IBM XL C/C++ Advanced Edition V8.0 for Linux

コンパイラー解説書

SD88-6729-00
(英文原典：SC09-8013-00)

IBM XL C/C++ Advanced Edition V8.0 for Linux

コンパイラー解説書

SD88-6729-00
(英文原典：SC09-8013-00)

お願い

本書の情報およびそれによってサポートされる製品を使用する前に、 341 ページの『特記事項』に記載する一般情報をお読みください。

本書は、XL C/C++ Advanced Edition V8.0 for Linux (プログラム番号 5724-M16) 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。

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

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

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

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

(URL は、変更になる場合があります)

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

原　典： SC09–8013–00
IBM XL C/C++ Advanced Edition V8.0 for Linux
Compiler Reference

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 2005.10

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

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

目次

本書について

本書にはコンパイル環境のセットアップ、C または C++ 言語で書かれたプログラムのコンパイル、リンク、実行方法、アプリケーションでのコンパイラー・オプション、プラグマ、マクロ、および組み込み関数の指定方法に関する情報が含まれています。このガイドには、XL C/C++ 文書スイートに入っている他の参照ガイドの関連セクションへの大量の相互参照も含まれています。

本書の対象読者

本書は XL C/C++ コンパイラーを使用して作業しようと考えており、Linux オペレーティング・システムに精通し、以前に C または C++ プログラミングの経験がある方を対象としています。しかし、C または C++ の経験がない方でもこの文書を使用することにより、XL C/C++ コンパイラーに固有の機能やフィーチャーに関する情報を見つけることができます。このガイドは、コンパイラーのフィーチャー (特にオプション)、およびそれらを使用した効率的なソフトウェア開発方法の理解を助けます。

本書の使用方法

ページ全体を通して、**xlc** および **xlc++** コマンド呼び出しは、コンパイラーのアクションを記述するために使用されます。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。

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

本書ではコンパイラーの構成方法、および XL C/C++ コンパイラーを使用した C または C++ アプリケーションのコンパイル、リンク、および実行方法に関する情報を扱っています。以下のトピックは含まれていません。

- C または C++ 言語の情報: C および 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/C++ 言語解説書*」を参照してください。
- プログラミング・トピックの情報: プログラムのポータビリティーと最適化については、「*XL C/C++ 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 コンパイラーの構成』では、環境変数の設定や構成ファイルのカスタマイズなど、コンパイル環境のセットアップに関連したトピックについて説明しています。

13 ページの『第 2 章 アプリケーションのコンパイルとリンク』 では、コンパイル・タスクに関連したトピックについて説明しています。このトピックにはコンパ

イラー、プリプロセッサー、およびリンク・エディターの呼び出し、入出力ファイルのタイプ、組み込みファイルのパス名とディレクトリー検索シーケンスの各種設定方法、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各種方法、およびコンパイラー・リストとメッセージなどがあります。

37 ページの『第 3 章 コンパイラー・オプション参照』 には最初に機能カテゴリー別のオプションの要約があり、これを使用して機能別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。そしてこの章には、アルファベット順に分類された各コンパイラー・オプションの個々の説明があります。各オプションの説明には、例示と関連トピックのリストがあります。

239 ページの『第 4 章 glxc および glxc++ での GNU C/C++ コンパイラー・オプションの再使用』には、コンパイラー・ユーティリティー **gxlc** および **gxlc++** の使用を通じた、GNU C/C++ コンパイラー・オプションの再使用方法に関する情報が含まれています。

243 ページの『第 5 章 コンパイラー・プラグマ参照』 には、OpenMP ディレクティブなど、プラグマの個々の説明が含まれています。

311 ページの『第 6 章 事前定義マクロ』 には、コンパイラー・マクロのリストがあります。

315 ページの『第 7 章 POWER および PowerPC アーキテクチャーの組み込み関数』 には、機能ごとに分類された、POWER™ および PowerPC® アーキテクチャー用の XL C/C++ 組み込み関数の個々の説明が含まれています。

335 ページの『付録 A. 再配布可能ライブラリー』 には、XL C/C++ に同梱されている再配布可能なライブラリーがリストされています。

337 ページの『付録 B. ASCII 文字セット』 には、ASCII 文字セットのリストがあります。

本書で使用されている規則

以下の節では、本書で使用されている規則について説明しています。

活字の規則

下記のテーブルは、本書で使用されている活字の規則について説明しています。

表 1. 活字の規則

書体	意味	例
太字	コマンド、実行可能ファイル名、コンパイラー・オプション、およびプラグマ・ディレクティブ。	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には、**-qmkshrobj** コンパイラー・オプションを使用します。
イタリック	パラメーターまたは変数。実際の名前と値はユーザーによって提供されます。イタリックは新規用語の導入にも使用されます。	要求される *size* を超えるサイズを戻す場合は、必ず *size* パラメーターを更新してください。
モノスペース	プログラミング・キーワードとライブラリー関数、コンパイラー組み込み関数、ファイル名とディレクトリー名、プログラム・コードの例、コンパイラー・メッセージ、コマンド・ストリング、またはユーザー定義名。	一般に `switch` ステートメントの 1 つまたは 2 つのケースが他のケースより頻繁に実行される場合は、`switch` ステートメントの前にそれらを別々に処理することによりこれらのケースを類別してください。

アイコン

本書に記述されているすべてのフィーチャーは、C と C++ 言語の両方に適用されます。フィーチャーが 1 つの言語に排他的である箇所や機能が言語間で異なる箇所では、以下のアイコンが使用されます。

C

このテキストは C 言語のみでサポートされているフィーチャーを記述しています。または、C 言語に特定の振る舞いを記述しています。

C++

このテキストは C ++ 言語のみでサポートされているフィーチャーを記述しています。または、C++ 言語に特定の振る舞いを記述しています。

構文図の読み方

- 構文図は線のパスに沿って、左から右、上から下へと読んでいきます。

  ▶▶── 記号は、コマンド、ディレクティブ、またはステートメントの先頭を示しています。

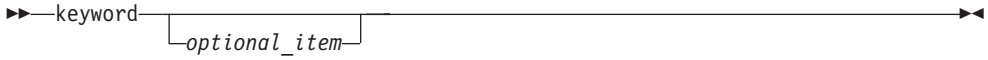
  ──▶ 記号は、コマンド、ディレクティブ、またはステートメントの構文が次の行に続いていることを示しています。

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

  ──▶◀ 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ています。

  完全なコマンド、ディレクティブ、またはステートメント以外の構文単位の図は、▶── 記号で始まり、──▶ 記号で終わります。
- 必要項目は水平線 (メインパス) 上に現れます。

- オプショナル項目はメインパスの下に表示されます。



- 2 つ以上の項目から選択できる場合は、縦に重ねて表示されます。

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

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

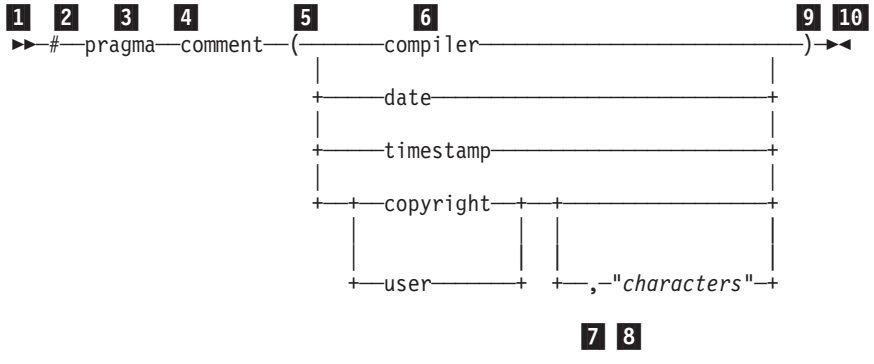
  デフォルトの項目はメインパスの上に表示されます。

- メインラインの上を左に戻る矢印は、繰り返し可能な項目を示しています。

  スタックの上の繰り返し矢印は、スタック項目から複数の選択項目を選択できること、または単一の選択項目を繰り返せることを示しています。
- キーワードは非イタリック文字で表示され、表示通り正確に入力する必要があります(例えば、`extern`)。

