IBM XL C for AIX, V10.1

コンパイラー・リファレンス

SC88-4917-00
(英文原典：SC23-8882-00)

IBM XL C for AIX, V10.1

コンパイラー・リファレンス

SC88-4917-00
(英文原典：SC23-8882-00)

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、 425 ページの『特記事項』に記載されている情報をお読みください。

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

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

原典： SC23-8882-00
IBM XL C for AIX, V10.1
Compiler Reference
Version 10.1

発行： 日本アイ・ビー・エム株式会社

担当： ナショナル・ランゲージ・サポート

第1刷 2008.5

目次

本書について

本書は、IBM® XL C for AIX®, V10.1 コンパイラーに関する参照情報です。本書には 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、事前定義されたマクロ、組み込み関数、環境変数、およびエラー・メッセージと戻りコードの参照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 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 の開発経験者です。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 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 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

本書全体を通して、**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に使用されます。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 コンパイラーを使用した 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 のインストールに関する情報については、「*XL C インストール・ガイド*」を参照してください。
- C プログラミング言語: C プログラミング言語の構文、セマンティクスおよび IBM インプリメンテーションについては、「*XL 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 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 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様々な方法、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様々な

方法、コンパイラー・ユーティリティー **gxlc** を使用した GNU 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。

25 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** 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。

43 ページの『第 3 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。

287 ページの『第 4 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 および SMP ディレクティブのそれぞれの説明が記載されています。

349 ページの『第 5 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。

361 ページの『第 6 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 PowerPC® アーキテクチャーの XL C 組み込み関数がそれぞれ説明されています。

表記規則

書体の規則

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

表 1. 書体の規則

書体	対象	例
太字	小文字コマンド、実行可能ファイル名、コンパイラー・オプション、およびディレクティブ。	コンパイラーには、基本的な呼び出しコマンドである **xlc** と、さまざまな C 言語レベルとコンパイル環境をサポートするためのその他のコンパイラー呼び出しコマンドが併せて用意されています。
イタリック	実際の名前や値をユーザーが指定するパラメーターまたは変数。新規用語を紹介する際にも、イタリックが使用されます。	要求された *size* より大きな値を返す場合は、必ず *size* パラメーターを更新してください。
下線	コンパイラー・オプションまたはディレクティブのパラメーターのデフォルト設定。	nomaf | maf

表 1. 書体の規則 (続き)

書体	対象	例
モノスペース	プログラミング・キーワードおよびライブラリー関数、コンパイラーの組み込み関数、プログラム・コードの例、コマンド・ストリング、またはユーザー定義名。	myprogram.cをコンパイルして、最適化するには、`xlc myprogram.c -O3` と入力します。

構文図

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

- 構文図は、線の経路に沿って左から右、上から下に読みます。

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

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

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

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

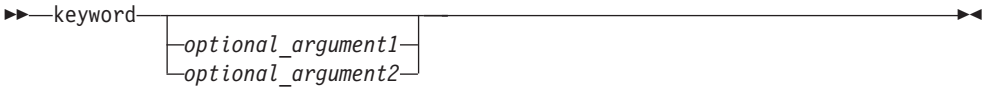
  完全なコマンド、ディレクティブ、またはステートメントではない構文単位の断片図は、|── 記号で始まり、──| 記号で終わります。
- 必須項目は水平線 (メインパス) 上に示されます。

- オプション項目はメインパスの下側に示されます。

- 複数の項目から選択できる場合、それらの項目は縦に並べて表示されます。

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

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



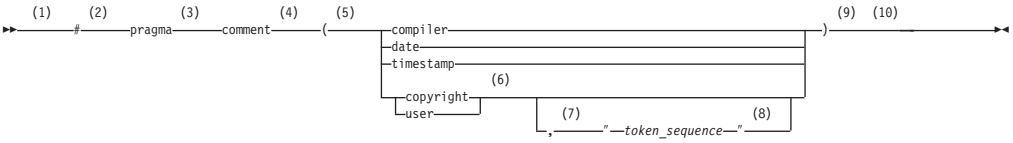
- メインライン上の左に戻る矢印 (繰り返し矢印) は、縦に並んだ項目から複数の項目を選択できること、または同じ項目を繰り返すことができることを示します。ブランク以外の分離文字も、次のように示されます。

- デフォルトの項目は、メインパスの上側に表示されます。

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

構文図の例

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



注:

1 これが構文図の始まりです。

2 記号 # で始まります。

3 キーワード pragma は # 記号の後に続けます。

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

5 ここに左括弧が必要です。

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

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

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

9 ここに右括弧が必要です。

10 これが構文図の終わりです。
次の **#pragma comment** ディレクティブの例は、上記の図に従った構文上正しいものです。

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

本書での例

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

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

関連情報

以下のセクションには、XL Cの関連情報が記載されています。

IBM XL C の資料

XL C には、次の形式の製品情報が用意されています。

- README ファイル

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

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

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

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

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

以下が、XL C のすべての製品情報を構成するファイルです。

表 2. XL C PDF ファイル

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

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

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

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

標準および仕様

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

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*、*C89* とも呼ばれています。
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*、*C99* とも呼ばれています。

- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769.* このドラフト技術レポートは、C 標準委員会によって受諾されており、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf から入手できます。
- *AltiVec Technology Programming Interface Manual*、Motorola Inc。ベクトル処理テクノロジーをサポートするためのベクトル・データ型に関するこの仕様は、http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf から入手できます。
- *Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732.* このドラフト技術レポートは C 標準委員会に提出済みであり、http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf から入手できます。
- *Decimal Types for C++: Draft 4* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1977.html

その他の IBM 資料

- *AIX コマンド・リファレンス 第 1 巻 - 第 6 巻、SC88-6875*
- *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2、SC23-4913*
- *AIX National Language Support Guide and Reference、SC88-6933*
- *AIX プログラミングの一般概念: プログラムの作成とデバッグ、SC88-6931*
- *AIX Assembler Language Reference、SC23-4923*

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

その他の資料

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

技術サポート

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

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

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

第 1 章 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、XL C コンパイラーを呼び出す場合、以下に示す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
- 実行可能ファイルへのリンク

これらの異なる変換フェーズは、実際、コンパイラー・コンポーネント と呼ばれる別の実行可能プログラムによって実行されます。ただし、プリプロセッシングやアセンブルなど特定のフェーズを実行するには、コンパイラー・オプションを使用できます。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 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 コンパイラー呼び出しコマンドが異なるレベルの 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ます。

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 3 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 4 で説明します。

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 /etc/vac.cfg ファイルを参照してください。

表 3. コンパイラー呼び出し

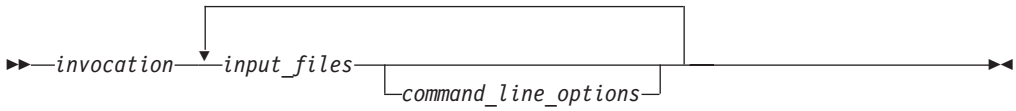
基本呼び出し	説明	同等の特殊呼び出し
xlc	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは、すべての ISO C99 標準機能および大半の IBM 言語拡張機能をサポートします。 この呼び出しは、すべてのアプリケーションに使用することをお勧めします。	xlc_r、 xlc128_r4、 xlc128_r7、 xlc128、 xlc128_r、 xlc128_r4、 xlc128_r7
c99	C ソース・ファイル用のコンパイラーを呼び出します。このコマンドは、すべての ISO C99 言語機能をサポートしますが、IBM 言語拡張機能はサポートしません。厳密な C99 規格への合致のためには、この呼び出しを使用してください。	c99_r、 c99_r4、 c99_r7、 c99_128、 c99_128_r、 c99_128_r4、 c99_128_r7
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。厳密な C89 規格への合致のためには、この呼び出しを使用してください。	c89_r、 c89_r4、 c89_r7、 c89_128、 c89_128_r、 c89_128_r4、 c89_128_r7
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、 cc_r4、 cc_r7、 cc128、 cc128_r、 cc128_r4、 cc128_r7
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。詳細については、 13 ページの『**gxlc** での GNU 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。	
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 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。詳細については、 13 ページの『**gxlc** での GNU 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 4. 特殊呼び出しのサフィックス

128 サフィックスの呼び出し	すべての **128** サフィックスの呼び出しコマンドは機能面では、対応する基本のコンパイラー呼び出しに似ています。これらは、**-qldbl128** オプションを指定し、それによりプログラム内の `long double` 型の長さは 64 から 128 ビットに増加します。また、これらは 128-bit バージョンの C ランタイム・ライブラリーともリンクします。
_r サフィックスの呼び出し	すべての **_r** サフィックスの呼び出しではスレッド・セーフ・コンパイルが許可され、それらを使用してマルチスレッドを使用するプログラムをリンクすることができます。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。 **_r7** 呼び出しは、Posix ドラフト 7 を基にしたプログラムを Posix ドラフト 10 にマイグレーションするのを助けるために提供されています。**_r4** 呼び出しは、DCE スレッド化アプリケーションに使用してください。

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されます。ここでの *invocation* は、表 3にリストされる有効な XL C 呼び出しコマンドに置換することができます。



コンパイラー呼び出しコマンドのパラメーターは、入力ファイル名、コンパイラー・オプション名、およびリンカー・オプション名にすることができます。

プログラムは、複数の入力ファイルで構成することができます。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 回の呼び出しを使用して複数のソース・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記述、幾つかのプリプロセッサー機能の実行など、幅広い種類の機能を実行します。

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。このコマンドは、リンカー・オプションをリンカーに渡します。 その結果、これらの呼び出しコマンドは、すべてのリンカー・オプションも受け入れます。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** オプションはコンパイルの完了後にコンパイラーを停止し、**-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・ファイルを現れた順に処理します。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。しかし、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せずにソース・ファイルをプリプロセスすることもできます。詳細は、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 コンパイラーに入力することができます。

C ソース・ファイル
これらは 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。

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されたソース・ファイル *file_name*.i を、.c ファイルと同じ方法で再度プリプロセスされるコンパイラーに送信します。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、および非ストリップの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、**-qsourcetype=assembler-with-cpp** 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)。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。

共用ライブラリー・ファイル

共用ライブラリー・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 Extended Common Object File Format (XCOFF) ファイルは、コンパイラーへの入力として使用できます。詳しくは、「*AIX コマンド解説書*」の『**strip** コマンド』、および「*AIX ファイル・リファレンス*」の a.out ファイル形式の説明を参照してください。

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 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。

a.out ファイルのフォーマットについては、「*AIX ファイル・リファレンス*」の説明を参照してください。

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。

共用ライブラリー・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用ライブラリー・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 shr.o 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*.u 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記述、幾つかのプリプロセッサー機能の実行など、幅広い種類の機能を実行します。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 はこれらの矛盾や非互換のほとんどを、以下のように一貫した方法で解決します。

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、コマンド行に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。同じコマンド行のコンパイラー呼び出しに、矛盾または非互換のコンパイラー・オプションが指定された場合は、その呼び出しの後の方に出現する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、コマンド行、またはソース・プログラム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します。

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。

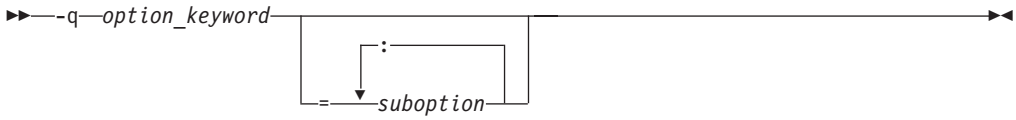
コマンド行でのコンパイラー・オプションの指定

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認識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。例えば、以下のように指定します。

```
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。例えば、以下のように指定します。

```
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、コマンド行に **-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。例えば、以下のように指定します。

```
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 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 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 (/etc/vac.cfg) 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 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、 35 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。同等のプラグマ・ディレクティブを含むオプションの場合、プラグマの構文の指定方法がいくつかあります。

- **#pragma options *option_name*** 構文を使用する - 多くのコマンド行オプションでは **#pragma options** 構文を使用できます。この構文にはオプションと同じ名前を使用でき、サブオプションにはそのオプションと同じ構文が使用できます。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
  ```

  各個別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どうかを示します。これらのオプションの完全なリストは、 309 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た対応するプラグマ・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少し異なるサブオプションが含まれることがあります。このセクション 59 ページの『個々のオプションの説明』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 - 上記に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` 演算子構文も使用できます。

プラグマ構文について詳しくは、 287 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、 287 ページの『第 4 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の矛盾が上記規則を使用して解決されるわけではありません。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 32 ビットまたは 64 ビット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
-qfloat=hssngl	-qfloat=spnans	-qfloat=hssngl

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	-qnostdinc

アーキテクチャー固有の 32 ビットまたは 64 ビットのコンパイルでのコンパイラー・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. OBJECT_MODE 環境変数の設定
3. 構成ファイルの設定
4. コマンド行コンパイラー・オプション (**-q32**、**-q64**、**-qarch**、**-qtune**)
5. 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。このいずれのコンパイラー・オプションも設定されていない場合、コンパイラー・モードは OBJECT_MODE 環

境変数の値によって設定されます。OBJECT_MODE 環境変数も設定されていない場合は、コンパイラーは 32 ビットのコンパイル・モードを想定します。

- 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、 70 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、 261 ページの『-qtune』を参照してください。

以下では、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出し、**-qtune** を **-qarch** の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、OBJECT_MODE 環境変数または **-q32/-q64** コンパイラー設定によって決定されます。

関連情報

- 70 ページの『-qarch』
- 261 ページの『-qtune』
- 61 ページの『-q32、-q64』

gxlc での GNU C コンパイラー・オプションの再使用

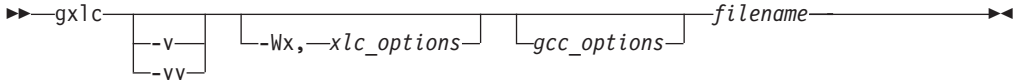
gxlc ユーティリティーは GNU C コンパイラー・オプションを受け入れて、同等の XL C オプションに変換します。ユーティリティーは XL C オプションを使用して **xlc** 呼び出しコマンドを作成し、それを XL C を呼び出すために使用します。**gxlc** ユーティリティーは、以前に GNU C を使用して開発されたアプリケーション向けに作成された Make ファイルの再利用を可能にするために提供されています。しかし、XL C の機能を完全に活用するためには、XL C 呼び出しコマンド **xlc** およびそれらの関連オプションを使用することをお勧めします。

gxlc のアクションは構成ファイル /etc/gxlc.cfg によって制御されます。XL C の相対オプションを持つ GNU C オプションがこのファイルに示されます。すべての GNU オプションが対応する XL C オプションを持っているわけではありません。**gxlc** ユーティリティーは、変換できない GNU C オプションに対して警告を戻します。

gxlc オプションのマッピングは変更可能です。**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、39 ページの『**gxlc** オプション・マッピングの構成』を参照してください。

gxlc 構文

以下の図は **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 の呼び出しに使用されるコマンドの検証を許可します。ユーティリティーは、作成した XL 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
: シミュレーションの実行を許可します。ユーティリティーは、作成した XL 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_ options*
: 特定の XL C オプションを直接 **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 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 オプションと共に使用してください。複数の *xlc_ options* は、コンマで区切られます。

gcc_options
: XL C オプションに変換される GNU C オプションです。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** によって認識される GNU 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_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

- 39 ページの『**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
103 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。
200 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。
209 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。
84 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。
94 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。
264 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。
230 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 は以下のタイプの組み込みファイルをサポートします。

- 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* ヘッダー として言及されます)
- 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・オプションによって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に検索します。
4. すべてのヘッダー・ファイル: コンパイラーは、標準ディレクトリーで XL C 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。これは通常 /usr/vac/include/ ですが、検索パスを **-qc_stdinc** コンパイラー・オプションで変更することもできます。 [2]
5. 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。 これは通常 /usr/vac/include/ ですが、検索パスは **-qc_stdinc** オプションで変更することができます。 [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

- 136 ページの『-I』
- 93 ページの『-qc_stdinc』
- 137 ページの『-qidirfirst』
- 141 ページの『-qinclude』
- 243 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、作成される共用オブジェクトのデフォルト名は shr.o 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、 56 ページの『リンク』を参照してください。

関連情報

191 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 ライブラリー
4. C 標準ライブラリー

関連情報

- 56 ページの『リンク』
- 『再配布可能ライブラリー』
- 「*AIX コマンド・リファレンス 第 5 巻 (s から u)*」の『**ld**』

再配布可能ライブラリー

XL 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ファイル・セット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、次の 1 つを実行することができます。

- 再配布可能なライブラリーを含むファイル・セット をアプリケーションと共に出荷することができます。ファイル・セットは、インストール CD の runtime/ 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ファイル・セットを、以下の XL 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/software/awdtools/caix/support/

これらのファイル・セット の配布に関連したライセンス交付の要件について詳しくは、CD 上の LicAgree.pdf を参照してください。

表 5. 再配布可能ライブラリー

ファイル・セット	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
xlsmp.rte	/usr/include/omp.h /usr/lpp/xlsmp/default_msg/smprt.cat	SMP ランタイム環境
xlsmp.aix53.rte	/usr/lpp/xlsmp/aix53/libxlomp_ser.a /usr/lpp/xlsmp/aix53/libxlsmp.a /usr/lpp/xlsmp/aix53/libxlsmpdebug.a	AIX向け SMP ランタイム環境
xlsmp.msg.en_US.rte	/usr/lib/nls/msg/en_US/smprt.cat	SMP ランタイム・メッセージ (英語、ISO8859-1)
xlsmp.msg.EN_US.rte	/usr/lib/nls/msg/EN_US/smprt.cat	SMP ランタイム・メッセージ (英語、UTF-8)
xlsmp.msg.ja_JP.rte	/usr/lib/nls/msg/ja_JP/smprt.cat	SMP ランタイム・メッセージ (日本語、IBM-eucJP)

表 5. 再配布可能ライブラリー (続き)

ファイル・セット	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
xlsmp.msg.Ja_JP.rte	/usr/lib/nls/msg/Ja_JP/smprt.cat	SMP ランタイム・メッセージ (日本語、IBM-943)
xlsmp.msg.JA_JP.rte	/usr/lib/nls/msg/JA_JP/smprt.cat	SMP ランタイム・メッセージ (日本語、UTF-8)
xlsmp.msg.zh_CN.rte	/usr/lib/nls/msg/zh_CN/smprt.cat	SMP ランタイム・メッセージ (中国語、IBM-eucCN)
xlsmp.msg.ZH_CN.rte	/usr/lib/nls/msg/ZH_CN/smprt.cat	SMP ランタイム・メッセージ (中国語、UTF-8)
xlsmp.msg.Zh_CN.rte	/usr/lib/nls/msg/Zh_CN/smprt.cat	SMP ランタイム・メッセージ (中国語、GBK)
vac.aix53.lib	/usr/vac/lib/aix53/libxl.a /usr/vac/lib/aix53/libxlopt.a	AIX 向け XL C ライブラリー
vacpp.memdbg.aix53.rte	/usr/vacpp/lib/aix53/libhC.a /usr/vacpp/lib/aix53/libhC_r.a /usr/vacpp/lib/profiled/aix53/libhC.a /usr/vacpp/lib/profiled/aix53/libhC_r.a	ユーザー動的データ域/メモリー・デバッグ・ツールキット (AIX 用)
vacpp.memdbg.rte	/usr/vacpp/lib/libhm.a /usr/vacpp/lib/libhm_r.a /usr/vappc/lib/libhmd.a /usr/vacpp/lib/libhmd_r.a /usr/vacpp/lib/libhmu.a /usr/vacpp/lib/libhmu_r.a /usr/vacpp/lib/libhu.a /usr/vacpp/lib/libhu_r.a /usr/vacpp/lib/profiled/libhm.a /usr/vacpp/lib/profiled/libhm_r.a /usr/vacpp/lib/profiled/libhmd.a /usr/vacpp/lib/profiled/libhmd_r.a /usr/vacpp/lib/profiled/libhmu.a /usr/vacpp/lib/profiled/libhmu_r.a /usr/vacpp/lib/profiled/libhu.a /usr/vacpp/lib/profiled/libhu_r.a	ユーザー動的データ域/メモリー・デバッグ・ツールキット

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』
- 21 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』
- 23 ページの『メッセージ・カタログ・エラー』
- 24 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 ソース・プログラムのコンパイル中にプログラミング・エラーを検出すると、標準エラー・デバイスに診断メッセージを発行し、**-qsource** オプショ

ンでコンパイルする場合は、リスト表示ファイルに診断メッセージを発行します。メッセージは、C 言語に固有のものです。

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
- 237 ページの『-qsource』
- 241 ページの『-qsrcmsg』
- 114 ページの『-qflag』
- 275 ページの『-w』
- 142 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、

ファイル
エラーのある 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

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。

重大度
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、20 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。

-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 - 15dd-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 は診断メッセージに対して複数段階の分類方式を使用します。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。以下のオプションの 1 つを使用すると、デフォルトのコンパイラー応答を調整できます。

- **-qhalt** を使用すると、デフォルトより低い重大度レベルのコンパイル段階を停止します。
- **-qmaxerr** を使用すると、特定の重大度レベルでのエラーが特定数に達したときにコンパイル段階が停止します。

表 6. 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。
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ラー・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。

表 6. コンパイラーのメッセージ重大度レベル (続き)

文字	重大度	コンパイラー応答
U	回復不能エラー	コンパイラーは停止します。内部コンパイラー・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

- 131 ページの『-qhalt』
- 185 ページの『-qmaxerr』
- 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。

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。例えば、ランタイム戻りコード 99 は、静的初期化が失敗したことを示します。

gxlc の戻りコード

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** 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。

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

これらのオプションのいずれかが有効であると、コンパイルで指定されたすべての入力ファイルに対するリスト・ファイル *filename*.lst が、現行ディレクトリーに保管されます。

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
コンパイラー名、バージョン、リリースのほか、ソース・ファイル名とコンパイルの日時もリストします。

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非デフォルトのオプションをリストします。有効なすべてのオプションをリストするには、**-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

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。

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、コンパイラーが生成したオブジェクト・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、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、正しいことを検証してください。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 /usr/lib/nls/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。

/usr/vacpp/exe/default_msg/ にあるデフォルトのメッセージ・カタログは次の場合にコンパイラーによって呼び出されます ロケールがデフォルトの **C** から変更されていない場合。

- LANG によって指定されたロケール用のメッセージ・カタログを検出できない場合。
- ロケールがデフォルトの **C** から変更されていない場合。

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の 1 つを発行します。

```
1501-229 Compilation ended due to lack of space.
1501-224 fatal error in ../exe/xlCcode: signal 9 received.
```

ページ・スペースの不足が原因で、ほかのコンパイラー・プログラムが失敗すると、以下のメッセージが表示されます。

```
Killed.
```

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。

現行のページ・スペース設定を確認するには、**lsps -a** コマンドを入力するか、AIX システム管理インターフェース・ツール (SMIT) コマンド **smit pgsp** を使用します。

ページング・スペースとその割り振り方法についての詳細は、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。

第 2 章 コンパイラーのデフォルトの構成

XL 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。インストールの作業中に、基本的な環境変数の一部を設定した可能性があります (詳細は、XL 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 /etc/vac.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 インストール・ガイド *XL 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などは指定できます。構成ファイルのカスタマイズに関する情報については、 35 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** 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 /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、 39 ページの『**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、Korn、および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。

すべてのユーザーがアクセスできるような変数を Bourne、Korn、および BASH シェルで設定するには、コマンドをファイル /etc/profile に追加します。特定のユーザー専用の変数を設定するには、コマンドをそのユーザーのホーム・ディレクトリーのファイル .profile に追加します。C シェルで、コマンドをファイル /etc/csh.cshrc に追加します。特定のユーザー専用の変数を設定するには、コマンドをそのユーザ

ーのホーム・ディレクトリーのファイル .cshrc 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 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。

- 『コンパイル時間およびリンク時間の環境変数』
- 27 ページの『ランタイム環境変数』

コンパイル時間およびリンク時間の環境変数

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 AIX 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、「*XL C インストール・ガイド*」のナショナル・ランゲージ・サポートを参照してください。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。

LIBPATH
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用ライブラリーが、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。

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 インストール・ガイド*」の *XL C エラー・メッセージの使用可能化*を参照してください。

OBJECT_MODE
コンパイルのビット・モードを 32 ビットにするか 64 ビットにするかをオプションで指定します。これは **-q32** および **-q64** のコンパイラー・オプションと同等です。OBJECT_MODE 環境変数を、32 ビットのコンパイル・モードの 32、または 64 ビットのコンパイル・モードの 64 に設定します。指定されない場合、デフォルトのコンパイル・モードは 32 ビットです。詳しくは、61 ページの『-q32、-q64』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。デフォルト・ロケーションにインストールされている場合、実行可能ファイルは /usr/vac/bin/ にあります。

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。高水準の最適化ではページング・ファイルと一時ファイルに大量のディスク・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/ では不適切な可能性があります。そのため、この環境変数を使用して別のディレクトリーを指定できます。

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。絶対パスを含むファイル名を付ける必要があります。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、35 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、実行時にシステム・ローダーまたはアプリケーションが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。

LIBPATH

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用ライブラリーが、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。

MALLOCALIGN=16

動的メモリー割り振りが 16 バイトの位置合わせアドレスを戻すことを指定します。 150 ページの『-qipa』も参照してください。

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、オプションで指定できます。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。アプリケーションを **-qpdf2** で再コンパイルまたは再リンクすると、コンパイラーは、このディレクトリーに保管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) を使用するのであれば、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、203 ページの『-qpdf1、-qpdf2』を参照してください。

XL_NOCLONEARCH

プログラムを **-qipa=clonearch** オプションでコンパイルすると、異なる実行時アーキテクチャーに最適化された関数の複数バージョンが生成されます。この場合、この環境変数を、コンパイルされたアプリケーションが汎用コード (特定アーキテクチャーのバージョン付けがされていないコード) のみを実行するように設定できます。この変数はデフォルトでは設定されていないため、アプリケーションのデバッグに役立つよう設定することができます。詳しくは、150 ページの『-qipa』を参照してください。

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 構造体を使用している場合は、32 ページの『並列処理のための OpenMP 環境変数』に説明があるように、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。

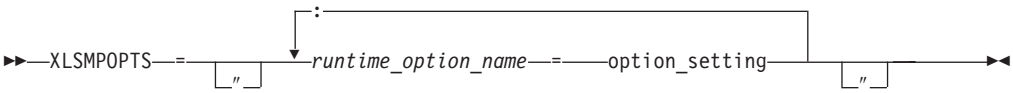
OMP- 環境変数および XLSMPOPTS 環境変数によって指定された実行時オプションが矛盾する場合は、OMP オプションが優先されます。

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

関連情報

- 327 ページの『並列処理のプラグマ・ディレクティブ』
- 415 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。

作業は、スケジューリングの種類およびチャンク・サイズに合わせて、異なる方法でスレッドに割り当てられます。チャンクの細分度を選択することは、オーバーヘッドとロード・バランシングのトレードオフです。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*) 繰り返しを含む *n* 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* 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) ループの繰り返しを含みます。

スレッドが解放されると、スレッドが最初に割り当てられた区画から次のチャンクを取得します。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。

アフィニティー・スケジューリングのタイプは、OpenMP API 標準には表示されません。

dynamic[=*n*]

ループの繰り返しが、それぞれ *n* 繰り返しを含むチャンクに分割されます。 *n* が指定されていないと、チャンクは **ceiling**(*number_of_iterations/number_of_threads*) 繰り返しを含みます。

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、すべての作業に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。

スレッドがスリープ状態にある場合、それに割り当てられた作業は、別のアクティブ・スレッドが使用可能になったら即座に引き継がれます。

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 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。

static[=*n*]

ループの繰り返しが、それぞれ *n* 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、必然的に、スケジューリング・タイプが*巡回スケジューリング* として参照されます。

n を指定しない場合、チャンクには **ceiling**(*number_of_iterations/number_of_threads*) 回の繰り返しが含まれます。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が必要とする最大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、32 ビット・モードの場合は最大 256 MB、64 ビット・モードの場合はシステム・リソースが指定する制限を超えない数値を使用できます。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。

glibc ライブラリーは、デフォルトで 2 MB のスタック・サイズを許可するようにコンパイルされます。num をこれより大きい値に設定すると、デフォルトのスタック・サイズが使用されます。これより大きなスタック・サイズが必要な場合は、FLOATING_STACKS パラメーターをオンにしてコンパイルした glibc ライブラリーにプログラムをリンクする必要があります。

stackcheck[=*num***]**

-qsmp=stackcheck が有効な場合、実行時にスレーブ・スレッドのスタック・オーバーフローのチェックが使用可能になります。*num* はスタック・サイズ (バイト) です。残りのスタック・サイズがこの値以下の場合、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、233 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して、最初のスレッドがバインドする 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。

stride=*num*

後続のスレッドがバインドされる CPU ID の判別に使用され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。

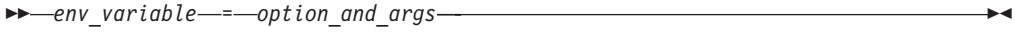
parthreshold=*num*
各ループが順次実行しなければならない時間をミリ秒で指定します。*num* を 0 に設定すると、コンパイラーによって並列化されたすべてのループが並列で実行されます。 デフォルト設定は 0.2 ミリ秒です。この場合、ループの並列実行に必要な時間が 0.2 ミリ秒より短いと、ループは順次実行に変更されることを意味します。

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。

並列処理のための OpenMP 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_SCHEDULE=algorithm 環境変数: **OMP_SCHEDULE** 環境変数は、**omp schedule** ディレクティブによって明示的にスケジューリング・アルゴリズムを割り当てられていないループに対して使用されるスケジューリング・アルゴリズムを指定します。 例を以下に示します。

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

algorithm の有効なオプションは、以下の通りです。

- auto
- dynamic[, *n*]
- guided[, *n*]
- runtime
- 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 1 以上の整数の値でなければなりません。

デフォルトのスケジューリング・アルゴリズムは **static** です。

関連資料

417 ページの『omp_set_schedule』

416 ページの『omp_get_schedule』

並列環境変数:
OMP_NUM_THREADS=*num*

OMP_NUM_THREADS 環境変数により、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。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。

num は、必要な並列スレッドの数を示します。この数は、システムで使用可能なプロセッサーの数と通常同じです。この数は、**omp_set_num_threads** 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。

特定の並列セクションの **OMP_NUM_THREADS** の設定は、いくつかの **#pragma omp** ディレクティブで使用可能な **num_threads** 節を使用することによりオーバーライドすることができます。

OMP_NESTED=TRUE | FALSE

OMP_NESTED 環境変数は、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。この設定は、**omp_set_nested** ランタイム・ライブラリー関数を呼び出してオーバーライドすることができます。

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。

現在のインプリメンテーションでは、ネストされた並列領域は、常に直列化されています。その結果、OMP_SET_NESTED は何の効果も持たず、**omp_get_nested** は常に 0 を戻します。 **-qsmp=nested_par** オプションがオンの場合 (厳密な OMP モードでない場合のみ) は、ネストされた並列領域は、追加スレッドを使用可能として使用する場合があります。ただし、ネストされた並列領域を実行するために、新規のチームが作成されることはありません。

OMP_NESTED のデフォルト値は **FALSE** です。

OMP_DYNAMIC=TRUE | FALSE 環境変数: **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。

TRUE に設定されている場合、並列領域の実行に使用可能なスレッドの数は、システム・リソースを最も有効に使用するために実行時に調整されます。詳しくは、28 ページの『XLSMPOPTS』の **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_WAIT_POLICY 環境変数:

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。

OMP_STACKSIZE 環境変数:

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 環境変数:

OMP_THREAD_LIMIT を使用して、*thread-limit-var* 内部制御変数を設定します。*thread-limit-var* では、プログラム全体で使用される OpenMP スレッドの数が示されます。関数 **omp_get_thread_limit** を使用すると、実行時にこの値を取得できます。OMP_THREAD_LIMIT の値は正整数です。選択する値がサポート可能なスレッド数より多い場合または正整数でない場合は、ランタイムにより、OMP_NUM_THREADS の *thread-limit-var* のデフォルト値か、または使用可能プロセッサー数のうち、大きい方が設定されます。**注**: *thread-limit-var* が設定される場合、*nthreads-var* 内部制御変数のデフォルト値は *thread-limit-var*、または使用可能プロセッサー数のうち、小さい方と同等になります。

```
►►—OMP_THREAD_LIMIT=n————————————————————————————————————►◄
```

OMP_MAX_ACTIVE_LEVELS 環境変数:

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。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_MAX_ACTIVE_LEVELS の値は正整数です。正整数が指定されない場合は、*max-active-levels-var* のデフォルト値がランタイムによって設定されます。

```
►►—OMP_MAX_ACTIVE_LEVELS=n———————————————————————————————►◄
```

カスタム・コンパイラー構成ファイルの使用

XL C はインストール時にデフォルトの構成ファイル /etc/vac.cfg.*nn* を生成します、ここで *nn* は、構成ファイルがどの OS バージョン向けかを示します。 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既にある場合は、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。

そうでない場合、特に多くのユーザーが幾つかのコンパイラー・オプションのセットから選択できるようにしたい場合は、特定のニーズに合わせてカスタム構成ファイルを使用することができます。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。このオプションは、コンパイルごとにコマンド行で使用する必要はなく、**xlc** コマンドでコンパイラーを呼び出すたびに自動的に有効になります。

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。オペレ

ーティング・システムをアップグレードする場合は、/etc/ 内の構成ファイルへのシンボリック・リンクを、正しいバージョンの構成ファイルを指すように変更する必要があります。

- 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。ここでも、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。
- 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:

- 112 ページの『-F』
- 26 ページの『コンパイル時間およびリンク時間の環境変数』

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。以下の例では、スタンザ A、C、および D がこれに当てはまります。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 vac.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 7. 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 vac.cfg

