempinc** が暗黙指定されます。ただし、**-qnotempinc** を指定しても、 **-qtemplateregistry** が暗黙指定されるわけではありません。

-qtempinc または **-qtemplateregistry** のいずれかを指定すると、**-qtmplinst=auto** が暗黙指定されます。

事前定義マクロ

__TEMPINC__ は、**-qtempinc** が有効である場合は 1 に事前定義され、それ以外の場合は、定義されません。

例

ファイル `myprogram.C` をコンパイルして、テンプレート関数について生成された組み込みファイルを `/tmp/mytemplates` ディレクトリーに配置するには、以下のように入力します。

```
xlc++ myprogram.C -qtempinc=/tmp/mytemplates
```

関連情報

- 427 ページの『#pragma implementation (C++ のみ)』
- 374 ページの『-qtmplinst (C++ のみ)』
- 368 ページの『-qtemplateregistry (C++ のみ)』
- 367 ページの『-qtemplaterecompile (C++ のみ)』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『C++ テンプレートの使用』

-qtemplatedepth (C++ のみ)

カテゴリー

テンプレート制御

プラグマ同等物

なし。

目的

再帰的にインスタンス化され、コンパイラーによって処理されるテンプレートの特殊化の最大数を指定する。

構文

デフォルト

-qtemplatedepth=300

パラメーター

number
テンプレートのインスタンスの再帰的な生成の最大数。number は 1 から INT_MAX までの値を取ることができます。コードが *number* よりも多くのテンプレートのインスタンスを再帰的に生成しようとする場合、コンパイルは停止し、エラー・メッセージが発行されます。無効値を指定する場合、デフォルト値の 300 が使用されます。

使用法

このオプションを高い値に設定すると、生成されたコードの複雑性および量が原因でメモリー不足エラーを引き起こすことがあるので注意してください。

事前定義マクロ

なし。

例

myprogram.cpp で以下のコードを正常にコンパイルするには、次のようにします。

```
template <int n> void foo() {
  foo<n-1>();
}

template <> void foo<0>() {}

   int main() {
   foo<400>();
}
```

以下のように入力します。

```
xlc++ myprogram.cpp -qtemplatedepth=400
```

関連情報

- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『C++ テンプレートの使用』

-qtemplaterecompile (C++ のみ)

カテゴリー

テンプレート制御

プラグマ同等物

なし。

目的

-qtemplateregistry コンパイラー・オプションを使用してコンパイルされたコンパイル単位間の依存性の管理に役立つ。

構文

デフォルト

-qtemplaterecompile

使用法

以前コンパイルされたソース・ファイルが再度コンパイルされる場合、**-qtemplaterecompile** オプションは、テンプレート・レジストリーに問い合わせて、このソース・ファイルへの変更が他のコンパイル単位の再コンパイルを必要とするかどうかを判断します。これは、指定されたインスタンス化およびそのインスタンス化を以前含んでいた対応オブジェクト・ファイルを最早参照しないような方法でソース・ファイルが変更された場合に起こります。この場合、影響を受けるコンパイル単位は自動的に再コンパイルされます。

-qtemplaterecompile オプションでは、コンパイラーによって生成されたオブジェクト・ファイルが、最初に書き込まれた サブディレクトリー内に残ることが必要となります。自動ビルド・プロセスがオブジェクト・ファイルを元の サブディレクトリーから移動した場合、 **-qtemplateregistry** が使用可能になっているときは、必ず **-qnotemplaterecompile** オプションを使用してください。

事前定義マクロ

なし。

関連情報

- 374 ページの『-qtmplinst (C++ のみ)』
- 365 ページの『-qtempinc (C++ のみ)』
- 『-qtemplateregistry (C++ のみ)』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『C++ テンプレートの使用』

-qtemplateregistry (C++ のみ)

カテゴリー

テンプレート制御

プラグマ同等物

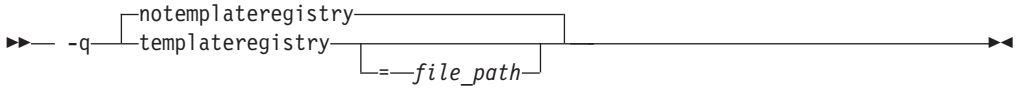
なし。

目的

ソース内で検出されるすべてのテンプレートについてそのレコードを保守し、テンプレートごとに一度だけインスタンス化が行われることを保証する。

コンパイラーが最初にテンプレートのインスタンス化の参照を検出すると、そのインスタンスが生成され、それに関連したオブジェクト・コードが現行オブジェクト・ファイルに配置されます。別のコンパイル単位で、同じテンプレートの同一のインスタンス化への参照がさらにある場合、それらの参照は記録はされますが、冗長のインスタンス化が生成されることはありません。 **-qtemplateregistry** オプションを使用するのに、特別なファイル編成は必要ありません。

構文



デフォルト

-qnotemplateregistry

パラメーター

file_path
テンプレートのインスタンス化情報を含むファイルのパス。場所を指定しない場合、コンパイラーは、現行作業ディレクトリーに保管されている `templateregistry` ファイルにすべてのテンプレート・レジストリー情報を保管します。

使用法

テンプレート・レジストリー・ファイルを異なるプログラム間で共有することはできません。ソースが同じディレクトリーに入っている複数のプログラムがある場合、現行作業ディレクトリーに保管されているデフォルトのテンプレート・レジストリー・ファイルに依存すると、誤った結果が作成される可能性があります。

-qtempinc コンパイラー・オプションと **-qtemplateregistry** コンパイラー・オプションは、相互に排他的です。**-qtempinc** を指定すると、 **-qnotemplateregistry** が暗黙指定されます。同様に、**-qtemplateregistry** を指定すると、**-qnotempinc** が暗黙指定されます。ただし、**-qnotemplateregistry** を指定しても **-qtempinc** が暗黙指定されるわけではありません。

-qtempinc または **-qtemplateregistry** のいずれかを指定すると、**-qtmplinst=auto** が暗黙指定されます。

事前定義マクロ

なし。

例

ファイル myprogram.C をコンパイルしてテンプレートのレジストリー情報を /tmp/mytemplateregistry ファイルに配置するには、以下のように入力します。

```
xlc++ myprogram.C -qtemplateregistry=/tmp/mytemplateregistry
```

関連情報

- 374 ページの『-qtmplinst (C++ のみ)』
- 365 ページの『-qtempinc (C++ のみ)』
- 367 ページの『-qtemplaterecompile (C++ のみ)』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『C++ テンプレートの使用』

-qtempmax (C++ のみ)

カテゴリー

テンプレート制御

プラグマ同等物

なし。

目的

-qtempinc オプションにより各ヘッダー・ファイルごとに生成されるテンプレート組み込みファイルの最大数を指定する。

構文

デフォルト

-qtempmax=1

パラメーター

number
テンプレート組み込みファイルの最大数。number は 1 から 99 999 までの値を取ることができます。

使用法

このオプションは、 **-qtempinc** オプションによって生成されるファイルのサイズが大きくなり過ぎて、新しいインスタンスの作成時の再コンパイルに膨大な時間がかかるときに使用してください。

インスタンス化は、複数のテンプレート組み込みファイルにわたって行われます。

事前定義マクロ

なし。

関連情報

- 365 ページの『-qtempinc (C++ のみ)』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『C++ テンプレートの使用』

-qthreaded

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

スレッド・セーフ・コードを生成する必要があるかどうかをコンパイラーに指示する。

マルチスレッド・アプリケーションをコンパイルまたはリンクする場合は、このオプションを必ず使用してください。このオプションはコードをスレッド・セーフにしませんが、既にスレッド・セーフのコードは、コンパイルおよびリンク後もスレッド・セーフのままになることを保証します。また、すべての最適化がスレッド・セーフであることも保証します。

構文

デフォルト

- **_r** サフィックスが付いていない呼び出しコマンドには **-qnothreaded** を使用します。
- **_r** サフィックスが付いたすべての呼び出しコマンドには **-qthreaded** を使用します。

使用法

このオプションはコンパイル操作とリンカー操作の両方に適用されます。

スレッド・セーフティーを保守するためには、**-qthreaded** オプションを指定してコンパイルされたファイルは、オプション選択によって明示的に行われた場合も **_r** コンパイラー呼び出しモードの選択によって暗黙的に行われた場合も、 **-qthreaded** オプションを使用してリンクする必要があります。

事前定義マクロ

なし。

関連情報

- 335 ページの『-qsmp』

-qtimestamps

カテゴリー

81 ページの『出力制御』

プラグマ同等物

なし。

目的

暗黙的なタイム・スタンプがオブジェクト・ファイルに挿入されるようにするかどうかを制御する。

構文

デフォルト

-qtimestamps

使用法

デフォルトでは、コンパイラーはオブジェクト・ファイルの作成時にその中に暗黙のタイム・スタンプを挿入します。場合によっては、比較ツールがそのようなバイナリーの情報を正しく処理できないおそれがあります。タイム・スタンプの生成を制御することにより、この問題を回避することができます。タイム・スタンプを省略するには、**-qnotimestamps** オプションを使用します。

プラグマまたはその他の明示的な手段によって挿入されたタイム・スタンプはこのオプションの対象になりません。

-qtls

カテゴリー

オブジェクト・コード制御

プラグマ同等物

なし。

目的

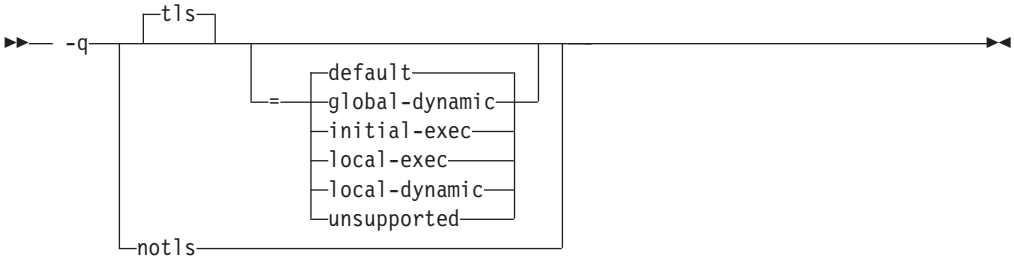
スレッド・ローカル・ストレージが割り振られる変数を指定する `__thread` ストレージ・クラス指定子の認識を使用可能にし、使用するスレッド・ローカル・ストレージ・モデルを指定する。

このオプションが有効な場合、`__thread` ストレージ・クラス指定子によってマークされた変数はマルチスレッド・アプリケーションでそれぞれのスレッドに対してローカルとして取り扱われます。実行時に、変数にアクセスする各スレッドにその変

数のコピーが作成され、その変数のコピーはスレッドが終了すると破棄されます。ご使用のアプリケーションの並列化に使用できる他の高水準の構造体のように、スレッド・ローカル・ストレージによって、スレッドを低水準で同期化する必要なく、グローバル・データへの競合状態が回避されます。

サブオプションを使用すると、スレッド・ローカル・ストレージ・モデルを指定することができます。これによって、パフォーマンスは向上しますが、適用可能性は制限されます。

構文



デフォルト

-qtls=default

パラメーター

unsupported
__thread キーワードは認識されず、スレッド・ローカル・ストレージは使用可能になりません。このサブオプションは **-qnotls** と同等です。

global-dynamic
このモデルは最も一般的で、すべてのスレッド・ローカル変数に使用することができます。

initial-exec
このモデルを使用すると、グローバルに動的またはローカルに動的なモデルよりもパフォーマンスを向上させることができます。またこのモデルは、動的にロードされたモジュールで定義されたスレッド・ローカル変数に使用することができます。ただし、これらのモジュールは実行可能ファイルと同時にロードされていなければなりません。つまり、このモデルは、dlopen によってロードされないモジュールにおいて、すべてのスレッド・ローカル変数が定義されるときにのみに使用することができます。

local-dynamic
このモデルを使用すると、グローバルに動的なモデルよりもパフォーマンスを向上させることができます。またこのモデルは、動的にロードされたモジュールで定義されたスレッド・ローカル変数に使用することができます。ただし、このモデルは、スレッド・ローカル変数へのすべての参照がその変数が定義されるのと同じモジュール内に含まれている場合にのみ使用することができます。

local-exec
このモデルのパフォーマンスはすべてのモジュールの中では最高ですが、このモデルを使用できるのは、主な実行可能ファイルがすべてのスレッド・ローカル変数を定義して参照するときのみです。

default
-qpic オプションの設定に応じて適切なモデルを使用します。これによって位置独立コードを生成するかどうかが判別されます。**-qpic** が有効な場合、このサブオプションは結果として **-qtls=global-dynamic** になります。**-qnopic** が有効な場合、このサブオプションは結果として **-qtls=initial-exec** (**-qpic** はデフォルトで 64 ビット・モードで有効で、使用不可にできません) になります。

サブオプションなしで **-qtls** を指定することは **-qtls=default** と同等です。

事前定義マクロ

なし。

関連情報
- 303 ページの『-qpic』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『__thread ストレージ・クラス指定子』

-qtmplinst (C++ のみ)

カテゴリー

テンプレート制御

プラグマ同等物

なし。

目的

テンプレートの暗黙のインスタンス生成を管理する。

構文

デフォルト

-qtmplinst=auto

パラメーター

always
暗黙のインスタンス生成を常に実行するようコンパイラーに命令します。これが指定されている場合は、**-qtempinc** と **-qtemplateregistry** コンパイラー・オプションは無視されます。

auto
-qtempinc および **-qtemplateregistry** オプションに応じて暗黙のインスタンス生成を管理します。 **-qtempinc** および **-qtemplateregistry** の両方が使用不可の場合、暗黙のインスタンス生成は必ず実行されます。そうでない場合は、オプションの 1 つが使用可能な場合、コンパイラーがそのオプションに従って暗黙のインスタンス生成を管理します。

noinline
暗黙のインスタンス生成を実行しないようコンパイラーに命令します。これが指定されている場合は、**-qtempinc** および **-qtemplateregistry** コンパイラー・オプションは無視されます。

none
インライン関数のインスタンスのみを生成するようコンパイラーに命令します。他の暗黙のインスタンス生成は実行されません。これが指定されている場合は、**-qtempinc** と **-qtemplateregistry** コンパイラー・オプションは無視されます。

使用法

#pragma do_not_instantiate を使用して、選択したテンプレート・クラスの暗黙のインスタンス生成を抑制することもできます。『 417 ページの『#pragma do_not_instantiate (C++ のみ)』』を参照してください。

事前定義マクロ

なし。

関連情報

- 368 ページの『-qtemplateregistry (C++ のみ)』
- 365 ページの『-qtempinc (C++ のみ)』
- 417 ページの『#pragma do_not_instantiate (C++ のみ)』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『明示的インスタンス生成』

-qtmplparse (C++ のみ)

カテゴリー

テンプレート制御

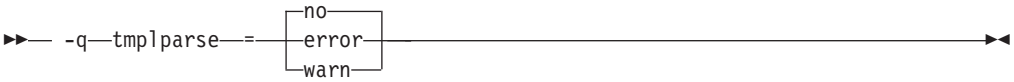
プラグマ同等物

なし。

目的

構文解析とセマンティック検査がテンプレート定義に適用されるかどうかを制御する。

構文



デフォルト

-qtmplparse=no

パラメーター

error
テンプレートのインスタンスが生成されない場合でも、テンプレート定義内の問題をエラーとして扱います。

no テンプレート定義を構文解析しません。これにより、前のバージョンの VisualAge C++ および先行製品用に作成されたコードで発生するエラーの数を減らすことができます。

warn
テンプレート定義を構文解析し、意味エラーの場合に警告メッセージを発行します。

使用法

このオプションは、テンプレートのインスタンス化ではなく、テンプレート定義に適用されます。このオプションの設定に関係なく、定義の外で問題が起こるとエラー・メッセージが生成されます。例えば、以下のように構造体の構文解析またはセマンティック検査中にエラーが見つかると、必ずメッセージが生成されます。

- 関数テンプレートの戻りの型
- 関数テンプレートのパラメーター・リスト

事前定義マクロ

なし。

関連情報

- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『C++ テンプレートの使用』

-qtrigraph

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

キーボードにない文字を表すために、 3 文字表記キーの組み合わせの認識を使用可能にする。

構文

デフォルト

-qtrigraph

使用法

3 文字表記は、すべてのキーボードで使用できない文字の作成を可能にする 3 つのキーの文字の組み合わせです。詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の 『3 文字表記』を参照してください。

C++ デフォルトの **-qtrigraph** 設定をオーバーライドするには、コマンド行で **-qlanglvl** オプションの*後に* **-qnotrigraph** を指定してください。

事前定義マクロ

なし。

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『3 文字表記』
- 146 ページの『-qdigraph』
- 227 ページの『-qlanglvl』

-qtune

カテゴリー

最適化およびチューニング

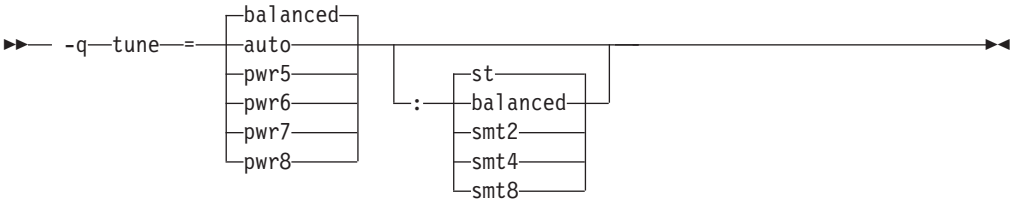
プラグマ同等物

なし。

目的

特定のハードウェア・アーキテクチャーで最も効率よく実行できるように、命令選択、スケジューリング、およびその他のアーキテクチャー依存のパフォーマンスの強化をチューニングする。ターゲット SMT モードの指定で、そのモードで最高のパフォーマンスが得られる最適化を指示できるようにする。

構文



デフォルト

-qtune=balanced:balanced (有効な **-qarch** 設定がない場合)。それ以外の場合は、デフォルトは、有効な **-qarch** 値によって異なります。詳しくは、『379 ページの表 33』を参照してください。

CPU サブオプション向けのパラメーター

以下の **-qtune** CPU サブオプションを使用すると、特定のアーキテクチャーを指定して、コンパイラーが最高のパフォーマンスを発揮できるようにすることができます。

auto
アプリケーションがコンパイルされるプラットフォームで最適化がチューニングされます。

balanced
最適化が、最新ハードウェアの選択範囲全体で調整されます。

pwr5
POWER5 ハードウェア・プラットフォーム用に、最適化が調整されます。

pwr6
POWER6 ハードウェア・プラットフォーム用に、最適化が調整されます。

pwr7
POWER7 または POWER7+ ハードウェア・プラットフォーム用に、最適化が調整されます。

pwr8
POWER8 ハードウェア・プラットフォーム用に、最適化が調整されます。

SMT サブオプション向けのパラメーター

以下の **-qtune** 同時マルチスレッド化 (SMT) サブオプションを使用すると、オプションでコンパイラーの実行モードを指定して、最高のパフォーマンスを発揮できるようにすることができます。

balanced
選択された範囲の最新ハードウェア用のさまざまな SMT モード全体でパフォーマンスが得られるよう、最適化が調整されます。

st 単一スレッド実行用に、最適化が調整されます。

smt2
SMT2 実行モード (2 スレッド) 用に、最適化が調整されます。

smt4
SMT4 実行モード (4 スレッド) 用に、最適化が調整されます。

smt8
SMT8 実行モード (8 スレッド) 用に、最適化が調整されます。

使用法

プログラムを複数のアーキテクチャーで稼働させるけれど、特定のアーキテクチャーで調整したい場合は、 **-qarch** および **-qtune** オプションを組み合わせて使用することができます。これらのオプションは、基本的には浮動小数点中心のプログラムに有効です。

キャッシュ・サイズおよびパイプラインなどのハードウェア・フィーチャーを最大限に活用するように、生成されたマシン命令を配置 (スケジューリング) することによって、**-qtune** オプションはパフォーマンスを改善することができます。このオプションは、最適化を使用可能にするオプションと組み合わせて使用した場合にのみ効果があります。

特定の **SMT** サブオプションは、有効な **-qarch** オプションが指定の **SMT** モードをサポートしている場合に有効です。**-qarch** および **SMT** 調整オプションの使用可能な組み合わせは、表 33 にリストされています。**-qarch/-qtune SMT** 組み合わせが無効な場合、コンパイラーはそれらを無視します。

-qtune 設定を変更すると、その結果作成される実行可能ファイルのパフォーマンスに影響する場合がありますが、実行可能ファイルが特定のハードウェア・プラットフォーム上で正しく実行できるかどうかには、まったく影響を与えません。

受け入れられる **-qarch** と **-qtune** の組み合わせを、次の表に示します。

表 33. 受け入れ可能な **-qarch/-qtune** *組み合わせ*

-qarch オプション	**デフォルトの -qtune 設定**	**使用可能な -qtune CPU 設定**	**使用可能な -qtune SMT 設定**
ppc	balanced:balanced	auto | pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
ppcgr	balanced:balanced	auto |pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
ppc64	balanced:balanced	auto |pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
ppc64gr	balanced:balanced	auto |pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
ppc64grsq	balanced:balanced	auto |pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
ppc64v	balanced:balanced	auto | pwr6 | pwr7 | pwr8 | balanced	balanced | st
pwr5	pwr5:st	auto | pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
pwr5x	pwr5:st	auto | pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st | smt2
pwr6	pwr6:st	auto | pwr6 | pwr7 | pwr8 | balanced	balanced | st | smt2
pwr6e	pwr6:st	auto | pwr6 | balanced	balanced | st

表 33. 受け入れ可能な **-qarch/-qtune** 組み合わせ (続き)

-qarch オプション	**デフォルトの -qtune 設定**	**使用可能な -qtune CPU 設定**	**使用可能な -qtune SMT 設定**
pwr7	pwr7:st	auto | pwr7 | pwr8 | balanced	balanced | st | smt2 | smt4
pwr8	pwr8:st	auto | pwr8 | balanced	balanced | st | smt2 | smt4 | smt8

事前定義マクロ

なし。

例

`myprogram.c` からコンパイルされた実行可能プログラム `testing` を POWER7 ハードウェア・プラットフォーム用に最適化するように指定するには、以下を入力します。

```
xlc -o testing myprogram.c -qtune=pwr7
```

`myprogram.c` からコンパイルされた実行可能プログラム `testing` が、SMT4 モード用に構成された POWER8 ハードウェア・プラットフォームで最適化されるよう指定するには、以下を入力します。

```
xlc -o testing myprogram.c -qtune=pwr8:smt4
```

関連情報

- 113 ページの『-qarch』
- 104 ページの『-q32、-q64』
- 11 ページの『アーキテクチャー固有のコンパイルでのコンパイラー・オプションの指定』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の 『アプリケーションの最適化』

-U

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

コンパイラーまたは **-D** コンパイラー・オプションによって定義されたマクロ名の定義を解除する。

構文

デフォルト

多くのマクロがコンパイラーによって事前定義されます。定義解除できるマクロ (つまり、*保護*されないマクロ) については、481 ページの『第 6 章 コンパイラーの事前定義マクロ』を参照してください。コンパイラー構成ファイルは、**-D** オプションを使用して、特定の呼び出しコマンドの複数のマクロ名も事前定義します。詳しくは、ご使用のシステムの構成ファイルを参照してください。

パラメーター

name
定義解除するマクロ。

使用法

-U オプションは、`#undef` プリプロセッサー・ディレクティブと同等ではありません。`#define` プリプロセッサー・ディレクティブによってソースに定義された名前を定義解除することはできません。定義解除できるのは、コンパイラーまたは **-D** オプションによって定義された名前のみです。

-U*name* オプションは、**-D***name* オプションよりも高い優先順位を持っています。

事前定義マクロ

なし。

例

ご使用のオペレーティング・システムが名前 `__unix` を定義していても、その名前 `__unix` の定義が以下を入力することによって無効になるように、その名前が定義される条件でコード・セグメントをコンパイル `myprogram.c` で入力しない場合を想定します。

```
xlc myprogram.c  -U__unix
```

関連情報

- 141 ページの『-D』

-qunroll

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma options [no]unroll[= yes|no|auto|n]

#pragma unroll

目的

パフォーマンスを改善するために、ループのアンロールを制御する。

unroll が有効な場合、最適化プログラムはループごとに最適なアンロール係数を判別して適用します。場合によっては、不要な分岐を避けるようにループ制御が変更される場合もあります。コンパイラーは、ループが実際にアンロールされるかどうかの最終アービターのままです。 **#pragma unroll** ディレクティブを使用すると、アンロールについてさらに細かく制御することができます。

構文

オプション構文

プラグマ構文

デフォルト

-qunroll=auto

パラメーター

auto (オプションのみ)
基本的なループのアンロールを実行するようにコンパイラーに命令する。

yes (オプションのみ)
auto を実行する機会よりも、ループのアンロールの機会を検索するようにコンパイラーに命令します。一般に、このサブオプションには、**auto** 処理よりもコンパイル時間またはプログラム・サイズを増やす機会が多くありますが、このサブオプションによってご使用のアプリケーションのパフォーマンスが向上する場合もあります。

no (オプションのみ)
ループをアンロールしないようにコンパイラーに命令します。

n コンパイラーに係数 *n* でループをアンロールするように命令します。言い替えれば、ループの本体は複製されて *n* 個のコピーが作成され、繰り返しの数は係数 *1/n* の分だけ削減されます。 **-qunroll=n** オプションは、アンロール・プラグマをまだ持っていないすべてのループに影響を与える、グローバル・アンロール係数を指定します。 *n* の値は、正の整数でなければなりません。

#pragma unroll(1) または **-qunroll=1** を指定すると、ループのアンロールが無効になります。この指定は、**#pragma nounroll** または **-qnounroll** と同等です。*n* が指定されておらず、**-qhot**、**-qsmp**、**-O4**、または **-O5** が指定されている場合は、最適化プログラムがそれぞれのネストされたループの適切なアンロール係数を決定します。

サブオプションなしで **-qunroll** を指定することは、 **-qunroll=yes** を指定することと同等です。

-qnounroll は **-qunroll=no** と同等です。

使用法

プラグマは、指定されたループの オプション設定をオーバーライドします。ただし、**#pragma unroll** が特定のループに指定されていても、コンパイラーは、ループが実際にアンロールされるかどうかの最終アービターのままです。

ループでは、1 つのプラグマしか指定されないことがあります。プラグマはループまたは **#pragma block_loop** ディレクティブの直前に現れなければ、影響を及ぼすことができません。

プラグマが影響を与えるのは、プラグマに後続のループのみです。希望するループ・アンロール方法が、一般的な オプションによる方法とは異なる場合には、内側のネストされたループに、**#pragma unroll** ディレクティブを先行させるすることが必要です。

#pragma unroll および **#pragma nounroll** ディレクティブは、`for` ループまたは **#pragma block_loop** ディレクティブでのみ使用できます。これらは、`do while` および `while` ループには適用できません。

ループ構造体は、以下の条件を満たしている必要があります。

- 1 つのループ・カウンター変数、その変数に対する 1 つの増分ポイント、および 1 つの終了変数のみが存在している必要があります。これらはループ・ネストのどのポイントでも変更できません。
- ループは複数の入り口点と出口点を持つことはできません。ループの終了がループを終了するための唯一の方法でなければなりません。
- ループ内の依存関係は「後方参照」にすることはできません。例えば、 `A[i][j] = A[i -1][j + 1] + 4` などのステートメントがループ内に現れることはできません。

事前定義マクロ

なし。

例

以下の例では、最初の `for` ループの **#pragma unroll(3)** ディレクティブが、ループの本体を 3 回複製するようにコンパイラーに要求します。2 番目の `for` ループの **#pragma unroll** によって、コンパイラーはアンロールを実行するかどうかを決定できます。

```
#pragma unroll(3)
for( i=0;i < n; i++)
{
      a[i] = b[i] * c[i];
}

#pragma unroll
for( j=0;j < n; j++)
```

```
{
      a[j] = b[j] * c[j];

}
```

この例における、最初の **#pragma unroll(3)** ディレクティブの結果は以下のとおりです。

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
  a[i]=b[i] * c[i];
  a[i+1]=b[i+1] * c[i+1];
  a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
  remainder:
  for (; i<n; i++) {
    a[i]=b[i] * c[i];
  }
}
```

関連情報

- 410 ページの『#pragma block_loop』
- 430 ページの『#pragma loopid』
- 452 ページの『#pragma stream_unroll』
- 454 ページの『#pragma unrollandfuse』

-qunwind

カテゴリー

最適化およびチューニング

プラグマ同等物

なし。

目的

スタックに保管されたレジスターを検索するコードによって呼び出しスタックをアンワインドできるかどうかを指定する。

-qnounwind を指定すると、スタックがアンワインドされないということをコンパイラーに表明することになり、不揮発性レジスターの保管および復元の最適化を改善することができます。

構文

デフォルト

-qunwind

使用法

ライブラリー関数の `setjmp` および `longjmp` ファミリーは **-qnounwind** と使用しても安全です。

C++ **-qnounwind** を指定すると、**-qnoeh** も暗黙指定されます。

事前定義マクロ

なし。

関連情報

- 151 ページの『-qeh (C++ のみ)』

-qupconv (C のみ)

カテゴリー

移植性とマイグレーション

プラグマ同等物

#pragma options [no]upconv

目的

整数拡張が実行されるときに符号なしの指定が保存されるかどうかを指定する。

noupconv が有効な場合は、`int` より小さい `unsigned` 型が整数拡張中に `int` に変換されます。**upconv** が有効な場合は、これらの型が整数拡張中に `unsigned int` に変換されます。この拡張規則は、`int` より大きな型には適用されません。

構文

デフォルト

- **classic** または **extended** 以外のすべての言語レベルの **-qnoupconv**
- **classic** または **extended** 言語レベルが有効な場合は **-qupconv**

使用法

符号の保存は C の古い方言との互換性のために提供されています。ANSI C 標準では符号保存と対立する値の保存が必要です。

事前定義マクロ

なし。

例

`int` より小さいすべての `unsigned` 型を `unsigned int` に変換するよう `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qupconv
```

次の短いリストは、**-qupconv** の効果を示したものです。

```
#include <stdio.h>
int main(void) {
  unsigned char zero = 0;
  if (-1 <zero)
    printf("Value-preserving rules in effect¥n");
  else
    printf("Unsignedness-preserving rules in effect¥n");
  return 0;
}
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『通常の算術変換』
- 227 ページの『-qlanglvl』

-qutf

カテゴリー

言語エレメント制御

プラグマ同等物

なし。

目的

UTF リテラル構文の認識を使用可能にする。

構文

デフォルト

- C -qnoutf
- C++ **-qlanglvl=strict98** 以外のすべての言語レベルには -qutf

使用法

コンパイラーは **iconv** を使用してソース・ファイルをユニコードに変換します。ソース・ファイルを変換できない場合、コンパイラーは **-qutf** オプションを無視して、警告を発行します。

事前定義マクロ

なし。

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『UTF リテラル』

-v, -V

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

呼び出されているプログラムと各プログラムに指定されているオプションの名前を戻すことによって、コンパイルの進行を報告する。

-v オプションが有効な場合、情報はコンマで区切られたリストに表示されます。**-V** オプションが有効な場合、情報はスペースで区切られたリストに表示されます。

構文

デフォルト

コンパイラーはコンパイルの進行を表示しません。

使用法

-v および **-V** オプションが **-#** オプションによってオーバーライドされます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルして、コンパイルの進行を監視したり、コンパイルの進行、呼び出し中のプログラム、および指定されているオプションを記述するメッセージを表示できるようにするには、以下のように入力します。

```
xlc myprogram.c -v
```

関連情報

- 103 ページの『-# (ポンド記号)』

-qversion

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

呼び出しているコンパイラーのバージョンおよびリリースを表示する。

構文

デフォルト

-qnoversion

パラメーター

verbose
インストール済みコンパイラー・コンポーネントのそれぞれのバージョン、リリース、およびレベルについての情報を追加表示します。

使用法

-qversion を指定すると、コンパイラーは、バージョン情報を表示し終了するので、コンパイルは停止されます。出力オブジェクト・ファイルにこの情報を保存する場合は、**-qsaveopt -c** オプションを指定して保存することができます。

verbose サブオプションを指定しないで **-qversion** を指定すると、次の形式でコンパイラー情報が表示されます。

*product_name*Version: *VV.RR.MMMM.LLLL*

ここで、
V バージョンを表します。
R リリースを表します。
M 変更を表します。
L レベルを表します。

詳しくは、『例 1』を参照してください。

-qversion=verbose は、以下の形式でコンポーネント情報を表示します。

component_name Version: *VV.RR*(*product_name*) Level: *component_build_date ID: component_level_ID*

ここで、
component_name
低水準最適化プログラムなどのインストール済みコンポーネントを指定します。
component_build_date
インストール済みコンポーネントのビルド日付を表します。

component_level_ID
インストール済みコンポーネントのレベルに関連付けられている ID を表します。

詳しくは、『例 2』を参照してください。

事前定義マクロ

なし。

例

例 1: 以下は、**-qversion** オプションを指定した場合の出力です。

```
IBM XL C/C++ for Linux, V13.1
Version: 13.01.0000.0001
```

例 2 以下は、**-qversion=verbose** オプションを指定した場合の出力です。

```
IBM XL C/C++ for Linux, V13.1
Version: 13.01.0000.0001
Driver Version: 13.01(C/C++) Level: 121020 ID: _acSDAheyEeK928eKYVtYGg
C Front End Version: 13.01(C/C++) 121021 ID: _mP6oMBvbEeK928eKYVtYGg
C++ Front End Version: 13.01(C/C++) Level: 121019 ID: _P_0O4hpJEeK928eKYVtYGg
High-Level Optimizer Version: 13.01(C/C++) and 15.01(Fortran) Level: 121021 ID:
_d6ZMohn3EeK928eKYVtYGg
Low-Level Optimizer Version: 13.01(C/C++) and 15.01(Fortran) Level: 121020 ID: _
H1wQwhn3EeK928eKYVtYGg
```

関連情報

- 325 ページの『-qsaveopt』

-qvisibility

カテゴリー

最適化およびチューニング

プラグマ同等物

#pragma GCC visibility push (default | protected | hidden | internal)

#pragma GCC visibility pop

目的

オブジェクト・ファイル内の外部リンケージ・エンティティーに visibility 属性を指定する。外部リンケージ・エンティティーがプラグマ・ディレクティブ、明示的に指定された属性、または伝搬規則から visibility 属性を取得しない場合、それらのエンティティーは、**-qvisibility** オプションによって指定された visibility 属性を持ちます。

構文

デフォルト

-qvisibility=default

パラメーター

default
影響を受ける外部リンケージ・エンティティーが default の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされ、優先使用できます。

protected
影響を受ける外部リンケージ・エンティティーが protected の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされますが、優先使用できません。

hidden
影響を受ける外部リンケージ・エンティティーが hidden の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされませんが、それらのアドレスをポインター経由で間接的に参照できます。

internal
影響を受ける外部リンケージ・エンティティーが internal の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされず、それらのアドレスは共有ライブラリー内の他のモジュールでは使用できません。

制約事項: このリリースでは、hidden および internal の visibility 属性は同じです。これらいずれかの visibility 属性を使用して指定されたエンティティーのアドレスは、ポインター経由で間接的に参照できます。

使用法

-qvisibility オプションは、外部リンケージ・エンティティーに対してグローバルに visibility 属性を設定し、1 つのモジュールで定義されているエンティティーを他のモジュールで参照または使用できるかどうか、およびその方法を記述します。エンティティーの visibility 属性は外部リンケージを持つエンティティーにのみ影響します。他のエンティティーの可視性を増加させることはできません。エンティティーの優先使用は、リンク時にエンティティー定義が解決されるにもかかわらず、ランタイム時に別のエンティティー定義に置き換えられる場合に発生します。

事前定義マクロ

なし。

例

コンパイル単位 myprogram.c で protected の visibility 属性を持つ外部リンケージ・エンティティーを設定するには、**-qvisibility=protected** オプションを指定して myprogram.c をコンパイルします。

```
xlc myprogram.c -qvisibility=protected -c
```

外部リンケージ・エンティティーがプラグマ・ディレクティブ、明示的に指定された属性、または伝搬規則から visibility 属性を取得しない場合、myprogram.c ファイル内のすべてのエンティティーの visibility 属性は protected になります。

関連情報

- 277 ページの『-qmkshrobj』
- 421 ページの『#pragma GCC visibility push、#pragma GCC visibility pop』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『visibility 属性の使用 (IBM 拡張)』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『外部リンケージ』、 『visibility 変数属性 (IBM 拡張)』、 『visibility 関数属性 (IBM 拡張)』、 『visibility 型属性 (C++ のみ) (IBM 拡張)』、および『visibility 名前空間属性 (C++ のみ) (IBM 拡張)』

-qvrsave

カテゴリー

オブジェクト・コード制御

プラグマ同等物

#pragma altivec_vrsave

目的

VRSAVE レジスターを保守するための関数のプロローグとエピローグのコードを使用可能にする。

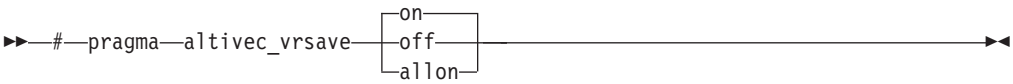
VRSAVE レジスターのそれぞれのビットはベクトル・レジスターに対応し、1 に設定されている場合は、コンテキスト・スイッチが発生したときに保管されるデータが対応するベクトルに含まれることを示します。**-qvrsave** を使用して、コンパイル単位の関数に VRSAVE レジスターの保守に必要なコードが組み込まれることをコンパイラーに示します。**-qnovrsave** を使用して、コンパイル単位の関数に VRSAVE レジスターの保守に必要なコードが組み込まれないことをコンパイラーに示します。

プログラム・ソース内の個々の関数に対するこのコンパイラー・オプションの現行設定をオーバーライドするには、このプラグマを使用することができます。

構文

オプション構文

プラグマ構文



デフォルト

vrsave: VRSAVE レジスターは常に保守されます。

パラメーター

on (プラグマのみ)
関数のプロローグとエピローグに VRSAVE レジスターを保守するためのコードが組み込まれます。

off (プラグマのみ)
関数のプロローグとエピローグに VRSAVE レジスターを保守するためのコードが組み込まれません。

allon (プラグマのみ)
プラグマを含む関数は、VRSAVE レジスターのすべてのビットを 1 に設定することにより、すべてのベクトルが使用され、コンテキスト・スイッチが発生した場合には保管されることを示します。

使用法

このオプションおよびプラグマは、**-qaltivec** が有効な場合にのみサポートされます。

このプラグマは 1 つの関数内でのみ使用でき、その効果はそのプラグマが現れる関数に対してのみ適用されます。同じ関数の中で異なる設定を使用してこのプラグマを指定した場合は、エラー条件が作成されます。

事前定義マクロ

なし。

関連情報

•

112 ページの『-qaltivec』

-w

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

なし。

目的

通知メッセージ、言語レベル・メッセージ、および警告メッセージを抑制する。

C このオプションは、**-qflag=e : e** を指定するのと同等です。C++ このオプションは、**-qflag=s : s** を指定するのと同等です。

構文

デフォルト

すべての通知メッセージおよび警告メッセージが報告されます。

使用法

重大エラーに追加情報を提供する通知メッセージと警告メッセージは、このオプションでは使用不可になりません。

事前定義マクロ

なし。

例

警告メッセージが表示されないように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -w
```

以下の例は、重大エラーの結果生じる通知メッセージ (この場合、C++ における多重定義解決での問題によって生じています) がどのように使用不可にならないかを示しています。

```
void func(int a){}
void func(int a, int b){}
int main(void)
{
func(1,2,3);
return 0;
}
```

出力は以下のようになります。

```
"x.cpp", line 6.4: 1540-0218 (S) The call does not match any parameter list for
                                 "func".
"x.cpp", line 1.6: 1540-1283 (I) "func(int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been
                                 specified for "func(int)".
"x.cpp", line 2.6: 1540-1283 (I) "func(int, int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified
                                 for "func(int, int)".
```

関連情報

- 158 ページの『-qflag』
- 357 ページの『-qsuppress』

-W

カテゴリー

コンパイラーのカスタマイズ

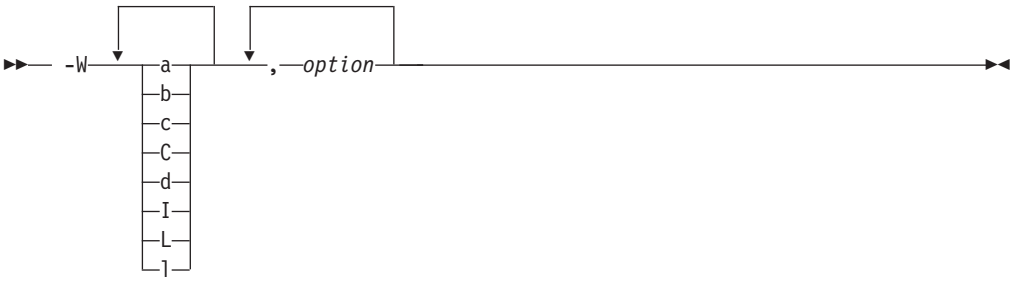
プラグマ同等物

なし。

目的

リストされたオプションをコンパイル中に実行されるコンポーネントに渡す。

構文



パラメーター

オプション
受け渡し先のコンポーネントに有効なオプション。スペースは、*option* の前には表示しません。

以下のテーブルは、**-W** パラメーターとコンポーネント名との間の対応を示しています。

パラメーター	説明	コンポーネント名
a	アセンブラー	as
b	低水準最適化プログラム	xlCcode
c	コンパイラーのフロントエンド	xlcentry、xlCentry

パラメーター	説明	コンポーネント名
C++ C	C++ コンパイラーのフロントエンド	xlCentry
d	逆アセンブラー	dis
I (大文字の i)	高水準最適化プログラム、コンパイル・ステップ	ipa
L	高水準最適化プログラム、リンク・ステップ	ipa
l (小文字の L)	リンカー	ld

使用法

-W オプションの後に続いているストリングでは、各オプションに対して分離文字としてコンマを使用し、スペースは入れないでください。オプション・ストリング内のシェルに特有の文字を入れる必要がある場合は、その文字の前にバックスラッシュを置いてください。例えば、構成ファイルに **-W** オプションを使用する場合、エスケープ・シーケンスの円記号とコンマ (¥,) を使用して、パラメーター・ストリング内にコンマを表示することができます。

ほとんどのオプションは、リンカー **ld** に渡す際に **-W** オプションを使用する必要はありません。**-q** オプション以外の認識されないコマンド行オプションは、自動的にリンカーに渡されるからです。絶対に **-W** を必要とするのは、**-v** や **-S** などのコンパイラー・オプションと同じ名前のリンカー・オプションのみです。

事前定義マクロ

なし。

例

ファイル `file.c` をコンパイルして、リンカー・オプション **-berok** をリンカーに引き渡すには、次のコマンドを入力します。

```
xlc -Wl,-berok file.c
```

ファイル `uses_many_symbols.c` およびアセンブリー・ファイル `produces_warnings.s` をコンパイルして、`produces_warnings.s` がアセンブラー・オプション **-x** とアセンブルされ (警告を発行し相互参照を作成)、オブジェクト・ファイルがオプション **-s** とリンクされるようにするには (オブジェクト・ファイルの書き込みと最終実行可能ファイルのストリップ)、以下のコマンドを発行します。

```
xlc -Wa,-x -Wl,-s produces_warnings.s uses_many_symbols.c
```

関連情報

- 1 ページの『コンパイラーの呼び出し』

-qwarn0x (C++11)

注: IBM は、C++11 (承認前は C++0x と呼ばれていました) の選択された機能をサポートしています。IBM は、この標準の機能の開発および実装を継続します。この言語レベルのインプリメンテーションは、標準に対する IBM の解釈に基づきま

す。新しい C++11 標準ライブラリーのサポートを含め、C++11 のすべての機能を IBM が実装し終えるまで、リリースごとに実装が変更される可能性があります。IBM では、IBM による C++11 の新機能の実装に関し、ソース、バイナリー、リスト作成などのコンパイラー・インターフェースにおいて、以前のリリースとの互換性を維持するための試みは行いません。

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

C++98 標準から C++11 標準へのマイグレーションによって発生したプログラム内の相違点について、ユーザーにメッセージで通知するかどうかを制御する。

例えば、**-qlanglvl=noc99preprocessor** および **-qwarn0x** を指定した場合、C++11 プリプロセッサーは `#if` および `#elif` 条件付き包含ディレクティブ内の制御式を評価し、評価の結果を非 C++11 プリプロセッサーの場合と比較します。それらが異なる場合、コンパイラーは次の警告メッセージを発行します。

```
The preprocessor controlling expression evaluates differently between C++11
and non-C++11 langlvls.
```

別の例として、**-qlanglvl=noc99longlong** オプションと **-qwarn0x** オプションの両方を指定すると、コンパイラーによって、整数リテラルのタイプが C++11 言語レベルと C++11 以外の言語レベルの間で異なることを示すメッセージが表示される場合があります。32 ビット・モードでは、整数リテラル 2147483648 を使用して変数を初期化すると、コンパイラーによって以下のメッセージが出されます。

```
整数定数「2147483648」は、非 C++11 言語レベルでは unsigned long int 型が
暗黙指定されます。また、C++11 言語レベルでは long long int 型が暗黙指定されます。
```