  変数はイタリックの小文字で表示されます (例えば、*identifier*)。それらはユーザー提供の名前と値を表します。
- 句読記号、括弧、算術演算子、または他のそのような記号が表示されている場合は、構文の一部として入力する必要があります。

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



1 これは構文図の最初です。

2 # 記号が先頭に現れる必要があります。

3 キーワード pragma は # 記号の後に現れる必要があります。

4 pragma comment の名前はキーワード pragma に続いて現れなければなりません。

5 左括弧が存在している必要があります。

6 コメント・タイプは次に示すタイプの 1 つとしてのみ入力する必要があります: compiler、date、timestamp、copyright、または user。

7 コメント・タイプ copyright または user とオプショナル文字ストリングの間にコンマが現れる必要があります。

8 コンマの後には文字ストリングが続いていなければなりません。文字ストリングは二重引用符で囲む必要があります。

9 右括弧が必要です。

10 これが構文図の終わりです。

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

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

例

本書の例は、特に明記されていない限り、単純スタイルでコーディングされており、ストレージの節約、エラーのチェック、高速パフォーマンスの達成、または特定結果を達成するためにすべての考えられるメソッドを説明するといったことは試行しません。

関連情報

IBM XL C/C++ 資料

XL C/C++ は、以下の形式で製品資料を提供しています。

- README ファイル

  README ファイルには製品資料の変更や訂正を含め、最新情報が含まれています。README ファイルはデフォルトで、/opt/ibmcmp/vacpp/8.0/ ディレクトリーとインストール CD のルート・ディレクトリーにあります。

- インストール可能な man ページ

  man ページは製品に準備されているコンパイラー呼び出しとすべてのコマンド行ユーティリティーに対して提供されています。man ページのインストール方法とアクセス方法についての説明は、「*XL C/C++ Advanced Edition V8.0 for Linux インストール・ガイド*」で提供されています。

- 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・センターは、ネットワーク上で起動でき、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。イ

ンフォメーション・センターのインストール方法とアクセス方法についての説明は、「*XL C/C++ Advanced Edition V8.0 for Linux インストール・ガイド*」で提供されています。インフォメーション・センターは Web でも表示可能です:

http://publib.boulder.ibm.com/infocenter/lnxpcomp/index.jsp

- PDF 文書

  PDF 文書はデフォルトで、`/opt/ibmcmp/vacpp/8.0/doc/`*`language`*`/pdf` ディレクトリーにあります。ここで、*language* はサポートされる以下の言語のいずれかになります。

  – en_US
  – ja_JP
  – zh_CN

  PDF 文書は、以下の Web ページからも入手可能です。

  www.ibm.com/software/awdtools/xlcpp/library

  この資料のほかに、以下のファイルが XL C/C++ 製品資料のフルセットを構成します。

表 2. XL C/C++ PDF ファイル

文書タイトル	PDF ファイル名	説明
XL C/C++ Advanced Edition V8.0 for Linux インストール・ガイド, GD88-6645-00	install.pdf	XL C/C++ のインストール方法と基本的なコンパイルおよびプログラム実行のための環境の構成方法に関する情報が含まれています。
XL C/C++ Advanced Edition V8.0 for Linux 入門, SD88-6653-00	getstart.pdf	XL C/C++ 製品の概要と、環境のセットアップと構成、プログラムのコンパイルとリンク、およびコンパイル・エラーのトラブルシューティングに関する情報が含まれています。
XL C/C++ Advanced Edition V8.0 for Linux 言語解説書, SD88-6654-00	language.pdf	IBM によってサポートされている C および C++ プログラム言語に関する情報 (ポータビリティーのための言語拡張子、および非専用標準に対する準拠など) が含まれています。
XL C/C++ Advanced Edition V8.0 for Linux プログラミング・ガイド, SD88-6644-00	proguide.pdf	アプリケーションの移植、FORTRAN コードでの言語間呼び出し、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/C++ の高性能ライブラリーなど、上級プログラミング・トピックに関する情報が含まれています。

これらの PDF ファイルは Adobe Reader から表示および印刷することができます。Adobe Reader がインストールされていない場合は、www.adobe.com からダウンロードすることができます。

追加文書

このほかにも、XL C/C++ に関連した文書 (Redbooks、白書、チュートリアル、およびその他の文書) を次の Web サイトから入手することができます。

www.ibm.com/software/awdtools/xlcpp/library

関連資料

次の資料も参照することができます。これらの資料も本書中で参照されています。

- 「*OpenMP Application Program Interface Version 2.5*」(http://www.openmp.org から入手可能)
- 「*AltiVec Technology Programming Interface Manual*」(http://www.freescale.com から入手可能)

テクニカル・サポート

XL C/C++ の「サポート」ページから追加のテクニカル・サポートを利用することができます。このページは、テクニカル・サポートの FAQ とその他のサポート文書の豊富なセレクションへの検索機能付きのポータルを提供します。XL C/C++ のサポート・ページは、次の Web サイトで見つけることができます。

www.ibm.com/software/awdtools/xlcpp/support

必要なものが見つからない場合は、下記宛てまで E メールをお送りください。

compinfo@ca.ibm.com

XL C/C++ の最新情報については、次の製品情報サイトにアクセスしてください。

www.ibm.com/software/awdtools/xlcpp

第 1 章 コンパイラーの構成

XL C/C++ を使用して C または C++ プログラムをコンパイルする前に、コンパイラーに必要な環境変数をセットアップし、コンパイル構成ファイルがシステムに存在している必要があります。通常、コンパイラーによって使用される構成ファイルは、インストール手順の間に自動的に生成され、 (詳しくは、「*XL C/C++ インストール・ガイド*」を参照してください。) インストール・プロセス中にいくつかの基本的な環境変数はすでに設定されている可能性があります。

『環境変数の設定』は、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要な環境変数とオプショナルの環境変数 (並列処理に使用される環境変数など) の完全リストを提供しています。

インストール後に生成したデフォルトまたは追加の構成ファイルをカスタマイズして代替ライブラリー・パスやデフォルトのコンパイラー・オプションなどを指定したい場合は、9 ページの『構成ファイルのカスタマイズ』を参照してください。デフォルト構成ファイルの構造と内容の説明が提供されています。

環境変数の設定

Linux® システムの Bourne Again SHell (**bash**) は、AIX® システムの Bourne Shell (**bsh**) と似ています。**bash** インターフェースを使用して、XL C/C++ コンパイラーに必要な環境変数を、コマンド行またはコマンド・ファイル・スクリプトのいずれかによって設定してください。

以下のステートメント (コマンド行に入力、またはコマンド・ファイル・スクリプトに挿入される) は、Bourne Again SHell での環境変数の設定方法を示しています。示されているパスは、コンパイラーをデフォルトのインストール・ロケーションにインストールすることを想定しています。

```
LANG=en_US
NLSPATH=$NLSPATH:/opt/ibmcmp/msg/%L/%N:/opt/ibmcmp/vacpp/8.0/msg/%L/%N
export LANG NLSPATH
```

すべてのユーザーがアクセスできるような変数を設定するには、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。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。

以下の節では、コンパイラーに対して設定できる環境変数について説明します。

- 『汎用環境変数』
- 3 ページの『並列処理のための環境変数』

汎用環境変数

コンパイラーを使用する前に、以下の環境変数が設定されていることを確認してください。

LD_LIBRARY_PATH　動的ロード・ライブラリーのディレクトリー検索パスを指定します。 GNU リンカーによって、リンク時と実行時に使用されます。

LD_RUN_PATH　動的ロード・ライブラリーのディレクトリー検索パスを指定します。実行時にのみ使用されます。

LANG　メッセージとヘルプ・ファイルのための各国語を指定します。

LANG 環境変数は、システムで提供されているどのロケールにも設定することができます。

米国英語用の各国語コードは **en_US** です。適切なメッセージ・カタログがすでにシステムにインストール済みであれば、**en_US** を有効な他の各国語コードに置き換えることができます。

ご使用システムの各国語の現在の設定を判別するには、以下の **echo** コマンドを使用してください。

```
echo $LANG
```

MANPATH　man ページを検索するためのディレクトリー検索パスをオプションで指定します。MANPATH には、デフォルトの man パスの前に、`/opt/ibmcmp/vacpp/8.0/man/EN_US/` が含まれている必要があります。