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

DEFLT:
     options=<C>
```

上記で示した構成ファイルのスタンザは、以下のオプション・セットを以下に示す順序で使用します。

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`
	この例では、**xlc** および **xlc_r** コマンドで呼び出されたすべてのコンパイルに **-qlist** を使用するよう指定しています。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。

gxlc オプション・マッピングの構成

gxlc ユーティリティーは構成ファイル /etc/gxlc.cf を使用して GNU C オプションを XL 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 オプションを XL 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 オプションのストリング、および XL 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 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** 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。

abc "gcc_option" "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 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_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_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_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** 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。

"gcc_option"
GNU 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__option"
XL 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_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** 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 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** 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 と XL 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=swg27011888

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・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 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』
- 13 ページの『**gxlc** での GNU 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

AIX プラットフォームで使用可能な XL 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』
- 44 ページの『入力制御』
- 45 ページの『言語エレメント制御』
- 47 ページの『浮動小数点および整数制御』
- 49 ページの『エラー・チェックおよびデバッグ』
- 51 ページの『リスト、メッセージ、およびコンパイラー情報』
- 53 ページの『最適化およびチューニング』
- 47 ページの『オブジェクト・コード制御』
- 56 ページの『リンク』
- 57 ページの『移植性およびマイグレーション』
- 58 ページの『コンパイラーのカスタマイズ』
- 59 ページの『推奨されないオプション』

出力制御

このカテゴリーのオプションは、出力の場所のみでなく、コンパイラーが作成するファイル出力のタイプも制御します。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 8. 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
83 ページの『-c』	なし。	完了済みのオブジェクトがリンカーへ送信されるのを防ぐ。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。
84 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。

表 8. 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
103 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。
129 ページの『-G』	なし。	ランタイム・リンクに使用できる共用オブジェクトを生成する。
183 ページの『-qmakedep、-M』	なし。	**make** コマンドの記述ファイルの組み込みに適したターゲットを含む出力ファイルを作成する。
189 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出力のターゲットを指定する。
191 ページの『-qmkshrobj』	なし。	生成されたオブジェクト・ファイルから共用オブジェクトを作成する。
192 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。
200 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。
225 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。
230 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。
257 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場所を指定します。

表 9. 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
136 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。

表 9. コンパイラー入力オプション (続き)

オプション名	同等のプラグマ名	説明
137 ページの『-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。
141 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。
238 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。
243 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を使用すると、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 10. 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
69 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。
74 ページの『-qasm』	なし	アセンブラー言語拡張のコードの解釈と以降の生成を制御する。
91 ページの『-qcpluscmt』	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。
94 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。
98 ページの『-qdfp』	なし。	10 進浮動小数点型およびリテラルに対するコンパイラー・サポートを使用可能にする。
99 ページの『-qdigraph』	#pragma options digraph	キーボードにない文字を表すために、連字キーの組み合わせおよびキーワードの認識を使用可能にする。

表 10. 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
101 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。
139 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。
163 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。
166 ページの『-qlanglvl』	#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。
180 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。
181 ページの『-qmacpstr』	#pragma options macpstr	(プレフィックスに `¥p` エスケープ・シーケンスが付いた) Pascal ストリング・リテラルを先頭バイトにストリングの長さが含まれるヌル終了ストリングに変換する。
188 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。
254 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。
261 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。
264 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。
271 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。

浮動小数点および整数制御

アプリケーションによる計算の実行方法の詳細を指定することによって、丸めの命令方法など、システムの浮動小数点パフォーマンスおよび精度をさらに活用することができます。ただし、IEEE 浮動小数点指定を厳密に順守するとご使用のアプリケーションのパフォーマンスに影響が出る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 標準の順守との間のトレードオフを制御することができます。

表 11. 浮動小数点および整数制御オプション

オプション名	同等のプラグマ名	説明
80 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。
87 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。
105 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。
116 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。
172 ページの『-qldbl128、-qlongdouble』	#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。
179 ページの『-qlonglit』	なし。	64 ビット・モードで暗黙の型の int を持つリテラルを long へプロモートする。
283 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

以下のオプションは、コンパイラーによって生成されるオブジェクト・コード、プリプロセス・コードまたはその他の出力の特性に影響を与えます。

表 12. 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
61 ページの『-q32、-q64』	なし。	32 ビットまたは 64 ビットのいずれかのコンパイラー・モードを選択する。
68 ページの『-qalloca、-ma』	#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。

表 12. 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
110 ページの『-qexpfile』	なし。	**-qmkshrobj** または **-G** オプションと一緒に使用して、1 つの指定ファイルにすべてのエクスポートされたシンボルを保管する。
149 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。
208 ページの『-qpic』	なし。	共用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。
209 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。
213 ページの『-qproto』	#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。
220 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。
221 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。
222 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。
223 ページの『-qroptr』	なし。	定数ポインターのストレージ・ロケーションを指定する。
225 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。
227 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、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 12. 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
242 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。
255 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。
257 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。
258 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。
265 ページの『-qunique』	なし。	静的コンストラクター/デストラクター・ファイルのコンパイル単位の固有名を生成する。
279 ページの『-qweakexp』	なし。	**-qmkshrobj** または **-G** オプションと一緒に使用して、共有オブジェクトを作成するときに生成されるエクスポート・リストから「弱い」とマークを付けられたグローバル・シンボルを組み込むかまたは除外する。
280 ページの『-qweaksymbol』	なし。	弱いシンボルの生成を使用可能にする。
281 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、 51 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 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 13. 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
60 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。
89 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。
97 ページの『-qdbxextra』	#pragma options dbxextra	**-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。
101 ページの『-qdpcl』	なし。	IBM Dynamic Probe Class Library (DPCL) 動的プローブ・クラス・ライブラリー (DPCL) を基にしたツールが実行可能ファイルの構造体を参照するために使用できるシンボルを生成する。
110 ページの『-qextchk』	#pragma options extchk	リンク時の型検査の情報を生成し、コンパイル時の整合性について検査する。
121 ページの『-qflttrap』	#pragma options flttrap	実行時に検出される浮動小数点例外条件のタイプを判別する。
124 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。
125 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。
128 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する。

表 13. 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
131 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。
132 ページの『-qheapdebug』	なし。	デバッグ・バージョンのメモリー管理関数を使用可能にする。
142 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。
148 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。
162 ページの『-qkeepparm』	なし。	**-O2** 以上の最適化で使用すると、関数仮パラメーターをスタックに保管するかどうかを指定する。
175 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。
185 ページの『-qmaxerr』	なし。	指定したレベル以上の重大度レベルのエラーの件数が指定した件数に達すると、コンパイルを停止する。
198 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。
251 ページの『-qsymtab』	なし。	シンボル・テーブルに表示する情報を決定する。
252 ページの『-qsyntaxonly』	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。
278 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 49 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 14. 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
77 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。
114 ページの『-qflag』	#pragma options flag	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。
176 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。
178 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。
207 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。
211 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。
218 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。
229 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。
237 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。
241 ページの『-qsrcmsg』	#pragma options srcmsg	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。
249 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。
272 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。

表 14. 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
274 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。
275 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。
281 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 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 15. 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
62 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。
63 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 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。
70 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。
85 ページの『-qcache』	なし。	**-O4**、**-O5**、または **-qipa** と指定すると、特定の実行マシンのキャッシュ構成を指定する。

表 15. 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
90 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。
96 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。
100 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。
109 ページの『-qenablevmx』	なし。	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生成を使用可能にする。
113 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。
133 ページの『-qhot』	#pragma nosimd、#pragma novector	最適化中に上位ループ分析およびトランスフォーメーション (HOT) を実行する。
139 ページの『-qignerrno』	#pragma options ignerrno	システム呼び出しによって errno が変更されないと想定してコンパイラーに最適化の実行を許可する。
150 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。
160 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。
171 ページの『-qlargepage』	なし。	ラージ・ページ・メモリー環境で実行するように設計されたアプリケーションのために、POWER4™ 以上のシステムで提供されるラージ・ページを活用する。
175 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。

表 15. 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
186 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。
190 ページの『-qminimaltoc』	なし。	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。
194 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。
199 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。
203 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。
210 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。
212 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。
214 ページの『-Q、-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。
231 ページの『-qshowpdf』	なし。	コンパイル・ステップおよびリンク・ステップで、**-qpdf1** と最小の最適化レベル **-O2** と共に指定された場合、追加のプロファイル情報をコンパイルされたアプリケーションに挿入し、そのアプリケーション内のすべてのプロシージャーの呼び出しとブロックのカウントを収集する。

表 15. 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
232 ページの『-qsmallstack』	なし。	スタック・フレームのサイズを削減する。
233 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。
240 ページの『-qspeculateabsolutes』	なし。	非 IPA リンクの -qtocmerge -bl:file および IPA リンクの -bl:file と連動して絶対アドレスでの投機的実行を使用不可にする。
244 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。
249 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。
260 ページの『-qtocmerge』	なし。	TOC マージを使用可能にして TOC ポインターのロードを削減し、外部ロードのスケジューリングを改善する。
261 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。
267 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。
269 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。

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 16. リンク・オプション

オプション名	同等のプラグマ名	説明
78 ページの『-b』	なし。	共用オブジェクトをリンカーで処理する方法を制御する。
81 ページの『-bmaxdata』	なし。	(初期化済みと初期化されていない両方の) 静的データによって共用される領域の最大サイズとヒープを設定する。
82 ページの『-brtl』	なし。	出力ファイルのランタイム・リンクを制御する。
92 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。
102 ページの『-e』	なし。	**-qmkshrobj** または **-G** オプションと一緒に使用して、共有オブジェクトのエントリー・ポイントを指定する。
111 ページの『-f』	なし。	コンパイラーがリンカーに渡すためのオブジェクト・ファイルのリストを保管するファイルを指定する。
165 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・パスを検索する。
164 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。
174 ページの『-qlib』	なし。	標準システム・ライブラリーおよび XL C ライブラリーがリンクされるかどうかを指定する。
284 ページの『-Z』	なし。	リンカーが使用するライブラリー検索パスのプレフィックスを指定する。

移植性およびマイグレーション

このカテゴリーでのオプションを使用すると、過去、現行、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アプリケーションの振る舞いの互換性を保守することができます。または変更は最小限にとどめたままでご使用のアプリケーションを XL コンパイラーに移動させることができます。

表 17. 移植性およびマイグレーション・オプション

オプション名	同等のプラグマ名	説明
65 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。
129 ページの『-qgenproto』	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。
270 ページの『-qupconv』	#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。
273 ページの『-qvecnvol』	なし。	揮発性ベクトル・レジスターを使用するか不揮発性ベクトル・レジスターを使用するかを指定する。

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。必要なのは、特殊なインストールまたはテスト・シナリオにおいて以下のオプションを使用することのみです。

表 18. 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
76 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。
79 ページの『-B』	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 実行可能ファイルの代替パス名を判別する。
93 ページの『-qc_stdinc』	なし。	XL C およびシステム・ヘッダー・ファイルの標準検索ロケーションを変更する。
112 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。
201 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 実行可能ファイルの代替パス名を判別する。
241 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 18. 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
253 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。
276 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に置き換えられました。アスタリスクが付いているオプション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 19. 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
-qansialias	-qalias=ansi
-qarch = 601 | 602 | 603 | pwr | pwr2 | p2sc | pwr2s | com	-qfloat=nosingle:norndsngl
-qassert	-qalias
-qfloat=emulate*	
-qfold	-qfloat=fold
-qhsflt	-qfloat=hsflt
-qhssngl	-qfloat=hssngl
-qipa=pdfname	-qpdf1=pdfname, -qpdf2=pdfname
-qmaf	-qfloat=maf
-qspnans	-qfloat=spnans
-qrrm	-qfloat=rrm

個々のオプションの説明

この節では、XL 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

ここには、そのオプションが属する機能カテゴリーがリストされています。

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、287 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・オプション設定は構文図にはっきりと示されています。ただし、多くのオプションの場合、有効な他のコンパイラー・オプションによってはデフォルト設定が複数あります。このセクションでは、適用される可能性がある別のデフォルトが示されています。

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、349 ページの『第 5 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。

-# (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

- 272 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 ビットのいずれかのコンパイラー・モードを選択する。

-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その出力を最適化します。

構文

デフォルト

-q32

使用法

-q32 および **-q64** オプションは、OBJECT_MODE 環境変数が存在する場合、この値によって設定されたコンパイラー・モードをオーバーライドします。

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、32 ビット PowerPC 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 -q32 -qarch=ppc
```

関連情報

- 11 ページの『アーキテクチャー固有の 32 ビットまたは 64 ビットのコンパイルでのコンパイラー・オプションの指定』
- 70 ページの『-qarch』
- 261 ページの『-qtune』

-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

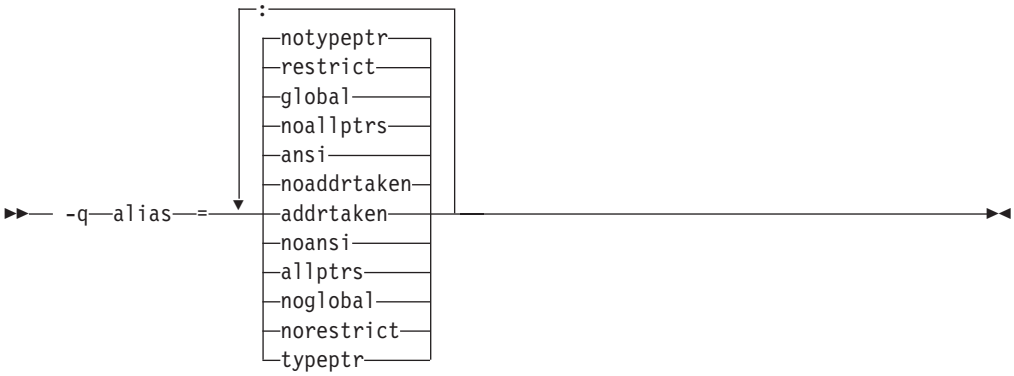
プラグマ同等物

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 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

- **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なのは、 **ansi** が指定されている場合のみです。

ansi | noansi

ansi が有効である場合、最適化で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。最適化プログラムは、ポインターが指すことができるのは、同じ型のオブジェクトだけ であると想定します。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。

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** および **-qalias=ansi** の両方が使用可能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** を指定すると、異なるタイプの 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なのは、 **ansi** が指定されている場合のみです。

使用法

-qalias は、コンパイル中のコードに関する表明をコンパイラーに対して行います。コードに関する表明が 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。

-qalias=[no]ansi オプションは、推奨されない **-q[no]ansialias** オプションに代わるものです。新規アプリケーションでは、**-qalias=[no]ansi** を使用してください。

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

- 150 ページの『-qipa』
- 297 ページの『#pragma disjoint』
- 「*XL 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 ランゲージ・リファレンス*」の*制限型の修飾子*

-qalign

カテゴリー

移植性およびマイグレーション

プラグマ同等物

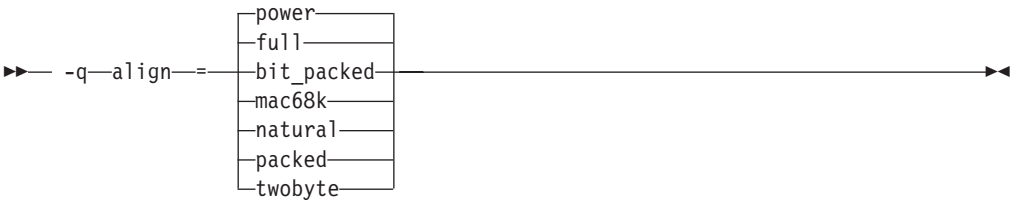
#pragma options align、
#pragma align

目的

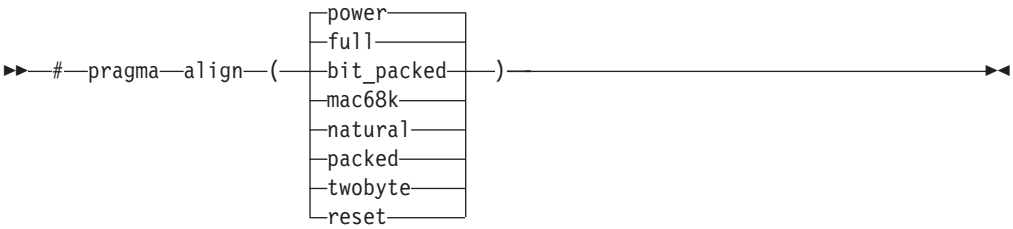
ストレージ内でデータ・オブジェクトの位置合わせを指定する。これによって、誤って配置されたデータが原因で起こるパフォーマンス上の問題を回避できます。

構文

オプション構文



プラグマ構文



デフォルト

full

パラメーター

bit_packed | 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。

full | power
RISC System/6000® の位置合わせ規則が使用されます。

mac68k | twobyte
Macintosh の位置合わせ規則が使用されます。 32 ビットのコンパイルに対してのみ有効です。

natural
構造体メンバーが自然な境界にマップされます。これは、構造体または共用体の最初のメンバーではない `double` および `long double` にも位置合わせ規則を適用することを除き、**power** サブオプションと同じ効果があります。

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

full サブオプションは、既存のオブジェクトとの後方互換性を保証するためのデフォルトです。後方互換性が必要でない場合は、潜在的なアプリケーション・パフォーマンスを改善するために、**natural** 位置合わせを使用することを検討してください。

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。

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 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 `xlc file2.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=power /* Applies to B; A is unaffected  */
   struct B {
```

```
   char c;
   double d;
            } BB;   /*  BB uses power alignment rules  */
} AA;               /*  AA uses bit_packed alignment rules  /*
```

関連情報

- 313 ページの『#pragma pack』
- 「*XL 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 ランゲージ・リファレンス*」の__align 指定子『__align 指定子』
- 「*XL C ランゲージ・リファレンス*」の『aligned 変数属性』
- 「*XL C ランゲージ・リファレンス*」の『packed 変数属性』

-qalloca、-ma

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。

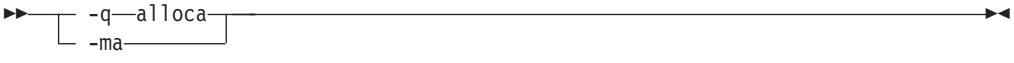
関数 void* alloca(size_t *size*) は、標準ライブラリー関数の malloc のように、メモリーを動的に割り振ります。コンパイラーは以下いずれかの場合に、システム alloca 関数への呼び出しを、インライン組み込み関数 __alloca に自動的に置き換えます。

- 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、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

- 94 ページの『-D』
- 411 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 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、ベクトル処理命令をサポートするターゲット・アーキテクチャーになるよう **-qarch** が設定または暗黙指定されており、**-qenablevmx** コンパイラー・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、コンパイラーは **-qaltivec** を無視して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・プログラミングに対するコンパイラー・サポートを使用可能にするには、以下のように入力します。

```
xlc myprogram.c -qenablevmx -qarch=ppc64v -qaltivec
```

関連情報

- 『-qarch』
- 109 ページの『-qenablevmx』
- 273 ページの『-qvecnvol』
- 「*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

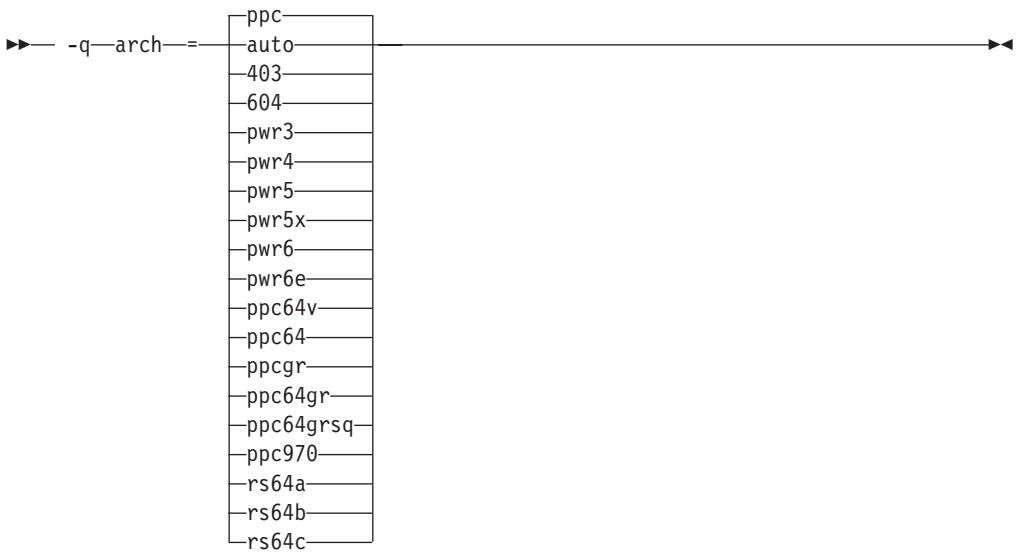
プラグマ同等物

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

- **-q32** が有効な場合は **-qarch=ppc**。
- **-q64** が有効な場合は **-qarch=ppc64**。
- **-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** 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。

403

PowerPC 403™ 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。

604

PowerPC 604™ 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。**-q64** が有効な場合は、このサブオプションは、無効です。

pwr3

POWER3™、POWER4、POWER5™、POWER5+™、POWER6™、または PowerPC 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。

pwr4

POWER4、POWER5、POWER5+、POWER6、または PowerPC 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。

pwr5

POWER5、POWER5+、または 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。

pwr5x
POWER5+ または 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。

pwr6
POWER6 アーキテクチャー・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。 10 進数浮動小数点命令のサポートを希望する場合は、コンパイル中にこのサブオプションを指定してください。

pwr6e
POWER6 ロー・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。

ppc
32 ビット・モードで、32 ビットの PowerPC 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。このサブオプションによって、コンパイラーが、単精度データを指定して使用できる単精度命令を作成できます。**-qarch=ppc** を **-q64** と一緒に暗黙に指定すると、アーキテクチャー設定を **-qarch=ppc64** にアップグレードします。

ppc64
64 ビットの PowerPC ハードウェア・プラットフォームで実行するオブジェクト・コードを作成します。このサブオプションは 32 ビット・モードでコンパイルするときに選択できますが、結果のオブジェクト・コードには、32 ビットの PowerPC プラットフォームで稼働したときに認識されなかったり異なる振る舞いをする命令が含まれる可能性があります。

ppcgr
32 ビット・モードでは、オプションのグラフィック命令をサポートする PowerPC プロセッサーのオブジェクト・コードを作成します。**-qarch=ppcgr** を **-q64** と一緒に暗黙に指定すると、アーキテクチャー設定を **-qarch=ppc64gr** にアップグレードします。

ppc64gr
オプションのグラフィックス命令をサポートする 64 ビットの PowerPC ハードウェア・プラットフォーム用のコードを作成します。

ppc64grsq
オプションのグラフィックスおよび平方根命令をサポートする 64 ビットの PowerPC ハードウェア・プラットフォーム用のコードを作成します。

ppc64v
ベクトル・プロセッサー (PowerPC 970 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。

ppc970
PowerPC 970 アーキテクチャーに固有の命令を生成します。

rs64a
RS64I プラットフォームで稼働するオブジェクト・コードを作成します。

rs64b
RS64II プラットフォームで稼働するオブジェクト・コードを作成します。

rs64c
RS64III プラットフォームで稼働するオブジェクト・コードを作成します。

注: com サブオプションと、POWER™ および POWER2™ アーキテクチャーを表すサブオプションはサポートされなくなりました。このサブオプションが提供する浮動小数点の動作を希望する場合は、**-qfloat=nosingle:norndsngl** オプションを使用してください。詳しくは、116 ページの『-qfloat』を参照してください。

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する **-qarch** サブオプションを指定します。表 20 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 20. 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	ラージ・ページのサポート
604	yes	no	no	no	no
rs64a	no	no	yes	no	no
rs64b	yes	yes	yes	no	no
rs64c	yes	yes	yes	no	no
pwr3	yes	yes	yes	no	no
pwr4	yes	yes	yes	no	yes
pwr5	yes	yes	yes	no	yes
pwr5x	yes	yes	yes	no	yes
ppc	no	no	no	no	yes
ppc64	no	no	yes	no	yes
ppc64gr	yes	no	yes	no	yes
ppc64grsq	yes	yes	yes	no	yes
ppc64v	yes	yes	yes	yes	yes
ppc970	yes	yes	yes	yes	yes
pwr6	yes	yes	yes	yes	yes
pwr6e	yes	yes	yes	yes	yes

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、261 ページの『-qtune』を参照してください。

-q64 を指定すると、以下のとおり有効な **-qarch** 設定が変更されます。

元の -qarch 設定	-q64 を指定したときの有効な設定
ppc	ppc64
ppcgr	ppc64gr

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。 互換性のない **-qarch** 設定でコンパイルされたオブジェクト・ファイルをリンカーとローダーで検出することができますが、それは信頼性がありません。

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、353 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` からコンパイルされた実行可能プログラム testing を、32 ビットの PowerPC 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 -q32 -qarch=ppc
```

関連情報

- 11 ページの『アーキテクチャー固有の 32 ビットまたは 64 ビットのコンパイルでのコンパイラー・オプションの指定』
- 261 ページの『-qtune』
- 61 ページの『-q32、-q64』
- 「*XL 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

アセンブラー言語拡張のコードの解釈と以降の生成を制御する。

-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

デフォルト

- -qasm=gcc

-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。

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

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。

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 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

- **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。

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』
- 166 ページの『-qlanglvl』
- 163 ページの『-qkeyword』
- 「*XL 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

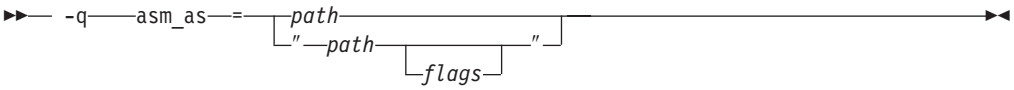
なし。

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

- 74 ページの『-qasm』

-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

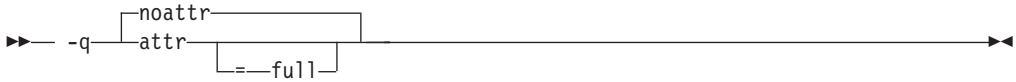
プラグマ同等物

#pragma options [no]attr

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。

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

- 211 ページの『-qprint』
- 281 ページの『-qxref』

-b

カテゴリー

リンク

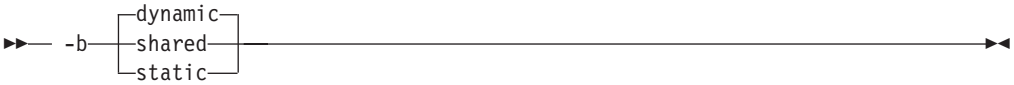
プラグマ同等物

なし。

目的

共用オブジェクトをリンカーで処理する方法を制御する。

構文



デフォルト

-bdynamic

パラメーター

dynamic | **shared**
リンカーに、後続の共用オブジェクトを動的モードで処理させます。動的モードでは、共用オブジェクトは、出力ファイルに静的には組み込まれません。代わりに、共用オブジェクトは、出力ファイルのローダー・セクションにリストされます。

static
リンカーに、後続の共用オブジェクトを静的モードで処理させます。静的モードでは、共用オブジェクトは、出力ファイルに静的にリンクされます。

使用法

デフォルト・オプション **-bdynamic** を使用すると、 C ライブラリー (libc) は必ず動的にリンクされます。 C ライブラリーにリンクするときに未解決のリンカー・エラーに関して起こる可能性のある問題を回避するには、 **-bstatic** オプションを使用するコンパイル・セクションの最後に **-bdynamic** オプションを追加しなければなりません。

事前定義マクロ

適用されません。

関連情報

- 82 ページの『-brtl』

-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

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 実行可能ファイルの代替パス名を判別する。

XL 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、 253 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/usr/vac/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

- 201 ページの『-qpath』
- 253 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=unsigned

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。

-bmaxdata

カテゴリー

リンク

プラグマ同等物

なし

目的

(初期化済みと初期化されていない両方の) 静的データによって共用される領域の最大サイズとヒープを設定する。

構文

デフォルト

-bmaxdata:0

パラメーター

number
システム・ローダーによって設定されたソフト **ulimit** を示すバイト数です。有効値は、0 および 0x10000000 の倍数 (0x10000000、0x20000000、0x30000000、...) です。システムが許可する最大値は、0x80000000 です。値が 0 だと、単一

の 256MB (0x10000000 バイト) データ・セグメント (セグメント 2) が、静的データ、ヒープ、およびスタックで共有されます。値がゼロ以外の場合、指定したサイズのデータ域 (セグメント 3 で始まる) は静的データおよびヒープによって共有され、別の 256MB データ・セグメント (セグメント 2) はスタックによって使用されます。そのため、0 が指定された場合の合計データ・サイズは 256MB になり、0x10000000 を指定した場合の合計サイズは 512MB で、256MB はスタック、別の 256MB は静的データおよびヒープによって共有されます。

事前定義マクロ

なし。

-brtl

カテゴリー

リンク

プラグマ同等物

なし。

目的

出力ファイルのランタイム・リンクを制御する。

ランタイム・リンクとは、プログラム実行が既に開始された後に、共用モジュールの未定義シンボルまたは非据え置きシンボルを解決する能力のことです。これはランタイム定義 (これらの機能定義はリンク時には使用できません) とシンボルの再バインド機能を提供するためのメカニズムです。 **-brtl** でコンパイルするとプログラムにランタイム・リンカーへの参照が追加され、プログラム実行を開始すると、プログラムの開始コード (/lib/crt0.o) によって呼び出されます。共用オブジェクト入力ファイルは、プログラム・ローダー・セクション内の従属として、コマンド行に指定されたときと同じ順序でリストされます。プログラム実行が開始されると、システム・ローダーはこれらの共用オブジェクトをロードし、その定義をランタイム・リンカーで使用できるようにします。

構文

使用法

ランタイム・リンクを使用可能にするには、メイン・アプリケーションを作成する必要があります。システム・ローダーは、主プログラムと呼び出されるモジュールで参照されるすべてのシンボルをロードして解決できなければなりません。それができないと、プログラムは実行されません。

DCE スレッド・ライブラリーおよびヒープ・デバッグ・ライブラリーは、ランタイム・リンクと互換性がありません。 **xlc_r4** を使用してコンパイラーを呼び出してい

る場合や、**-qheapdebug** コンパイラー・オプションが指定されている場合は、**-brtl** コンパイラー・オプションを指定しないでください。

事前定義マクロ

なし。

関連情報

- 78 ページの『-b』
- 129 ページの『-G』

-c

カテゴリー

出力制御

プラグマ同等物

なし。

目的

完了済みのオブジェクトがリンカーへ送信されるのを防ぐ。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だと、コンパイラーは *file_name*.c、*file_name*.i など、有効なソース・ファイルごとに出力オブジェクト・ファイル *file_name*.o を作成します。**-o** オプションを使用すると、オブジェクト・ファイルに明示的な名前を付けられます。

-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

- 103 ページの『-E』
- 192 ページの『-o』
- 200 ページの『-P』
- 252 ページの『-qsyntaxonly』

-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

- 103 ページの『-E』
- 200 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

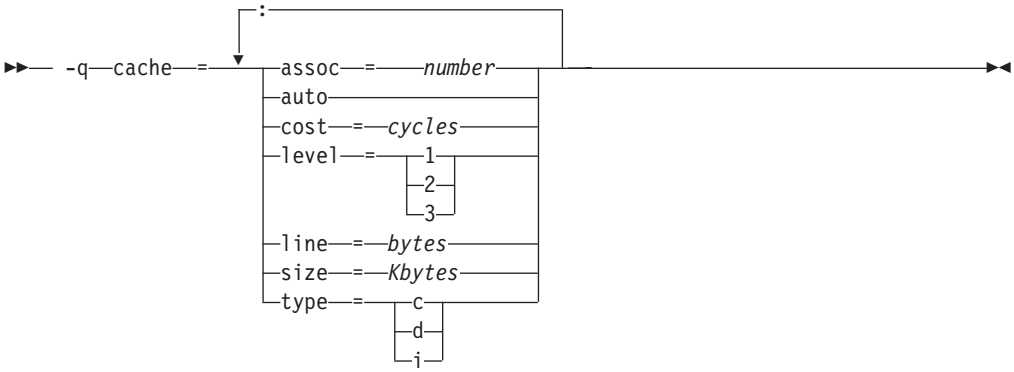
なし。

目的

-O4、**-O5**、または **-qipa** と指定すると、特定の実行マシンの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。

バイト
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 オプションは、**-O4**、**-O5**、または **-qipa** と共に指定する必要があります。

-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。

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

- 85 ページの『-qcache』
- 194 ページの『-O、-qoptimize』
- 261 ページの『-qtune』
- 150 ページの『-qipa』
- 「*XL 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

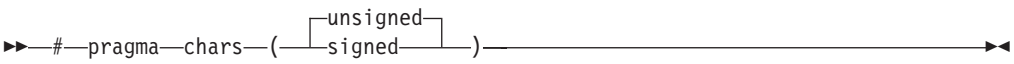
目的

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 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

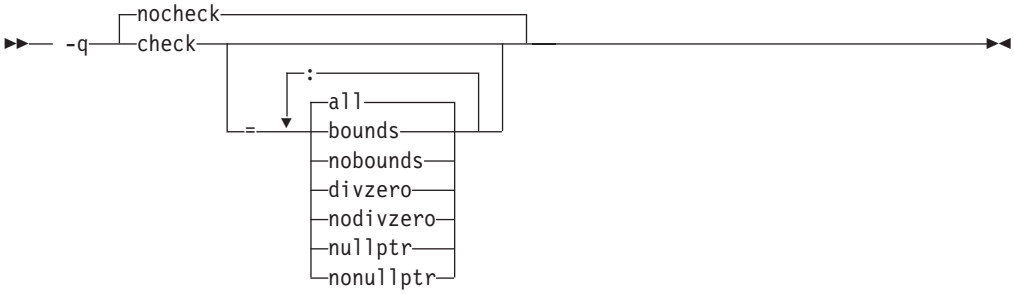
#pragma options [no]check

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り 、ランタイム・エラーが出されます。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。

サブオプションを指定せずに **-qcheck** オプションを指定することは、 **-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ともに **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  */
}
```

-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。

コード・サイズは、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したり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。

構文

デフォルト

-qnocompact

使用法

このオプションが有効になるのは、最適化オプションとともに指定された場合のみです。

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
```

-qcpluscmt

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 //* 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

- 84 ページの『-C、-C!』
- 166 ページの『-qlanglvl』
- 「*XL 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

- 174 ページの『-qlib』

-qc_stdinc

カテゴリー

コンパイラーのカスタマイズ

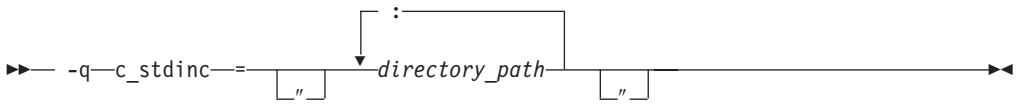
プラグマ同等物

なし。

目的

XL C およびシステム・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

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 /usr/vac/include/) およびシステム・ヘッダー・ファイル (これは通常 /usr/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・ファイルおよびシステム・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 ヘッダーおよびシステム・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

- 243 ページの『-qstdinc』
- 141 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。

プログラムの移植性と標準の準拠を支援するために、オペレーティング・システムでは **-D** オプションで設定できるマクロ名を参照する幾つかのヘッダー・ファイルを提供しています。これらのヘッダー・ファイルのほとんどは、/usr/include ディレクトリーまたは /usr/include/sys ディレクトリーのいずれかに入っています。ソース・ファイルに対する正しいマクロを確実に定義するには、適切なマクロ名を指定して **-D** オプションを使用してください。例えば、ソース・ファイルに /usr/include/sys/stat.h ヘッダー・ファイルが含まれている場合は、-D_POSIX_SOURCE オプションを指定してコンパイルし、そのファイルの正しい定義を選択する必要があります。