コンパイラーは、同じオプション設定で、リテラル 10000000000000000000 に対しても同様のメッセージを出します。

```
整数定数「10000000000000000000」は、unsigned long long 型が暗黙指定されるか、
C++11 では「-qlanglvl=extendedintegersafe」に許可されません。
(Integral constant "10000000000000000000" has implied type unsigned long long
or is  not allowed with "-qlanglvl=extendedintegersafe" under C++11.)
その暗黙指定された型は、非 C++11 言語レベルでは unsigned long long ではありません。
(Its implied type is not unsigned long long under  non-C++11 language levels.)
```

-qwarn0x オプションが有効な場合、対応する C++11 の機能およびキーワードが使用不可になっていれば、以下のキーワードが出現するたびに、コンパイラーは警告メッセージを出します。

- `constexpr`
- `decltype`
- `static_assert`

例えば、**-qwarn0x** オプションが使用可能な場合に、**-qlanglvl=nostatic_assert** と **-qnokeyword=static_assert** オプションの両方を指定すると、コンパイラーは検出した static_assert を ID トークンとして扱い、static_assert ID が出現するたびに以下のメッセージを出します。

```
C++0x will reserve "static_assert" as a keyword whose C++0x feature can
be enabled by -qlanglvl=static_assert.
```

構文

デフォルト

-qnowarn0x

使用法

このオプションは、**-qwarn0x** が設定されている場合に有効です。

事前定義マクロ

なし。

関連情報

- 227 ページの『-qlanglvl』
- 222 ページの『-qkeyword』

-qwarn64

カテゴリー

エラー・チェックおよびデバッグ

プラグマ同等物

なし。

目的

32 ビットと 64 ビットのコンパイラー・モード間で発生する可能性のあるデータ変換問題の検査を使用可能にする。

-qwarn64 が有効なときに、データ変換によって、64 ビット・コンパイル・モードで問題が発生する可能性がある場合には、以下のような通知メッセージが表示されます。

- long 型から int 型への明示的または暗黙的変換が原因の切り捨て
- int 型から long 型への明示的または暗黙的変換が原因の予期しない結果
- pointer 型から int 型へのキャスト演算による明示的変換が原因の無効なメモリー参照
- int 型から pointer 型へのキャスト演算による明示的変換が原因の無効なメモリー参照

- constants から long 型への明示的または暗黙的変換が原因の問題
- constants から pointer 型へのキャスト演算による明示的または暗黙的変換が原因の問題

構文

デフォルト

-qnowarn64

使用法

このオプションは、32 ビットまたは 64 ビットのいずれのコンパイラー・モードでも機能します。32 ビット・モードでは、32 ビットから 64 ビットへのマイグレーションで発生する可能性のある問題を発見するためのプレビュー・エイドとして機能します。

事前定義マクロ

なし。

関連情報

- 104 ページの『-q32、-q64』
- 19 ページの『コンパイラー・メッセージ』

-qxcall

カテゴリー

オブジェクト・コード制御

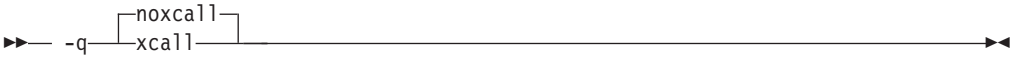
プラグマ同等物

なし。

目的

コンパイル内の静的関数を外部関数であるかのように扱うためのコードを生成する。

構文



デフォルト

-qnoxcall

使用法

-qxcall は、**-qnoxcall** よりも処理が遅いコードを生成します。

事前定義マクロ

なし。

例

すべての静的関数が外部関数としてコンパイルされるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qxcall
```

-qxref

カテゴリー

リスト、メッセージ、およびコンパイラー情報

プラグマ同等物

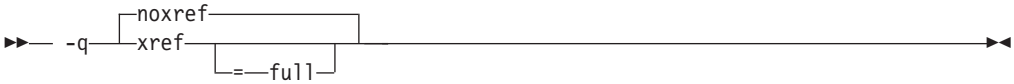
#pragma options [no]xref

目的

リストの属性および相互参照の相互参照コンポーネントを含むコンパイラー・リストを作成する。

xref が有効な場合、コマンド行で指定された各ソース・ファイルに .lst サフィックスが付いたリスト・ファイルが生成されます。リスト・ファイルの内容について詳しくは、22 ページの『コンパイラー・リスト』を参照してください。

構文



デフォルト

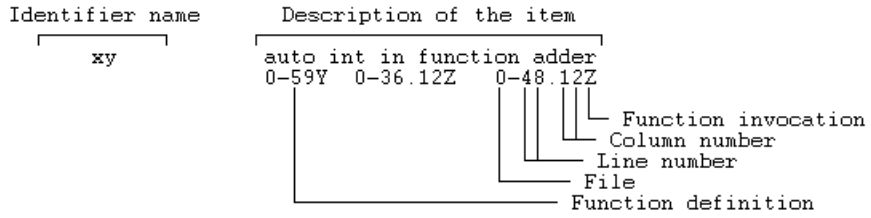
-qnoxref

パラメーター

full
プログラムにある ID をすべて報告します。このサブオプションなしで **xref** を指定すると、使用される ID のみが報告されます。

使用法

一般的な相互参照リストの形式は、以下のとおりです。



リストでは、以下の文字コードが使用されます。

表 34. 相互参照リスト・コード

文字	意味
X	関数が宣言されます。
Y	関数が定義されます。
Z	関数が呼び出されます。
$	タイプが定義され、変数が宣言または定義されます。
#	変数が割り当てられます。
&	変数が定義され、初期化されます。
[ブランク]	ID が参照されます。
{ および }	構造体定義における { 記号および } 記号の位置。

-qnoprint オプションは、このオプションをオーバーライドします。

#pragma mc_func ディレクティブに定義された関数は、いずれもプラグマ・ディレクティブの行に定義されているようにリストされます。

事前定義マクロ

なし。

例

`myprogram.c` をコンパイルし、使用されているかどうかに関係なく、すべての ID の相互参照リストを作成するには、以下のように入力します。

```
xlc myprogram.c -qxref=full
```

関連情報

- 120 ページの『-qattr』
- 433 ページの『#pragma mc_func』

-y

カテゴリー

浮動小数点および整数のコントロール

プラグマ同等物

なし。

目的

コンパイル時に定数浮動小数点式を評価する場合にコンパイラーが使用する丸めモードを指定する。

構文

デフォルト

-yn、-ydn

パラメーター

以下のサブオプションはバイナリー浮動小数点型のみに有効です。

m 負の無限大の方向に丸める。

n 表現可能な偶数の近似値まで丸める。

p 正の無限大の方向に丸める。

z ゼロの方向に丸める。

使用法

ご使用のプログラムに long double に関わる操作が組み込まれている場合は、丸めモードを **-yn** (表現可能な偶数の近似値まで丸める) に設定する必要があります。

事前定義マクロ

なし。

例

浮動小数点定数式をコンパイル時にゼロの方向に丸めるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -yz
```

第 5 章 コンパイラー・プラグマ参照

以下のセクションでは、使用可能なプラグマについて説明します。

プラグマ・ディレクティブ構文

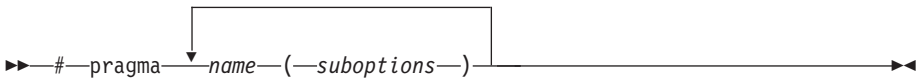
XL C/C++ がサポートするプラグマ・ディレクティブの形式は 3 つです。

#pragma options *option_name*

このプラグマでは、対応するコマンド行オプションとまったく同じ構文が使用されます。このタイプのサポートされているプラグマの正確な構文およびリストは、436 ページの『#pragma options』に記載されています。

#pragma *name*

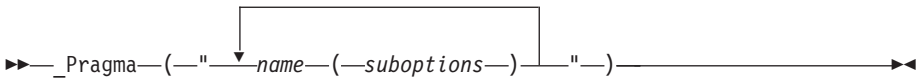
この形式は以下の構文を使用します。



name はプラグマ・ディレクティブ名で、*suboptions* は、適用可能であれば、プラグマに指定できる必須のサブオプションまたは任意指定のサブオプションです。

_Pragma (**"***name***"**)

この形式は以下の構文を使用します。



例えば、以下のステートメントでは、

```
_Pragma ( "pack(1)" )
```

これは以下のステートメントと同じになります。

```
#pragma pack(1)
```

プラグマ・ステートメントのすべての形式では、単一の **#pragma** ステートメントで複数の *name* および *suboptions* を指定できます。

プラグマの *name* は、特に指定がない限り、マクロ置換されます。コンパイラーは、認識されないプラグマを無視し、無視したことを示す通知メッセージを発行します。

C と C++ の両方のコンパイラーでコンパイルされるコード内に、C と C++ のいずれかだけで有効なプラグマが記述されている場合は、このプラグマの前後に条件付きコンパイル・ディレクティブを追加することができます。(認識されないプラグマは無視されるので、これは厳密には必要ではありません。) 例えば、**#pragma object_model** は C++ コンパイラーでのみ認識されるため、プラグマの前後に条件付きコンパイル・ディレクティブを追加することができます。

```
#ifdef __cplusplus
#pragma object_model(pop)
#endif
```

プラグマ・ディレクティブの範囲

多くのプラグマ・ディレクティブは、コンパイル単位のソース・コード内の任意の位置に指定することができます。その他のプラグマ・ディレクティブは、その他のディレクティブまたはソース・コード・ステートメントの前に指定してください。それぞれのプラグマの個々の説明の『使用法』セクションには、プラグマの配置の制約がすべて説明されています。

一般的に、プラグマ・ディレクティブは、ソース・プログラムのどのコードの前に指定しても、組み込まれているヘッダー・ファイルなど、コンパイル単位全体に適用されます。ソース・コードの任意の場所に指定できるディレクティブの場合、指定された位置からコンパイル単位の終わりまで適用されます。

選択したコード・セクションの前後でプラグマ・ディレクティブの相互補完的ペアを使用すると、プラグマの適用範囲をさらに制限することができます。例えば、ソース・コードの選択した部分のみをコンパイラー・リストに組み込むように要求するには、**#pragma options source** および **#pragma options nosource** ディレクティブを以下のように使用します。

```
#pragma options source

/*  Source code between the source and nosource pragma
    options is included in the compiler listing               */

#pragma options nosource
```

多くのプラグマには、「pop」サブオプションまたは「reset」サブオプションが用意されています。これらのサブオプションを使用すると、スタック・ベース方式でプラグマ設定を使用可能にしたり使用不可にすることができます。これらの例は、関連するプラグマの説明の中に記載されています。

機能カテゴリー別コンパイラー・プラグマの要約

使用可能な XL C/C++ プラグマは、以下のカテゴリーにグループ化されています。

- 405 ページの『言語エレメント制御』
- 405 ページの『C++ テンプレート・プラグマ』
- 405 ページの『浮動小数点および整数制御』
- 406 ページの『エラー・チェックおよびデバッグ』
- 406 ページの『リスト、メッセージ、およびコンパイラー情報』
- 406 ページの『最適化およびチューニング』
- 407 ページの『オブジェクト・コード制御』
- 408 ページの『移植性とマイグレーション』

- 409 ページの『コンパイラーのカスタマイズ』
- 408 ページの『推奨されないディレクティブ』

これらのカテゴリーの説明は、 81 ページの『機能カテゴリー別コンパイラー・オプションの要約』を参照してください。

言語エレメント制御

表 35. 言語エレメント制御プラグマ

プラグマ	説明
#pragma langlvl (C のみ)	ソース・コードおよびコンパイラー・オプションが固有の言語標準、または標準のサブセットまたはスーパーセットに準拠しているかを検査するかどうかを判別する。
433 ページの『#pragma mc_func』	マシン・インストラクション「inline」の短いシーケンスをプログラムのソース・コードの中に埋め込むことを可能にする。
436 ページの『#pragma options』	使用しているソース・プログラムでコンパイラー・オプションを指定する。

C++ テンプレート・プラグマ

表 36. C++ テンプレート・プラグマ

プラグマ	説明
415 ページの『#pragma define、#pragma instantiate (C++ のみ)』	テンプレート・クラスのインスタンスを明示的に生成するための代替メソッドを提供する。
417 ページの『#pragma do_not_instantiate (C++ のみ)』	指定したテンプレート宣言がインスタンス化されないようにする。
427 ページの『#pragma implementation (C++ のみ)』	**-qtempinc** コンパイラー・オプションと一緒に使用するために、ヘッダー・ファイル内のテンプレート宣言に対応するテンプレート定義を含むファイルの名前を指定する。

浮動小数点および整数制御

表 37. 浮動小数点および整数制御プラグマ

プラグマ	説明
#pragma chars	char 型のすべての変数を符号付きまたは符号なしのいずれかとして処理するかを判別する。
#pragma enum	列挙が占有するストレージの量を指定する。

エラー・チェックおよびデバッグ

表 38. エラー・チェックおよびデバッグ・プラグマ

プラグマ	説明
427 ページの『#pragma ibm snapshot』	ブレークポイントを設定できるロケーションを指定し、プログラム実行がそのロケーションに到達したときに検査できる変数のリストを定義する。
#pragma info	通知メッセージのグループを作成または抑制する。

リスト、メッセージ、およびコンパイラー情報

表 39. リスト、メッセージ、およびコンパイラー情報のプラグマ

プラグマ	説明
448 ページの『#pragma report (C++ のみ)』	診断メッセージの生成を管理する。

最適化およびチューニング

表 40. 最適化およびチューニングのプラグマ

プラグマ	説明
410 ページの『#pragma block_loop』	スコープ固有の ID を使用してブロックにマークを付ける。
451 ページの『#pragma STDC cx_limited_range』	中間計算のオーバーフローや重要度の喪失が発生しないような値だけを使用して複素数除算と絶対値を計算することを、コンパイラーに指示する。
415 ページの『#pragma disjoint』	その使用スコープ内で互いに別名ではない ID をリストする。
418 ページの『#pragma execution_frequency』	実行頻度が非常に高いか非常に低いと予期されるプログラム・ソース・コードにマークを付ける。
420 ページの『#pragma expected_value』	関数呼び出しで渡されるパラメーターが実行時に最も取る可能性が高い値を指定します。コンパイラーはこの情報を使用して、関数のクローン作成やインライン化など、特定の最適化を実行できます。
421 ページの『#pragma GCC visibility push、#pragma GCC visibility pop』	オブジェクト・ファイル内の外部リンケージ・エンティティーに visibility 属性を指定する。
424 ページの『#pragma ibm iterations』	選択したループについて、おおよそのループの平均の繰り返し回数を指定します。
425 ページの『#pragma ibm max_iterations』	選択したループについて、おおよそのループの最大の繰り返し回数を指定します。
426 ページの『#pragma ibm min_iterations』	選択したループについて、おおよそのループの最小の繰り返し回数を指定します。

表 40. 最適化およびチューニングのプラグマ (続き)

プラグマ	説明
#pragma isolated_call	パラメーターに暗黙指定されるもの以外の副次作用がない関数をソース・ファイルで指定する。
429 ページの『#pragma leaves』	指定した関数が、その関数の呼び出しの後の命令に戻らないことをコンパイラーに通知する。
430 ページの『#pragma loopid』	スコープ固有の ID を使用してブロックにマークを付ける。
#pragma nosimd	**-qsimd=auto** と一緒に使用すると、次のループの SIMD 命令の生成が使用不可になります。
#pragma novector	**-qhot=vector** と一緒に使用すると、次のループの自動ベクトル化が使用不可になります。
438 ページの『#pragma option_override』	コマンド行で指定された最適化オプションをオーバーライドするサブプログラム・レベルで最適化オプションを指定できるようにする。
446 ページの『#pragma reachable』	ある名前の関数の後のプログラム内の位置が、不明なロケーションから分岐先のターゲットとして指定される可能性があることをコンパイラーに通知する。
447 ページの『#pragma reg_killed_by』	**#pragma mc_func** が指定する関数によって変更される可能性のあるレジスターを指定する。
450 ページの『#pragma simd_level』	個別ループのベクトル命令のコンパイラー・コード生成を制御する。
452 ページの『#pragma stream_unroll』	最適化が使用可能な場合、`for` ループに含まれたストリームを複数のストリームに切断する。
#pragma unroll	パフォーマンスを改善するために、ループのアンロールを制御する。
454 ページの『#pragma unrollandfuse』	ネストされた `for` ループでアンロールおよびヒューズ操作を試行するようコンパイラーに命令する。

オブジェクト・コード制御

表 41. オブジェクト・コード制御プラグマ

プラグマ	説明
#pragma alloca (C のみ)	`alloca.h` ヘッダーを含まないソース・コードから呼び出されるとき、システム関数 `alloca` のインライン定義を提供する。
#pragma altivec_vrsave	VRSAVE レジスターを保守するための関数のプロローグとエピローグのコードを使用可能にする。
413 ページの『#pragma comment』	オブジェクト・モジュールにコメントを入れる。

表 41. オブジェクト・コード制御プラグマ (続き)

プラグマ	説明
422 ページの『#pragma hashome (C++ のみ)』	指定されたクラスに **#pragma ishome** で指定されるホーム・モジュールがあることをコンパイラーに通知する。
428 ページの『#pragma ishome (C++ のみ)』	指定されたクラスのホーム・モジュールが現行のコンパイル単位であることをコンパイラーに通知する。
431 ページの『#pragma map』	ID へのすべての参照を外部的に定義された別の ID へ変換する。
440 ページの『#pragma pack』	すべての集合体メンバーの位置合わせを、指定したバイト境界に設定する。
#pragma priority (C++ のみ)	静的オブジェクトの初期化の優先順位を指定する。
447 ページの『#pragma reg_killed_by』	**#pragma mc_func** が指定する関数によって変更される可能性のあるレジスターを指定する。
#pragma strings	ストリング・リテラルのストレージ・タイプを指定する。
456 ページの『#pragma weak』	リンク時に複数定義されたシンボルが見つかった場合、またはシンボルの定義が見つからなかった場合、リンカーがエラー・メッセージを出さないようにする。

移植性とマイグレーション

表 42. 移植性とマイグレーション・プラグマ

プラグマ	説明
#pragma align	ストレージ内でデータ・オブジェクトの位置合わせを指定する。これによって、誤って配置されたデータが原因で起こるパフォーマンス上の問題を回避できます。

推奨されないディレクティブ

以下の表にリストする SMP ディレクティブは非推奨であり、将来のリリースで除去される可能性があります。同じ動作を得るには、対応する OpenMP ディレクティブを使用してください。

表 43. 推奨されない SMP ディレクティブ

SMP ディレクティブ名	OpenMP ディレクティブ/文節名
#pragma ibm schedule	**schedule** 節を伴う 470 ページの『#pragma omp parallel for』 プラグマ。

非推奨の SMP ディレクティブは、対応する OpenMP ディレクティブで置き換えることができます。例を以下に示します。

```
#pragma omp parallel for schedule(static, 5)
for (i=0; i<N; i++)
{
  // ...
}
```

コンパイラーのカスタマイズ

表 44. コンパイラーのカスタマイズ・プラグマ

プラグマ	説明
#pragma complexgcc	複素数データ型に GCC パラメーター受け渡し規則を使用するかどうかを指定します (**-qfloat=complexgcc** を使用可能にすることと同等です)。

個々のプラグマの説明

この節には XL C/C++ で使用可能な個々のプラグマの説明があります。

プラグマにはそれぞれ、以下の情報が与えられています。

カテゴリー
ここには、そのプラグマが属する機能カテゴリーがリストされています。

目的 このセクションでは、プラグマの効果とその使用目的が簡単に説明されています。

構文 このセクションには、プラグマの構文が記載されています。便宜上、ディレクティブの **#pragma *name*** 形式がどのケースにも使用されています。ただし、代替の C99-style `_Pragma` 演算子構文もまったく問題なく使用することができます。詳しくは、403 ページの『プラグマ・ディレクティブ構文』を参照してください。

パラメーター
このセクションでは、適用可能である場合に、プラグマで使用可能なサブオプションが説明されています。

使用法 このセクションでは、プラグマを使用するときに注意すべき規則および使用上の考慮事項が説明されています。プラグマの適用可能性における制限、プラグマの有効な配置などが説明されています。

例 プラグマ・ディレクティブの使用例がこのセクションの適切な箇所で提供されています。

#pragma align

『109 ページの『-qalign』』を参照してください。

#pragma alloca (C のみ)

『111 ページの『-qalloca、-ma (C のみ)』』を参照してください。

#pragma altivec_vr_save

『391 ページの『-qvrsave』』を参照してください。

#pragma block_loop

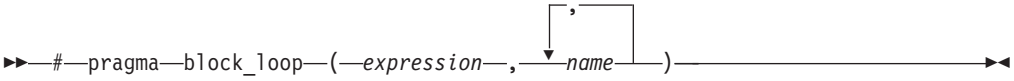
カテゴリー

最適化およびチューニング

目的

スコープ固有の ID を使用してブロックにマークを付ける。

構文



パラメーター

expression
繰り返しグループのサイズを表す整数式です。

name
スコープ単位内で固有の ID です。*name* を指定しないと、最初の `for` ループ、または **#pragma block_loop** ディレクティブの後のループでブロッキングが発生します。

使用法

ループ・ブロッキングが発生するには、**#pragma block_loop** ディレクティブが `for` ループに先行している必要があります。

ブロッキング・ループに **#pragma unroll**、**#pragma unrollandfuse**、または **#pragma stream_unroll** を指定すると、ブロッキング・ループが実際に作成された場合、ブロッキング・ループがそれぞれアンロール、アンロールおよびヒューズ、またはストリーム・アンロールされます。それ以外の場合、このディレクティブは何の効果もありません。

ブロックされたループに **#pragma unrollandfuse**、**#pragma unroll**、または **#pragma stream_unroll** ディレクティブを指定すると、ディレクティブはブロッキング・ループの作成後に、ブロックされたループに適用されます。ブロッキング・ループが作成されないと、対応する **#pragma block_loop** ディレクティブが指定されていないかのように、このディレクティブがブロッキングを意図したループに適用されます。

同じ `for` ループに対して **#pragma block_loop** を複数回指定したり、このディレクティブを **#pragma nounroll**、**#pragma unroll**、**#pragma nounrollandfuse**、**#pragma unrollandfuse**、または **#pragma stream_unroll** ディレクティブと結合しないでください。また、単一のブロック・ループ・ディレクティブに複数の **#pragma unroll** ディレクティブを適用しないでください。

すべての **#pragma block_loop** ディレクティブの処理は常に、いずれかの unroll ディレクティブによって示されるアンロールの実行前に完了します。

例

以下の 2 つの例では、ループ・タイルの **#pragma block_loop** および **#pragma loop_id** の使用方法を示しています。

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
  for (i=0; i < n; i++)
  {
#pragma loopid(myfirstloop)
    for (j=0; j < m; j++)
    {
#pragma loopid(mysecondloop)
      for (k=0; k < m; k++)
      {
         ...
      }
    }
  }
```

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
       for (i=0; i < n; n++)
       {
#pragma loopid(myfirstloop)
              for (j=0; j < m; j++)
              {
#pragma loopid(mysecondloop)
                     for (k=0; k < m; k++)
                     {
                            ...
                     }
              }
       }
```

以下の例では、ループ交換の **#pragma block_loop** および **#pragma loop_id** の使用方法を示しています。

```
       for (i=0; i < n; i++)
       {
              for (j=0; j < n; j++)
              {
#pragma block_loop(1,myloop1)
                     for (k=0; k < m; k++)
                     {
#pragma loopid(myloop1)
                            for (l=0; l < m; l++)
                            {
                                   ...
                            }
                     }
              }
       }
```

以下の例では、マルチ・レベルのメモリー階層用のループ・タイルの **#pragma block_loop** および **#pragma loop_id** の使用方法を示しています。

```
#pragma block_loop(l3factor, first_level_blocking)
  for (i=0; i < n; i++)
  {
#pragma loopid(first_level_blocking)
#pragma block_loop(l2factor, inner_space)
    for (j=0; j < n; j++)
    {
```

```
#pragma loopid(inner_space)
      for (k=0; k < m; k++)
      {
        for (l=0; l < m; l++)
        {
          ...
        }
      }
    }
  }
```

以下の例では、**#pragma unrollandfuse** および **#pragma block_loop** を使用して、ブロッキング・ループをアンロールおよびヒューズしています。

```
#pragma unrollandfuse
#pragma block_loop(10)
   for (i = 0; i < N; ++i) {
   }
```

この場合、ブロック・ループ・ディレクティブが無視されると、アンロール・ディレクティブは何の効果も及ぼしません。

以下の例では、ブロックされたループをアンロールするための **#pragma unroll** および **#pragma block_loop** の使用法を示しています。

```
#pragma block_loop(10)
#pragma unroll(2)
  for (i = 0; i < N; ++i) {
  }
```

この場合、ブロック・ループ・ディレクティブが無視されても、非ブロック化されたループはアンロールされます。ブロッキングが発生した場合は、アンロール・ディレクティブはブロックされたループに適用されます。

以下の例は、ディレクティブの無効な使用方法を示しています。最初の例は、未定義のループ ID で使用された **#pragma block_loop** を示しています。

```
#pragma block_loop(50, myloop)
  for (i=0; i < n; i++)
  {
  }
```

myloop は、ネスト内になく定義されない場合があるため、参照できません。

以下の例では、myloop は、同じループ・ネスト内にないため参照できません。

```
   for (i=0; i < n; i++)
   {
#pragma loopid(myLoop)
     for (j=0; j < i; j++)
     {
       ...
     }
   }
#pragma block_loop(myLoop)
  for (i=0; i < n; i++)
  {
    ...
  }
```

以下の例は無効です。これは、アンロール・ディレクティブがお互いに矛盾するためです。

```
#pragma unrollandfuse(5)
#pragma unroll(2)
  #pragma block_loop(10)
            for (i = 0; i < N; ++i) {
        }
#pragma block_loop(10)
#pragma unroll(5)
#pragma unroll(10)
  for (i = 0; i < N; ++i) {
  }
```

関連情報

- 430 ページの『#pragma loopid』
- 381 ページの『-qunroll』
- 454 ページの『#pragma unrollandfuse』
- 452 ページの『#pragma stream_unroll』

#pragma chars

『 128 ページの『-qchars』』を参照してください。

#pragma comment

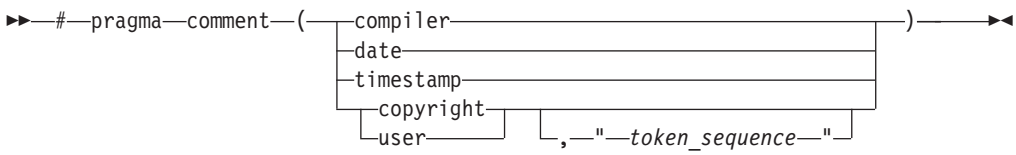
カテゴリー

オブジェクト・コード制御

目的

オブジェクト・モジュールにコメントを入れる。

構文



パラメーター

compiler
コンパイラーの名前とバージョンを生成されたオブジェクト・モジュールの最後に追加します。

date
コンパイルの日付と時刻が生成されたオブジェクト・モジュールの最後に追加されます。

timestamp
ソースの最後の変更の日付と時刻を生成されたオブジェクト・モジュールの最後に追加します。

copyright
token_sequence によって指定されたテキストがあれば、それを生成されたオブジ

エクト・モジュール内に入れます。*token_sequence* は生成された実行可能ファイルに組み込まれ、プログラムの実行時にはメモリーにロードされます。

user
token_sequence によって指定されたテキストがあれば、それを生成されたオブジェクト・モジュール内に入れます。*token_sequence* は生成された実行可能ファイルに組み込まれますが、プログラムの実行時にメモリーにはロードされません。

token_sequence
このフィールドの文字は、指定する場合に二重引用符 (") で囲まなければなりません。*token_sequence* に指定されたストリング・リテラルが 32 767 バイトを超えると、通知メッセージが出され、プラグマが無視されます。

使用法

変換単位には複数の **comment** ディレクティブが現れることがあり、また **comment** ディレクティブの各タイプは複数回現れることがあります。ただし、1 回しか現れることができない **copyright** は除きます。

オペレーティング・システムの **strings** コマンドを使用してオブジェクト・ファイル・コメントを表示できます。

例

以下のプログラム・コード `tt.c` が存在すると仮定します。

```
#pragma comment(date)
#pragma comment(compiler)
#pragma comment(timestamp)
#pragma comment(copyright,"My copyright")
int main() { return 0; }
```

以下のコマンドを発行すると、

```
xlc -c tt.c
strings -a tt.o
```

`tt.o` に組み込まれたコメント情報が、`tt.o`で検出された他のストリングと共に表示されます。

```
@.text
.data
@.bss
.comment
Thu Sep 24 16:44:25 EDT 2014IBM XL C for Linux ---- Version 13.1.0.0
Thu Sep 24 16:44:09 EDT 2014
main
My copyright
.file
tt.c
.text
.data
.bss
.main
_$STATIC
_$STATIC
main
main
Thu Sep 24 16:44:25 2014
IBM XL C for Linux, Version 13.1.0.0 ---
```

#pragma complexgcc

『 135 ページの『-qcomplexgccincl』』を参照してください。

#pragma define、#pragma instantiate (C++ のみ)

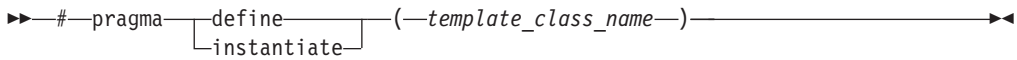
カテゴリー

テンプレート制御

目的

テンプレート・クラスのインスタンスを明示的に生成するための代替メソッドを提供する。

構文



パラメーター

template_class_name
インスタンスが生成されるテンプレート・クラスの名前。

使用法

このプラグマは、C++ の明示的インスタンス生成定義と同等の機能を提供します。これは、以前のリリースとの互換性のためにのみ提供されています。新しいアプリケーションでは、C++ の明示的インスタンス生成定義を使用してください。

このプラグマは、明示的インスタンス生成定義を置くことができる場所であれば、どこにでも置くことができます。

例

以下のディレクティブ

```
#pragma define(Array<char>)
```

は、以下の明示的なインスタンス生成と同等です。

```
template class Array<char>;
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『明示的インスタンス生成』
- 417 ページの『#pragma do_not_instantiate (C++ のみ)』

#pragma disjoint

カテゴリー

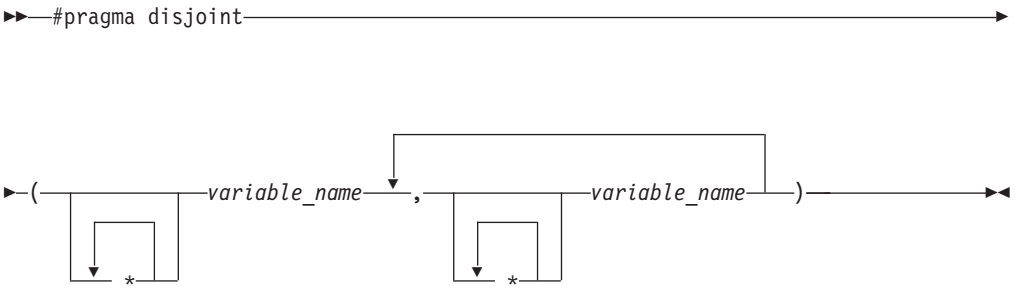
最適化およびチューニング

目的

その使用スコープ内で互いに別名ではない ID をリストする。

プラグマでリストされているどの ID も同じ物理ストレージを共有していないことをコンパイラーに通知することによって、プラグマは最適化のための機会をより多く提供します。

構文



パラメーター

variable_name
変数の名前。これは、次のいずれも参照できません。

- 構造体、クラス、または共用体のメンバー
- 構造体、共用体または列挙タグ
- 列挙定数
- typedef 名
- ラベル

使用法

#pragma disjoint ディレクティブは、プラグマでリストされたどの ID も物理ストレージを共有していないことを表明します。いずれかの ID が実際に物理ストレージを共有している 場合は、プラグマは間違った結果を出す可能性があります。

プラグマは、宣言が許可されていればソース・プログラム内のどこにでも指定することができます。ディレクティブの中の ID は、プログラム中にプラグマが現れる点で可視でなくてはなりません。

ID は、プラグマで使用する前に宣言する必要があります。ご使用のプログラムでは、ID リスト中のポインターを間接参照しないでください。また、そのポインターをディレクティブ内で指定する前に関数引数として使用しないでください。

このプラグマは、**-qignprag** コンパイラー・オプションによって使用不可にできます。

例

以下の例は、**#pragma disjoint** の使用方法を示しています。

```
int a, b, *ptr_a, *ptr_b;

#pragma disjoint(*ptr_a, b)   /* *ptr_a never points to b */
#pragma disjoint(*ptr_b, a)   /* *ptr_b never points to a */
one_function()
{
     b = 6;
     *ptr_a = 7;   /* Assignment will not change the value of b  */

   another_function(b);    /* Argument "b" has the value 6  */
}
```

外部ポインター ptr_a は、外部変数 b とストレージを共有することもその外部変数を指すこともありません。その結果、ptr_a が指すオブジェクトに 7 を代入しても、b の値は変わりません。同様に、外部ポインター ptr_b は、外部変数 a とストレージを共有することもそれを指すこともありません。コンパイラーは another_function の引数が 6 の値を持っていると想定することができるため、メモリーから変数を再ロードしません。

#pragma do_not_instantiate (C++ のみ)

カテゴリー

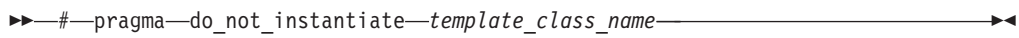
テンプレート制御

目的

指定したテンプレート宣言がインスタンス化されないようにする。

このプラグマを使用して、定義が提供されているテンプレートの暗黙のインスタンス生成を抑制することができます。

構文



パラメーター

template_class_name
インスタンスを生成されるべきではないテンプレート・クラスの名前。

使用法

テンプレートのインスタンス生成を手動で処理しており (つまり、**-qnotempinc** と **-qnotemplateregistry** が指定されている)、指定されたテンプレートのインスタンス生成が既に別のコンパイル単位に存在している場合は、**#pragma do_not_instantiate** を使用すると、リンク・ステップ中に複数のシンボル定義を取得しないことが保証されます。

C++11

クラス・テンプレートの特殊化における **#pragma do_not_instantiate** は、テンプレートの明示的インスタンス生成宣言として扱われます。このプラグマは、C++11 標準に導入された、明示的インスタンス生成宣言の機能のサブセットを提供します。

これは互換性の目的でのみ提供されているもので、推奨されていません。新しいアプリケーションでは、明示的インスタンス生成宣言を代わりに使用してください。

C++11

-qtmplinst オプションを使用して、複数のコンパイル単位のテンプレート宣言を暗黙的にインスタンス化するのを抑制することもできます。『 374 ページの『-qtmplinst (C++ のみ)』』を参照してください。

例

以下はプラグマの使用法を示しています。

```
#pragma do_not_instantiate Stack < int >
```

関連情報

- 415 ページの『#pragma define、#pragma instantiate (C++ のみ)』
- 374 ページの『-qtmplinst (C++ のみ)』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『明示的インスタンス生成』
- 365 ページの『-qtempinc (C++ のみ)』
- 368 ページの『-qtemplateregistry (C++ のみ)』

#pragma enum

『 152 ページの『-qenum』』を参照してください。

#pragma execution_frequency

カテゴリー

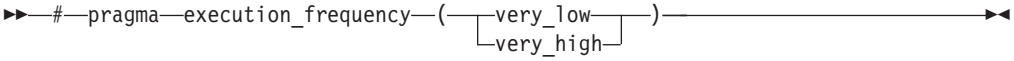
最適化およびチューニング

目的

実行頻度が非常に高いか非常に低いと予期されるプログラム・ソース・コードにマークを付ける。

最適化が使用可能な場合、プラグマは、最適化プログラムに対するヒントとして使用されます。

構文



パラメーター

very_low
実行頻度が非常に低いと予期されるソース・コードにマークを付けます。

very_high
実行頻度が非常に高いと予期されるソース・コードにマークを付けます。

使用法

このプラグマを最適化オプションと共に使用します。最適化が使用可能でない場合は、プラグマは何の効果も及ぼしません。

プラグマは、ブロック・スコープ内に配置する必要があり、次の分岐ポイントで実行されます。

例

以下の例では、プラグマは、実行頻度の低いコードにマークを付けるために `if` 文のブロックで使用されます。

```
int *array = (int *) malloc(10000);

if (array == NULL) {
    /* Block A */
    #pragma execution_frequency(very_low)
    error();
}
```

次の例では、コード・ブロック `Block B` に実行頻度が低いというマークが付けられ、分岐の際に `Block C` が選択される可能性が高くなります。

```
if (Foo > 0) {
    #pragma execution_frequency(very_low)
    /* Block B */
    doSomething();
} else {
    /* Block C */
    doAnotherThing();
}
```

以下の例では、プラグマは、実行頻度の高いコードにマークを付けるために `switch` 文のブロックで使用されます。

```
while (counter > 0) {
    #pragma execution_frequency(very_high)
    doSomething();
} /* This loop is very likely to be executed.    */

switch (a) {
    case 1:
        doOneThing();
        break;
    case 2:
        #pragma execution_frequency(very_high)
        doTwoThings();
        break;
    default:
        doNothing();
}    /* The second case is frequently chosen.   */
```

以下の例は、プラグマをブロック・スコープでどのように適用すべきかを示しており、次の分岐に影響を与えています。

```
int a;
#pragma execution_frequency(very_low)
int b;

int foo(boolean boo) {
    #pragma execution_frequency(very_low)
    char c;
```

```
    if (boo) {
        /* Block A */
        doSomething();
        {
            /* Block C */
            doSomethingAgain();
            #pragma execution_frequency(very_low)
            doAnotherThing();
        }
    } else {
        /* Block B */
        doNothing();
    }

    return 0;
}

#pragma execution_frequency(very_low)
```

#pragma expected_value

カテゴリー

最適化およびチューニング

目的

関数呼び出しで渡されるパラメーターが実行時に最も取る可能性が高い値を指定します。コンパイラーはこの情報を使用して、関数のクローン作成やインライン化など、特定の最適化を実行できます。

構文

パラメーター

argument
予期された値を提供するパラメーターの名前。パラメーターは、単純な組み込み整数型、ブール型、文字型、または浮動小数点型でなければなりません。

value
パラメーターが、予期する値を表す定数リテラルを実行時に取る場合が多くあります。*value* は、コンパイル時の定数式であれば、式であることがあります。

使用法

ディレクティブは、関数定義の本体の内部で、最初のステートメント (宣言ステートメントなど) の前に現れなければなりません。ネストされた関数内ではサポートされていません。

型の予期された値を指定するときにパラメーター変数の宣言された型の値とは異なる値を指定する場合、その値は許可されている場合に限り暗黙的に変換されます。それ以外の場合は、警告が発行されます。

予期される値が提供される各パラメーターには、ディレクティブは 1 つのみという制限があります。予期される値が提供されないパラメーターは、ディレクティブを要求しません。

例

以下の例で、コンパイラーは、パラメーター `a` および `b` が最も取りうる値はそれぞれ 1 および 0 であると指示されます。

```
int func(int a,int b)
{
#pragma expected_value(a,1)
#pragma expected_value(b,0)
...
...
}
```

関連情報

- 418 ページの『#pragma execution_frequency』

#pragma GCC visibility push、#pragma GCC visibility pop

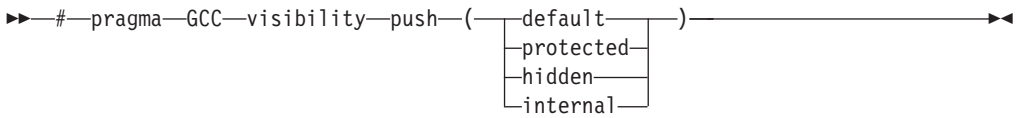
カテゴリー

最適化およびチューニング

目的

オブジェクト・ファイル内の外部リンケージ・エンティティーに visibility 属性を指定する。

構文



パラメーター

default
影響を受ける外部リンケージ・エンティティーが default の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされ、優先使用できます。

protected
影響を受ける外部リンケージ・エンティティーが protected の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされますが、優先使用できません。

hidden
影響を受ける外部リンケージ・エンティティーが hidden の visibility 属性を持

つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされませんが、それらのアドレスをポインター経由で間接的に参照できます。

internal
影響を受ける外部リンケージ・エンティティーが internal の visibility 属性を持つことを示します。これらのエンティティーは共有ライブラリーにエクスポートされず、他のモジュールではそれらのアドレスを使用できません。

制約事項: このリリースでは、hidden および internal の visibility 属性は同じです。これらいずれかの visibility 属性を使用して指定されたエンティティーのアドレスは、ポインター経由で間接的に参照できます。

使用法

ソース・プログラム全体で `#pragma GCC visibility push` および `#pragma GCC visibility pop` コンパイラー・ディレクティブのペアを使用することにより、エンティティーに対して選択的に visibility 属性を設定できます。`#pragma GCC visibility pop` ディレクティブを対応する `#pragma GCC visibility push` ディレクティブなしで指定した場合、コンパイラーによって警告メッセージが出されます。エンティティーの visibility 属性は、1 つのモジュールで定義されているエンティティーを他のモジュールで参照または使用できるかどうか、およびその方法を記述します。Visibility 属性は外部リンケージを持つエンティティーにのみ影響します。他のエンティティーの可視性を増加させることはできません。エンティティーの優先使用は、リンク時にエンティティー定義が解決されるにもかかわらず、ランタイム時に別のエンティティー定義に置き換えられる場合に発生します。

関連情報

- 389 ページの『-qvisibility』
- 277 ページの『-qmkshrobj』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『visibility 属性の使用 (IBM 拡張)』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『外部リンケージ』、 『visibility 変数属性 (IBM 拡張)』、 『visibility 関数属性 (IBM 拡張)』、 『visibility 型属性 (C++ のみ) (IBM 拡張)』、および『visibility 名前空間属性 (C++ のみ) (IBM 拡張)』

#pragma hashome (C++ のみ)

カテゴリー

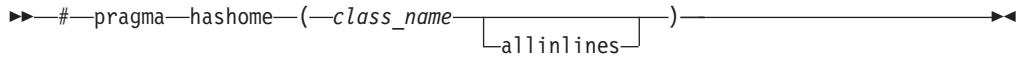
オブジェクト・コード制御

目的

指定されたクラスに **#pragma ishome** で指定されるホーム・モジュールがあることをコンパイラーに通知する。

このクラスの仮想関数テーブルは、特定のインライン関数と共に、静的として生成されません。代わりにこれらは、**#pragma ishome** が指定されるクラスのコンパイル単位内で外部として参照されます。

構文



パラメーター

class_name
外部参照されるクラスの名前。*class_name* は、クラスであり、定義する必要があります。

allinlines
class_name 内から、すべてのインライン関数を外部参照する必要があることを指定します。

使用法

一致する **#pragma hashome** を持たない **#pragma ishome** がある場合は、警告が出されます。

例

以下の例では、コード・サンプルをコンパイルすると仮想関数テーブルと、コンパイル単位 `a.o` のみの `S::foo()` の定義が生成されます (`b.o` 用はありません)。これによりアプリケーションに対して生成されるコードの量が削減されます。

```
// a.h
struct S
{
   virtual void foo() {}

   virtual void bar();
};



// a.C
#pragma ishome(S)
#pragma hashome (S)

#include "a.h"

int main()
{
   S s;
   s.foo();
   s.bar();
}



// b.C
#pragma hashome(S)
#include "a.h"

void S::bar() {}
```

関連情報

- 428 ページの『#pragma ishome (C++ のみ)』

#pragma ibm iterations

カテゴリー

最適化およびチューニング

目的

iterations プラグマは、選択したループについて、おおよそのループの平均反復回数を指定します。

構文

パラメーター

iteration_count
正の整数定数式を使用して、おおよそのループの繰り返し回数を指定します。

使用法

コンパイラーは、ループ最適化に *iteration_count* の情報を使用します。複数の #pragma ibm iterations(*iteration_count*) を指定できます。

#pragma ibm iterations で指定された *iteration_count* は、#pragma ibm min_iterations で指定された *iteration_count* よりも小さくすることはできません。さらに、#pragma ibm max_iterations で指定された *iteration_count* よりも大きくすることもできません。それ以外の場合、整合しない値は無視され、メッセージが出されます。

例

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

関連資料:

425 ページの『#pragma ibm max_iterations』

426 ページの『#pragma ibm min_iterations』

#pragma ibm max_iterations

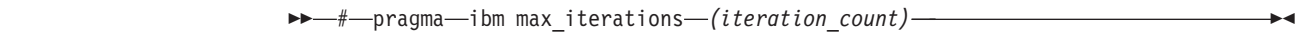
カテゴリー

最適化およびチューニング

目的

max_iterations プラグマは、選択したループについて、おおよそのループの最大繰り返し回数を指定します。

構文



パラメーター

iteration_count
正の整数定数式を使用して、おおよその最大ループ繰り返し回数を指定します。

使用法

コンパイラーは、ループ最適化に *iteration_count* の情報を使用します。 #pragma ibm max_iterations(*iteration_count*) は 1 回しか指定できません。 #pragma ibm max_iterations(*iteration_count*) を 2 回以上指定した場合は、最初に指定したプラグマが受け入れられ、それ以降のプラグマは無視され、メッセージが出されます。

#pragma ibm max_iterations で指定された *iteration_count* は、#pragma ibm iterations または #pragma ibm min_iterations で指定された *iteration_count* よりも小さくすることはできません。それ以外の場合、整合しない値は無視され、メッセージが出されます。

例

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

関連資料:

424 ページの『#pragma ibm iterations』

426 ページの『#pragma ibm min_iterations』

#pragma ibm min_iterations

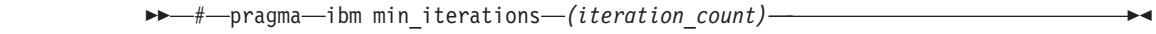
カテゴリー

最適化およびチューニング

目的

min_iterations プラグマは、選択したループについて、おおよそのループの最小繰り返し回数を指定します。

構文



パラメーター

iteration_count
正の整数定数式を使用して、おおよその最小ループ繰り返し回数を指定します。

使用法

コンパイラーは、ループ最適化に *iteration_count* の情報を使用します。 #pragma ibm min_iterations(*iteration_count*) は 1 回しか指定できません。 #pragma ibm min_iterations(*iteration_count*) を 2 回以上指定した場合は、最初に指定したプラグマが受け入れられ、それ以降のプラグマは無視され、メッセージが出されます。

#pragma ibm min_iterations で指定された *iteration_count* は、#pragma ibm iterations または #pragma ibm max_iterations で指定された *iteration_count* よりも大きくすることはできません。それ以外の場合、整合しない値は無視され、メッセージが出されます。

例

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

関連資料:

424 ページの『#pragma ibm iterations』

425 ページの『#pragma ibm max_iterations』

#pragma ibm snapshot

カテゴリー

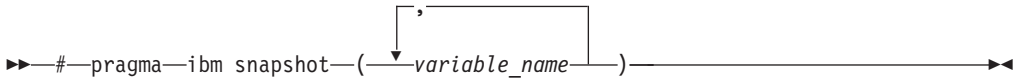
エラー・チェックおよびデバッグ

目的

ブレークポイントを設定できるロケーションを指定し、プログラム実行がそのロケーションに到達したときに検査できる変数のリストを定義する。

このプラグマを使用して、コンパイラーによって作成された最適化コードのデバッグを可能にすることができます。

構文



パラメーター

variable_name
変数名。構造体、クラス、または共用体のメンバーを参照できません。

使用法

デバッグ・セッションの間、ディレクティブが現れる行にブレークポイントを配置して、名前付き変数の値を表示することができます。最適化および **-g** オプションを指定してコンパイルすると、名前付き変数はデバッガーから可視となることが保証されます。

このプラグマは、高い最適化レベルでは静的ストレージ・クラスを持つ変数のコンテンツを一貫して保存しません。ディレクティブに指定された変数は、デバッガーで監視している間は読み取り専用と考え、変更しないようにしてください。デバッガーでこれらの変数を変更した場合、予測不能の振る舞いが生じる可能性があります。

例

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

関連情報

- 173 ページの『-g』
- 279 ページの『-O、-qoptimize』

#pragma implementation (C++ のみ)

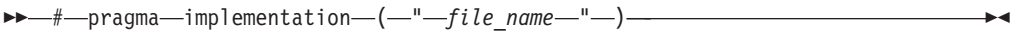
カテゴリー

テンプレート制御

目的

-qtempinc コンパイラー・オプションと一緒に使用するために、ヘッダー・ファイル内のテンプレート宣言に対応するテンプレート定義を含むファイルの名前を指定する。

構文



パラメーター

file_name
ヘッダー・ファイルで宣言されたテンプレート・クラスのメンバーの定義を含むファイルの名前。

使用法

テンプレート・インプリメンテーション・ファイルがテンプレート宣言を含むヘッダー・ファイルと同じ名前、および .c 拡張子を持っている場合は、このプラグマは、通常要求されません。テンプレート・インプリメンテーション・ファイルがこのファイル命名規則に準拠していない場合は、プラグマのみを使用してください。テンプレート・インプリメンテーション・ファイルの使用について詳しくは、『C++ テンプレートの使用』を参照してください。

#pragma implementation は、**-qtempinc** オプションが有効な場合にのみ有効です。それ以外の場合は、プラグマは意味を持たないので無視されます。

プラグマはテンプレート宣言を含むヘッダー・ファイルか、ヘッダー・ファイルを含むソース・ファイルに現れることができます。宣言が許可されていればどこにでも現れることができます。

関連情報

- 365 ページの『-qtempinc (C++ のみ)』
- 『C++ プログラムでのテンプレートの使用』

#pragma info

『 193 ページの『-qinfo』』を参照してください。

#pragma ishome (C++ のみ)

カテゴリー

オブジェクト・コード制御

目的

指定されたクラスのホーム・モジュールが現行のコンパイル単位であることをコンパイラーに通知する。

ホーム・モジュールは、仮想関数テーブルなどの項目が保管されている場所です。項目は、コンパイル単位の外側から参照されると、そのホームの外部には生成され

ません。これによりアプリケーションに対して生成されるコードの量を削減することができます。

構文

パラメーター

class_name
ホームが現行のコンパイル単位になるクラスの名前です。

使用法

一致する **#pragma hashome** を持たない **#pragma ishome** がある場合は、警告が出されます。

例

『 422 ページの『#pragma hashome (C++ のみ)』』を参照してください。

関連情報

- 422 ページの『#pragma hashome (C++ のみ)』

#pragma isolated_call

『 217 ページの『-qisolated_call』』を参照してください。

#pragma langlvl (C のみ)

『 227 ページの『-qlanglvl』』を参照してください。

#pragma leaves

カテゴリー

最適化およびチューニング

目的

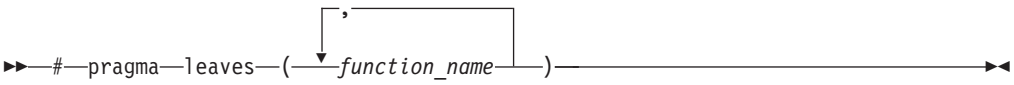
指定した関数が、その関数の呼び出しの後の命令に戻らないことをコンパイラーに通知する。

関数の後のコードを無視することができるとコンパイラーに通知することによって、ディレクティブは最適化のための機会を追加することができます。

一般的に、このプラグマはカスタムのエラー処理関数に使用されますが、この場合特定のエラーが発生すると、プログラムを終了することができます。

注: setjmp.h ヘッダーが組み込まれると、コンパイラーは `longjmp` 関数ファミリー (`longjmp`、`_longjmp`、`siglongjmp`、および `_siglongjmp`) への呼び出しの **#pragma leaves** ディレクティブを自動的に挿入します。

構文



パラメーター

function_name
関数の呼び出しの後の命令に戻らない関数の名前。

デフォルト

適用されません。

例

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
 if (value == ERROR_VALUE)
 {
  handle_error_and_quit(value);
  TryAgain(); // optimizer ignores this because
    // never returns to execute it
 }
}
```

関連情報

- 446 ページの『#pragma reachable』.