NLSPATH　メッセージとヘルプ・ファイルのためのパス名を指定します。

ご使用システムの NLSPATH 変数の現在の設定を判別するには、以下の **echo** コマンドを使用してください。

```
echo $NLSPATH
```

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 `/opt/ibmcmp/vacpp/8.0/bin/` にあります。

PDFDIR　プロファイル・データ・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。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。この変数は、プロファイル指定のフィードバックのために、絶対パスに設定することをお勧めします。

TMPDIR　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。

XL_NOCLONEARCH　プログラムに汎用コードだけを実行するよう命令します。この場合の汎用コードとは、アーキテクチャーに対してバージョン管理されていないコードのことです。 XL_NOCLONEARCH 環境変数はデフォルトでは設定されていません。これはアプリケーションでのデバッグ目的のために設定することができます。

注: LANG および NLSPATH 環境変数は、オペレーティング・システムのインストール時に初期設定されるため、使用したい環境変数と異なっていることがあります。

関連情報

- 34 ページの『メッセージ・カタログ・エラー』

- 174 ページの『-qpdf1、-qpdf2』
- 122 ページの『-qipa』

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、『並列処理のための XLSMPOPTS 環境変数のサブオプション』を参照してください。

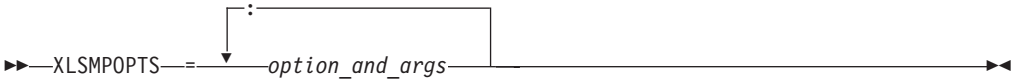
並列化に OpenMP 構成体を使用している場合は、7 ページの『並列処理のための OpenMP 環境変数』に説明があるように、OMP 環境変数を使用してランタイム・オプションを指定することもできます。

OMP および XLSMPOPTS 環境変数によって指定されたランタイム・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。

並列処理のための XLSMPOPTS 環境変数のサブオプション

並列処理に影響を及ぼすランタイム・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。



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
```

XLSMPOPTS 環境変数のランタイム・オプションの設定は下記の通りです。カテゴリー別にグループ化されています。

スケジューリング・アルゴリズム・オプション:

XLSMPOPTS 環境変数オプション	説明
schedule=*algorithm*=[*n*]	このオプションは、明示的にスケジューリング・アルゴリズムに割り当てられていないループに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。 *algorithm* の有効なオプションは、以下の通りです。 • guided • affinity • dynamic • static 指定する場合、チャンク・サイズ *n* は 1 以上の整数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。

並列環境オプション:

XLSMPOPTS 環境変数オプション	説明
parthds=*num*	*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。 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*	*num* は、予期されるユーザー・スレッドの数を示します。 プログラム・コードによって明示的にスレッドが作成される場合は、このオプションを使用してください。その場合は、*num* を、作成するスレッドの数に設定してください。 *num* のデフォルト値は 0 です。

XLSMPOPTS 環境変数オプション	説明
stack=*num*	*num* は、スレッドのスタックに必要なスペースの最大量を指定します。 *num* のデフォルト値は 2097152 です。 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。

パフォーマンス・チューニング・オプション:

XLSMPOPTS 環境変数オプション	説明
spins=*num*	*num* は、譲歩するまでのループ・スピンまたは反復の数を示します。 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。
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、SMP ランタイムが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=*Number*	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。*Number* 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。
yields=*num*	*num* は、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。

XLSMPOPTS 環境変数 オプション	説明
delays=*num*	*num* は、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・オプション:

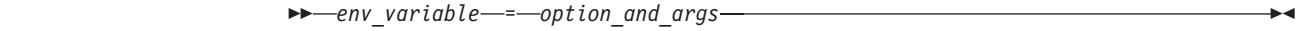
XLSMPOPTS 環境変数 オプション	説明
profilefreq=*num*	*num* は、並列処理が適正であるかどうかを判別するために各ループを再調査するサンプリング率を示します。 ランタイム・ライブラリーは動的プロファイルを使用して、自動的に並列化されるループのパフォーマンスを動的に調整します。 動的プロファイルはループの実行時間に関する情報を収集して、次回ループを実行するときに順次実行すべきか並列に実行すべきかを判別します。 実行時間のしきい値は、以下で説明する **parthreshold** および **seqthreshold** 動的プロファイル・オプションによって設定します。 *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。最大のサンプリング率は 32 です。*num* の値が 32 を超えている場合は、32 に変更されます。
parthreshold=*mSec*	*mSec* は、予期される実行時間をミリ秒で指定します。この時間に満たない場合、ループは順次実行しなければなりません。 *mSec* は、小数点以下の桁を使用して指定することができます。 **parthreshold** を 0 に設定すると、並列化されたループは動的プロファイラーによって直列化されなくなります。 *mSec* のデフォルト値は 0.2 ミリ秒です。
seqthreshold=*mSec*	*mSec* は、動的プロファイラーによって直列化されたループを再び並列モードで実行するように戻す実行時間の下限をミリ秒単位で指定します。 *mSec* は、小数点以下の桁を使用して指定することができます。 *mSec* のデフォルト値は 5 ミリ秒です。

関連情報

- 245 ページの『OpenMP プラグマ・ディレクティブの要約』
- 332 ページの『並列処理のための組み込み関数』

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 ランタイム・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 ランタイム・オプションは、下記の各種カテゴリーに分類されます。

スケジューリング・アルゴリズム環境変数:

OMP_SCHEDULE=*algorithm*

このオプションは、**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。例えば、以下のように指定します。

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

algorithm の有効なオプションは、以下の通りです。

- 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。

並列環境変数:

OMP_NUM_THREADS=*num*

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、プログラム実行中にオーバーライド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。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。

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

この環境変数は、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。この環境変数の設定は、**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。

動的プロファイル環境変数:

OMP_DYNAMIC=TRUE|FALSE
: この環境変数は、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、6 ページの『動的プロファイル・オプション』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。

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

構成ファイルのカスタマイズ

XL C/C++ は、インストール時にデフォルト構成ファイル `/etc/opt/ibmcmp/vac/8.0/vac.cfg` を生成します (インストール中に構成ファイルの生成に使用できる各種ツールについて詳しくは、「*XL C/C++ インストール・ガイド*」を参照してください)。構成ファイルは、コンパイラーが呼び出し時に使用する情報を指定します。

単一ユーザー・システムで稼働している場合や、コンパイル・スクリプトや Make ファイルを持つコンパイル環境がすでにある場合は、デフォルトの構成ファイルをそのままにしておくことができます。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の中から選択できるようにしたい場合は、既存のスタンザを変更するか、構成ファイルに新規の名前付きスタンザを追加することができます。例えば、**-qnoro** を **xlc** コンパイラー呼び出しコマンドのデフォルトにするには、構成ファイルのコピー・バージョンの **xlc** スタンザに **-qnoro** を追加します。

既存のコマンドへのリンクである新規コマンドを作成することができます。例えば、**xlc_r** コマンドへのリンクを作成するには、以下のように指定することができます。

```
ln -s /opt/ibmcmp/vacpp/8.0/bin/xlc_r /home/lisa/bin/my_xlc
```

コンパイラー呼び出しコマンドは幾つかの異なる名前にリンクすることができます。コンパイラーの呼び出し時に指定した名前により、コンパイラーが使用する構成ファイルのスタンザが決まります。構成ファイルのコピーに他のスタンザを追加して、独自のコンパイル環境をカスタマイズすることができます。コンパイラー呼び出しコマンドに **-F** オプションを使用して、追加のスタンザを選択したり別の構成ファイル内の特定のスタンザを指定するためのリンクを作成することができます。例えば、以下のように指定します。