-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

AIX v4.2 以降は、2 ギガバイトよりも大きいサイズのファイルをサポートしており、大容量のデータを単一ファイルで保管することができます。ユーザーのアプリケーションで大容量ファイルを操作できるようにするには、 -D_LARGE_FILES および **-qlonglong** コンパイラー・オプションを指定してコンパイルします。例を以下に示します。

```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

`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

- 264 ページの『-U』
- 349 ページの『第 5 章 コンパイラーの事前定義マクロ』
- 「*AIX* ファイル・リファレンス」の『ヘッダー・ファイル』

-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。

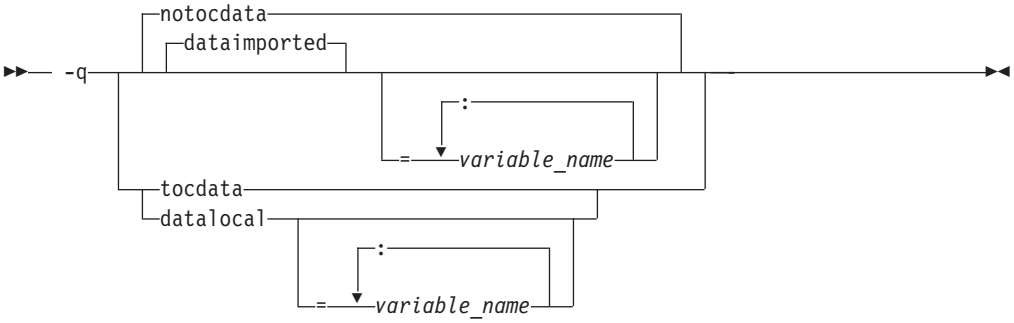
目的

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、ライブラリーの共用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された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された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されたものと想定する変数の名前です (指定したオプションによって異なる)。

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、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

- 212 ページの『-qprocimported、-qproclocal、-qprocunknown』

-qdbxextra

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

- 125 ページの『-qfullpath』
- 175 ページの『-qlinedebug』
- 128 ページの『-g』
- 309 ページの『#pragma options』
- 251 ページの『-qsymtab』

-qdfp

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

10 進浮動小数点型およびリテラルに対するコンパイラー・サポートを使用可能にする。

構文

デフォルト

-qnodfp

使用法

10 進浮動小数点の命令をサポートしない **-qarch** 値に対して **-qdfp** を有効にすると、**-qfloat=dfpemulate** が自動的に使用可能になり、10 進浮動小数点の演算がソフトウェアによって実行されます。 これによって、アプリケーションのランタイム・パフォーマンスが低下する場合があります。

10 進浮動小数点の演算のランタイム・サポートが使用可能なのは、AIX for POWER V5.3 5300-06 Technology Level 以降です。ランタイム・サポートを提供しないオペレーティング・システムのバージョンで **-qdfp** を有効にすると、アプリケーションはコンパイルできますが、リンクまたは実行できない場合があります。

<math.h> 組み込みファイルで定義された 10 進浮動小数点の関数またはマクロを使用するプログラムは、AIX 5.2 または前のレベルの AIX 5.3 あるいは 5.4 ではコンパイルしないでください。これらの関数やマクロは 5.2 に改造できないからです。

事前定義マクロ

-qdfp が有効な場合、__IBM_DFP__ の値は 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

- 70 ページの『-qarch』
- 116 ページの『-qfloat』

-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

キーボードにない文字を表すために、連字キーの組み合わせおよびキーワードの認識を使用可能にする。

構文

デフォルト

- **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。

使用法

連字とは、すべてのキーボードで使用できるわけではない文字を指定することが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 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

- 166 ページの『-qlanglvl』
- 261 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

- 166 ページの『-qlanglvl』

-qdpcl

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

IBM Dynamic Probe Class Library (DPCL) 動的プローブ・クラス・ライブラリー (DPCL) を基にしたツールが実行可能ファイルの構造体を参照するために使用できるシンボルを生成する。

DPCL は、アプリケーションのパフォーマンス分析ツールによって使用されるライブラリーのオープン・ソース・セットです (詳しくは、http://dpcl.sourceforge.net を参照してください)。 **-qdpcl** が有効な場合、コンパイラーはプログラムのコード・ブロックを定義するために記号を出力します。DPCL インターフェースを使用するツールを使用して、このオプションでコンパイルしたオブジェクト・ファイルのメモリー使用量などのパフォーマンス情報を検査することができます。

構文

デフォルト

-qnodpcl

使用法

デバッグおよび分析ツールに必要なデバッグ情報をコンパイラーが確実に生成するようにするためには、**-qdpcl** を **-g** オプションと一緒に指定してください。

-qdpcl は、ゼロを除き、どの最適化レベルでもサポートされていません。ゼロ以外の最適化レベルが他のオプションによって指定されている場合や暗黙指定されている場合は、**-qdpcl** が使用不可になります。

-qipa オプションまたは **-qsmp** オプションを **-qdpcl** と共に指定することはできません。

事前定義マクロ

なし。

関連情報

- 128 ページの『-g』
- 150 ページの『-qipa』
- 233 ページの『-qsmp』

-e

カテゴリー

リンク

プラグマ同等物

なし。

目的

-qmkshrobj または **-G** オプションと一緒に使用して、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

-e=noentry

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

オブジェクト・ファイルをリンクする場合、**-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。 **-e** フラグを指定してこのラベルを変更すると、エラーの原因となる可能性があります。

このオプションと併用できるのは、**-qmkshrobj** または **-G** オプションのみです。詳しくは、 191 ページの『-qmkshrobj』の説明を参照してください。

事前定義マクロ

なし。

関連情報

- 191 ページの『-qmkshrobj』
- 129 ページの『-G』

-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

-E オプションにはどのようなファイル名でも使用できます。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るため、エラー・メッセージは生成されません。

-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

- 209 ページの『-qppline』
- 84 ページの『-C、-C!』
- 200 ページの『-P』
- 252 ページの『-qsyntaxonly』

-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

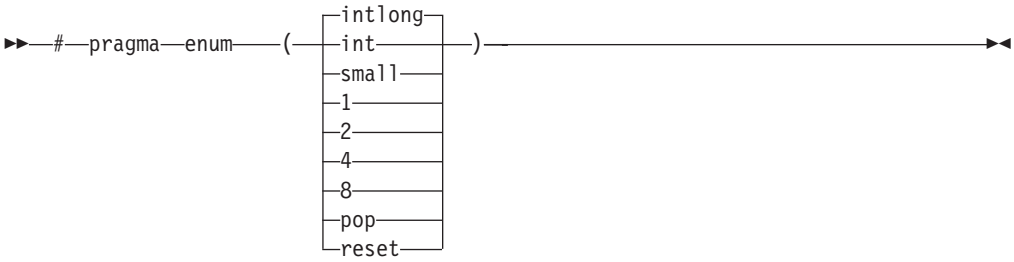
目的

列挙が占有するストレージの量を指定する。

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 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。 値は signed int の範囲を超えることはできません。

4 | int
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 の場合は long 型、そうでない場合は unsigned long 型となります。

intlong
列挙型の値の範囲が int の限度を超えた場合、列挙型はストレージの 8 バイトを占有することを指定します。列挙型の値の範囲が int の限度を超えないと、列挙型はストレージの 4 バイトを占有し、int で表されます。

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、値の範囲が負の値を含んでいない限り、符号なし 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 21. 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	char	int	short	int	int	int	long long	long long	long	long
-128..127	char	int	short	int	int	int	long long	long long	long	long

表 21. 列挙のサイズおよび型 (続き)

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	ERROR[1]	unsigned int	unsigned int	unsigned int	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	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int
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:

1. これらの列挙型は **-qenum=1|2|48** 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。
2. 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** 言語レベルが有効な場合、列挙型の値が int の範囲を超えていると、コンパイラーは以下のように振る舞います。
   a. **-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。
   b. **-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。

#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、警告とともに診断の対象となるためです。

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 /* second reset isn't required */
```

enum 定数の範囲は、unsigned int または int (signed int) の範囲内でなければなりません。例えば、以下のコード・フラグメントにはエラーがあります。

```
#pragma enum=small
enum e_tag { a=-1,
             b=2147483648   /* error: larger than maximum int */
           } e_var;
#pragma options enum=reset
```

事前定義マクロ

なし。

-qenablevmx

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

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生成を使用可能にする。

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

-qnoenablevmx

使用法

ご使用のオペレーティング・システムのバージョンがベクトル命令をサポートしない場合は、**-qenablevmx** を指定しないでください。

-qnoenablevmx が有効であると、**-qaltivec**、**-qvecnvol**、 および **-qhot=simd** は使用できません。

事前定義マクロ

なし。

関連情報

- 69 ページの『-qaltivec』
- 70 ページの『-qarch』
- 133 ページの『-qhot』
- 273 ページの『-qvecnvol』

-qexpfile

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

-qmkshrobj または **-G** オプションと一緒に使用して、1 つの指定ファイルにすべてのエクスポートされたシンボルを保管する。

構文

パラメーター

file_name
エクスポートされた記号が書き込まれるファイルの名前です。

使用法

このオプションが有効なのは、**-qmkshrobj** または **-G** オプションとともに使用した場合のみです。

事前定義マクロ

なし。

関連情報

- 191 ページの『-qmkshrobj』
- 129 ページの『-G』

-qextchk

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]extchk

目的

リンク時の型検査の情報を生成し、コンパイル時の整合性について検査する。

構文

デフォルト

-qnoextchk

使用法

このオプションは、不完全な型への参照を含む関数またはオブジェクトに対する型の検査を行いません。

事前定義マクロ

なし。

例

リンク時の検査情報が生成されるように `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 -qextchk
```

-f

カテゴリー

リンク

プラグマ同等物

なし。

目的

コンパイラーがリンカーに渡すためのオブジェクト・ファイルのリストを保管するファイルを指定する。

構文

使用法

filelistname ファイルには、オブジェクト・ファイルの名前のみを含めます。 1 行につき 1 つのオブジェクト・ファイルを指定します。

このオプションは、**ld** コマンド用の **-f** オプションと同じです。

事前定義マクロ

なし。

例

myobjlistfile に含まれているファイルのリストをリンカーに渡すには、以下のように入力してください。

```
xlc -f/usr/tmp/myobjlistfile
```

-F

カテゴリー

コンパイラーのカスタマイズ

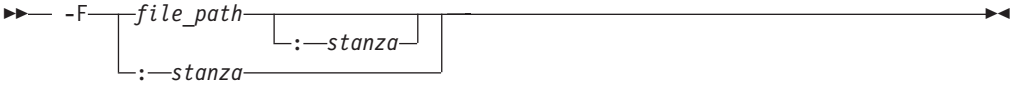
プラグマ同等物

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提供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** でコンパイルする場合、**c99** スタンザを指定すると、コンパイラーは **c99** スタンザで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てください。

```
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てください。

```
xlc myprogram.c -F/usr/tmp/myconfig.cfg:c99
```

関連情報

- 35 ページの『カスタム・コンパイラー構成ファイルの使用』
- 79 ページの『-B』
- 253 ページの『-t』
- 276 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』
- 26 ページの『コンパイル時間およびリンク時間の環境変数』

-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

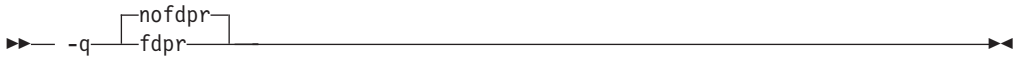
なし。

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

- 203 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

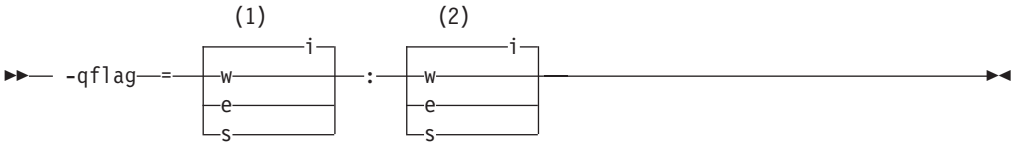
目的

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 syntax – 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。

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 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。

使用法

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。

-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。

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

- 142 ページの『-qinfo』
- 275 ページの『-w』

- 18 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

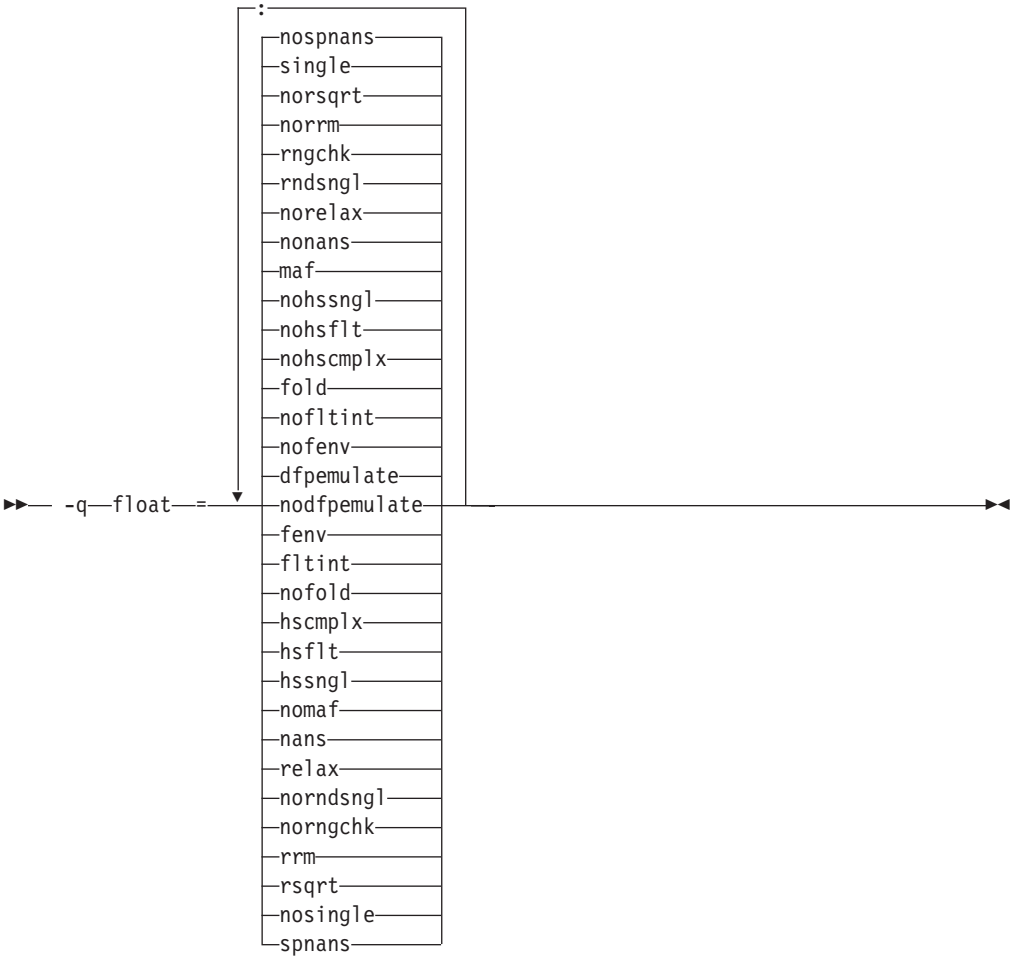
#pragma options float

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

オプション:



デフォルト

- **-qfloat=dfpemulate:nofenv:nofltint:fold:nohscmplx:nohsflt:nohssnglmaf:nonans:norelax:rndsngl:rngchk:norrm:norsqrt:single: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)
- **-qfloat=nocomplexgcc**（64 ビット・モードが使用可能な場合)
- **-qfloat=nodfpemulate**（**-qarch=pwr6** が有効な場合)

パラメーター

dfpemulate | **nodfpemulate**

10 進数浮動小数点計算がハードウェア命令でインプリメントされるか、ライブラリー関数への呼び出しによってソフトウェア内でエミュレートされるかどうかを指定します。 **nodfpemulate** は、10 進数浮動小数点命令をサポートしているシステム上 (すなわち、AIX 5.3 以降で、かつ **-qarch=pwr6** が有効な場合) でのみ有効です。 **nodfpemulate** は、これらのシステムで推奨される設定であり、10 進数浮動小数点の演算および全体のプログラムの実行時パフォーマンスを高めることができます。**dfpemulate** は、他のすべての **-qarch** 値で必要です。

サブオプションが有効になるために **-qdfp** も使用可能にする必要があることに注意してください。

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。

-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。

-qfloat=[no]fold オプションは、非推奨の **-q[no]fold** オプションと置き換わります。新しいアプリケーションで **-qfloat=[no]fold** を使用してください。

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。

hssngl | **nohssngl**

単精度式が、式評価後ではなく、結果がメモリー・ロケーションに格納されるときのみ丸められるよう指定します。 **hssngl** を使用すると、実行時のパフォーマンスが改善され、**hsflt** を使用するよりも安全です。

このオプションでは、**-qfloat=nosingle** が有効なとき、単精度 (float) への倍精度 (double) 式のキャストにのみ影響を与え、格納命令が生成される代入演算子で使用されます。デフォルトの **-qfloat=single** を指定してコンパイルする場合、このオプションは使用しないでください。

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。

-qfloat=[no]maf オプションは、非推奨の **-q[no]maf** オプションと置き換わります。新しいアプリケーションで **-qfloat=[no]maf** を使用してください。

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。

hsflt オプションは、**nans** オプションをオーバーライドします。

-qfloat=[no]nans オプションは、非推奨の **-qfloat=[no]spnans** オプションおよび **-q[no]spnans** オプションに置き換わります。新しいアプリケーションで **-qfloat=[no]nans** を使用してください。

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。

norndsngl | **rndsngl**
式全体が評価されるまで待つのではなく、個々の単精度演算の結果を単精度に丸めます。他のタイプのコンピューターで同様の計算を行った場合の結果と整合性をとるために、スピードを犠牲にします。

このオプションのみが、単精度への倍精度式キャストに効力を与えます。**-qfloat=nosingle** が有効な場合のみ、**norndsngl** を指定してください。

hsflt サブオプションは、**rndsngl** オプションをオーバーライドします。

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。

-qfloat=[no]rrm オプションは、非推奨の **-q[no]rrm** オプションと置き換わります。新しいアプリケーションで **-qfloat=[no]rrm** を使用してください。

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。

single | **nosingle**
単精度算術演算命令が、単精度浮動小数点値で生成されることを許可します。すべての PowerPC プロセッサーは、単精度の命令をサポートします。ただし、前のアーキテクチャーでコンパイルされたアプリケーションの動作 (すべての浮動小数点算術計算が倍精度で実行され、単制度に切り捨てられる) を保持したい場合は、**-qfloat=nosingle:norndsngl** を使用することができます。このサブオプションは、非推奨オプション **-qarch=com|pwr|pwrx|pwr2|p2sc|601|602|603** によって提供される結果と互換性のある計算精度結果を提供します。 **-qfloat=nosingle** は、32 ビット・モードでのみ指定することができます。

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。

hsflt サブオプションは、 **spnans** サブオプションをオーバーライドします。

注: V9.0 リリース以降のコンパイラーでは、**emulate|noemulate** サブオプションは推奨されません。

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。詳しくは、「*XL C 最適化およびプログラミング・ガイド*」の『浮動小数点演算の処理』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

-qfloat=dfpemulate が有効なとき、__IBM_DFP_SW_EMULATION__ が、値 1 に事前定義されます。そうでない場合は、定義されません。

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

- 70 ページの『-qarch』
- 『-qflttrap』
- 172 ページの『-qldbl128、-qlongdouble』
- 244 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

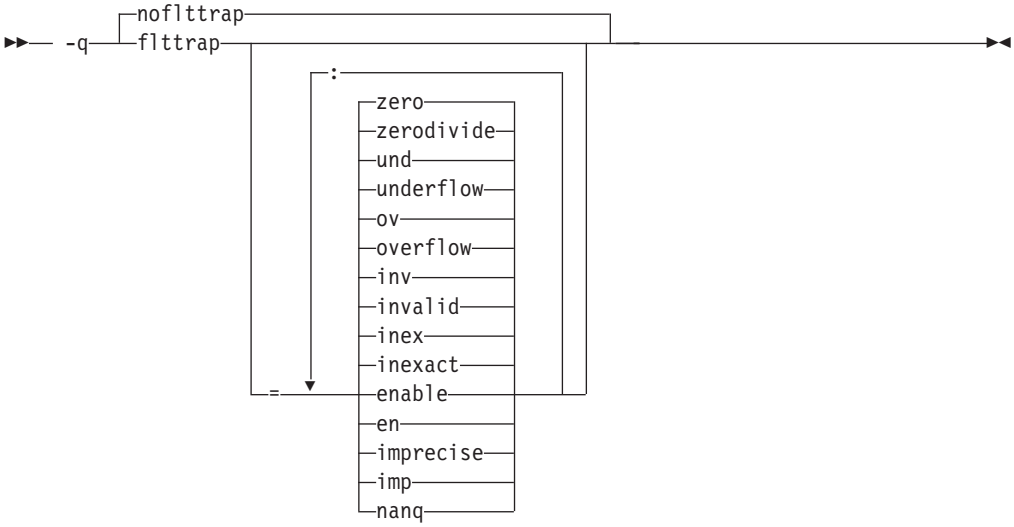
プラグマ同等物

#pragma options [no]flttrap

目的

実行時に検出される浮動小数点例外条件のタイプを判別する。

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**) の

発生時にトラッピングを使用可能にします。ソース・コードを変更せずに例外トラッピングをオンにするには、このサブオプションを指定してください。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ともにプロセスに送信されます。**imprecise** が有効な場合、トラップは例外が発生した正確なロケーションを報告しません。

imprecise、imp
指定された例外の不正確な検出を有効にします。 コンパイラーは、コード・ブロックの後かつ `main` プログラムの直前に命令を生成することで、指定された例外のいずれか (**overflow**、**underflow**、**zerodivide**、**invalid**、または **inexact**) が発生したかを検査します。例外が発生した場合は、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で設定されますが、例外の正確な位置は判別されません。 例外の有無を検査するために、各浮動小数点の演算および関数呼び出しの後で命令が生成されないため、このサブオプションを使用すると、パフォーマンスがやや向上する場合があります。

inexact、inex
浮動小数点の非整数演算の検出が有効になります。 **imprecise** も指定されていない場合、コンパイラーは各浮動小数点演算および関数呼び出しの後に命令を生成することで、非整数演算の例外が発生したかを検査します。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。 **imprecise** も指定されていない場合、コンパイラーは各浮動小数点演算および関数呼び出しの後に命令を生成することで、無効の演算例外が発生したかを検査します。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
代入を含む各浮動小数点の演算の前後に、および値が NaN の場合は浮動小数点の結果をトラップに戻す関数への各呼び出しの後に、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。 **imprecise** も指定されていない場合、コンパイラーは各浮動小数点演算および関数呼び出しの後に命令を生成することで、オーバーフローの例外が発生したかを検査します。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。 **imprecise** も指定されていない場合、コンパイラーは各浮動小数点演算および関数呼び出しの後に命令を生成することで、アンダーフロー例外が発生したかを検査します。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。 **imprecise** も指定されていない場合、コンパイラーは各浮動小数点演算および関数呼び出しの後に命令を生成

することで、ゼロ除算例外が発生したかを検査します。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。

サブオプションを指定せずに **-qflttrap** オプションを指定することは、**-qflttrap=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。このデフォルトには **enable** が含まれないため、ソースで既に `fpsets` または類似のサブルーチンを使用していると便利です。

使用法

-qflttrap 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。

このオプションは、IPA とのリンク時に認識されます。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がオーバーライドされます。

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。

-qflttrap=inv:en を使用して IEEE 無効 SQRT 演算を含むプログラムをコンパイルし、`sqrt` 命令セットを実装しない **-qarch** ターゲットを指定すると、プログラムの実行時に予期された SIGTRAP シグナルは発生しません。この問題を修正するには、プログラム実行前に以下のコマンドを指定してください。

```
export SQRT_EXCEPTION=3.1
```

事前定義マクロ

なし。

例

このプログラムを以下のようにコンパイルします。

```
#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ます。

```
xlc -qfltrap=zerodivide:enable divide_by_zero.c
```

プログラムは、除算の実行時に停止します。

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** サブオプションを使用すると、例外の発生時に SIGTRAP シグナルまたは SIGFPE シグナルが生成されます。

関連情報

- 116 ページの『-qfloat』
- 70 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

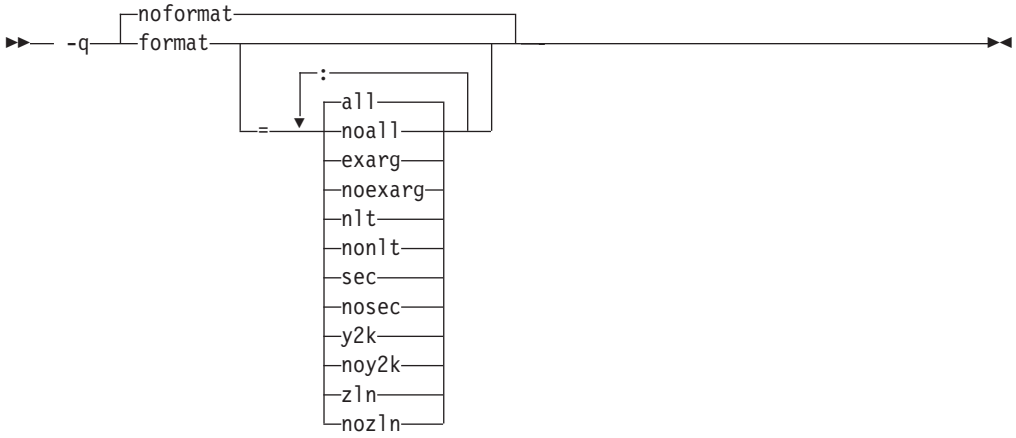
なし。

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

- 175 ページの『-qlinedebug』
- 128 ページの『-g』

-qfuncsect

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]funcsect

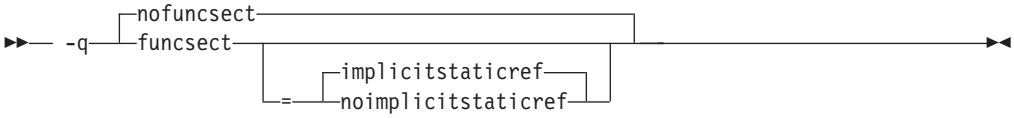
目的

それぞれの関数の命令を別々のオブジェクト・ファイルの制御セクションまたは CSECT に配置する。

-qfuncsect を指定すると、コンパイラーは各関数から静的データ領域 (存在する場合) への参照を生成します。参照が存在する場合、そのオブジェクト・ファイルの関数が最終実行可能ファイルに確実に組み込まれるように、静的データ領域も組み込まれます。これを実行するのは、著作権情報を含む可能性がある、すべての静的ストリングまたはプラグマ・コメントのストリングを実行可能ファイルに確実に組み込むためです。場合によっては、リンク時のコード膨張や未解決のシンボルの原因になることもあります。

-qnofuncsect が有効な場合、各オブジェクト・ファイルは、対応するソース・ファイルに定義されたすべての関数を結合した単一の制御セクションで構成されます。**-qfuncsect** を使用すると、各関数を別の制御セクションに配置できます。

構文



デフォルト

-qnofuncsect

パラメーター

implicitstaticref | noimplicitstaticref

静的変数、仮想関数テーブル、または例外処理テーブルに含まれる関数によるオブジェクト・ファイルの静的データ・セクションへの参照を保持するかどうかを指定します。

#pragma comment ディレクティブまたはコードに著作権情報の静的ストリングが含まれる場合、コンパイラーは、これらのストリングを静的データ領域に自動的に配置し、これらの静的データ領域への参照をオブジェクト・コードに生成します。

implicitstaticref が有効である場合、リンカーのガーベッジ・コレクション・プロシージャーが削除する関数による静的領域へのすべての参照が保持されます。これによって、リンカーによる未解決の関数定義エラーが発生する可能性があります。

noimplicitstaticref が有効な場合、静的領域へのこれらの参照が除去されるため、正常にリンクできるだけでなく、実行可能ファイルのサイズを縮小できる可能性があります。ただし、これによって静的データ領域およびそれに含まれる著作権情報が組み込まれない場合があります。

サブオプションなしで **-qfuncsect** を指定すると、**implicitstaticref** が暗黙指定されます。

使用法

複数の制御セクションを使用すると、オブジェクト・ファイルのサイズが増加しますが、呼び出されない関数や、呼び出されるすべての場所で最適化プログラムによ

ってインライン化された関数の除去をリンカーに許可することにより、最終実行可能ファイルのサイズを縮小することができます。

プラグマ・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

- 295 ページの『#pragma comment』

-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する。

構文

デフォルト

適用されません。

使用法

-g を指定すると、最適化オプションを使用して明示的に要求しない限り、すべてのインライン化がオフになります。

-g とともに使用されるソース・ファイルが絶対パス名または相対パス名のいずれかによって参照されるように指定するには、**-qfullpath** オプションを使用します。

-qlinedebug オプションを使用して、省略したデバッグ情報を生成させ、オブジェクト・サイズを小さくすることもできます。

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、デバッグ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 -g
```

関連情報

- 97 ページの『-qdbxextra』
- 125 ページの『-qfullpath』
- 175 ページの『-qlinedebug』
- 194 ページの『-O、-qoptimize』
- 251 ページの『-qsymtab』

-G

プラグマ同等物

なし。

目的

ランタイム・リンクに使用できる共用オブジェクトを生成する。

構文

使用法

コンパイラーは、**-bE:**、**-bexport:**、または **-bnoexpall** 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、共用オブジェクトからすべてのグローバル・シンボルを自動的に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。エクスポート・リストをファイルに保管するには、**-qexpfile** オプションを使用します。

事前定義マクロ

なし。

関連情報

- 78 ページの『-b』
- 82 ページの『-brtl』
- 110 ページの『-qexpfile』
- 191 ページの『-qmkshrobj』
- 279 ページの『-qweakexp』
- リンクを制御するオプション: リンカー出力制御のオプション
- 「*AIX プログラミングの一般概念: プログラムの作成とデバッグ*」の『共用オブジェクトとランタイム・リンク』
- 「*AIX コマンド・リファレンス 第 3 巻: i から m*」の中の『**ld**』

-qgenproto

カテゴリー

移植性およびマイグレーション

プラグマ同等物

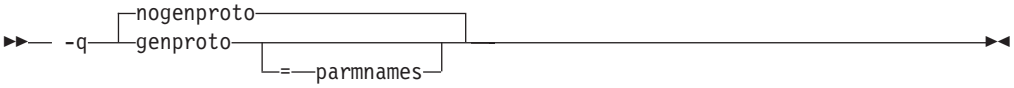
なし。

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

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。

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 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。

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、21 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。

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

- 114 ページの『-qflag』
- 185 ページの『-qmaxerr』

-qheapdebug

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

デバッグ・バージョンのメモリー管理関数を使用可能にする。

コンパイラーは、stdlib.h (_debug_calloc および _debug_malloc など) で定義された標準メモリー管理関数のデバッグ・バージョンのセットとともに出荷されます。これらの関数のヘッダー・ファイルは、製品の組み込みディレクトリー (usr/vac/include) に格納されます。デフォルトでは、コンパイラーは通常のメモリー管理関数 (calloc および malloc など) を使用し、それぞれのローカル・ストレージを事前に初期化しません。**-qheapdebug** が有効な場合、コンパイラーはまず、メモリー管理関数のデバッグ・バージョンが保管されている製品の組み込みディレクトリーでヘッダー・ファイルを検索し、次にシステムの組み込みディレクトリーを検索します。

構文

デフォルト

-qnoheapdebug

使用法

デバッグ・メモリー管理関数の詳細については、「*XL C 最適化およびプログラミング・ガイド*」の 『メモリー・デバッグ・ライブラリー関数』を参照してください。

事前定義マクロ

-qheapdebug が有効な場合は、__DEBUG_ALLOC__ が 1 に定義されます。それ以外の場合は未定義です。

例

デバッグ・バージョンのメモリー管理関数を使用して `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc -qheapdebug myprogram.c -o testing
```

関連情報

- 「*XL C 最適化およびプログラミング・ガイド*」の 『メモリー・ヒープのデバッグ』

-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、#pragma nosimd

目的

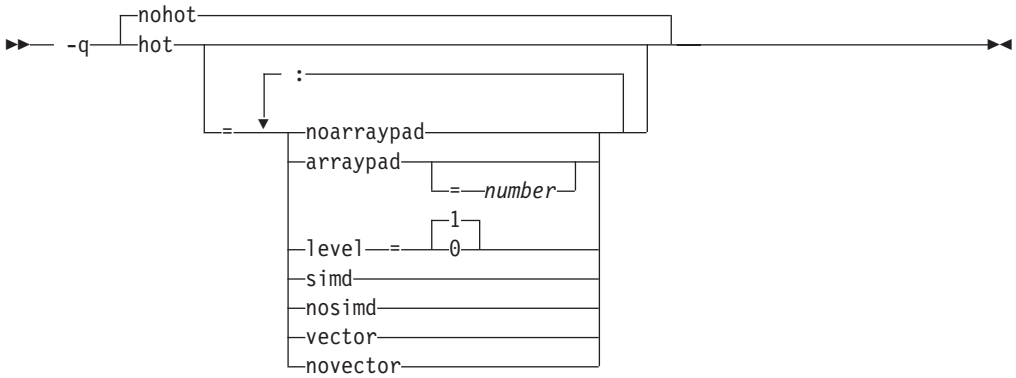
最適化中に上位ループ分析およびトランスフォーメーション (HOT) を実行する。

-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。

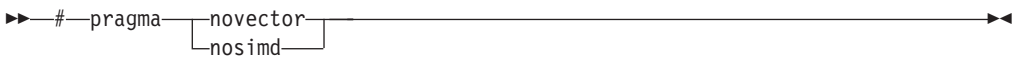
プラグマ・ディレクティブを使用して、選択したセクションのコードの変換を使用不可にすることができます。

構文

オプション　構文



プラグマ構文



デフォルト

- -qnohot
- **-O3** が有効なとき、**-qhot=noarraypad:level=0:nosimd:vector**
- **-qsmp**、**-O4** または **-O5** が有効なとき、**-qhot=noarraypad:level=1:nosimd:vector**
- **-qhot** をサブオプションなしで指定すると、**-qhot=noarraypad:level=1:nosimd:vector**と同等になります。 SPU では **-qhot** をオプションなしで指定すると、-qhot=level=1:simd:noarraypad:novector と等価になります。-qhot をサブオプションなしで **-qenablevmx** およびベクトル処理をサポートする **-qarch** 値と一緒に指定すると、**-qhot=simd** はデフォルトで使用可能になります。

パラメーター

arraypad | 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:nosimd:noarraypad** 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。

simd (オプションのみ) | nosimd
simd が有効なとき、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操作をベクトル命令への呼び出しに変換します。この呼び出しは 1 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。このサブオプションの適用は、重要なイメージ処理要求を持つアプリケーションに役立ちます。

このサブオプションは、ベクトル処理をサポートするアーキテクチャーを指定し、**-qenablevmx** が使用可能な場合のみ有効です。

注: このサブオプションは 5300-03 を使用した AIX 5L™ Version 5.3 推奨保守パッケージ以降でサポートされます。

nosimd は、ループ配列演算のベクトル命令呼び出しへの変換を使用不可にします。

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。操作がループしている場合、ルーチンのベクトル・バージョンが呼び出されます。操作がスカラーである場合、ルーチンのスカラー・バージョンが呼び出されます。**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。

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。

-O3 を指定した場合、コンパイラーは **-qhot=level=0** と見なします。**-O3** でのすべてのホット最適化を防ぐには、**-qnohot** を指定してください。

-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** 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** 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した疑似C レポートを作成することもできます。詳しくは、218 ページの『-qreport』を参照してください。

事前定義マクロ

なし。

例

次の例は、**-qhot=simd** を特定のループで使用不可にする **#pragma nosimd** の使用法を示します。

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}

...
```

関連情報

- 70 ページの『-qarch』
- 109 ページの『-qenablevmx』
- 194 ページの『-O、-qoptimize』
- 244 ページの『-qstrict』
- 233 ページの『-qsmp』
- 「*XL 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、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。

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

- 『-qidirfirst』
- 243 ページの『-qstdinc』
- 141 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 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

- 136 ページの『-I』
- 141 ページの『-qinclude』
- 243 ページの『-qstdinc』
- 93 ページの『-qc_stdinc』
- 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

システム呼び出しによって errno が変更されないと想定して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。これによってコンパイラーは、システム呼び出しによって errno が変更されないと想定して最適化を実行できます。

構文

デフォルト

- -qnoignerrno
- **-O3** より高いレベルの最適化が有効な場合は **-qignerrno** です。

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

なし。

関連情報

- 194 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

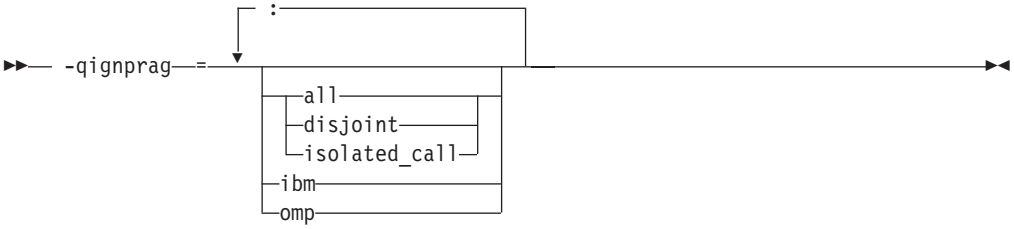
#pragma options [no]ignprag

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
ソース・ファイル内のすべての **#pragma ibm snapshot** ディレクティブおよびすべての IBM SMP ディレクティブ (**#pragma ibm parallel_loop** や **#pragma ibm schedule** など) 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

- 297 ページの『#pragma disjoint』
- 160 ページの『-qisolated_call』

- 302 ページの『#pragma ibm snapshot』
- 327 ページの『並列処理のプラグマ・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

適用されません。

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 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。

-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。

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。

事前定義マクロ

なし。

例

ファイル `foo1.h` およびファイル `foo2.h` をソース・ファイル `foo.c` に組み込むには、以下のように入力します。

```
xlc -qinclude=foo1.h foo.c  -qinclude=foo2.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

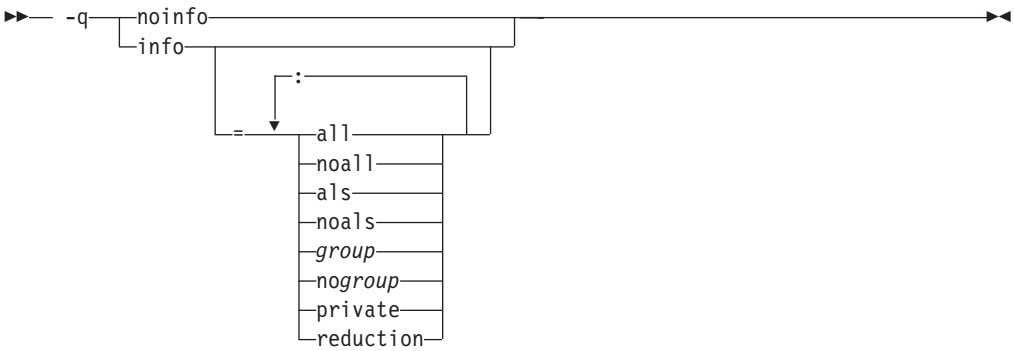
目的

通知メッセージのグループを作成または抑制する。

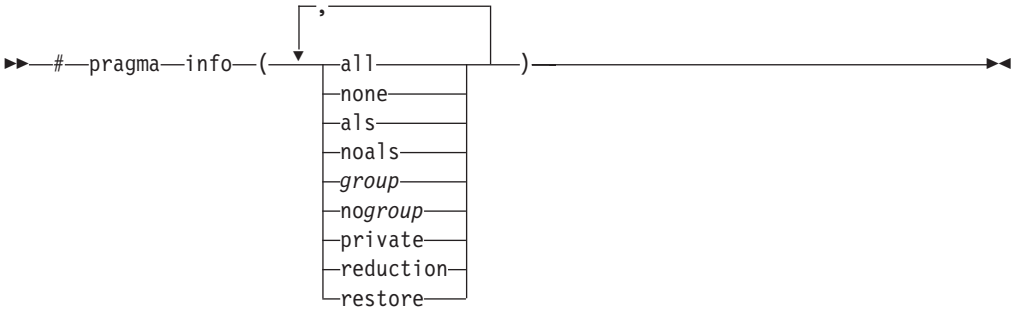
メッセージが標準出力に書き込まれますが、オプションで、リスト・ファイルが生成されていればそれに書き込むこともできます。

構文

オプション構文



プラグマ構文



デフォルト

-qnoinfo

パラメーター

all すべてのグループのすべての診断メッセージを有効にします。

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。

private
並列ループに対して private になった共用変数をリストします。

reduction
並列ループの中で縮小変数として認識されたすべての変数をリストします。

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

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。

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。

ini | noini
初期化で起こりうる問題。

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。

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。

zea | nozea
ゼロ・エクステントの配列。

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。

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。

使用法

-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。
- どのレベルの最適化またはインライン化が行われた場合 **-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ついて通知メッセージを作成するには、以下のように入力します。