#pragma loopid

カテゴリー

最適化およびチューニング

目的

スコープ固有の ID を使用してブロックにマークを付ける。

構文

パラメーター

name
スコープ単位内で固有の ID です。

使用法

#pragma loopid ディレクティブは、**#pragma block_loop** ディレクティブまたは for ループのすぐ前になければなりません。指定された名前は、**#pragma**

block_loop がループでの変換を制御するために使用することができます。また、これは **-qreport** コンパイラー・オプションの使用を通じてループ変換での情報を提供するためにも使用することができます。

ある特定のループに対して **#pragma loopid** を複数回指定しないでください。

例

#pragma loopid の使用法の例については、410 ページの『#pragma block_loop』を参照してください。

関連情報

- 381 ページの『-qunroll』
- 410 ページの『#pragma block_loop』
- 454 ページの『#pragma unrollandfuse』

#pragma map

カテゴリー

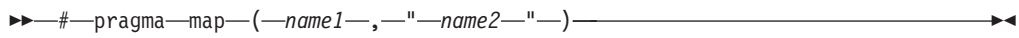
オブジェクト・コード制御

目的

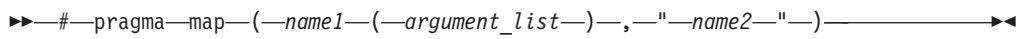
ID へのすべての参照を外部的に定義された別の ID へ変換する。

構文

#pragma map 構文 - C



#pragma map 構文 - C++



パラメーター

name1

ソース・コードで使用される名前。 C *name1* は、外部リンケージを持つデータ・オブジェクトまたは関数を表すことができます。 C++ *name1* は、外部リンケージを持つデータ・オブジェクト、非多重定義関数または多重定義関数、多重定義演算子のいずれかを表すことができます。マップされる名前は、グローバル名前空間にない場合、完全修飾する必要があります。

name1 は、自身が参照されるコンパイル単位内で宣言し、他のコンパイル単位で定義しないでください。*name1* を、別の **#pragma map** ディレクティブまたは任意のアセンブリー・ラベル宣言をプログラム内の任意の場所で使用しないでください。

C++ *argument_list*

name1 によって指定された多重定義関数または演算子関数の引数のリスト。

name1 によって多重定義関数が指定されると、この関数は括弧で囲み、引数リスト (存在する場合) を組み込む必要があります。*name1* が非多重定義関数を指定する場合は、*name1* のみが要求され、括弧および引数リストはオプションです。

name2

オブジェクト・コードに現れる名前。 C *name2* は、外部リンケージを持つデータ・オブジェクトまたは関数を表すことができます。

C++ *name2* は、外部リンケージを持つデータ・オブジェクト、非多重定義関数または多重定義関数、多重定義演算子のいずれかを表すことができます。*name2* はマングル名を使用して指定してください。C++ マングル名を取得するには、**-c** コンパイラー・オプションを使用してオブジェクト・ファイルのみにソースをコンパイルし、その結果生じるオブジェクト・ファイルで **nm** オペレーティング・システム・コマンドを使用します。 (また、名前マングリングを回避するための、宣言に対する `extern "C"` リンケージ指定子の使用について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『名前マングリング』も参照してください。)

名前が 65535 バイトを超えると、通知メッセージが出され、プラグマが無視されます。

name2 は、*name1* が参照されるのと同じコンパイル単位で宣言できる場合とできない場合がありますが、同じコンパイル単位で定義しないでください。また、*name1* を参照するコンパイル単位の任意の場所で *name2* を参照しないでください。*name2* を、別の **#pragma map** ディレクティブまたは任意のアセンブリー・ラベル宣言を同じコンパイル単位で使用しないでください。

使用法

#pragma map ディレクティブは、プログラムのどこに指定してもかまいません。関数を実際にマップするには、マップ・ターゲット関数 (*name2*) が (別のコンパイル単位からの) リンク時に使用可能な定義を持つ必要があり、マップ・ソース関数 (*name1*) をご使用のプログラムで呼び出す必要があります。

コンパイラー組み込み関数と一緒に **#pragma map** を使用することはできません。

例

以下は、(C++ のマップ名にマングル名を使用して) 関数名をマップするために使用する **#pragma map** の例です。

```
/* Compilation unit 1: */

#include <stdio.h>

void foo();
extern void bar(); /* optional */

#if __cplusplus
#pragma map (foo, "_Z3barv")
#else
#pragma map (foo, "bar")
#endif
int main()
{
foo();
```

```
}

/* Compilation unit 2: */

#include <stdio.h>

void bar()
{
printf("Hello from foo bar!¥n");
}
```

以下のように、コンパイル単位 1 の foo への呼び出しは bar への呼び出しに解決されます。

```
Hello from foo bar!
```

C++ 以下は、(マップ名のマングル名の使用を避けるために C リンケージを使用して) 多重定義関数名をマップするために使用する **#pragma map** の例です。

```
// Compilation unit 1:

#include <iostream>
#include <string>

using namespace std;

void foo();
void foo(const string&);
extern "C" void bar(const string&); // optional

#pragma map (foo(const string&), "bar")

int main()
{
foo("Have a nice day!");
}

// Compilation unit 2:

#include <iostream>
#include <string>

using namespace std;

extern "C" void bar(const string& s)
{
cout << "Hello from foo bar!" << endl;
cout << s << endl;
}
```

以下のように、コンパイル単位 1 の foo(const string&) への呼び出しは bar(const string&) への呼び出しに解決されます。

```
Hello from foo bar!
Have a nice day!
```

関連情報

- (「*XL C/C++ ランゲージ・リファレンス*」の『アセンブリー・ラベル』

#pragma mc_func

カテゴリー

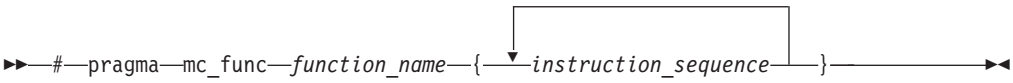
言語エレメント制御

目的

マシン・インストラクション「inline」の短いシーケンスをプログラムのソース・コードの中に埋め込むことを可能にする。

このプラグマは、通常のリンケージ・コードではなく、決まった所に指定された命令を生成するようコンパイラーに命令します。このプラグマを使用すると、アセンブラーでコーディングされた外部関数の呼び出しに関連したパフォーマンス上のペナルティーが避けられます。このプラグマは機能面において、このコンパイラーや他のコンパイラーでサポートされるインライン `asm` 文に似ています。詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の 『インライン・アセンブリー・ステートメント』を参照してください。

構文



パラメーター

function_name
: マシン・インストラクションを含む、以前に定義された関数の名前。関数が以前に定義されていない場合、コンパイラーはこのプラグマを関数定義として扱います。

instruction_sequence
: ゼロ以上の 16 進数字のシーケンスを含むストリング。桁の数は 32 ビットの整数倍で構成されていなければなりません。ストリングが 16384 バイトを超える場合、警告メッセージが出され、プラグマが無視されます。

使用法

このプラグマは関数を定義し、プログラム・ソースの中で関数が通常定義される場所のみに現れます。

コンパイラーは他の関数に渡すのと同じ方法でこの関数にパラメーターを渡します。例えば、整数型の引数を取る関数では、1 番目のパラメーターが GPR3 に、2 番目が GPR4 に、と順番に渡されます。この関数によって戻される値は、整数値の場合は GPR3 で、浮動小数点または倍精度の値の場合は FPR1 になります。

instruction_sequence から生成されたコードは、**#pragma reg_killed_by** を使用して、関数によって使用される特定のレジスター・セットをリストしない限り、ご使用システムで使用可能なすべての揮発性レジスターを使用することができます。ご使用システムで使用可能な揮発性レジスターのリストについては、447 ページの『#pragma reg_killed_by』を参照してください。

インライン化オプションは、**#pragma mc_func** で定義された関数には影響しません。ただし、**#pragma isolated_call** を使用すると、そのような関数のランタイム・パフォーマンスを改善することができます。

例

以下の例では、**#pragma mc_func** は add_logical と呼ばれる関数を定義するために使用されています。この関数は、*循環桁上げ* で 2 つの整数を加算するためのマシン・インストラクションで構成されています。つまり加算の結果桁上げが発生すると、その桁上げを合計に加算します。この公式はチェックサムの計算で頻繁に使用されます。

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4          */
                /*   addze      r3 <- r3, carry bit   */


main() {

      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf(":¥n:¥nresult = %d:¥n:¥n",j);
}
```

プログラムを実行した結果は以下のようになります。

```
result = 1
```

関連情報

- 217 ページの『-qisolated_call』
- 447 ページの『#pragma reg_killed_by』
- 「*XL C/C++ ランゲージ・リファレンス*」の 『インライン・アセンブリー・ステートメント』

#pragma nofunctrace

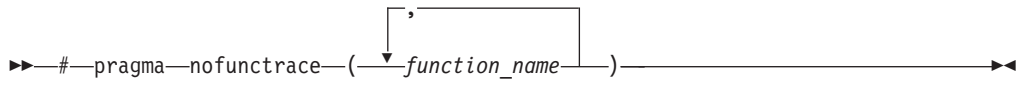
カテゴリー

エラー・チェックおよびデバッグ

目的

指定された関数または指定された関数のリストのトレースを無効にする。

構文



パラメーター

function_name
トレースを無効にする関数の名前。

使用法

#pragma nofunctrace を使用してトレースを無効にする関数のリストを指定する場合には、括弧 () を使用して、関数をその中にカプセル化してください。関数のリストで、各関数を分離するにはコンマ , を使用します。例えば、関数 a のトレースを無効にするには、#pragma nofunctrace(a) を使用します。関数 a、b、および c のトレースを無効にするには、#pragma nofunctrace(a,b,c) を使用します。

2 つの関数 foo(int) と foo(double) が存在する場合で、foo(int) のトレースを無効にして foo(double) のトレースを無効にしない場合には、#pragma nofunctrace(foo(int)) を使用します。

行内に 2 つのコロン :: が存在する場合、スコープ修飾子と解釈されます。例えば、-qfunctrace+A::B:C を呼び出すと、コンパイラーは、修飾子 A::B または C で始まる関数をトレースします。

注: コンパイラー・オプション **-qfunctrace** を使用して、指定した関数または関数のリストのトレースを無効にするには、そのサブオプション - の後に関数の名前を指定する必要があります。**-qfunctrace** および関連するサブオプションについて詳しくは、 170 ページの『-qfunctrace』を参照してください。

例

```
#pragma nofunctrace(a,b,c)
```

関連情報

- 170 ページの『-qfunctrace』

#pragma nosimd

『 183 ページの『-qhot』』を参照してください。

#pragma novector

『 183 ページの『-qhot』』を参照してください。

#pragma options

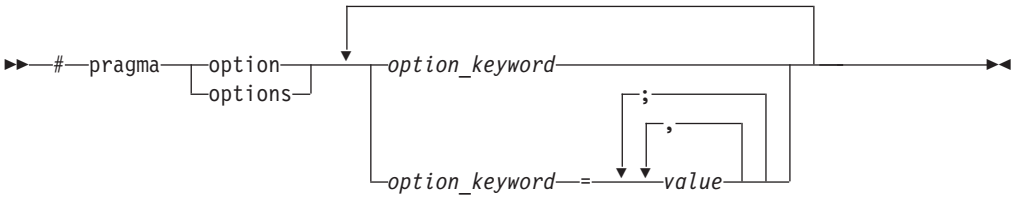
カテゴリー

言語エレメント制御

目的

使用しているソース・プログラムでコンパイラー・オプションを指定する。

構文



パラメーター

下記の表の設定は、**#pragma options** に有効なオプション です。詳しくは、同等のコンパイラー・オプションのページを参照してください。

pragma options *option_keyword* **に有効な設定**	**同等のコンパイラー・オプション**
align=*option*	109 ページの『-qalign』
[no]attr attr=full	120 ページの『-qattr』
chars=*option*	128 ページの『-qchars』
[no]check	129 ページの『-qcheck』
[no]compact	134 ページの『-qcompact』
[no]dbcs	273 ページの『-qmbcs, -qdbcs』
[no]digraph	146 ページの『-qdigraph』
[no]dollar	147 ページの『-qdollar』
enum=*option*	152 ページの『-qenum』
flag=*option*	158 ページの『-qflag』
float=[no]*option*	160 ページの『-qfloat』
[no]flttrap	165 ページの『-qflttrap』
[no]fullpath	169 ページの『-qfullpath』
halt	179 ページの『-qhalt』
[no]idirfirst	188 ページの『-qidirfirst』
[no]ignerrno	189 ページの『-qignerrno』
ignprag=*option*	190 ページの『-qignprag』
[no]info=*option*	193 ページの『-qinfo』
initauto=*value*	204 ページの『-qinitauto』
isolated_call=*names*	217 ページの『-qisolated_call』
C langlvl	227 ページの『-qlanglvl』
[no]ldbl128	256 ページの『-qldbl128』
[no]libansi	258 ページの『-qlibansi』
[no]list	261 ページの『-qlist』
[no]longlong	267 ページの『-qlonglong』
[no]maxmem=*number*	272 ページの『-qmaxmem』
[no]mbcs	273 ページの『-qmbcs, -qdbcs』

pragma options *option_keyword* に有効な設定	**同等のコンパイラー・オプション**
[no]optimize=*number*	279 ページの『-O、-qoptimize』
C++ priority=*number*	309 ページの『-qpriority (C++ のみ)』
proclocal、procimported、procunknown	310 ページの『-qprocimported、-qproclocal、-qprocunknown』
C [no]proto	313 ページの『-qproto (C のみ)』
[no]ro	320 ページの『-qro』
[no]roconst	321 ページの『-qroconst』
[no]showinc	328 ページの『-qshowinc』
[no]source	340 ページの『-qsource』
spill=*number*	343 ページの『-qspill』
[no]stdinc	351 ページの『-qstdinc』
[no]strict	352 ページの『-qstrict』
tbtable=*option*	364 ページの『-qtbtable』
tune=*option*	377 ページの『-qtune』
C [no]unroll=[*yes*/*no*/*auto*/*n*]	381 ページの『-qunroll』
C [no]upconv	385 ページの『-qupconv (C のみ)』
[no]xref	399 ページの『-qxref』

使用法

ほとんどの **#pragma options** ディレクティブは、ソース・プログラムのどのステートメントよりも前に指定しなければなりません。ただし、コメント、ブランク行、および他の #pragma の指定に限り、これらのディレクティブの前に置くことができます。例えば、以下のように、プログラムの最初の数行をコメントにして、その後に **#pragma options** ディレクティブを続けることができます。

```
/* The following is an example of a #pragma options directive: */

#pragma options langlvl=stdc89 halt=s spill=1024 source

/* The rest of the source follows ... */
```

#pragma options ディレクティブで複数のコンパイラー・オプションを指定するには、ブランク・スペースを使用してオプションを分割します。例を以下に示します。

```
#pragma options langlvl=stdc89 halt=s spill=1024 source
```

#pragma option_override

カテゴリー

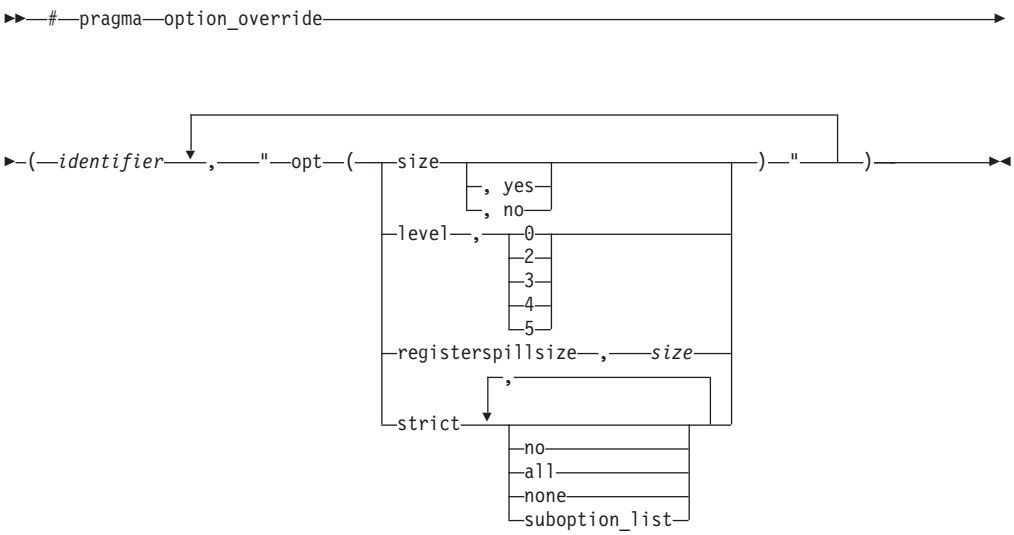
最適化およびチューニング

目的

コマンド行で指定された最適化オプションをオーバーライドするサブプログラム・レベルで最適化オプションを指定できるようにする。

これによってプログラムの最適化の微制御が可能となり、最適化でのみ発生するエラーのデバッグが可能になります。

構文



パラメーター

identifier
最適化オプションがオーバーライドされる関数の名前。

以下の表には、それぞれのプラグマ・サブオプションに同等なコマンド行オプションが示されています。

#pragma option_override 値	**同等なコンパイラー・オプション**
level, 0	-O
level, 2	-O2
level, 3	-O3
level, 4	-O4
level, 5	-O5
registerspillsize, *size*	-qspill=*size*
size	-qcompact
size, yes	
size, no	-qnocompact
strict, all	-qstrict, -qstrict=all
strict, no, none	-qnostrict
strict, *suboption_list*	-qstrict=*suboption_list*

デフォルト

デフォルト設定については、上記の表にリストされたオプションの説明を参照してください。

使用法

プラグマは、コマンド行オプションによって最適化が既に使用可能になっている場合にのみ有効です。プラグマで指定できる最適化レベルは、コンパイル中の残りのプログラムに適用されたレベルよりも低いレベルのみです。

#pragma option_override ディレクティブが影響を与えるのは、同じコンパイル単位で定義される関数のみです。プラグマ・ディレクティブは、変換単位のどこに指定してもかまいません。つまり、関数定義の前後、関数宣言の前後、関数が参照される前または参照された後、および関数定義の内部または外部に指定することができます。

C++ このプラグマは多重定義のメンバー関数には使用できません。

例

-O2 を使用して、関数 `foo` および `faa` を含む以下のコード・フラグメントをコンパイルするとします。関数 `faa` には `#pragma option_override(faa, "opt(level, 0)")` が含まれているため、最適化されません。

```
foo(){
     .
     .
     .
     }

#pragma option_override(faa, "opt(level, 0)")

faa(){
     .
     .
     .
     }
```

関連情報

- 279 ページの『-O、-qoptimize』
- 134 ページの『-qcompact』
- 343 ページの『-qspill』
- 352 ページの『-qstrict』

#pragma pack

カテゴリー

オブジェクト・コード制御

目的

すべての集合体メンバーの位置合わせを、指定したバイト境界に設定する。

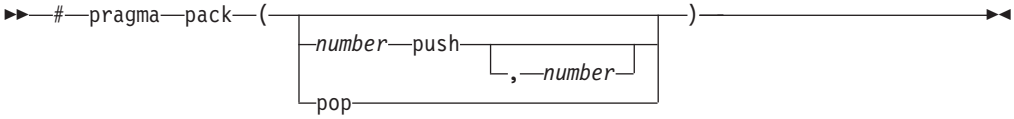
バイト境界の数がメンバーの自然な位置合わせよりも小さい場合は、埋め込みバイトが除去されるので、全体的な構造体または共用体のサイズが減少します。

このプラグマの構文およびセマンティクスは、**-qpack_semantic** オプションの設定に応じて異なります。

構文

デフォルトの #pragma pack 構文 (有効な -qpack_semantic=ibm)

有効な -qpack_semantic=gnu を使用した #pragma pack 構文



デフォルト

集合体 (構造体、共用体、およびクラス) のメンバーが自然な境界に位置合わせされ、構造体はその自然な境界上で終わります。集合体の位置合わせは、最も厳密なメンバー (最大の位置合わせ要件を持つメンバー) の位置合わせです。

パラメーター

nopack
パッキングを使用不可にします。このパラメーターは、**-qpack_semantic=gnu** が有効なときには認識されません。警告メッセージが出され、プラグマが無視されますので注意してください。

push
number なしで指定すると、現在有効な任意の値をパッキングの『スタック』の先頭へプッシュします。 *number* を指定して使用すると、その値をパッキング・スタックの先頭へプッシュし、そのパッキング値を次の構造体の *number* のパッキング値に設定します。このパラメーターは、**-qpack_semantic=ibm** が有効なときには認識されません。警告メッセージが出され、プラグマが無視されますので注意してください。

number
以下のいずれかです。

1 1 バイトの境界上または自然な位置合わせの境界上のどちらか小さい方に、構造体メンバーを位置合わせします。

2 2 バイトの境界上または自然な位置合わせの境界上のどちらか小さい方に、構造体メンバーを位置合わせします。

4 4 バイトの境界上または自然な位置合わせの境界上のどちらか小さい方に、構造体メンバーを位置合わせします。

8 8 バイトの境界上または自然な位置合わせの境界上のどちらか小さい方に、構造体メンバーを位置合わせします。

16 16 バイトの境界上または自然な位置合わせの境界上のどちらか小さい方に、構造体メンバーを位置合わせします。

pop

-qpack_semantic=ibm が有効な場合は、パッキング規則を現行設定の前に有効であったパッキング規則に設定します。 **-qpack_semantic=gnu** が有効な場合は、最後の **push** 文で指定された値をスタックからポップし、現行のパッキング値をスタックの先頭にある値にリセットして、**push** 文なしで指定された可能性のある介在値をすべてオーバーライドします。

パラメーターなしで (つまり、空の括弧で) **#pragma pack()** を指定することには、次の効果があります。

- **-qpack_semantic=ibm** が有効な場合、すべてのパッキングを使用不可にします (**#pragma pack(nopack)** を指定することと同等です)。
- **-qpack_semantic=gnu** が有効な場合、現行のパッキング値をコンパイル単位の最初で有効だったパッキング値に設定します。

使用法

#pragma pack ディレクティブは、集合体型のインスタンスの宣言よりも、集合体型の定義に適用されます。そのため、指定したタイプの宣言された変数のすべてに自動的に適用されます。

#pragma pack ディレクティブは、このディレクティブの後に続く宣言を持つ構造体のメンバーのみに適用される現行の位置合わせ規則を変更します。これにより、構造体の位置合わせに直接、影響することはありませんが、構造体のメンバーの位置合わせに影響することで、構造体全体の位置合わせに影響する場合があります。

#pragma pack ディレクティブでは、メンバーの位置合わせを強化することはできず、むしろ位置合わせを低下させる可能性があります。例えば、短整数データ型のメンバーの場合、**#pragma pack(1)** ディレクティブでは、当該メンバーは構造体内で 1 バイト境界でパックされますが、 **#pragma pack(4)** ディレクティブでは有効ではありません。

#pragma pack ディレクティブは、ビット・フィールドにビット・フィールド・コンテナーの境界をクロスさせます。

```
#pragma pack(2)
struct A{
int a:31;
int b:2;
}x;

int main(){
printf("size of S = %d¥n", sizeof(s));
}

When compiled and run, the output is:
size of S = 6

But if you remove the #pragma pack directive, you get this output:
size of S = 8
```

#pragma pack ディレクティブは、構造体または共用体の完全な宣言にのみ適用されます。ただし、メンバー・リストが指定されないフォワード宣言は除きます。例えば、以下のコード・フラグメントでは、 struct S の位置合わせは 4 です。これは、この規則がメンバー・リストが宣言されるときに有効であるためです。

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

以下の例のように、ネストされた構造体は、それが含まれている構造体の位置合わせを持たず、その宣言に先行する位置合わせを持ちます。

```
#pragma pack (4)                // 4-byte alignment
         struct nested {
           int  x;
           char y;
           int  z;
         };

         #pragma pack(1)           // 1-byte alignment
         struct packedcxx{
            char   a;
            short  b;
            struct nested  s1;     // 4-byte alignment
         };
```

複数の **#pragma pack** ディレクティブが、インライン化された関数で定義された構造体に現れる場合は、構造体の先頭で有効な **#pragma pack** ディレクティブが優先されます。

例

以下の例は、**#pragma pack** ディレクティブを使用して構造体定義の位置合わせを設定する方法を示しています。

```
//  header file file.h

   #pragma pack(1)

   struct jeff{           //    this structure is packed
     short bill;          //    along 1-byte boundaries
     int *chris;
   };
   #pragma pack(pop)       //   reset to previous alignment rule

// source file anyfile.c

   #include "file.h"

   struct jeff j;           //   uses the alignment specified
                            //   by the pragma pack directive
                            //   in the header file and is
                            //   packed along 1-byte boundaries
```

この例は、**#pragma pack** ディレクティブが構造体のサイズとマッピングにどのような影響を与えるかを示したものです。

```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

デフォルト・マッピング:

s_t のサイズは 16
a のオフセットは 0
b のオフセットは 4
c のオフセットは 8
d のオフセットは 12
a の位置合わせは 1
b の位置あわせは 4
c の位置合わせは 2
d の位置合わせは 4

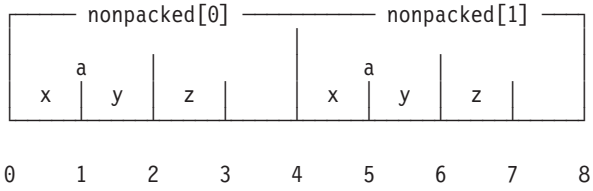
#pragma pack(1):

s_t のサイズは 11
a のオフセットは 0
b のオフセットは 1
c のオフセットは 5
d のオフセットは 7
a の位置合わせは 1
b の位置合わせは 1
c の位置合わせは 1
d の位置合わせは 1

以下の例では、構造体をメンバーの 1 つとして含む共用体 uu を定義し、タイプ uu の 2 つの共用体の配列を宣言しています。

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu nonpacked[2];
```

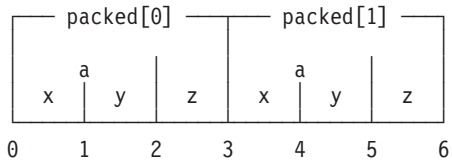
共用体メンバーの中では位置合わせに関する最大の要件が short a、つまり 2 バイトの要件なので、埋め込みの 1 バイトが配列中の各共用体の終わりに追加され、この要件を強制します。



次の例では、**#pragma pack(1)** を使用して、タイプ uu の共用体の位置合わせを 1 バイトに設定しています。

```
#pragma pack(1)

union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu pack_array[2];
```

配列 packed 中の各共用体の長さは、以前は 4 バイトでしたが、現在は 3 バイトしかありません。



以下の例は、このプラグマのセマンティクスにおいて **-qpack_semantic=ibm** または **-qpack_semantic=ibm** のどちらが有効であるかによって生じる相違の結果を示しています。

以下の例は、**push** パラメーターを指定した場合の効果を示しています。

```
#pragma pack(1)
#pragma pack(push)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(push,2)  // ignored when -qpack_semantic=ibm is in effect

struct s_t {
   char a;
   int b;
} S;
```

-qpack_semantic=gnu が有効な場合は、構造体 S が宣言されるときに有効なパッキング値が 2 であり、構造体が 2 バイトの境界上に位置合わせされます。**-qpack_semantic=ibm** が有効な場合は、2 番目の 2 つのディレクティブが無視され、構造体 S に有効なパッキング値が 1 であり、1 バイト境界上に位置合わせされます。

以下の例は、**push** および **pop** パラメーターを一緒に指定した場合の効果を示しています。

```
#pragma pack(push,1)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(push,4)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(2)
#pragma pack(pop)
#pragma pack(pop)
#pragma pack(pop)

struct s_t {
   char a;
   int b;
} S;
```

-qpack_semantic=gnu が有効な場合は、**pop** によってポップされるのは **push** ディレクティブによってスタック上にプッシュされた値のみであるため、最初の **pop** ディレクティブはスタックから 4 をポップし、2 番目のディレクティブはスタックから 1 をポップし、位置合わせは、コンパイル単位の先頭で有効な設定です (介在する **#pragma pack(2)** ディレクティブがオーバーライドされます)。
-qpack_semantic=ibm が有効な場合は、**pop** 文がスタックから値 2 をポップし、位置合わせはコンパイル単位の先頭で有効な設定です。

以下の例は、ネストされた構造体の内部でディレクティブを指定した場合の効果を示しています。

```
struct s_t {
   char a;
   int b;

   #pragma pack(1)
```

```
   struct t_t {
      char x;
      int y;
   }T;

   char c;

   #pragma pack(2)
   #pragma pack(1)

   int d;

   #pragma align(natural)   ¥¥ this only affects u_t.
   #pragma pack(2)          ¥¥ this only affects u_t.

   struct u_t {
      char j;
      int k;
   }U;

}S;
```

-qpack_semantic=gnu が有効な場合は、最初の **#pragma pack(1)** ディレクティブは構造体 `t_t` および `s_t` の両方に適用されます。 **-qpack_semantic=ibm** によって、最初の **#pragma pack(1)** ディレクティブは構造体 `t_t` のみに適用されます。

関連情報

- 109 ページの『-qalign』
- 289 ページの『-qpack_semantic』
- 「*XL C/C++ 最適化およびプログラミング・ガイド*」の『位置合わせ修飾子の使用法』

#pragma priority (C++ のみ)

『 309 ページの『-qpriority (C++ のみ)』』を参照してください。

#pragma reachable

カテゴリー

最適化およびチューニング

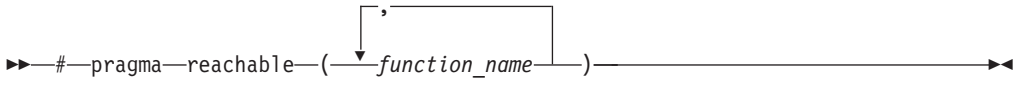
目的

ある名前の関数の後のプログラム内の位置が、不明なロケーションから分岐先のターゲットとして指定される可能性があることをコンパイラーに通知する。

指定された関数の後の命令に、指定された関数の戻りステートメント以外のプログラムのポイントから到達できることをコンパイラーに通知することによって、プラグマは最適化のための機会を追加することができます。

注: setjmp.h ヘッダー・ファイルを組み込むときに、コンパイラーは、関数 (`setjmp`、`_setjmp`、`sigsetjmp`、および `_sigsetjmp`) の `setjmp` ファミリーの **#pragma reachable** ディレクティブを自動的に挿入します。

構文



パラメーター

function_name
関数の戻りステートメント以外のプログラムのポイントから到達可能な命令の前の関数の名前。

デフォルト

適用されません。

関連情報

- 429 ページの『#pragma leaves』

#pragma reg_killed_by

カテゴリー

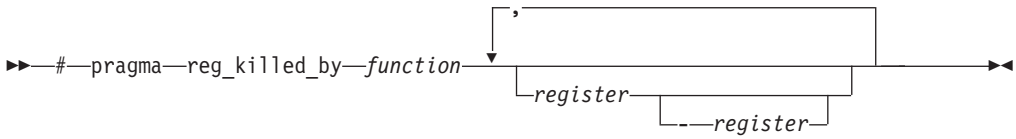
最適化およびチューニング

目的

#pragma mc_func が指定する関数によって変更される可能性のあるレジスターを指定する。

通常、**#pragma mc_func** によって指定された関数に対して生成されたコードは、システム上で使用可能なすべての揮発性レジスターを変更することができます。そのような関数によって変更される特定の揮発性レジスターのセットを明示的にリストするには、**#pragma reg_killed_by** を使用することができます。このリストに入っていないレジスターは変更されません。

構文



パラメーター

function
以前に **#pragma mc_func** ディレクティブを使用して定義した関数の名前。

register
指定された *function* によって変更される単一のレジスターまたはレジスターの範囲のシンボル名。シンボル名はターゲット・プラットフォーム上で有効なレジスター名でなければなりません。有効なレジスターは以下のとおりです。

cr0、cr1、および cr5 から cr7
条件レジスター

ctr カウント・レジスター

gr0 および gr3 から gr12
汎用レジスター

fp0 から fp13
浮動小数点レジスター

fsr 浮動小数点および状況制御レジスター

lr リンク・レジスター

vr0 から vr31
ベクトル・レジスター (選択されたプロセッサー専用)

xer 固定小数点例外レジスター

レジスターの範囲は、ダッシュで区切って開始レジスターと終了レジスターの両方のシンボル名を提供することにより識別することができます。

register が指定されていない場合は、どの揮発性レジスターも、指定された *function* によって変更 (kill) されません。

例

以下の例は、**#pragma reg_killed_by** を使用して、**#pragma mc_func** で定義された関数によって使用される揮発性レジスターの特定セットをリストする方法を示しています。

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4           */
                /*   addze      r3 <- r3, carry bit    */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf(":¥n:¥nresult = %d:¥n:¥n",j);
}
```

関連情報

- 433 ページの『#pragma mc_func』

#pragma report (C++ のみ)

カテゴリー

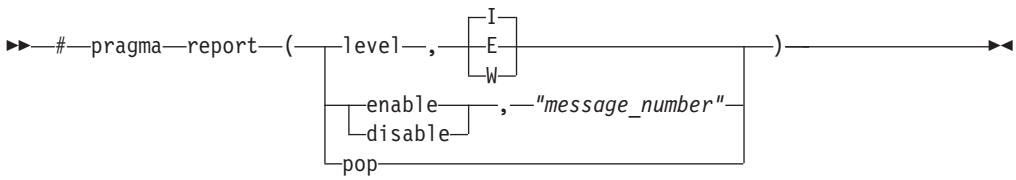
リスト、メッセージ、およびコンパイラー情報

目的

診断メッセージの生成を管理する。

プラグマを使用して、メッセージで表示する最小の重大度レベルを指定するか、一般的なレポート・レベルかどうかにかかわらず特定のメッセージを使用可能または使用不可にすることができます。

構文



デフォルト

デフォルトのレポート・レベルは「通知 **(I)**」で、すべてのタイプのメッセージを表示します。

パラメーター

level
表示する診断メッセージの最小の重大度レベルによってプラグマが設定されることを示します。

E エラー・メッセージのみが表示されることを示します。エラー・メッセージの重大度は最高です。これは、**-qflag=e:e** コンパイラー・オプションと同等です。

W 警告メッセージおよびエラー・メッセージが表示されることを示します。これは、**-qflag=w:w** コンパイラー・オプションと同等です。

I すべての診断メッセージ (警告メッセージ、エラー・メッセージ、および通知メッセージ) が表示されることを示します。通知メッセージの重大度は最低です。これは、**-qflag=i:i** コンパイラー・オプションと同等です。

enable
指定した *"message_number"* を使用可能にします。

disable
指定した *"message_number"* を使用不可にします。

"message_number"
接頭部とメッセージ番号を引用符で囲んだメッセージ ID (例えば、"CCN1004") を表します。

注: 前の例の "CCN1004" のように、*message_number* に引用符を使用する必要があります。

pop
以前に有効だったレポート・レベルに戻します。レポート・レベルが以前に指定されなかった場合は、警告が出され、レポート・レベルはそのまま変更されません。

使用法

プラグマは、 **#pragma info** およびほとんどのコンパイラー・オプションよりも優先されます。例えば、**#pragma report** を使用してコンパイラー・メッセージを使用不可にすると、そのメッセージは **-qflag** コンパイラー・オプション設定で表示されません。

関連情報

- 158 ページの『-qflag』

#pragma simd_level

カテゴリー

最適化およびチューニング

目的

個別ループのベクトル命令のコンパイラー・コード生成を制御する。

ベクトル命令を、マルチメディア・アプリケーションなどのアルゴリズム集約型のタスクで使用すると、高いハイパフォーマンスを得ることができます。ループごとに自動 SIMD 化の積極性を柔軟に制御することができます。また、このような詳細な制御を行うことにより、パフォーマンスを向上できる可能性があります。

サポートされるレベルは 0 から 10 です。level(0) は、プラグマ・ディレクティブの後のループで自動 SIMD 化を行わないことを示します。 level(10) は、ループで最も積極的な形式の自動 SIMD 化を行うことを示します。このプラグマ・ディレクティブにより、ループごとに自動 SIMD 化の動作を制御できます。

構文

パラメーター

n プラグマ・ディレクティブの後のループでの自動 SIMD 化の積極性を指定する、0 から 10 のスカラー整数初期化式。

使用法

-qsimd=auto を有効にする場合、simd_level プラグマが指定されていないループの SIMD レベルは、デフォルトで 5 に設定されます。

#pragma simd_level(0) は **#pragma nosimd** と等価です。この場合、プラグマ・ディレクティブの後のループで自動 SIMD 化は実行されません。

#pragma simd_level(10) は、プラグマ・ディレクティブの後のループで、バイパス・コスト分析も含め、最も積極的に自動 SIMD 化を実行するようにコンパイラーに示します。

規則

#pragma simd_level ディレクティブの規則を以下にリストします。

- **#pragma simd_level** ディレクティブが有効になるのは、ベクトル命令をサポートするアーキテクチャーの場合および **-qsimd=auto** と共に使用された場合のみです。
- **#pragma simd_level** ディレクティブは、その直後のループにのみ適用されます。このディレクティブは、指定したループ内にネストされたその他のループに対しては無効です。個別の **#pragma simd_level** ディレクティブを指定することで、内部のループおよび外部のループに対して異なる SIMD レベルを設定することができます。
- **#pragma simd_level** ディレクティブは、ループ最適化 (**-qhot**) および OpenMP ディレクティブと混用できます。特定の最適化レベルは必要とされません。**-qhot** および OpenMP ディレクティブについて詳しくは、本書の 183 ページの『-qhot』、および「*IBM XL C/C++ 最適化およびプログラミング・ガイド*」の『OpenMP ディレクティブの使用』を参照してください。

例

```
...
#pragma simd_level(10)
for (i=1; i<1000; i++) {
/* program code */

} ...
```

関連情報

•

#pragma STDC cx_limited_range

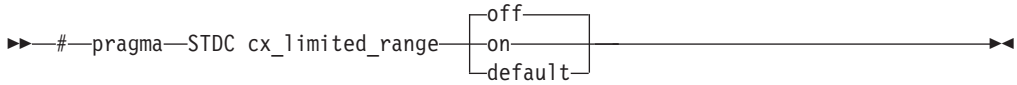
カテゴリー

最適化およびチューニング

目的

中間計算のオーバーフローや重要度の喪失が発生しないような値だけを使用して複素数除算と絶対値を計算することを、コンパイラーに指示する。

構文



使用法

限定範囲外の値を使用すると誤った結果が生成される可能性があります。ここで、限定範囲とは「明確なシンボリック定義」がオーバーフローしたり精度を欠かないことと定義されています。

プラグマはその最初の発生場所から、別の **cx_limited_range** プラグマを検出するか変換単位の終わりまで有効となります。プラグマは、複合ステートメント (ネストされた複合ステートメント内など) の中で発生した場合、その最初の発生場所から、別の **cx_limited_range** プラグマを検出するか、複合ステートメントの終わりまで有効になります。

例

以下の例は、複素数除算のプラグマの使用方法を示しています。

```
#include <complex.h>

_Complex double a, b, c, d;
void p() {

d = b/c;

{

#pragma STDC CX_LIMITED_RANGE ON

a = b / c;

}
}
```

以下の例は、複素数絶対値のプラグマの使用方法を示しています。

```
#include <complex.h>

_Complex double cd = 10.10 + 10.10*I;
int p() {

#pragma STDC CX_LIMITED_RANGE ON

double d = cabs(cd);
}
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の 『標準プラグマ』

#pragma stream_unroll

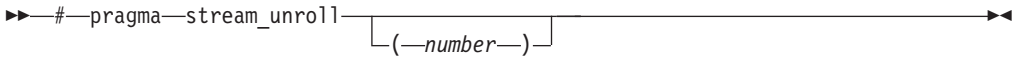
カテゴリー

最適化およびチューニング

目的

最適化が使用可能な場合、for ループに含まれたストリームを複数のストリームに切断する。

構文



パラメーター

number

ループのアンロール係数。 C *number* の値は正の整数定数式です。 C++ *number* の値は正のスカラー整数またはコンパイル時の定数初期化式です。

アンロール係数 1 はアンロールを使用不可にします。

number が指定されていない場合は、最適化プログラムはそれぞれネストされたループごとに適切なアンロール係数を判別します。

使用法

ストリームのアンロールを使用可能にするには、 **-qhot** および **-qstrict**、または **-qsmp** を指定するか、最適化レベル **-O4** 以上を使用してください。**-qstrict** が有効な場合、ストリームのアンロールは行われません。

ストリームのアンロールが発生するためには、**#pragma stream_unroll** ディレクティブが `for` ループの前に指定される最後のプラグマでなければなりません。

C 同じ `for` ループに対して **#pragma stream_unroll** を複数回指定したり、それを他のループのアンロール・プラグマ (**#pragma unroll**、**#pragma nounroll**、**#pragma unrollandfuse**、**#pragma nounrollandfuse**) と結合すると警告が発行されます。 C++ コンパイラーは、同じ `for` ループに指定された複数のループのアンロール・プラグマのうち最後のプラグマを除き、そのまますべて無視します。

例

#pragma stream_unroll がどのようにパフォーマンスを向上させることができるかを以下の例に示します。

```
int i, m, n;
int a[1000];
int b[1000];
int c[1000];

....