```
xlc myfile.c -Fmyconfig.cfg:SPECIAL
```

こうすると、`myconfig.cfg` 構成ファイルの SPECIAL スタンザを使用して `myfile.c` がコンパイルされます。

別の名前でコンパイラーを実行すると、コンパイラーは対応するスタンザにリストされたオプションやライブラリーなどを使用します。

注:

1. デフォルト構成ファイルを変更して、別のシステムに Make ファイルを移動またはコピーする場合は、変更した構成ファイルもコピーする必要があります。
2. コンパイラーのプログラム一時修正 (PTF) またはアップグレードをインストールすると、vac.cfg ファイルが上書きされる可能性があります。したがって、そのようなインストールの前には必ず、行った変更のコピーを保管しておいてください。
3. 構成ファイル内の分離文字としてタブを使用することはできません。構成ファイルを変更する場合は、インデントには必ずスペースを使用してください。
4. Message-Passing Interface (MPI) とスレッド化プログラミングを混用する場合は、vac.cfg ファイルで適切なスタンザを使用して、適切なライブラリーでリンクし、正しいデフォルトの振る舞いを設定してください。
5. コンパイラー戻りコード 41 は、構成ファイル・エラーが検出されたことを示しています。

構成ファイルの属性

構成ファイルには、幾つかのスタンザが含まれています。構成ファイルのスタンザによって定義される項目には、以下のものがあります。

as	アセンブラーに使用されるパス名。デフォルトは /usr/bin/as です。
ccomp	C フロントエンドです。デフォルトは /opt/ibmcmp/vac/8.0/exe/xlcentry です。
cppcomp	C++ フロントエンドです。デフォルトは /opt/ibmcmp/vacpp/8.0/exe/xlCentry です。
code	コンパイラーのコード生成フェーズに使用されるパス名。デフォルトは /opt/ibmcmp/vac/8.0/exe/xlCcode です。
codeopt	コンパイラーのコード生成フェーズのオプションのリストです。
crt	リンケージ・エディターへの第 1 パラメーターとして渡されるオブジェクト・ファイルのパス名です。 **-p** と **-pg** オプションのいずれも指定しないと、**crt** 値が使用されます。デフォルトは /usr/lib/crt1.o です。
csuffix	ソース・プログラムのサフィックスです。デフォルトは c (小文字の c) です。
dis	逆アセンブラーのパス名です。デフォルトは /opt/ibmcmp/vac/8.0/exe/dis です。
gcrt	リンケージ・エディターへの第 1 パラメーターとして渡されるオブジェクト・ファイルのパス名です。 **-pg** オプションを指定すると、gcrt 値が使用されます。デフォルトは /usr/lib/gcrt1.oです。
ld	C または C++ プログラムのリンクに使用されるパス名です。デフォルトは /usr/bin/ld です。
ldopt	リンケージ・エディターに誘導され、コンパイラーによるすべての通常処理をオーバーライドするオプションのリストです。対応するフラグがパラメーターを取る場合、ストリングは 1 つの文字の後にコロン (:) が続く、フラグ文字の連結として、getopt サブルーチン用にフォーマットされます。
libraries2	コンパイラーがリンケージ・エディターに最後のパラメーターとして渡す、コンマで区切られたライブラリー・オプションのリストです。**libraries2** は、リンケージ・エディターがプロファイルと非プロファイルの両方に対してリンク・エディット時に使用するライブラリーを指定します。デフォルトは空です。

mcrt	**-p** オプションを指定した場合に、リンケージ・エディターへの第 1 パラメーターとして渡されるオブジェクト・ファイルのパス名です。デフォルトは `/usr/lib/gcrt1.o` です。
options	コマンド行に入力されたようにコンパイラーによって処理される、コンマで区切られたオプション・フラグのストリングです。
osuffix	オブジェクト・ファイルのサフィックスです。デフォルトは .o です。
use	属性の値は名前付きスタンザかローカル・スタンザから取られます。単一値属性の場合、ローカルまたはデフォルトのスタンザに値が提供されていないと、**use** スタンザの値が適用されます。コンマで区切られたリストの場合、use スタンザの値がローカル・スタンザの値に追加されます。
xlc	**xlc** コンパイラー・コンポーネントのパス名です。デフォルトは `/opt/ibmcmp/vac/8.0/bin/xlc` です。
xlC	**xlC** コンパイラー・コンポーネントのパス名です。デフォルトは `/opt/ibmcmp/vacpp/8.0/bin/xlC` です。

関連情報

- 22 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』

第 2 章 アプリケーションのコンパイルとリンク

デフォルトで、XL C/C++ コンパイラーの呼び出しはプログラム・ソースのプリプロセッシングを実行し、コンパイルしてオブジェクト・ファイルを作成し、実行可能ファイルにリンクします。これらの変換フェーズは実際には、コンパイラー・コンポーネント と呼ばれる別の実行可能プログラムによって実行されます。コマンド行オプションが指定されていない場合、XL C/C++ コンパイラー呼び出しコマンドは自動的にプリプロセッサーとリンケージ・エディターを呼び出し、さらにソース・プログラムからオブジェクト・コードへの変換を実行するコンポーネント (コンパイラー とも呼ばれる) を呼び出します。 ただし、プリプロセッサーとリンケージ・エディターは独立して呼び出すことができます。

以下の節では、XL C/C++ コンパイラーを呼び出してソース・ファイルとライブラリーをプリプロセス、コンパイル、およびリンクする方法について説明します。

- 『コンパイラーの呼び出し』
- 16 ページの『入力ファイルのタイプ』
- 18 ページの『出力ファイルのタイプ』
- 19 ページの『コンパイラー・オプションの指定』
- 25 ページの『プリプロセッシング』
- 28 ページの『リンク』
- 30 ページの『コンパイラーのメッセージおよびリスト』

コンパイラーの呼び出し

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

特定の環境で必要な場合は、他の形式のコマンドを使用することができます。各種コンパイラー呼び出しコマンドは、以下のとおりです。

基本	特殊
xlC	xlC_r
xlc++	xlc++_r
xlc	xlc_r
cc	cc_r
c99	c99_r
c89	c89_r

XL C/C++ では、下記のように、基本コンパイラー呼び出しの **_r** バリエーションを提供しています。

表 3. 特殊呼び出しのサフィックス

_r サフィックスの呼び出し	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。これらは追加で、マクロ名 __VACPP_MULTI__ と REENTRANT を定義し、ライブラリー**-lpthread** を追加します。 コンパイラー・オプション **-qthreaded** も追加されます。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。

呼び出しコマンドの選択

基本的なコンパイラー呼び出しコマンドは、表 4 の各行の最初の項目として表れます。以下の基準で、基本呼び出しを選択してください。

表 4. コンパイラー呼び出し

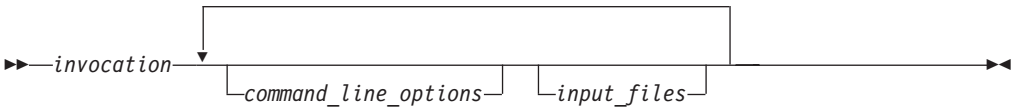
呼び出し	基準
xlC xlc++	両方ともソース・ファイルが C++ 言語ソース・コードとしてコンパイルされるように、コンパイラーを呼び出します。いずれかのソース・ファイルが C++ の場合は、適正なランタイム・ライブラリーとリンクするために、この呼び出しを使用する必要があります。ソース・ファイルは **-qalias=ansi** を設定してコンパイルされます。 **.c** サフィックスのファイルは、**-+** コンパイラー・オプションが使用されていないと想定した場合、 **-qlanglvl=extc89** が有効なときには、C 言語ソース・コードとしてコンパイルされます。
xlc	C ソース・ファイル用のコンパイラーを呼び出します。この呼び出しでは、以下のコンパイラー・オプションが暗黙指定されます。 • -qlanglvl=extc89 • -qalias=ansi • -qcpluscmt • -qkeyword=inline
cc	C ソース・ファイル用のコンパイラーを呼び出します。この呼び出しでは、以下のコンパイラー・オプションが暗黙指定されます。 • -qlanglvl=extended • -qnoro • -qnoroconst

表 4. コンパイラー呼び出し (続き)

呼び出し	基準
c99	ISO C99 言語フィーチャーをサポートする、C ソース・ファイル用のコンパイラーを呼び出します。完全な ISO C99 (ISO/IEC 9899:1999) の規格合致には、C99 準拠のヘッダー・ファイルとランタイム・ライブラリーが存在している必要があります。この呼び出しでは、以下のオプションが暗黙指定されます。 • -qlanglvl=stdc99 • -qalias=ansi • -qstrict_induction • -D_ANSI_C_SOURCE • -D_ISOC99_SOURCE • -D__STRICT_ANSI__ ANSI 規格 (ISO/IEC 9899:1999) への厳密な規格合致のためには、この呼び出しを使用してください。
c89	ISO C89 言語フィーチャーをサポートする、C ソース・ファイル用のコンパイラーを呼び出します。この呼び出しでは、以下のオプションが暗黙指定されます。 • -qlanglvl=stdc89 • -qalias=ansi • -qstrict_induction • -qnolonglong • -D_ANSI_C_SOURCE • -D__STRICT_ANSI__ 厳密な ANSI 規格 (ISO/IEC 9899:1990) への規格合致のためには、この呼び出しを使用してください。
gxlc++	このユーティリティーを使用して C++ ファイルをコンパイルすることができます。これは多くの共通の GNU C/C++ オプションを受け入れ、それらを XL C/C++ オプションの同等のものにマップして、**xlc** を呼び出します。詳しくは、239 ページの『第 4 章 glxc および glxc++ での GNU C/C++ コンパイラー・オプションの再使用』を参照してください。
gxlc	このユーティリティーを使用して C ファイルをコンパイルすることができます。これは多くの共通の **gcc** オプションを受け入れて、それらを **xlc** オプションの同等のものにマップし、**xlc** を呼び出します。詳しくは、239 ページの『第 4 章 glxc および glxc++ での GNU C/C++ コンパイラー・オプションの再使用』を参照してください。

呼び出し構文

XL C/C++ は以下の構文を使用して呼び出されます。ここで、*invocation* は任意の有効な XL C/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、およびリンケージ・エディター・オプション名にすることができます。

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ケージ・エディターの*両方を* 呼び出します。このコマンドは、リンケージ・エディター・オプションをリンケージ・エディターに渡します。 その結果、これらの呼び出しコマンドは、すべてのリンケージ・エディターのオプションも受け入れます。リンク・エディット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*.c 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。リンケージ・エディターは呼び出されません。**-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すると、後でそのオブジェクト・ファイルをリンク・エディットすることができます。

関連情報
- 『入力ファイルのタイプ』
- 30 ページの『コンパイラーのメッセージおよびリスト』

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・ファイルが見つけからないと、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、リンケージ・エディターは実行されず、一時オブジェクト・ファイルは除去されます。