```
xlc myprogram.c -qinfo=all -qinfo=nocnv:norea
```

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);
}
```

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.
```

関連情報

- 114 ページの『-qflag』

-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
2 桁の 16 進バイト値です。

使用法

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ランタイム・パフォーマンスが低下するため、デバッグ目的のみに使用してください。

事前定義マクロ

なし。

例

自動変数が 16 進 FF (10 進 255) に初期化されるように 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 -qinitauto=FF
```

-qinlglue

カテゴリー

オブジェクト・コード制御

プラグマ同等物

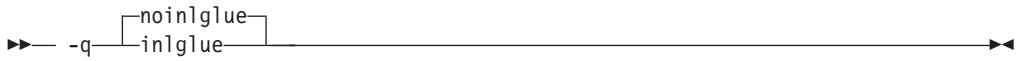
#pragma options [no]inlglue

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。 **inlglue** が有効な場合、最適化プログラムはグルー・コードをインライン化してパフォーマンスを向上させます。 **noinlglue** が有効な場合、グルー・コードのインライン化は回避されます。

構文



デフォルト

- -qnoinlglue
- **-qtune=pwr4** 以上、**-qtune=auto**、または **-qtune=balanced** が有効な場合 (つまり、**-qtune=pwr4 | pwr5 | pwr6 | ppc970**、または **-qtune=auto | balanced** が適切な POWER4 またはそれより新しいプロセッサーを搭載したマシンで有効な場合) は、**-qinlglue** です。

使用法

-qinlglue を暗黙指定するサブオプションのいずれかを指定して **-qtune** オプションを使用し、グルー・コードのインライン化を無効にする場合は、**-qnoinlglue** も必ず指定してください。

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。指定された他のオプションに関わらず **-qcompact** は **-qinlglue** 設定をオーバーライドします。**-qinlglue** を有効にする場合は **-qcompact** を指定しないでください。

-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

- 90 ページの『-qcompact』
- 212 ページの『-qprocimported、-qproclocal、-qprocunknown』
- 261 ページの『-qtune』

-qinline

214 ページの『-Q、-qinline 』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

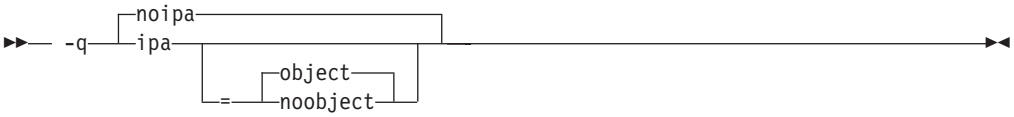
プロシージャー間分析 (IPA) と呼ばれる最適化のクラスを使用可能にしたり、カスタマイズする。

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。

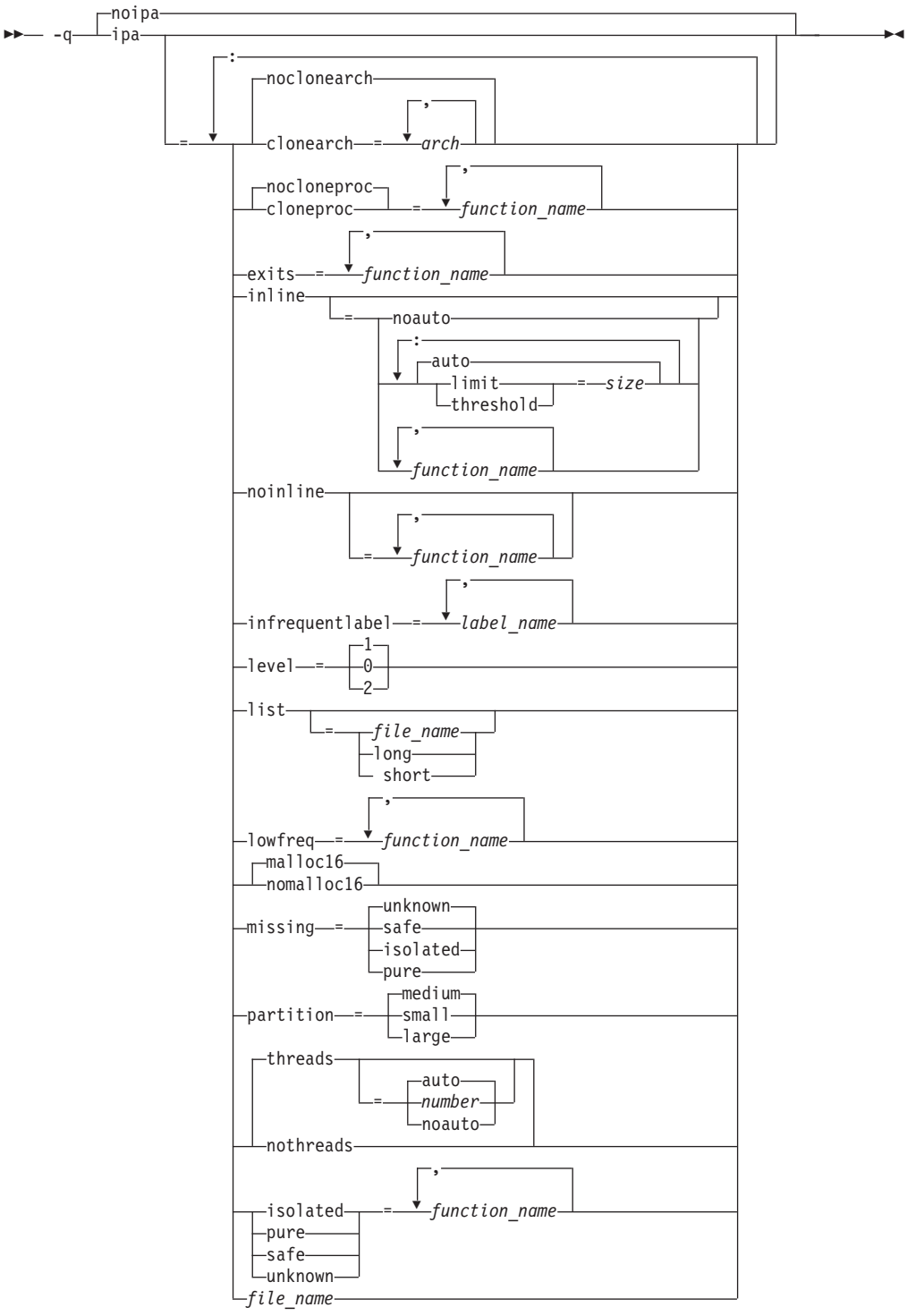
コンパイル・ステップまたはリンク・ステップ、あるいは両ステップの実行中に **-qipa** を使用できます。単一のコンパイラー呼び出しでコンパイルおよびリンクする場合、リンク時のサブオプションのみが関連性を持ちます。別のコンパイラー呼び出しでコンパイルおよびリンクする場合、コンパイル・ステップ実行中にはコンパイル時のサブオプションのみが関連性を持ち、リンク・ステップ実行中にはリンク時のサブオプションのみが関連性を持ちます。

構文

-qipa コンパイル時構文



-qipa リンク時構文



デフォルト

- **-qnoipa**
- **-O4** が有効な場合は
  -qipa=inline=auto:level=1:missing=unknown:partition=medium:threads=auto

- **-O5** が有効な場合は
  -qipa=inline=auto:level=2:missing=unknown:partition=medium:threads=auto
- **-qpdf1** または **-qpdf2** が有効な場合は
  -qipa=inline=auto:level=0:missing=unknown:partition=medium:threads=auto

パラメーター

別のコンパイル・ステップの実行中のみに指定できるパラメーターを以下に示します。

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。

同じコンパイラー呼び出しでコンパイルとリンクを合わせて実行している間に指定できるパラメーター、および個別のリンク・ステップのみの実行中に指定できるパラメーターを以下に示します。

clonearch | **noclonearch**
同じ命令セットの複数のバージョンが作成されるアーキテクチャーを指定します。

clonearch が有効な場合、IPA リンク・フェーズの間、コンパイラーは有効な **-qarch** 設定に基づいて命令セットの汎用バージョンを生成し、必要であれば、**clonearch** サブオプションで指定したアーキテクチャーに対する命令セットの特殊バージョンを*クローン作成* します。コンパイラーは、アプリケーションにコードを挿入することで、ランタイムのプロセッサー・アーキテクチャーを検査します。実行時に、ランタイム環境に最適化されたアプリケーションの命令セットのバージョンが選択されます。ご使用のアプリケーションと同じコピーを実行する複数の異なるマシン上で最適のパフォーマンスを実現するには、このサブオプションを使用します。

-qcompact が有効な場合、このサブオプションは使用不可になります。

arch
コンマで区切られたアーキテクチャーのリストです。 サポートされる値は以下のとおりです。

- pwr4
- pwr5
- ppc970
- pwr6

値を指定しなかったり、無効な値や **-qarch** 設定と等しい値を指定すると、このオプションでは関数のバージョン管理が実行されません。以下のテーブルには、異なるアーキテクチャーごとに使用できる **clonearch** がリストされています。

表 22. 互換性のあるアーキテクチャーと clonearch 設定

-qarch 設定	**許可される clonearch 値**
ppc, pwr3, ppc64, ppcgr, ppc64gr, ppc64grsq	pwr4, pwr5, ppc970, pwr6
pwr4	pwr5, ppc970, pwr6
ppc64v	ppc970, pwr6
pwr5	pwr6
ppc970	pwr6
pwr6	なし

複数のプラットフォーム間の互換性を保証するには、**-qarch** オプションが、**-qipa=clonearch** によって指定されたアーキテクチャーのサブセットと同じ値に設定されていなければなりません。**-qipa=clonearch** および **-qarch** に対してターゲット・アーキテクチャーと一致しないサブオプションが指定されている場合、コンパイラーはアプリケーションが現在稼働しているシステム上で最も一致度が高いサブオプションを基に命令を生成します。

-qreport オプションを **-qipa=clonearch** と併用すると、関数のクローン方法を説明するレポートが作成されます。詳しくは、218 ページの『-qreport』を参照してください。

cloneproc | **nocloneproc**

-qipa=clonearch が有効な場合、**cloneproc** は、指定された 関数のみをクローン作成することを指定します。**nocloneproc** は、クローン作成しない関数を指定します。コンパイラーはデフォルトで、**lowfreq** サブオプションを使用して頻度の低い関数として指定される関数のクローン作成を回避します。

function_name

すべてのサブオプションの場合、関数の名前、またはコンマ区切りされた関数のリストです。

正規表現の構文を使用すると、関数名をサブオプションとして認識するすべてのサブオプションの名前を指定できます。正規表現を指定するための構文規則は以下の通りです。

式	**説明**
string	*string* で指定された任意の文字と一致します。例えば、`test` は、`testimony`、`latest`、および `intestine` と一致します。
^*string*	*string* で指定されたパターンが行の先頭にある場合にのみ、そのパターンと一致します。
*string***$**	*string* で指定されたパターンが行の終わりにある場合にのみ、そのパターンと一致します。
str.ing	ピリオド (.) は任意の 1 文字と一致します。例えば、`t.st` は、`test`、`tast`、`tZst`、および `t1st` と一致します。

式	説明
string¥*special_char*	円記号 (¥) は、特殊文字をエスケープするために使用できます。例えば、ピリオドで終わる行を検索したい場合、式 `.$` を指定するだけで、任意の文字を少なくとも 1 つ含むすべての行が表示されます。`¥.$` を指定すると、ピリオド (.) がエスケープされ、突き合わせでは通常の文字として扱われます。
[*string*]	*string* で指定された任意の文字と一致します。例えば、`t[a-g123]st` は、`tast` および `test` と一致しますが、`t-st` または `tAst` とは一致しません。
[^*string*]	*string* で指定されたどの文字とも突き合わせを行いません。例えば、`t[^a-zA-Z]st` は、`t1st`、`t-st`、および `t,st` と一致しますが、`test` または `tYst` とは一致しません。
*string**	*string* で指定されたパターンの 0 回以上のオカレンスと一致します。例えば、`te*st` は、`tst`、`test`、および `teeeeeest` と一致します。
string+	*string* で指定されたパターンの 1 回以上のオカレンスと一致します。例えば、`t(es)+t` は、`test` および `tesest` と一致しますが、`tt` とは一致しません。
string?	***string*** で指定されたパターンの 0 または 1 回のオカレンスと一致します。例えば、`te?st` は、`tst` または `test` と一致します。
string{*m*,*n*}	*string* で指定されたパターンの、*m* から *n* 回のオカレンスと一致します。例えば、`a{2}` は `aa` と一致し、`b{1,4}` は `b`、`bb`、`bbb`、および `bbbb` と一致します。
string1 | *string2*	*string1* または *string2* のいずれかで指定されたパターンと一致します。例えば、`s | o` は文字 `s` と `o` の両方と一致します。

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。

inline
ハイレベルの最適化プログラムによる関数のインライン化を有効にします。有効なサブオプションは以下のいずれかです。

auto | **noauto**
ハイレベルの最適化プログラムによる関数の自動インライン化を有効または無効にします。**-qipa=inline=auto** が有効な場合、コンパイラーは、最大サイズの制限 (以下を参照) を下回るすべての関数をインライン化の対象として認識します。 **-qipa=inline=noauto** が有効な場合、*function_name* サブオプションでリストされた関数のみがインライン化の対象として認識されます。

limit

-qipa=inline=auto が有効な場合、インライン化後の呼び出し関数のサイズ制限を指定します。

threshold

-qipa=inline=auto が有効な場合、呼び出された関数がインライン化対象として認識されるためのサイズ制限を指定します。

size

インライン化前後の関数の相対サイズを示す負でない整数です。*size* は、呼び出された関数の見積もりサイズや 関数への呼び出し回数などの要素の組み合わせを示す任意の値です。*size* を指定しないと、デフォルトで **threshold** サブオプションが 1024 に、**limit** サブオプションが 8192 に設定されます。この数値をもっと大きくすると、コンパイラーはより大きな関数のインライン化、またはもっと多くの関数呼び出しのインライン化、あるいはその両方を実行できます。

サブオプションなしで **-qipa=inline** を指定することは、**-qipa=inline=auto** を指定することと同等です。

注: デフォルトで、コンパイラーは *function_name* サブオプションで指定した関数のみでなく、すべての関数のインライン化を試みます。特定の関数のみに対するインライン化をオンにするには、**inline=***function_name* を指定してから、**inline=noauto** を指定します (サブオプションはこの順序で指定してください)。例えば、`sub1` を除くすべての関数に対するインライン化をオフにするには、**-qipa=inline=sub1:inline=noauto** を指定します。

noinline

サブオプションなしで指定すると、ハイレベルの最適化プログラムによる関数の自動インライン化を無効にします (**-qipa=inline=noauto** と同等です)。(インライン化は、コンパイラーのフロントエンドまたは低レベル最適化プログラムによって継続される場合があります。詳しくは、214 ページの『-Q、-qinline 』を参照してください。) *function_name* サブオプションと一緒に使用すると、ハイレベルの最適化プログラムによる自動インライン化の対象として認識されない関数を指定します。

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いずれかです。

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。

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。

malloc16 | nomalloc16

動的メモリー割り振りルーチンが 16 バイトの位置合わせされたメモリー・アドレスを戻すことをコンパイラーに通知します。その後、コンパイラーはその表明を基にコードを最適化することができます。

64 ビット・モードでは、AIX は常に 16 バイトの位置合わせアドレスを戻すため、デフォルトで **-qipa=malloc16** が有効になります。**-qipa=nomalloc16** を使用して、デフォルト設定をオーバーライドすることができます。

注: -qipa=malloc16 を使用して生成した実行可能プログラムが、動的メモリー割り振りが 16 バイトの位置合わせアドレスを戻す環境で稼働することを確認する必要があります。稼働しない場合、誤った結果が生成される可能性があります。例えば、32 ビット・モードではアドレスは 16 バイトに調整されていません。この場合、MALLOCALIGN=16 ランタイム環境変数を設定しなければなりません。

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。共用ライブラリーからバインドされた関数は*分離された (isolated)* 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 (isolated)* 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 (isolated)*、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
パス 2 中に IPA によって作成される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 (isolated)* 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サブオプションは以下のとおりです。

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。

ファイルの形式は以下のとおりです。

```
# ... comment
attribute{, attribute} = name{, name}
clonearch=arch,{arch}
cloneproc=name,{name}

nocloneproc=name,{name}missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
inline
inline [ = auto | = noauto ]
inline = name{, name} [ from name{, name}]
inline-threshold = unsigned_int
inline-limit = unsigned_int
list [ = file-name | short | long ]
noinline
noinline = name{, name} [ from name{, name}]
level = 0 | 1 | 2
partition = small | medium | large
```

ここで、*attribute* は以下のいずれかです。

- clonearch
- cloneproc
- nocloneproc
- exits
- lowfreq
- unknown
- safe
- isolated
- pure

リンク・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=inline=auto:level=1:missing=unknown:partition=medium:threads=auto** と同等です。

注: 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、203 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。パフォーマンスをさらに向上するには、**-Q** オプションを指定します。 **-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**、**dump**、 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 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 -qipaxlc
 -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

- 214 ページの『-Q、-qinline 』
- 160 ページの『-qisolated_call』
- 298 ページの『#pragma execution_frequency』
- 203 ページの『-qpdf1、-qpdf2』
- 225 ページの『-S』

- 「*XL 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』

-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。

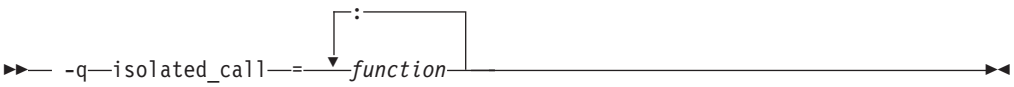
基本的には、ランタイム環境の状態における変更は、以下のような副次作用と見なされます。

- 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

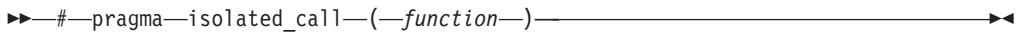
関数を isolated としてマーク付けすると、呼び出された関数によって外部変数および静的変数を変更できないことと、必要に応じてストレージへの不正な参照を呼び出し関数から削除できることが最適化プログラムに通知されます。命令はより自由にリオーダーでき、その結果、パイプラインの遅延が少なくなり、プロセッサーの実行が速くなります。同じパラメーターを指定している同じ関数に対する複数の呼び出しを結合することが可能で、結果が不要であれば、呼び出しを削除することができて、呼び出し順序を変更することができます。

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 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。

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してください。**#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

- 139 ページの『-qignprag』

-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すると、関数仮パラメーターをスタックに保管するかどうかを指定する。

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

- 194 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 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。

このオプションは、以下の C キーワードに使用できます。

- asm
- inline
- restrict
- typeof

注: asm は、**-qlanglvl** オプションが **stdc89** または **stdc99** に設定されている場合、キーワードではありません。

事前定義マクロ

- **-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。
- **-qkeyword=asm** が有効な場合、__IBM_GCC_ASM は 1 に定義されています。
- **-qkeyword=typeof** が有効な場合、__IBM__TYPEOF__ は 1 に定義されています。

例

以下の呼び出しを使用すると typeof を復元できます。

```
xlc -qkeyword=typeof
```

関連情報

- 74 ページの『-qasm』

-l

カテゴリー

リンク

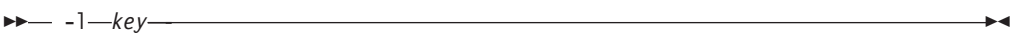
プラグマ同等物

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 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** または **-Z** オプションを使用して変更できます。(静的リンクまたは動的リンクの場合に) 検索されるライブラリーのタイプの指定に関する詳細については、79 ページの『-B』、 82 ページの『-brtl』、および 78 ページの『-b』を参照してください。

-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』
- 78 ページの『-b』
- 82 ページの『-brtl』
- 284 ページの『-Z』
- 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

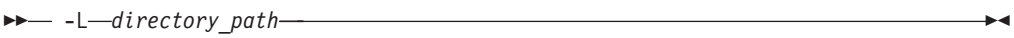
プラグマ同等物

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

共用ライブラリーを実行可能ファイルにリンクするとき、リンク中に **-L** オプションでライブラリーへのパスを指定すると、パス情報が実行可能ファイルにも組み込まれます。このため、共有ライブラリーは実行時に正しく検出されます。このリンク中に **-L** でパスを指定せず、さらに **-bnolibpath** リンカー・オプションを使用してコンパイラーが **-L** 引数をリンカーに自動的に渡すのを妨げた場合、LIBPATH 環境変数によって指定されたパスのみが実行可能ファイルに組み込まれます。

-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

- 164 ページの『-l』

-qlanglvl

カテゴリー

言語エレメント制御

プラグマ同等物

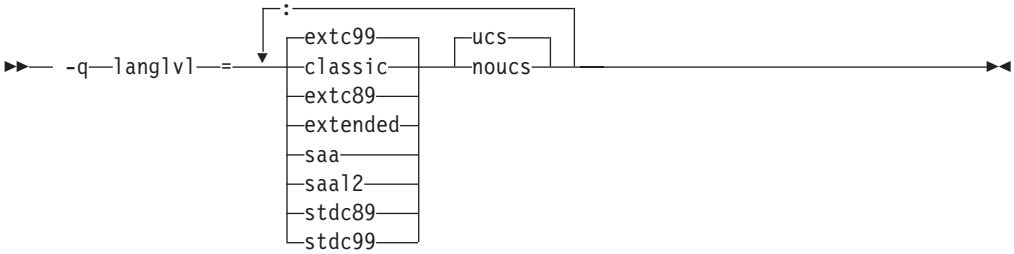
#pragma options langlvl、#pragma langlvl

目的

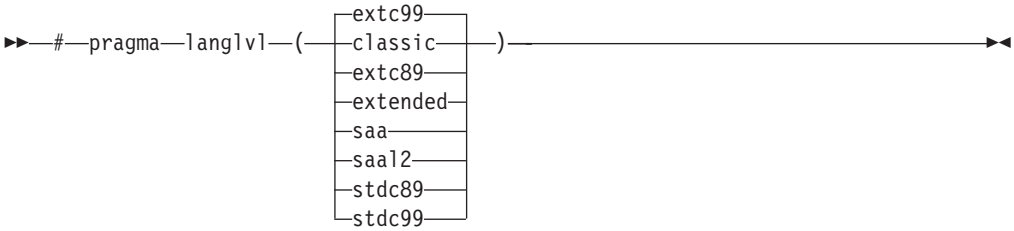
ソース・コードおよびコンパイラー・オプションが固有の言語標準、または標準のサブセットまたはスーパーセットに準拠しているかを検査するかどうかを判別する。

構文

オプションの構文



プラグマ構文



デフォルト

- 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**

パラメーター

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。この言語レベルは AIX V5.1 以上のシステムのヘッダー・ファイル (math.h など) ではサポートされていません。 AIX V5.1 以上のシステ

ムのヘッダー・ファイルを使用する場合は、プログラムを **stdc89** または **extended** 言語レベルにコンパイルすることを検討してください。

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っている可能性がありますが、 `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。ディレクティブの名前、および以下の例では (が見えている場合にのみ、\ を使用して行を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。

extc89

コンパイルは、ANSI C89 標準に準拠しており、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 標準に準拠しており、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。

saal2

SAA C レベル 2 CPI 言語定義に準拠するコンパイル。これには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。

注: C99 に必要なヘッダー・ファイルとランタイム・ライブラリーは、すべてのオペレーティング・システム・リリースでサポートされているわけではありません。

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 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。

使用法

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。

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、 355 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

- 249 ページの『-qsuppress』
- 「*XL C ランゲージ・リファレンス*」の『IBM XL C 言語拡張』

-qlargepage

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

ラージ・ページ・メモリー環境で実行するように設計されたアプリケーションのために、POWER4 以上のシステムで提供されるラージ・ページを活用する。

ラージ・ページ環境用に設計されたプログラムをコンパイルするために **-qlargepage** が有効だと、パフォーマンスが向上する場合があります。

構文

デフォルト

-qnolargepage

使用法

このオプションが役立つのは、以下の条件に当てはまる場合のみです。

- システム上でラージ・ページが使用可能で、構成されている場合。
- **-O3** または **-qhot** などのループ最適化を使用可能にするオプションでコンパイルする場合。
- **-blpdata** オプションでリンクする場合。

ラージ・ページ・サポートの使用についての詳細は、AIX オペレーティング・システムの資料を参照してください。

事前定義マクロ

なし。

例

大容量のページ・ヒープを使用できるように `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 -qlargepage -blpdata
```

-qldbl128、-qlongdouble

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

-qnoldbl128

使用法

128 ビットの long double 型をサポートする別個のライブラリーが提供されます。**128** のサフィックス (**xlc128**、 **cc128**、 **xlc128_r**、または **cc128_r**) を付けて、いずれかの呼び出しコマンドを使用した場合は、これらのライブラリーが自動的にリンクされます。以下の表に示すように、 **-l***key* オプションを使用して、 128 ビット・バージョンのライブラリーを手操作でリンクすることもできます。

デフォルト (64 ビット) の long double		**128 ビット long double**	
ライブラリー	**-l*key* オプションの形式**	**ライブラリー**	**-l*key* オプションの形式**
libC.a	-lC	libC128.a	-lC128
libC_r.a	-lC_r	libC128_r.a	-lC128_r

プログラムが 128 ビットの long doubles を使用するとき (例えば、**-qldbl128** を単独で指定した場合) に、128 ビット・バージョンのライブラリーなしでリンクすると、予測できない結果になる場合があります。

#pragma options ディレクティブは、ソース・ファイルの最初の 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

- **-qldbl128** が有効な場合、 __LONGDOUBLE128 が 1 に定義されます。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 -lC128
```

関連情報

- 164 ページの『-l』

-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 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 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/aix51/、lib/aix52/、および lib/aix53/ 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

- 92 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

なし。

-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。

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

- 128 ページの『-g』
- 194 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

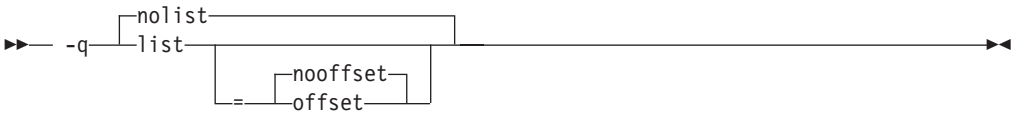
目的

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

- 178 ページの『-qlistopt』
- 211 ページの『-qprint』

- 237 ページの『-qsource』

-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

- 176 ページの『-qlist』
- 211 ページの『-qprint』

- 237 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・モードで暗黙の型の int を持つリテラルを long へプロモートする。

構文

デフォルト

-qnolonglit

使用法

下記の表は、デフォルトの定数の暗黙の型および **-qlonglit** が有効な場合の暗黙の型を示します。

サフィックス	10 進数のリテラル		16 進数または 8 進数のリテラル	
	デフォルトの暗黙の型	-qlonglit が有効な場合の暗黙の型	デフォルトの暗黙の型	-qlonglit が有効な場合の暗黙の型
サフィックスなしの場合	int long int	long int	int unsigned int long int unsigned long int	long int unsigned long int
u または U	unsigned int unsigned long int	unsigned long int	unsigned int unsigned long int	unsigned long int
l または L	long int	long int	long int unsigned long int	long int unsigned long int
u または U、および l または L の両方	unsigned long int	unsigned long int	unsigned long int	unsigned long int
ll または LL	long long int	long long int	long long int unsigned long long int	long long int unsigned long long int
u または U、および ll または LL の両方	unsigned long long int	unsigned long long int	unsigned long long int	unsigned long long int

事前定義マクロ

なし。

-qlonglong

カテゴリー

言語エレメント制御

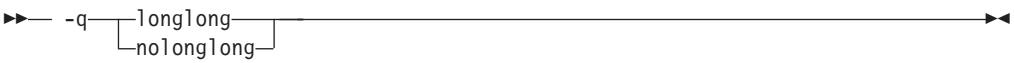
プラグマ同等物

#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

- **cc** 呼び出しコマンドあるいは **-qlanglvl=extended | extc89** オプションの場合は **-qlonglong**。**c89** 呼び出しコマンドあるいは **-qlanglvl=stdc89** オプションの場合は **-qnolonglong**。

使用法

このオプションが有効なのは、**cc** あるいは **c89** 呼び出しコマンドに対してのみです。または **-qlanglvl** オプションが **extended | stdc89 | extc89** に設定されている場合のみです。このオプションは、**xlc** 呼び出しコマンドに対して無効であり、または言語レベル **stdc99 | extc99** が有効になっている場合は無効です。このオプションで提供される long long サポートは、C99 標準が命令した long long 型のセマンティクスと互換性がないからです。

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
```

AIX v4.2 以降は、2 ギガバイトよりも大きいサイズのファイルをサポートしており、大容量のデータを単一ファイルで保管することができます。ユーザーのアプリ

ケーションで大容量ファイルを操作できるようにするには、 -D_LARGE_FILES および **-qlonglong** コンパイラー・オプションを指定してコンパイルします。例を以下に示します。

```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

関連情報

- 「*XL C ランゲージ・リファレンス*」の『整数型』

-ma

68 ページの『-qalloca、-ma』を参照してください。

-qmacpstr

カテゴリー

言語エレメント制御

プラグマ同等物

#pragma options [no]macpstr

目的

(プレフィックスに ¥p エスケープ・シーケンスが付いた) Pascal ストリング・リテラルを先頭バイトにストリングの長さが含まれるヌル終了ストリングに変換する。

例えば、**-qmacpstr** オプションが有効なとき、コンパイラーが以下を変換すると、

```
"¥pABC"
```

以下のようになります。

```
'¥03' , 'A' , 'B' , 'C' , '¥0'
```

構文

デフォルト

-qnomacpstr

使用法

Pascal ストリング・リテラルには、必ず文字 "¥p が含まれます。ストリングの途中にある文字 ¥p は Pascal ストリング・リテラルを形成せず、この文字の*直前に* ″ (二重引用符) 文字がなければなりません。

以下の文字を入れる場合について考えます。

```
'¥p' , 'A' , 'B' , 'C' , '¥0'
```

これを文字配列に入れても、 Pascal ストリング・リテラルは形成されません。

Pascal ストリング・リテラルの処理は 1 バイト文字にしか有効でないため、**-qmbcs** または **-qdbcs** オプションが有効である場合は、コンパイラーは **-qmacpstr** オプションを無視します。

#pragma options のキーワード **macpstr** は、すべての C ソース・ステートメントの前のソース・ファイルの先頭でのみ有効です。ソース・ファイルの途中で使用すると、指定は無視され、コンパイラーによってエラー・メッセージが出されます。

ここでは、Pascal ストリング・リテラルの処理方法について説明します。

- ワイド・ストリングについては Pascal ストリング・リテラル処理は行われないため、**-qmacpstr** オプションを指定してワイド・ストリング・リテラルでエスケープ・シーケンス `¥p` を使用すると、警告メッセージが生成されてそのエスケープ・シーケンスは無視されます。
- Pascal ストリング・リテラルを通常のストリングに連結すると、非 Pascal ストリングが作成されます。 例えば、以下のストリング

  ```
  "ABC" "¥pDEF"
  ```

  を連結すると、以下のようになります。

  ```
  "ABCpDEF"
  ```

- 2 つの Pascal ストリング・リテラルを連結しても (`strcat` など)、結果は 1 つの Pascal ストリング・リテラルにはなりません。 しかし、前述のとおり、 2 つの隣接する Pascal ストリング・リテラルを連結して、先頭バイトが新しいストリング・リテラルの長さである 1 つの Pascal ストリング・リテラルを形成することはできます。 例えば、以下のストリング

  ```
  "¥p ABC" "¥p DEF"
  ```

  または

  ```
  "¥p ABC" "DEF"
  ```

  を連結すると、以下のようになります。

  ```
  "¥06ABCDEF"
  ```

- Pascal ストリング・リテラルをワイド・ストリング・リテラルと連結することはできません。
- (長さバイトおよび終端のヌルを除いて) 255 バイトより長い Pascal ストリング・リテラルは、コンパイラーによって 255 文字に切り捨てられます。
- Pascal ストリング・リテラルは、他の C のストリング・リテラルと異なる基本型ではありません。 Pascal ストリング・リテラル処理の完了後は、結果として得られたストリングは、他のすべてのストリングと同様に扱われます。 Pascal ストリングを予期する関数にプログラムが C ストリングを渡した場合やその逆の場合、その振る舞いは未定義です。
- 処理の完了後に Pascal ストリング・リテラルの任意のバイトを変更しても、先頭バイトの元の長さの値は変更されません。例えば、ストリング `"¥06ABCDEF"` で、ストリングの真ん中でヌル文字を既存文字の 1 つに置換しても、ストリングの長さの含まれているストリングの先頭バイトの値は変わりません。
- 処理済みの Pascal ストリング・リテラルのバイトを変更しても、エラーや警告は出されません。

事前定義マクロ

なし。

例

mypascal.c をコンパイルして、ストリング・リテラルを Pascal 型ストリングに変換するには、以下のように入力します。

```
xlc mypascal.c -qmacpstr
```

関連情報

- 188 ページの『-qmbcs, -qdbcs』

-qmakedep、-M

カテゴリー

出力制御

プラグマ同等物

なし。

目的

make コマンドの記述ファイルの組み込みに適したターゲットを含む出力ファイルを作成する。

出力ファイルには .u サフィックスが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。記述ファイルには、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、または **-M** を指定すると、記述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

コマンド行で指定された .cまたは .i サフィックスを含む各ソース・ファイルに対して出力ファイルが生成され、それぞれにオブジェクト・ファイルと同じ名前が .u サ

フィックスとともに付けられます。その他の種類の入力ファイルのタイプに対しては、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、出力ファイルは **-o** オプションで指定した名前を使用します。以下の例を参照してください。

これらのオプションで生成した出力ファイルは **Make** 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、リンクされている必要があります。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*

以下のオプションを **qmakedep** および **-M** と一緒に使用することもできます。

-MF=*file_path*
出力ファイルの名前を設定します。ここで *file_path* は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。以下の例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、.u ファイルには追加されません。

include 文を持たないファイルは、入力ファイル名だけをリストした 1 行を含む出力ファイルを生成します。

事前定義マクロ

なし。

例

mysource.c をコンパイルして、出力ファイル mysource.u を作成するには、以下のように入力してください。

```
xlc -c -qmakedep mysource.c
```

foo_src.c をコンパイルして、出力ファイル mysource.u を作成するには、以下のように入力してください。

```
xlc -c -qmakedep foo_src.c -MF mysource.u
```

foo_src.c をコンパイルして、deps/ ディレクトリーに出力ファイル mysource.u を作成するには、以下のように入力してください。

```
xlc -c -qmakedep foo_src.c -MF deps/mysource.u
```

foo_src.c をコンパイルして、オブジェクト・ファイル foo_obj.o および出力ファイル foo_obj.u を作成するには、以下のように入力してください。

```
xlc -c -qmakedep foo_src.c -o foo_obj.o
```

foo_src.c をコンパイルして、オブジェクト・ファイル foo_obj.o および出力ファイル mysource.u を作成するには、以下のように入力してください。

```
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.u
```

foo_src1.c および foo_src2.c をコンパイルして、c:/tmp/ ディレクトリーに 2 つの出力ファイル foo_src1.u および foo_src2.u をそれぞれ作成するには、以下のように入力してください。

```
xlc -c -qmakedep foo_src1.c foo_src2.c -MF c:/tmp/
```