#pragma stream_unroll(4)
for (i=0; i<n; i++) {
  a[i] = b[i] * c[i];
}
```

以下のように、アンロール係数 4 は繰り返し回数を `n` から `n/4` に削減します。

```
m = n/4;

for (i=0; i<n/4; i++){
  a[i] = b[i] + c[i];
  a[i+m] = b[i+m] + c[i+m];
  a[i+2*m] = b[i+2*m] + c[i+2*m];
  a[i+3*m] = b[i+3*m] + c[i+3*m];
}
```

読み取り操作と格納操作の増加分は、コンパイラーが決定した数のストリームに振り分けられ、その結果として計算時間が短くなり、パフォーマンスが向上します。

関連情報

- 381 ページの『-qunroll』
- 『#pragma unrollandfuse』

#pragma strings

『 320 ページの『-qro』』を参照してください。

#pragma unroll

『 381 ページの『-qunroll』』を参照してください。

#pragma unrollandfuse

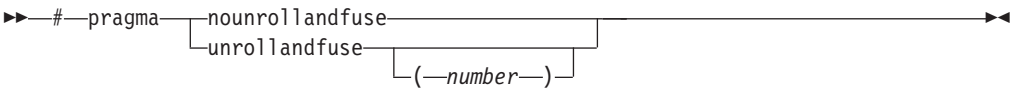
カテゴリー

最適化およびチューニング

目的

ネストされた `for` ループでアンロールおよびヒューズ操作を試行するようコンパイラーに命令する。

構文



パラメーター

number

ループのアンロール係数。 C *number* の値は正の整数定数式です。 C++ *number* の値は正のスカラー整数またはコンパイル時の定数初期化式です。

number が指定されていない場合は、最適化プログラムはそれぞれネストされたループごとに適切なアンロール係数を判別します。

使用法

#pragma unrollandfuse ディレクティブは、以下の条件を満たすネストされた `for` ループの外部ループのみに適用されます。

- 1 つのループ・カウンター変数、その変数に対する 1 つの増分ポイント、および 1 つの終了変数のみが存在している必要があります。これらはループ・ネストのどのポイントでも変更できません。
- ループは複数の入り口点と出口点を持つことはできません。ループの終了がループを終了するための唯一の方法でなければなりません。
- ループ内の依存関係は「後方参照」にすることはできません。例えば、 `A[i][j] = A[i -1][j + 1] + 4)` などのステートメントがループ内に現れることはできません。

ループのアンロールが発生するためには、**#pragma unrollandfuse** ディレクティブが for ループに先行している必要があります。最も内部の for ループに対して **#pragma unrollandfuse** を指定することはできません。

同じ for ループに対して **#pragma unrollandfuse** を複数回指定したり、このディレクティブを **#pragma nounrollandfuse**、**#pragma nounroll**、**#pragma unroll**、または **#pragma stream_unroll** ディレクティブと結合しないでください。

事前定義マクロ

なし。

例

以下の例では、**#pragma unrollandfuse** ディレクティブはループの本体を複製して、ヒューズしています。これにより、配列 b に対するキャッシュの欠落の数が削減されます。

```
int i, j;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];


....

#pragma unrollandfuse(2)
for (i=1; i<1000; i++) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
    }
}
```

以下の for ループは、**#pragma unrollandfuse(2)** ディレクティブを上記のループに適用した場合に生じる可能性のある結果を示しています。

```
for (i=1; i<1000; i=i+2) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
        a[j][i+1] = b[i+1][j] * c[j][i+1];
    }
}
```

また、ネストされたループ構造体に複数の **#pragma unrollandfuse** ディレクティブを指定することもできます。

```
int i, j, k;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];
int d[1000][1000];
int e[1000][1000];


....

#pragma unrollandfuse(4)
for (i=1; i<1000; i++) {
#pragma unrollandfuse(2)
    for (j=1; j<1000; j++) {
   for (k=1; k<1000; k++) {
```

```
            a[j][i] = b[i][j] * c[j][i] + d[j][k] * e[i][k];
        }
    }
}
```

関連情報

- 381 ページの『-qunroll』
- 452 ページの『#pragma stream_unroll』

#pragma weak

カテゴリー

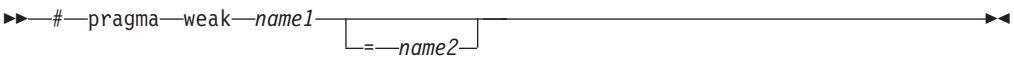
オブジェクト・コード制御

目的

リンク時に複数定義されたシンボルが見つかった場合、またはシンボルの定義が見つからなかった場合、リンカーがエラー・メッセージを出さないようにする。

プラグマを使用して、プログラムがライブラリー関数と同じ名前のユーザー定義関数を呼び出すのを許可することができます。ライブラリー関数定義を『弱い』とマークを付けることによって、プログラマーは関数の『強い』バージョンを参照し、リンカーにオブジェクト・コードのグローバル・シンボルの定義を複数受け取らせることができます。このプラグマは関数とともに使用することを基本条件として設計されていますが、ほとんどのデータ・オブジェクトに対しても有効に機能します。

構文



パラメーター

name1
外部リンケージを持つデータ・オブジェクトまたは関数の名前。

name2
外部リンケージを持つデータ・オブジェクトまたは関数の名前。

C++ *name2* はメンバー関数にできません。*name2* がテンプレート関数である場合は、明示的にテンプレート関数のインスタンスを生成する必要があります。

C++ 名前は、マングル名を使用して指定する必要があります。C++ マングル名を取得するには、**-c** コンパイラー・オプションを使用してオブジェクト・ファイルのみにソースをコンパイルし、その結果生じるオブジェクト・ファイルで **nm** オペレーティング・システム・コマンドを使用します。(名前マングリングを回避するための、宣言に対する `extern "C"` リンケージ指定子の使用について詳しくは、「*XL C/C++ ランゲージ・リファレンス*」の『名前マングリング』も参照してください。)

使用法

弱いプラグマの形式は 2 つあります。

#pragma weak *name1*

このプラグマの形式は、特定のコンパイル単位において *name1* の定義を『弱い』とマークを付けます。*name1* がプログラム内の任意の場所から参照される場合に、リンカーは定義の『強い』バージョン (つまり、**#pragma weak** とマークを付けられていない定義) があればそれを使用します。強い定義がない場合、リンカーは弱い定義を使用します。弱い定義が複数ある場合は、リンカーが選択する弱い定義は指定されません (通常、リンカーはリンク・ステップ中にコマンド行で指定される最初のオブジェクト・ファイルにある定義を使用します)。 *name1* は **#pragma weak** と同じコンパイル単位で定義されなければなりません。*name1* が参照されてもその定義が見つからない場合は、値に 0 が割り当てられます。

#pragma weak *name1*=*name2*

このプラグマの形式は、特定のコンパイル単位の *name1* の弱い定義と、*name2* の別名を作成します。*name1* がプログラム内の任意の場所から参照される場合に、リンカーは定義の『強い』バージョン (つまり、**#pragma weak** とマークを付けられていない定義) があればそれを使用します。強い定義がない場合は、リンカーは弱い定義を使用します。これによって、*name2* の定義に解決されます。弱い定義が複数ある場合は、リンカーが選択する弱い定義は指定されません (通常、リンカーはリンク・ステップ中にコマンド行で指定される最初のオブジェクト・ファイルにある定義を使用します)。

name2 は **#pragma weak** と同じコンパイル単位で定義されなければなりません。*name1* は **#pragma weak** と同じコンパイル単位で宣言できる場合とできない場合がありますが、このコンパイル単位では定義しないでください。 *name1* がこのコンパイル単位で宣言される場合、*name1* の宣言は *name2* の宣言と互換性がなければなりません。例えば、*name2* が関数の場合、*name1* は *name2* と同じ戻りの型および引数の型を持っている必要があります。

このプラグマは初期化されていないグローバル・データや、実行可能ファイルにエクスポートされる共有ライブラリー・データ・オブジェクトと共に使用しないでください。

例

以下は **#pragma weak** *name1* 形式の例です。

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
       foo();
}

// Compilation unit 2:
```

```
#include <stdio.h>

#if __cplusplus
#pragma weak _Z3foov
#else
#pragma weak foo
#endif
void foo()
{
       printf("Foo called from compilation unit 2¥n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
       printf("Foo called from compilation unit 3¥n");
}
```

3 つのコンパイル単位がすべてコンパイルされ、一緒にリンクされると、リンカーはコンパイル単位 1 の `foo` への呼び出しのためのコンパイル単位 3 における `foo` の強い定義を使用します。出力は次のようになります。

```
Foo called from compilation unit 3
```

コンパイル単位 1 および 2 のみがコンパイルされ、一緒にリンクされると、リンカーはコンパイル単位 2 の `foo` の弱い定義を使用します。出力は次のようになります。

```
Foo called from compilation unit 2
```

以下は **#pragma weak** *name1=name2* 形式の例です。

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
foo();
}

// Compilation unit 2:

#include <stdio.h>

void foo(); // optional

#if __cplusplus
#pragma weak _Z3foov = _Z4foo2v
#else#pragma weak foo = foo2
#endif
void foo2()
{
printf("Hello from foo2!¥n");
}

// Compilation unit 3:

#include <stdio.h>
```

```
void foo()
{
printf("Hello from foo!¥n");
}
```

3 つのコンパイル単位がすべてコンパイルされ、一緒にリンクされると、リンカーはコンパイル単位 1 からの foo への呼び出しのためのコンパイル単位 3 における foo の強い定義を使用します。出力は次のようになります。

```
Hello from foo!
```

コンパイル単位 1 および 2 のみがコンパイルされ、一緒にリンクされると、リンカーは、foo2 の別名であるコンパイル単位 2 の foo の弱い定義を使用します。出力は次のようになります。

```
Hello from foo2!
```

関連情報

- 「*XL C/C++ ランゲージ・リファレンス*」の『weak 変数属性』
- 「*XL C/C++ ランゲージ・リファレンス*」の『weak 関数属性』
- 431 ページの『#pragma map』

並列処理のためのプラグマ・ディレクティブ

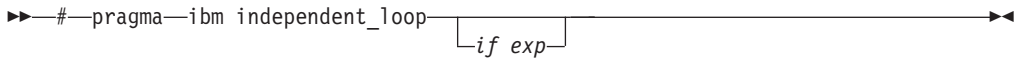
並列処理操作は、プログラム・ソースのプラグマ・ディレクティブによって制御されます。プラグマは、**-qsmp** コンパイラー・オプションで並列化が使用可能になっているときにのみ有効です。

#pragma ibm independent_loop

目的

independent_loop プラグマは、選択したループの繰り返しが独立しており、かつそれらの繰り返しを並列で実行できることを明示的に示します。

構文



ここで、exp はスカラー式を表します。

使用法

ループの繰り返しが独立である場合は、ループ・ブロックの前にプラグマを配置できます。その場合、コンパイラーはそれらの繰り返しを並列で実行します。exp 引数を指定すると、ループの繰り返しは、exp が実行時に TRUE に評価された場合にのみ独立であると見なされます。

注:

- 選択したループの繰り返しが独立でない場合は、コンパイラーは、**independent_loop** プラグマが指定されているかどうかに関わらず、ループの繰り返しを順次実行します。

- **independent_loop** プラグマをループに対して有効にするには、このプラグマを当該ループの直前に配置する必要があります。そうしないと、このプラグマは無視されます。
- 複数の **independent_loop** プラグマをループの前に指定した場合は、最後のプラグマのみが有効になります。

このプラグマを **omp parallel for** プラグマと組み合わせて、特定の並列処理スケジューリング・アルゴリズムを選択することができます。詳しくは、『 470 ページの『#pragma omp parallel for』』を参照してください。

例

以下の例では、引数 k の値が 2 より大きい場合に、ループの繰り返しが並列で実行されます。

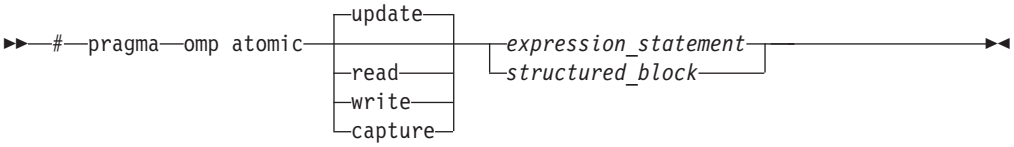
```
int a[1000], b[1000], c[1000];
int main(int k){
   if(k>0){
      #pragma ibm independent_loop if (k>2)
      for(int i=0; i<900; i++){
         a[i]=b[i]*c[i];
      }
   }
}
```

#pragma omp atomic

目的

omp atomic ディレクティブにより、特定のメモリー・ロケーションにアトミックにアクセスできます。特定のメモリー・ロケーションに対して読み書きする可能性がある並行スレッドの直接制御により、競合状態が確実に回避されます。 **omp atomic** ディレクティブを使用することで、ロックの少ない、より効率的な並行アルゴリズムを作成できます。

構文



ここで、*expression_statement* はスカラー型の式ステートメントであり、*structured_block* は 2 つの式ステートメントの構造化ブロックです。

節

update

変数の値をアトミックに更新します。一度に 1 つのスレッドしか共有変数を更新しないことが保証され、同じ変数に同時に書き込むことによるエラーは発生しません。節のない **omp atomic** ディレクティブは、**omp atomic** 更新と等価です。

注: アトミック更新では、メモリー・ロケーションに任意のデータを書き込むことはできず、メモリー・ロケーションの以前のデータに依存します。

read
変数の値をアトミックに読み取ります。共有変数の値は安全に読み取ることができ、並行スレッドが同時にアクセスして変数の中間値を読み取る危険は生じません。

write
変数の値をアトミックに書き込みます。共有変数の値は排他的に書き込むことができるため、同時書き込みによるエラーは生じません。

capture
変数の元の値または最終値をアトミックに取り込んで、変数の値を更新します。

expression_statement または *structured_block* は、アトミック・ディレクティブ節に応じて、以下のいずれかの形式を取ります。

ディレクティブ節	*expression_statement*	*structured_block*

update
(節なしと等価)
	`x++;`
	`x--;`
	`++x;`
	`--x;`
	`x binop = expr;`
	`x = x binop expr;`
	`x = expr binop x;`	

read	`v = x;`	
write	`x = expr;`	

capture	`v = x++;`
	`v = x--;`
	`v = ++x;`
	`v = --x;`
	`v = x binop = expr;`
	`v = x = x binop expr;`
	`v = x = expr binop x;`
		`{v = x; x binop = expr;}`
		`{v = x; xOP;}`
		`{v = x; OPx;}`
		`{x binop = expr; v = x;}`
		`{xOP; v = x;}`
		`{OPx; v = x;}`
		`{v = x; x = x binop expr;}`
		`{x = x binop expr; v = x;}`
		`{v = x; x = expr binop x;}`
		`{x = expr binop x; v = x;}`
		`{v = x; x = expr;}`[1]

ディレクティブ節	*expression_statement*	*structured_block*
注: 1. この式は、アトミックなスワップ操作をサポートするためのものです。		

ここで、

x、*v*　どちらもスカラー型の左辺値式です。

expr　は、*x* を参照しないスカラー型の式です。

binop　以下の 2 項演算子のいずれかです。

```
+ * - / & ^ | << >>
```

OP　++ または -- のいずれかです。

注: *binop*、*binop*=、および *OP* は多重定義された演算子ではありません。

使用法

並列更新が可能で、競合状態の対象となる可能性のあるオブジェクトは、**omp atomic** ディレクティブで保護する必要があります。

プログラム全体での *x* によって指定されたストレージ・ロケーションへのアトミック・アクセスはすべて、互換型を持つ必要があります。

アトミック領域内で、*x* を構文で複数回使用する場合は、同じストレージ・ロケーションを指定する必要があります。

並行プログラム全体での特定のストレージ・ロケーションへのすべてのアクセスはアトミックでなければなりません。メモリー・ロケーションへの非アトミック・アクセスにより、そのストレージ・ロケーションに対してすべてアトミックにアクセスするという、期待されるアトミック動作に従うことができなくなる可能性があります。

v も *expr* も、*x* によって指定されたストレージ・ロケーションにアクセスできません。

x も *expr* も、*v* によって指定されたストレージ・ロケーションにアクセスできません。

x によって指定されたストレージ・ロケーションへのアクションはすべてアトミックです。式 *expr*、*v*、*x* の評価はアトミックではありません。

アトミックな取り込みアクセスでは、取り込まれた値を *v* によって表されたストレージ・ロケーションへ書き込む操作は、アトミックには行われません。

例

例 1: アトミック更新

```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates. */
#pragma omp atomic
x[index[i]] += y;
```

```
/* Protect against race conditions with updates through x. */
#pragma omp atomic
p[i] -= 1.0f;
```

例 2: アトミックな読み取り、書き込み、および更新

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;

  #pragma omp atomic update
  x[i] *= 2;
}
```

例 3: アトミックな取り込み

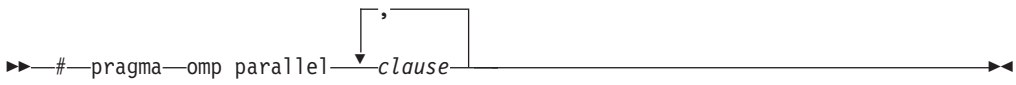
```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic capture
  temp[i] = x[f(i)]++;

  #pragma omp atomic capture
  {
    temp[i] = x[f(i)]; //the two occurences of x[f(i)] must evaluate to the
    x[f(i)] -= 3; //same memory location, otherwise behavior is undefined.
  }
}
```

#pragma omp parallel

目的

omp parallel ディレクティブは、選択したコードのブロックを並列化するようコンパイラーに明示的に指示します。

構文



パラメーター

clause は、以下の節のいずれかです。

if (*exp*)
: if 引数が指定されると、*exp* によって示されたスカラー式が実行時に非ゼロの値に評価された場合にのみ、プログラム・コードが並行して実行されます。 if 節は 1 つのみ指定することができます。

private (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内のデータ変数は、コンマで区切られています。

firstprivate (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。それぞれの新規の private オブジェクトは、あたかもステートメント・ブロック内に暗黙の宣言があるように、元の変数の値を使用して初期化されます。 *list* 内のデータ変数は、コンマで区切られています。

num_threads (*int_exp*)
int_exp の値は、並行領域に使用するスレッドの数を指定する整数式です。スレッドの数の動的調整も使用可能になっている場合は、 *int_exp* は使用されるスレッドの最大数を指定します。

shared (*list*)
list 内のコンマで区切られたデータ変数のスコープがすべてのスレッドの間で共有されることを宣言します。

default (shared | none)
各スレッド内の変数のデフォルトのデータ・スコープを定義します。 **default** 節は、 1 つの **omp parallel** ディレクティブ上に 1 つのみ指定することができます。

default(shared) の指定は、 **shared**(*list*) 節内の各変数を指定するのと同じです。

default(none) の指定には、並列化されたステートメント・ブロックに対して可視である各データ変数が、データ・スコープ節に明示的にリストされていることが必要です。ただし、次のような変数の例外があります。
- const によって限定されている
- 囲まれたデータ・スコープ属性の文節内に指定されている
- 対応する **omp for** または **omp parallel for** ディレクティブによってのみ参照されるループ制御変数として使用されている

copyin (*list*)
list 内に指定されているデータ変数ごとに、マスター・スレッド内のデータ変数の値は、並列領域の開始地点のスレッド private コピーにコピーされます。 *list* 内のデータ変数は、コンマで区切られています。

copyin 節内に指定する各データ変数は、**threadprivate** 変数でなければなりません。

reduction (*operator*: *list*)
指定された *operator* を使用して、 *list* 内のすべてのスカラー変数の縮約を実行します。 *list* 内の縮約変数は、コンマで区切られています。

list 内の各変数の private コピーは、スレッドごとに作成されます。ステートメント・ブロックの最後で、縮約変数のすべての private コピーの最終値は、その演算子に適切な方法で結合され、その結果は、共有の縮約変数の元の値に再格納されます。例えば、 max 演算子を指定した場合は、以下の式を使用して、元の縮約変数値が、private コピーの最終値に結合されます。

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

reduction 節で指定される変数は、以下の条件を満たす必要があります。

- 演算子に適切な型でなければならない。 max 演算子または min 演算子を指定した場合、変数は、long、short、signed、または unsigned の有無を問わず、以下の型のいずれかでなければなりません。
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- 囲んでいるコンテキスト内で共有されていなければならない。
- const によって修飾された変数であってはならない。
- ポインター型があってはならない。

使用法

並列領域が検出されると、スレッドの論理チームが形成されます。チーム内の各スレッドは、作業共有構成を除いて、並列領域内のすべてのステートメントを実行します。作業共有構成内の作業は、チーム内のスレッド間で配布されます。

ループの繰り返しが独立していなければ、ループを並列化することはできません。暗黙のバリアが、並列化されたステートメント・ブロックの終了地点にあります。

デフォルトで、ネストされた並列領域は直列化されています。

関連情報:

36 ページの『OMP_NESTED』

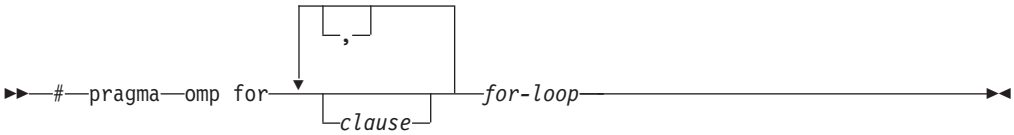
38 ページの『OMP_PROC_BIND』

#pragma omp for

目的

omp for ディレクティブは、この作業共有構成を検出するスレッドのチーム内でループの繰り返しを分配するようコンパイラーに命令します。

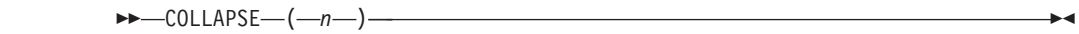
構文



パラメーター

clause は、以下の節のいずれかです。

collapse (*n*)
ネストされた並列処理を導入せずにネスト内の複数ループの並列処理を行うことができます。



- ワーク・シェアリング **for** または **parallel for** プラグマで使用できる collapse 文節は 1 つだけです。
- 指定された数のループが字句として実在していなければなりません。すなわち、これらのどのループも呼び出されたサブルーチン内にあってはなりません。
- これらのループは長方形の繰り返しスペースを形成していなければならず、各ループの境界とストライドはすべてループ不変でなければなりません。
- ループ指標が異なるサイズのものである場合、縮小されたループには最大サイズの指標が使用されます。
- ループは完全にネストされている必要があります。すなわち、縮小されるループ同士の間にいかなるコードも OpenMP プラグマも挿入されてはなりません。
- 関連する DO ループは構造化ブロックであることが必要です。これらの実行が **break** ステートメントによって終了されてはなりません。
- 複数のループがループ構成体と関連付けられている場合は、最も内側の関連ループの繰り返しのみが、**continue** ステートメントによって省略できます。複数のループがループ構成体と関連付けられている場合は、最も内側の関連ループの場合を除き、どのループ終了ステートメントへの分岐も存在してはなりません。

ordered 構文

ループ領域内の 1 つのループまたはループ・ネストの繰り返しの実行中、実行スレッドは、同じループ領域にバインドされた複数の ordered 領域を実行してはなりません。この結果、複数のループが collapse 文節によってループ構成体に関連付けられている場合は、ordered 構文をすべての関連ループの内側に配置する必要があります。

lastprivate 文節

lastprivate 文節がワーク・シェアリング構文を識別するプラグマ上にある場合、関連ループで順序が最後の繰り返しからの各新規リスト項目の値は、collapse 文節がそのループに関連付けられている場合でも、元のリスト項目に割り当てられます。

その他の SMP およびパフォーマンス・プラグマ

stream_unroll、**unroll**、**unrollandfuse**、**nounrollandfuse** プラグマは、**collapse** 文節のループ・ネストに関連付けられたどのループにも使用することはできません。

private (*list*)

list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内のデータ変数は、コンマで区切られています。

firstprivate (*list*)

list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。それぞれの新規の private オブジェクトは、ステートメント・ブロック内に暗黙の宣言がある場合のように初期化されます。 *list* 内のデータ変数は、コンマで区切られています。

lastprivate (*list*)

list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内の各変数の最終値は、割り当てられる場合、最後の繰り返しでその変数に割り当てられる値となります。値が割り当てられていない変数は、不確定値を持っています。 *list* 内のデータ変数は、コンマで区切られています。

reduction (*operator*: *list*)

指定された *operator* を使用して、 *list* 内のすべてのスカラー変数の縮約を実行します。 *list* 内の縮約変数は、コンマで区切られています。

list 内の各変数の private コピーは、スレッドごとに作成されます。ステートメント・ブロックの最後で、縮約変数のすべての private コピーの最終値は、その演算子に適切な方法で結合され、その結果は、共有の縮約変数の元の値に再格納されます。例えば、 max 演算子を指定した場合は、以下の式を使用して、元の縮約変数値が、private コピーの最終値に結合されます。

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

reduction 節で指定される変数は、以下の条件を満たす必要があります。

- 演算子に適切な型でなければならない。 max 演算子または min 演算子を指定した場合、変数は、long、short、signed、または unsigned の有無を問わず、以下の型のいずれかでなければなりません。
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- 囲んでいるコンテキスト内で共有されていなければならない。
- const によって修飾された変数であってはならない。
- ポインター型があってはならない。

ordered

ordered 構文が **omp for** ディレクティブの動的範囲内に存在する場合、この文節を指定します。

schedule (*type*)

for ループの繰り返しを使用可能なスレッド間で分割する方法を指定します。 *type* の許容値は、以下のとおりです。

auto **auto** では、スケジューリングはコンパイラーとランタイム・システムに委任されます。コンパイラーとランタイム・システムは、実行可能なスレッドへの繰り返しのマッピングを任意に選択することができ (考えられるすべての有効なスケジュールを含みます)、別のループではそれらが異なっていることがあります。

dynamic

ループの繰り返しはサイズ **ceiling** (*number_of_iterations*/*number_of_threads*) のチャンクに分割されます。

チャンクは、「先着順実行」の基準ですべての作業が割り当てられるまで、アクティブ・スレッドに動的に割り当てられます。

dynamic,*n*
チャンクのサイズが *n* に設定されることを除いて、上記と同様です。 *n* は、1 以上の値の整数代入式でなければなりません。

guided チャンクは、デフォルトの最小チャンク・サイズに達するまで順次小さくされます。 最初のチャンクのサイズは **ceiling** (*number_of_iterations*/*number_of_threads*) です。それ以外のチャンクのサイズは、**ceiling**(*number_of_iterations_left*/*number_of_threads*) です。

チャンクの最小サイズは 1 です。

チャンクは、「先着順実行」の基準ですべての作業が割り当てられるまで、アクティブ・スレッドに割り当てられます。

guided,*n*
最小チャンク・サイズが *n* に設定されることを除いて、上記と同様です。 *n* は、1 以上の値の整数代入式でなければなりません。

runtime
スケジューリング方針は実行時に決定されます。 OMP_SCHEDULE 環境変数を使用して、スケジューリング・タイプおよびチャンク・サイズを設定します。

static ループの繰り返しはサイズ **ceiling** (*number_of_iterations*/*number_of_threads*) のチャンクに分割されます。スレッドにはそれぞれ別個のチャンクが割り当てられます。

このスケジューリング方針は、*ブロック・スケジューリング* とも呼ばれます。

static,*n* ループの繰り返しがサイズ *n* のチャンクに分割されます。チャンクはそれぞれラウンドロビン 方式でスレッドに割り当てられます。

n は、1 以上の値の整数代入式でなければなりません。

このスケジューリング方針は、*ブロック巡回スケジューリング* とも呼ばれます。

注: *n* = 1 の場合、ループの繰り返しは 1 のチャンク・サイズに分割されます。そして各チャンクはラウンドロビン 方式でスレッドに割り当てられます。このスケジューリング方針は、*ブロック巡回スケジューリング* とも呼ばれます。

nowait
この文節は、**for** ディレクティブの終わりにある暗黙の**バリア**を回避するために使用します。これは、指定した並列領域内に複数の独立した作業共有セクションまたは繰り返しループがある場合に有効です。 **nowait** 節は、1 つの **for** ディレクティブに 1 回しか現れることができません。

また、*for_loop* の個所は、以下の規範的形状を持つ `for` ループ構造体です。

```
for (init_expr; exit_cond; incr_expr)
 statement
```

ここで、

init_expr 形式:

```
iv = b
integer-type iv = b
```

exit_cond 形式:

```
iv <= ub
iv <  ub
iv >= ub
iv >  ub
```

incr_expr 形式:

```
++iv
iv++
--iv
iv--
iv += incr
iv -= incr
iv = iv + incr
iv = incr + iv
iv = iv - incr
```

また、ここでは以下のとおりです。

iv 繰り返し変数。繰り返し変数は、for ループ内のどこの箇所でも変更されない符号付き整数でなければなりません。繰り返し変数は、for 演算の間は、暗黙的に private にされます。 **lastprivate** として指定されていない場合、繰り返し変数は演算の完了後に不確定値を持つことになります。

b, *ub*, *incr* ループ・インバリアント符号付き整数式。これらの式を評価しているときは同期は実行されず、評価された副次作用が不確定値の結果になる可能性があります。

使用法

このプラグマは影響を受けるループまたはループ・ブロック・ディレクティブの直前に現れなければなりません。

omp for プラグマを使用するプログラム・セクションでは、いずれのスレッドが特定の繰り返しを実行するかにかかわらず、正しい結果を生成できなければなりません。同様に、プログラムの正確さは、特定のスケジューリング・アルゴリズムの使用に依存するものであってはなりません。

for ループの繰り返し変数は、ループの実行の間、スコープ内で暗黙的に private にされます。この変数は、for ループの本体内で変更してはなりません。増分変数の値は、その変数がデータ・スコープの **lastprivate** を持つよう指定されていない限り、不確定です。

nowait 節が指定されていない限り、**for** ループの終わりに暗黙のバリアが存在します。

制約事項:

- for ループは構造化ブロックでなければならず、break 文で終了することはできません。
- ループ制御式の値は、ループのすべての繰り返しについて同一でなければなりません。
- **omp for** ディレクティブが受け入れることができる **schedule** 節は、1 つのみです。

- *n* の値 (チャンク・サイズ) は、並列領域のすべてのスレッドについて同一でなければなりません。

#pragma omp ordered

目的

omp ordered ディレクティブは、順次配列で実行されなければならないコードの構造化ブロックを識別します。

構文

使用法

omp ordered ディレクティブは、以下のように使用しなければなりません。

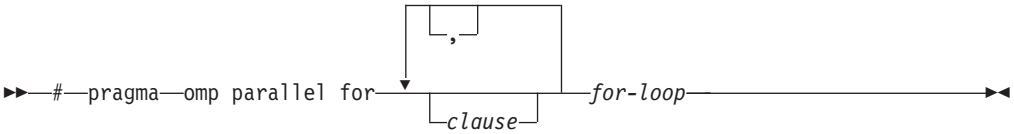
- **ordered** 節を含んでいる **omp for** または **omp parallel for** 構成の範囲内に表示されなければなりません。
- すぐ後に続くステートメント・ブロックに適用します。そのブロックのステートメントは、繰り返しが順次ループ内で実行されるのと同じ順序で実行されます。
- ループの繰り返しは、同一の **omp ordered** ディレクティブを 2 回以上実行してはなりません。
- ループの繰り返しでは、複数の特殊 **omp ordered** ディレクティブを実行してはなりません。

#pragma omp parallel for

目的

omp parallel for ディレクティブは、 **omp parallel** ディレクティブと **omp for** ディレクティブを効果的に結合します。このディレクティブを使用すると、単一の **for** ディレクティブを含んでいる並列領域をワンステップで定義することができます。

構文



使用法

nowait 節を除き、**omp parallel** および **omp for** ディレクティブに記述されている文節と制限は **omp parallel for** ディレクティブにも適用されます。

#pragma omp section, #pragma omp sections

目的

omp sections ディレクティブは、定義済みの並列領域にバインドされたスレッド間で作業を配布します。

構文

パラメーター

clause は、以下の節のいずれかです。

private (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内のデータ変数は、コンマで区切られています。

firstprivate (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。それぞれの新規の private オブジェクトは、ステートメント・ブロック内に暗黙の宣言がある場合のように初期化されます。 *list* 内のデータ変数は、コンマで区切られています。

lastprivate (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内の各変数の最終値は、割り当てられる場合、最後の **section** でその変数に割り当てられる値となります。値が割り当てられていない変数は、不確定値を持っています。 *list* 内のデータ変数は、コンマで区切られています。

reduction (*operator*: *list*)
指定された *operator* を使用して、 *list* 内のすべてのスカラー変数の縮約を実行します。 *list* 内の縮約変数は、コンマで区切られています。

list 内の各変数の private コピーは、スレッドごとに作成されます。ステートメント・ブロックの最後で、縮約変数のすべての private コピーの最終値は、その演算子に適切な方法で結合され、その結果は、共有の縮約変数の元の値に再格納されます。例えば、 max 演算子を指定した場合は、以下の式を使用して、元の縮約変数値が、private コピーの最終値に結合されます。

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

reduction 節で指定される変数は、以下の条件を満たす必要があります。

- 演算子に適切な型でなければならない。 max 演算子または min 演算子を指定した場合、変数は、long、short、signed、または unsigned の有無を問わず、以下の型のいずれかでなければなりません。
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int

  - float
  - double
- 囲んでいるコンテキスト内で共有されていなければならない。
- const によって修飾された変数であってはならない。
- ポインター型があってはならない。

nowait
この文節は、**sections** ディレクティブの終わりにある暗黙の**バリア**を回避するために使用します。これは、指定した並列領域内の複数の独立した作業共有セクションがある場合に有効です。 **nowait** 節が、所定の **sections** ディレクティブ上に現れるのは、1 回のみです。

使用法

omp section ディレクティブは、 **omp sections** ディレクティブの中にある最初のプログラム・コードのセグメントのためのオプションです。後に続くセグメントは、その前に **omp section** ディレクティブがなければなりません。すべての **omp section** ディレクティブは、 **omp sections** ディレクティブに関連したプログラム・ソース・コードのセグメントの字句構成内になければなりません。

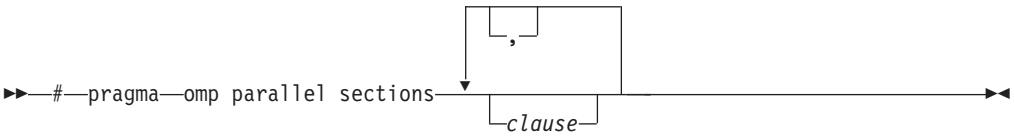
プログラム実行が **omp sections** ディレクティブに到達すると、後続の **omp section** ディレクティブによって定義されたプログラム・セグメントは、並列実行のために使用可能なスレッド間で配布されます。 **nowait** 節が指定されていない限り、バリアは **omp sections** ディレクティブに関連した、より広いプログラム領域の終了地点に暗黙的に定義されます。

#pragma omp parallel sections

目的

omp parallel sections ディレクティブは、 **omp parallel** ディレクティブと **omp sections** ディレクティブを効果的に結合します。このディレクティブを使用すると、単一の **sections** ディレクティブを含んでいる並列領域をワンステップで定義することができます。

構文



使用法

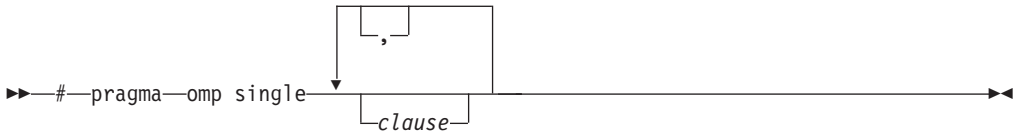
omp parallel および **omp sections** ディレクティブに記載されているすべての文節および制限は、 **omp parallel sections** ディレクティブに適用されます。

#pragma omp single

目的

omp single ディレクティブは、単一の使用可能スレッドで実行しなければならないコードのセクションを識別します。

構文



パラメーター

clause は、次のいずれかです。

private (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内のデータ変数は、コンマで区切られています。

private 節の変数は、同じ **omp single** ディレクティブに対して、**copyprivate** 節にも現れることはできません。

copyprivate (*list*)
list に指定された変数の値を、チームの 1 人のメンバーから他のメンバーにブロードキャストします。これは、**omp single** ディレクティブと関連した構造化ブロックの実行後で、なおかつスレッドが構造体の終わりにあるバリアから去る前に発生します。チーム内の他のすべてのスレッドについては、*list* 内の各変数が、その構造化されたブロックを実行したスレッド内の対応する変数の値を使用して定義されるようになります。 *list* 内のデータ変数は、コンマで区切られています。この文節の使用上の制約事項は以下の通りです。

- **copyprivate** 節の変数は、同じ **omp single** ディレクティブに対する **private** 節または **firstprivate** 節にも現れることはできません。
- **copyprivate** 節を持つ **omp single** ディレクティブが並列領域の動的エクステントで検出された場合、**copyprivate** 節に指定されたすべての変数はエンクロージング・コンテキストの中で private でなければなりません。
- 並列領域内の動的エクステント内の **copyprivate** 節に指定された変数は、囲んでいるコンテキストの中で private でなければなりません。
- **copyprivate** 節に指定される変数は、アクセス可能であり、あいまいでないコピー代入演算子を持っていなければなりません。
- **copyprivate** 節は、**nowait** 節と一緒に使用することはできません。

firstprivate (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。それぞれの新規の private オブジェクトは、ステートメント・ブロック内に暗黙の宣言がある場合のように初期化されます。 *list* 内のデータ変数は、コンマで区切られています。

firstprivate 節の変数は、同じ **omp single** ディレクティブに対して、**copyprivate** 節にも現れることはできません。

nowait
この文節は、**single** ディレクティブの終わりにある暗黙の**バリア**を回避するために使用します。**nowait** 節が、所定の **single** ディレクティブ上に現れるのは、1 回のみです。 **nowait** 節は、**copyprivate** 節と一緒に使用することはできません。

使用法

nowait 節が指定されていない限り、暗黙のバリアが並列化されたステートメント・ブロックの最後にあります。

#pragma omp master

目的

omp master ディレクティブは、マスター・スレッドによってのみ実行されなければならないコードのセクションを識別します。

構文

使用法

マスター・スレッド以外のスレッドは、この構成に関連したステートメント・ブロックを実行しません。

暗黙のバリアは、マスター・セクションの出入り口には存在しません。

#pragma omp critical

目的

omp critical ディレクティブは、単一スレッドによって一度に実行されなければならないコードのセクションを識別します。

構文

ここで、*name* は、オプションで棄却域を識別するために使用することができます。棄却域を命名する ID には外部リンケージがあり、通常の ID が使用する名前空間とは異なる名前空間を占めます。

使用法

スレッドはプログラム内の他のスレッドが同じ名前で棄却域を実行しなくなるまで、特定の名前で識別される棄却域の開始時点で待機します。 **omp critical** ディレ

クティブ呼び出しによって特に命名されてはいないクリティカル・セクションは、指定されていない同じ名前にマップされます。

#pragma omp barrier

目的

omp barrier ディレクティブは、チーム内の他のすべてのスレッドが領域内の明示的なタスクをすべて完了するまで、並列領域のスレッドが **omp barrier** を超えては実行されない同期点を識別します。

構文

使用法

omp barrier ディレクティブは、 1 つのブロック内、または複合ステートメント内に現れなければなりません。例を以下に示します。

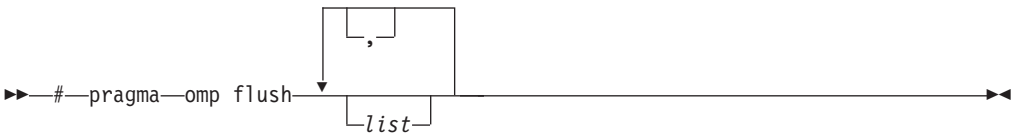
```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}

if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#pragma omp flush

目的

omp flush ディレクティブは、並列領域内のすべてのスレッドがメモリー内で指定されたオブジェクトの同じビューを持っていることをコンパイラーが保証するポイントを識別します。

構文



ここで、*list* は、同期化される変数のコンマ区切りのリストです。

使用法

list にポインターが含まれる場合、ポインターに参照されているオブジェクトではなく、ポインターがフラッシュされます。*list* が指定されていない場合は、自動ストレージ期間にアクセス不能なオブジェクトを除くすべての共有オブジェクトが同期化されます。

暗黙の **flush** ディレクティブは、以下のディレクティブとともに現れます。

- **omp barrier**
- **omp critical** の出入り口。

- **omp parallel** からの出口。
- **omp for** からの出口。
- **omp sections** からの出口。
- **omp single** からの出口。

omp flush ディレクティブは、1 つのブロック内、または複合ステートメント内に現れなければなりません。例を以下に示します。

```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#pragma omp threadprivate

目的

omp threadprivate ディレクティブは、指定されたファイル・スコープ、名前空間スコープ、または静的ブロック・スコープ変数を 1 つのスレッド専用にします。

構文

ここで、*identifier* ファイル・スコープ、名前空間スコープ、または静的ブロック・スコープ変数です。

使用法

omp threadprivate データ変数の各コピーは、そのコピーを最初に使用する前に一度初期化されます。 **threadprivate** データ変数を初期化するために使用される前にオブジェクトが変更された場合、振る舞いは指定されていません。

スレッドは、別のスレッドの **omp threadprivate** データ変数のコピーを参照することはできません。プログラムの直列領域およびマスター領域の実行時に、参照は常にデータ変数のマスター・スレッドのコピーに対して行われます。

omp threadprivate ディレクティブの使用は、以下の点で管理されています。

- **omp threadprivate** ディレクティブは、すべての定義および宣言外のファイル・スコープになければならない。
- **omp threadprivate** ディレクティブは静的ブロック・スコープ変数に適用され、ブロック・スコープ変数を参照する字句ブロックに現れる場合がある。このディレクティブは、ネストされたスコープではなく、変数のスコープに現れなければならず、そのリスト内の変数に対するすべての参照より前に指定される必要があります。
- データ変数は、**omp threadprivate** ディレクティブの *list* に組み込む前に、ファイル・スコープで宣言しなければならない。

- **omp threadprivate** ディレクティブとその *list* は、字句的にその *list* 内にあるデータ変数への参照の前になければならない。
- ある変換単位で **omp threadprivate** ディレクティブに指定しているデータ変数は、その変数が宣言されている他のすべての変換単位でも同様に指定しておかなければならない。
- **omp threadprivate** *list* に指定されるデータ変数は、**copyin**、**copyprivate**、**if**、**num_threads**、および **schedule** 節以外の文節に現れることはできない。
- **omp threadprivate** *list* 内のデータ変数のアドレスは、アドレス定数ではない。
- **omp threadprivate** *list* で指定しているデータ変数には、不完全型または参照型があってはならない。

#pragma omp task

目的

task プラグマを使用して、タスクを明示的に定義することができます。

task プラグマは、タスク領域外のコードと並行して実行するコード・ブロックを識別するときに使用します。 **task** プラグマは、ポインター追跡または再帰的アルゴリズムなどの不規則アルゴリズムを並列処理する際に役立つ場合があります。**task** ディレクティブは、**-qsmp** コンパイラー・オプションを指定した場合にのみ有効になります。

構文

パラメーター

節 パラメーターは、以下のいずれかのタイプの節になります。

default (shared | none)