プログラムは、幾つかのソース・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、**-E** または **-P** オプションのいずれかを指定すると、コンパイルを行わずにコンパイラーを使用してソース・ファイルをプリプロセスすることができます。**-P** オプションを指定した場合は、プリプロセスされたソース・ファイル、*file_name* .i が作成され、処理は終了します。

-E オプションは、ソース・ファイルをプリプロセスし、標準出力へ書き込み、出力ファイルを生成せずに処理を停止します。

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

表 5. 有効な入力ファイルのタイプ

C および C++ ソース・ファイル	これらは C または C++ ソース・コードを含むファイルです。 C コンパイラーを使用して C 言語ソース・ファイルをコンパイルするには、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 (例えば、`mysource.c`)。 C++ コンパイラーを使用するためには、ソース・ファイルが .C (大文字の C)、.cc、.cp、.cpp、.cxx、または .c++ のサフィックスを持っている必要があります。他のファイルを C++ ソース・ファイルとしてコンパイルするには、**-+** コンパイラー・オプションを使用します。このオプションを使用して指定され、.a、.o、.so、または .s 以外のサフィックスを持っているすべてのファイルは、C++ ソース・ファイルとしてコンパイルされます。
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 サフィックスを持ちます (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 または .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 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*.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実行可能ファイルはリンケージ・エディターの入力として使用できます。コンパイル後、リンケージ・エディタ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 .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。
Assembler-with-cpp	アセンブラー・ファイルは .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 の拡張子を持つすべてのソース・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・ファイルは .so 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*.so)。
ストリップされていない実行可能ファイル	Linux の **strip** コマンドを使用してストリップされていない executable and linking format (ELF) ファイルは、コンパイラーへの入力として使用できます。

関連情報

- 機能カテゴリー別のオプション要約: 入力制御

出力ファイルのタイプ

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

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

オブジェクト・ファイル
-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、全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)。

-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 が入力ソース・ファイル *file_name.x* ごとに作成されます。ここで、*x* は **C** または **C++** ファイル名拡張子と認識されます。リンケージ・エディターは呼び出されず、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。

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・エディット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。

アセンブラー・ファイル
アセンブラー・ファイルは .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)。

これらは、**-S** オプションを指定することにより作成されます。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 サフィックスを持ちます (例えば、*file_name*.i)。

プリプロセスされたソース・ファイルを作成するには、**-P** オプションを指定します。ソース・ファイルはプリプロセスされますが、コンパイルはされません。 **-E** オプションの出力を宛先変更して、`#line` ディレクティブを含むプリプロセス・ファイルを生成することもできます。

プリプロセスされたソース・ファイル *file_name*.i がそれぞれのソース・ファイルごとに作成され、その作成元のソース・ファイルと同じファイル名 (.i 拡張子が付く) になります。

リスト・ファイル　リスト・ファイルは .lst のサフィックスが付きます (例えば、*file_name*.lst)。

呼び出しコマンドへのリスト関連オプションのいずれか 1 つを指定して、コンパイラー・リストを作成します (**-qnoprint** オプションを指定していない場合)。このリストを含むファイルは、現行ディレクトリーに置かれ、作成元のソース・ファイルと同じファイル名 (.lst の拡張子) になります。

共用ライブラリー・ファイル　共用ライブラリー・ファイルは .so サフィックスが付きます (例えば、`my_shrlib.so`)。

ターゲット・ファイル　**-M** or **-qmakedep** オプションと関連した出力ファイルは、.d のサフィックスが付きます (例えば、conversion .d)。

このファイルには、**make** コマンドの記述ファイルへの組み込みに適したターゲットが含まれます。 .d ファイルはすべての入力 C または C++ ファイルに対して作成され、**make** コマンドにより、別の入力ファイルに対して行われた変更の結果、指定された入力ファイルの再コンパイルが必要かどうかを判断するために使用されます。.d ファイルは他のどのファイルに対しても作成されません (**-+** オプションを使用して他のファイル・サフィックスが .C ファイルとして扱われるようにした場合を除きます)。

関連情報
- 機能カテゴリー別のオプション要約: 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
- 構成ファイル (.cfg 拡張子を持つファイル)
- ソース・プログラム
- Make ファイル

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

コンパイラー・オプションを指定した場合、オプションの矛盾や非互換が起きる可能性があります。 XL C/C++ はこれらの矛盾や非互換のほとんどを、以下のように一貫性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します。

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します。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. 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、23 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。

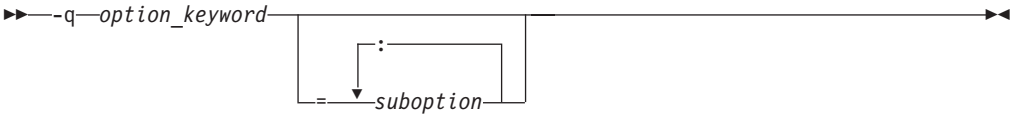
コマンド行でのコンパイラー・オプションの指定

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の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、23 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。例を以下に示します。

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

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

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

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

多くのコンパイラー・オプションは省略することもできます。例えば、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。例を以下に示します。

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

これは、報告されるメッセージの重大度レベルを指定するために、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。**-qflag** 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

Linux システムで使用可能なコンパイラーは、多くの共通の従来型フラグ・オプションを使用します。IBM XL C/C++ はこれらのフラグをサポートします。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、リンケージ・エディタ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ために **-P** または **-E** とともに使用することができます。

IBM XL C/C++ は、他の Linux プログラミング・ツールおよびユーティリティー (例えば、Linux **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・エディット時に、**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を報告し (**-v**) ますが、リンケージ・エディタ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/opt/ibmcmp/vacpp/8.0/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 または C++ プログラムをコンパイルするときにこのファイルを参照します。構成ファイルはプレーン・テキスト・ファイルで、特定のコンパイル要件をサポートしたり、他の C または C++ コンパイル環境をサポートするために、このファイルに項目を作成することができます。

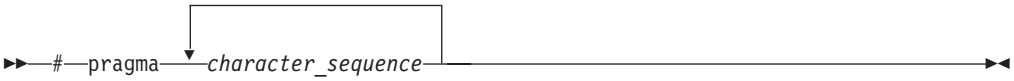
コンパイラーによる矛盾または非互換オプションの解決法について詳しくは、 23 ページの『矛盾するコンパイラー・オプションの解決』を参照してください。

関連情報
- 9 ページの『構成ファイルのカスタマイズ』

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。

プラグマハコンパイラーに対してインプリメンテーションを定義した命令です。これは、以下の一般形式の 1 つを持っています。



ここで、*character_sequence* は特定のコンパイラー命令と引数がある場合にそれを提供する一連の文字です。複数のプラグマの構成を、単一のプラグマ・ディレクティブに指定することができます。

単項演算子 _Pragma は、プリプロセッサー・マクロがプラグマ・ディレクティブに含まれることを許可します。

string_literal には、それを wide-string リテラルにするプレフィックス **L** が付いていることがあります。このストリング・リテラルはストリング変換されてトークン化されます。結果のトークンのシーケンスは、それがプラグマ・ディレクティブに現れているかのように処理されます。例:

```
_Pragma ( "pack(full)" )
```

これは以下と同等です:

```
#pragma pack(full)
```

プラグマの *character_sequence* 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。これらのプラグマ・ディレクティブは、それが適用されるオプションの詳細記述にリストされています。

XL C/C++ でサポートされるさまざまなプラグマ・プリプロセッサー・ディレクティブについて詳しくは、243 ページの『第 5 章 コンパイラー・プラグマ参照』を参照してください。

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。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. 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。

オプション	矛盾するオプション	解決
-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
-p | **-pg** | **-qprofile**	**-p** | **-pg** | **-qprofile**	最後に指定されたオプション
-E	**-P** | **-o** | **-S**	**-E**
-P	**-c** | **-o** | **-S**	**-P**
-#	**-v**	**-#**
-F	**-B** | **-t** | **-W** | **-qpath** | 構成ファイル設定	**-B** | **-t** | **-W** | **-qpath**
-qpath	**-B** | **-t**	**-qpath** が **-B** と **-t** をオーバーライドする
-S	**-c**	**-S**

関連情報
- 234 ページの『コンパイラー・モードとプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ』
- 37 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』

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/C++ コンパイラー・オプションを使用して、特定のプロセッサー・アーキテクチャーで使用するためのコンパイラー出力を最適化することができます。また、32 ビット・モードまたは 64 ビット・モードでコンパイルするようコンパイラーに命令することもできます。

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

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

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

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

- アーキテクチャー・ターゲット は、指定された **-qarch** の設定値がコンパイラー・モード の設定値と互換性がある場合は、 **-qarch** コンパイラー・オプションの最後に検出されたインスタンスに応じて設定される。 **-qarch** オプションが設定されていない場合は、コンパイラーは有効なコンパイラー・モード設定を基に、**-qarch** を適切なデフォルトに設定します。
- 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によって選択された有効な **-qarch** を基にした適切なデフォルトに設定します。

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、**-q32/-q64** コンパイラー設定によって決定されます。

関連情報
- 37 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』

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。XL C/C++ は統合単一パス・コンパイラーで、コンパイラー・オプションの使用によりマルチパス・コンパイラーとして機能する能力を保持しています。XL C/C++ プリプロセッサーは、独立したコンパイラー・コンポーネントとして提供されています。

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
E	ソース・ファイルをプリプロセスするようコンパイラーに命令します。`#line` ディレクティブが生成されます。
P	コンパイラー呼び出しに指定された C または C++ ソース・ファイルをプリプロセスし、 各ソース・ファイルごとに .i のファイル名拡張子を持つ中間ファイルを作成します。
C	プリプロセスされた出力にコメントを保持します。
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。
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。

組み込みファイルのパス名の指定

`#include` プリプロセッサー・ディレクティブを使用して 1 つのソース・ファイルを別のソース・ファイルに組み込むときには、組み込むファイル名を指定する必要があります。絶対パス名または相対パス名を使用して、ファイル名を指定することができます。

- **絶対パス名を使用したファイルの組み込み**

  絶対パス名 (*フルパス名* とも呼ばれる) は、ルート・ディレクトリーから始まるファイルの完全な名前です。これらのパス名は、**/** (スラッシュ) 文字で始まります。絶対パス名は、ユーザーが現在いるディレクトリー (*作業* ディレクトリーまたは*現行* ディレクトリーと呼ばれる) に関係なく、指定されたファイルを探し出します。

  以下の例は、John Doe のサブディレクトリー `example_prog` にあるファイル `mine.h` への絶対パスを指定しています。