関連情報

- 189 ページの『-MF』
- 192 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

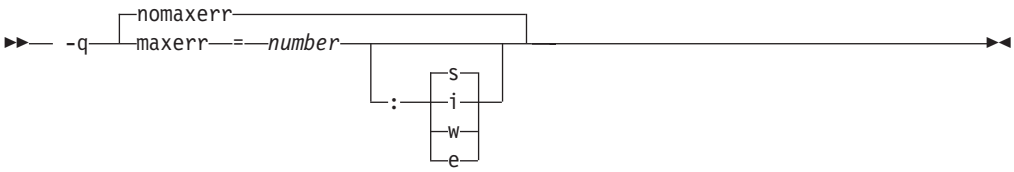
なし。

目的

指定したレベル以上の重大度レベルのエラーの件数が指定した件数に達すると、コンパイルを停止する。

構文

-qmaxerr 構文 — C



デフォルト

-qnomaxerr: コンパイラーは、コード生成が不可能になるまで、入力をできる限り処理し続けます。

パラメーター

number
1 以上の整数値でなければなりません。 エラーの数が指定した制限に達すると、回復不能エラーが起こり、コンパイルが停止します。

i 通知 (I) の最小の重大度レベルを指定します。

w 警告 (W) の最小の重大度レベルを指定します。

e　エラー (E) の最小の重大度レベルを指定します。

s　重大エラー (S) の最小の重大度レベルを指定します。

重大度レベルなしで **-qmaxerr** を指定し、**-qhalt** オプションまたはプラグマが有効な場合、**halt** で指定された重大度レベルが使用されます。重大度レベルなしで **-qmaxerr** を指定し、**halt** が有効でない場合、デフォルトの重大度レベルは **s** です。

使用法

-qmaxerr オプションを複数回指定した場合は、最後に指定した **-qmaxerr** オプションによって、オプションの処理が決まります。**-qmaxerr** と **-qhalt** オプションの両方を指定した場合は、最後に指定された **-qmaxerr** または **-qhalt** オプションが、**-qmaxerr** オプションで使用される重大度レベルを決定します。

診断メッセージは、**-qflag** オプションで制御できます。

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

- 114 ページの『-qflag』
- 131 ページの『-qhalt』
- 20 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

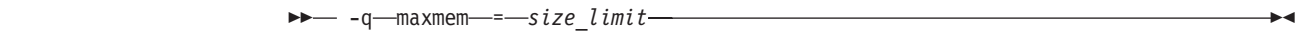
プラグマ同等物

#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。
- **-O3** より高い最適化が有効な場合は **-qmaxmem=-1** です。

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。制限は、コンパイラー全体ではなく、特定の最適化に使用するメモリーの量です。 コンパイル処理全体で必要となるテーブルは、この制限の影響を受けないため、この制限には含まれません。

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 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

- 94 ページの『-D』

-MF

カテゴリー

出力制御

プラグマ同等物

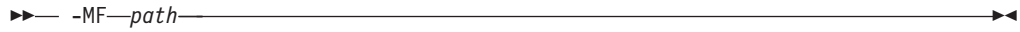
なし。

目的

-qmakedep または **-M** オプションによって生成される出力のターゲットを指定する。

このオプションと併用できるのは、**-qmakedep** オプションまたは **-M** オプションのみです。詳しくは、 183 ページの『-qmakedep、-M』の説明を参照してください。

構文



デフォルト

適用されません。

パラメーター

path

ターゲット出力のパスです。*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. *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、コマンド行で最後に指定されたファイルの **make** 規則を含む単一の従属関係ファイルのみが生成されます。

事前定義マクロ

なし。

関連情報

- 183 ページの『-qmakedep、-M』
- 192 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

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・ファイルから共用オブジェクトを作成する。

このオプションを、リンカーを直接呼び出す (または C++ で **makeC++SharedLib** ユーティリティーを使用する) 代わりに下記の関連オプションと共に使用すると、共用オブジェクトを作成できます。 このオプションを使用する利点は **-qipa** リンク時の最適化 (**-O5** で実行されたものなど) との互換性を持たせることができることです。

構文

-qmkshrobj 構文 — C

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

コンパイラーは、**-bE**:、**-bexport**:、または **-bnoexpall** オプションによってエクスポートするシンボルを明示的に指定しない場合、または **-qnoweakexp** オプションを使用して弱いシンボルのエクスポートを回避すると、共用オブジェクトからすべてのグローバル・シンボルを自動的にエクスポートします。

-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*
共用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 shr.oです。

-qexpfile=*filename*
filename にエクスポートされたシンボルをすべて保管します。

-e *name*
共用実行可能ファイルの入り口名を *name* に設定します。デフォルトは **-enoentry** です。

-q[no]weakexp
弱いとマークされたシンボル (**#pragma weak** ディレクティブを使用する) をエクスポート・リストに組み込むかどうかを指定します。このオプションを明示的に設定しない場合、デフォルトは **-qweakexp** (グローバルな弱いシンボルがエクスポートされる) です。

-qmkshrobj を使用した共用ライブラリーの作成方法の詳細については、「*XL 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』を参照してください。

事前定義マクロ

なし。

例

3 つの小さなオブジェクト・ファイルから共用ライブラリー `big_lib.so` を構成するには、以下のように入力します。

```
xlc -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

関連情報

- 『-o』
- 102 ページの『-e』
- 208 ページの『-qpic』
- 78 ページの『-b』
- 110 ページの『-qexpfile』
- 279 ページの『-qweakexp』

-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、4 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 のソース・ファイル・サフィックス (、.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

- 83 ページの『-c』
- 103 ページの『-E』
- 200 ページの『-P』
- 252 ページの『-qsyntaxonly』

-O、-qoptimize

カテゴリー

最適化およびチューニング

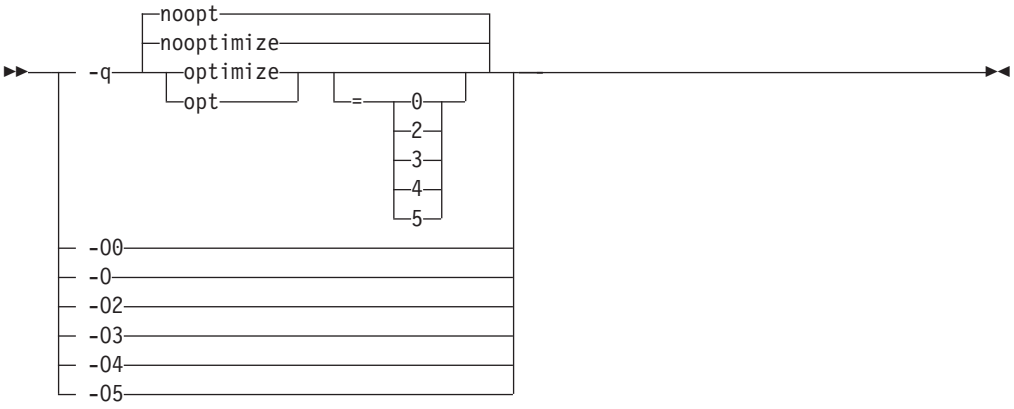
プラグマ同等物

#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。
   - ループを通るすべてのパスで発生するわけではない。

   -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=fltint: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。

積極的な最適化には、浮動小数点サブオプション **-qfloat=hsflt | hssngl**、またはプログラムの精度モードに影響を与える他のすべてのオプションは含まれません。

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 **-qfltrap** 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、 121 ページの『-qfltrap』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。

-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。

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

- 133 ページの『-qhot』
- 150 ページの『-qipa』
- 203 ページの『-qpdf1、-qpdf2』
- 244 ページの『-qstrict』
- 「*XL 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**、**-qipa**、あるいは **-qpdf** 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。

-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 最適化およびプログラミング・ガイド*」の 『最適化されたプログラムのデバッグのための -qoptdebug の使用』を参照してください。

関連情報

- 194 ページの『-O、-qoptimize』
- 133 ページの『-qhot』
- 150 ページの『-qipa』
- 203 ページの『-qpdf1、-qpdf2』
- 233 ページの『-qsmp』

- 128 ページの『-g』
- 175 ページの『-qlinedebug』

-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。**-p** で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 mon.out ファイルに書き込まれます。**-pg** でコンパイルされたプログラムは gmon.out ファイルを書き込みます。**prof** または **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 -cxlc
 myprogram.o -p -o program
```

関連情報

- 255 ページの『-qtbtable』
- **prof** および **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は入力ファイルと同じ名前で、サフィックス .i が付いています。

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

-P オプションは、.i サフィックスが付いていないファイル名を受け入れます。そうでない場合、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るため、エラー・メッセージは生成されません。

-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

- 84 ページの『-C、-C!』
- 103 ページの『-E』
- 209 ページの『-qppline』
- 252 ページの『-qsyntaxonly』

-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

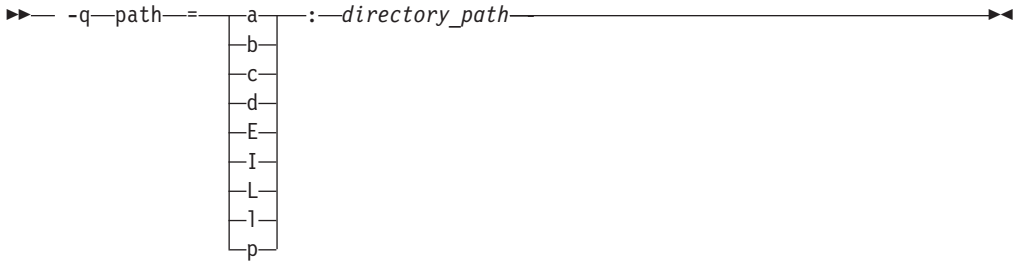
なし。

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 実行可能ファイルの代替パス名を判別する。

XL 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。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実行可能ファイル名との対応を示しています。

パラメーター	説明	実行可能ファイル名
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
d	逆アセンブラー	dis
E	**CreateExportList** ユーティリティー	CreateExportList
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	リンカー	ld
p	プリプロセッサー	該当なし

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。

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ので注意してください。

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

- 79 ページの『-B』
- 112 ページの『-F』
- 253 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。

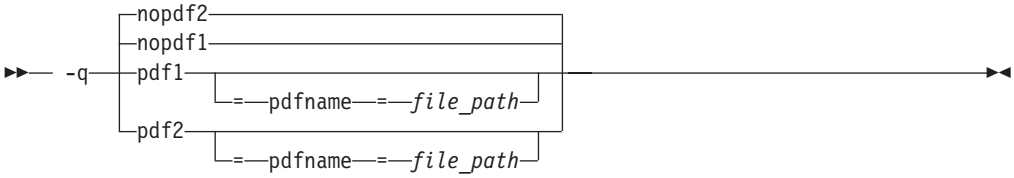
目的

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。

PDF は、2 つのステップのプロセスがあります。最初に、**-qpdf1** および 最小最適化レベル **-O2** を指定して、リンクしながらアプリケーションをコンパイルします。次に、その結果生じたアプリケーションを一般的なデータ・セットと一緒に実行します。テスト実行中に、プロファイル・データが、プロファイル・ファイルに書き込まれます (デフォルトでは、このファイルは ._pdf と名付けられ、現行作業ディレクトリー、あるいは PDFDIR 環境変数が設定されていればその環境変数が指定したディレクトリーに保存されます)。次に、**-qpdf2** および最小最適化レベル **-O2** を指定して、アプリケーションを再コンパイル、またはリンク (再リンク)、あるいはその両方を行います。これにより、プログラム実行中に収集されたプロファイル・データに従って適用される最適化を適切に調整します。

PDF は、他のデバッグと調整が済んだ後、アプリケーションを実動させる前の最終ステップの 1 つとして使用されるよう設計されています。

構文



デフォルト

-qnopdf1、-qnopdf2

パラメーター

pdfname= *file_path*

プロファイル・データを保持するファイルへのパスを指定します。デフォルトでは、そのファイル名は ._pdf であり、現行作業ディレクトリーまたは PDFDIR 環境変数で指定されたディレクトリーに配置されます。**pdfname** サブオプションを使用して、同時に複数の実行可能ファイルを同じ PDF ディレクトリーを使用して稼働することができます。こうすると、動的ライブラリーの PDF を使用したチューニングに特に有効です。

使用法

実行時にプロファイル情報を収集するには、主プログラムを PDF を使用してコンパイルしなければなりません。

PDF1 コンパイルで使用した場合と同じコンパイル・オプションを PDF2 コンパイルでも使用する必要があります。

最適化されたオブジェクト・ファイルが、2 番目のステップで再リンクされないようにするには、**-qpdf2 -qnoipa** を指定します。ただし、**-qpdf1** を指定して既にコンパイルされたソース・ファイルを変更する場合は、最初の受け渡しプロセス全体を再度行います。

プロファイル・ファイルの代替パスおよびファイル名を指定する場合は、**pdfname** サブオプションを使用してください。また、PDFDIR 環境変数を使用して、ディレクトリーの絶対パス名を指定することができます。プロファイル情報のセットを識別するために **pdfname** サブオプションを使用していない限り、同時に同じプロファイル・ディレクトリーを使用する 2 つの別々のアプリケーションをコンパイルしたり、実行しないでください。例えば、「*XL 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。

-qpdf1 を指定して、以下のオプションを使用することもできます。

-qshowpdf
ブロックおよび関数呼び出し数などの追加情報がプロファイル・ファイルに提供されます。詳しくは、231 ページの『-qshowpdf』を参照してください。

PDF の使用に関する推奨手順については、「*XL C 最適化およびプログラミング・ガイド*」の『Profile-Directed Feedback の使用』を参照してください。

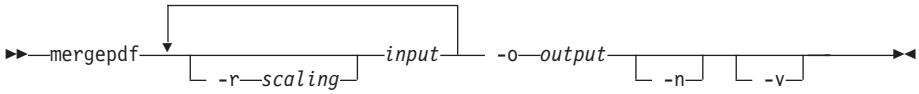
以下のユーティリティー・プログラムは、/usr/vac/bin/ にあり、プロファイル・データの書き込み先のディレクトリーを管理するのに有効です。

cleanpdf

directory_path が指定したディレクトリーからすべてのプロファイル情報を削除します。または、*pathname* が指定されていない場合は、PDFDIR 環境変数が設定したディレクトリーから現行ディレクトリーから削除されます。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。

特定のアプリケーションの PDF プロセスが終了したときだけに **cleanpdf** を実行してください。そうでないと、そのアプリケーションで PDF の使用を再開したい場合、**-qpdf1** を使用してすべてのファイルをもう一度再コンパイルする必要が生じます。

mergepdf



複数の PDF レコードを単一の PDF 出力レコードにマージします。

-r *scaling*
PDF レコード・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 入力レコード・ファイルの名前、または PDF レコード・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 出力レコード・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。

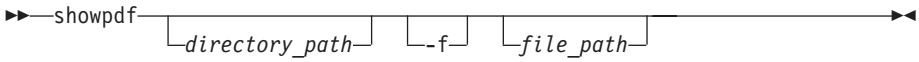
-n これを指定すると、PDF レコード・ファイルは正規化されません。指定されていない場合は、ユーザー定義の位取り係数を適用する前に、内部で計算された比率を基に **mergepdf** がレコードを正規化します。

-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

前述の **cleanpdf** と同じ。

showpdf



プログラム実行中に、プロファイル・ファイルに書き込まれ、**-f** オプションによって指定された関数呼び出しおよびブロック数を表示します。このコマンドを使用するには、まず コマンド行に **-qpdf1** と **-qshowpdf** の両方のコンパイラー・オプションを指定してアプリケーションをコンパイルする必要があります。

事前定義マクロ

なし。

例

簡単な例を以下に示します。

```
// Compile all 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 files with -qpdf2.
xlc -qpdf2 -O3 file1.c file2.c file3.c

// The program should now run faster than
// without PDF if the sample data is typical.
```

もう少し複雑な例を以下に示します。

```
// Set the PDFDIR variable.
export PDFDIR=$HOME/project_dir

// Compile most of the files with -qpdf1.
xlc -qpdf1 -O3 -c file1.c file2.c file3.c

// This file is not so important to optimize.
xlc -c file4.c

// Non-PDF object files such as file4.o can be linked in.
xlc -qpdf1 -O3 file1.o file2.o file3.o file4.o

// Run several times with different input data.
./a.out < polar_orbit.data
./a.out < elliptical_orbit.data
./a.out < geosynchronous_orbit.data

// No need to recompile the source of non-PDF object files (file4.c).
xlc -qpdf2 -O3 file1.c file2.c file3.c

// Link all the object files into the final application.   */
xlc -qpdf2 -O3 file1.o file2.o file3.o file4.o
```

-qpdf2 を指定してソースの再コンパイルを迂回する例を次に示します。

```
// Compile source with -qpdf1.
xlc -O3 -qpdf1 -c file.c

// Link in object file.
xlc -O3 -qpdf1 file.o

// Run with one set of input data.
./a.out < sample.data

// Link in object file from qpdf1 pass.
// (Bypass source recompilation with -qpdf2.)
  xlc -O3 -qpdf2 file.o
```

pdf1 オブジェクトおよび pdf2 オブジェクトの使用の例を次に挙げます。

```
// Compile source with -qpdf1.
xlc -c -qpdf1 -O3 file1.c file2.c

// Link in object files.
xlc -qpdf1 -O3 file1.o file2.o

// Run with one set of input data.
./a.out < sample.data

// Link in the mix of pdf1 and pdf2 objects.
  xlc -qpdf2 -O3 file1.o file2.o
```

ここに、実行可能ファイルに再リンクしないで、PDF の最適化されたオブジェクト・ファイルを作成する例を示します。

```
// Compile source with -qpdf1.
xlc -c -O3 -qpdf1 file1.c file2.c file3.c

// Link in object files.
xlc -O3 -qpdf1 file1.o file2.o file3.o

// Run with one set of input data.
./a.out < sample data

// Recompile the instrumented source files
xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
```

関連情報

- 231 ページの『-qshowpdf』
- 150 ページの『-qipa』
- 「*XL 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』

-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* はコンパイラー時間と 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。

事前定義マクロ

なし。

例

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表示されます。

```
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
```

-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

共用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

- -qpic=small
- **-G** または **-qmkshrobj** コンパイラー・オプションが指定された場合の **-qpic=small**。

パラメーター

small
目次のサイズが 64 Kb 以下であると想定するようにコンパイラーに命令します。

large
サイズが 64 Kb よりも大きい目次を許可します。これにより、テーブルにより多くのアドレスを保管できます。通常、このオプションを指定して生成されたコードは、**-qpic=small** を指定して生成されたコードよりも大きくなります。

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

-qpic=large を指定すると、**-bbigtoc** を **ld** に渡す場合と同じ効果があります。

事前定義マクロ

なし。

例

共用ライブラリー `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

- 61 ページの『-q32、-q64』
- 129 ページの『-G』
- 191 ページの『-qmkshrobj』

-qppline

カテゴリー

オブジェクト・コード制御

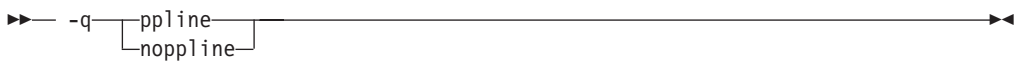
プラグマ同等物

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

- 103 ページの『-E』
- 200 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

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阻止しません。

事前定義マクロ

なし。

-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
```

-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

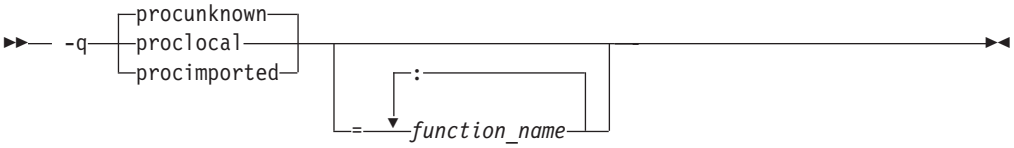
64 ビットのコンパイルで、関数にローカル、インポート、または不明のマークを付ける。

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。**proclocal** 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、ライブラリーの共用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。 **procimported** オプションまたはプラグマを使用して、インポートされた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。 **procunkown** 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

-qprocunknown: 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
指定されたオプションに応じて、ローカル、インポート、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。*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。

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用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用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

- 96 ページの『-qdataimported、-qdatalocal、-qtocdata』

-qproto

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、このオプションのみが有効です。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
```

-Q、-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。

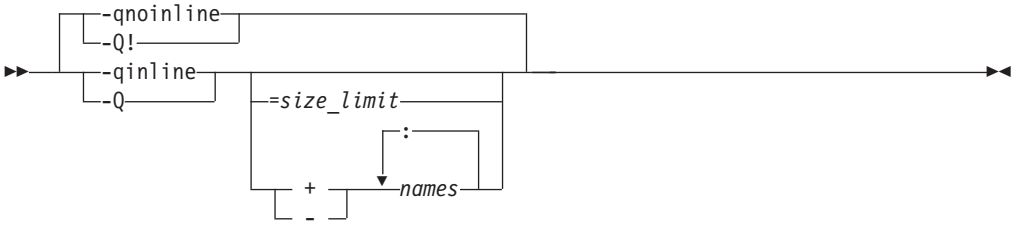
-Q (または **-qinline**) と一緒に、最小の最適化レベルである **-O** を指定して、関数プロシージャー (inline 指定子によって宣言された関数など) のインライン化を使用可能にする必要があります。また、**-Q** オプションを使用してインライン化すべき、またはすべきでない関数に制限を指定することもできます。

-Q(または **-qinline**) が有効なすべての場合において、コンパイラーはヒューリスティックを使用して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に効果が出る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、結果として生じるインライン化された呼び出しの数およびコード・サイズの増加量への制限によって異なります。そのため、インライン化を使用可能にすることのみでは、特定の関数がインライン化されることを保証することにはなりません。

-Q! (または **-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 inline として明示的に宣言された関数を使用不可にします。

構文

-qinline および -Q 構文 — C



デフォルト

-qnoinline または -Q!

パラメーター

size_limit
関数内の実行可能ステートメントの数を表す正整数。以下の例で分かるように、宣言はカウントされません。

```
increment()
{
  int a, b, i;
   for (i=0; i<10; i++) /* statement 1 */
   {
      a=i;              /* statement 2 */
      b=i;              /* statement 3 */
   }
}
```

インライン化を考慮する場合、関数内の実行可能ステートメントの数は、*size_limit* 以下である必要があります。値を **0** に指定すると、*name* サブオプションにリストされていない関数または `inline` 関数指定子のサポートされた形式でマークを付けられていない関数はインライン化されません。*size* を指定しない場合のデフォルト値は 20 です。

+ コンパイラーは、*name* がリストする関数のみでなく、 *size* が指定する基準を満たすすべての関数のインライン化を試行します。

- コンパイラーは、*name* がリストする関数以外で、 *size* が指定する基準を満たすすべての関数のインライン化を試行します。

name
インライン化される関数の名前。それぞれの関数名は、コロン **(:)** で分離します。 このサブオプションは、*size* 値の設定をすべてオーバーライドします。このサブオプションは、`inline` 指定子によって明示的に宣言された関数に影響は与えません。 **-O** および **-Q** | **-qinline** が有効な場合は、これらの関数は*常に*インライン化に考慮されます。このサブオプションを、オプションの正または負の形式に対する引数として指定して、最もインライン化されやすい関数を正確に制御します。

コンパイル中のソース・ファイルに定義されていない関数については、警告メッセージが出されます。

使用法

インライン化を最大化する場合は、最適化 **(-O)** に加え、適切な **-qinline** オプションまたは **-Q** オプションを指定します。

インライン化は必ずしもランタイム・パフォーマンスを改善するわけではないので、ユーザー自身のコードでこのオプションの効果をテストしてください。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。

-g オプションを指定してデバッグ情報を生成する場合は、インライン化が抑制される可能性があります。

事前定義マクロ

なし。

例

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、以下のように入力します。

```
xlc myprogram.c -O -qnoinline
```

12 未満のステートメントを持つすべての関数のインライン化がコンパイラーによって試行されるように `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 -qinline=12
```

関数 `salary`、`taxes`、`expenses`、および `benefits` にはそれぞれ 21 以上の実行可能ステートメントがある場合、コンパイラーがすべての適切な関数 (つまり、ステ

ートメントの数がデフォルトの 20 に満たない関数) およびこれらの関数のインライン化を試行するように 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 -qinline+salary:taxes:expenses:benefits
```

関数 salary、taxes、expenses、および benefits にはそれぞれ 20 未満の実行可能ステートメントがある場合、コンパイラーが、これらの関数*以外*のすべての適切な関数 (つまり、ステートメントの数がデフォルトの 20 に満たない関数) のインライン化を試行するように 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 -qinline-salary:taxes:expenses:benefits
```

関数名と一緒に、ゼロのサイズ値を使用して、特定の関数をインライン化することができます。例を以下に示します。

```
-O -qinline=0
```

これに以下を続けて指定します。

```
-qinline+salary:taxes:benefits
```

これによって、salary、taxes、または benefits という名前の関数のみ が、可能な場合にインライン化され、それ以外はインライン化されません。

関連情報

- 128 ページの『-g』
- 150 ページの『-qipa』
- 194 ページの『-O、-qoptimize』
- 「*XL C ランゲージ・リファレンス*」の『インライン関数指定子』

-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

ファイルに未解決のシンボルが含まれていても、再配置可能なオブジェクトを作成する。

構文

デフォルト

適用されません。

使用法

このフラグを指定して作成されたファイルは、別のコンパイラー呼び出しのファイル・パラメーターとして使用されることが期待されています。

事前定義マクロ

なし。

例

myprogram.c および myprog2.c をコンパイルして単一のオブジェクト・ファイル mytest.o を作成するには、以下のように入力します。

```
xlc myprogram.c myprog2.c -r -o mytest.o
```

-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を使用して、リスト・ファイルは、疑似 C コード・リスト、およびプログラム・ループがどのように並列化および / または最適化されるかの概要を示します。また、レポートには、特定のループが並列化および / またはベクトル化されない理由を示す診断情報が含まれます。例えば、**-qhot=simd** および **-qenablevmx** と一緒に使用すると、ループのベクトル化を回避する可能性のあるストライド 1 でない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を含むループを配布することができます。 POWER4 と POWER5 はロード・ストリームのプリフェッチをサポートし、POWER6 はロード・ストリームとストア・ストリームの両方のプリフェッチをサポートします。

-qipa=clonearch と一緒に使用すると、**-qreport** は、このオプションによって指定されるアーキテクチャーでクローン作成されるプロシージャーについての変換レポートを作成します。

構文

デフォルト

-qnoreport

使用法

また、**-qreport** が、ループ変換リストを作成するには、コマンド行で以下のオプションのいずれかを指定してください。

- **-qhot[=simd]**
- **-qsmp**
- **-O5**
- **-qipa=level=2**

また、**-qreport** が、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、コマンド行で以下のオプションのいずれかを指定してください。

- **-qsmp**
- **-O5**
- **-qipa=level=2**

また、**-qreport** が、機能のクローン作成リストを生成するようにするには、**-qipa=clonearch** を指定してください。

-qreport を **-O5** または **-qipa=level=2** と一緒に使用する場合は、リンク・ステップの後にレポートが生成されます。

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

- 133 ページの『-qhot』
- 150 ページの『-qipa』
- 233 ページの『-qsmp』
- 198 ページの『-qoptdebug』
- 「*XL C 最適化およびプログラミング・ガイド* 」の 『最適化されたプログラムのデバッグのための -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。以下は、有効なレジスターです。

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 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』

-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

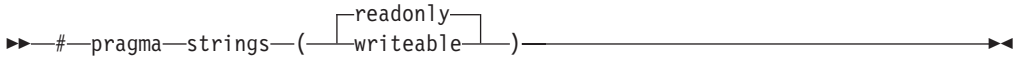
ストリング・リテラルのストレージ・タイプを指定する。

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

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。

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

- 221 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

- **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**。

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、共用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

- 221 ページの『-qro』
- 『-qroptr』

-qroptr

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

定数ポインターのストレージ・ロケーションを指定する。

-qroptr が有効な場合は、定数ポインターを、読み取り専用ストレージに入れます。**-qnoroptr** が有効な場合は、ポインターを読み取り/書き込みストレージに入れます。

構文

デフォルト

-qnoroptr

使用法

定数ポインターはアドレス定数と同等です。例を以下に示します。

```
int* const p = &n;
```

-qnoroptr が有効な場合は、定数ポインターの値をエラーを生成することなく変更することができます。

-qroptr により、実行時パフォーマンスは改善され、ストレージが節約され、共用アクセスが提供されますが、読み取り専用の定数値を変更しようとしたコードはメモリー・エラーを生成します。例えば、以下のコードを想定します。このコードは `c1_ptr` が指すアドレスの変更を試行します。

```
char c1 = 10;
char c2 = 20;
char* const c1_ptr = &c1;

int main() {
    *(char**)&c1_ptr = &c2;
}
```

-qroptr オプションを指定してこのコードをコンパイルすると、実行時にセグメンテーション障害が発生します。

共用ライブラリーの一部となるコンパイル済みコードには、**-qroptr** を使用しないでください。

事前定義マクロ

なし。

関連情報

- 221 ページの『-qro』
- 222 ページの『-qroconst』

-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は制限されます。

事前定義マクロ

なし。

関連情報

- 128 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

- 103 ページの『-E』
- 200 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、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

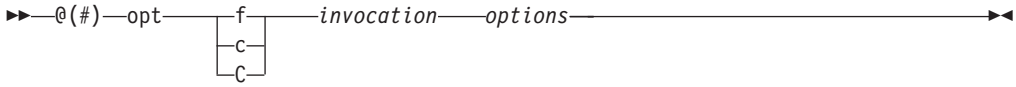
デフォルト

-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。

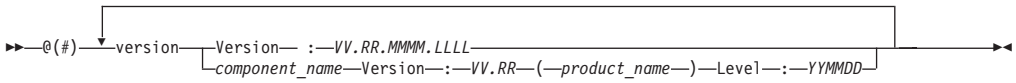
c C 言語のコンパイルを指定します。

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。

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 (PTF)。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。

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**what** 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 /usr/vac/bin/xlc t.f -c -qsaveopt -qhot
version IBM XL C for AIX, V10.1
version Version: 10.01.0000.0001
version Driver Version: 10.01(C) Level: YYMMDD
version Front End  Version: 10.01(C)  Level: YYMMDD
version C Front End Version : 10.01(C)  Level: YYMMDD
version High Level Optimizer Version: 10.01(C) and 12.01(Fortran)  Level: YYMMDD
version Low Level Optimizer  Version: 10.01(C) and 12.01(Fortran) Level: YYMMDD
```

1 行目で、c は、C として使用されたソースを識別し、/usr/vac/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 (PTF) によって異なる場合があります。

関連情報

- 274 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

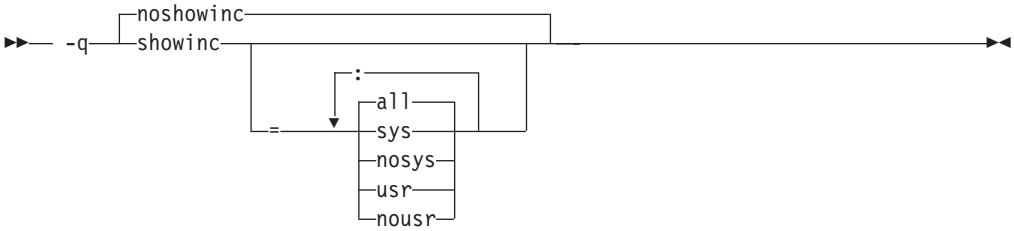
プラグマ同等物

#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

- 237 ページの『-qsource』

-qshowmacros

カテゴリー

43 ページの『出力制御』

プラグマ同等物

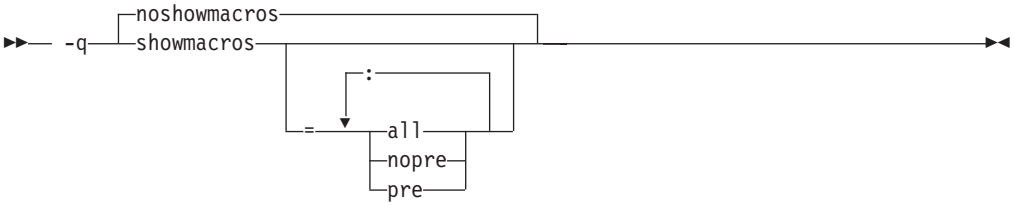
なし

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

- 103 ページの『-E』
- 200 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・ステップおよびリンク・ステップで、**-qpdf1** と最小の最適化レベル **-O2** と共に指定された場合、追加のプロファイル情報をコンパイルされたアプリケーションに挿入し、そのアプリケーション内のすべてのプロシージャーの呼び出しとブロックのカウントを収集する。

構文

使用法

トレーニング・データを使用してアプリケーションを実行した後は、呼び出しおよびブロック数がプロファイル・ファイル (デフォルトは named ._pdf) で記録されます。 203 ページの『-qpdf1、-qpdf2』で説明されている **showpdf** ユーティリティーを使用してプロファイル・ファイルの内容を検索できます。

-qshowdpf および **showpdf** を使用するプロシージャーと例については、「*XL 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。

関連情報

- 203 ページの『-qpdf1、-qpdf2』
- 「*XL 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』

-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

AIX では、スタック・サイズが 256 MB までに制限されています。 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

- 128 ページの『-g』
- 150 ページの『-qipa』

-qsmp

カテゴリー

最適化およびチューニング

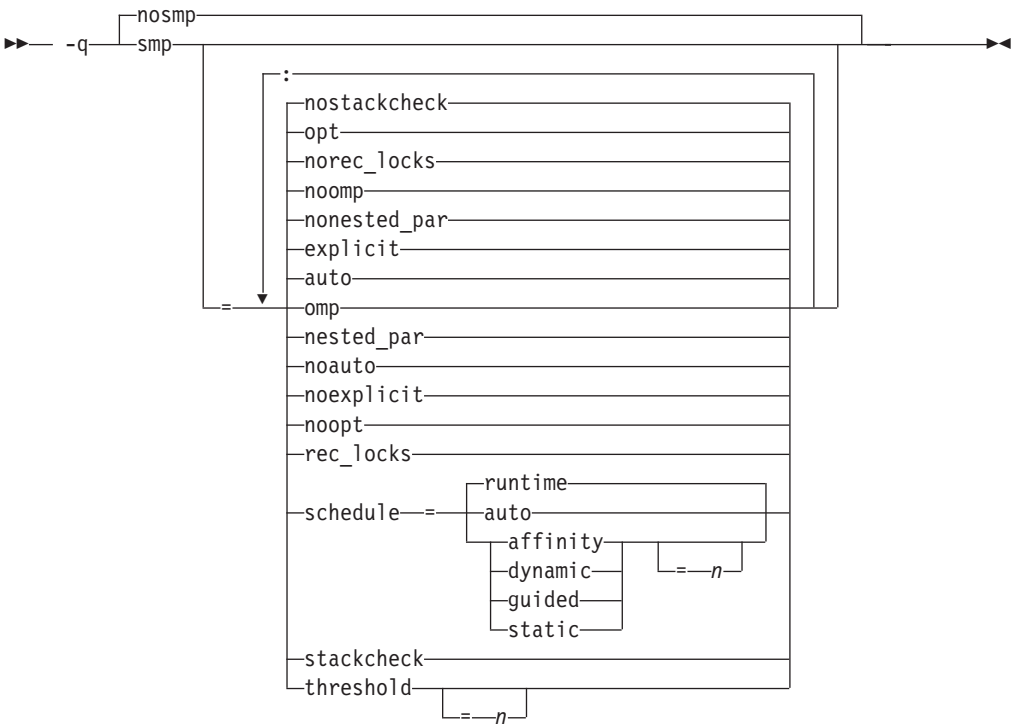
プラグマ同等物

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、SMP または 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文は並列化します。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。 **nested_par** は真のネストされた並列性を提供しません。その理由は、これを使用してもネストされた並列領域に対して新規のスレッドのチームが作成されないためです。代わりに、現在使用可能なスレッドが再利用されます。

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。

nested_par サブオプションを設定しても、これは OpenMP API に従っているわけではないことに注意してください。このサブオプションを指定する場合、ランタイム・ライブラリーは、 構文を囲むのに使用した場合と同じスレッドを、ネストされた 構文にも使用します。

omp | noomp
OpenMP 標準への厳しい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。

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。

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*) 繰り返しを含む *n* 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* 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) ループの繰り返しを含みます。

スレッドが解放されると、スレッドが最初に割り当てられた区画から次のチャンクを取得します。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。