各タスク内の変数のデフォルトのデータ・スコープを定義します。default 節は、 1 つの omp task ディレクティブ上に 1 つのみ指定することができます。

default(shared) の指定は、 shared(*list*) 節内の各変数を指定するのと同じです。

default(none) を指定するには、以下の特性を持つ変数を除き、構文に対して可視である各データ変数が、データ・スコープ節に明示的にリストされている必要があります。

- threadprivate
- 自動かつ構文内のスコープで宣言されている
- 動的なストレージ期間を持つオブジェクト
- 静的データ・メンバー
- 作業共有 **for** または **parallel for** 構文の関連 for ループにおけるループ繰り返し変数
- 静的かつ構文内のスコープで宣言されている

final (*exp*)
`final` 節を指定し、*exp* がゼロ以外の値に評価される場合、生成されるタスクは final タスクです。final タスクの内側でタスク構文が検出された場合は、必ず final な包含タスク (included task)が作成されます。

task プラグマで指定できる `final` 節は 1 つだけです。

firstprivate (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。それぞれの新規の private オブジェクトは、あたかもステートメント・ブロック内に暗黙の宣言があるように、元の変数の値を使用して初期化されます。 *list* 内のデータ変数は、コンマで区切られています。

if (*exp*)
`if` 節を指定すると、スカラー式 *exp* がゼロ以外の値に評価された場合に、非据え置きタスク (undeferred task) が生成されます。`if` 節は 1 つのみ指定することができます。

mergeable
`mergeable` 節を指定すると、生成されるタスクが非据え置きタスク (undeferred task) または包含タスク (included task)である場合に、マージ・タスク (merged task) が生成される場合があります。

private (*list*)
list 内のデータ変数のスコープが各スレッドに対して private であることを宣言します。 *list* 内のデータ変数は、コンマで区切られています。

shared (*list*)
list 内のコンマで区切られたデータ変数のスコープがすべてのスレッドの間で共有されることを宣言します。

untied
タスク領域が中断状態の場合、untied タスクをチーム内の任意のスレッドによって再開することができます。タスク構文で `untied` 節を指定しても、以下のいずれかの条件がゼロ以外の値である場合は無視されます。

- 同じタスク構文で `final` 節が指定されており、その `final` 節の式がゼロ以外の値に評価される。
- タスクが包含タスク (included task)である。

使用法

final タスクでは、すべての子タスクが final な包含タスク (included task) になります。以下の条件のいずれかがゼロ以外の値である場合は、final タスクが生成されます。

- タスク構文で `final` 節が指定されており、その `final` 節の式がゼロ以外の値に評価される。
- 生成されるタスクが final タスクの子タスクである。

非据え置きタスク (undeferred task) は、生成側タスク領域について実行が据え置かれないタスクです。つまり、非据え置きタスク (undeferred task) の実行が完了するまで、生成側タスク領域は中断されます。タスク構文で `if` 節が指定されており、その `if` 節の式がゼロに評価される場合は、非据え置きタスク (undeferred task) が生成されます。

包含タスク (included task) は、実行が生成側タスク領域に順次組み込まれるタスクです。つまり、包含タスク (included task) は据え置かれず、検出側スレッドによって直ちに実行されます。生成されるタスクが final タスクの子タスクである場合は、包含タスク (included task) が生成されます。

マージ・タスク (merged task) は、生成側タスク領域と同じデータ環境を持つタスクです。マージ・タスク (merged task) は、以下の両方の条件がゼロ以外の値であるときに生成される場合があります。

- タスク構文で `mergeable` 節が指定されている。
- 生成されるタスクが非据え置きタスク (undeferred task) または包含タスク (included task) である。

`if` 節の式および `final` 節の式はタスク構文の外部で評価され、評価順序は規定されていません。

関連資料:

『#pragma omp taskwait』

#pragma omp taskyield

目的

omp taskyield プラグマは、現在のタスクを中断して別のタスクを実行するようにコンパイラーに指示します。**taskyield** 領域には、現在のタスク領域の明示的タスク・スケジューリング・ポイントが入ります。

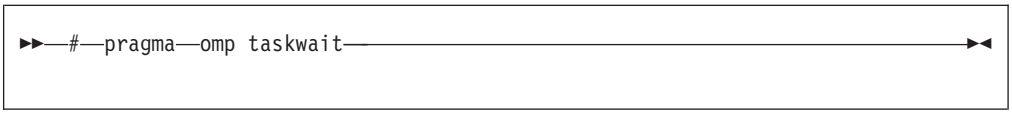
構文

#pragma omp taskwait

目的

taskwait プラグマを使用して、現在のタスクによって生成された、これから完了する子タスクに *wait* を指定します。

構文



関連資料:

477 ページの『#pragma omp task』

第 6 章 コンパイラーの事前定義マクロ

事前定義マクロを使用すると、特定コンパイラー、コンパイラーの特定バージョン、特定環境および特定言語機能のコード、またはこれらのいずれかのコードを条件付きでコンパイルできます。

事前定義マクロには複数のカテゴリーがあります。

- 『汎用マクロ』
- 483 ページの『プラットフォームに関連したマクロ』
- 485 ページの『コンパイラー機能に関連したマクロ』

496 ページの『事前定義マクロの例』では、コード内での事前定義マクロの使用方法を示します。

汎用マクロ

以下の事前定義マクロは、常にコンパイラーによって事前定義されます。特に断りがなければ、以下のすべてのマクロは*保護* されています。これは、マクロの定義解除または再定義を行おうとした場合にコンパイラーが警告を発行することを意味します。

表 45. 汎用の事前定義マクロ

事前定義マクロ名	説明	事前定義値
__BASE_FILE__	基本ソース・ファイルの名前を示します。	基本ソース・ファイルの完全修飾ファイル名です。

__COUNTER__	0 から始まる整数に展開します。値はこのマクロが展開されるたびに 1 ずつ増えます。
	
	このマクロを ## 演算子と一緒に使用して、固有変数または関数名を生成できます。以下の例は、単一トークンでの個別 ID の宣言を示しています。
	`#define CONCAT(a, b) a##b`
	`#define CONCAT_VAR(a, b) CONCAT(a, b)`
	`#define VAR CONCAT_VAR(var, __COUNTER__)`
	
	`//Equivalent to int var0 = 1;`
	`int VAR = 1;`
	
	`//Equivalent to char var1 = 'a';`
	`char VAR = 'a';`
		0 から始まる整数変数。値はこのマクロが展開されるたびに 1 ずつ増えます。

__DATE__	ソース・ファイルがプリプロセスされた日付を示します。	ソース・ファイルがプリプロセスされた日付を含む文字ストリング。
__FILE__	プリプロセスされたソース・ファイルの名前を示します。	プリプロセスされたソース・ファイルの名前を含む文字ストリング。
__FUNCTION__	現在コンパイル中の関数の名前を示します。	現在コンパイル中の関数の名前を含む文字ストリングです。

表 45. 汎用の事前定義マクロ (続き)

事前定義マクロ名	説明	事前定義値
__LINE__	ソース・ファイルの現在行番号を示します。	ソース・ファイルの行番号を含む整数定数。
__SIZE_TYPE__	現行のプラットフォームでの基礎となる型 size_t を示します。保護されていません。	unsigned int (32 ビット・コンパイル・モードの場合)。unsigned long (64 ビット・コンパイル・モードの場合)。
__TIME__	ソース・ファイルがプリプロセスされた時刻を示します。	ソース・ファイルがプリプロセスされた時刻を含む文字ストリング。
__TIMESTAMP__	ソース・ファイルの最終変更日時を示します。コンパイラーが、ソース・プログラムの一部である組み込みファイルを処理すると値が変わります。	"*Day Mmm dd hh:mm:ss yyyy*" というフォームの文字ストリング・リテラルです。以下で各部分を説明します。 *Day* 曜日 (Mon、Tue、Wed、Thu、Fri、Sat、または Sun) を示します。 *Mmm* 月を省略形 (Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、または Dec) で示します。 *dd* 日を示します。日が 10 より小さければ、最初の d がブランク文字になります。 *hh* 時間を示します。 *mm* 分を示します。 *ss* 秒を示します。 *yyyy* 年を示します。

XL C/C++ コンパイラー製品を示すマクロ

XL C/C++ コンパイラーに関連するマクロは、常に事前定義および保護 (マクロの定義解除または再定義を行おうとした場合に警告が発行される) されています。**-qshowmacros=pre -E** コンパイラー・オプションを使用して、事前定義マクロの値を表示できます。

表 46. コンパイラー製品の事前定義マクロ

事前定義マクロ名	説明	事前定義値
C __IBMC__	XL C コンパイラーのレベルを示します。	フォーマットが *VRM* の整数です。以下で各部分を説明します。 *V* バージョン番号を示します。 *R* リリース番号を示します。 *M* モディフィケーション番号を示します。

表 46. コンパイラー製品の事前定義マクロ (続き)

事前定義マクロ名	説明	事前定義値
C++ __IBMCPP__	XL C++ コンパイラーのレベルを示します。	フォーマットが *VRM* の整数です。以下で各部分を説明します。 *V* バージョン番号を示します。 *R* リリース番号を示します。 *M* モディフィケーション番号を示します。
C __xlc__	XL C コンパイラーのレベルを示します。	フォーマットが「*V.R.M.F*」のストリングです。以下で各部分を説明します。 *V* バージョン番号を示します。 *R* リリース番号を示します。 *M* モディフィケーション番号を示します。 *F* 定着レベルを示します。
__xlC__	XL C コンパイラーおよび XL C++ コンパイラーの VR レベルを 16 進形式で示します。XL C コンパイラーを使用しても、このマクロを自動的に定義できます。	フォーマットが 0x*VVRR* の 4 桁の 16 進整数です。以下で各部分を説明します。 *V* バージョン番号を示します。 *R* リリース番号を示します。
__xlC_ver__	XL C コンパイラーおよび XL C++ コンパイラーの MF レベルを 16 進形式で示します。XL C コンパイラーを使用しても、このマクロを自動的に定義できます。	フォーマットが 0x*0000MMFF* の 8 桁の 16 進整数です。以下で各部分を説明します。 *M* モディフィケーション番号を示します。 *F* 定着レベルを示します。 例えば、PTF 3 では、マクロの値は 0x00000003 です。

プラットフォームに関連したマクロ

プラットフォーム間でのアプリケーションの移植を可能にするために以下の事前定義マクロが提供されています。すべてのプラットフォーム関連の事前定義マクロは無保護であるため、特に指定されていない限り、警告を受けずにマクロの定義を解除または再定義できます。

表 47. プラットフォーム関連の事前定義マクロ

事前定義マクロ名	説明	事前定義値	事前定義の条件
_BIG_ENDIAN、 __BIG_ENDIAN__	プラットフォームがビッグ・エンディアン (最上位バイトが、最も低いアドレスのメモリー・ロケーションに格納される) であることを示します。	1	常に事前定義されています。
__ELF__	ELF オブジェクト・モデルが有効であることを示します。	1	Linux プラットフォームの場合は常に事前定義されています。

表 47. プラットフォーム関連の事前定義マクロ (続き)

事前定義マクロ名	説明	事前定義値	事前定義の条件
C++ __GXX_WEAK__	弱いシンボルがサポートされていることを示します (リンカーによって、テンプレートのインスタンス化に使用されます)。	1	常に事前定義されています。
__HOS_LINUX__	ホスト・オペレーティング・システムが Linux であることを示します。保護されています。	1	常にすべての Linux プラットフォームに対して事前定義されています。
_ILP32, __ILP32__	ターゲット・プラットフォームで int、long int、およびポインター型に 32 ビットが使用されていることを示します。	1	ターゲット・プラットフォームで int、long int、およびポインター型に 32 ビットが使用される場合に事前定義されます。
__linux、 __linux__	プラットフォームが Linux であることを示します。	1	常にすべての Linux プラットフォームに対して事前定義されています。
_LP64, __LP64__	ターゲット・プラットフォームで、long int およびポインター型に 64 ビットが使用され、int 型に 32 ビットが使用されていることを示します。	1	ターゲット・プラットフォームで、long int およびポインター型に 64 ビットが使用され、int 型に 32 ビットが使用されている場合に事前定義されます。
__powerpc、 __powerpc__	ターゲットが Power アーキテクチャー であることを示します。	1	ターゲットが Power アーキテクチャーである場合に事前定義されます。
__powerpc64__	ターゲットが Power アーキテクチャー で、64 ビットのコンパイル・モードが有効になっていることを示します。	1	ターゲットが Power アーキテクチャーであり、**-q64** が有効である場合に事前定義されます。
__PPC, __PPC__	ターゲットが Power アーキテクチャー であることを示します。	1	ターゲットが Power アーキテクチャーである場合に事前定義されます。
__PPC64__	ターゲットが Power アーキテクチャー で、64 ビットのコンパイル・モードが有効になっていることを示します。	1	ターゲットが Power アーキテクチャーであり、**-q64** が有効である場合に事前定義されます。

表 47. プラットフォーム関連の事前定義マクロ (続き)

事前定義マクロ名	説明	事前定義値	事前定義の条件
__TOS_LINUX__	ターゲット・オペレーティング・システムが Linux であることを示します。	1	ターゲットが Power アーキテクチャーである場合に事前定義されます。
__unix、 __unix__	オペレーティング・システムは、 UNIX の一種であることを示します。	1	常に事前定義されています。
__IBM_CONFIG_GNUC__	XL C/C++ の構成に使用された GCC コンパイラーのメジャー・バージョンを示します。	構成ファイルで指定された __GNUC__ の値。詳しくは、『カスタム・コンパイラー構成ファイルの使用』を参照してください。	常にすべての Linux プラットフォームに対して事前定義されています。
__IBM_CONFIG_GNUC_MINOR__	XL C/C++ の構成に使用された GCC コンパイラーのマイナー・バージョンを示します。	構成ファイルで指定された __GNUC_MINOR__ の値。詳しくは、『カスタム・コンパイラー構成ファイルの使用』を参照してください。	常にすべての Linux プラットフォームに対して事前定義されています。
__IBM_CONFIG_GNUC_PATCHLEVEL__	XL C/C++ の構成に使用された GCC コンパイラーのパッチ・レベルを示します。	構成ファイルで指定された __GNUC_PATCHLEVEL__ の値。詳しくは、『カスタム・コンパイラー構成ファイルの使用』を参照してください。	常にすべての Linux プラットフォームに対して事前定義されています。

コンパイラー機能に関連したマクロ

機能関連のマクロは、特定のコンパイラー・オプションまたはプラグマの設定に応じて事前定義されます。特に断りがなければ、すべての機能関連のマクロは保護されています (マクロの定義解除または再定義を行おうとした場合、コンパイラーは警告を発行します)。

機能関連のマクロについては、以下のセクションで説明します。

- 486 ページの『コンパイラー・オプション設定に関連したマクロ』
- 488 ページの『アーキテクチャー設定に関連したマクロ』
- 489 ページの『言語レベルに関連したマクロ』

コンパイラー・オプション設定に関連したマクロ

以下のマクロは、ソース入力特性、出力ファイル特性、および最適化などさまざまな機能についてテストすることができます。これらのマクロはすべて、特定のコンパイラー・オプションまたはサブオプション、あるいはそのサブオプションを暗黙指定する呼び出しまたはプラグマによって事前定義されます。機能を使用可能にするサブオプションが有効でない場合、そのマクロは未定義です。

表 48. 汎用のオプション関連の事前定義マクロ

事前定義マクロ名	説明	事前定義値	以下のコンパイラー・オプションまたは同等のプラグマが有効である場合に事前定義されます。
__ALTIVEC__	Vector データ型がサポートされていることを示します。 (保護されていない)	1	-qaltivec
__64BIT__	64 ビットのコンパイル・モードが有効であることを示します。	1	-q64
_CHAR_SIGNED, __CHAR_SIGNED__	デフォルトの文字タイプが `signed char` であることを示します。	1	-qchars=signed
_CHAR_UNSIGNED, __CHAR_UNSIGNED__	デフォルトの文字タイプが `unsigned char` であることを示します。	1	-qchars=unsigned
C++ __EXCEPTIONS	C++ 例外処理が有効になっていることを示します。	1	-qeh
__IBM_GCC_ASM	GCC インラインの `asm` ステートメントがサポートされていることを示します。	1	C **-qasm=gcc** および **-qlanglvl=extc99** | **extc89** | **extended** または **-qkeyword=asm** C++ **-qasm=gcc** および **-qlanglvl=extended**
		0	C **-qnoasm** および **-qlanglvl=extc99** | **extc89** | **extended** または **-qkeyword=asm** C++ **-qnoasm** および **-qlanglvl=extended**
C++ __IBM_STDCPP_ASM	GCC インラインの `asm` ステートメントのサポートが使用不可であることを示します。	0	**-qnoasm=stdcpp**

表 48. 汎用のオプション関連の事前定義マクロ (続き)

事前定義マクロ名	説明	事前定義値	以下のコンパイラー・オプションまたは同等のプラグマが有効である場合に事前定義されます。
C _IBMSMP	IBM SMP ディレクティブが認識されることを示します。	1	-qsmp
C++ __IGNERRNO__	システム呼び出しは `errno` を変更しないため、特定のコンパイラーの最適化が使用可能になることを示します。	1	-qignerrno
C++ __INITAUTO__	ソース・プログラムで明示的に初期設定されていない自動変数の値が、初期設定されることを示します。	**-qinitauto** コンパイラー・オプションに指定されている 2 桁の 16 進値。	-qinitauto= *16 進数*
C++ __INITAUTO_W__	ソース・プログラムで明示的に初期設定されていない自動変数の値が、初期設定されることを示します。	4 回繰り返された **-qinitauto** コンパイラー・オプションで指定された値に対応する 8 桁の 16 進数です。	-qinitauto= *16 進数*
C++ __LIBANSI__	C 標準ライブラリーの関数名に一致する関数への呼び出しは、実際は C ライブラリー関数であるため、特定のコンパイラーの最適化が使用できます。	1	-qlibansi
__LONGDOUBLE64	`long double` 型のサイズが 64 ビットであることを示します。	1	**-qnoldbl128**
__LONGDOUBLE128、__LONG_DOUBLE_128__	`long double` 型のサイズが 128 ビットであることを示します。	1	-qldbl128
__OPTIMIZE__	最適化レベルが有効であることを示します。	2	-O | -O2
		3	-O3 | -O4 | -O5
__OPTIMIZE_SIZE__	コード・サイズに対する最適化が有効であることを示します。	1	-O | -O2 | -O3 | -O4 | -O5 および -qcompact

表 48. 汎用のオプション関連の事前定義マクロ (続き)

事前定義マクロ名	説明	事前定義値	以下のコンパイラー・オプションまたは同等のプラグマが有効である場合に事前定義されます。
C++ __RTTI_DYNAMIC_CAST__	dynamic_cast 演算子のランタイム型 ID 情報が生成されることを示します。	1	-qrtti
C++ __RTTI_TYPE_INFO__	typeid 演算子のランタイム型 ID 情報が生成されることを示します。	1	-qrtti
C++ __NO_RTTI__	ランタイム型 ID 情報が無効になっていることを示します。	1	-qnortti
C++ __TEMPINC__	コンパイラーは、テンプレート関数を解決するテンプレート実装ファイル方式を使用していることを示します。	1	-qtempinc
__VEC__	Vector データ型がサポートされていることを示します。	10205	-qaltivec

アーキテクチャー設定に関連したマクロ

以下のマクロのターゲットのアーキテクチャー設定をテストできます。これらのマクロすべては、**-qarch** コンパイラー・オプション設定またはその設定を暗黙指定するその他のコンパイラー・オプションによって、値が 1 に事前定義されています。機能を使用可能にする **-qarch** サブオプションが有効でない場合、そのマクロは未定義です。

表 49. **-qarch** 関連のマクロ

マクロ名	説明	以下の -qarch サブオプションによって事前定義されています。
_ARCH_PPC	アプリケーションが、すべての Power プロセッサー上で実行するターゲットになっていることを示します。	auto を除く、すべての **-qarch** サブオプションに対して定義されます。
_ARCH_PPC64	アプリケーションが、64 ビットをサポートする Power プロセッサー上で実行するターゲットになっていることを示します。	ppc64 | ppc64gr | ppc64grsq | ppc64v | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PPCGR	アプリケーションが、グラフィックスをサポートする Power プロセッサー上で実行するターゲットになっていることを示します。	ppcgr | ppc64gr | ppc64grsq | ppc64v | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8

表 49. **-qarch** 関連のマクロ (続き)

マクロ名	説明	以下の -qarch サブオプションによって事前定義されています。
_ARCH_PPC64GR	アプリケーションが、64 ビットおよびグラフィックスをサポートする Power プロセッサー上で実行するターゲットになっていることを示します。	ppc64gr | ppc64v | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PPC64GRSQ	アプリケーションが、64 ビット、グラフィックス、および平方根をサポートする Power プロセッサー上で実行するターゲットになっていることを示します。	ppc64grsq | ppc64v | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PPC64V	アプリケーションが、64 ビットおよびベクトル処理をサポートする Power プロセッサー上で実行するターゲットになっていることを示します。	ppc64v | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR5	アプリケーションが、POWER5 プロセッサー上で実行するターゲットになっていることを示します。	pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR5X	アプリケーションが、POWER5+ プロセッサー上で実行するターゲットになっていることを示します。	pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR6	アプリケーションが、POWER6 プロセッサー上で実行するターゲットになっていることを示します。	pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR6E	アプリケーションが、POWER6 ロー・モードで実行される POWER6 プロセッサー上で実行するターゲットになっていることを示します。	pwr6e
_ARCH_PWR7	アプリケーションが、POWER7 または POWER7+ プロセッサー上で実行するターゲットになっていることを示します。	pwr7 | pwr8
_ARCH_PWR8	アプリケーションが、POWER8 プロセッサー上で実行するターゲットになっていることを示します。	pwr8

言語レベルに関連したマクロ

次のマクロは、C99 機能、GNU C または C++ に関連する機能、およびその他の IBM 言語拡張機能に関してテストすることができます。これらのマクロすべては、**-qlanglvl** コンパイラー・オプションのサブオプションで示される特定言語レベル、またはこのサブオプションを暗黙指定する呼び出しやプラグマによって、値が 1 に事前定義されています。 機能を使用可能にするサブオプションが有効でない場合、そのマクロは未定義です。これらのマクロに関連する機能の説明については、「*XL C/C++ ランゲージ・リファレンス*」を参照してください。

表 50. 言語機能の事前定義マクロ

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
C++ __BOOL__	bool キーワードが使用できることを示します。	**-qnokeyword=bool** が有効な場合を除き、常に定義されます。
C __C99_BOOL	_Bool データ型がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C++ __C99_COMPLEX	C99 複合タイプのサポートを使用可能にするか、C99 複合ヘッダーを組み込むことを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_CPLUSCMT	C++ 形式コメントがサポートされていることを示します。	extc1x | stdc99 | extc99 | stdc89 | extc89 | extended (および **-qcpluscmt**)
__C99_COMPOUND_LITERAL	複合リテラルがサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x
C __C99_DESIGNATED_INITIALIZER	指定された初期設定がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_DUP_TYPE_QUALIFIER	重複タイプの修飾子がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_EMPTY_MACRO_ARGUMENTS	空のマクロ引数がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_FLEXIBLE_ARRAY_MEMBER	柔軟な配列メンバーがサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
__C99__FUNC__	__func__ 事前定義 ID がサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x |c99__func__
__C99_HEX_FLOAT_CONST	16 進浮動小数点定数がサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x | c99hexfloat
C __C99_INLINE	インライン関数指定子がサポートされていることを示します。	extc1x | stdc99 | extc99 (および **-qkeyword=inline**)

表 50. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
__C99_LLONG	C99 型の long long データ型およびリテラルがサポートされていることを示します。	C extc1x | stdc99 | extc99 C++ extended0x | c99longlong
__C99_MACRO_WITH_VA_ARGS	変数引数を持つ関数のようなマクロがサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x | varargmacros
__C99_MAX_LINE_NUMBER	最大行番号が 2147483647 であることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended0x | c99preprocessor
C __C99_MIXED_DECL_AND_CODE	混用された宣言およびコードがサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
__C99_MIXED_STRING_CONCAT	幅の広いストリング・リテラルと幅の広くないストリング・リテラルの連結がサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended0x | c99preprocessor
C __C99_NON_LVALUE_ARRAY_SUB	配列に対する非左辺値の添え字がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_NON_CONST_AGGR_INITIALIZER	非定数の集合体初期化指定子がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
__C99_PRAGMA_OPERATOR	_Pragma 演算子がサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended extended0x
C __C99_REQUIRE_FUNC_DECL	暗黙的な関数宣言がサポートされていないことを示します。	stdc99
__C99_RESTRICT	C99 restrict 修飾子がサポートされていることを示します。	C extc1x | stdc99 | extc99 (および **-qkeyword=restrict**) C++ extended | extended0x (および **-qkeyword=restrict**)

表 50. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
C __C99_STATIC_ARRAY_SIZE	関数に対する配列パラメーターの `static` キーワードがサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_STD_PRAGMAS	標準プラグマがサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_TGMATH	tgmath.h の型総称マクロがサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
__C99_UCN	汎用文字名がサポートされていることを示します。	C extc1x | stdc99 | extc99 | ucs C++ ucs
C __C99_VAR_LEN_ARRAY	可変長配列がサポートされていることを示します。	extc1x | stdc99 | extc99 | extc89 | extended
C++ __C99_VARIABLE_LENGTH_ARRAY	可変長配列がサポートされていることを示します。	extended | extended0x | c99vla
__DIGRAPHS__	ダイグラフがサポートされていることを示します。	C extc1x | stdc99 | extc99 | extc89 | extended (および **-qdigraph**) C++ extended | extended0x | compat366 | strict98(および **-qdigraph**)
__EXTENDED__	言語拡張機能がサポートされていることを示します。	extended
C++ __IBM__ALIGN	`__align` 指定子がサポートされていることを指定します。	**-qnokeyword=__alignof** が指定されている場合を除き、常に定義されます。
__IBM_ALIGNOF__, __IBM__ALIGNOF__	`__alignof__` 演算子がサポートされていることを示します。	C extc1x | extc99 | extc89 | extended C++ extended
__IBM_ATTRIBUTES	型、変数、および関数属性がサポートされていることを示します。	C extc1x | extc99 | extc89 | extended C++ extended | extended0x
__IBM_COMPUTED_GOTO	計算された `goto` ステートメントがサポートされていることを意味します。	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_computedgoto

表 50. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
__IBM_EXTENSION_KEYWORD	__extension__ キーワードがサポートされていることを示します。	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | compat366 | strict98
C __IBM_GCC__INLINE__	GCC __inline__ 指定子がサポートされていることを示します。	extc1x | extc99 | extc89 | extended
C __IBM_DOLLAR_IN_ID	ID でドル記号がサポートされることを示します。	extc1x | extc99 | extc89 | extended
C __IBM_GENERALIZED_LVALUE	汎用左辺値がサポートされていることを示します。	extc1x | extc99 | extc89 | extended
__IBM_INCLUDE_NEXT	#include_next プリプロセス・ディレクティブがサポートされていることを示します。	C 常に定義されています。 C++ **-qlanglvl=nognu_include_next** が有効な場合を除き、常に定義されています。
__IBM_LABEL_VALUE	ラベルが値としてサポートされていることを示します。	C extc1x | extc99 | extc89 | extended C++ extended | extended0x |gnu_labelvalue
__IBM_LOCAL_LABEL	ローカル・ラベルがサポートされていることを示します。	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_locallabel
__IBM_MACRO_WITH_VA_ARGS	Variadic マクロ拡張がサポートされていることを示します。	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_varargmacros
C __IBM_NESTED_FUNCTION	ネストされた関数がサポートされていることを示します。	extc1x | extc99 | extc89 | extended
C __IBM_PP_PREDICATE	#assert、#unassert、#cpu、#machine、および #system プリプロセス・ディレクティブがサポートされていることを示します。	extc1x | extc99 | extc89 | extended

表 50. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
C __IBM_PP_WARNING	#warning プリプロセス・ディレクティブがサポートされていることを示します。	extc1x | extc99 | extc89 | extended
C __IBM_REGISTER_VARS	指定したレジスターで変数がサポートされていることを示します。	常に定義されています。
__IBM__TYPEOF__	__typeof__ または typeof キーワードがサポートされていることを示します。	C 常に定義されています。 C++ extended | extended0x (および **-qkeyword=typeof**)
__IBMC_COMPLEX_INIT	複素数型 float _Complex、double _Complex、および long double _Complex の初期化がサポートされていることを示します。	extc1x
__IBMC_GENERIC	汎用選択機能がサポートされていることを示します。	C extc89 | extc99 | extended | extc1x
__IBMC_NORETURN	_Noreturn 関数指定子がサポートされていることを示します。	C extc89 | extc99 | extended | extc1x C++ extended | extended0x | c1xnoreturn
C11 __IBMC_STATIC_ASSERT	静的アサーション機能がサポートされていることを示します。	extc89 | extc99 | extended | extc1x
C++11 __IBMCPP_AUTO_TYPEDEDUCTION	auto 型推定機能がサポートされていることを示します。	extended0x | autotypedededuction
C++11 __IBMCPP_C99_LONG_LONG	C99 long long 機能がサポートされていることを示します。	extended0x | c99longlong
C++11 __IBMCPP_C99_PREPROCESSOR	C++11 標準で採用された C99 プリプロセッサー機能がサポートされていることを示します。	extended0x | c99preprocessor
__IBMCPP_COMPLEX_INIT	複素数型 float _Complex、double _Complex、および long double _Complex の初期化がサポートされていることを示します。	extended

表 50. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
C++11 __IBMCPP_CONSTEXPR	一般化された定数式の機能のサポートを示します。	extended0x | constexpr
C++11 __IBMCPP_DECLTYPE	decltype 機能がサポートされていることを示します。	extended0x | decltype
C++11 __IBMCPP_DEFAULTED_AND_DELETED_FUNCTIONS	デフォルト関数および削除済み関数の機能がサポートされていることを示します。	extended0x | defaultanddelete
C++11 __IBMCPP_DELEGATING_CTORS	委任コンストラクター機能がサポートされていることを示します。	extended0x | delegatingctors
C++11 __IBMCPP_EXPLICIT_CONVERSION_OPERATORS	明示的な型変換演算子機能のサポートを示します。	extended0x | explicitconversionoperators
C++11 __IBMCPP_EXTENDED_FRIEND	拡張フレンド宣言機能がサポートされていることを示します。	extended0x | extendedfriend
C++11 __IBMCPP_EXTERN_TEMPLATE	明示的インスタンス生成宣言機能がサポートされていることを示します。	extended | extended0x | externtemplate
C++11 __IBMCPP_INLINE_NAMESPACE	インライン名前空間定義機能がサポートされていることを示します。	extended0x | inlinenamespace
C++11 __IBMCPP_NULLPTR	nullptr 機能がサポートされていることを示します。	extended0x | nullptr
C++11 __IBMCPP_REFERENCE_COLLAPSING	参照の縮約 (reference collapsing) 機能のサポートを示します。	extended0x | referencecollapsing
C++11 __IBMCPP_RIGHT_ANGLE_BRACKET	右不等号括弧機能のサポートを示します。	extended0x | rightanglebracket
C++11 __IBMCPP_RVALUE_REFERENCES	右辺値参照機能のサポートを示します。	extended0x | rvaluereferences
C++11 __IBMCPP_SCOPED_ENUM	スコープ付き列挙型機能のサポートを示します。	extended0x | scopedenum
C++11 __IBMCPP_STATIC_ASSERT	静的アサーション機能がサポートされていることを示します。	C++ extended0x | static_assert
C++11 __IBMCPP_VARIADIC_TEMPLATES	可変数引数テンプレート機能がサポートされていることを示します。	extended0x | variadic[templates]

表 50. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	以下の言語レベルが有効な場合に事前定義されます。
_LONG_LONG	long long データ型がサポートされていることを示します。	C extc1x | stdc99 | extc99 || stdc89 | extc89 | extended (および -qlonglong) C++ extended0x | c99longlong | extended (および **-qlonglong**)
__STDC__	コンパイラーが ANSI/ISO C 標準に準拠していることを示します。	C ANSI/ISO C 標準への準拠が有効であれば、1 に事前定義されています。 C++ 明示的に 0 に定義されます。
__STDC_HOSTED__	そのインプリメンテーションが ANSI/ISO C 標準のホストされたインプリメンテーションであることを示します。(すなわち、ホストされた環境には使用可能な標準 C の機能がすべて備わっているということです。)	C extc1x | stdc99 | extc99 C++ extended0x
C __STDC_VERSION__	コンパイラーが準拠している ANSI/ISO C 標準のバージョンを示します。	フォーマットは yyyymmL です。(例えば、C99 の場合のフォーマットは 199901L です。)

事前定義マクロの例

この例では、現行関数名が戻されるよう C99 __func__ ID の可用性をテストするための、__FUNCTION__ マクロおよび __C99__FUNC__ マクロの使用方法を示します。

```
#include <stdio.h>

#if defined(__C99__FUNC__)
#define PRINT_FUNC_NAME() printf (" In function %s ¥n", __func__);
#elif defined(__FUNCTION__)
#define PRINT_FUNC_NAME() printf (" In function %s ¥n", __FUNCTION__);
#else
#define PRINT_FUNC_NAME() printf (" Function name unavailable¥n");
#endif

void foo(void);

int main(int argc, char **argv)
{
   int k = 1;
   PRINT_FUNC_NAME();
   foo();
   return 0;
```

```
}

void foo (void)
{
   PRINT_FUNC_NAME();
   return;
}
```

この例の出力は以下のようになります。

```
In function main
In function foo
```

C++ これは、C++ プログラムで仮想関数とともに __FUNCTION__ マクロを使用した例です。

```
#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
   public: void func() { printf("In function %s ¥n", __FUNCTION__);}
};

int main() {
  Y aaa;
  aaa.func();
}
```

この例の出力は以下のようになります。

```
In function Y::func()
```

第 7 章 コンパイラー組み込み関数

組み込み関数は C および C++ のコーディング拡張であり、これによりプログラマーは C 関数呼び出しと C 変数の構文を使用してコンパイラー・マシンのプロセッサーの命令セットにアクセスすることができます。IBM Power アーキテクチャーには、高度に最適化されたアプリケーションの開発を可能にする特殊な命令があります。一部の Power 命令へのアクセスは、C および C++ 言語の標準構造体を使用して生成することができません。他の命令は標準の構造体を通じて生成できますが、組み込み関数を使用すると生成されたコードを正確に制御することができます。これらの命令を直接使用するインライン・アセンブリー言語プログラミングは、XL C/C++ V12.1 から完全にサポートされます。さらに、この手法はインプリメントに時間がかかる可能性があります。

XL C/C++ 組み込み関数はアセンブリー言語を通じてハードウェア・レジスターを管理する代わりに、最適化された Power 命令セットへのアクセスを提供して、コンパイラーが命令のスケジューリングを最適化できるようにします。

C++ C++ で XL C/C++ 組み込み関数を呼び出すには、ヘッダー・ファイル builtins.h をソース・コードに組み込む必要があります。 C++

以下のセクションには、Linux プラットフォームで使用可能な組み込み関数が記述されています。

- 『固定小数点組み込み関数』
- 508 ページの『2 進浮動小数点組み込み関数』
- 519 ページの『2 進コード化 10 進数組み込み関数』
- 523 ページの『同期およびアトミック組み込み関数』
- 531 ページの『キャッシュ関連組み込み関数』
- 549 ページの『暗号化組み込み関数』
- 555 ページの『ブロックに関連した組み込み関数』
- 639 ページの『各種組み込み関数』
- 645 ページの『並列処理のための組み込み関数』

固定小数点組み込み関数

固定小数点組み込み関数は以下のカテゴリーにグループ化されています。

- 500 ページの『絶対値関数』
- 500 ページの『表明関数』
- 501 ページの『ゼロ・カウント関数』
- 503 ページの『ロード関数』
- 504 ページの『乗算関数』
- 504 ページの『ポピュレーション・カウント関数』
- 505 ページの『回転関数』
- 507 ページの『保管関数』
- 507 ページの『トラップ関数』

絶対値関数

__labs、__llabs

目的

絶対値 long、絶対値 long long。

引数の絶対値を戻します。

プロトタイプ

```
signed long __labs (signed long);

signed long long __llabs (signed long long);
```

表明関数

__assert1、__assert2

目的

トラップ命令を生成します。

プロトタイプ

```
int __assert1(int, int, int);

void __assert2(int);
```

ビット置換関数

__bpermd

目的

バイト置換ダブルワード

ビット順列演算の結果を返します。

プロトタイプ

```
long long __bpermd (long long bit_selector, long long source);
```

使用法

8 ビットが返され、各ビットはソース内の 1 つのビットに対応します。これらのビットは、bit_selector のバイトによって選択されたものです。 bit_selector のバイト i が 64 より小さい場合、置換されたビット i は bit_selector のバイト i によって指定されたソースのビットにセットされます。それ以外の場合、置換されたビット i は 0 にセットされます。置換された各ビットは、結果値の最下位バイト内に配置され、残りのビットは 0 で埋められます。

-qarch が、64 ビット・モードの POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

比較関数

__cmpb

目的

バイトの比較

8 バイトの *source1* の各バイトを、*source2* の対応するバイトと比較します。*source1* のバイト *i* と *source2* のバイト *i* が等しければ、結果の対応するバイトに 0xFF が置かれ、等しくなければ、0x00 が置かれます。

プロトタイプ

```
long long __cmpb (long long source1, long long source2);
```

使用法

-qarch が、POWER6 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

ゼロ・カウント関数

__cntlz4、__cntlz8

目的

先行ゼロ・カウント、4 または 8 バイト整数。

プロトタイプ

```
unsigned int __cntlz4(unsigned int);

unsigned int __cntlz8(unsigned long long);
```

__cnttz4、__cnttz8

目的

後続ゼロ・カウント、4 または 8 バイト整数。

プロトタイプ

```
unsigned int __cnttz4(unsigned int);

unsigned int __cnttz8(unsigned long long);
```

除算関数

これらの除算関数は、**-qarch** が POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

__divde

目的

ダブルワード拡張除算

ダブルワード拡張除算の結果を返します。結果は、*dividend/divisor* に等しい値になります。

プロトタイプ

```
long long __divde (long long dividend, long long divisor);
```

使用法

-qarch が、64 ビット・モードの POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

注: 除算の結果が 32 ビットより大きいか、除数が 0 の場合、この関数の戻り値は未定義です。

__divdeu

目的

ダブルワード拡張符号なし除算

ダブルワード拡張符号なし除算の結果を返します。結果は、*dividend/divisor* に等しい値になります。

プロトタイプ

```
unsigned long long __divdeu (unsigned long long dividend, unsigned long long divisor);
```

使用法

-qarch が、64 ビット・モードの POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

注: 除算の結果が 32 ビットより大きいか、除数が 0 の場合、この関数の戻り値は未定義です。

__divwe

目的

ワード拡張除算

ワード拡張除算の結果を返します。結果は、*dividend/divisor* に等しい値になります。

プロトタイプ

```
int __divwe(int dividend, int divisor);
```

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

注: 除数が 0 の場合、この関数の戻り値は未定義です。

__divweu

目的

ワード拡張符号なし除算

ワード拡張符号なし除算の結果を返します。結果は、*dividend/divisor* に等しい値になります。

プロトタイプ

```
unsigned int __divweu(unsigned int dividend, unsigned int divisor);
```

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

注: 除数が 0 の場合、この関数の戻り値は未定義です。

ロード関数

__load2r、__load4r

目的

反転のハーフワード・バイトのロード、反転のワード・バイトのロード

プロトタイプ

```
unsigned short __load2r(unsigned short*);

unsigned int __load4r(unsigned int*);
```

__load8r

目的

バイト反転付きロード (8 バイト整数)

指定されたアドレスから 8 バイトのバイト反転ロードを実行します。

プロトタイプ

```
unsigned long long __load8r (unsigned long long * address);
```

使用法

-qarch が、64 ビット・モードの POWER7 以上のプロセッサーをターゲットにするように設定されている場合にのみ有効です。

乗算関数

__mulhd、__mulhdu

目的

上位の符号付きダブルワードの乗算、上位の符号なしダブルワードの乗算。

2 つのパラメーターの 128 ビットの積から、上位の 64 ビットを戻す。

プロトタイプ

```
long long int __mulhd ( long int, long int);

unsigned long long int __mulhdu (unsigned long int, unsigned long int);
```

使用法

64 ビット・モードでのみ有効。

__mulhw、__mulhwu

目的

上位の符号付きワードの乗算、上位の符号なしワードの乗算。

2 つのパラメーターの 64 ビットの積から、上位の 32 ビットを戻す。

プロトタイプ

```
int __mulhw (int, int);

unsigned int __mulhwu (unsigned int, unsigned int);
```

ポピュレーション・カウント関数

__popcnt4、__popcnt8

目的

ポピュレーション・カウント、4 または 8 バイト整数。

32 または 64 ビット整数のビット・セットの数を戻します。

プロトタイプ

```
int __popcnt4(unsigned int);

int __popcnt8(unsigned long long);
```

__popcntb

目的

ポピュレーション・カウント・バイト。

パラメーターの各バイトの 1 ビットをカウントし、そのカウントを結果の対応するバイトに入れます。

プロトタイプ

```
unsigned long __popcntb(unsigned long);
```

__poppar4、__poppar8

目的

ポピュレーション・パリティー、4 または 8 バイト整数。

32 または 64 ビット整数に設定されたビット数が偶数または奇数かどうかを検査する。

プロトタイプ

```
int __poppar4(unsigned int);

int __poppar8(unsigned long long);
```

戻り値

入力パラメーターに設定されたビット数が奇数の場合は 1 を戻す。それ以外は、0 を戻します。

回転関数

__rdlam

目的

ダブルワードを左方に回転し、マスクと AND 演算します。

rs の内容を左方に *shift* ビット回転させ、回転したデータをマスク と AND 演算する。

プロトタイプ

```
unsigned long long __rdlam(unsigned long long rs, unsigned int shift, unsigned long long mask);
```

パラメーター

mask
隣接するビット・フィールドを表す定数でなければなりません。

__rldimi、__rlwimi

目的

ダブルワードを左方に即時回転してから、挿入をマスクします。ワードを左方に即時回転してから、挿入をマスクします。

rs を左方に *shift* ビットだけ回転してから、 *rs* をビット・マスク *mask* の下の *is* に挿入します。

プロトタイプ

unsigned long long __rldimi(unsigned long long *rs*, unsigned long long *is*, unsigned int *shift*, unsigned long long *mask*);

unsigned int __rlwimi(unsigned int *rs*, unsigned int *is*, unsigned int *shift*, unsigned int *mask*);

パラメーター

shift
定数値 0 から 63 (`__rldimi`) または 31 (`__rlwimi`)。

mask
隣接するビット・フィールドを表す定数でなければなりません。

__rlwnm

目的

ワードを左方に回転し、マスクと AND 演算します。

rs を左方に *shift* ビットだけ回転し、ビット・マスク *mask* に *rs* を AND 演算します。

プロトタイプ

unsigned int __rlwnm(unsigned int *rs*, unsigned int *shift*, unsigned int *mask*);

パラメーター

mask
隣接するビット・フィールドを表す定数でなければなりません。

__rotatel4、__rotatel8

目的

ワードを左方に回転します。ダブルワードを左方に回転します。

rs を左方に *shift* ビットだけ回転します。

プロトタイプ

unsigned int __rotatel4(unsigned int *rs*, unsigned int *shift*);

unsigned long long __rotatel8(unsigned long long *rs*, unsigned long long *shift*);

保管関数

__store2r、__store4r

目的

2 バイトまたは 4 バイトの反転を保管します。

プロトタイプ

```
void __store2r(unsigned short, unsigned short *);

void __store4r(unsigned int, unsigned int *);
```

__store8r

目的

バイト反転付き保管 (8 バイト整数)

ロードされた 8 バイトの整数値に対して、バイト反転保管操作を行います。

プロトタイプ

```
void __store8r (unsigned long long source, unsigned long long * address);
```

使用法

-qarch が、64 ビット・モードの POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

トラップ関数

__tdw, __tw

目的

ダブルワードをトラップします。ワードをトラップします。

パラメーター *a* をパラメーター *b* と比較します。この比較の結果、5 ビットの定数 *TO* に AND 演算される条件は 5 つです。結果が 0 でない場合、システム・トラップ・ハンドラーが呼び出されます。

プロトタイプ

```
void __tdw ( long a, long b, unsigned int TO);

void __tw(int a, int b, unsigned int TO);
```

パラメーター

TO 包括的な 0 から 31 の値。ビット位置がそれぞれ設定されている場合は、次の可能な条件のうちの 1 つ以上を示しています。

0 (上位ビット)
a は *b* より小 (符号付きの比較を使用)。

1 *a* は *b* より大 (符号付きの比較を使用)。

2 *a* は *b* と等しい。

3 *a* は *b* より小 (符号なしの比較を使用)。

4 (下位ビット)
a は *b* より大 (符号なしの比較を使用)。

使用法

__tdw は 64 ビット・モードでのみ有効です。

__trap、__trapd
目的

パラメーターがゼロでない場合はトラップします。パラメーターがゼロ・ダブルワードでない場合はトラップします。

プロトタイプ

```
void __trap(int);

void __trapd ( long);
```

使用法

__trapd は 64 ビット・モードでのみ有効です。

2 進浮動小数点組み込み関数

浮動小数点組み込み関数は以下のカテゴリーにグループ化されています。

- 500 ページの『絶対値関数』
- 509 ページの『変換関数』
- 512 ページの『FPSCR 関数』
- 514 ページの『乗算-加算/減算関数』
- 515 ページの『逆数見積もり関数』
- 515 ページの『丸め関数』
- 517 ページの『選択関数』
- 517 ページの『平方根関数』
- 518 ページの『ソフトウェア除算関数』