  ```
  /u/johndoe/example_prog/mine.h
  ```

- **相対パス名を使用したファイルの組み込み**

  相対パス名 は、現行ソース・ファイルを保持するディレクトリー、または **-I***directory* オプションを使用して定義されたディレクトリーに関連するファイルを探し出します。

相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス

C および C++ は `#include` プリプロセッサー・ディレクティブの 2 つのバージョンを定義しています。IBM XL C/C++ は両方をサポートします。`#include` ディレクティブでは、組み込みファイル名は `< >` または `" "` 区切り文字で囲まれます。

区切り文字の選択により、指定された組み込みファイル名の検索に使用される検索パスが決定されます。以下のように、コンパイラーはその組み込みファイルが検出されるまで検索パスのすべてのディレクトリーで組み込みファイルを検索します。

#include の型	**ディレクトリーの検索順序**
#include <*file_name*>	1. コンパイラーはまず、**-I***directory* コンパイラー・オプションによって指定された各ユーザー・ディレクトリーで、コマンド行にユーザー・ディレクトリーが指定された順に、*file_name* を検索します。 2. C++ のコンパイルの場合、コンパイラーはその後、**-qcpp_stdinc** および **-qgcc_cpp_stdinc** コンパイラー・オプションによって指定されたディレクトリーを検索します。 3. 最後に、コンパイラーは **-qc_stdinc** および **-qgcc_c_stdinc** コンパイラー・オプションによって指定されたディレクトリーを検索します。
#include "*file_name*"	1. コンパイラーは最初に、現行ソース・ファイルが常駐するディレクトリーで組み込みファイルを検索します。現行ソース・ファイルとは、ディレクティブ `#include "`*file_name*`"` を含むファイルです。 2. その後コンパイラーは、`#include <`*file_name*`>` に対する前述の検索順序に従って、組み込みファイルを検索します。

注:

1. *file_name* は、組み込まれるファイル名を指定し、必要に応じて、そのファイルへの完全または部分ディレクトリー・パスを含むことができます。
   - ファイル名のみを指定した場合、コンパイラーは、ディレクトリー検索リストの中でそのファイルを探します。
   - ファイル名を部分ディレクトリー・パスと一緒に指定した場合、コンパイラーは、検索パスの中の各ディレクトリーにその部分パスを付加し、完全になったディレクトリー・パスの中でそのファイルの検索を試みます。
   - 絶対パス名を指定した場合、`#include` ディレクティブの 2 つのバージョンの効果は同じです。これは、組み込まれるファイルのロケーションが完全に指定されているからです。
2. `#include` ディレクティブの 2 つのバージョンの唯一の違いは、 `" "` (ユーザー組み込み) バージョンでは、最初に現行ソース・ファイルの常駐するディレクトリーから検索を開始するということです。一般に、標準ヘッダー・ファイルは `< >` (システム組み込み) バージョンを使用して組み込まれ、ユーザーが作成したヘッダー・ファイルは `" "` (ユーザー組み込み) バージョンを使用して組み込まれます。
3. **-qstdinc** および **-qidirfirst** オプションと一緒に、 **-I***directory* オプションを指定することにより、検索順序を変更することができます。

   該当する場合は、**-I***directory* オプションで指定したディレクトリーと、現行のソース・ファイル・ディレクトリーのみを検索するため、 **-qnostdinc** オプションを使用します。

-qidirfirst オプションを #include "*file_name*" ディレクティブと共に使用すると、**-I***directory* オプションを使用して指定されたディレクトリーを他のディレクトリーを検索する前に検索します。

ディレクトリー検索パスを指定するには、**-I** オプションを使用します。

リンク

リンケージ・エディターは、指定されたオブジェクト・ファイルをリンク・エディットして、1 つの実行可能ファイルを作成します。 1 つ以上の呼び出しコマンド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、自動的にリンケージ・エディタ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、リンケージ・エディターの入力データとして使用できます。オブジェクト・ファイルにはサフィックスが必要です (例えば、year.o など)。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、libold.a など)。動的ライブラリー・ファイル名は .so のサフィックスが付きます (例えば、libold.so など)。

出力ファイル

リンケージ・エディターは、*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用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は .so のファイル名拡張子を持ちます。

ld コマンドにより、明示的にリンケージ・エディタ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ケージ・エディター・オプションを設定し、デフォルトで幾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・エディットするには、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。

注: オブジェクト・ファイルをリンク・エディットする場合、**ld** コマンドの **-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。

関連情報

- 48 ページの『リンクを制御するオプション』
- 335 ページの『付録 A. 再配布可能ライブラリー』

リンクの順序

XL C/C++ は、次の順序でライブラリーをリンクします。

1. ユーザー .o ファイルおよびライブラリー
2. XL C/C++ ライブラリー
3. C++ 標準ライブラリー
4. C 標準ライブラリー

次の表は、″Hello World″ 型プログラムのリンク順序をより詳細に示しています。

示されているディレクトリー・パスは、特定のコンパイラー構成によっては異なる場合があります。ユーザーの特定のコンパイラー構成に固有の情報については、システムにインストールされているデフォルトの構成ファイルを参照してください。一般的なコンパイラーのデフォルト構成ファイルについては、 22 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』を参照してください。

ld コマンド・コンポーネント	**オプション**	**ld 引数**	**xldriver 属性**
ld	gcc, g++	ld	ld / ld_64
	xlc、xlC	ld	
例外処理パーソナリティー・ハンドラーを使用可能にする	all	--eh-frame-hdr	xldriver によってコマンド行に追加されるオプション
.ident ディレクティブを生成する	-Qn		
	これ以外の場合	-Qy	xldriver によってコマンド行に追加されるオプション
出力の種類	-shared -static	-shared	xldriver によってコマンド行に追加されるオプション
	-shared	-shared	
	-static	-static	
	これ以外の場合		
arch	32 ビット	-melf32ppclinux	xldriver によってコマンド行に追加されるオプション
	64 ビット	-mel64ppc	
動的ローダー	32 ビット !-shared !-static	-dynamic-linker /lib/ld.so.1	dynlib
	64 ビット !-shared !-static	-dynamic-linker /lib64/ld64.so.1	dynlib64
call to main( )	32 ビット !-shared	/usr/libcrt1.o	crt
	64 ビット !-shared	/usr/lib64/crt1.o	crt_64
	32 ビット !-shared -p	/usr/lib/gcrt1.o	mcrt
	32 ビット !-shared -pg		gcrt
	64 ビット !-shared -p	/usr/lib64/gcrt1.o	mcrt_64
	64 ビット !-shared -pg		gcrt_64
init/fini 関数 prolog	32 ビットすべて	/usr/lib/crti.o	crtp
	64 ビットすべて	/usr/lib64/crti.o	crtp_64
init/fini レジスター	-shared -static	crtbeginT.o	crtbegin_t / crtbegin_t_64
	-static		
	-shared	crtbeginS.o	crtbegin_s / crtbegin_s_64
	これ以外の場合	crtbegin.o	crtbegin / crtbegin_64

ld コマンド・コンポーネント	オプション	ld 引数	xldriver 属性
ライブラリー検索パス	32 ビット gcc	-L<*gcc*>/gcc-lib	gcc_libdirs
	64 ビット gcc	-L<*gcc64*>/gcc-lib	gcc_libdirs_64
	32 ビット g++	-L<*gcc*>/gcc-lib/powerpc-suse-linux-gnu/3.2 -L<*gcc*>/gcc-lib	gcc_libdirs
	64 ビット g++	-L*gcc64*/gcc-lib/powerpc64-linux-gnu/3.2 -L*gcc64*/gcc-lib	gcc_libdirs_64
ユーザー .o ファイルおよびライブラリー	all		
vacpp ライブラリー	all		libraries2 / libraries2_64
C++ 標準ライブラリー	g++	-lstdc++ -lm	gcc_cpp_libs / gcc_cpp_libs_64
C 標準ライブラリー	gcc -static -static -shared-libgcc -shared -static-libgcc	-lgcc -lgcc_eh -lc -lgcc -lgcc_eh	gcc_static_libs / gcc_static_libs_64
	g++ -shared-libgcc	-lgcc_s -lgcc -lc -lgcc_s -lgcc	gcc_shared_libs / gcc_shared_libs_64
	all		gcc_libs / gcc_libs_64
保管/復元ルーチン	all	crtsavres.o	crtsavres / crtsavres_64
init/fini run		crtend.o	crtend / crtend_64
	-shared	crtendS.o	crtend_s / crtend_s_64
init/fini 関数 epilog	all	/usr/lib/crtn.o	crte
		/usr/lib64/crtn.o	crte_64

コンパイラーのメッセージおよびリスト

以下の節では、コンパイル後にコンパイラーによって提供されるさまざまな報告方法について説明します。

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

コンパイラー・メッセージ

コンパイラーは C または C++ 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、該当するオプションが選択されている場合は、リスト表示ファイルに診断メッセージを発行します。

この節では、コンパイラーがコンパイル・エラーの記述に使用するいくつかの基本的な報告メカニズムについても概説します。

コンパイラーは C または C++ 言語に特定のメッセージを発行します。

C コンパイラー・オプション 203 ページの『-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 STDERR ファイルに組み込まれます。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。

200 ページの『-qsource』コンパイラー・オプションを指定すると、コンパイラーはメッセージをソース・リストに入れます。例えば、コマンド行呼び出し **xlc -qsource filename.c** を使用してファイルをコンパイルすると、現行ディレクトリーで filename.lst と呼ばれるファイルが検出されます。

95 ページの『-qflag』 オプションまたは 231 ページの『-w』オプションを指定すると、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情報メッセージを得るには、 114 ページの『-qinfo』オプションを使用します。

コンパイラー・メッセージ・フォーマット

-qnosrcmsg オプションがアクティブ (これがデフォルト) の場合、診断メッセージのフォーマットは以下の通りです。

"*file*"、line *line_number.column_number*: 15*dd-nnn* (*severity*) *text*

ここで、

file エラーのある C または C++ ソース・ファイルの名前です。
line_number エラーの行番号です。
column_number エラーの列番号です。
15 コンパイラー製品 ID です。
dd このメッセージを発行した XL C/C++ コンポーネントを示す 2 桁のコードです。*dd* は、以下の値のいずれかになります。
- **00** - メッセージを生成または最適化するコード
- **01** - コンパイラー・サービス・メッセージ
- **05** - C コンパイラーに特定のコンパイラー・サービス・メッセージ
- **06** - C コンパイラーに特定のコンパイラー・サービス・メッセージ
- **40** - C++ コンパイラーに特定のコンパイラー・サービス・メッセージ
- **86** - プロシージャー間分析 (IPA) に特定のメッセージ

nnn メッセージ番号です。
severity エラーの重大度を表す文字です。
text エラーを記述するメッセージです。

-qsrcmsg オプションが指定されている場合、診断メッセージのフォーマットは以下の通りです。

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

ここで、*x* はフィンガー行のフィンガーを示す文字です。

メッセージ重大度レベルとコンパイラー応答

XL C/C++ は診断メッセージに対して 5 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。すべてのエラーがコンパイルを停止するわけではありません。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コンパイラー・オプションを提供します。

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	C 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。
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。 • 回復不能エラーの前に報告されたその他のエラーを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 技術員に報告してください。

関連情報
- 機能カテゴリー別のオプションの要約: リストとメッセージ
- 機能カテゴリー別のオプションの要約: エラー検査とデバッグ

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のタイプの 1 つです。デバッグ援助機能として、コンパイラー・リストは、コンパイルで何が問題だったのか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。

リストを要求するには、**-qsource** オプションを使用します。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
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
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。このセクションは、**-qattr** と **-qxref** オプションが有効な場合にのみ作成されます。これらの各オプションは独立して、コンパイルで使用された ID に関するさまざまな情報を提供します。

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

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

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

コンパイラー戻りコード

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

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

- **-qhaltonmsg** コンパイラー・オプションによって指定されたメッセージが発行されない。

それ以外の場合、コンパイラーは以下の値の 1 つに戻りコードを設定します。

戻りコード	エラー・タイプ
1	**-qhalt** コンパイラー・オプションの設定よりも高い重大度レベルを持つエラーが検出された。
40	オプション・エラーまたは回復不能エラーが検出された。
41	構成ファイル・エラーが検出された。
249	ファイルが指定されていない。
250	メモリー不足のエラーが検出された。コンパイラーは、使用するためのメモリーをこれ以上割り振ることはできません。
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または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。

メッセージ・カタログ・エラー

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

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

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

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

その後、メッセージ・カタログと環境変数が適所にあり、正しいことを検証してください。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 IBM XL C/C++ 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ロケーションにインストールしていると想定した場合、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断するには、以下のコマンドによってカタログのすべてのファイル名をリストすることができます。