アフィニティー・スケジューリングのタイプは、OpenMP API 標準には表示されません。

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。チャンク・サイズ (*n*) は、**auto** を使用する場合は指定してはなりません。チャンク・サイズ (*n*) が指定された場合、コンパイラーは重大エラー・メッセージを発行します。 **-qsmp=schedule** オプションと *OMP_SCHEDULE* を両方とも使用すると、このオプションがこの環境変数をオーバーライドするので注意してください。

dynamic[=*n*]
ループの繰り返しが、それぞれ *n* 繰り返しを含むチャンクに分割されます。 *n* が指定されていないと、チャンクは **ceiling**(*number_of_iterations*/*number_of_threads*) 繰り返しを含みます。

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、すべての作業に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。

スレッドがスリープ状態にある場合、それに割り当てられた作業は、別のアクティブ・スレッドが使用可能になったら即座に引き継がれます。

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、必然的に、スケジューリング・タイプが*巡回スケジューリング*として参照されます。

n を指定しない場合、チャンクには **ceiling**(*number_of_iterations*/*number_of_threads*) 回の繰り返しが含まれます。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=runtime** 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、デバッグ用に使用します。
XLSMPOPTS=stackcheck も設定されているときのみ有効です。詳細については、「*XL C 最適化およびプログラミング・ガイド*」の 28 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=runtime:nostackcheck:threshold=100
```

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。を参照してください。**-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:ibm** 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。

事前定義マクロ

-qsmp が有効なとき、_IBMSMP は値 1 に事前定義されており、これは、IBM SMP ディレクティブが認識されていることを示します。そうでない場合は、定義されません。

関連情報

- 194 ページの『-O、-qoptimize』
- 257 ページの『-qthreaded』
- 28 ページの『並列処理のための環境変数』
- 327 ページの『並列処理のプラグマ・ディレクティブ』
- 415 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。

source 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 a .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

-qnosource

使用法

#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` 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
```

関連情報

- 176 ページの『-qlist』
- 178 ページの『-qlistopt』
- 211 ページの『-qprint』

-qsourcetype

カテゴリー

入力制御

プラグマ同等物

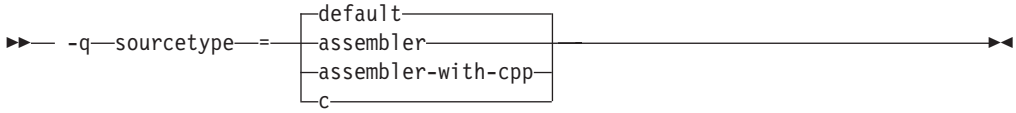
なし。

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。

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++ コンパイラーが使用可能であると想定し、C++ ファイルとしてコンパイルされます。

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
```

-qspeculateabsolutes

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

非 IPA リンクの -qtocmerge -bl:file および IPA リンクの -bl:file と連動して絶対アドレスでの投機的実行を使用不可にする。

コンパイラーがどのアドレスが絶対であるかを知るには、bl: ファイルが必要です。

構文

デフォルト

-qspeculateabsolutes

事前定義マクロ

なし。

関連情報

- 260 ページの『-qtocmerge』

-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

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。

例

エラー発生時に診断メッセージと共にソース行を表示するように `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 -qsrcmsg
```

-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: 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。

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 -qstatsym
```

-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。

- 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 (これは通常 /usr/vac/include/) または **-qc_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 (これは通常 /usr/include/)、または **-qc_stdinc** オプションまたは **-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

- 93 ページの『-qc_stdinc』
- 136 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*)")

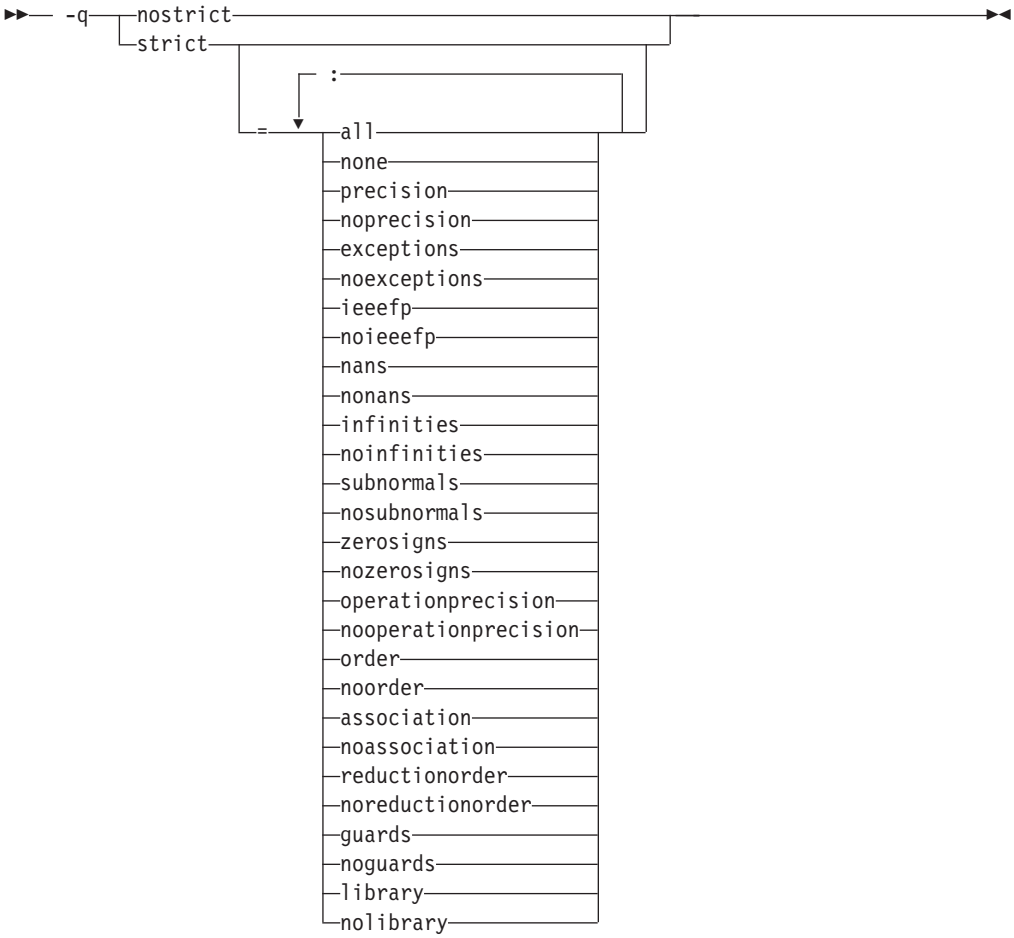
目的

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

オプション　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、常に **-qstrict** または **-qstrict=all** 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**、**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**、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 浮動小数点シグナリング 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。

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** ではそれらの縮約が有効になります。

guards | noguards
guards は、保護領域の境界を越えた (すなわち、**IF** 文**if**s を過ぎた、またはループ外) の移動操作、またはその操作を実行すべきかどうかを制御する呼び出しを無効にします。 では、それらの移動操作が有効になります。

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。

さまざまな **-qstrict[=subtopions]** または **-qnostrict** の組み合わせを指定すると、**-qfloat** サブオプションが次のように設定されます。

- **-qstrict** または **-qstrict=all** では **-qfloat=nofltint:norsqrt:rngchk** が設定されます。**-qnostrict** または **-qstrict=none** では **-qfloat=fltint: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。

これらの設定のいずれかをオーバーライドするには、コマンド行の **-q[no]strict[=suboption_list]** 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、範囲検査がオフにされ (**-qfloat=fltint**)、 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=fltint: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
- 116 ページの『-qfloat』
- 133 ページの『-qhot』
- 194 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
- 194 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

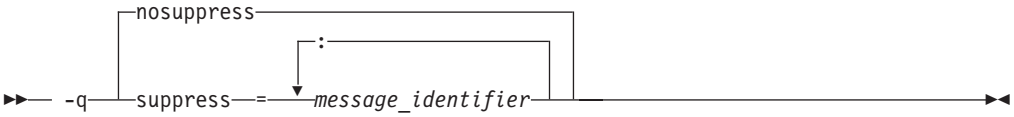
プラグマ同等物

なし。

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



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、

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。これらの説明については、 19 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。

-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。

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
```

関連情報

- 114 ページの『-qflag』

-qsymtab

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

- 128 ページの『-g』
- 97 ページの『-qdbxextra』
- 242 ページの『-qstatsym』

-qsyntaxonly

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

- 84 ページの『-C、-C!』
- 83 ページの『-c』
- 103 ページの『-E』
- 192 ページの『-o』
- 200 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実行可能ファイルのデフォルト・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

以下のテーブルは、**-t** パラメーターとコンポーネント実行可能ファイル名との間の対応を示しています。

パラメーター	説明	実行可能ファイル名
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
d	逆アセンブラー	dis
E	**CreateExportList** ユーティリティー	CreateExportList
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	リンカー	ld
p	プリプロセッサー	該当なし

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てください。 **-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。

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ので注意してください。

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

- 79 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

- 128 ページの『-g』

-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

- 233 ページの『-qsmp』

-qtimestamps

カテゴリー

43 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。

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

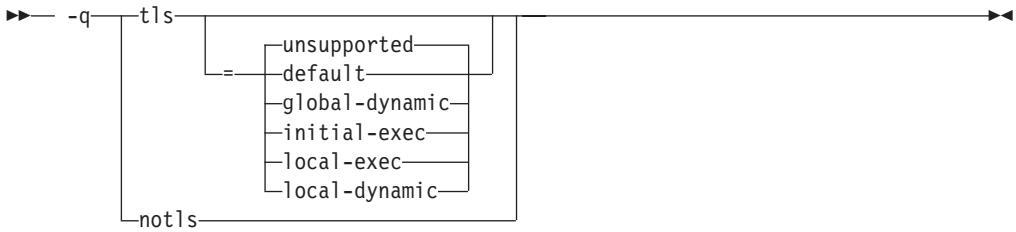
スレッド・ローカル・ストレージが割り振られる変数を指定する `__thread` ストレージ・クラス指定子の認識を使用可能にし、使用するスレッド・ローカル・ストレージ・モデルを指定する。

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。

注: このオプションがサポートされているのは、AIX for POWER バージョン 5.3 (5300-05 Technology Level) 以上のみです。

構文



デフォルト

-qtls=unsupported

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 コンパイラー・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** はデフォルトで有効です)(**-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

- 208 ページの『-qpic』
- 「*XL 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』

-qtocdata

96 ページの『-qdataimported、-qdatalocal、-qtocdata』を参照してください。

-qtocmerge

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

TOC マージを使用可能にして TOC ポインターのロードを削減し、外部ロードのスケジューリングを改善する。

構文

デフォルト

-qnotocmerge

使用法

-qtocmerge を使用するには、**-bImportfile** リンカー・オプションを使用してコンパイラーが読み取るファイル名を指定する必要もあります。

事前定義マクロ

なし。

-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* 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。

事前定義マクロ

なし。

関連情報

- 「*XL C* ランゲージ・リファレンス」の 『3 文字表記』
- 99 ページの『-qdigraph』
- 166 ページの『-qlanglvl』

-qtune

カテゴリー

最適化およびチューニング

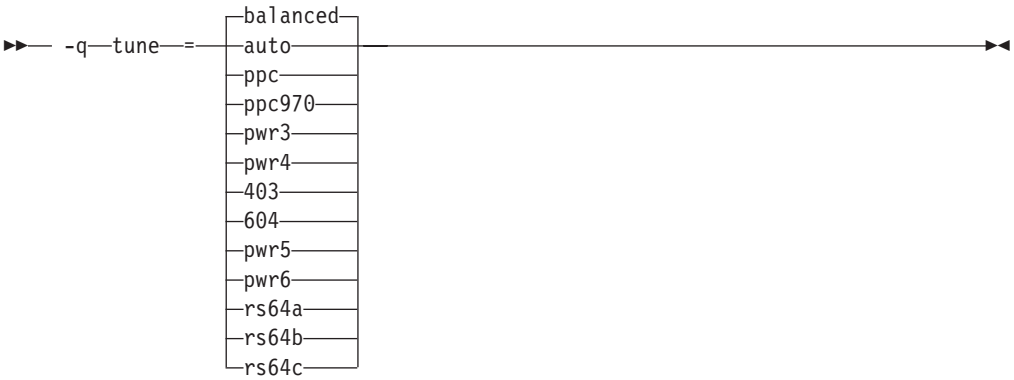
プラグマ同等物

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。

構文



デフォルト

-qtune=balanced (デフォルト **-qarch** 設定が有効な場合)。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、 263 ページの表 23 を参照してください。

パラメーター

403
PowerPC 403 プロセッサー用に、最適化が調整されます。

604
PowerPC 604 プロセッサー用に、最適化が調整されます。

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。

ppc970
PowerPC 970 プロセッサー用に、最適化が調整されます。

pwr3
POWER3 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。

pwr4
POWER4 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。

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。

rs64a
RS64I プロセッサー用に、最適化が調整されます。

rs64b
RS64II プロセッサー用に、最適化が調整されます。

rs64c
RS64III プロセッサー用に、最適化が調整されます。

注: V9.0 リリースのコンパイラーでは、601、602、603、POWER および POWER2 アーキテクチャーに対応するサブオプションは推奨しません。

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。

-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。

-qarch と **-qtune** の受け入れ可能な組み合わせは、以下の表に示します。

表 23. 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 設定**
403	403	auto | 403
604	604	auto | 604
ppc	balanced	auto | 604 | rs64a | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppcgr	balanced	auto | 604 | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64	balanced	autors64a | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64gr	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64grsq	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64v	ppc970	auto | ppc970 | pwr6 | balanced
ppc970	ppc970	auto | ppc970 | balanced
pwr3	pwr3	auto | pwr3 | pwr4 | pwr5 | ppc970 | balanced
pwr4	pwr4	auto | pwr4 | pwr5 | ppc970 | balanced

表 23. 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 設定**
pwr5	pwr5	auto | pwr5 | balanced
pwr5x	pwr5	auto | pwr5 | balanced
pwr6	pwr6	auto | pwr6 | balanced
pwr6e	pwr6	auto | pwr6 | balanced
rs64a	rs64a	auto | rs64a
rs64b	rs64b	auto | rs64b
rs64c	rs64c	auto | rs64c

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` が、POWER3 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=pwr3
```

関連情報

- 70 ページの『-qarch』
- 61 ページの『-q32、-q64』
- 11 ページの『アーキテクチャー固有の 32 ビットまたは 64 ビットのコンパイルでのコンパイラー・オプションの指定』
- 「 *XL 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、349 ページの『第 5 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

- 94 ページの『-D』

-qunique

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

静的コンストラクター/デストラクター・ファイルのコンパイル単位の固有名を生成する。

構文

デフォルト

-qnounique

使用法

固有名は、**-qunique** を指定し、乱数を静的コンストラクター関数とデストラクター関数の名前にエンコードすることにより生成されます。デフォルトの振る舞いでは、コンストラクター関数とデストラクター関数内のソース・ファイルの絶対パス名がエンコードされます。絶対パス名を複数のコンパイルについて同一にする場合 (**make** スクリプトを使用する場合など) は、 **-qunique** オプションが必要です。

-qunique を使用する場合は、必ずすべての .o および .a ファイルとリンクしなければなりません。リンク・ステップでは、実行可能ファイルを組み込まないでください。

事前定義マクロ

なし。

例

静的構造体が正常に機能することを確認して、同じパス名を使用して幾つかのファイルをコンパイルしたいと想定します。Make ファイルは以下のステップを生成する可能性があります。

```
sqlpreprocess file1.sql > t.C
  xlc++ -qunique t.C -o file1.o
rm -f t.C
sqlpreprocess file2.sql > t.C
  xlc++ -qunique t.C -o file2.o
rm -f t.C
  xlc++ file1.o file2.o
```

以下は上記の例のサンプル Make ファイルです。

```
# rule to get from file.sql to file.o
.SUFFIXES:      .sql
.sql.o:
       sqlpreprocess $< > t.C
       $(CCC) t.C -c $(CCFLAGS) -o $@
       rm -f t.C
```

関連情報

- 301 ページの『#pragma fini』
- 303 ページの『#pragma init』

-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、#pragma unroll

目的

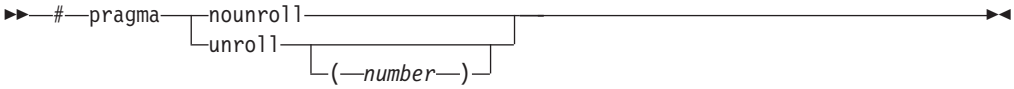
パフォーマンスを改善するために、ループのアンロールを制御する。

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。

number **(プラグマのみ)**
指定したループ本体またはループのフル・アンロールのどちらが先に起こっても、そのいずれかの *number*- 1 レプリケーションを強制します。 *number* の値は無制限で、正の整数でなければなりません。**#pragma unroll(1)** を効果的に指定すると、ループのアンロールを使用不可にします。これは、**#pragma nounroll** を指定することと同等です。*number* が指定されておらず、 **-qhot**、 **-qsmp**、または **-O4** 以上が指定されている場合は、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。

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 **-q[no]unroll** コンパイラー・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。内部にネストされたループは、該当のループのアンロール方法が一般的な **-q[no]unroll** オプションの方法と異なる場合は、**#pragma unroll** ディレクティブの後に続く必要があります。

#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

- 292 ページの『#pragma block_loop』
- 305 ページの『#pragma loopid』
- 321 ページの『#pragma stream_unroll』
- 322 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。

事前定義マクロ

なし。

-qupconv

カテゴリー

移植性およびマイグレーション

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。

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* ランゲージ・リファレンス」の 『整数および浮動小数点プロモーション』
- 166 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

-qnoutf

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 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

- 60 ページの『-# (ポンド記号)』

-qvecnvol

カテゴリー

移植性およびマイグレーション

プラグマ同等物

なし。

目的

揮発性ベクトル・レジスターを使用するか不揮発性ベクトル・レジスターを使用するかを指定する。

揮発性ベクトル・レジスターとは、その値が関数呼び出し間、または保管コンテキスト、あるいはジャンプまたはスイッチ・コンテキストのシステム・ライブラリー関数の間で保存されないレジスターのことです。 **-qvecnvol** が有効な場合、コンパイラーは揮発性ベクトル・レジスターおよび不揮発性ベクトル・レジスターの両方を使用します。 **-qnovecnvol** が有効な場合、コンパイラーは揮発性ベクトル・レジスターのみを使用します。

このオプションは、AIX 5.3 (5300-03) よりも前の AIX ライブラリーを使用してビルドされたモジュールとベクトル・レジスター使用の間で相互作用の危険性があるプログラムに必要です。揮発性レジスターのみを使用するようにコンパイラーを制限すると、ベクトル・プログラムは安全になりますが、潜在的にコンパイラーにベクトル・データをもっと頻繁にメモリーに保管するよう強制することになるため、パフォーマンスが低下する結果になります。

構文

デフォルト

-qnovecnvol

使用法

- ベクトルが使用可能なコードを生成するには、明示的に **-qenablevmx** オプションを指定してください。
- **-qvecnvol** オプションを使用するためには、`bos.adt.include` バージョン 5.3.0.30 以上がシステムにインストールされている必要があります。
- **-qnoenablevmx** コンパイラー・オプションが有効な場合、**-qnovecnvol** オプションは無視されます。
- **-qnovecnvol** オプションは、**-qhot=simd | nosimd** および **-qaltivec | -qnoaltivec** および **pragma=nosimd** とは独立して実行されます。

- AIX 5.3 (5300-03) では、デフォルトで 20 の揮発性レジスター (vr0-vr19) のみが使用され、12 の不揮発性ベクトル・レジスター (vr20 - vr31) は使用されません。これらのレジスターは、**-qvecnvol** が有効な場合にのみ使用できます。
- **-qvecnvol** は、不揮発性レジスターを保管および復元するレガシー・コードが関係しない場合にのみ使用可能にしてください。**-qvecnvol** とリンクをレガシー・コードと共に使用すると、ランタイム障害が発生することがあります。

事前定義マクロ

なし。

関連情報

- 69 ページの『-qaltivec』
- 109 ページの『-qenablevmx』
- 133 ページの『-qhot』

-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

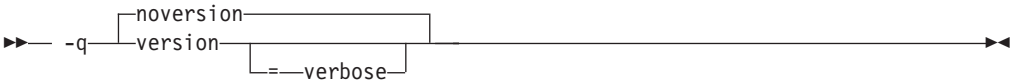
プラグマ同等物

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。

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。

例:

```
IBM XL C for AIX, V10.1
Version: 10.01.0000.0001
```

-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_level*

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_level
　　インストール済みコンポーネントのレベルを表します。

例:

```
IBM XL C for AIX, V10.1
Version: 10.01.0000.0001
Driver Version: 10.01(C) Level: 060414
C Front End Version: 10.01(C) Level: 060419
High Level Optimizer Version: 10.01(C) and 12.01(Fortran) Level: 060411
Low Level Optimizer Version: 10.01(C) and 12.01(Fortran) Level: 060418
```

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。

事前定義マクロ

なし。

関連情報

- 227 ページの『-qsaveopt』

-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。

このオプションは、**-qflag=e : e** 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
```

関連情報

- 114 ページの『-qflag』
- 249 ページの『-qsuppress』

-W

カテゴリー

コンパイラーのカスタマイズ

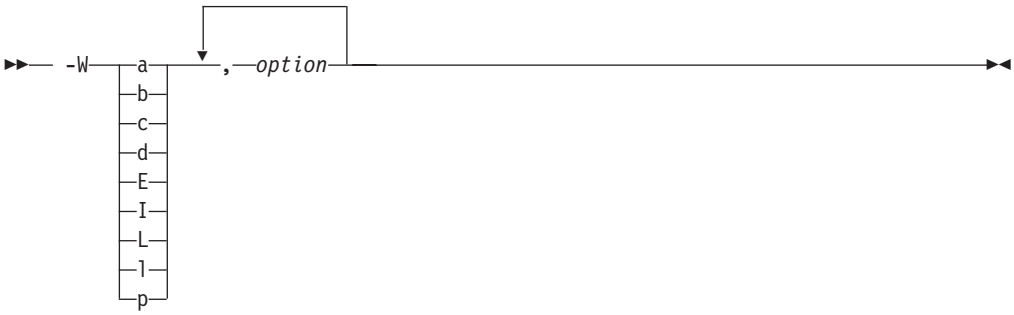
プラグマ同等物

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実行可能ファイル名との間の対応を示しています。

パラメーター	説明	実行可能ファイル名
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
d	逆アセンブラー	dis
E	**CreateExportList** ユーティリティー	CreateExportList
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	リンカー	ld
p	プリプロセッサー	該当なし

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』

-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

- -q32、-q64
- 18 ページの『コンパイラー・メッセージ』

-qweakexp

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

-qmkshrobj または **-G** オプションと一緒に使用して、共有オブジェクトを作成するときに生成されるエクスポート・リストから「弱い」とマークを付けられたグローバル・シンボルを組み込むかまたは除外する。

構文

デフォルト

-qweakexp: 弱いシンボルはエクスポートされます。

使用法

弱いシンボルの説明については、 280 ページの『-qweaksymbol』を参照してください。

このオプションは、**-qmkshrobj** または **-G** オプションと一緒に使用されます。詳しくは、 191 ページの『-qmkshrobj』 または 129 ページの『-G』 の説明を参照してください。

事前定義マクロ

なし。

例

`myprogram.c` を共有オブジェクトにコンパイルし、弱いシンボルがエクスポートされないようにするには、以下のように入力します。

```
xlc myprogram.c -qmkshrobj -qnoweakexp
```

関連情報

- 『-qweaksymbol』
- 324 ページの『#pragma weak』
- 191 ページの『-qmkshrobj』
- 129 ページの『-G』

-qweaksymbol

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

弱いシンボルの生成を使用可能にする。

-qweaksymbol が有効な場合、コンパイラーは以下に対して弱いシンボルを生成します。

- 外部リンケージとのインライン関数。
- **#pragma weak** または `__attribute__((weak))` を使用して弱いと指定された ID。

構文

デフォルト

-qweaksymbol

事前定義マクロ

なし。

関連情報

- 324 ページの『#pragma weak』
- 279 ページの『-qweakexp』
- 「*XL C ランゲージ・リファレンス*」の『weak 変数属性』、および『weak 関数属性』

-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

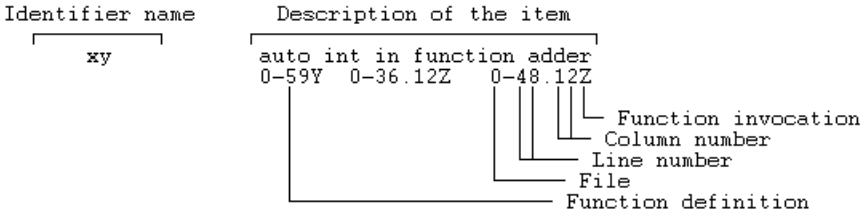
-qnoxref

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 24. 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。

-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

- 77 ページの『-qattr』
- 307 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。

以下のサブオプションは 10 進浮動小数点型のみに有効です。

di (ゼロから離れて) 無限大の方向に丸める。

dm
負の無限大の方向に丸める。

dn 表現可能な偶数の近似値まで丸める。

dna
ゼロから離れた方向に表現可能な近似値まで丸める。

dnz
ゼロの方向に表現可能な近似値まで丸める。

dp 正の無限大の方向に丸める。

dz 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 -ydz
```

-Z

カテゴリー

リンク

プラグマ同等物

なし。

目的

リンカーが使用するライブラリー検索パスのプレフィックスを指定する。

構文

デフォルト

デフォルトでは、リンカーは /usr/lib/ ディレクトリーでライブラリー・ファイルを検索します。

パラメーター

string
ライブラリー・ファイルのディレクトリー検索パスに追加されるプレフィックスを表します。

事前定義マクロ

なし。

第 4 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

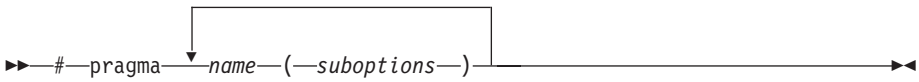
XL C がサポートするプラグマ・ディレクティブの形式は 3 つです。

#pragma options *option_name*

これらのプラグマは、コマンド行オプションと同等なものと正確に同じ構文を使用します。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、309 ページの『#pragma options』に記載されています。

#pragma *name*

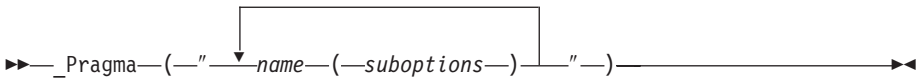
この形式は以下の構文を使用します。



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)" )
```

は、以下と同等です。

```
#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。

プラグマ・ディレクティブの範囲

コンパイル単位のソース・コード内ならどの時点でも多くのプラグマ・ディレクティブを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、指定される時点からコンパイル単位の終わりまで適用されます。

コードの選択したセクションのまわりでプラグマ・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
```

多くのプラグマは『pop』サブオプションまたは『reset』サブオプションを提供します。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 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。

- 『言語エレメント制御』
- 289 ページの『浮動小数点および整数制御』
- 289 ページの『エラー・チェックおよびデバッグ』
- 289 ページの『最適化およびチューニング』
- 290 ページの『オブジェクト・コード制御』
- 291 ページの『移植性およびマイグレーション』

これらのカテゴリーの説明は、 43 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 25. 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。

表 25. 言語エレメント制御プラグマ (続き)

プラグマ	説明
307 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。
309 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。

浮動小数点および整数制御

表 26. 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 27. エラー・チェックおよびデバッグ・プラグマ

プラグマ	説明
302 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。

最適化およびチューニング

表 28. 最適化およびチューニングのプラグマ

プラグマ	説明
292 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。
319 ページの『#pragma STDC cx_limited_range』	複素数除算と絶対値は中間計算がオーバーフローしたり重要度を失わないような値のみを使用して呼び出されることをコンパイラーに命令する。
297 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。
298 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。
300 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、関数のクローン化およびインライン化など、特定の最適化を実行することができます。

表 28. 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。
304 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。
305 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	**-qhot=simd** 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=novector** 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。
311 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。
317 ページの『#pragma reachable』	指定した関数のあとのプログラムでのポイントがいくつかの認識されないロケーションからの分岐のターゲットとなれることをコンパイラーに通知する。
318 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。
321 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。
322 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 29. 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。
295 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。
301 ページの『#pragma fini』	main() が完了した後、または exit() が呼び出された後に、ランタイム・ライブラリーがリストの関数を呼び出す順序を指定する。

表 29. オブジェクト・コード制御プラグマ (続き)

プラグマ	説明
303 ページの『#pragma init』	main() の呼び出し前に、ランタイム・ライブラリーが関数のリストを呼び出す順序を指定する。
306 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。
313 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。
318 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。
324 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。

移植性およびマイグレーション

表 30. 移植性およびマイグレーション・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。

個々のプラグマの説明

この節には XL 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、287 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

65 ページの『-qalign』を参照してください。

#pragma alloca

68 ページの『-qalloca、-ma』を参照してください。

#pragma block_loop

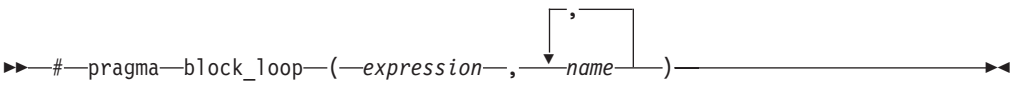
カテゴリー

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

- 305 ページの『#pragma loopid』
- 267 ページの『-qunroll』
- 322 ページの『#pragma unrollandfuse』
- 321 ページの『#pragma stream_unroll』

#pragma chars

87 ページの『-qchars』を参照してください。

#pragma comment

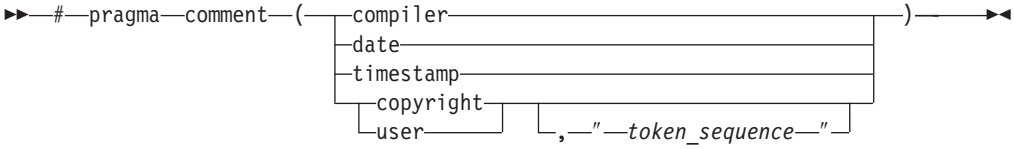
カテゴリー

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、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・コードがコンパイルされて出力ファイル `a.out` が作成されると想定します。

```
#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、

```
strings a.out
```

a.out に組み込まれたコメント情報が、a.out で検出された他のストリングと共に表示されます。例えば上記のプログラム・コードを想定した場合、以下のようになります。

```
Mon Mar 1 10:28:03 2008
XL C/C++ for AIX Compiler Version 10.1
Mon Mar 1 10:28:09 2008
My copyright
```

#pragma disjoint

カテゴリー

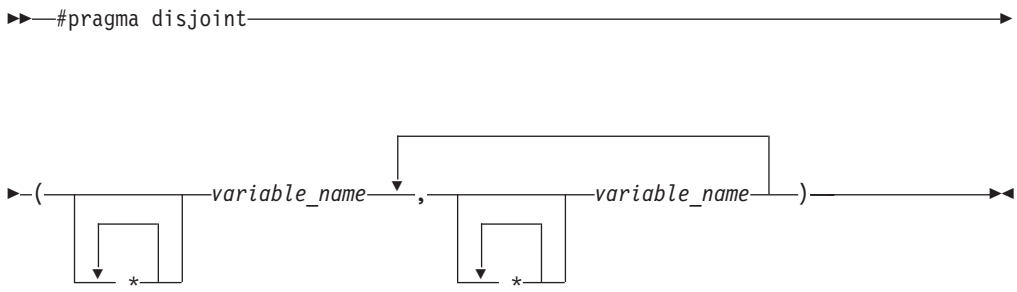
最適化およびチューニング

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用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用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用している 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 とストレージを共用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 とストレージを共用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 enum

105 ページの『-qenum』を参照してください。

#pragma execution_frequency

カテゴリー

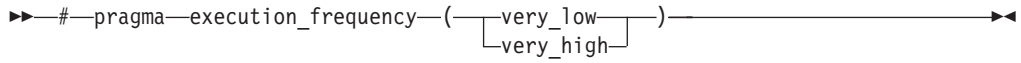
最適化およびチューニング

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

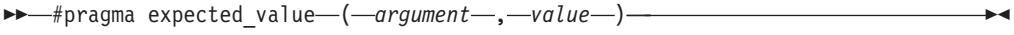
カテゴリー

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、関数のクローン化およびインライン化など、特定の最適化を実行することができます。

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

- 298 ページの『#pragma execution_frequency』

#pragma fini

カテゴリー

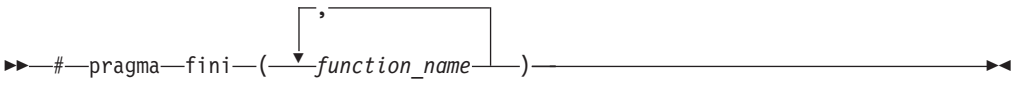
290 ページの『オブジェクト・コード制御』

目的

main() が完了した後、または exit() が呼び出された後に、ランタイム・ライブラリーがリストの関数を呼び出す順序を指定する。

共用ライブラリーの場合、fini 関数はその共用ライブラリーがメモリーからロードされるときに呼び出されます。例えば、動的ロードを使用する場合、これは、dlclose() が呼び出される時点で行われます。

構文



使用法

このプラグマで指定する関数は、パラメーターなしの戻り型 void (例えば、`void fA();`) を持つ必要があります。 非 void 戻り型を持つ関数は受け入れられますが、その戻り値は破棄されます。

パラメーターが指定された関数は、そのパラメーターに無効な値が含まれる可能性があるため、無視されて警告が出されます。

同じコンパイル単位内では、`pragma fini` 内のリストされた関数は指定された順序で呼び出されます。同様に、同じコンパイル単位内では、複数の pragma fini 内に指定された関数は、それらのプラグマがソース内に検出される順序で呼び出されます。

一般に、複数のファイルやライブラリーにまたがる静的終了の順序は標準外であるため、振る舞いに移植性がありません。この振る舞いに従属関係を持たせることはお勧めできません。複数のファイルにまたがる関数の順序は、**-Wm** オプションを使用する場合であっても未定義となります。

C ファイルおよび C++ ファイルを混合した場合、C++ ファイル内の静的コンストラクター/デストラクターに対する C ファイル内の init 関数および fini 関数の相対順序は未定義になります。**-qunique** オプション は `pragma fini` と相互作用することができます。

注: C++ 呼び出し (**xlC** など) または再配布可能ツール (**linkxlC** または **makeC++SharedLib**) は、リンク時に使用する必要があります。

関連情報

- 303 ページの『#pragma init』
- 265 ページの『-qunique』

#pragma ibm snapshot

カテゴリー

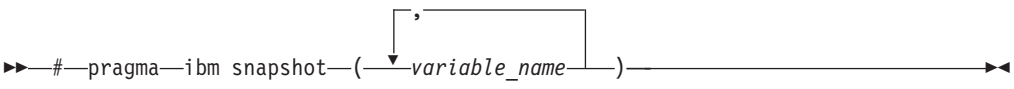
エラー・チェックおよびデバッグ

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

- 128 ページの『-g』
- 194 ページの『-O、-qoptimize』

#pragma info

142 ページの『-qinfo』を参照してください。

#pragma init

カテゴリー

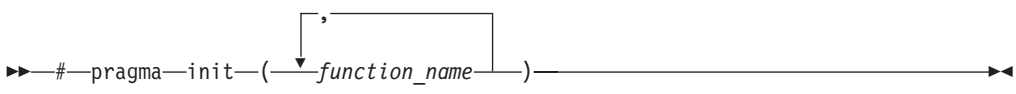
290 ページの『オブジェクト・コード制御』

目的

main() の呼び出し前に、ランタイム・ライブラリーが関数のリストを呼び出す順序を指定する。

共用ライブラリーの場合、init 関数はその共用ライブラリーがメモリーにロードされるときに呼び出されます。例えば、動的ロードを使用する場合、これは、dlopen() が呼び出される時点で行われます。

構文



使用法

このプラグマで指定する関数は、パラメーターなしの戻り型 void (例えば、`void fA();`) を持つ必要があります。 非 void 戻り型を持つ関数は受け入れられますが、その戻り値は破棄されます。

パラメーターが指定された関数は、そのパラメーターに無効な値が含まれる可能性があるため、無視されて警告が出されます。

同じコンパイル単位内では、`pragma init` 内のリストされた関数は、指定された順序で呼び出されます。同様に、同じコンパイル単位内では、複数の `pragma init` 内に指定された関数は、それらのプラグマがソース内で検出される順序で呼び出されます。

一般に、複数のファイルやライブラリーにまたがる静的初期化の順序は標準外であるため、振る舞いに移植性がありません。この振る舞いに従属関係を持たせることはお勧めできません。複数のファイルにまたがる関数の順序は、**-Wm** オプションを使用する場合であっても未定義となります。

C ファイルおよび C++ ファイルを混合した場合、C++ ファイル内の静的コンストラクター/デストラクターに対する C ファイル内の init 関数の相対順序は未定義になります。**-qunique** オプション は `pragma init` と相互作用することができます。

注: xlC のような C++ 呼び出しや再配布可能ツール (**linkxlC** または **makeC++SharedLib**) はリンク時に使用する必要があります。

関連情報

- 301 ページの『#pragma fini』
- 265 ページの『-qunique』

#pragma isolated_call

160 ページの『-qisolated_call』を参照してください。

#pragma langlvl

166 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

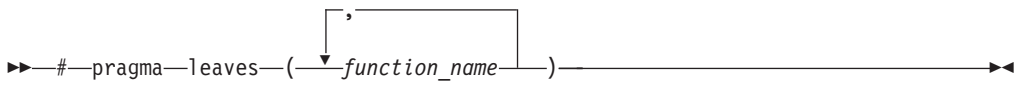
指定した関数が、その関数の呼び出しの後の命令に戻らないことをコンパイラーに通知する。

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

- 317 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、292 ページの『#pragma block_loop』を参照してください。

関連情報

- 267 ページの『-qunroll』
- 292 ページの『#pragma block_loop』
- 322 ページの『#pragma unrollandfuse』

#pragma map

カテゴリー

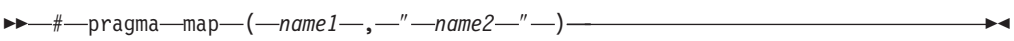
オブジェクト・コード制御

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



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、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。

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。

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、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 */

#pragma map (foo, "bar")

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!
```