絶対値関数

__fabss
目的

単精度浮動絶対値。

引数の絶対値を戻します。

プロトタイプ

```
float __fabss(float);
```

__fnabs
目的

負の浮動絶対値、負の単精度浮動絶対値。

引数の負の絶対値を戻します。

プロトタイプ

```
double __fnabs(double);

float __fnabss(float);
```

変換関数

__cmplx、__cmplxf、__cmplxl
目的

2 つの実際のパラメーターを単一の複合値に変換します。

プロトタイプ

```
double _Complex __cmplx (double, double) ;

float _Complex __cmplxf (float, float);

long double _Complex __cmplxl (long double, long double) ;
```

__fcfid
目的

整数ダブルワードからの浮動変換です。

ダブルワードで保管された 64 ビット符号付き整数を倍精度浮動小数点値に変換します。

プロトタイプ

```
double __fcfid (double);
```

__fcfud
目的

符号なし整数のダブルワードからの浮動小数点変換

ダブルワードで保管された 64 ビット符号なし整数を倍精度浮動小数点値に変換します。

プロトタイプ

```
double __fcfud(double);
```

__fctid
目的

整数ダブルワードへの浮動変換です。

現行の丸めモードを使用して倍精度の引数を 64 ビット符号付き整数に変換し、ダブルワードで結果を戻します。

プロトタイプ

```
double __fctid (double);
```

__fctidz
目的

ゼロの方向への丸めによる、整数ダブルワードへの浮動変換です。

ゼロの方向への丸めモードを使用して倍精度の引数を 64 ビット符号付き整数に変換し、ダブルワードで結果を戻します。

プロトタイプ

```
double __fctidz (double);
```

__fctiw
目的

整数ワードへの浮動変換です。

現行の丸めモードを使用して倍精度の引数を 32 ビット符号付き整数に変換し、ダブルワードで結果を戻します。

プロトタイプ

```
double __fctiw (double);
```

__fctiwz
目的

ゼロの方向への丸めによる、整数ワードへの浮動変換です。

ゼロの方向への丸めモードを使用して倍精度の引数を 32 ビット符号付き整数に変換し、ダブルワードで結果を戻します。

プロトタイプ

```
double __fctiwz (double);
```

__fctudz

目的

ゼロの方向への丸めを伴う、符号なし整数のダブルワードへの浮動小数点変換

浮動小数点値を符号なし整数のダブルワードに変換し、ゼロに向けて丸めます。

プロトタイプ

```
double __fctudz(double);
```

結果値

結果は、ゼロに向けて丸めた double の数値です。

__fctuwz

目的

ゼロに向けての丸めを伴う、符号なし整数ワードへの浮動小数点変換

浮動小数点数を 32 ビットの符号なし整数に変換し、ゼロに向けて丸めます。変換結果は、double の戻り値で保管されます。この関数は、__stfiw 組み込み関数と共に使用することを意図しています。

プロトタイプ

```
double __fctuwz(double);
```

結果値

結果は、double の数値です。結果の下位 32 ビットには、double のパラメーターを符号なし int に変換し、0 に向けて丸めた、符号なし int 値が含まれます。上位 32 ビットには、未定義の値が含まれます。

例

以下の例は、この関数の使用法を示しています。

```
#include <stdio.h>

int main(){
  double result;
  int y;

  result = __fctuwz(-1.5);
  __stfiw(&y, result);
  printf("%d¥n", y);            /* prints 0 */

  result = __fctuwz(1.5);
  __stfiw(&y, result);
  printf("%d¥n", y);            /* prints 1 */

  return 0;
}
```

__ibm2gccldbl、__ibm2gccldbl_cmplx (IBM 拡張)

目的

IBM スタイル long double データ型を GCC long double へ変換します。

プロトタイプ

```
long double __ibm2gccldbl (long double);

_Complex long double __ibm2gccldbl_cmplx (_Complex long double);
```

戻り値

変換された結果は long double の GCC 要件に準拠します。ただし、IBM のコンパイルされたコードで実行される long double の計算では、GCC そのものが取得する結果とビット単位で同一の結果を作成しない場合もあります。

FPSCR 関数

__mtfsb0

目的

浮動小数点状況および制御レジスター (FPSCR) ビット 0 への移動です。

FPSCR のビット *bt* を 0 に設定します。

プロトタイプ

```
void __mtfsb0(unsigned int bt);
```

パラメーター

bt 0 から 31 の値を持つ定数でなければなりません。

__mtfsb1

目的

FPSCR ビット 1 への移動です。

FPSCR のビット *bt* を 1 に設定します。

プロトタイプ

```
void __mtfsb1(unsigned int bt);
```

パラメーター

bt 0 から 31 の値を持つ定数でなければなりません。

__mtfsf

目的

FPSCR フィールドへの移動です。

frb の内容が、 *flm* で指定するフィールド・マスクの制御下の FPSCR に入れられます。フィールド・マスク *flm* は、影響を受ける FPSCR の 4 ビットのフィールドを識別します。

プロトタイプ

```
void __mtfsf(unsigned int flm, unsigned int frb);
```

パラメーター

flm
定数の 8 ビット・マスクでなければなりません。

__mtfsfi
目的

FPSCR フィールドへの即時移動です。

bf が指定する FPSCR フィールドに *u* の値を入れます。

プロトタイプ

```
void __mtfsfi(unsigned int bf, unsigned int u);
```

パラメーター

bf 0 から 7 の値を持つ定数でなければなりません。

u 0 から 15 の値を持つ定数でなければなりません。

__readflm
目的

64 ビット倍精度浮動小数点を戻します。この 64 ビット倍精度浮動小数点の 32 下位ビットには FPSCR の内容が含まれています。32 下位ビットは、最上位ビットから数えて 32 から 63 のビットです。

プロトタイプ

```
double __readflm (void);
```

__setflm
目的

倍精度浮動小数点数を取り、下位 32 ビットを FPSCR に入れます。32 下位ビットは、最上位ビットから数えて 32 から 63 のビットです。 FPSCR の以前の内容を戻します。

プロトタイプ

```
double __setflm(double);
```

__setrnd

目的

丸めモードを設定します。

プロトタイプ

```
double __setrnd (int mode);
```

パラメーター

mode の許容値は以下のとおりです。

- 0 — 最新値への丸め
- 1 — ゼロへの丸め
- 2 — 正の無限大への丸め
- 3 — 負の無限大への丸め

乗算-加算/減算関数

__fmadd、__fmadds

目的

浮動乗算-加算、単精度浮動乗算-加算。

最初の 2 つの引数を乗算し、3 番目の引数を加算してその結果を戻します。

プロトタイプ

```
double __fmadd (double, double, double);

float __fmadds (float, float, float);
```

__fmsub、__fmsubs

目的

浮動乗算-減算、単精度浮動乗算-減算。

最初の 2 つの引数を乗算し、3 番目の引数を減算してその結果を戻します。

プロトタイプ

```
double __fmsub(double, double, double);

float __fmsubs (float, float, float);
```

__fnmadd、__fnmadds

目的

負の浮動乗算-加算、負の単精度浮動乗算-加算。

最初の 2 つの引数を乗算し、3 番目の引数を加算してその結果を否定します。

プロトタイプ

```
double __fnmadd(double, double, double);

float __fnmadds (float, float, float);
```

__fnmsub、__fnmsubs

目的

負の浮動乗算-減算。

最初の 2 つの引数を乗算し、3 番目の引数を減算してその結果を否定します。

プロトタイプ

```
double __fnmsub(double, double, double);

float __fnmsubs (float, float, float);
```

逆数見積もり関数

517 ページの『平方根関数』も参照してください。

__fre、__fres

目的

浮動逆数見積もり、単精度浮動逆数見積もり。

プロトタイプ

```
double __fre (double);

float __fres (float);
```

使用法

-qarch が POWER5 以降のプロセッサーをターゲットにするように設定されている場合にのみ `__fre` は有効です。

丸め関数

__fric

目的

現行の丸めモードで浮動小数点を整数に丸める

倍精度浮動小数点値を現行の丸めモードで整数に丸めます。

プロトタイプ

```
double __fric(double);
```

__frim、__frims

目的

負の浮動整数へ丸めます。

負の無限大の方向への丸めモードを使用して浮動小数点引数を整数へ丸め、浮動小数点値として値を戻す。

プロトタイプ

```
double __frim (double);

float __frims (float);
```

使用法

-qarch が POWER5+ 以降のプロセッサーをターゲットにするように設定されている場合にのみ有効です。

__frin、__frins

目的

最近値の浮動整数へ丸めます。

最近値の方向への丸めモードを使用して浮動小数点引数を整数へ丸め、浮動小数点値として値を戻す。

プロトタイプ

```
double __frin (double);

float __frins (float);
```

使用法

-qarch が POWER5+ 以降のプロセッサーをターゲットにするように設定されている場合にのみ有効です。

__frip、__frips

目的

正の浮動整数へ丸めます。

正の無限大の方向への丸めモードを使用して浮動小数点引数を整数へ丸め、浮動小数点値として値を戻す。

プロトタイプ

```
double __frip (double);

float __frips (float);
```

使用法

-qarch が POWER5+ 以降のプロセッサーをターゲットにするように設定されている場合にのみ有効です。

__friz、__frizs
目的

ゼロの浮動整数へ丸めます。

ゼロの方向への丸めモードを使用して浮動小数点引数を整数へ丸め、浮動小数点値として値を戻す。

プロトタイプ

```
double __friz (double);

float __frizs (float);
```

使用法

-qarch が POWER5+ 以降のプロセッサーをターゲットにするように設定されている場合にのみ有効です。

選択関数

__fsel、__fsels
目的

浮動選択、単精度浮動選択。

最初の引数がゼロ以上である場合に 2 番目の引数を戻します。その他の場合には、3 番目の引数を戻します。

プロトタイプ

```
double __fsel (double, double, double);

float __fsels (float, float, float);
```

平方根関数

__frsqrte、__frsqrtes
目的

浮動小数点数の平方根の逆数の見積もり、単精度浮動小数点数の平方根の逆数の見積もり。

プロトタイプ

```
double __frsqrte (double);

float __frsqrtes (float);
```

使用法

-qarch が POWER5+ 以降のプロセッサーをターゲットにするように設定されている場合にのみ `__frsqrtes` は有効です。

__fsqrt、__fsqrts
目的

浮動平方根、単精度浮動平方根。

プロトタイプ

```
double __fsqrt (double);

float __fsqrts (float);
```

ソフトウェア除算関数

__swdiv、__swdivs
目的

ソフトウェア除算、単精度ソフトウェア除算。

最初の引数を 2 番目の引数で除算し、結果を戻します。

プロトタイプ

```
double __swdiv(double, double);

float __swdivs(float, float);
```

__swdiv_nochk、__swdivs_nochk
目的

ソフトウェア除算検査なし、単精度ソフトウェア除算検査なし。

範囲検査を実行せずに、最初の引数を 2 番目の引数で除算し、結果を戻します。

プロトタイプ

```
double __swdiv_nochk (double a, double b);

float __swdivs_nochk (float a, float b);
```

パラメーター

a 無限大に等しくてはなりません。**-qstrict** が有効な場合、*a* は 2^{-970} より大で無限大より小の絶対値を持っていなければなりません。

b 無限大、ゼロ、または正規化されていない値に等しくてはなりません。**-qstrict** が有効な場合は、*b* は 2^{-1022} より大で 2^{1021} より小の絶対値を持っていなければなりません。

戻り値

結果は正または負の無限大に等しくてはなりません。**-qstrict** が有効な場合は、結果は 2^{-1021} より大で 2^{1023} より小の絶対値を持っていなければなりません。

使用法

この関数は、ループ内で除算が繰り返し行われ、引数が許可範囲内の状態の場合、通常の除算演算子または `__swdiv` 組み込み関数よりもよいパフォーマンスを提供することができます。

保管関数

__stfiw

目的

整数ワードとして浮動小数点を保管します。

value の下位 32 ビットの内容が、 *addr* でアドレッシングするストレージのワードに、変換されずに保管されます。

プロトタイプ

```
void __stfiw( const int* addr, double value);
```

2 進コード化 10 進数組み込み関数

2 進コード化 10 進数 (BCD) 値は、それぞれ小数桁数と 4 ビットを占める符号ビットで構成される、圧縮された値です。桁は重みの順に右から左に配列され、最後の 4 ビットは符号のエンコードです。有効なエンコードには、それぞれの 31 桁に 0 から 9 までの範囲内の値があり、符号フィールドには 10 から 15 までの範囲内の値があることが必要です。

0b1010、0b1100、0b1110、または 0b1111 の符号コードを持つソース・オペランドは、正の値と解釈されます。 0b1011 または 0b1101 の符号コードを持つソース・オペランドは、負の値と解釈されます。

BCD 算術演算では、結果の符号は次のようにエンコードされます。優先符号 (PS) ビットの値に応じて、値 0b1101 は負の値を示し、0b1100 および 0b1111 は正の値またはゼロを示します。これらの組み込み関数は、最大で 31 桁の値を演算できます。

BCD 値は、1 バイトから 16 バイトまでの連続配列としてメモリーに格納されます。

BCD 組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするように設定されているときにのみ有効です。

BCD の加算と減算

__bcdadd

目的

BCD 値 *a* と *b* への加算の結果を返します。

結果の符号は以下のように決定されます。

- 結果が非負の値であり、*ps* が 0 の場合、符号は 0b1100 (0xC) に設定されます。
- 結果が非負の値であり、*ps* が 1 の場合、符号は 0b1111 (0xF) に設定されます。
- 結果が負の値の場合、符号は 0b1101 (0xD) に設定されます。

プロトタイプ

vector unsigned char __bcdadd (vector unsigned char *a*, vector unsigned char *b*, long *ps*);

パラメーター

ps コンパイル時の既知の定数。

__bcdsub

目的

BCD 値 *a* と *b* への減算の結果を返します。*ps* が 0 の場合は、非負の結果の符号を 0b1100 に、そうでない場合は、0b1111 に設定します。

結果の符号は以下のように決定されます。

- 結果が非負の値であり、*ps* が 0 の場合、符号は 0b1100 (0xC) に設定されます。
- 結果が非負の値であり、*ps* が 1 の場合、符号は 0b1111 (0xF) に設定されます。
- 結果が負の値の場合、符号は 0b1101 (0xD) に設定されます。

プロトタイプ

vector unsigned char __bcdsub (vector unsigned char *a*, vector unsigned char *b*, long *ps*);

パラメーター

ps コンパイル時の既知の定数。

オーバーフローの BCD テストの加算と減算

__bcdadd_ofl

目的

対応する BCD 加算演算の結果がオーバーフローになる場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdadd_ofl (vector unsigned char *a*, vector unsigned char *b*);

__bcdsub_ofl
目的

対応する BCD 減算演算の結果がオーバーフローになる場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdsub_ofl (vector unsigned char *a*, vector unsigned char *b*);

__bcd_invalid
目的

a が BCD 値の無効なエンコードである場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcd_invalid (vector unsigned char *a*);

BCD の比較

__bcdcmpeq
目的

BCD 値 *a* が *b* と等しい場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdcmpeq (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmpge
目的

BCD 値 *a* が *b* 以上の場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdcmpge (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmpgt
目的

BCD 値 *a* が *b* より大きい場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdcmpgt (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmple
目的

BCD 値 *a* が *b* 以下の場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdcmple (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmplt
目的

BCD 値 *a* が *b* より小さい場合は 1、そうでない場合は 0 を返します。

プロトタイプ

long __bcdcmplt (vector unsigned char *a*, vector unsigned char *b*);

BCD のロードおよび保管

__vec_ldrmb
目的

バイトのストリングをベクトル・レジスターに右寄せでロードします。左端エレメント (16-*cnt*) を 0 に設定します。

プロトタイプ

vector unsigned char __vec_ldrmb (char **ptr*, size_t *cnt*);

パラメーター

ptr
基底アドレスを指します。

cnt
ロードするバイト数。値の範囲は 1 から 16 までです。

__vec_strmb
目的

バイトの右寄せストリングを保管します。

プロトタイプ

void __vec_strmb (char **ptr*, size_t *cnt*, vector unsigned char *data*);

パラメーター

ptr
基底アドレスを指します。

cnt
保管するバイト数。値の範囲は 1 から 16 までです。

使用法

マルチスレッド・プログラムではこの関数は慎重に使用してください。この関数は、変更されたメモリー・ロケーションが含まれる 4 倍長ワードを、最大で 2 つまでロードして保管する場合があります。

同期およびアトミック組み込み関数

同期およびアトミック組み込み関数は以下のカテゴリーにグループ化されます。

- 『ロック検査関数』
- 525 ページの『ロック・クリア関数』
- 526 ページの『比較および交換関数』
- 526 ページの『フェッチ関数』
- 528 ページの『ロード関数』
- 529 ページの『保管関数』
- 530 ページの『同期関数』

ロック検査関数

__check_lock_mp、__check_lockd_mp

目的

マルチプロセッサー・システムのロックの検査、マルチプロセッサー・システムのロック・ダブルワード検査。

条件付きで、シングル・ワード変数またはダブルワード変数をアトミック更新します。

プロトタイプ

unsigned int __check_lock_mp (const int* *addr*, int *old_value*, int *new_value*);

unsigned int __check_lockd_mp (const long long* *addr*, long long *old_value*, long long *new_value*);

パラメーター

addr
更新する変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

old_value
addr の現行値に照らし合わせて検査される元の値。

new_value
addr の変数に条件付きで割り当てられる新規の値。

戻り値

addr の値が *old_value* に等しく、*new_value* に設定された場合は false (0) を戻します。*addr* の値が *old_value* に等しくなく、左方に変更されなかった場合は true (1) を戻します。

使用法

`__check_lockd_mp` は 64 ビット・モードでのみ有効です。

__check_lock_up、__check_lockd_up

目的

ユニプロセッサー・システムのロックの検査、ユニプロセッサー・システムのロック・ダブルワード検査。

条件付きで、シングル・ワード変数またはダブルワード変数をアトミック更新します。

プロトタイプ

unsigned int __check_lock_up (const int* *addr*, int *old_value*, int *new_value*);

unsigned int __check_lockd_up (const long* *addr*, long *old_value*, long *new_value*);

パラメーター

addr
更新する変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

old_value
addr の現行値に照らし合わせて検査される元の値。

new_value
addr の変数に条件付きで割り当てられる新規の値。

戻り値

addr の値が *old_value* に等しく、新規の値に設定された場合は、false (0) を戻します。*addr* の値が *old_value* に等しくなく、左方に変更されなかった場合は true (1) を戻します。

使用法

`__check_lockd_up` は 64 ビット・モードでのみ有効です。

ロック・クリア関数

__clear_lock_mp、__clear_lockd_mp

目的

マルチプロセッサー・システムのロックのクリア、マルチプロセッサー・システムのロック・ダブルワードのクリア。

アドレス *addr* にある変数に、*value* をアトミック保管します。

プロトタイプ

```
void __clear_lock_mp (const int* addr, int value);

void __clear_lockd_mp (const long* addr, long value);
```

パラメーター

addr
更新する変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

value
addr の変数に割り当てられる新規の値。

使用法

`__clear_lockd_mp` は 64 ビット・モードでのみ有効です。

__clear_lock_up、__clear_lockd_up

目的

ユニプロセッサー・システムのロックのクリア、ユニプロセッサー・システムのロック・ダブルワードのクリア。

アドレス *addr* にある変数に、*value* をアトミック保管します。

プロトタイプ

```
void __clear_lock_up (const int* addr, int value);

void __clear_lockd_up (const long* addr, long value);
```

パラメーター

addr
更新する変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

value
addr の変数に割り当てられる新規の値。

使用法

`__clear_lockd_up` は 64 ビット・モードでのみ有効です。

比較および交換関数

__compare_and_swap、__compare_and_swaplp

目的

条件付きで、シングル・ワード変数またはダブルワード変数をアトミック更新します。

プロトタイプ

int __compare_and_swap (volatile int* *addr*, int* *old_val_addr*, int *new_val*);

int __compare_and_swaplp (volatile long* *addr*, long* *old_val_addr*, long *new_val*);

パラメーター

addr
コピーする変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

old_val_addr
addr の値のコピー先となるメモリー・ロケーション。

new_val
addr の変数に条件付きで割り当てられる値。

戻り値

addr の値が *old_value* に等しく、新規の値に設定された場合は、true (1) を戻します。*addr* の値が *old_value* に等しくなく、左方に変更されなかった場合は false (0) を戻します。いずれの場合も、*addr* によって指定されたメモリー・ロケーションが *old_val_addr* によって指定されたメモリー・ロケーションにコピーされます。

使用法

__compare_and_swap 関数は、シングル・ワード値が更新する必要があるときに、この値が最後に読み取られてからまだ変更されていないという場合に限って役立ちます。__compare_and_swap をロック・プリミティブとして使用する場合は、__isync 組み込み関数への呼び出しをクリティカル・セクションの最初に挿入してください。

__compare_and_swaplp は 64 ビット・モードでのみ有効です。

フェッチ関数

__fetch_and_and、__fetch_and_andlp

目的

単一アトミック演算で、*addr* によって指定されたワードまたはダブルワードのビットを、その値を *val* で指定された値と AND 演算することにより消去し、*addr* の元の値を戻します。

プロトタイプ

unsigned int __fetch_and_and(volatile unsigned int* *addr*, unsigned int *val*);

unsigned long __fetch_and_andlp (volatile unsigned long* *addr*, unsigned long *val*);

パラメーター

addr
AND 演算される変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

value
addr の値が AND 演算される値。

使用法

この演算は、ビット・フラグを含む変数がいくつかのスレッドまたはプロセスの間で共有されている場合に役立ちます。

`__fetch_and_andlp` は 64 ビット・モードでのみ有効です。

__fetch_and_or、__fetch_and_orlp

目的

単一アトミック演算で、*addr* によって指定されたワードまたはダブルワードのビットを、その値を *val* で指定された値と OR 演算することにより設定し、*addr* の元の値を戻します。

プロトタイプ

unsigned int __fetch_and_or(volatile unsigned int* *addr*, unsigned int*val*);

unsigned long __fetch_and_orlp (volatile unsigned long* *addr*, unsigned long *val*);

パラメーター

addr
OR 演算される変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

value
addr の値が OR 演算される値。

使用法

この演算は、ビット・フラグを含む変数がいくつかのスレッドまたはプロセスの間で共有されている場合に役立ちます。

`__fetch_and_orlp` は 64 ビット・モードでのみ有効です。

__fetch_and_swap、__fetch_and_swaplp

目的

単一アトミック演算で、*addr* によって指定されるワードまたはダブルワードを *val* の値に設定し、*addr* の元の値を戻します。

プロトタイプ

unsigned int __fetch_and_swap(volatile unsigned int* *addr*, unsigned int*val*);

unsigned long __fetch_and_swaplp (volatile unsigned long* *addr*, unsigned long *val*);

パラメーター

addr
更新する変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

value
addr に割り当てられる値。

使用法

この演算は、変数が幾つかのスレッドまたはプロセス間で共有されており、1 つのスレッドが元々そのロケーションに保管されていた値を失うことなく、変数の値を更新する必要があるときに役立ちます。

`__fetch_and_swaplp` は 64 ビット・モードでのみ有効です。

ロード関数

__lqarx、__ldarx、__lwarx、__lharx、__lbarx

目的

4 倍長ワードをロードして索引付きで予約する、ダブルワードをロードして索引付きで予約する、ワードをロードして索引付きで予約する、ハーフワードをロードして索引付きで予約する、バイトをロードして索引付きで予約する。

addr によって指定されたメモリー・ロケーションから値をロードして、その結果を戻します。 `__lwarx` の場合は 64 ビット・モードで、コンパイラーは符号拡張された結果を戻します。

プロトタイプ

void __lqarx (volatile long* *addr*, long *dst[2]*);

long __ldarx (volatile long* *addr*);

int __lwarx (volatile int* *addr*);

short __lharx(volatile short* *addr*);

char __lbarx(volatile char* *addr*);

パラメーター

addr
ロードされる変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブル・ワードは 8 バイト境界上で、4 倍長ワードは 16 バイト境界上で位置合わせしなければなりません。

dst
値のロード先となるアドレス。

使用法

この関数は、後続の __stqcx (__stdcx、__stwcx、__sthcx、または __stbcx) 組み込み関数と共に使用して、指定されたメモリー・ロケーションで read-modify-write をインプリメントできます。 2 つの組み込み関数は一緒に機能して、保管が正常に行われた場合、load 関数が実行された時間と store 関数が完了するまでの時間の間に、他のプロセッサーまたはメカニズムがターゲット・メモリーを変更していないことを保証します。これはコードの動きに対して、__fence 組み込み関数を load 関数の前後に挿入した場合と同じ効果を持ち、周辺のコードをコンパイラーが最適化するのを禁止できます (__fence 組み込み関数について詳しくは、『 639 ページの『__alignx』』を参照してください)。

__ldarx および __lqarx は 64 ビット・モードでのみ有効です。__lqarx、__lharx、および __lbarx は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

保管関数

__stqcx、__stdcx、__stwcx、__sthcx、__stbcx

目的

4 倍長ワードを条件付き索引で保管する、ダブルワードを条件付き索引で保管する、ワードを条件付き索引で保管する、ハーフワードを条件付き索引で保管する、バイトを条件付き索引で保管する

val によって指定された値を *addr* によって指定されたメモリー・ロケーションに保管します。

プロトタイプ

int __stqcx(volatile long* *addr*, long *val[2]*);

int __stdcx(volatile long* *addr*, long *val*);

int __stwcx(volatile int* *addr*, int *val*);

int __sthcx(volatile short* *addr*, short *val*);

int __stbcx(volatile char* *addr*, char *val*);

パラメーター

addr
更新する変数のアドレス。シングル・ワードは 4 バイト境界上で、ダブルワードは 8 バイト境界上で位置合わせしなければなりません。

val
addr に割り当てられる値。

戻り値

addr の更新が正常に行われた場合は 1、正常に行われなかった場合は 0 を戻します。

使用法

この関数は、前の `__lqarx` (`__ldarx`、`__lwarx`、`__lharx`、または `__lbarx`) 組み込み関数と共に使用して、指定されたメモリー・ロケーションで read-modify-write をインプリメントできます。 2 つの組み込み関数は一緒に機能して、保管が正常に行われた場合、`__ldarx` 関数が実行された時間と `__stdcx` 関数が完了するまでの時間の間に、他のプロセッサーまたはメカニズムがターゲットのダブルワードを変更できないことを保証します。これは、`__fence` 組み込み関数を `__stdcx` 組み込み関数の前後に挿入した場合と同じ効果を持ち、周辺のコードをコンパイラーが最適化するのを禁じることができます。

`__stdcx` は 64 ビット・モードでのみ有効です。 `__stqcx`、`__sthcx`、および `__stbcx` は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

同期関数

__eieio、__iospace_eioio

目的

入出力のインオーダー実行を強制します。

`_eioeio` への呼び出しの前の入出力ストレージ・アクセス命令のすべてが、関数呼び出しの後の入出力ストレージ・アクセス命令の前にメイン・メモリーで完了することを保証します。

プロトタイプ

```
void __eieio(void);

void __iospace_eieio(void);
```

使用法

この関数は、ロードまたは保管アクセスの実行順序が重要な共有データ命令の管理に役立ちます。この関数は、他の同期命令によって発生することがあるパフォーマンスのコストなしで入出力保管を制御するために必要な機能を提供することができます。

__isync

目的

命令を同期化します。

前の命令がすべて完了するまで待ってから、プリフェッチされた命令をすべて破棄し、後続の命令が前の命令によって確立されたコンテキストでフェッチ (または再フェッチ) され、実行されるようにします。

プロトタイプ

```
void __isync(void);
```

__lwsync、__iospace_lwsync

目的

ワードのロードを同期化します。

__lwsync の呼び出しの前にあるすべての命令が完了した後でなければ、この関数を実行するプロセッサー上で後続の保管命令を実行できないようにします。また、__lwsync の呼び出しの前にあるすべてのロード命令が完了した後でなければ、この関数を実行するプロセッサー上で後続のロード命令を実行できないようにします。これによって、__lwsync がそれぞれのプロセッサーからの確認を待たないため、パフォーマンスへの影響を最小限にして複数のプロセッサー間での同期化が可能になります。

プロトタイプ

```
void __lwsync (void);

void __iospace_lwsync (void);
```

__sync、__iospace_sync

目的

同期化します。

_sync への関数呼び出しの前のすべての命令が関数呼び出しの後の命令が実行される前に完了することを保証します。

プロトタイプ

```
void __sync (void);

void __iospace_sync (void);
```

キャッシュ関連組み込み関数

キャッシュ関連の組み込み関数は以下のカテゴリーにグループ化されます。

- 532 ページの『データ・キャッシュ関数』
- 534 ページの『プリフェッチ組み込み関数』

データ・キャッシュ関数

__dcbf

目的

データ・キャッシュ・ブロックをフラッシュします。

変更されたブロックの内容をデータ・キャッシュからメイン・メモリーにコピーし、そのコピーをデータ・キャッシュからフラッシュします。

プロトタイプ

```
void __dcbf(const void* addr);
```

__dcbfl

目的

データ・キャッシュ・ブロックの行をフラッシュします。

L1 データ・キャッシュから指定したアドレスのキャッシュ行をフラッシュします。

プロトタイプ

```
void __dcbfl (const void* addr );
```

使用法

ターゲットのストレージ・ブロックは L2 キャッシュに保存されます。

-qarch が、POWER6 以上のプロセッサーをターゲットとするように設定されている場合に有効です。

__dcbflp

目的

1 次データ・キャッシュ・ブロックの行のフラッシュ

address にあるキャッシュ・ラインをシングル・プロセッサーの 1 次データ・キャッシュからフラッシュします。

プロトタイプ

```
void __dcbflp(const void* address);
```

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__dcbst

目的

データ・キャッシュ・ブロックを保管します。

変更されたブロックの内容をデータ・キャッシュからメイン・メモリーにコピーします。

プロトタイプ

```
void __dcbst(const void* addr);
```

__dcbt
目的

データ・キャッシュ・ブロックをタッチします。

指定のアドレスを含むメモリーのブロックを L1 データ・キャッシュ内にロードします。

プロトタイプ

```
void __dcbt (void* addr);
```

__dcbtna
目的

データ・キャッシュ・ブロック・ヒントはアクセスされなくなりました。

アドレスを含んでいるブロックが長期間アクセスされなくなるため、L1 データ・キャッシュに保持しておくべきでないことを示します。

注: この関数を使用しても、含んでいるブロックがデータ・キャッシュから必ずしも除去されるわけではありません。

プロトタイプ

```
void __dcbtna (void *addr);
```

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

__dcbtst
目的

保管用のデータ・キャッシュ・ブロックをタッチします。

指定のアドレスを含むメモリーのブロックをデータ・キャッシュ内にフェッチします。

プロトタイプ

```
void __dcbtst(void* addr);
```

__dcbz

目的

データ・キャッシュ・ブロックをゼロに設定します。

データ・キャッシュに指定されたアドレスを含むキャッシュ行をゼロ (0) に設定します。

プロトタイプ

```
void __dcbz (void* addr);
```

__icbt

目的

命令キャッシュ・ブロック・タッチ

プログラムがアドレスを含む命令キャッシュ・ブロック内のコードをすぐに実行し、アドレスを含むブロックが命令キャッシュにロードされる必要があることを示します。

プロトタイプ

```
void __icbt (void *addr) ;
```

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

プリフェッチ組み込み関数

__dcbtstt

目的

保管の一時的なタッチを行うと、プログラムが保管アクセスを実行する可能性があるブロックについて説明したヒントが得られます。このブロックは一時的なものと考えられます。つまり、プログラムが単位にアクセスする時間間隔は、短いと考えられます。

プロトタイプ

```
void __dcbtstt (void * address);
```

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__dcbtt

目的

一時的なデータ・キャッシュ・ブロックのタッチ

ロードの一時的なタッチを行うと、プログラムがロード・アクセスを実行する可能性があるブロックについて説明したヒントが得られます。このブロックは一時的なものと考えられます。つまり、プログラムが単位にアクセスする時間間隔は、短いと考えられます。

プロトタイプ

```
void __dcbtt (void * address);
```

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__default_prefetch_depth

目的

ハードウェア検出のストリームおよびソフトウェア定義のストリームのプリフェッチ深さを定義します。この関数は、以下のいずれかの場合に有効です。

- プリフェッチ深さに 0 が指定されている場合
- `__dcbt` または `__dcbtst` が `TH=1010` と共に使用されていない場合

プロトタイプ

```
void __default_prefetch_depth (unsigned int prefetch_depth);
```

パラメーター

prefetch_depth
ストリームのプリフェッチの定常状態 fetch-ahead 距離を設定する相対的で質的な値。fetch-ahead 距離は、データのロード元またはデータの保管先である行の前にプリフェッチされる行の数です。有効値は以下のとおりです。

0 データ・ストリーム制御レジスターで定義されるデフォルト。

1 なし。

2 最も浅い。

3 浅い。

4 中間。

5 深い。

6 さらに深い。

7 最も深い。

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

__depth_attainment_urgency

目的

ハードウェア検出のストリームに対してプリフェッチ深さに達する緊急度を設定します。

プロトタイプ

```
void __depth_attainment_urgency (unsigned int urgency);
```

パラメーター

urgency
ハードウェア検出のストリームに対してプリフェッチ深さに達する速さを示します。有効値は以下のとおりです。

0 データ・ストリーム制御レジスターで定義されるデフォルト値。

1 緊急度なし。

2 最も低い緊急度。

3 より低い緊急度。

4 中間。

5 緊急。

6 より高い緊急度。

7 最も高い緊急度。

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

__hardware_transient_enable

目的

ハードウェア検出のストリームに対して transient 属性を使用可能または使用不可にします。

プロトタイプ

```
void __hardware_transient_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 は、ハードウェア検出のストリームに対して transient 属性を使用不可にします。値 1 はハードウェア検出のストリームに対して transient 属性を使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。データ・ストリーム制御レジスター (DSCR) でストリームに transient 属性を適用する場合、4 つの使用可能化組み込み関数のう

ち、少なくとも 2 つを設定する必要があります。この場合、1 つの関数はアクセスのタイプ (ロードまたは保管) を選択するように設定し、もう 1 つの関数はプリフェッチのタイプ (ソフトウェア定義またはハードウェア検出) を選択するように設定します。

__hardware_unit_count_enable

目的

ハードウェア検出のストリームに対して単位カウントを使用可能または使用不可にします。単位カウントは、キャッシュ・ラインの数を示す、0 から 1023 の値を持つ整数です。

プロトタイプ

```
void __hardware_unit_count_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 は、ハードウェア検出のストリームに対して単位カウントを使用不可にします。値 1 は、ハードウェア検出のストリームに対して単位カウントを使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

__load_stream_disable

目的

ロード・ストリームのハードウェア検出および開始を使用不可にします。

プロトタイプ

```
void __load_stream_disable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 の場合、効果はありません。値 1 は、ロード・ストリームのハードウェア検出および開始を使用不可にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

__load_transient_enable

目的

ロード・ストリームに対して transient 属性を使用可能または使用不可にします。

プロトタイプ

```
void __load_transient_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 は、ロード・ストリームに対して transient 属性を使用不可にします。値 1 は、ロード・ストリームに対して transient 属性を使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。データ・ストリーム制御レジスター (DSCR) でストリームに transient 属性を適用する場合、4 つの使用可能化組み込み関数のうち、少なくとも 2 つを設定する必要があります。この場合、1 つの関数はアクセスのタイプ (ロードまたは保管) を選択するように設定し、もう 1 つの関数はプリフェッチのタイプ (ソフトウェア定義またはハードウェア検出) を選択するように設定します。

__partial_dcbt

目的

部分データ・キャッシュ・ブロック・タッチ

指定されたアドレスを含んでいるキャッシュ・ラインの半分を L3 データ・キャッシュにロードします。

プロトタイプ

```
void __partial_dcbt (void * address);
```

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__prefetch_by_load

目的

明示的ロードを使用してメモリー・ロケーションをタッチします。

プロトタイプ

```
void __prefetch_by_load(const void*);
```

__prefetch_by_stream

目的

明示的ストリームを使用して連続するメモリー・ロケーションをタッチします。

プロトタイプ

```
void __prefetch_by_stream(const int, const void*);
```

__prefetch_get_dscr_register
目的

データ・ストリーム制御レジスター (DSCR) の現行値を返します。

プロトタイプ

```
unsigned long long __prefetch_get_dscr_register (void);
```

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

__prefetch_set_dscr_register
目的

データ・ストリーム制御レジスター (DSCR) の値を入力値に設定します。

プロトタイプ

```
void __prefetch_set_dscr_register (unsigned long long value);
```

パラメーター

value
値は 4 バイト整数でなければなりません。

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

__protected_stream_count
目的

特定の長さ制限のある protected ストリームのキャッシュ行の数を設定します。

プロトタイプ

```
void __protected_stream_count (unsigned int unit_cnt, unsigned int stream_ID);
```

パラメーター

unit_cnt
キャッシュ行の数。0 から 1023 の値を持つ整数でなければなりません。

stream_ID
POWER5 プロセッサー上で 0 から 7 まで、POWER6 プロセッサー上で 0 から 15 まで、および POWER7 およびPOWER8 プロセッサー上で 0 から 11 までの値を持つ整数。

使用法

-qarch が、POWER5 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_stream_count_depth

目的

特定の長さ制限のある protected ストリームのキャッシュ行の数およびプリフェッチの深さを設定します。

プロトタイプ

```
void _protected_stream_count_depth (unsigned int unit_cnt, unsigned int
prefetch_depth, unsigned int stream_ ID);
```

パラメーター

unit_cnt
キャッシュ行の数。0 から 1023 の値を持つ整数でなければなりません。

prefetch_depth
ストリームのプリフェッチの定常状態 *fetch-ahead* 距離を設定する相対的で質的な値。fetch-ahead 距離は、データの現在のロード元またはデータの現在の保管先である行の前にプリフェッチされる行の数です。 有効値は以下のとおりです。

0 データ・ストリーム制御レジスターで定義されるデフォルト。

1 なし。

2 最も浅い。

3 浅い。

4 中間。

5 深い。

6 さらに深い。

7 最も深い。

stream_ID
POWER6 プロセッサー上で 0 から 15 まで、および POWER7 および POWER8 プロセッサー上で 0 から 11 までの値を持つ整数。

使用法

-qarch が、POWER6 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_stream_go

目的

長さ制限のあるすべての protected ストリームのプリフェッチを開始します。

プロトタイプ

```
void __protected_stream_go (void);
```

使用法

-qarch が、POWER5 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_stream_set
目的

長さ制限のある protected ストリームを確立します。これはインクリメンタル (前方) またはデクリメンタル (後方) メモリー・アドレスのいずれかからフェッチします。このストリームはハードウェア検出のストリームによる置換から保護されています。

プロトタイプ

```
void __protected_stream_set (unsigned int direction, const void* addr, unsigned int stream_ID);
```

パラメーター

direction
1 (前方) または 3 (後方) の値を持つ整数。

addr
キャッシュ行の先頭。

stream_ID
POWER5 プロセッサー上で 0 から 7 まで、POWER6 プロセッサー上で 0 から 15 まで、および POWER7 およびPOWER8 プロセッサー上で 0 から 11 までの値を持つ整数。

使用法

-qarch が、POWER5 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_unlimited_stream_set
目的

非限定の長さの protected ストリームを確立します。これはインクリメンタル (前方) またはデクリメンタル (後方) メモリー・アドレスのいずれかからフェッチします。このストリームはハードウェア検出のストリームによる置換から保護されています。

プロトタイプ

```
void _protected_unlimited_stream_set (unsigned int direction, const void* addr, unsigned int ID);
```

パラメーター

direction
1 (前方) または 3 (後方) の値を持つ整数。

addr
キャッシュ行の先頭。

stream_ID
POWER5 プロセッサー上で 0 から 7 まで、POWER6 プロセッサー上で 0 から 15 まで、および POWER7 およびPOWER8 プロセッサー上で 0 から 11 までの値を持つ整数。

使用法

-qarch が、POWER5 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_stream_stride
目的

ID が stream_id である保護されたロードまたは保管ストリームについて、ストリームの最初の単位のワード・オフセット address_offset と、ワード・サイズのストライドを設定します。

プロトタイプ

void__protected_stream_stride (unsigned int *address_offset*, unsigned int *stride*, unsigned int *stream_id*);

パラメーター

address_offset
プリフェッチ変数の最初の単位のアドレス。

stride
プリフェッチ・ストリームの 2 つの連続するエレメントの距離をワード数で表したもの。

stream_id
0 から 11 の値を持つ整数。

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_stream_stop
目的

protected ストリームのプリフェッチを停止します。

プロトタイプ

void __protected_stream_stop (unsigned int *stream ID*);

パラメーター

stream_id
POWER5 プロセッサー上で 0 から 7 まで、POWER6 プロセッサー上で 0 から 15 まで、および POWER7 およびPOWER8 プロセッサー上で 0 から 11 までの値を持つ整数。

使用法

-qarch が、POWER5 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__protected_stream_stop_all

目的

すべての protected ストリームのプリフェッチを停止します。

プロトタイプ

```
void __protected_stream_stop_all (void);
```

使用法

-qarch が、POWER5 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

__protected_store_stream_set

目的

長さ制限のある protected ストア・ストリームを確立します。これはインクリメンタル (前方) またはデクリメンタル (後方) メモリー・アドレスのいずれかからフェッチします。このストリームはハードウェア検出のストリームによる置換から保護されています。

プロトタイプ

```
void _protected_store_stream_set (unsigned int direction, const void* addr,
unsigned int stream_ID );
```

パラメーター

direction
1 (前方) または 3 (後方) の値を持つ整数。

addr
キャッシュ行の先頭。

stream_ID
POWER6 プロセッサーに対しては 0 から 15 まで、POWER7 および POWER8 プロセッサーに対しては 0 から 11 までの値の整数。

使用法

-qarch が、POWER6 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効です。

__protected_unlimited_store_stream_set

目的

非限定の長さの protected ストア・ストリームを確立します。これはインクリメンタル (前方) またはデクリメンタル (後方) メモリー・アドレスのいずれかからフェッチします。このストリームはハードウェア検出のストリームによる置換から保護されています。

プロトタイプ

```
void _protected_unlimited_store_stream_set (unsigned int direction, const void*
addr, unsigned int stream_ID);
```

パラメーター

direction
1 (前方) または 3 (後方) の値を持つ整数。

addr
キャッシュ行の先頭。

stream_ID
POWER6 プロセッサーに対しては 0 から 15 まで、POWER7 および POWER8 プロセッサーに対しては 0 から 11 までの値の整数。

使用法

-qarch が、POWER6 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__set_prefetech_unit_count

目的

データ・ストリーム内のユニット数を設定します。

プロトタイプ

```
void __set_prefetch_unit_count (unsigned int count);
```

パラメーター

count
キャッシュ行の数。0 から 1023 の値を持つ整数でなければなりません。

使用法

-qarch がターゲット POWER8 プロセッサーに設定されるときのみ有効です。

__software_transient_enable

目的

ソフトウェア定義のストリームに対する transient 属性を使用可能または使用不可にします。

プロトタイプ

```
void __software_transient_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 は、ソフトウェア定義のストリームに対して transient 属性を使用不可にします。値 1 はソフトウェア定義のストリームに対して transient 属性を使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。データ・ストリーム制御レジスター (DSCR) でストリームに transient 属性を適用する場合、4 つの使用可能化組み込み関数のうち、少なくとも 2 つを設定する必要があります。この場合、1 つの関数はアクセスのタイプ (ロードまたは保管) を選択するように設定し、もう 1 つの関数はプリフェッチのタイプ (ソフトウェア定義またはハードウェア検出) を選択するように設定します。

__software_unit_count_enable

目的

ソフトウェア定義のストリームに対して単位カウントを使用可能または使用不可にします。単位カウントは、キャッシュ・ラインの数を示す、0 から 1023 の値を持つ整数です。

プロトタイプ

```
void __software_unit_count_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 は、ソフトウェア定義のストリームに対して単位カウントを使用不可にします。値 1 は、ソフトウェア定義のストリームに対して単位カウントを使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

__store_transient_enable

目的

ストア・ストリームに対して transient 属性を使用可能または使用不可にします。

プロトタイプ

```
void __store_transient_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 は、ストア・ストリームに対して transient 属性を使用不可にします。値 1 は、ストア・ストリームに対して transient 属性を使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。データ・ストリーム制御レジスター (DSCR) でストリームに transient 属性を適用する場合、4 つの使用可能化組み込み関数のうち、少なくとも 2 つを設定する必要があります。この場合、1 つの関数はアクセスのタイプ (ロードまたは保管) を選択するように設定し、もう 1 つの関数はプリフェッチのタイプ (ソフトウェア定義またはハードウェア検出) を選択するように設定します。

__stride_n_stream_enable

目的

単一のキャッシュ・ブロックより大きいストライドを持つロード・ストリームとストア・ストリームのハードウェア検出および開始を使用可能または使用不可にします。このようなロード・ストリームは、Loop Stream Disable (LSD) ビットの値が 0 の場合にのみ検出されます。このようなストア・ストリームは、Store Stream Enable (SSE) ビットの値が 1 の場合にのみ検出されます。

プロトタイプ