関連情報

- 「*XL 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

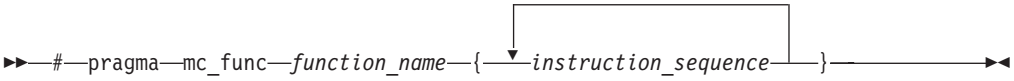
言語エレメント制御

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 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、 318 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 =
}
```

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

- 160 ページの『-qisolated_call』
- 318 ページの『#pragma reg_killed_by』
- 「*XL 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 nosimd

133 ページの『-qhot』を参照してください。

#pragma novector

133 ページの『-qhot』を参照してください。

#pragma options

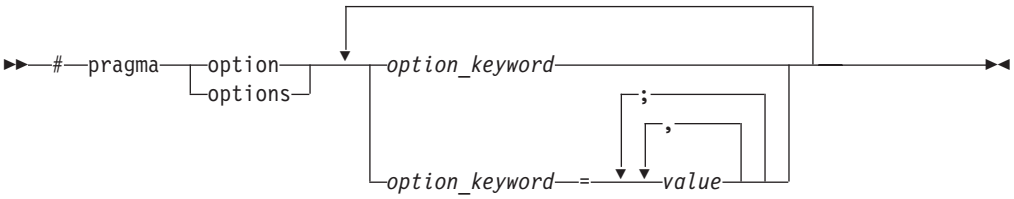
カテゴリー

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*	65 ページの『-qalign』
[no]attr attr=full	77 ページの『-qattr』
chars=*option*	87 ページの『-qchars』
[no]check	89 ページの『-qcheck』
[no]compact	90 ページの『-qcompact』
[no]dbcs	188 ページの『-qmbcs, -qdbcs』
[no]dbxextra	97 ページの『-qdbxextra』
[no]digraph	99 ページの『-qdigraph』
[no]dollar	101 ページの『-qdollar』
enum=*option*	105 ページの『-qenum』
[no]extchk	110 ページの『-qextchk』
flag=*option*	114 ページの『-qflag』
float=[no]*option*	116 ページの『-qfloat』
[no]flttrap=*option*	121 ページの『-qflttrap』
[no]fullpath	125 ページの『-qfullpath』
halt	131 ページの『-qhalt』
[no]idirfirst	137 ページの『-qidirfirst』
[no]ignerrno	139 ページの『-qignerrno』
ignprag=*option*	139 ページの『-qignprag』
[no]info=*option*	142 ページの『-qinfo』
initauto=*value*	148 ページの『-qinitauto』
[no]inlglue	149 ページの『-qinlglue』
isolated_call=*names*	160 ページの『-qisolated_call』
langlvl	166 ページの『-qlanglvl』
[no]ldbl128	172 ページの『-qldbl128、-qlongdouble』
[no]libansi	175 ページの『-qlibansi』
[no]list	176 ページの『-qlist』
[no]longlong	180 ページの『-qlonglong』
[no]macpstr	181 ページの『-qmacpstr』

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]maxmem=*number*	186 ページの『-qmaxmem』
[no]mbcs	188 ページの『-qmbcs, -qdbcs』
[no]optimizeoptimize=*number*	194 ページの『-O、-qoptimize』
proclocal、procimported、procunknown	212 ページの『-qprocimported、-qproclocal、-qprocunknown』
[no]proto	213 ページの『-qproto』
[no]ro	221 ページの『-qro』
[no]roconst	222 ページの『-qroconst』
[no]showinc	229 ページの『-qshowinc』
[no]source	237 ページの『-qsource』
spill=*number*	241 ページの『-qspill』
[no]srcmsg	241 ページの『-qsrcmsg』
[no]stdinc	243 ページの『-qstdinc』
[no]strict	244 ページの『-qstrict』
tbtable=*option*	255 ページの『-qtbtable』
tune=*option*	261 ページの『-qtune』
[no]unrollunroll=*number*	267 ページの『-qunroll』
[no]upconv	270 ページの『-qupconv』
[no]xref	281 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

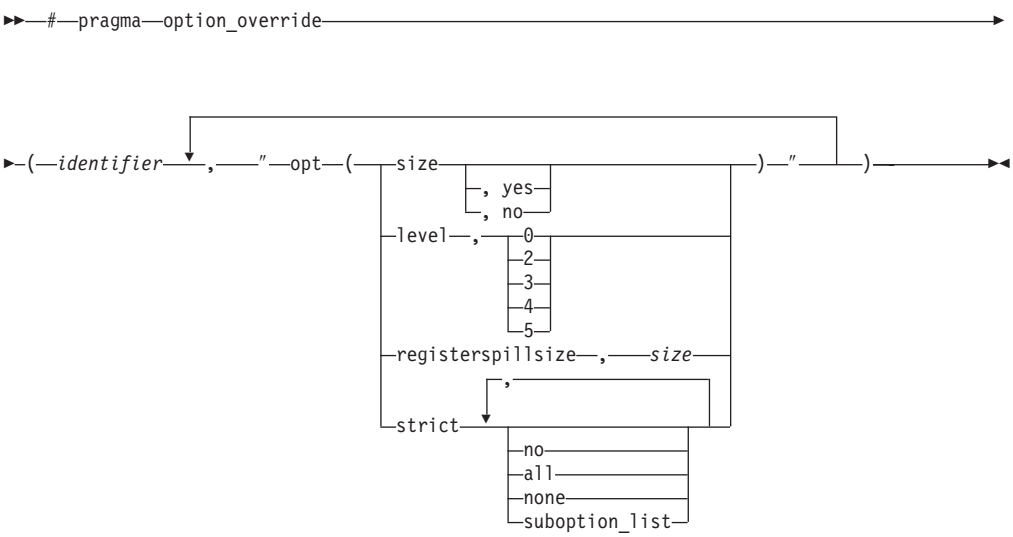
最適化およびチューニング

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。

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

- 194 ページの『-O、-qoptimize』
- 90 ページの『-qcompact』
- 241 ページの『-qspill』
- 244 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。

構文

デフォルトの #pragma pack 構文 (有効な -qpack_semantic=ibm)

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。

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
#pragma pack によって追加された以前の値を除去します。パラメーターなしで **#pragma pack()** を指定することは、**pop** と同等です。

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、1 ビット境界上の構造体/共用体のすべてのビット・フィールドの位置合わせを行います。例:

```
#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。

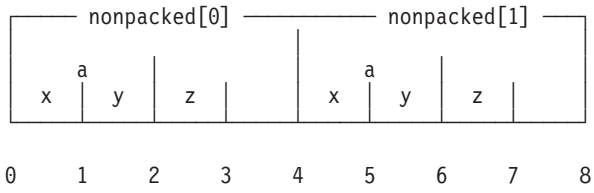
```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

デフォルト・マッピング:	**#pragma pack(1):**
s_t のサイズは 16	s_t のサイズは 11
a のオフセットは 0	a のオフセットは 0
b のオフセットは 4	b のオフセットは 1
c のオフセットは 8	c のオフセットは 5
d のオフセットは 12	d のオフセットは 7
a の位置合わせは 1	a の位置合わせは 1
b の位置あわせは 4	b の位置合わせは 1
c の位置合わせは 2	c の位置合わせは 1
d の位置合わせは 4	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];
```

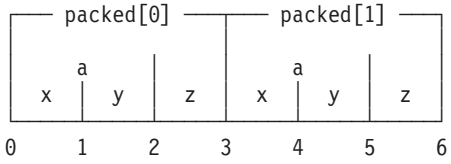
共用体メンバーの中では位置合わせに関する最大の要件が short a、つまり 2 バイトの要件なので、埋め込みの 1 バイトが配列中の各共用体の終わりに追加され、この要件を強制します。



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。



関連情報

- 65 ページの『-qalign』
- 「*XL 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 reachable

カテゴリー

最適化およびチューニング

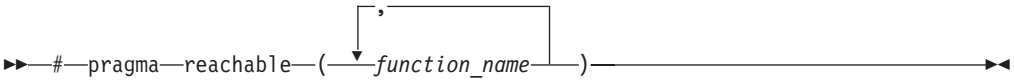
目的

指定した関数のあとのプログラムでのポイントがいくつかの認識されないロケーションからの分岐のターゲットとなれ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

- 304 ページの『#pragma leaves』

#pragma reg_killed_by

カテゴリー

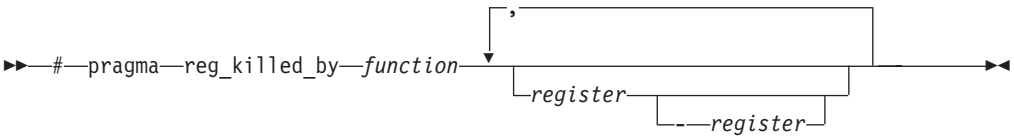
最適化およびチューニング

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。以下は、有効なレジスターです。

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

fs 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 =
}
```

関連情報

- 307 ページの『#pragma mc_func』

#pragma STDC cx_limited_range

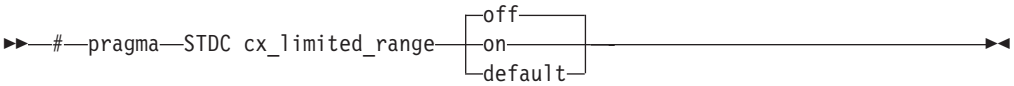
カテゴリー

最適化およびチューニング

目的

複素数除算と絶対値は中間計算がオーバーフローしたり重要度を失わないような値のみを使用して呼び出されることをコンパイラーに命令する。

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 ランゲージ・リファレンス*」の 『標準プラグマ』

#pragma stream_unroll

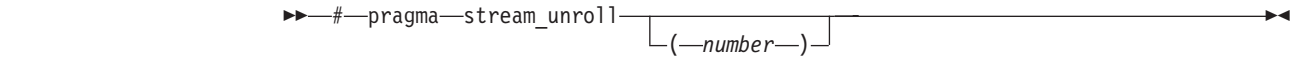
カテゴリー

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。 *number* の値は正の整数定数式です。

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。 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。

例

以下は、**#pragma stream_unroll** がどのようにパフォーマンスを改善できるかを示した例です。

```
int i, m, n;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];


....

#pragma stream_unroll(4)
for (i=1; 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。

```
for (i=1; 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];
}
```

関連情報

- 267 ページの『-qunroll』
- 『#pragma unrollandfuse』

#pragma strings

221 ページの『-qro』を参照してください。

#pragma unroll

267 ページの『-qunroll』を参照してください。

#pragma unrollandfuse

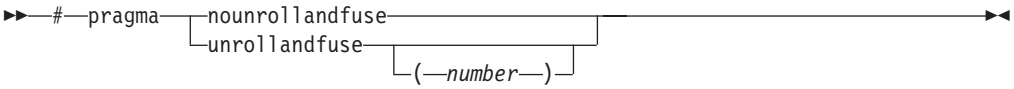
カテゴリー

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。 *number* の値は正の整数定数式です。

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

- 267 ページの『-qunroll』
- 321 ページの『#pragma stream_unroll』

#pragma weak

カテゴリー

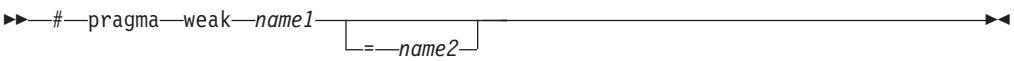
オブジェクト・コード制御

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。

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。

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。

#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用ライブラリー・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>

#pragma weak foo

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

#pragma weak foo = foo2

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 ランゲージ・リファレンス*」の『weak 変数属性』
- 「*XL C ランゲージ・リファレンス*」の『weak 関数属性』
- 306 ページの『#pragma map』
- 280 ページの『-qweaksymbol』
- 279 ページの『-qweakexp』

並列処理のプラグマ・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。このプラグマは、**-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。

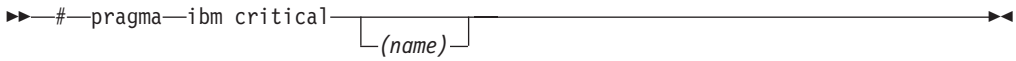
C プログラムでは IBM SMP または OpenMP ディレクティブ、C++ プログラムでは OpenMP ディレクティブを使用することができます。それぞれ独自の使用特性を持っています。

#pragma ibm critical

目的

critical プラグマは、一度に 1 つのプロセスでしか実行してはならない、プログラム・コードのクリティカル・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ます。

使用法

クリティカル・セクションに出入りする分岐を行おうとすると、コンパイラーによってエラーが報告されます。 エラーの原因となる状態を、以下にいくつか示します。

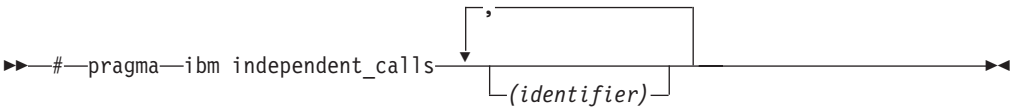
- クリティカル・セクションに return 文が含まれている。
- クリティカル・セクションに、クリティカル・セクションの外側にプログラム・フローを受け渡す、goto、continue、または break 文が含まれている。
- クリティカル・セクション内に定義されているラベルにプログラム・フローを転送する goto 文がクリティカル・セクションの外側にある。

#pragma ibm independent_calls

説明

independent_calls プラグマは、選択されたループ内の指定された関数呼び出しが、ループの実行に依存しないことを表明します。この情報は、コンパイラーによる依存性の分析に役立ちます。

構文



ここで、*identifier* は、関数の名前を表すコンマで区切られたリストです。

使用法

identifier を関数へのポインターの名前にすることはできません。

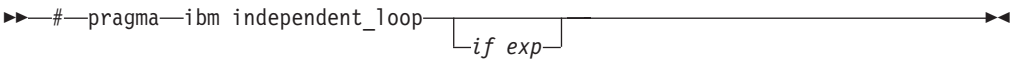
関数 ID が指定されていない場合、コンパイラーは、ループ内のすべての関数について実行に対する依存性がないと見なします。

#pragma ibm independent_loop

目的

independent_loop プラグマは、選択されたループの繰り返しが独立しており、なおかつそのループが並列化できることを表明します。

構文



ここで、*exp* はスカラー式を表します。

使用法

`if` 引数が指定されている場合、*exp* が実行時に TRUE と評価される場合に限り、ループの繰り返しは独立していると見なされます。

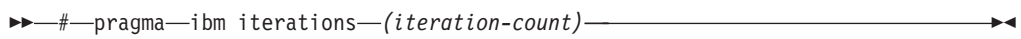
このプラグマを **schedule** プラグマと組み合わせて、特定の並列処理スケジューリング・アルゴリズムを選択することができます。詳しくは、**schedule** プラグマに対する 330 ページの『#pragma ibm schedule』の説明を参照してください。

#pragma ibm iterations

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繰り返し回数を指定します。

構文



ここで、*iteration-count* は正の整数定数式を表します。

使用法

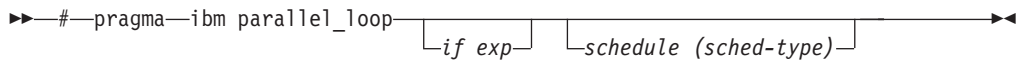
コンパイラーは、*iteration-count* 変数の情報を使用して、ループの並列化が効率的であるかどうかを判別します。

#pragma ibm parallel_loop

目的

parallel_loop プラグマは、選択したループを並列化するようにコンパイラーに明示的に指示します。

構文



ここで、*exp* はスカラー式を、*sched-type* は *schedule* ディレクティブに有効な任意のスケジューリング・アルゴリズムを表します。

使用法

`if` 引数が指定されると、実行時に *exp* が TRUE に評価された場合にのみ、ループが並行して実行されます。それ以外の場合は、ループは順次実行されます。ループは、クリティカル・セクションにある場合にも順次実行されます。

このプラグマは、多種多様な C ループに適用することができ、コンパイラーは、ループがカウント可能であるかどうかを判別しようとします。

parallel_loop プラグマを使用するプログラム・セクションでは、順次モードでも並列モードでも正しい結果が生成できなければなりません。例えば、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。条件付き同期化に関連する明示的な並列プログラミング技法は許されていません。

コンパイラーは、このプラグマでマークを付けられたループに対する縮小を自動的には検出しません。縮小を伴うループを正しく並列化するには、以下を行います。

- **parallel for** を使用して、明示的に縮小を指定する。
- **#pragma ibm independent_loop** を使用して、コンパイラーに縮小を発見させる。

このプラグマを **schedule** 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、330 ページの『#pragma ibm schedule』プラグマの説明を参照してください。

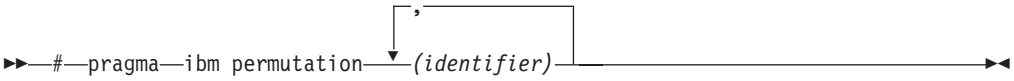
このプラグマの後にカウント可能なループを指定していない場合には、警告が生成されます。

#pragma ibm permutation

目的

permutation プラグマは、それに続くループで、名前付き配列の異なるエレメントが異なる値を持つことを断言します (つまり、`a[i] == a[j] iff i == j`)。

構文



ここで、*identifier* は配列の名前を表します。 *identifier* を関数仮パラメーターまたはポインターの名前にすることはできません。

使用法

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。

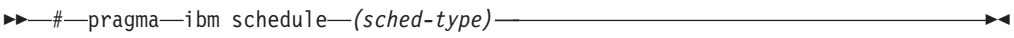
エレメントが他の配列の指標に使用される場合、この断言がループ変換を使用可能にする可能性があります。このプラグマは、疎データ構造を扱うプログラムに有用です。

#pragma ibm schedule

目的

schedule プラグマは、並列処理に使用するスケジューリング・アルゴリズムを指定します。

構文



パラメーター

sched-type は以下のオプションのいずれかを表します。

affinity
最初にループの繰り返しは、サイズ **ceiling**(*number_of_iterations*/*number_of_threads*) のローカル区画に分割されます。その後、各ローカル区画はさらにサイズ **ceiling**(*number_of_iterations_remaining_in_partition*/2) のチャンクに分割されます。

スレッドは使用可能になると、そのローカル区画から次のチャンクを取得します。ローカル区画にチャンクがなくなると、スレッドは別のスレッドの区画から使用可能なチャンクを取得します。

affinity,*n*
ローカル区画がそれぞれサイズ *n* のチャンクに再分割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。

dynamic
ループの繰り返しがサイズ 1 のチャンクに分割されます。

チャンクは、スレッドが使用可能になると、最初に提供されたものから順に処理されるようにスレッドに割り当てられます。これは、すべての作業が完了するまで続けられます。

dynamic,*n*
すべての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、チャンク・サイズが 1 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_remaining*/*number_of_threads*) です。

チャンクは、スレッドが使用可能になると、最初に提供されたものから順に処理されるようにスレッドに割り当てられます。 これは、すべての作業が完了するまで続けられます。

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。

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。

注: *n*=1 が 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。

使用法

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。

並列処理のためのスケジューリング・アルゴリズムは、以下に示す任意の方法を使用して指定することができます。 リストの上位の方式を使用すると、その方式によってリストの下位の項目がオーバーライドされます。

- プラグマ・ステートメント

- コンパイラーのコマンド行オプション
- ランタイム・コマンド行オプション
- ランタイム・デフォルト・オプション

スケジューリング・アルゴリズムは、**parallel_loop** および **independent_loop** プラグマ・ステートメントの **schedule** 引数を使用して指定することもできます。例えば、以下のステートメントの設定は同等です。

```
#pragma ibm parallel_loop
#pragma ibm schedule (sched_type)
<countable for|while|do loop>
```
および
```
#pragma ibm parallel_loop (sched_type)
<countable for|while|do loop>
```

あるループに異なるスケジューリング・タイプを指定した場合は、最後に指定したものが適用されます。

#pragma ibm sequential_loop

目的

sequential_loop プラグマは、選択したループを順次実行するようにコンパイラーに明示的に指示します。

構文

使用法

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。

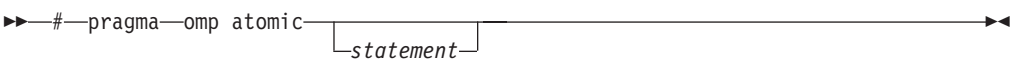
このプラグマは、選択したループの自動的な並列化を使用不可にし、コンパイラーは常にこのプラグマに従います。

#pragma omp atomic

目的

omp atomic ディレクティブは、アトミックに更新しなければならない、また複数の同時書き込みスレッドに公開してはならない、特定のメモリー・ロケーションを識別します。

構文



ここで、*statement* は、以下に続くいずれかの形式をとるスカラー型の式ステートメントです。

statement	**条件**
x bin_op = *expr*	ここで、 *bin_op* は、以下のいずれかです。 + * - / & ^ | << >> *expr* は、*x* を参照しないスカラー型の式です。
x++	
++x	
x--	
--x	

使用法

ロードおよび保管の操作は、オブジェクト *x* に対してのみアトミックです。*expr* の評価は、アトミックではありません。

プログラム内の指定オブジェクトに対するすべてのアトミック参照は、互換タイプを持っていなければなりません。

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。

例

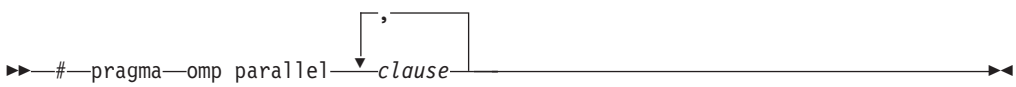
```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates.  */
#pragma omp atomic
x[index[i]] += y;

/* Protect against races with updates through x.            */
#pragma omp atomic
p[i] -= 1.0f;
```

#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、次のいずれかです。

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用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、共用の縮約変数の元の値に戻されます。

reduction 節に指定される変数は以下の通りです。
- 演算子に適切な型でなければならない。

- 囲んでいるコンテキスト内で共用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。

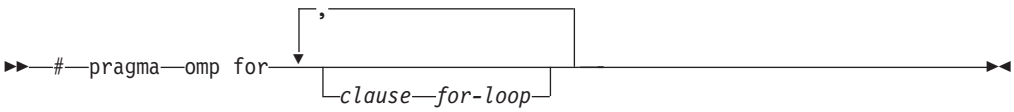
ネストされた並列領域は、常に直列化されています。

#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、次のいずれかです。

collapse (*n*)
collapse 文節を指定すると、ネストされた並列処理を導入せずに、ネスト内の複数のループを並列化することができます。詳しくは、トピック 338 ページの『collapse』を参照してください。

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、共用の縮約変数の元の値に戻されます。

reduction 節に指定される変数は以下の通りです。

- 演算子に適切な型でなければならない。
- 囲んでいるコンテキスト内で共用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、スレッドが使用可能になると、最初に提供されたものから順に処理されるようにスレッドに動的に割り当てられます。 これは、すべての作業が完了するまで続けられます。

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、スレッドが使用可能になると、最初に提供されたものから順に処理されるようにスレッドに割り当てられます。 これは、すべての作業が完了するまで続けられます。

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 が 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 の形式は

```
iv = b
integer-type iv = b
```

exit_cond の形式は

```
iv <= ub
iv <  ub
iv >= ub
iv >  ub
```

incr_expr の形式は

```
++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。

制限は、以下のとおりです。

- 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。

関連資料

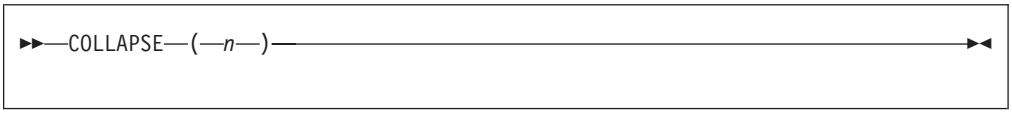
『collapse』

collapse

目的

collapse 文節を指定すると、ネストされた並列処理を導入せずに、ネスト内の複数のループを並列化することができます。この文節は、**for** および **parallel for** プラグマと一緒に使用します。

構文



規則

- 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。**independent_loop** プラグマは、**collapse** 文節に関連付けられたどのループにも使用できます。**independent_loop** は OpenMP に固有のものではありません。

関連資料

335 ページの『#pragma omp for』

340 ページの『#pragma omp parallel for』

#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。

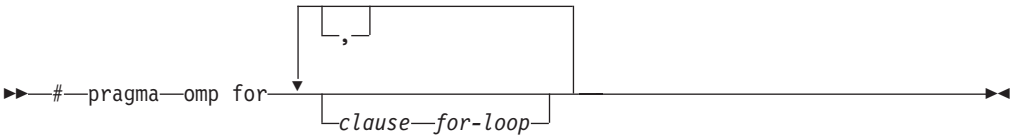
- **ordered** 節を含んでいる **omp for** または **omp parallel for** 構成の範囲内に表示されなければなりません。
- すぐ後に続くステートメント・ブロックに適用します。そのブロックのステートメントは、繰り返しが順次ループ内で実行されるのと同じ順序で実行されます。
- ループの繰り返しは、同一の **omp ordered** ディレクティブを 2 回以上実行してはなりません。
- ループの繰り返しでは、複数の特殊 **omp ordered** ディレクティブを実行してはなりません。

#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、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。

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、共用の縮約変数の元の値に戻されます。

reduction 節に指定される変数は以下の通りです。
- 演算子に適切な型でなければならない。
- 囲んでいるコンテキスト内で共用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。

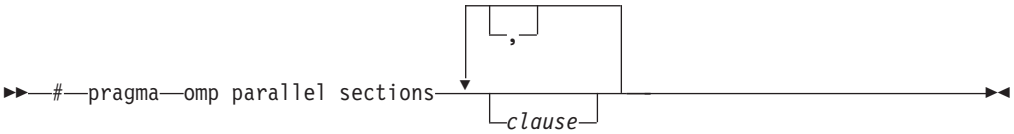
プログラム実行が **omp sections** ディレクティブに到達すると、後続の **omp section** ディレクティブによって定義されたプログラム・セグメントは、並列実行のために使用可能なスレッド間で配布されます。 **nowait** 節が指定されていない限り、バリアは **omp sections** ディレクティブに関連した、より広いプログラム領域の終了地点に暗黙的に定義されます。

#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

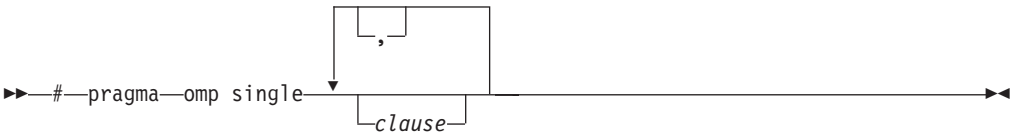
omp parallel および **omp sections** ディレクティブに記載されているすべての文節および制限は、 **omp parallel sections** ディレクティブに適用されます。

#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 が使用するネーム・スペースとは異なるネーム・スペースを占めます。

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、そのセクション内の他のすべてのスレッドが同じポイントに達するまで並列領域のスレッドが待機する同期点を識別します。**omp barrier** ポイントを過ぎたステートメントの実行は、その後、並列で続行します。

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。

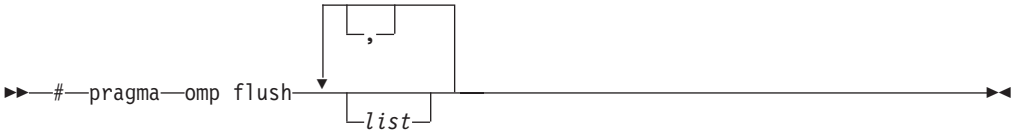
```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#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用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。

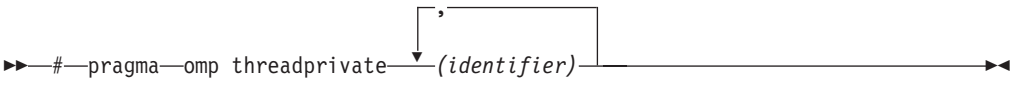
```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#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、ネーム・スペース・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、ネーム・スペース・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

OpenMP **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、ポインター追跡などの規格外のアルゴリズム、または他の OpenMP ワーク・シェアリング構文が不適当である再帰的アルゴリズムを並列処理する場合に役立ちます。 **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

clause は、次のいずれかです。

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。

default

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。

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用されることを宣言します。

関連資料

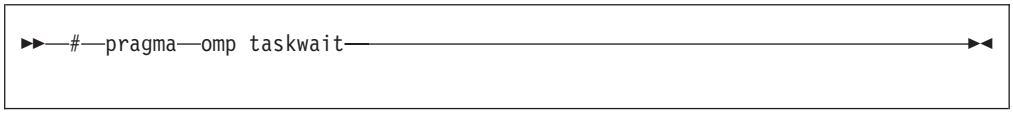
『#pragma omp taskwait』

#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

347 ページの『#pragma omp task』

第 5 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』
- 351 ページの『プラットフォームに関連したマクロ』
- 351 ページの『コンパイラー機能に関連したマクロ』

358 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 31. 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。
__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。
__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 ビットのコンパイル・モードの場合)。`unsigned long` (64 ビットのコンパイル・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。

表 31. 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
__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 コンパイラー製品を示すマクロ

XL 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。

表 32. 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。 XL C V10.1 では、マクロの値は 1010 です。
__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。 XL C V10.1 では、マクロの値は「10.1.0.0」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 33. 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。
__powerpc、 __powerpc__	ターゲット・アーキテクチャーが PowerPC であることを示します。	1	ターゲット・アーキテクチャーが PowerPC である場合に事前定義されます。
__PPC, __PPC__	ターゲット・アーキテクチャーが PowerPC であることを示します。	1	ターゲット・アーキテクチャーが PowerPC 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。

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。

- 『コンパイラー・オプション設定に関連したマクロ』
- 353 ページの『アーキテクチャー設定に関連したマクロ』
- 355 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

以下のマクロのソース入力特性、出力ファイル特性、最適化などのさまざまな機能についてテストできます。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 34. 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
__DEBUG_ALLOC__	標準メモリー管理関数のデバッグ・バージョンが使用されていることを示します。	1	-qheapdebug
__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	**-qasm=gcc** および **-qlanglvl=extc99** | **extc89** | **extended** または **-qkeyword=asm**
		0	**-qnoasm** および **-qlanglvl=extc99** | **extc89** | **extended** または **-qkeyword=asm**
__IBM_DFP__	10 進浮動小数点型がサポートされていることを示します。	1	-qdfp
__IBM_DFP_SW_EMULATION__	10 進浮動小数点の計算が、ハードウェア命令においてではなく、ソフトウェア・エミュレーションによって実装されていることを示します。	1	-qfloat=dfpemulate
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
__IBM_UTF_LITERAL	UTF-16 および UTF-32 ストリング・リテラルがサポートされていることを示します。	1	-qutf

表 34. 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。
__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 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
__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 35. **-qarch**-*related macros*

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_403	アプリケーションが、PowerPC 403 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。	403
_ARCH_604	アプリケーションが、PowerPC 604 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。	604
_ARCH_COM	アプリケーションが、すべての PowerPC 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_PPC	アプリケーションが、すべての PowerPC 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 PowerPC 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 | pwr3 | rs64b | rs64c | ppc64gr | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970

表 35. **-qarch**-*related macros (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_PPCGR	アプリケーションが、グラフィックスをサポートする PowerPC 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 | 604 | pwr3 | rs64b | rs64c | ppc64gr | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_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 PowerPC 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。	pwr3 | rs64b | rs64c | ppc64gr | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_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 PowerPC 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。	pwr3 | rs64b | rs64c | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_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 PowerPC 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 | ppc970 | pwr6 | pwr6e
_ARCH_PPC970	アプリケーションが、PowerPC 970 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。	ppc970
_ARCH_PWR3	アプリケーションが、POWER3 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。	pwr3 | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_ARCH_PWR4	アプリケーションが、POWER4 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。	pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_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
_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
_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
_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_RS64A	アプリケーションが、RS64I プロセッサーで実行するターゲットになっていることを示します。	rs64a
_ARCH_RS64B	アプリケーションが、RS64II プロセッサーで実行するターゲットになっていることを示します。	rs64b

表 35. **-qarch**-*related macros (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_RS64C	アプリケーションが、RS64III プロセッサーで実行するターゲットになっていることを示します。	rs64c

言語レベルに関連したマクロ

次のマクロは、C99 機能、GNU 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 ランゲージ・リファレンス*」を参照してください。

表 36. *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 __C99_BOOL	`_Bool` データ型がサポートされていることを示します。	stdc99 | extc99 | extc89 | extended
C __C99_COMPLEX	複素数データ型がサポートされていることを示します。	stdc99 | extc99 | extc89 | extended
__C99_COMPLEX_HEADER__	C99 型の複素数ヘッダーがサポートされていることを示します。	c99complexheader
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。	stdc99 | extc99 (および **-qcpluscmt**)
C __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。	stdc99 | extc99 | extc89 | extended
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。	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。	stdc99 | extc99 | extc89 | extended
__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。	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。	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 stdc99 | extc99 | extc89 | extended
__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。	stdc99 | extc99 | extc89 | extended
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。	stdc99 | extc99 (および **-qkeyword=inline**)

表 36. 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_LLONG	C99 型の long long データ型がサポートされていることを示します。	stdc99 | extc99
__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。	stdc99 | extc99 | extc89 | extended
__C99_MAX_LINE_NUMBER	最大行番号が 2147483647 であることを示します。	stdc99 | extc99 | extc89 | extended
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。	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。	stdc99 | extc99 | extc89 | extended
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。	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。	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。	stdc99 | extc99 | extc89 | extended
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 stdc99 | extc99 (および **-qkeyword=restrict**)
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。	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。	stdc99 | extc99 | extc89 | extended
C __C99_TGMATH	tgmath.h の型汎用マクロがサポートされていることを示します。	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 stdc99 | extc99 | 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。	stdc99 | extc99 | extc89 | extended
__DIGRAPHS__	ダイグラフがサポートされていることを示します。	C stdc99 | extc99 | extc89 | extended (および **-qdigraph**)

表 36. 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。
__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
__IBM__ALIGN	__align 指定子がサポートされていることを指定します。	C++ **-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__	__alignof__ 演算子がサポートされていることを示します。	extc99 | extc89 | 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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
__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。	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。	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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
_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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended
__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。	extc99 | extc89 | extended

表 36. 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_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。
_LONG_LONG	IBM long long データ型がサポートされていることを示します。	C extended | extc89 (および -qlonglong)
C __SAA__	SAA C 標準の最新レベルをサポートする言語構造体のみが許可されていることを示します。	saa
C __SAA_L2__	SAA レベル 2 C 標準に準拠する言語構造体のみが許可されていることを示します。	saal2
__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。	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。
__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。)	これが該当する場合は 1 に事前定義され、そうでない場合は 0 に定義されています。
__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
```

第 6 章 コンパイラー組み込み関数

組み込み関数は 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 PowerPC 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 PowerPC 命令へのアクセスは、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 およびその他のコンパイラーでは完全にサポートされていません。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 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 PowerPC 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。

このコンパイラーは、AltiVec 指定によって定義されたベクトル処理関数のすべてをサポートします。 上記のすべての組み込み関数について詳しくは、http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf で入手可能な「*AltiVec Technology Programming Interface Manual*」を参照してください。