```
void __stride_n_stream_enable (unsigned int flag);
```

パラメーター

flag
0 または 1 の値を持つ整数。値 0 の場合、効果はありません。値 1 は、単一のキャッシュ・ブロックより大きいストライドを持つロード・ストリームとストア・ストリームのハードウェア検出および開始を使用可能にします。

使用法

この組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

__transient_protected_stream_count_depth

目的

特定の長さ制限のある保護されたロードまたは保管ストリーム (ID が *stream_id*) のキャッシュ・ラインの数 *unit_cnt* とプリフェッチの深さ *prefetch_depth* を設定します。用語「transient (一時的)」は、プログラムによるそのストリームのメモリーへのアクセス時間間隔が短いと考えられ、したがって、先にプロセッサーによってキャッシュから除去される可能性があることを示します。

プロトタイプ

void __transient_protected_stream_count_depth (unsigned int *unit_cnt*, unsigned int *prefetch_depth*, unsigned int *stream_id*);

パラメーター

unit_cnt
キャッシュ行の数。0 から 1023 の値を持つ整数でなければなりません。

prefetch_depth
ストリームのプリフェッチの定常状態 *fetch-ahead* 距離を設定する相対的で質的な値。fetch-ahead 距離は、データの現在のロード元またはデータの現在の保管先である行の前にプリフェッチされる行の数です。 有効値は以下のとおりです。

0 データ・ストリーム制御レジスターで定義されるデフォルト。

1 なし。

2 最も浅い。

3 浅い。

4 中間。

5 深い。

6 さらに深い。

7 最も深い。

stream_id
0 から 11 の値を持つ整数。

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__transient_unlimited_protected_stream_depth

目的

長さ制限のない保護されたロードまたは保管ストリーム (ID が *stream_id*) のプリフェッチの深さ prefetch_depth を設定します。ストリームは一時的なものと考えられます。つまり、プログラムが単位にアクセスする時間間隔は、短いと考えられます。

プロトタイプ

void __transient_unlimited_protected_stream_depth (unsigned int *prefetch_depth*, unsigned int *stream_id*);

パラメーター

prefetch_depth
ストリームのプリフェッチの定常状態 *fetch-ahead* 距離を設定する相対的で質的な値。fetch-ahead 距離は、データの現在のロード元またはデータの現在の保管先である行の前にプリフェッチされる行の数です。 有効値は以下のとおりです。

0 データ・ストリーム制御レジスターで定義されるデフォルト。

1 なし。

2 最も浅い。

3 浅い。

4 中間。

5 深い。

6 さらに深い。

7 最も深い。

stream_id
0 から 11 の値を持つ整数。

使用法

-qarch が、POWER7 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

__unlimited_protected_stream_depth

目的

長さ制限のない保護されたロードまたは保管ストリーム (ID が *stream_id*) のプリフェッチの深さ *prefetch_depth* を設定します。

プロトタイプ

void __unlimited_protected_stream_depth (unsigned in *prefetch_depth*, unsigned int *stream_id*);

パラメーター

prefetch_depth
ストリームのプリフェッチの定常状態 *fetch-ahead* 距離を設定する相対的で質的な値。fetch-ahead 距離は、データの現在のロード元またはデータの現在の保管先である行の前にプリフェッチされる行の数です。 有効値は以下のとおりです。

0 データ・ストリーム制御レジスターで定義されるデフォルト。

1 なし。

2 最も浅い。

3 浅い。

4 中間。

5 深い。

6 さらに深い。

7 最も深い。

stream_id
POWER6 プロセッサーに対しては 0 から 15 まで、POWER7 および POWER8 プロセッサーに対しては 0 から 11 までの値の整数。

使用法

-qarch が、POWER6 以上のプロセッサーをターゲットとするように設定されている場合にのみ有効。

暗号化組み込み関数

暗号化組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするように設定されているときにのみ有効です。

拡張暗号化標準関数

拡張暗号化標準 (AES) 関数は、暗号と復号の仕様書である、連邦情報処理標準資料 197 (FIPS-197) のサポートを提供します。

__vcipher

目的

特定の *round_key* を使用して、中間状態の *state_array* に対して 1 ラウンドの AES 暗号化操作を実行します。

プロトタイプ

```
vector unsigned char __vcipher (vector unsigned char state_array, vector unsigned char round_key);
```

パラメーター

state_array
暗号化する入力データ・チャンク、または直前の vcipher 操作の結果。

round_key
暗号化に使用する 128 ビット AES ラウンド・キー値。

結果

結果の中間状態を返します。

__vcipherlast

目的

特定の *round_key* を使用して、中間状態の *state_array* に対して最終ラウンドの AES 暗号化操作を実行します。

プロトタイプ

vector unsigned char __vcipherlast (vector unsigned char *state_array*, vector unsigned char *round_key*);

パラメーター

state_array
直前の vcipher 操作の結果。

round_key
暗号化に使用する 128 ビット AES ラウンド・キー値。

結果

結果の最終状態を返します。

__vncipher

目的

特定の *round_key* を使用して、中間状態の *state_array* に対して 1 ラウンドの AES 復号操作を実行します。

プロトタイプ

vector unsigned char __vncipher (vector unsigned char *state_array*, vector unsigned char *round_key*);

パラメーター

state_array
復号する入力データ・チャンク、または直前の vncipher 操作の結果。

round_key
復号に使用する 128 ビット AES ラウンド・キー値。

結果

結果の中間状態を返します。

__vncipherlast

目的

特定の *round_key* を使用して、中間状態の *state_array* に対して最終ラウンドの AES 逆暗号化操作を実行します。

プロトタイプ

vector unsigned char __vncipherlast (vector unsigned char *state_array*, vector unsigned char *round_key*);

パラメーター

state_array
直前の vncipher 操作の結果。

round_key
復号に使用する 128 ビット AES ラウンド・キー値。

結果

結果の最終状態を返します。

__vsbox
目的

state_array に対して、FIPS-197 で定義されている SubBytes 演算を実行します。

プロトタイプ

vector unsigned char __vsbox (vector unsigned char *state_array*);

パラメーター

state_array
暗号化する入力データ・チャンク、または直前の vcipher 操作の結果。

結果

演算の結果を返します。

セキュア・ハッシュ・アルゴリズム関数

セキュア・ハッシュ・アルゴリズム (SHA) 関数は、連邦情報処理標準資料 180-3 (FIPS-180-3) の、セキュア・ハッシュ標準のサポートを提供します。すべての SHA 関数は、符号なしベクトル整数タイプを演算します。

__vshasigmad
目的

セキュア・ハッシュ標準の仕様書である、連邦情報処理標準資料 FIPS-180-3 のサポートを提供します。

プロトタイプ

vector unsigned long long __vshasigmad (vector unsigned long long *x*, int *type*, int *fmask*);

パラメーター

type
範囲 0 から 1 のコンパイル時定数。*type* パラメーターにより、小文字シグマまたは大文字シグマのいずれかとなる、関数型が選択されます。

fmask
範囲 0 から 15 のコンパイル時定数。 *fmask* パラメーターにより、sigma-0 または sigma-1 のいずれかとなる、関数サブタイプが選択されます。

結果

`mask` を `fmask` の右端の 4 ビットにします。

x の各エレメント `i` (i=0,1) に対して、戻り値のエレメント `i` は、以下の結果 SHA-512 関数です。

- `type` が 0 で `mask` のビット `2*i` が 0 の場合、結果 SHA-512 関数は `sigma0(x[i])` です。
- `type` が 0 で `mask` のビット `2*i` が 1 の場合、結果 SHA-512 関数は `sigma1(x[i])` です。
- `type` が非ゼロで `mask` のビット `2*i` が 0 の場合、結果 SHA-512 関数は `Sigma0(x[i])` です。
- `type` が非ゼロで `mask` のビット `2*i` が 1 の場合、結果 SHA-512 関数は `Sigma1(x[i])` です。

__vshasigmaw
目的

セキュア・ハッシュ標準の仕様書である、連邦情報処理標準資料 FIPS-180-3 のサポートを提供します。

プロトタイプ

vector unsigned int __vshasigmaw (vector unsigned int *x*, int *type*, int *fmask*)

パラメーター

type
範囲 0 から 1 のコンパイル時定数。*type* パラメーターにより、小文字シグマまたは大文字シグマのいずれかとなる、関数型が選択されます。

fmask
範囲 0 から 15 のコンパイル時定数。 *fmask* パラメーターにより、sigma-0 または sigma-1 のいずれかとなる、関数サブタイプが選択されます。

結果

`mask` を `fmask` の右端の 4 ビットにします。

x の各エレメント `i` (i=0,1,2,3) に対して、戻り値のエレメント `i` は、以下の結果 SHA-256 関数です。

- type が 0 で、マスクのビット i が 0 の場合、結果の SHA-256 関数は sigma0(x[i]) です。
- type が 0 で、マスクのビット i が 1 の場合、結果の SHA-256 関数は sigma1(x[i]) です。
- type が非ゼロで、マスクのビット i が 0 の場合、結果の SHA-256 関数は Sigma0(x[i]) です。
- type が非ゼロで、マスクのビット i が 1 の場合、結果の SHA-256 関数は Sigma1(x[i]) です。

その他の関数

__vpermxor

目的

2 つのバイト・ベクトルに対して、順序を変える、排他 OR 演算を適用します。

プロトタイプ

vector unsigned char __vpermxor (vector unsigned char *a*, vector unsigned char *b*, vector unsigned char *mask*);

結果

それぞれの i (0 <= i < 16) について、*mask* のバイト・エレメント i は、indexA はビット 0 から 3、indexB はビット 4 から 7 とします。

結果のバイト・エレメント i は、*a* の indexA および *b* の indexB のバイト・エレメントの、排他 OR に設定されます。

__vpmsumb

目的

対応するエレメントの多項式乗算結果の、偶数/奇数の各ペアに、排他 OR 演算を実行します。

プロトタイプ

vector unsigned char __vpmsumb (vector unsigned char *a*, vector unsigned char *b*)

結果

それぞれの i (0 <= i < 16) に対して、prod[i] を、*a* および *b* のバイト・エレメント i の多項式乗算の結果にします。

それぞれの i (0 <= i < 8) に対して、結果の各ハーフワード・エレメント i は、以下のように設定されます。

- ビット 0 は、0 に設定されます。
- ビット 1 から 15 は、prod[2*i] (xor) prod[2*i+1] に設定されます。

__vpmsumd

目的

対応するエレメントの多項式乗算結果の、偶数/奇数の各ペアに、排他 OR 演算を実行します。

プロトタイプ

vector unsigned long long __vpmsumd (vector unsigned long long *a*, vector unsigned long long *b*);

結果

それぞれの `i` (0 <= `i` < 2 に対して、`prod[i]` を、*a* および *b* のダブルワード・エレメント `i` の多項式乗算の結果にします。

結果のビット 0 は、0 に設定されます。

結果のビット 1 から 127 は、`prod[0] (xor) prod[1]` に設定されます。

__vpmsumh

目的

対応するエレメントの多項式乗算結果の、偶数/奇数の各ペアに、排他 OR 演算を実行します。

プロトタイプ

vector unsigned short __vpmsumh (vector unsigned short *a*, vector unsigned short *b*);

結果

それぞれの `i` (0 <= `i` < 8 に対して、`prod[i]` を、*a* および *b* のハーフワード・エレメント `i` の多項式乗算の結果にします。

それぞれの `i` (0 <= `i` < 4) に対して、結果の各ワード・エレメント `i` は、以下のように設定されます。

- ビット 0 は、0 に設定されます。
- ビット 1 から 31 は、`prod[2*i] (xor) prod[2*i+1]` に設定されます。

__vpmsumw

目的

対応するエレメントの多項式乗算結果の、偶数/奇数の各ペアに、排他 OR 演算を実行します。

プロトタイプ

vector unsigned int __vpmsumw (vector unsigned int *a*, vector unsigned int *b*);

結果

それぞれの i (0 <= i < 4 に対して、prod[i] を、*a* および *b* のワード・エレメント i の多項式乗算の結果にします。

それぞれの i (0 <= i < 2) に対して、結果の各ダブルワード・エレメント i は、以下のように設定されます。

- ビット 0 は、0 に設定されます。
- ビット 1 から 63 は、prod[2*i] (xor) prod[2*i+1] に設定されます。

ブロックに関連した組み込み関数

__bcopy

目的

n バイトを *src* から *dest* にコピーします。結果は、両方の領域がオーバーラップしても適正です。

プロトタイプ

```
void __bcopy(const void* src, void* dest, size_t n);
```

パラメーター

src
コピーするデータのソース・アドレス。

dest
コピーするデータの宛先アドレス。

n データのサイズ。

ベクトル組み込み関数

ベクトルの個々のエレメントには、Vector Multimedia eXtension (VMX) または Vector Scalar eXtension (VSX) 組み込み関数を使用してアクセスできます。このセクションでは、VMX および VSX 組み込み関数をアルファベット順に解説します。これらの関数を使用すると、ベクトルを操作できます。

これらの組み込み関数を使用する場合は、使用するアーキテクチャーに合った適切なコンパイラー・オプションを指定する必要があります。**vector unsigned long long**、**vector signed long long**、**vector bool long long**、または **vector double** 型を使用する組み込み関数には、VSX 命令セット拡張機能をサポートするアーキテクチャー、例えば POWER7 などが必要です。これらの型を使用するときは、**-qarch=pwr7** を指定する必要があります。

このセクションでは、以下に示すように、疑似コードの記述を使用して関数の構文を表します。

```
d=func_name(a, b, c)
```

この記述では、

- d は関数の戻り値を表しています。
- a、b、および c は関数の引数を表しています。
- func_name は関数の名前です。

例えば、関数 vector double vec_xld2(int, double*); の構文は、d=vec_xld2(a, b) によって表されます。

注: このセクションでは IBM 固有のベクトル組み込み関数と、IBM 拡張を備えた AltiVec 組み込み関数のみを説明します。他の AltiVec 組み込み関数について詳しくは、AltiVec Application Programming Interface 仕様を参照してください。

vec_abs

目的

指定されたベクトルの内容の絶対値が入っているベクトルを返します。

構文

```
d=vec_abs(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector float	vector float
vector double	vector double

結果値

結果の各エレメントの値は、a の対応するエレメントの絶対値です。

vec_add

目的

指定されたベクトル同士の対応する各エレメント・セットの和が入っているベクトルを返します。

この関数は、long long ベクトルに対する演算をエミュレートします。

構文

```
d=vec_add(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

結果値

結果の各エレメントの値は、a と b の対応するエレメントの和です。整数ベクトルおよび符号なしベクトルの場合、算術計算はモジュラーです。

vec_add_u128

目的

符号なし 4 倍長ワード値を加算します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_add_u128(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

結果値

a + b の下位 128 ビットを返します。

vec_addc_u128

目的

2 つの 4 倍長ワード値の 128 ビット加算のキャリー・ビットを取得します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_addc_u128(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

結果値

a + b の桁上げを返します。

vec_adde_u128

目的

直前の演算からのキャリー・ビットを持つ、符号なし 4 倍長ワード値を加算します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_adde_u128(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

結果値

a + b + (c & 1) の下位 128 ビットを返します。

vec_addec_u128

目的

直前の演算からのキャリー・ビットを持つ、2 つの 4 倍長ワード値の 128 ビット加算のキャリー・ビットを取得します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_addec_u128(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

結果値

a + b + (c & 1) の桁上げを返します。

vec_all_eq

目的

指定されたベクトル同士の対応するエレメント・セットのすべてが、等しいかどうかをテストします。

構文

```
d=vec_all_eq(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a の各エレメントが、 b の対応するエレメントに等しければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_ge

目的

第 1 引数のすべてのエレメントが、第 2 引数の対応するエレメント以上であるかどうかをテストします。

構文

```
d=vec_all_ge(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のすべてのエレメントが、 b の対応するエレメント以上であれば、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_gt

目的

第 1 引数のすべてのエレメントが、第 2 引数の対応するエレメントより大きいかどうかをテストします。

構文

```
d=vec_all_gt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のすべてのエレメントが、 b の対応するエレメントより大きければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_le

目的

第 1 引数のすべてのエレメントが、第 2 引数の対応するエレメント以下であるかどうかをテストします。

構文

```
d=vec_all_le(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のすべてのエレメントが、 b の対応するエレメント以下であれば、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_lt

目的

第 1 引数のすべてのエレメントが、第 2 引数の対応するエレメントより小さいかどうかをテストします。

構文

```
d=vec_all_lt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のすべてのエレメントが、 b の対応するエレメントより小さければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_nan

目的

指定されたベクトルの各エレメントが NaN であるかどうかをテストします。

構文

```
d=vec_all_nan(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
int	vector float
	vector double

結果値

a の各エレメントが NaN である場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_ne

目的

指定されたベクトル同士の対応するエレメント・セットが、すべて等しくないかどうかをテストします。

構文

```
d=vec_all_ne(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a の各エレメントが、 b の対応するエレメントに等しくなければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_nge

目的

最初の引数の各エレメントが 2 番目の引数の対応するエレメント以上でないかどうかをテストします。

構文

```
d=vec_all_nge(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a の各エレメントが、 b の対応するエレメント以上でない場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_ngt

目的

第 1 引数の各エレメントが、第 2 引数の対応するエレメントより大きくないかどうかをテストします。

構文

```
d=vec_all_ngt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a の各エレメントが、 b の対応するエレメントより大きくない場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_nle

目的

第 1 引数の各エレメントが、第 2 引数の対応するエレメント以下でないかどうかをテストします。

構文

```
d=vec_all_nle(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a の各エレメントが、 b の対応するエレメント以下でない場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_nlt

目的

第 1 引数の各エレメントが、第 2 引数の対応するエレメントより小さくないかどうかをテストします。

構文

```
d=vec_all_nlt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a の各エレメントが、 b の対応するエレメントより小さくなければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_all_numeric

目的

指定されたベクトルの各エレメントが、数値である (NaN でない) かどうかをテストします。

構文

```
d=vec_all_numeric(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
int	vector float
	vector double

結果値

a の各エレメントが数値である (NaN でない) 場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_and

目的

指定されたベクトル同士のビット単位の AND 演算を行います。

構文

```
d=vec_and(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int

d	a	b
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector double
		vector bool long long

vec_andc

目的

第 1 引数と、第 2 引数のビット単位の補数をビット単位で AND 演算します。

構文

```
d=vec_andc(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short

d	a	b
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

結果値

結果は、a と、b のビット単位の補数をビット単位で AND した値です。

vec_any_eq

目的

指定されたベクトル同士の対応するエレメント・セットに、等しいものがあるかどうかをテストします。

構文

```
d=vec_any_eq(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメントに等しければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_ge

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメント以上であるかどうかをテストします。

構文

```
d=vec_any_ge(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメント以上であれば、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_gt

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメントより大きいかどうかをテストします。

構文

```
d=vec_any_gt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメントより大きければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_le

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメント以下であるかどうかをテストします。

構文

```
d=vec_any_le(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメント以下であれば、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_lt

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメントより小さいかどうかをテストします。

構文

```
d=vec_any_lt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメントより小さければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_nan

目的

指定されたベクトルのいずれかのエレメントが NaN であるかどうかをテストします。

構文

```
d=vec_any_nan(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
int	vector float
	vector double

結果値

a のいずれかのエレメントが NaN である場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_ne

目的

指定されたベクトル同士の対応するエレメント・セットに、等しくないものがあるかどうかをテストします。

構文

```
d=vec_any_ne(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメントに等しくなければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_nge

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメント以下であるかどうかをテストします。

構文

```
d=vec_any_nge(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメント以下であれば、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_ngt

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメントより大きくないかどうかをテストします。

構文

```
d=vec_any_ngt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメントより大きくなければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_nle

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメント以下でないかどうかをテストします。

構文

```
d=vec_any_nle(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメント以下でない場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_nlt

目的

第 1 引数のいずれかのエレメントが、第 2 引数の対応するエレメントより小さくないかどうかをテストします。

構文

```
d=vec_any_nlt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
int	vector float	vector float
	vector double	vector double

結果値

a のいずれかのエレメントが、b の対応するエレメントより小さくなければ、結果は 1 です。それ以外の場合、結果は 0 です。

vec_any_numeric

目的

指定されたベクトルのいずれかのエレメントが、数値である (NaN でない) かどうかをテストします。

構文

```
d=vec_any_numeric(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
int	vector float
	vector double

結果値

a のいずれかのエレメントが数値である (NaN でない) 場合、結果は 1 です。それ以外の場合、結果は 0 です。

vec_bperm

目的

指定の順序で 4 倍長ワードをまとめて 16 ビット値にします。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_bperm(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

結果値

それぞれの i (0 <= i < 16) について、index が b の i 番目のエレメントのバイト値を示すようにします。

index が 128 以上の場合、結果のビット 48+i は 0 に設定されます。

index が 128 より小さい場合、結果のビット 48+i は、入力 a の index 番目の値に設定されます。

vec_ceil

目的

指定されたベクトルの対応するエレメントの値以上で、表現可能な最小の浮動小数点整数値が入っているベクトルを返します。

注: vec_ceil は、vec_roundp のもう 1 つの名前です。詳しくは、『613 ページの『vec_roundp』』を参照してください。

vec_cmpeq

目的

指定されたベクトルの対応する各エレメント・セットについて、それらが等しいかどうかを比較した結果が入っているベクトルを返します。

この関数は、long long ベクトルに対する演算をエミュレートします。

構文

```
d=vec_cmpeq(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector bool long long	vector bool long long
	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

結果値

a およびb の対応するエレメントが等しければ、結果の各エレメントは、各ビットの値が 1 です。そうでない場合、各ビットの値は 0 です。

vec_cmpge

目的

指定されたベクトルの対応する各エレメント・セットについて、一方が他方以上であるかどうかを比較した結果が入っているベクトルを返します。

構文

```
d=vec_cmpge(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

結果値

a の対応するエレメントの値が b の対応するエレメントの値以上である場合、結果の各エレメントは、各ビットの値が 1 です。そうでない場合、各ビットの値は 0 です。

vec_cmpgt

目的

指定されたベクトルの対応する各エレメント・セットについて、一方が他方より大きいかどうかを比較した結果が入っているベクトルを返します。

この関数は、long long ベクトルに対する演算をエミュレートします。

構文

```
d=vec_cmpgt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float

d	a	b
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

結果値

結果の各エレメントについて、a の対応するエレメントの値が b の対応するエレメントの値より大きい場合、各ビットの値は 1 になります。そうでない場合、各ビットの値は 0 です。

vec_cmple

目的

指定されたベクトルの対応する各エレメント・セットについて、一方が他方以下であるかどうかを比較した結果が入っているベクトルを返します。

構文

```
d=vec_cmple(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

結果値

a の対応するエレメントの値が b の対応するエレメントの値以下である場合、結果の各エレメントは、各ビットの値が 1 です。そうでない場合、各ビットの値は 0 です。

vec_cmplt

目的

指定されたベクトルの対応する各エレメント・セットについて、一方が他方より小さいかどうかを比較した結果が入っているベクトルを返します。

この操作は、long long ベクトルに対する演算をエミュレートします。

構文

```
d=vec_cmplt(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

結果値

結果の各エレメントについて、a の対応するエレメントの値が b の対応するエレメントの値より小さい場合、各ビットの値は 1 になります。そうでない場合、各ビットの値は 0 です。

vec_cntlz

目的

入力の各エレメントの先行ゼロ・ビットの数を計算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_cntlz(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector unsigned char	vector unsigned char
vector unsigned char	vector signed char
vector unsigned short	vector unsigned short
vector unsigned short	vector signed short
vector unsigned int	vector unsigned int
vector unsigned int	vector signed int
vector unsigned long long	vector unsigned long long
vector unsigned long long	vector signed long long

結果値

結果の各エレメントは、a の対応するエレメントの先行ゼロの数に設定されます。

vec_cpsgn

目的

ベクトル a のエレメントの符号をベクトル b の対応するエレメントの符号にコピーして、ベクトルを返します。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_cpsgn(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector float	vector float	vector float
vector double	vector double	vector double

vec_ctd

目的

a の各エレメントの型を整数から浮動小数点単精度に変換し、結果を b の 2 乗で除算します。

構文

```
d=vec_ctd(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector double	vector signed int	0-31
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

vec_ctf

目的

固定小数点数のベクトルを浮動小数点数のベクトルに変換します。

構文

```
d=vec_ctf(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector float	vector signed int	0-31
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

結果値

結果の各エレメントの値は、a の対応するエレメントを b の 2 乗で除算した値に最も近い浮動小数点推定値です。

vec_cts

目的

浮動小数点数のベクトルを、符号付き固定小数点数のベクトルに変換します。

構文

```
d=vec_cts(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed int	vector float	0-31
	vector double	

結果値

結果の各エレメントの値は、a の対応するエレメントを b の 2 乗で乗算して得られた飽和値です。

vec_ctsl

目的

a の各エレメントに b の 2 乗を乗算し、結果を整数に変換します。

注: この関数は、a が double ベクトルの場合、a のエレメント 1 と 3 を使用しません。

構文

```
d=vec_ctsl(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed long long	vector float	0-31
	vector double	

vec_ctu

目的

浮動小数点数のベクトルを、符号なし固定小数点数のベクトルに変換します。

注: 結果ベクトルのエレメント 1 と 3 は、a が double ベクトルである場合、未定義です。

構文

```
d=vec_ctu(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned int	vector float	0-31
	vector double	

結果値

結果の各エレメントの値は、a の対応するエレメントを b の 2 乗で乗算して得られた飽和値です。

vec_ctul

目的

a の各エレメントに b の 2 乗を乗算し、結果を unsigned 型に変換します。

構文

```
d=vec_ctul(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned long long	vector float	0-31
	vector double	

結果値

この関数は、a が float ベクトルの場合、a のエレメント 1 と 3 を使用しません。

vec_cvf

目的

単精度浮動小数点ベクトルを倍精度浮動小数点ベクトルに変換するか、倍精度浮動小数点ベクトルを単精度浮動小数点ベクトルに変換します。

構文

```
d=vec_cvf(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector double
vector double	vector float

結果値

この関数で vector float を vector double に変換する場合、ベクトル内のエレメント 0 と 2 の型が変換されます。

この関数で vector double を vector float に変換する場合、結果ベクトル内のエレメント 1 と 3 の型は未定義です。

vec_div

目的

ベクトル a のエレメントをベクトル b の対応するエレメントによって除算し、その結果を結果ベクトル内の対応するエレメントに代入します。

この関数は整数ベクトルに対する演算をエミュレートします。この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_div(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

vec_eqv

目的

入力ベクトルのビット単位の等価演算を行います。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_eqv(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

表 51. 戻り値と関数引数のタイプ

d	a	b
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long	vector signed long long

表 51. 戻り値と関数引数のタイプ (続き)

d	a	b
vector unsigned long long	vector unsigned long long	vector bool long long
vector signed long long	vector signed long long	vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
vector signed long long	vector bool long long	vector signed long long
vector bool long long	vector bool long long	vector bool long long
vector unsigned int	vector unsigned int	vector unsigned int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector bool int
vector signed int	vector signed int	vector bool int
vector unsigned int	vector bool int	vector unsigned int
vector signed int	vector bool int	vector signed int
vector bool int	vector bool int	vector bool int
vector unsigned short	vector unsigned short	vector unsigned short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector bool short
vector signed short	vector signed short	vector bool short
vector unsigned short	vector bool short	vector unsigned short
vector signed short	vector bool short	vector signed short
vector bool short	vector bool short	vector bool short
vector unsigned char	vector unsigned char	vector unsigned char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector bool char
vector signed char	vector signed char	vector bool char
vector unsigned char	vector bool char	vector unsigned char
vector signed char	vector bool char	vector signed char
vector bool char	vector bool char	vector bool char
vector float	vector float	vector float
vector float	vector bool int	vector float
vector float	vector float	vector bool int
vector double	vector double	vector double
vector double	vector bool long long	vector double
vector double	vector double	vector bool long long

結果値

結果の各ビットは、a および b の対応するビットの、ビット単位演算 (a = b) の結果に設定されます。 0 <= i < 128 の場合、結果のビット i は、a のビット i が b のビット i と等しい場合にのみ、1 に設定されます。

vec_extract

目的

ベクトル b からエレメント a の値を返します。

構文

```
d=vec_extract(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
signed char	vector signed char	signed int
unsigned char	vector unsigned char	
	vector bool char	
signed short	vector signed short	
unsigned short	vector unsigned short	
	vector bool short	
signed int	vector signed int	
unsigned int	vector unsigned int	
	vector bool int	
signed long long	vector signed long long	
unsigned long long	vector unsigned long long	
	vector bool long long	
float	vector float	
double	vector double	

結果値

この関数は、b に対するモジュロ演算を使用して、エレメント番号を判別します。例えば、b が範囲外の場合、コンパイラーは b に対するベクトル内のエレメント数のモジュロを計算して、エレメントの位置を判別します。

vec_floor

目的

指定されたベクトルの対応するエレメントの値以下で、表現可能な最大の浮動小数点整数値が入っているベクトルを返します。

注: vec_floor は、vec_roundm のもう 1 つの名前です。詳しくは、『612 ページの『vec_roundm』』を参照してください。

vec_gbb

目的

入力に対する gather-bits-by-bytes 演算を実行します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_gbb(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long

結果値

結果の各ダブルワード・エレメントは、以下のように設定されます。x(i) (0 <= i <8) が、x(7) を最上位バイトとする、対応する入力ダブルワード・エレメントのバイト・エレメントを示すようにします。 i と j (0 <= i < 8、0 <= j < 8) の各ペアに対して、結果の i 番目のバイト・エレメントの j 番目のビットは、入力の j 番目のバイト・エレメントの i 番目のビットの値に設定されます。

vec_insert

目的

ベクトル b のコピーを、そのエレメント c の値を a に置き換えて返します。

構文

```
d=vec_insert(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector signed char	signed char	vector signed char	signed int
vector unsigned char	unsigned char	vector bool char	
		vector unsigned char	
vector signed short	signed short	vector signed short	
vector unsigned short	unsigned short	vector bool short	
		vector unsigned short	
vector signed int	signed int	vector signed int	
vector unsigned int	unsigned int	vector bool int	
		vector unsigned int	
vector signed long long	signed long long	vector signed long long	
vector unsigned long long	unsigned long long	vector bool long long	
		vector unsigned long long	
vector float	float	vector float	
vector double	double	vector double	

結果値

この関数は、c に対するモジュロ演算を使用して、エレメント番号を判別します。例えば、c が範囲外の場合、コンパイラーは c に対するベクトル内のエレメント数のモジュロを計算して、エレメントの位置を判別します。

vec_madd

目的

指定されたベクトルの対応するエレメントからなる各セットに対してFMA(fused multiply-add)を実行した結果が入ったベクトルを戻します。

構文

```
d=vec_madd(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

結果値

結果の各エレメントの値は、a と b の対応するエレメントの値の積を c の対応するエレメントの値に加算したものです。

vec_max

目的

指定されたベクトル同士の対応する各エレメント・セットから、最大値が入っているベクトルを返します。

構文

```
d=vec_max(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector float	vector float	vector float
vector double	vector double	vector double
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

結果値

結果の各エレメントの値は、a と b の対応するエレメントの最大値です。

vec_mergeh

目的

2 つのベクトルから、それぞれの最上位の半分ずつをマージします。

構文

```
d=vec_mergeh(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

結果値

各ベクトルのエレメントに 0 から始まる番号が付いているとします。結果の偶数番号が付いたエレメントは、a の最上位 8 バイト内のエレメントから順にとられます。結果の奇数番号が付いたエレメントは、b の最上位 8 バイト内のエレメントから順にとられます。

vec_mergel

目的

2 つのベクトルから、それぞれの最下位の半分ずつをマージします。

構文

```
d=vec_mergel(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

結果値

各ベクトルのエレメントに 0 から始まる番号が付いているとします。結果の偶数番号が付いたエレメントは、a の最下位 8 バイト内のエレメントから順にとられます。結果の奇数番号が付いたエレメントは、b の最下位 8 バイト内のエレメントから順にとられます。

vec_min

目的

指定されたベクトル同士の対応する各エレメント・セットから、最小値が入っているベクトルを返します。

構文

```
d=vec_min(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char

d	a	b
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector float	vector float	vector float
vector double	vector double	vector double
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

結果値

結果の各エレメントの値は、`a` と `b` の対応するエレメントの最小値です。

vec_msub

目的

指定されたベクトルを使用して、乗算-減算演算を実行した結果が入っているベクトルを返します。

この組み込み関数は、`-qarch` が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_msub(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

結果値

この関数は、a の各エレメントを b の対応するエレメントと乗算し、その結果から c の対応するエレメントを減算します。

vec_mul

目的

指定されたベクトルを使用して乗算演算を実行した結果が入っているベクトルを返します。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_mul(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

結果値

この関数は指定されたベクトル同士の対応するエレメントを乗算し、その結果を結果ベクトル内の対応するエレメントに代入します。

vec_nabs

目的

指定されたベクトルを使用して負の絶対値演算を実行した結果が入っているベクトルを返します。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_nabs(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

結果値

この関数は、指定されたベクトル内の各エレメントの絶対値を計算し、その結果の負の値を結果ベクトル内の対応するエレメントに代入します。

vec_nand

目的

入力ベクトルのビット単位の negated-and 演算を行います。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_nand(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

表 52. 戻り値と関数引数のタイプ

d	a	b
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector bool long long
vector signed long long	vector signed long long	vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
vector signed long long	vector bool long long	vector signed long long
vector bool long long	vector bool long long	vector bool long long
vector unsigned int	vector unsigned int	vector unsigned int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector bool int
vector signed int	vector signed int	vector bool int
vector unsigned int	vector bool int	vector unsigned int

表 52. 戻り値と関数引数のタイプ (続き)

d	a	b
vector signed int	vector bool int	vector signed int
vector bool int	vector bool int	vector bool int
vector unsigned short	vector unsigned short	vector unsigned short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector bool short
vector signed short	vector signed short	vector bool short
vector unsigned short	vector bool short	vector unsigned short
vector signed short	vector bool short	vector signed short
vector bool short	vector bool short	vector bool short
vector unsigned char	vector unsigned char	vector unsigned char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector bool char
vector signed char	vector signed char	vector bool char
vector unsigned char	vector bool char	vector unsigned char
vector signed char	vector bool char	vector signed char
vector bool char	vector bool char	vector bool char
vector float	vector float	vector float
vector float	vector bool int	vector float
vector float	vector float	vector bool int
vector double	vector double	vector double
vector double	vector bool long long	vector double
vector double	vector double	vector long long

結果値

結果の各ビットは、a および b の対応するビットの、ビット単位操作 !(a & b) の結果に設定されます。 0 <= i < 128 の場合、結果のビット i は、a および b の i 番目のビットが 1 である場合にのみ、0 に設定されます。

vec_neg

目的

指定されたベクトル内の対応するエレメントの負の値が入ったベクトルを返します。

注: vector signed long long の場合、この関数は演算をエミュレートします。この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_neg(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector signed long long	vector signed long long
vector float	vector float
vector double	vector double

結果値

この関数は、指定されたベクトル内の各エレメントの値に -1.0 を乗算し、その結果を結果ベクトル内の対応するエレメントに代入します。

vec_nmadd

目的

指定されたベクトルに対して負の乗算-加算演算を実行した結果が入っているベクトルを返します。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_nmadd(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector double	vector double	vector double	vector double
vector float	vector float	vector float	vector float

結果値

結果の各エレメントの値は、a と b の対応するエレメントの積を、c の対応するエレメントと加算した後、-1.0 を乗算したものです。

vec_nmsub

目的

指定されたベクトルに対して負の乗算-減算演算を実行した結果が入っているベクトルを返します。

構文

```
d=vec_nmsub(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

結果値

結果の各エレメントの値は、a と b の対応するエレメントの値の積を、c の対応するエレメントから減算したものです。

vec_nor

目的

指定されたベクトル同士のビット単位の NOR 演算を行います。

構文

```
d=vec_nor(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int

d	a	b
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector bool int	vector float
	vector float	vector bool int
vector double	vector double	vector double

結果値

結果は、a と b のビット単位の NOR です。

vec_or

目的

指定されたベクトル同士のビット単位の OR 演算を行います。

構文

```
d=vec_or(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short

d	a	b
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

結果値

結果は、a と b のビット単位の OR です。

vec_orc

目的

入力ベクトルのビット単位の OR-with-complement 演算を行います。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_orc(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

表 53. 戻り値と関数引数のタイプ

d	a	b
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector bool long long
vector signed long long	vector signed long long	vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
vector signed long long	vector bool long long	vector signed long long
vector bool long long	vector bool long long	vector bool long long
vector unsigned int	vector unsigned int	vector unsigned int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector bool int
vector signed int	vector signed int	vector bool int
vector unsigned int	vector bool int	vector unsigned int
vector signed int	vector bool int	vector signed int
vector bool int	vector bool int	vector bool int
vector unsigned short	vector unsigned short	vector unsigned short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector bool short
vector signed short	vector signed short	vector bool short
vector unsigned short	vector bool short	vector unsigned short
vector signed short	vector bool short	vector signed short
vector bool short	vector bool short	vector bool short
vector unsigned char	vector unsigned char	vector unsigned char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector bool char
vector signed char	vector signed char	vector bool char
vector unsigned char	vector bool char	vector unsigned char
vector signed char	vector bool char	vector signed char
vector bool char	vector bool char	vector bool char
vector float	vector float	vector float
vector float	vector bool	vector float
vector float	vector float	vector bool int
vector double	vector double	vector double
vector double	vector bool long long	vector double
vector double	vector double	vector bool long long

結果値

結果の各ビットは、a および b の対応するビットの、ビット単位操作 (a | ~b) の結果に設定されます。 0 <= i < 128 の場合、結果のビット i は、a の i 番目のビットが 1 であるか、または b の i 番目のビットが 0 である場合にのみ、1 に設定されます。

vec_pack

目的

2 つのベクトルの各エレメントの情報を圧縮して、結果ベクトルに入れます。

構文

```
d=vec_pack(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed short	vector signed short
vector unsigned char	vector unsigned short	vector unsigned short
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int
vector signed long	vector signed long long	vector signed long long
vector unsigned long	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

結果値

結果ベクトルの各エレメントの値は、a と b を連結した結果の、対応するエレメントの下位半分から取られます。

vec_packs

目的

2 つのベクトルの各エレメントの情報を圧縮し、飽和値を使用して、結果ベクトルに入れます。

構文

```
d=vec_packs(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed short	vector signed short

d	a	b
vector unsigned char	vector unsigned short	vector unsigned short
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int
vector signed int	vector signed long long	vector signed long long
vector unsigned int	vector unsigned long long	vector unsigned long long

結果値

結果ベクトルの各エレメントの値は、a と b を連結した結果の、対応するエレメントの飽和値です。

vec_packsu

目的

2 つのベクトルの各エレメントの情報を圧縮し、飽和値を使用して、結果ベクトルに入れます。

構文

```
d=vec_packsu(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned char	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector unsigned short	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector unsigned int	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

結果値

結果ベクトルの各エレメントの値は、a と b を連結した結果の、対応するエレメントの飽和値です。

vec_permi

目的

a と b の 2 つの 8 バイト長ベクトル・エレメントを、c の値に基づいて順序を変えて結合することにより、1 つのベクトルを返します。

構文

```
d=vec_permi(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector bool long long	vector bool long long	vector bool long long	0-3
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector double	vector double	vector double	

結果値

`a[0]` と `a[1]` を使用して `a` 内の 1 番目と 2 番目の 8 バイト長エレメントを表し、`b` 内のエレメントに `b[0]` と `b[1]` を使用した場合、この関数は `c` のバイナリー値に基づいて、結果ベクトル内のエレメントを決定します。その例を以下に示します。

- 00 - `a[0]`, `b[0]`
- 01 - `a[0]`, `b[1]`
- 10 - `a[1]`, `b[0]`
- 11 - `a[1]`, `b[1]`

vec_popcnt

目的

入力の各エレメントの取り込みカウント (設定されたビット数) を計算します。

この組み込み関数は、`-qarch` が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_popcnt(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector unsigned char	vector signed char
vector unsigned char	vector unsigned char
vector unsigned short	vector signed short
vector unsigned short	vector unsigned short
vector unsigned int	vector signed int
vector unsigned int	vector unsigned int
vector unsigned long long	vector signed long long
vector unsigned long long	vector unsigned long long

結果値

結果の各エレメントは、入力の対応するエレメント内の設定されたビット数に設定されます。

vec_promote

目的

エレメント位置 b に a が入ったベクトルを返します。

構文

```
d=vec_promote(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	signed char	signed int
vector unsigned char	unsigned char	
vector signed short	signed short	
vector unsigned short	unsigned short	
vector signed int	signed int	
vector unsigned int	unsigned int	
vector signed long long	signed long long	
vector unsigned long long	unsigned long	
vector float	float	
vector double	double	

結果値

結果は、エレメント位置 b に a が入ったベクトルです。この関数は、b に対するモジュロ演算を使用して、エレメント番号を判別します。例えば、b が範囲外の場合、コンパイラーは b に対するベクトル内のエレメント数のモジュロを計算して、エレメントの位置を判別します。それ以外のベクトルのエレメントは未定義です。

vec_re

目的

指定されたベクトルの対応するエレメントの逆数の推定値が入っているベクトルを返します。

構文

```
d=vec_re(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

結果値

結果の各エレメントには、a の対応するエレメントの逆数の推定値が入っています。

vec_rl

目的

ベクトルの各エレメントを、指定されたビット数だけ左に回転させます。

構文

```
d=vec_rl(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

結果値

結果の各エレメントは、a の対応するエレメントを、b の対応するエレメントで指定されたビット数だけ左に回転することで得られます。

vec_round

目的

指定されたベクトルの対応するエレメントを丸めた値が入っているベクトルを返します。

構文

```
d=vec_round(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

結果値

結果の各エレメントには、a の対応するエレメントを、IEEE の最近隣丸め (round-to-nearest) を使用して表現可能な最も近い浮動小数点整数に丸めた値が入っています。

注: この関数は、**-qstrict=nooperationprecision** コンパイラー・オプションを指定した場合、丸め時のタイの解決に関する厳密な操作定義に従わない場合があります。

vec_roundc

目的

指定されたベクトル内のすべての単精度または倍精度浮動小数点エレメントを整数に丸めることにより、ベクトルを返します。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_roundc(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

vec_roundm

目的

指定されたベクトルの対応するエレメントの値以下で、表現可能な最大の浮動小数点整数値が入っているベクトルを返します。

注: vec_roundm は、vec_floor のもう 1 つの名前です。詳しくは、『592 ページの『vec_floor』』を参照してください。

構文

```
d=vec_roundm(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

vec_roundp

目的

指定されたベクトルの対応するエレメントの値以上で、表現可能な最小の浮動小数点整数値が入っているベクトルを返します。

注: vec_roundp は、vec_ceil のもう 1 つの名前です。詳しくは、『 581 ページの『vec_ceil』』を参照してください。

構文

```
d=vec_roundp(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

vec_roundz

目的

指定されたベクトルの対応するエレメントに切り捨てを実行した値が入っているベクトルを返します。

注: vec_roundz は、vec_trunc のもう 1 つの名前です。詳しくは、『 623 ページの『vec_trunc』』を参照してください。

構文

```
d=vec_roundz(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

結果値

結果の各エレメントには、a の対応するエレメントを整数値までに切り捨てた値が入っています。

vec_rsqrte

目的

指定されたベクトルの対応するエレメントの平方根逆数の推定値が入っているベクトルを返します。

構文

```
d=vec_rsqrte(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

結果値

結果の各エレメントには、a の対応するエレメントの平方根逆数の推定値が入っています。

vec_sel

目的

c の値に応じて、a または b の値が入ったベクトルを返します。

構文

```
d=vec_sel(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector bool char	vector bool char	vector bool char	vector bool char
			vector unsigned char
vector signed char	vector signed char	vector signed char	vector bool char
			vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char	vector bool char
			vector unsigned char

d	a	b	c
vector bool short	vector bool short	vector bool short	vector bool short
			vector unsigned short
vector signed short	vector signed short	vector signed short	vector bool shot
			vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short	vector bool short
			vector unsigned short
vector bool int	vector bool int	vector bool int	vector bool int
			vector unsigned int
vector signed int	vector signed int	vector signed int	vector bool int
			vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int	vector bool int
			vector unsigned int
vector bool long long	vector bool long long	vector bool long long	vector bool long long
			vector unsigned long long
vector signed long long	vector signed long long	vector signed long long	vector bool long long
			vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector bool long long
			vector unsigned long long
vector float	vector float	vector float	vector bool int
			vector unsigned int
vector double	vector double	vector double	vector bool long long
			vector unsigned long long

結果値

結果ベクトルの各ビットは、`c` の対応するビットが `0` の場合は `a` の対応するビットの値になり、それ以外の場合は `b` の対応するビットの値になります。

vec_sl

目的

ベクトルの各エレメントに対して、左シフトを実行します。

構文

```
d=vec_sl(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

結果値

結果ベクトルの各エレメントは、a の対応するエレメントを、b の対応するエレメントの値とそのエレメント内のビット数によるモジュロで指定されるビット数だけ、左シフトした結果です。シフトアウトされたビットは、ゼロで置き換えられます。

vec_sldw

目的

Shift Left Double by Word Immediate

a と b を連結した後、結果ベクトルを 4 バイトの倍数だけ左シフトすることにより、ベクトルを返します。c は、シフト操作のオフセットを指定します。

構文