以下のセクションには、AIX 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

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。

- 366 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);
```

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);
```

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*);
```

乗算関数

__imul_dbl

目的

2 つの `long` 整数の積を計算し、その結果をポインターに保管します。

プロトタイプ

```
void __imul_dbl (long, long, long*);
```

__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 または 4 バイト・レジスターを保管します。

プロトタイプ

void __store2r(unsigned short, unsigned short *);

void __store4r(unsigned int, unsigned int *);

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。

バイナリー浮動小数点組み込み関数

バイナリー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。

- 362 ページの『絶対値関数』
- 368 ページの『加算関数』
- 368 ページの『変換関数』
- 370 ページの『FPSCR 関数』
- 373 ページの『乗算関数』
- 373 ページの『乗算-加算/減算関数』
- 374 ページの『逆数見積もり関数』
- 374 ページの『丸め関数』
- 376 ページの『選択関数』
- 376 ページの『平方根関数』

- 377 ページの『ソフトウェア除算関数』

10 進浮動小数点組み込み関数の場合、10 進浮動小数点組み込み関数を参照してください。

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);
```

加算関数

__fadd、__fadds

目的

浮動追加、単精度浮動追加。

2 つの引数を加算して、結果を戻します。

プロトタイプ

```
double __fadd (double, double);

float __fadds (float, 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);
```

__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);
```

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 - 最も近い値への丸め
- 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

__dfp_set_rounding_mode

目的

丸めモードを設定します。

現行 10 進丸めモードを設定します。

プロトタイプ

```
void __dfp_set_rounding_mode (unsigned long rounding_mode);
```

パラメーター

rounding_mode
389 ページの表 38にリストされたコンパイル時定数値 (0 から 7) またはマクロのいずれか。

使用法

関数内で丸めモードを変更する場合、関数が戻る前に丸めモードを復元する必要があります。

__dfp_get_rounding_mode

目的

丸めモードを取得します。

現行の 10 進丸めモードを取得します。

プロトタイプ

```
unsigned long __dfp_get_rounding_mode (void);
```

戻り値

389 ページの表 38にリストされた値 (0 to 7) のいずれかとしての現行丸めモード。

乗算関数

__fmul、__fmuls

目的

浮動乗算、単精度浮動乗算。

2 つの引数を乗算して、結果を戻します。

プロトタイプ

```
double __fmul (double, double);

float __fmuls (float, float);
```

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

376 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

```
float __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

__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);
```

10 進浮動小数点の組み込み関数

10 進浮動小数点 (DFP) の組み込み関数は、次のカテゴリーにグループ分けされます。

- 『絶対値関数』
- 379 ページの『係数関数』
- 380 ページの『比較関数』
- 381 ページの『変換関数』
- 386 ページの『指数関数』
- 387 ページの『NaN 関数』
- 388 ページの『レジスター転送関数』
- 389 ページの『丸め関数』
- 391 ページの『テスト関数』

2 進浮動小数点の組み込み関数については、『 2 進浮動小数点の組み込み関数』を参照してください。

-qarch が POWER6 プロセッサーをターゲットにするよう pwr6 または pwr6e に設定されている場合、**-qfloat=nodfpemulate** がデフォルトになります。これは、DFP ハードウェア命令が生成されることを意味します。低パフォーマンス・ソフトウェア・エミュレーション・コードが生成されるのは、以下の場合のみです。

- **-qarch** が pwr6 または pwr6e に設定されていない場合。
- **-qarch** が pwr6 または pwr6e に設定され、**-qfloat=dfpemulate** が有効な場合。

絶対値関数

絶対値関数は、戻り値の符号を決定します。

__d64_abs、__d128_abs

目的

絶対値

パラメーターの絶対値を戻します。

プロトタイプ

_Decimal64 __64_abs (_Decimal64);

_Decimal128 __d128_abs (_Decimal128);

__d64_nabs、__d128_nabs
目的

負の絶対値

パラメーターの負の絶対値を戻します。

プロトタイプ

_Decimal64 __d64_nabs (_Decimal64);

_Decimal128 __d128_nabs (_Decimal128);

__d64_copysign、__d128_copysign
目的

コピー符号

最初のパラメーターの絶対値を戻します。このとき 2 つ目のパラメーターの符号が付いています。

プロトタイプ

_Decimal64 __d64_copysign (_Decimal64、_Decimal64);

_Decimal128 __d128_copysign (_Decimal128、_Decimal128);

係数関数

係数関数は、10 進浮動小数点変換ライブラリー関数をサポートするために、指数と符号部分に影響を与えることなく小数部分を操作します。

__d64_shift_left、__d128_shift_left
目的

係数を左方にシフトします。

パラメーターの係数を左方にシフトします。

プロトタイプ

_Decimal64 __d64_shift_left (_Decimal64、unsigned long *digits*);

_Decimal128 __d128_shift_left (_Decimal128、unsigned long *digits*);

パラメーター

digits
左方にシフトする桁数です。シフト・カウントは、0 から 63 までの範囲でなければなりません。この範囲外だと結果が定義されません。

戻り値

符号および指数は同じままです。指定の桁数が左方にシフトされます。

__d64_shift_right、__d128_shift_right

目的

係数を右方にシフトします。

パラメーターの係数を右方にシフトします。

プロトタイプ

_Decimal64 __d64_shift_right (_Decimal64、unsigned long *digits*);

_Decimal128 __d128_shift_right (_Decimal128、unsigned long *digits*);

パラメーター

digits
右方にシフトする桁数です。シフト・カウントは、0 から 63 までの範囲でなければなりません。この範囲外だと結果が定義されません。

戻り値

符号および指数は同じままです。指定した桁数が右方にシフトされます。

比較関数

比較関数は、拡張例外処理と指数比較をサポートします。

__d64_compare_exponents、__d128_compare_exponents

目的

指数を比較します。

2 つの 10 進数浮動小数点値の指数を比較します。

プロトタイプ

long __d64_compare_exponents (_Decimal64、_Decimal64);

long __d128_compare_exponents (_Decimal128、_Decimal128);

戻り値

以下の値を戻します。

- 1 つ目のパラメーターの指数が、2 つ目のパラメーターの指数より小さい場合は、0 より小さい値を戻す。

- 両方のパラメーターにおいて、指数値が同じである場合、両方とも静止 NaN やシグナリング NaNs (静止およびシグナリングは同等と認識される) のいずれかである場合、あるいは両方が無限大である場合は、0 を戻す。
- 1 つ目の引数の指数が、2 つ目の引数の指数より大きい場合は、0 より大きい数値を戻す。
- 2 つのパラメーターのうち、一方が静止 NaN またはシグナリング NaN のいずれか、あるいは 一方が無限大である場合は、-2 を戻す。

__d64_compare_signaling、__d128_compare_signaling

目的

NaN に対するシグナリング例外を比較します。

2 つの 10 進数浮動小数点値を比較し、いずれも静止 NaN またはシグナリング NaN でない場合は、無効な演算例外が発生します。

プロトタイプ

long __d64_compare_signaling (_Decimal64、_Decimal64);

long __d128_compare_signaling (_Decimal128、_Decimal128);

戻り値

以下の値を戻します。

- 1 つ目のパラメーターの値が、2 つ目のパラメーターの値より小さい場合は、0 より小さい値を戻す。
- 両方のパラメーターの値が同じ場合は、0 を戻す。
- 1 つ目のパラメーターの値が、2 つ目のパラメーターの値より大きい場合は、0 より大きい値を戻す。

いずれの値も静止 NaN またはシグナリング NaN でない場合は、例外が発生します。例外ハンドラーが例外をトラップするように有効になっていないと、関数は -2 を戻します。

使用法

いずれか一方の値が NaN である場合、関係演算子 (==、!=、<、<=、> および >=) を使用した通常の比較では常に false が戻され、静止 NaN ではなくシグナリング NaN に対して例外が発生します。いずれか一方の値が静止 NaN またはシグナリング NaN の場合に例外が発生するようにするには、関係演算子の代わりに NaN 組み込み関数でシグナリング例外の比較を使用します。

変換関数

変換関数は、10 進浮動小数点の変換を実行します。その一部は、現行の丸めモードをオーバーライドします。

__d64_to_long_long、__d128_to_long_long

目的

整数に変換します。

10 進数浮動小数点値を、現行の丸めモードを使用して 64 ビットの符号付き 2 進整数に変換します。

プロトタイプ

long long __d64_to_long_long (_Decimal64);

long long __d128_to_long_long (_Decimal128):

戻り値

現行の丸めモードを使用して、long long 型に変換された入力値 (キャスト変換または暗黙変換と同様、常に 0 に丸められない)。

__d64_to_long_long_rounding、__d128_to_long_long_rounding

目的

整数に変換します。

10 進数浮動小数点値を、指定した丸めモードを使用して 64 ビットの符号付き 2 進整数に変換します。

プロトタイプ

long long __d64_to_long_long_rounding (_Decimal64、long *rounding_mode*);

long long __d128_to_long_long_rounding (_Decimal128、long *rounding_mode*);

パラメーター

mode
コンパイル時間の定数値または 389 ページの表 38 で定義されたマクロの 1 つです。

戻り値

指定した丸めモードを使用して、long long 型に変換された入力値 (キャスト変換または暗黙変換と同様、常に 0 に丸められない)。

使用法

これらの関数は、現行の操作で使用中の丸めモードを一時的に無効にします。

__d64_to_signed_BCD

目的

符号付きの 2 進コード化された 10 進数に変換します。

64 ビットの 10 進数浮動小数点値の下桁を、符号付きパック・フォーマット (パック 10 進数) に変換します。

プロトタイプ

unsigned long long __d64_to_signed_BCD (_Decimal64、_Bool *value*);

戻り値

16 ビットの結果での 10 進数の符号の前に、15 の 10 進数字を作成します。左端の桁は無視されます。

正の値に対して、*value* が TRUE であれば 0xF が、*value* が FALSE であれば 0xC の符号が適用されます。

負の値には符号 0xD が適用されます。

使用法

左端の桁にアクセスするには、`__d64_shift_right` 関数を使用します。

_d128_to_signed_BCD
目的

符号付きの 2 進コード化された 10 進数に変換します。

128 ビットの 10 進数浮動小数点値の下桁を、符号付きパック・フォーマット (パック 10 進数) に変換します。

プロトタイプ

void __d128_to_signed_BCD (_Decimal128、_Bool *value*、unsigned long long **upper*, unsigned long long **lower*);

パラメーター

upper
結果の上桁を保持する変数のアドレスです。

lower
結果の下桁を保持する変数のアドレスです。

戻り値

128 ビットの結果での 10 進数の符号の前に、31 の 10 進数字を作成します。左方の桁は無視されます。上 16 桁は、パラメーター *upper* に格納されます。下 15 桁および符号は、パラメーター *lower* に格納されます。

正の値に対して、*value* が TRUE であれば 0xF が、*value* が FALSE であれば 0xC の符号が適用されます。

負の値には符号 0xD が適用されます。

使用法

左方の桁にアクセスするには、__d128_shift_right 関数を使用します。

__d64_to_unsigned_BCD
目的

符号なしの 2 進コード化された 10 進数に変換します。

64 ビットの 10 進数浮動小数点値の下桁を、符号なしパック・フォーマットに変換します。

プロトタイプ

unsigned long long __d64_to_unsigned_BCD (_Decimal64);

戻り値

64 ビットの結果で、符号なしの 16 の 10 進数字を返します。

使用法

左方の桁にアクセスするには、__d64_shift_right 関数を使用します。

__d128_to_unsigned_BCD
目的

符号なしの 2 進コード化された 10 進数に変換します。

128 ビットの 10 進数浮動小数点値の下桁を、符号なしパック・フォーマットに変換します。

プロトタイプ

void __d128_to_unsigned_BCD (_Decimal128, unsigned long long *upper, unsigned long long *lower);

パラメーター

upper
結果の上桁を保持する変数のアドレスです。

lower
結果の下桁を保持する変数のアドレスです。

戻り値

128 ビットの結果で、符号なしの 32 の 10 進数字を返します。左方の桁は無視されます。上 16 桁は、パラメーター *upper* に格納されます。下 16 桁は、パラメーター *lower* に格納されます。

使用法

左方の桁にアクセスするには、__d128_shift_right 関数を使用します。

__signed_BCD_to_d64

目的

符号付きの 2 進コード化された 10 進数から変換します。

64 ビットの符号付きパック・フォーマット (パック 10 進数 - 10 進数符号が付いた 15 の 10 進数) を 64 ビットの 10 進数浮動小数点値に変換します。

プロトタイプ

```
_Decimal64 __signed_BCD_to_d64 (unsigned long long);
```

パラメーター

符号 0xA、0xC、0xE、および 0xF は正の値として処理されます。符号 0xB および 0xD は負の値として処理されます。

__signed_BCD_to_d128

目的

符号付きの 2 進コード化された 10 進数から変換します。

128 ビットの符号付きパック・フォーマット (パック 10 進数 - 10 進数符号が付いた 31 の 10 進数) を 128 ビットの 10 進数浮動小数点値に変換します。

プロトタイプ

```
_Decimal128 __signed_BCD_to_d128 ( unsigned long long upper, unsigned long long lower);
```

パラメーター

upper
入力値の上 16 桁です。

lower
入力値の下 15 桁および符号です。

パラメーター

符号 0xA、0xC、0xE、および 0xF は正の値として処理されます。符号 0xB および 0xD は負の値として処理されます。

__unsigned_BCD_to_d64

目的

符号なしの 2 進コード化された 10 進数から変換します。

64 ビットの符号なしパック・フォーマット (符号なしの 16 の 10 進数) を 64 ビットの 10 進数浮動小数点値に変換します。

プロトタイプ

```
_Decimal64 __unsigned_BCD_to_d64 (unsigned long long);
```

__unsigned_BCD_to_d128

目的

符号なしの 2 進コード化された 10 進数から変換します。

128 ビットの符号なしパック・フォーマット (符号なしの 32 の 10 進数) を 128 ビットの 10 進数浮動小数点値に変換します。

プロトタイプ

_Decimal128 __unsigned_BCD_to_d128 (unsigned long long *upper*, unsigned long long *lower*);

パラメーター

upper
入力値の上 16 桁です。

lower
入力値の下 16 桁です。

指数関数

指数関数は、主に 10 進浮動小数点変換ライブラリー関数をサポートするため、値から指数を抽出したり、値に指数を挿入したりします。これらは特殊値を使用して、指数タイプを識別または指定します。

表 37. バイアス指数のマクロと値

マクロ	整数値
DFP_BIASED_EXPONENT_FINITE	0
DFP_BIASED_EXPONENT_INFINITY	-1
DFP_BIASED_EXPONENT_QNAN	-2
DFP_BIASED_EXPONENT_SNAN	-3

__d64_biased_exponent, __d128_biased_exponent

目的

バイアス指数を抽出します。

10 進数浮動小数点値の指数を整数として戻します。

プロトタイプ

long __d64_biased_exponent (_Decimal64);

long __d128_biased_exponent (_Decimal128);

戻り値

表 37 のリストに従って、無限大、静止 NaN、およびシグナリング NaN の特殊値を戻します。

有限値の場合、結果は DFP_BIASED_EXPONENT_FINITE、指数バイアス (_Decimal64 の場合は 398、_Decimal128 の場合は 6176)、および実際の指数です。

__d64_insert_biased_exponent、__d128_insert_biased_exponent

目的

バイアス指数を挿入します。

10 進数浮動小数点値の指数を置き換えます。

プロトタイプ

_Decimal64 __d64_insert_biased_exponent (_Decimal64, long *exponent*);

_Decimal128 __d128_insert_biased_exponent (_Decimal128, long *exponent*);

パラメーター

exponent

1 つ目のパラメーターに適用する指数値です。無限大、静止 NaN、およびシグナリング NaN の場合、コンパイル時間の定数値または 386 ページの表 37にリストされたマクロの 1 つを使用します。

有限値の場合、結果は DFP_BIASED_EXPONENT_FINITE、指数バイアス (_Decimal64 の場合は 398、_Decimal128 の場合は 6176)、および望ましい指数です。

NaN 関数

NaN 関数は、静止またはシグナリング NaN を作成します。

__d32_sNaN、__d64_sNaN、__d128_sNaN

目的

シグナリング NaN を作成します。

指定した精度のシグナリング NaN を、正の符号を付けてゼロのペイロードで作成します。

プロトタイプ

_Decimal32 __d32_sNan (void);

_Decimal64 __d64_sNaN (void);

_Decimal128 __d128_sNaN (void);

__d32_qNaN、__d64_qNaN、__d128qNaN

目的

静止 NaN を作成します。

指定した精度の静止 NaN を、正の符号を付けてゼロのペイロードで作成します。

プロトタイプ

_Decimal32 __d32_qNaN (void);

_Decimal64 __d64_qNaN (void);

_Decimal128 __d128_qNaN (void);

レジスター転送関数

レジスター転送関数は、汎用レジスターと浮動小数点レジスターの間でデータを転送します。変換は行われません。レジスター転送関数は、浮動小数点レジスターでは整数データを処理し、汎用レジスターでは浮動小数点データを処理します。これらの関数は、POWER6e (ロー) モードで実行されている POWER6において、**-qarch=pwr6** または **-qarch=pwr6e** のみで使用可能な命令を使用します。

__gpr_to_d64

目的

汎用レジスターから浮動小数点レジスターに転送します。

汎用レジスター (64 ビット・モード) または汎用レジスター・ペア (32 ビット・モード) から値を転送します。

プロトタイプ

_Decimal64 __gpr_to_d64 (long long);

__gprs_to_d128

目的

汎用レジスターから浮動小数点レジスターに転送します。

汎用レジスター (64 ビット・モード) のペアまたは 4 つの汎用レジスター (32 ビット・モード) から値を転送します。

プロトタイプ

_Decimal128 __gprs_to_d128 (unsigned long long*_upper_, unsigned long long*_lower_);

パラメーター

upper
結果の上部の 64 ビットを保持する変数のアドレスです。

lower
結果の下部の 64 ビットを保持する変数のアドレスです。

戻り値

上部の 64 ビットはパラメーター *upper* に格納されます。下部の 64 ビットは、パラメーター *lower* に格納されます。

__d64_to_gpr

目的

浮動小数点レジスターから汎用レジスターに転送します。

浮動小数点レジスターから、汎用レジスター (64 ビット・モード) または汎用レジスター・ペア (32 ビット・モード) に値を転送します。

プロトタイプ

long long __d64_to_gpr (_Decimal64);

__d128_to_gprs

目的

浮動小数点レジスターから汎用レジスターに転送します。

浮動小数点レジスターのペアから、汎用レジスター (64 ビット・モード) のペア、または 4 つの汎用レジスター (32 ビット・モード) に値を転送します。

プロトタイプ

void __d128_to_gprs (_Decimal128, unsigned long long**upper*, unsigned long long**lower*);

パラメーター

upper
入力値の上部の 64 ビットを保持する変数のアドレスです。

lower
入力値の下部の 64 ビットを保持する変数のアドレスです。

丸め関数

丸め関数は、浮動小数点値の丸めや切り捨てなどの演算を実行します。

表 38. 丸めモードのマクロと値

マクロ	整数値
DFP_ROUND_TO_NEAREST_WITH_TIES_TO_EVEN	0
DFP_ROUND_TOWARD_ZERO	1
DFP_ROUND_TOWARD_POSITIVE_INFINITY	2
DFP_ROUND_TOWARD_NEGATIVE_INFINITY	3
DFP_ROUND_TO_NEAREST_WITH_TIES_AWAY_FROM_ZERO	4
DFP_ROUND_TO_NEAREST_WITH_TIES_TOWARD_ZERO	5
DFP_ROUND_AWAY_FROM_ZERO	6
DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION	7
DFP_ROUND_USING_CURRENT_MODE[1]	8

注:

1. この値が有効なのは、現行の丸めモードをオーバーライドする関数のみです。__dfp_set_rounding_mode には有効でなく、__dfp_get_rounding_mode を使用しても戻すことはできません。

__d64_integral、__d128_integral

目的

整数に丸めます。

10 進数浮動小数点値を整数に丸めるため、不正確な例外が発生します。

プロトタイプ

```
_Decimal64 __d64_integral (_Decimal64);

_Decimal128 __d128_integral (_Decimal128);
```

戻り値

現行の丸めモードで丸めた 10 進数浮動小数点フォーマットで整数が戻されます。小数点以下の桁は破棄されます。

__d64_integral_no_inexact、__d128_integral_no_inexact

目的

整数に丸めます。

10 進数浮動小数点値を整数に丸め、不正確な例外の発生を回避します。

プロトタイプ

```
_Decimal64 __d64_integral_no_inexact (_Decimal64);

_Decimal128 __d128_integral_no_inexact (_Decimal128);
```

戻り値

現行の丸めモードで丸めた 10 進数浮動小数点フォーマットで整数が戻されます。小数点以下の桁は破棄されます。

__d64_quantize、__d128_quantize

目的

量子化します。

最初のパラメーターの演算値を戻します。このとき、指定した丸めモードを使用して、指数が 2 つ目のパラメーターと同等になるよう調整されます。

プロトタイプ

```
_Decimal64 __d64_quantize (_Decimal64, _Decimal64, long rounding_mode);

_Decimal128 __d128_quantize (_Decimal128, _Decimal128, long rounding_mode);
```

パラメーター

rounding_mode
コンパイル時間の定数値または 389 ページの表 38で定義されたマクロの 1 つです。

使用法

これらの関数は、現行の操作で使用中の丸めモードを一時的に無効にします。

__d64_reround、__d128_reround
目的

丸め直し

部分的に丸められた値を完全に丸めることで、エラーの原因となる二重丸めを回避します。

プロトタイプ

_Decimal64 __d64_reround (_Decimal64, unsigned long *number_of_digits*, unsigned long *rounding_mode*);

_Decimal128 __d128_reround (_Decimal128, unsigned long *number_of_digits*, unsigned long *rounding_mode*);

パラメーター

number_of_digits
丸める桁数です。__d64_reround の場合は 1 から 15、__d128_reround の場合 1 から 33 です。

rounding_mode
コンパイル時間の定数値または 389 ページの表 38で定義されたマクロの 1 つです。

使用法

これらの関数は、現行の操作で使用中の丸めモードを一時的に無効にします。丸める値は、正確に丸められるよう、丸めモード DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION または 7 を使用して既に丸めてある必要があります。

テスト関数

テスト関数を使用して、主に数学ライブラリー関数をサポートするために、無効な結果の拡張例外処理または入力値のカテゴリー化を行うことができます。

__d64_is または __d128_ で始まる関数を使用すると、シグナリング NaN に対してさえも例外が発生しません。

表 39. テスト・データ・クラス・マスクのマクロと値

マクロ	整数値
DFP_PPC_DATA_CLASS_ZERO	0x20

表 39. テスト・データ・クラス・マスクのマクロと値 (続き)

マクロ	整数値
DFP_PPC_DATA_CLASS_SUBNORMAL	0x10
DFP_PPC_DATA_CLASS_NORMAL	0x08
DFP_PPC_DATA_CLASS_INFINITY	0x04
DFP_PPC_DATA_CLASS_QUIET_NAN	0x02
DFP_PPC_DATA_CLASS_SIGNALING_NAN	0x01

表 40. テスト・データ・グループ・マスクのマクロと値

マクロ	整数値
DFP_PPC_DATA_GROUP_SAFE_ZERO	0x20
DFP_PPC_DATA_GROUP_ZERO_WITH_EXTREME_EXPONENT	0x10
DFP_PPC_DATA_GROUP_NONZERO_WITH_EXTREME_EXPONENT	0x08
DFP_PPC_DATA_GROUP_SAFE_NONZERO	0x04
DFP_PPC_DATA_GROUP_NONZERO_LEFTMOST_DIGIT_NONEXTREME_EXPONENT	0x02
DFP_PPC_DATA_GROUP_SPECIAL	0x01

表 41. テスト・データ・クラスおよびグループ結果のマクロと値

マクロ	整数値
DFP_PPC_DATA_POSITIVE_NO_MATCH	0x00
DFP_PPC_DATA_POSITIVE_MATCH	0x02
DFP_PPC_DATA_NEGATIVE_NO_MATCH	0x08
DFP_PPC_DATA_NEGATIVE_MATCH	0x0A

表 42. テスト・データ・クラスおよびグループの結果のマスク・マクロと値

マクロ	整数値
DFP_PPC_DATA_NEGATIVE_MASK	0x08
DFP_PPC_DATA_MATCH_MASK	0x02

__d64_same_quantum、__d128_same_quantum

目的

同じ量子

2 つの値の量子が同じ場合、TRUE を戻します。

プロトタイプ

```
_Bool __d64_same_quantum (_Decimal64, _Decimal64);

_Bool __d128_same_quantum (_Decimal28, _Decimal128);
```

__d64_issigned、__d128_issigned

目的

符号付き

パラメーターが負、負のゼロ、負の無限大、または負の NaN のいずれかである場合、TRUE を戻します。

プロトタイプ

```
_Bool __d64_issigned (_Decimal64);

_Bool __d128_issigned (_Decimal128);
```

__d64_isnormal、__d128_isnormal

目的

標準

パラメーターが標準範囲 (正常以下、無限大、または NaN でない) 内にありゼロでない場合、TRUE を戻します。

プロトタイプ

```
_Bool _d64_isnormal (_Decimal64);

_Bool _d128_isnormal (_Decimal128);
```

__d64_isfinite、__d128_isfinite

目的

有限

パラメーターが正または負の無限大でなく、静止またはシグナリング NaN でない場合、TRUE を戻します。

プロトタイプ

```
_Bool __d64_isfinite (_Decimal64);

_Bool __d128_isfinite (_Decimal128);
```

__d64_iszero、__d128_iszero

目的

ゼロ

パラメーターが正または負のゼロである場合、TRUE を戻します。

プロトタイプ

_Bool __d64_iszero (_Decimal64);

_Bool __d128_iszero (_Decimal128);

__d64_issubnormal、__d128_issubnormal
目的

標準以下

パラメーターが標準以下である場合、TRUE を戻します。

プロトタイプ

_Bool _d64_issubnormal (_Decimal64);

_Bool _d128_issubnormal (_Decimal128);

__d64_isinf、__d128_isinf
目的

無限大

パラメーターが正または負の無限大である場合、TRUE を戻します。

プロトタイプ

_Bool __d64_isinf (_Decimal64);

_Bool __d128_isinf (_Decimal128);

__d64_isnan、__d128_isnan
目的

NaN

パラメーターが正または負の静止やシグナリング NaN である場合、TRUE を戻します。

プロトタイプ

_Bool __d64_isnan (_Decimal64);

_Bool __d128_isnan (_Decimal128);

__d64_issignaling、__d128_issignaling
目的

シグナリング NaN

パラメーターが正または負のシグナリング NaN である場合、TRUE を戻します。

プロトタイプ

_Bool __d64_issignaling (_Decimal64);

_Bool __d128_issignaling (_Decimal128);

__d64_test_data_class、__d128_test_data_class

目的

テスト・データ・クラス

値がゼロ、標準以下、標準、無限大、静止 NaN、またはシグナリング NaN のいずれであるか、および値が正または負であるかを報告します。

プロトタイプ

long __d64_test_data_class (_Decimal64, unsigned long *mask*);

long __d128_test_data_class (_Decimal128, unsigned long *mask*);

パラメーター

mask
391 ページの表 39 で定義された値またはマクロの 1 つ、または複数が論理和演算されたものです。 パラメーターは、コンパイル時間の定数式でなければなりません。

戻り値

392 ページの表 41にリストされている値の 1 つです。

使用法

適切なマスクを使用すると、同時に値の組み合わせを確認できます。 391 ページの表 39にリストされるマスクを使用すると、入力値を確認できます。 392 ページの表 42にリストされるマスクを使用すると、結果値を確認できます。

__d64_test_data_group、__d128_test_data_group

目的

テスト・データ・グループ

値が安全なゼロ、極度指数を含むゼロ、標準以下、安全な非ゼロ、先行ゼロなしの標準、または無限大や NaN のいずれかであるか、および値が正または負であるかを報告します。「安全な」は、先行ゼロ付きの桁および極端でない指数を意味します。標準以下は、極端な非ゼロまたは安全な非ゼロのいずれかとして表示されます。一部のマスクの正確な意味は、特定の CPU モデルによって異なります。

プロトタイプ

long _d64_test_data_group (_Decimal64, unsigned long *mask*);

long _d128_test_data_group (_Decimal128, unsigned long *mask*);

パラメーター

mask
392 ページの表 40で定義された値またはマクロの 1 つ、または複数が論理和演算されたものです。 パラメーターは、コンパイル時間の定数式でなければなりません。

戻り値

392 ページの表 41にリストされている値の 1 つです。

使用法

適切なマスクを使用すると、同時に値の組み合わせを確認できます。 392 ページの表 40にリストされるマスクを使用すると、入力値を確認できます。 392 ページの表 42にリストされるマスクを使用すると、結果値を確認できます。

__d64_test_significance、__d128_test_significance

目的

重要度をテストします。

10 進数浮動小数点値に、重要な指定桁数が含まれるかを確認します。

プロトタイプ

long __d64_test_significance (_Decimal64, unsigned long *digits*);

long __d128_test_significance (_Decimal128, unsigned long *digits*);

パラメーター

digits
テスト対象となる重要な桁の数です。*digits* は、0 から 63 までの範囲でなければなりません。この範囲外だと結果が定義されません。桁数が 0 だと、ゼロを含むすべての値には有効数字がより多く含まれると認識され、桁数が 0 でなければ、ゼロ値にはより少ない有効数字が含まれると認識されます。

戻り値

以下の値を戻します。

- 1 つ目のパラメーターの有効数字の桁数が 2 つ目のパラメーターより小さい場合は、0 より小さい値を戻す。
- 有効数字の桁数が 2 つ目のパラメーターと同じ場合は、0 を戻す。
- 1 つ目のパラメーターの有効数字の桁数が 2 つ目のパラメーターより大きい場合、または桁数が 0 の場合は、0 より大きい値を戻す。
- いずれか一方のパラメーターが静止 NaN かシグナリング NaN、または正の無限大か負の無限大の場合、-2 を戻す。

これらの関数では、値が 0 の場合、有効数字はゼロと認識されます。

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』
- 398 ページの『ロック・クリア関数』
- 399 ページの『比較および交換関数』
- 400 ページの『フェッチ関数』
- 402 ページの『ロード関数』
- 403 ページの『保管関数』
- 403 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* *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、*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用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用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用されており、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

__ldarx、__lwarx

目的

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。

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

```
long __ldarx (volatile long* addr);

int __lwarx (volatile int* 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 バイト境界上で位置合わせしなければなりません。

使用法

この関数は、後続の `__stdcx` (または `__stwcx`) 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 をインプリメントすることができます。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 `__ldarx` 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 (`__fence` 組み込み関数について詳しくは、411 ページの『__alignx』 を参照してください)。

__ldarx は 64 ビット・モードでのみ有効です。

保管関数

__stdcx、__stwcx

目的

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。

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 __stdcx(volatile long* addr, long val);

int __stwcx(volatile int* addr, int 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。

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 __ldarx (または __lwarx) 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 をインプリメントすることができます。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。

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用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、__iospace_sync

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);

void __iospace_sync (void);
```

__lwsync、__iospace_lwsync

目的

ワードのロードを同期化します。

__lwsync への呼び出しの前の保管命令のすべてが、関数を実行したプロセッサーで新規の命令が実行される前に完了するように保証します。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

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);
```

キャッシュ関連の組み込み関数

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。

- 『データ・キャッシュ関数』
- 406 ページの『プリフェッチ関数』
- 407 ページの『保護されたストリーム関数』

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 がターゲット POWER6 プロセッサーに設定されるときのみ有効です。

__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*);

__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*);

プリフェッチ関数

__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*);
```

保護されたストリーム関数

__protected_store_stream_set、__protected_unlimited_store_stream_set

目的

限定または非限定の長さの protected 保管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*);

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
0 から 15 の値を持つ整数。

使用法

-qarch がターゲット POWER6 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
0 から 15 の値を持つ整数。

使用法

-qarch が POWER5 または POWER6 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。

__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
0 から 15 の値を持つ整数。

使用法

-qarch がターゲット POWER6 プロセッサーに設定されるときのみ有効です。

__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 または POWER6 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。

__protected_stream_set、 __protected_unlimited_stream_set, __protected_unlimited_stream_set_go

目的

限定または非限定の長さの 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_stream_set (unsigned int direction, const void* addr, unsigned int stream_ID);

void _protected_unlimited_stream_set (unsigned int direction, const void* addr, unsigned int ID);

void __protected_unlimited_stream_set_go (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
0 から 15 の値を持つ整数。

使用法

-qarch が POWER5 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 `__protected_stream_set` および `__protected_unlimited_stream_set_go` は有効です。**-qarch** が POWER5 または POWER6 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 `_protected_unlimited_stream_set` は有効です。

__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);
```

使用法

-qarch が POWER5 または POWER6 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。

__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 または POWER6 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。

ブロックに関連した組み込み関数

__bcopy

目的

ブロック・コピー。

プロトタイプ

```
void __bcopy (char*, char*, size_t);
```

__bzero

目的

ブロック・ゼロ。

プロトタイプ

```
void __bzero (char*, int);
```

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』
- 412 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』
- 414 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

alignment
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 63 ビットの内容を、MSR に移動します。

プロトタイプ

```
void __mtmsr (unsigned long);
```

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。

関連情報
- 388 ページの『レジスター転送関数』

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。

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。

- 『IBM SMP 組み込み関数』
- 416 ページの『OpenMP 組み込み関数』

IBM SMP 組み込み関数

__parthds
目的

parthds 実行時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、選択された実行時オプションに関係なく、1 を戻します。

__usrthds
目的

usrthds 実行時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かった場合は、選択された実行時オプションに関係なく、0 を戻します。

OpenMP 組み込み関数

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・ライブラリー関数についての完全な情報は、www.openmp.org の OpenMP C/C++ アプリケーション・プログラム・インターフェースの仕様を参照してください。

関連情報
- 28 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取得します。 *max-active-levels-var* は、*OMP_MAX_ACTIVE_LEVELS* 環境変数または omp_set_max_active_levels 関数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設定します。 *max-active-levels-var* は、*OMP_MAX_ACTIVE_LEVELS* 環境変数を使用して設定することもできます。*max-active-levels-var* の値を取得するには、omp_get_max_active_levels 関数を使用します。

プロトタイプ

```
void omp_set_max_active_levels(int max_levels);
```

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。

modifier

スケジュール・タイプが dynamic、guided、および 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

32 ページの『OMP_SCHEDULE=*algorithm* 環境変数』

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・タイプが dynamic、guided、および static の場合、modifier は設定するチャンク・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

416 ページの『omp_get_schedule』

関連情報

32 ページの『OMP_SCHEDULE=*algorithm* 環境変数』

omp_get_thread_limit

目的

thread-limit-var 内部制御変数の値を取得します。 *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

omp_get_level では、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。

プロトタイプ

```
int  omp_get_level(void);
```

omp_get_ancestor_thread_num
目的

omp_get_ancestor_thread_num では、指定されたネスト・レベルの上位スレッドの現行レベル・スレッド番号が戻されます。 **omp_get_ancestor_thread_num** 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 を戻します。

プロトタイプ

```
int  omp_get_ancestor_thread_num(int  level);
```

omp_get_team_size
目的

omp_get_team_size では、上位スレッドが属するスレッド・チーム・サイズを戻します。 **omp_get_team_size** は、ネスト・レベルが 0 から **omp_get_nested_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 を戻します。

プロトタイプ

```
int  omp_get_team_size(int  level);
```

omp_get_active_level
目的

omp_get_active_level では、ネストされたアクティブな並列領域の数が戻されます。

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、このディレクティブに続く並列領域で使用され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_get_num_threads に対する呼び出しで戻すことができる最大値を戻します。

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_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);
```

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戻し、それ以外は 0 を戻します。

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);
```

戻り値

現在のインプリメンテーションでは、ネストされた並列領域は、常に直列化されています。その結果、何の効果も及ぼしません。

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);
```

戻り値

現在のインプリメンテーションでは、ネストされた並列領域は、常に直列化されています。その結果、常に 0 を戻します。

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);
```

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);
```

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*);

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*);

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*);

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。

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

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

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

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

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

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

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

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

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

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

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

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

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

著作権使用許諾:

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

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

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

商標

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

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

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

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

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

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

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

索引

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

[ア行]

[カ行]

[サ行]

[タ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

A

C

F

G

I

L

M

O

P

R

S

V

X

[特殊文字]

プログラム番号: 5724-U80

Printed in Japan

SC88-4917-00

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