```
d=vec_sldw(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector bool char	vector bool char	vector bool char	0-3
vector signed char	vector signed char	vector signed char	
vector unsigned char	vector unsigned char	vector unsigned char	
vector bool short	vector bool short	vector bool short	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
vector bool int	vector bool int	vector bool int	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
vector bool long long	vector bool long long	vector bool long long	
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector float	vector float	vector float	
vector double	vector double	vector double	

結果値

c によって指定された 4 バイトの倍数分、連結した a および b を左シフトした後に、この関数は、左端の 4 つの 4 バイト値を使用して、結果ベクトルを形成します。

vec_splat

目的

すべてのエレメントが、指定された値にセットされているベクトルを返します。

構文

```
d=vec_splat(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	0 - 15
vector signed char	vector signed char	0 - 15
vector unsigned char	vector unsigned char	0 - 15
vector bool short	vector bool short	0 - 7
vector signed short	vector signed short	0 - 7

d	a	b
vector unsigned short	vector unsigned short	0 - 7
vector bool int	vector bool int	0-3
vector signed int	vector signed int	0-3
vector unsigned int	vector unsigned int	0-3
vector bool long long	vector bool long long	0-1
vector signed long long	vector signed long long	0-1
vector unsigned long long	vector unsigned long long	0-1
vector float	vector float	0-3
vector double	vector double	0-1

結果値

結果の各エレメントの値は、b によって指定された a のエレメントの値です。

vec_splats

目的

各エレメントの値が a に設定されたベクトルを返します。

構文

```
d=vec_splats(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector signed char	signed char
vector unsigned char	unsigned char
vector signed short	signed short
vector unsigned short	unsigned short
vector signed int	signed int
vector unsigned int	unsigned int
vector signed long long	signed long long
vector unsigned long long	unsigned long long
vector float	float
vector double	double

vec_sqrt

目的

指定されたベクトルの各エレメントの平方根が入っているベクトルを返します。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_sqrt(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector float	vector float
vector double	vector double

vec_sr

目的

ベクトルの各エレメントに対して、右シフトを実行します。

構文

```
d=vec_sr(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

結果値

結果ベクトルの各エレメントは、a の対応するエレメントを、b の対応するエレメントの値とそのエレメント内のビット数によるモジュロで指定されるビット数だけ、右シフトした結果です。シフトアウトされたビットは、ゼロで置き換えられます。

vec_sra

目的

ベクトルの各エレメントに対して、算術右シフトを実行します。

構文

```
d=vec_sra(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

結果値

結果ベクトルの各エレメントは、a の対応するエレメントを、b の対応するエレメントの値とそのエレメント内のビット数によるモジュロで指定されるビット数だけ、算術右シフトした結果です。シフトアウトされたビットは、a のエレメントの最上位ビットのコピーによって置き換えられます。

vec_sub

目的

b の各エレメントを、a の対応するエレメントから減算した結果が入ったベクトルを返します。

この関数は、long long ベクトルに対する演算をエミュレートします。

構文

```
d=vec_sub(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long

d	a	b
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

結果値

結果の各エレメントの値は、b の対応するエレメントの値を、a の対応するエレメントの値から減算した結果です。整数ベクトルの場合、この算術計算はモジュラーです。

vec_sub_u128

目的

符号なし 4 倍長ワード値を減算します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_sub_u128(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

結果値

a - b の下位 128 ビットを返します。

vec_subc_u128

目的

2 つの 4 倍長ワード値の 128 ビット減算のキャリー・ビットを取得します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、-qarch が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_subc_u128(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char

結果値

`a - b` の桁上げを返します。

vec_sube_u128

目的

直前の演算からのキャリー・ビットを持つ、符号なし 4 倍長ワード値を減算します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、`-qarch` が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_sube_u128(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

結果値

`a - b - (c & 1)` の下位 128 ビットを返します。

vec_subec_u128

目的

直前の演算からのキャリー・ビットを持つ、2 つの 4 倍長ワード値の 128 ビット減算のキャリー・ビットを取得します。

この関数は、ベクトルを 128 ビットの符号なし整数として演算します。

この組み込み関数は、`-qarch` が POWER8 プロセッサーをターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_subec_u128(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b	c
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char

結果値

`a - b - (c & 1)` の桁上げを返します。

vec_trunc

目的

指定されたベクトルの対応するエレメントに切り捨てを実行した値が入っているベクトルを返します。

注: `vec_trunc` は、`vec_roundz` のもう 1 つの名前です。詳しくは、『613 ページの『vec_roundz』』を参照してください。

vec_unpackh

目的

ベクトルの最上位の半分をアンパックして、より大きいエレメントからなるベクトルに入れます。

構文

```
d=vec_unpackh(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector signed short	vector signed char
vector signed int	vector signed short
vector signed long long	vector signed int
vector bool long long	vector bool int

結果値

結果の各エレメントの値は、`a` の最上位の半分の対応するエレメントの値です。

vec_unpackl

目的

ベクトルの最下位の半分をアンパックして、より大きいエレメントからなるベクトルに入れます。

構文

```
d=vec_unpackl(a)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a
vector signed short	vector signed char
vector signed int	vector signed short
vector signed long long	vector signed int
vector bool long long	vector bool int

結果値

結果の各エレメントの値は、a の最下位の半分の対応するエレメントの値です。

vec_xld2

目的

16 バイトのベクトルを、変位 a およびポインター b によって指定したメモリー・アドレスにある 2 つの 8 バイト・エレメントからロードします。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_xld2(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

注: 以下の表内のオペランド a の型は、32 ビット・モードでは int、64 ビット・モードでは long です。

d	a	b
vector signed char	int	signed char *
	long	
vector unsigned char	int	unsigned char *
	long	
vector signed short	int	signed short *
	long	
vector unsigned short	int	unsigned short *
	long	
vector signed int	int	signed int *
	long	

d	a	b
vector unsigned int	int	unsigned int *
	long	
vector signed long long	int	signed long long *
	long	
vector unsigned long long	int	unsigned long long *
	long	
vector float	int	float *
	long	
vector double	int	double *
	long	

結果値

この関数は変位とポインター R 値を加算して、ロード操作用のアドレスを取得します。影響を受けるアドレスを 16 バイトの倍数まで切り捨てることはしません。

vec_xlds

目的

変位 a およびポインター b によって指定したメモリー・アドレスから 8 バイトのエレメントをロードし、それをベクトルにスプラッティングします。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_xlds(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

注: 以下の表内のオペランド a の型は、32 ビット・モードでは int、64 ビット・モードでは long です。

d	a	b
vector signed long long	int	signed long long *
	long	
vector unsigned long long	int	unsigned long long *
	long	
vector double	int	double *
	long	

結果値

この関数は変位とポインター R 値を加算して、ロード操作用のアドレスを取得します。影響を受けるアドレスを 16 バイトの倍数まで切り捨てることはしません。

vec_xlw4

目的

16 バイトのベクトルを、変位 a およびポインター b によって指定したメモリー・アドレスにある 4 つの 4 バイト・エレメントからロードします。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_xlw4(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

注: 以下の表内のオペランド a の型は、32 ビット・モードでは int、64 ビット・モードでは long です。

d	a	b
vector signed char	int	signed char *
	long	
vector unsigned char	int	unsigned char *
	long	
vector signed short	int	signed short *
	long	
vector unsigned short	int	unsigned short *
	long	
vector signed int	int	signed int *
	long	
vector unsigned int	int	unsigned int *
	long	
vector float	int	float *
	long	

結果値

この関数は変位とポインター R 値を加算して、ロード操作用のアドレスを取得します。影響を受けるアドレスを 16 バイトの倍数まで切り捨てることはしません。

vec_xor

目的

指定されたベクトル同士のビット単位の XOR 演算を行います。

構文

```
d=vec_xor(a, b)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float

d	a	b
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

結果値

結果は、a と b のビット単位の XOR です。

vec_xstd2

目的

16 バイトのベクトル a を 2 つの 8 バイト・エレメントとして、変位 b およびポインター c によって指定したメモリー・アドレスに書き込みます。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_xstd2(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

注: 以下の表内のオペランド a の型は、32 ビット・モードでは int、64 ビット・モードでは long です。

d	a	b	c
void	vector signed char	int	signed char *
		long	
	vector unsigned char	int	unsigned char *
		long	
	vector signed short	int	signed short *
		long	
	vector unsigned short	int	unsigned short *
		long	
	vector signed int	int	signed int *
		long	
	vector unsigned int	int	unsigned int *
		long	
	vector signed long long	int	signed long long *
		long	
	vector unsigned long long	int	unsigned long long *
		long	
	vector float	int	float *
		long	
	vector double	int	double *
		long	
	vector pixel	int	signed short *
			unsigned short *
		long	signed short *
			unsigned short *

結果値

この関数は変位とポインター R 値を加算して、保管操作用のアドレスを取得します。影響を受けるアドレスを 16 バイトの倍数まで切り捨てることはしません。

vec_xstw4

目的

16 バイトのベクトル a を 4 つの 4 バイト・エレメントとして、変位 b およびポインター c によって指定したメモリー・アドレスに書き込みます。

この組み込み関数は、-qarch が POWER7 プロセッサー以上をターゲットとするよう設定されている場合にのみ有効です。

構文

```
d=vec_xstw4(a, b, c)
```

結果と引数の型

以下の表で、戻り値と関数引数の型を説明します。

注: 以下の表内のオペランド `b` の型は、32 ビット・モードでは `int`、64 ビット・モードでは `long` です。

d	a	b	c
void	vector signed char	int	signed char *
		long	
	vector unsigned char	int	unsigned char *
		long	
	vector signed short	int	signed short *
		long	
	vector unsigned short	int	unsigned short *
		long	
	vector signed int	int	signed int *
		long	
	vector unsigned int	int	unsigned int *
		long	
	vector float	int	float *
		long	
	vector pixel	int	signed short *
			unsigned short *
		long	signed short *
			unsigned short *

結果値

この関数は変位とポインター R 値を加算して、保管操作用のアドレスを取得します。影響を受けるアドレスを 16 バイトの倍数まで切り捨てることはしません。

GCC アトミック・メモリー・アクセス組み込み関数 (IBM 拡張)

このセクションでは、アトミック・メモリー・アクセス組み込み関数についての参照情報を示します。これらの関数の動作は、GNU Compiler Collection (GCC) によって提供されるものと対応しています。複数のスレッドを持つプログラムでこれらの関数を使用することで、あるスレッドのデータを他のスレッドからの干渉を受けずに、アトミックかつ安全に変更できます。

これらの組み込み関数は、ホスト・マシンに搭載されているプロセッサー数に関係なく、データをアトミックに操作します。

各関数のプロトタイプでは、パラメーター型 *T*、*U*、および *V* をポインターまたは整数型にすることができます。*U* および *V* は実数の浮動小数点型にすることもできますが、これは *T* が整数型の場合のみです。以下の表は、これらの組み込み関数でサポートされている整数型および浮動小数点型を示しています。

表 54. サポートされている整数データ型

signed char	unsigned char
short int	unsigned short int
int	unsigned int
long int	unsigned long int
long long int 1	unsigned long long int 1
C++ bool	C _Bool
1 **制限:** この型は 64 ビット・プラットフォームでのみサポートされます。	

表 55. サポートされている浮動小数点データ型

float
double
long double

各関数のプロトタイプにおける省略符号 (...) は、オプションのパラメーターのリストを表しています。 XL C/C++ はこれらのオプションのパラメーターを無視し、グローバルにアクセス可能なすべての変数を保護します。

GCC アトミック・メモリー・アクセス組み込み関数は、以下のカテゴリーにグループ化されます。

アトミック・ロック、解放、および同期化の関数

__sync_lock_test_and_set

目的

この関数は、*__v* の値を *__p* が指す変数にアトミックに代入します。

この関数が呼び出されると、取得メモリー・バリアが作成されます。

プロトタイプ

T __sync_lock_test_and_set (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
設定される変数のポインター。

__v
__p が指す変数に設定する値。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_lock_release

目的

この関数は、__sync_lock_test_and_set 関数によって取得されたロックを解放して、*__p* が指す変数に値ゼロを代入します。

この関数が呼び出されると、リリース・メモリー・バリアが作成されます。

プロトタイプ

```
void __sync_lock_release (T* __p, ...);
```

パラメーター

__p
設定される変数のポインター。

__sync_synchronize

目的

この関数は、すべてのスレッド内のデータを同期化します。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

```
void __sync_synchronize ();
```

アトミック・フェッチおよび演算関数

__sync_fetch_and_and

目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位 AND 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

```
T __sync_fetch_and_and (T* __p, U __v, ...);
```

パラメーター

__p
ビット単位 AND 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位 AND 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_fetch_and_nand
目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位 NAND 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

```
T __sync_fetch_and_nand (T* __p, U __v, ...);
```

パラメーター

__p
ビット単位 NAND 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位 NAND 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_fetch_and_or
目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位包含 OR (包含論理和) 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

```
T __sync_fetch_and_or (T* __p, U __v, ...);
```

パラメーター

__p
ビット単位包含 OR (包含論理和) 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位包含 OR (包含論理和) 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_fetch_and_xor

目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位排他 OR 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

```
T __sync_fetch_and_xor (T* __p, U __v, ...);
```

パラメーター

__p
ビット単位排他 OR 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位排他 OR 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_fetch_and_add

目的

この関数は、*__v* の値を *__p* が指す変数にアトミックに加算します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

```
T __sync_fetch_and_add (T* __p, U __v, ...);
```

パラメーター

__p
__v を加算する対象の変数のポインター。この変数の値は、add 演算の結果に変更されます。

__v
__p が指す変数に加算する値を持つ変数。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_fetch_and_sub

目的

この関数は、*__v* の値を *__p* が指す変数からアトミックに減算します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_fetch_and_sub (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
__v を減算する対象の変数のポインター。この変数の値は、sub 演算の結果に変更されます。

__v
__p が指す変数から減算する値を持つ変数。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

アトミック演算およびフェッチの関数

__sync_and_and_fetch

目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位 AND 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_and_and_fetch (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
ビット単位 AND 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位 AND 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の新しい値を戻します。

__sync_nand_and_fetch

目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位 NAND 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_nand_and_fetch (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
ビット単位 NAND 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位 NAND 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の新しい値を戻します。

__sync_or_and_fetch

目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位包含 OR (包含論理和) 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_or_and_fetch (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
ビット単位包含 OR (包含論理和) 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位包含 OR (包含論理和) 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の新しい値を戻します。

__sync_xor_and_fetch

目的

この関数は、*__p* が指す変数を使用して、変数 *__v* に対してアトミック・ビット単位排他 OR 演算を実行します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_xor_and_fetch (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
ビット単位排他 OR 演算を実行する対象の変数のポインター。この変数の値は、演算の結果に変更されます。

__v
ビット単位排他 OR 演算を実行する際に使用される変数。

戻り値

この関数は、*__p* が指す変数の新しい値を戻します。

__sync_add_and_fetch

目的

この関数は、*__v* の値を *__p* が指す変数にアトミックに加算します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_add_and_fetch (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
__v を加算する対象の変数のポインター。この変数の値は、add 演算の結果に変更されます。

__v
__p が指す変数に加算する値を持つ変数。

戻り値

この関数は、*__p* が指す変数の新しい値を戻します。

__sync_sub_and_fetch

目的

この関数は、*__v* の値を *__p* が指す変数からアトミックに減算します。結果は、*__p* によって指定されるアドレスに格納されます。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_sub_and_fetch (*T** *__p*, *U* *__v*, ...);

パラメーター

__p
__v を減算する対象の変数のポインター。この変数の値は、sub 演算の結果に変更されます。

__v
__p が指す変数から減算する値を持つ変数。

戻り値

この関数は、*__p* が指す変数の新しい値を戻します。

アトミック比較および交換関数

__sync_val_compare_and_swap

目的

この関数は、*__compVal* の値を *__p* が指す変数の値と比較します。これらの値が同じである場合は、値 *__exchVal* が、*__p* によって指定されたアドレスに保管されます。そうでない場合は、操作は実行されません。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

T __sync_val_compare_and_swap (*T** *__p*, *U* *__compVal*, *V* *__exchVal*, ...);

パラメーター

__p
値を比較する変数へのポインター。

__compVal
__p が指す変数の値と比較する値。

__exchVal
__p が指すアドレスに保管される値。

戻り値

この関数は、*__p* が指す変数の初期値を戻します。

__sync_bool_compare_and_swap

目的

この関数は、*__compVal* の値を *__p* が指す変数の値と比較します。これらの値が同じである場合は、値 *__exchVal* が、*__p* によって指定されたアドレスに保管されます。そうでない場合は、操作は実行されません。

この関数が呼び出されると、フル・メモリー・バリアが作成されます。

プロトタイプ

bool __sync_bool_compare_and_swap (*T** *__p*, *U* *__compVal*, *V* *__exchVal*, ...);

パラメーター

__p
値を比較する変数へのポインター。

__compVal
__p が指す変数の値と比較する値。

__exchVal
__p が指すアドレスに保管される値。

戻り値

__compVal の値と *__p* が指す変数の値が同じ場合、関数は `true` を返します。それ以外の場合は、`false` を返します。

各種組み込み関数

各種関数は以下のカテゴリーにグループ化されています。

- 『最適化に関連した関数』
- 640 ページの『レジスターへの移動またはレジスターからの移動関数』
- 643 ページの『メモリー関連の関数』

最適化に関連した関数

__alignx

目的

pointer によってポイントされたデータが既知のコンパイル時オフセットで位置合わせされることをコンパイラーに通知することによって自動ベクトル化などの最適化を許可します。

プロトタイプ

void __alignx (int *alignment*, const void* *pointer*);

パラメーター

位置合わせ
ゼロより大きく 2 乗の値を持つ定数整数でなければなりません。

__builtin_expect

目的

指定した値に対して式が評価を行う可能性があることを示します。コンパイラーはこれを認識することで最適化を指示する場合があります。

プロトタイプ

long __builtin_expect (long *expression*, long *value*);

パラメーター

expression
整数型の式でなければなりません。

value
定数リテラルでなければなりません。

使用法

expression が予測された値に対して実行時に実際に評価を行わない場合、パフォーマンスが低下します。そのため、この組み込み関数を使用する場合は、注意が必要です。

__fence

目的

コードの動きまたはマシン・インストラクションの再配列にかかわるコンパイラーの最適化に対するバリアとして動作します。コンパイラーの最適化は、`__fence` 呼び出しのロケーションを過ぎてマシン・インストラクションを移動することはありません。

プロトタイプ

void __fence (void);

例

この関数は、最適化が使用可能である場合、コンパイラーが生成するオブジェクト・コード内の命令の順序付けを保証するのに役立ちます。

レジスターへの移動またはレジスターからの移動関数

__mftb

目的

時間基準からの移動。

32 ビット・コンパイル・モードでは、時間基準レジスターの下位のワードを戻します。64 ビット・モードでは、時間基準レジスターのダブルワード全体を戻します。

プロトタイプ

```
unsigned long __mftb (void);
```

使用法

32 ビット・モードでは、この関数は __mftbu 組み込み関数と共に使用して時間基準レジスター全体を読み取ることができます。64 ビット・モードでは、時間基準レジスターのダブルワード全体が戻されるので __mftbu を別々に使用する必要はありません。

__fence 組み込み関数を __mftb 組み込み関数の前後に挿入することをお勧めします。

__mftbu

目的

上位の時間基準からの移動。

時間基準レジスターの上位ワードを戻します。

プロトタイプ

```
unsigned int __mftbu (void);
```

使用法

32 ビット・モードでは、この関数を __mftb 組み込み関数と共に使用して時間基準レジスター全体を読み取ることができます。

__fence 組み込み関数を __mftbu 組み込み関数の前後に挿入することをお勧めします。

__mfmsr

目的

マシン状態レジスターからの移動。

マシン状態レジスター (MSR) の内容を、指定された汎用レジスターの 32 から 63 ビットへ移動します。

プロトタイプ

```
unsigned long __mfmsr (void);
```

使用法

この命令の実行は特権が付いており、監視プログラム・モードのみに制限されます。

__mfspr

目的

特殊目的レジスターからの移動。

指定された特殊目的レジスターの値を戻します。

プロトタイプ

```
unsigned long __mfspr (const int registerNumber);
```

パラメーター

registerNumber
値が戻される特殊目的レジスターの数。 *registerNumber* はコンパイル時に既知でなければなりません。

__mtmsr

目的

マシン状態レジスターへの移動。

指定された GPR の 32 ビットから 62 ビットの内容を、MSR に移動します。

プロトタイプ

```
void __mtmsr (unsigned long value);
```

パラメーター

value
value のビット 48 と 49 のビット単位 OR 結果が MSR_{48} に配置されます。
value のビット 58 と 49 のビット単位 OR 結果が MSR_{58} に配置されます。
value のビット 59 と 49 のビット単位 OR 結果が MSR_{59} に配置されます。
value のビット 32:47、49:50、52:57、および 60:62 が MSR の対応するビットに配置されます。

使用法

この命令の実行は特権が付いており、監視プログラム・モードのみに制限されます。

__mtspr

目的

特殊目的レジスターへの移動。

特殊目的レジスターの値を設定します。

プロトタイプ

```
void __mtspr(const int registerNumber, unsigned long value);
```

パラメーター

registerNumber
値が設定される特殊目的レジスターの数。 *registerNumber* はコンパイル時に既知でなければなりません。

value
コンパイル時に既知でなければなりません。

メモリー関連の関数

__alloca

目的

オブジェクトのスペースを割り振ります。割り振られたスペースはスタックに入れられ、呼び出し関数が戻るときに解放されます。

プロトタイプ

```
void* __alloca (size_t size)
```

パラメーター

size
バイト単位で割り振られて測定される、スペースの量を表す整数。

__builtin_frame_address、__builtin_return_address

目的

現行関数、またはその呼び出し側の 1 つのスタック・フレームのアドレス、または戻りアドレスを戻します。

プロトタイプ

```
void* __builtin_frame_address (unsigned int level);

void* __builtin_return_address (unsigned int level);
```

パラメーター

level
呼び出しスタックをスキャンするフレームの数を示す定数リテラルです。*level* は 0 から 63 の範囲です。0 の値は現行関数のフレームまたは戻りアドレスを戻し、1 の値は現行関数などの呼び出し側のフレームまたは戻りアドレスを戻します。

戻り値

スタックの先頭に到達すると、0 を戻します。インライン化などの最適化が、追加のスタック・フレームが導入されたり、または予期されたよりスタック・フレーム数が少なくなることにより、予期された戻り値に影響することがあります。関数がインライン化されている場合、フレームまたは戻りアドレスは戻り先の関数のフレーム・アドレスに対応します。

__mem_delay

目的

__mem_delay 組み込み関数は、特定のロードにどれだけの遅延サイクルがあるかを指定します。これらの特定のロードは、キャッシュ・ミスによる長時間のメモリー・アクセス待ち時間を伴う delinquent ロード (キャッシュ・ミス頻発ロード) です。

どのロードが delinquent ロード (キャッシュ・ミス頻発ロード) であるかを指定すると、コンパイラーはその情報を使用して、データ・プリフェッチなどの最適化を行います。さらに、**-qprefetch=assistthread** を実行すると、コンパイラーは delinquent ロード (キャッシュ・ミス頻発ロード) 情報を使用して分析を行い、プリフェッチ支援スレッドを生成します。詳しくは、『 305 ページの『-qprefetch』』を参照してください。

プロトタイプ

void* __mem_delay (const void *address, const unsigned int *cycles*);

パラメーター

address
ロードまたは保管するデータのアドレス。

cycles
コンパイル時定数。一般的には L1 ミス待ち時間または L2 ミス待ち時間。

使用法

__mem_delay 組み込み関数は、指定したメモリー参照を含んでいるステートメントの直前に置かれます。

例

以下に、**__mem_delay** で支援スレッドを使用したコードの生成方法を示します。

初期のコード:

```
int y[64], x[1089], w[1024];

  void foo(void){
    int i, j;
    for (i = 0; i &l; 64; i++) {
      for (j = 0; j < 1024; j++) {

        /* what to prefetch? y[i]; inserted by the user */
        __mem_delay(&y[i], 10);
        y[i] = y[i] + x[i + j] * w[j];
        x[i + j + 1] = y[i] * 2;
    }
  }
}
```

支援スレッド生成コード:

```
void foo@clone(unsigned thread_id, unsigned version)

{ if (!1) goto lab_1;
```

```
/* version control to synchronize assist and main thread */
if (version == @2version0) goto lab_5;

goto lab_1;

lab_5:

@CIV1 = 0;

do { /* id=1 guarded */ /* ~2 */

if (!1) goto lab_3;

@CIV0 = 0;

do { /* id=2 guarded */ /* ~4 */

/* region = 0 */

/* __dcbt call generated to prefetch y[i] access */
__dcbt(((char *)&y + (4)*(@CIV1)))
@CIV0 = @CIV0 + 1;
} while ((unsigned) @CIV0 < 1024u); /* ~4 */

lab_3:
@CIV1 = @CIV1 + 1;
} while ((unsigned) @CIV1 < 64u); /* ~2 */

lab_1:

return;
}
```

関連情報

- 305 ページの『-qprefetch』

並列処理のための組み込み関数

以下の組み込み関数を使用して、並列環境に関する情報を取得します。

- 653 ページの『第 8 章 並列処理のための OpenMP ランタイム関数』
- 646 ページの『トランザクション・メモリー組み込み関数』

IBM SMP 組み込み関数

__parthds (C のみ)

目的

parthds ランタイム・オプションの値を戻します。

プロトタイプ

```
int __parthds(void);
```

戻り値

parthds オプションが明示的に設定されていない場合、ランタイム・ライブラリーによって設定されたデフォルト値を戻します。 **-qsmp** コンパイラー・オプションが

プログラムのコンパイル中に指定されなかった場合は、選択されたランタイム・オプションに関係なく、1 を戻します。

__usrthds (C のみ)

目的

usrthds ランタイム・オプションの値を戻します。

プロトタイプ

```
int __usrthds(void);
```

戻り値

usrthds オプションが明示的に設定されていないか、**-qsmp** コンパイラー・オプションがプログラムのコンパイル中に指定されない場合は、選択されたランタイム・オプションに関係なく、0 を戻します。

トランザクション・メモリー組み込み関数

トランザクション・メモリーは、並列プログラミングのモデルです。このモジュールは、アトミックに処理される命令やステートメントのブロックを指定できる関数を提供します。このようなアトミック・ブロックは、トランザクションと呼ばれます。あるスレッドがトランザクションを実行すると、トランザクション内のすべてのメモリー操作は、他のスレッドから見て同時に行われます。

並列プログラムによっては、トランザクションの実装を、ロックなどの他の実装メソッドよりも効率的なものにできます。これらの組み込み関数を使用して、トランザクションの開始と終了を示したり、失敗の理由を診断したりできます。

トランザクション・メモリー組み込み関数では、*TM_buff* パラメーターの指定によって、トランザクション状態とデバッグ情報を格納するために、ユーザー提供のメモリー・ロケーションを使用できます。

トランザクション状態には、__TM_begin または __TM_simple_begin への正常な呼び出しの後に入ります。また、__TM_end、__TM_abort、__TM_named_abort、またはトランザクションの失敗によって終わります。

トランザクションの失敗は、以下のいずれかの条件が満たされると起きます。

- トランザクション状態でアクセスされるメモリーは、そのトランザクションが完了するまで、別のスレッドまたは延期状態で実行する同じスレッドによってアクセスされます。
- 1 トランザクション内で、メモリー・アクセスのためのアーキテクチャー定義のフットプリントを超過しています。
- ネストされたトランザクションのアーキテクチャー定義のネスト制限を超過しています。

トランザクションはネストすることができます。トランザクション状態で、__TM_begin または __TM_simple_begin を使用できます。 __TM_begin で開始された最外部のトランザクション内では、ネストしたトランザクションは、

__TM_simple_begin を使用するか、またはネストの最外部のトランザクションと同じバッファーを使用する __TM_begin によって開始する必要があります。

ネストしたトランザクションは、外側のトランザクションに含まれます。したがって、ネストしたトランザクションの失敗は、外側のすべてのトランザクションの失敗として扱われ、外側のすべてのトランザクションが完了しないと、ネストしたトランザクションは完了しません。

注: トランザクション・メモリー組み込み関数は、**-qarch** が POWER8 プロセッサーをターゲットとするように設定されている場合にのみ有効です。

トランザクション開始/終了関数

__TM_begin:
目的

トランザクションの開始を示します。

プロトタイプ

long __TM_begin (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションの失敗または打ち切り時の、トランザクション状態情報を格納します。

使用法

トランザクションの失敗時 (ユーザーによる打ち切りを含む) には、__TM_begin が失敗したかのように、実行は、失敗したトランザクションを開始した __TM_begin の直後の点から再開されます。

トランザクション状況を照会するには、トランザクション照会関数を使用できます。

戻り値

成功した場合、この関数は 0 を返します。そうでない場合、非ゼロの値を返します。

__TM_end:
目的

トランザクションの終了を示します。

プロトタイプ

long __TM_end ();

戻り値

この関数は、命令の開始前にスレッドがトランザクション状態になっている場合は 0、そうでない場合は 1 を返します。

__TM_simple_begin:
目的

トランザクションの開始を示します。

プロトタイプ

long __TM_simple_begin ();

使用法

トランザクションの失敗時 (ユーザーによる打ち切りを含む) には、__TM_simple_begin が失敗したかのように、実行は、失敗したトランザクションを開始した __TM_simple_begin 関数の直後の点から再開されます。

戻り値

成功した場合、この関数は 0 を返します。そうでない場合、非ゼロの値を返します。 __TM_simple_begin を使用して開始されたトランザクションのトランザクション状況は、トランザクション照会関数を使用して照会することはできません。

トランザクション打ち切り関数

__TM_abort:
目的

トランザクションが打ち切られ、エラー・コード 0 が出されます。

プロトタイプ

```
void __TM_abort ();
```

__TM_named_abort:
目的

トランザクションが打ち切られ、エラー・コード *code* が出されます。

プロトタイプ

void __TM_named_abort (unsigned char const *code*);

パラメーター

code
0 から 255 までの範囲内のリテラルでなければなりません。

トランザクション照会関数

__TM_failure_address:

目的

最新のトランザクションが打ち切られたコード・アドレスを取得します。

プロトタイプ

long __TM_failure_address (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、トランザクションが打ち切られたアドレスを返します。

__TM_failure_code:
目的

トランザクションのロー障害コードを取得します。

プロトタイプ

long long __TM_failure_code (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、TEXASR レジスターの内容を返します。戻り値の解釈方法については、ハードウェアの仕様書を参照できます。

__TM_is_conflict:
目的

トランザクションが競合が原因で打ち切られたのかどうかを照会します。

プロトタイプ

long __TM_is_conflict (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、競合が原因で打ち切られた場合は 1、そうでない場合は 0 を返します。

__TM_is_failure_persistent:

目的

トランザクションが持続的な理由が原因で打ち切られたのかどうかを照会します。

プロトタイプ

long __TM_is_failure_persistent (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、持続的な理由で打ち切られた場合は 1、そうでない場合は 0 を返します。

__TM_is_footprint_exceeded:
目的

キャッシュ・ラインの最大数の超過が原因で、トランザクションが打ち切られたのかどうかを照会します。

プロトタイプ

long __TM_is_footprint_exceeded (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、キャッシュ・ラインの最大数を超過したために打ち切られた場合は 1、そうでない場合は 0 を返します。

__TM_is_illegal:
目的

トランザクション・モードで許可されていない命令や他の何らかの無許可アクセスなどの、無許可の動作を試行したために、トランザクションが打ち切られたかどうかを照会します。

プロトタイプ

long __TM_is_illegal (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、無許可の動作を試行したために打ち切られた場合は 1、そうでない場合は 0 を返します。

__TM_is_named_user_abort:
目的

ユーザーの打ち切り命令が原因でトランザクションが失敗したのかどうかを照会します。

プロトタイプ

long __TM_is_named_user_abort (void* const *TM_buffer*, unsigned char* *code*);

パラメーター

code
code は、トランザクション打ち切り命令に渡されたコードに設定されます。コードが命令に渡されない場合、*code* は 0 に設定されます。

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、ユーザーの打ち切り命令が原因で失敗した場合は 1、そうでない場合は 0 を返します。

__TM_is_nested_too_deep:
目的

最大のネストの深さを超えようとしたことが原因で、トランザクションが打ち切られたのかどうかを照会します。

プロトタイプ

long __TM_is_nested_too_deep (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、最大のネストの深さを超過しようとしたために打ち切られた場合は 1、そうでない場合は 0 を返します。

__TM_is_user_abort:
目的

ユーザーの打ち切り命令が原因でトランザクションが失敗したのかどうかを照会します。

プロトタイプ

long __TM_is_user_abort (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

戻り値

この関数は、*TM_buffer* 引数に状況が格納されているトランザクションが、ユーザーの打ち切り命令が原因で失敗した場合は 1、そうでない場合は 0 を返します。

__TM_nesting_depth:
目的

現在のネストの深さを取得します。トランザクション・モードでない場合は、最新のトランザクションが打ち切られた深さを取得します。

プロトタイプ

long __TM_nesting_depth (void* const *TM_buffer*);

パラメーター

TM_buffer
トランザクションに関する状況情報を保管します。

使用法

TM_buffer パラメーターを指定した `__TM_nesting_depth` の結果は、*TM_buffer* のトランザクション状態と同じく、TEXASR レジスターの内容に基づいています。

戻り値

この関数は、状況が *TM_buffer* 引数に格納されているトランザクションの現在のネストの深さを返します。トランザクション・モードでない場合、トランザクションが打ち切られた深さを返します。これらに当てはまらない場合は、0 を返します。

第 8 章 並列処理のための OpenMP ランタイム関数

omp_ 関数の関数定義は、omp.h ヘッダー・ファイルにあります。

OpenMP ランタイム・ライブラリー関数の完全な説明については、www.openmp.org の OpenMP Application Program Interface 仕様を参照してください。

関連情報
- 30 ページの『並列処理のための環境変数』

omp_get_max_active_levels

目的

max-active-levels-var 内部制御変数の値を返します。この変数は、ネストしたアクティブな並列領域の最大数を決定します。*max-active-levels-var* は、*OMP_MAX_ACTIVE_LEVELS* 環境変数または **omp_set_max_active_levels** ランタイム・ルーチンを使用して設定できます。

プロトタイプ

```
int  omp_get_max_active_levels(void);
```

omp_set_max_active_levels

目的

max-active-levels-var 内部制御変数の値を、引数内の値に設定します。要求した並列レベルの数が、サポートされている並列処理レベルの数を超える場合、*max-active-levels-var* の値はランタイムによってサポートされている並列レベル数に設定されます。要求した並列レベルの数が正整数でない場合、このルーチン呼び出しは無視されます。

ネストされた並列処理がオフの場合、このルーチンは効果がなく、*max-active-levels-var* の値は 1 のままです。*max-active-levels-var* は、*OMP_MAX_ACTIVE_LEVELS* 環境変数で設定することもできます。*max-active-levels-var* の値を取り出すには、**omp_get_max_active_levels** 関数を使用します。

omp_set_max_active_levels は、プログラムの順次領域でのみ使用します。このルーチンは、プログラムの並列領域では効果はありません。

プロトタイプ

```
void  omp_set_max_active_levels(int  max_levels);
```

パラメーター

max_levels
ネストされたアクティブな並列領域の最大数を指定する整数。

omp_get_schedule

目的

並列領域を処理しているチームの *run-sched-var* 内部制御変数を戻します。引数 *kind* は、使用されるスケジュールのタイプを戻します。 *modifier* は、適用可能なスケジュール・タイプ用に設定されるチャンク・サイズを表します。 *run-sched-var* は、*OMP_SCHEDULE* 環境変数または **omp_set_schedule** 関数を使用して設定できます。

プロトタイプ

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

パラメーター

kind
kind に対して戻される値は、スケジュール・タイプ affinity、 auto、 dynamic、guided、 runtime、または static の 1 つです。

注: 類縁性スケジュール・タイプは非推奨であり、将来のリリースでは除去される可能性があります。類似の機能には動的スケジュール・タイプを使用できます。

modifier
スケジュール・タイプ dynamic、guided、または static の場合、*modifier* は設定されるチャンク・サイズです。スケジュール・タイプが auto の場合、*modifier* は適用されません。

関連資料:

『omp_set_schedule』

関連情報:

39 ページの『OMP_SCHEDULE』

omp_set_schedule

目的

run-sched-var 内部制御変数の値を設定します。 *OMP_SCHEDULE* 環境変数とは別個にスケジュール・タイプを設定する場合は、**omp_set_schedule** を使用します。

プロトタイプ

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

パラメーター

kind
affinity、auto、dynamic、guided、runtime、または static のいずれかのスケジュール・タイプでなければなりません。

modifier
スケジュール・タイプ dynamic、guided、または static の場合、*modifier* は設定したいチャンク・サイズです。一般に、これは正整数でなければなりません。この値が 1 より小さい場合は、デフォルトが使用されます。スケジュール・タイプが auto の場合、*modifier* は適用されません。

関連資料:

654 ページの『omp_get_schedule』

関連情報:

39 ページの『OMP_SCHEDULE』

omp_get_thread_limit

目的

プログラムで使用可能な OpenMP スレッドの最大数を返します。値は *thread-limit-var* 内部制御変数に格納されます。*thread-limit-var* は、*OMP_THREAD_LIMIT* 環境変数を使用して設定できます。

プロトタイプ

```
int omp_get_thread_limit(void);
```

omp_get_level

目的

生成側のタスクが実行されているアクティブまたは非アクティブなネストされた並列領域の数が戻されます。これには、暗黙の並列領域は含まれません。プログラムの順次部分から呼び出された場合、0 を返します。それ以外の場合、負でない整数を返します。

プロトタイプ

```
int omp_get_level(void);
```

omp_get_ancestor_thread_num

目的

指定されたネスト・レベルの現行スレッドの上位スレッドのレベル・スレッド番号を返します。ネスト・レベルが 0 から **omp_get_level** によって返された現行スレッドのネスト・レベルまでの範囲内にない場合、-1 を返します。

プロトタイプ

```
int  omp_get_ancestor_thread_num(int  level);
```

パラメーター

level
現行スレッドの特定のネスト・レベルを指定します。

---

omp_get_team_size

目的

上位スレッドまたは現行スレッドが属するスレッド・チーム・サイズを返します。**omp_get_team_size** は、ネスト・レベルが 0 から **omp_get_level** によって返された現在のスレッドのネスト・レベルまでの範囲内にない場合、-1 を返します。

プロトタイプ

```
int  omp_get_team_size(int  level);
```

パラメーター

level
現行スレッドの特定のネスト・レベルを指定します。

---

omp_get_active_level

目的

呼び出しが含まれるタスクを囲んでいる、ネストされたアクティブな並列領域の数を返します。このルーチンは負でない整数を常に返し、プログラムの順次部分から呼び出された場合、0 を返します。

プロトタイプ

```
int  omp_get_active_level(void);
```

---

omp_get_num_threads

目的

この関数が呼び出されている並列領域を実行しているチームに現在あるスレッドの数を戻します。

プロトタイプ

```
int  omp_get_num_threads(void);
```

omp_set_num_threads

目的

OMP_NUM_THREADS 環境変数の設定をオーバーライドし、*OMP_NUM_THREADS* の *num_list* の最初の値を設定することによって、後続の並列領域で使用するスレッドの数を指定します。

プロトタイプ

```
void omp_set_num_threads(int num_threads);
```

パラメーター

num_threads
正の整数でなければなりません。

使用法

num_threads 節がある場合は、それが適用される並列領域に対して、この関数または *OMP_NUM_THREADS* 環境変数によって要求されたスレッドの数が置き換えられます。後続の並列領域はこの影響を受けません。

omp_get_max_threads

目的

OMP_NUM_THREADS 環境変数の *num_list* の最初の値を返します。この値は、**num_threads** 節がない並列領域が検出された場合に新規チームを形成するために使用可能なスレッドの最大数です。

プロトタイプ

```
int omp_get_max_threads(void);
```

omp_get_thread_num

目的

そのチームの範囲内で、この関数を実行しているスレッドのスレッド番号を戻します。

プロトタイプ

```
int omp_get_thread_num(void);
```

戻り値

スレッド番号は、0 と **omp_get_num_threads()**-1 の間です。チームのマスター・スレッドは、スレッド 0 です。

omp_get_num_procs

目的

プログラムに割り当てることができるプロセッサーの最大数を戻します。

プロトタイプ

```
int  omp_get_num_procs(void);
```

omp_in_final

目的

関数が final タスク領域で呼び出される場合にゼロ以外の整数値を返します。それ以外の場合は 0 を返します。

プロトタイプ

```
int  omp_in_final(void);
```

omp_in_parallel

目的

並列で実行している並列領域の動的範囲内で呼び出された場合、非ゼロを戻します。それ以外は 0 を戻します。

プロトタイプ

```
int  omp_in_parallel(void);
```

omp_set_dynamic

目的

並列領域の実行に使用可能なスレッドの数の動的調整を使用可能または使用不可にします。

プロトタイプ

```
void  omp_set_dynamic  (int  dynamic_threads);
```

パラメーター

dynamic_threads
後続の並列領域で使用可能なスレッドの数をランタイム・ライブラリーによって調整できるかどうかを示します。*dynamic_threads* がゼロ以外の場合、ランタイム・ライブラリーはスレッド数を調整できます。*dynamic_threads* がゼロの場合、ランタイム・ライブラリーはスレッド数を動的に調整できません。

omp_get_dynamic

目的

動的スレッドの調整が使用可能な場合に非ゼロを返し、それ以外は 0 を返します。

プロトタイプ

```
int  omp_get_dynamic(void);
```

omp_set_nested

目的

ネストされた並列性を使用可能または使用不可にします。

プロトタイプ

```
void  omp_set_nested  (int  nested);
```

使用法

omp_set_nested の引数が true に評価された場合、現在のタスクに対してネストされた並列処理が使用可能になります。それ以外の場合、ネストされた並列処理は現在のタスクでは使用不可になります。**omp_set_nested** の設定は、*OMP_NESTED* 環境変数の設定をオーバーライドします。

注: 並列領域内およびそのネストした並列領域内のスレッド数が、使用可能なプロセッサー数を超えると、プログラムで性能低下が発生する可能性があります。

omp_get_nested

目的

ネストされた並列性が使用可能な場合に非ゼロを戻し、使用不可の場合は 0 を戻します。

プロトタイプ

```
int  omp_get_nested(void);
```

omp_init_lock、omp_init_nest_lock

目的

以降の呼び出しで使用するために、パラメーター *lock* と関連したロックを初期化します。

プロトタイプ

```
void  omp_init_lock(omp_lock_t  *lock);

void  omp_init_nest_lock(omp_nest_lock_t  *lock);
```

パラメーター

lock
型 omp_lock_t の変数である必要があります。

omp_destroy_lock、omp_destroy_nest_lock

目的

指定されたロック変数 *lock* が初期化されていないことを保証します。

プロトタイプ

```
void omp_destroy_lock(omp_lock_t *lock);

void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

パラメーター

lock
omp_init_lock または **omp_init_nest_lock** で初期化される型 **omp_lock_t** の変数である必要があります。

omp_set_lock、omp_set_nest_lock

目的

指定されたロックが使用可能になりロックを設定するまで、その関数を実行しているスレッドをブロックします。

プロトタイプ

```
void omp_set_lock (omp_lock_t * lock);

void omp_set_nest_lock (omp_nest_lock_t * lock);
```

パラメーター

lock
omp_init_lock または omp_init_nest_lock で初期化される型 omp_lock_t の変数である必要があります。

使用法

アンロックされている場合は、単純ロックが使用可能です。ネスト可能なロックは、アンロックされている場合、またはこの関数を実行しているスレッドによって既に所有されている場合は使用可能です。

omp_unset_lock、omp_unset_nest_lock

目的

ロックの所有権を解放します。

プロトタイプ

```
void omp_unset_lock (omp_lock_t * lock);

void omp_unset_nest_lock (omp_nest_lock_t * lock);
```

パラメーター

lock
`omp_init_lock` または **omp_init_nest_lock** で初期化される型 **omp_lock_t** の変数である必要があります。

---

omp_test_lock、omp_test_nest_lock

目的

ロックを設定しようとしますが、スレッドの実行はブロックしません。

プロトタイプ

```
int omp_test_lock (omp_lock_t * lock);

int omp_test_nest_lock (omp_nest_lock_t * lock);
```

パラメーター

lock
`omp_init_lock` または **omp_init_nest_lock** で初期化される型 **omp_lock_t** の変数である必要があります。

---

omp_get_wtime

目的

固定された開始時刻からの経過時間を戻します。

プロトタイプ

```
double omp_get_wtime(void);
```

使用法

固定された開始時刻の値は現行プログラムの開始時に決定され、プログラム実行中、定数となります。

---

omp_get_wtick

目的

クロックの刻みの間の秒数を戻します。

プロトタイプ

```
double omp_get_wtick(void);
```

使用法

固定された開始時刻の値は現行プログラムの開始時に決定され、プログラム実行中、定数となります。

特記事項

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

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

〒103-8510
東京都中央区日本橋箱崎町19番21号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

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

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

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

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

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

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

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

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

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

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

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

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

著作権使用許諾:

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

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

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

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. 1998, 2014.

商標

IBM、IBM ロゴ、および ibm.com® は、世界の多くの国で登録された International Business Machines Corporation の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

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

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

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

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

索引

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

[ア行]

[カ行]

[サ行]

[タ行]

[ハ行]

[マ行]

[ヤ行]

［ラ行］

A

C

D

F

G

L

M

N

O

P

R

S

V

X

[特殊文字]

プログラム番号: 5765-J08; 5725-C73

Printed in Japan

SA88-5388-00

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21