```
ls /opt/ibmcmp/vacpp/8.0/msg/$LANG/*.cat
```

ここで、LANG はシステムのロケールを指定する、システム上の環境変数です。

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

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

コンパイル中のページ・スペース・エラー

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

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

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

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

ページング・スペース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 3 章 コンパイラー・オプション参照

本章には、XL C/C++ で使用可能な個々のオプションの詳細記述があります。本章の始めには機能カテゴリー別のオプションの要約ビューがあります。最後に、互換性のあるハードウェアに関連したオプションの参照リストが提供されています。

機能カテゴリー別コンパイラー・オプションの要約

Linux プラットフォームで使用可能な XL C/C++ オプションは、そのオプションが提供する機能の本質または性質を基に、以下のカテゴリーにグループ化されています。

- 入力制御。受け入れられる言語機能、入力ファイルの検索パス。
- 出力制御。オブジェクト・コード、データのサイズと位置合わせ、出力ファイルのファイル名の特性。
- 最適化。最適化の事前定義レベル、特殊な最適化手法、コード・サイズ。
- エラー検査とデバッグ。自動変数のプロファイル作成と初期化のためのオプションが含まれます。
- リストとメッセージ。**-qsource** または **-qinfo** よりももっと専門化された出力を作成するためのオプションが含まれます。
- 互換性。旧版のコンパイラー、ハードウェアの特定の機能を復元します。
- 整数および浮動小数点の制御。丸めと `long long` 型と `floating-point` 型の処理を支持するオプション。
- リンク。リンケージ・エディターへの入出力のための検索パス。
- コンパイラーのカスタマイズ。内部コンパイラー操作の制御。例えばテンプレートの処理方法など。

リストした任意の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ページを参照してください。これらのページでは、以下を含むコンパイラー・オプションについてそれぞれ説明します。

- オプションの目的とその振る舞いに関する追加情報。特に注釈がない限り、オプションはすべて C と C++ のプログラム・コンパイルの両方に適用されます。
- コンパイラー・オプションのコマンド行構文。**構文**の見出しの下にある最初の行は、コマンド行または構成ファイルでの指定方法を示します。2 行目がある場合は、ソース・ファイルで使用する **#pragma options** キーワードです。
- コマンド行、構成ファイル、またはプログラム内のプラグマ・ディレクティブでオプションを指定しない場合のオプションのデフォルトの設定。

入力を制御するオプション

表 6. 標準準拠のオプション

オプション名	タイプ	デフォルト	説明
-qlanglvl	**-q***opt*	137 ページの『-qlanglvl』を参照してください。	コンパイルのために C または C++ 言語レベルを選択する。

表 7. 言語拡張のオプション

オプション名	タイプ	デフォルト	説明
-qaltivec	**-q***opt*	**-qnoaltivec**	VMX ベクトル・データ型に対するコンパイラー・サポートを使用可能にする。
-qasm	**-q***opt*	**-qasm=gcc**	`asm` ステートメントの解釈と後続のコードの生成を制御する。
-qdigraph	**-q***opt*	83 ページの『-qdigraph』を参照してください。	プログラム・ソースでの連字の文字シーケンスの使用を可能にする。
-qdollar	**-q***opt*	**-qnodollar**	ID の名前に $ シンボルを使用できるようにする。
-qkeyword	**-q***opt*	135 ページの『-qkeyword』を参照してください。	指定されたストリングをキーワードとして処理するか、ID として処理するかを制御する。
-qtrigraph	**-q***opt*	**-qtrigraph**	プログラム・ソースでの 3 文字表記の文字シーケンスの使用を可能にする。
-qutf	**-q***opt*	**-qnoutf**	UTF リテラル構文の認識を使用可能にする。

表 8. 検索パスのオプション

オプション名	タイプ	デフォルト	説明
C -qc_stdinc	**-q***opt*	-	C ヘッダーの標準検索ロケーションを変更する。
C++ -qcinc	**-q***opt*	**-qnocinc**	`extern "C" { }` ラッパーを組み込みファイルのコンテンツのまわりに配置するようコンパイラーに命令する。
-qcomplexgccincl	**-q***opt*	**-qcomplexgccincl =/usr/include**	指定されたディレクトリーにある組み込みファイルのまわりの **#pragma complexgcc(on)** および **#pragma complexgcc(pop)** ディレクティブを内部的にラップするようにコンパイラーに命令する。
C++ -qcpp_stdinc	**-q***opt*	-	C++ ヘッダーの標準検索ロケーションを変更する。
C -qgcc_c_stdinc	**-q***opt*	-	gcc ヘッダーの標準検索ロケーションを変更する。
C++ -qgcc_cpp_stdinc	**-q***opt*	-	g++ ヘッダーの標準検索ロケーションを変更する。
-I	**-***flag*	-	絶対パスを指定しない `#include` ファイル名の追加の検索パスを指定する。
-qidirfirst	**-q***opt*	**-qnoidirfirst**	`#include "` *file_name*`"` ディレクティブで組み込むファイルの検索順序を指定する。

表 8. 検索パスのオプション (続き)

オプション名	タイプ	デフォルト	説明
-qstdinc	**-q***opt*	**-qstdinc**	`#include < file_name>` および `#include " file_name"` ディレクティブで組み込むファイルを指定する。

表 9. その他の入力オプション

オプション名	タイプ	デフォルト	説明
C++ -+ (正符号)	**-***flag*	-	任意のファイル *filename.nnn* を C++ 言語ファイルとしてコンパイルする。ここで、*nnn* は .o、.a、または .s 以外のサフィックスです。
-C	**-***flag*	-	プリプロセスされた出力にコメントを保持する。
C -qcpluscmt	**-q***opt*	74 ページの『-qcpluscmt』を参照してください。	C ソース・ファイルでの C++ コメントの認識を可能にする。
-D	**-***flag*	-	ID *name* を `#define` プリプロセッサー・ディレクティブ内と同じに定義する。
-qmbcs、-qdbcs	**-q***opt*	**-qnombcs**、**-qnodbcs**	ソース・コードでのマルチバイト文字の認識を可能にする。
-qignprag	**-q***opt*	-	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。
C -qsyntaxonly	**-q***opt*	-	コンパイラーに、オブジェクト・ファイルを生成せずに構文検査を行わせる。
-qsourcetype	**-q***opt*	**-qsourcetype=default**	実際のソースのファイル名サフィックスに関係なく、すべてのソース・ファイルをこのオプションによって指定されたソース・タイプとして扱うようコンパイラーに命令します。
-U	**-***flag*	-	コンパイラーまたは **-D** オプションによって定義された ID のうち指定した ID の定義を解除する。

出力を制御するオプション

表 10. ファイル出力のオプション

オプション名	タイプ	デフォルト	説明
-E	**-***flag*	-	ソース・ファイルをプリプロセスするようにコンパイラーに命令する。
-M	**-***flag*	-	**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。

表 10. ファイル出力のオプション (続き)

オプション名	タイプ	デフォルト	説明
-o	*-flag*	-	コンパイラーによって作成されるオブジェクト・ファイル、アセンブラー・ファイル、または実行可能ファイルの出力位置を指定する。
-P	*-flag*	-	コンパイラー呼び出しに指定された C または C++ ソース・ファイルをプリプロセスし、それぞれの入力ソース・ファイルごとにプリプロセスされた出力ソース・ファイルを作成する。
-S	*-flag*	-	ソース・ファイルごとにアセンブリー言語ファイル (**.s**) を生成する。
-s	*-flag*	-	シンボル・テーブルをストリップする。
-qfuncsect	**-q***opt*	**-qnofuncsect**	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。
-qmakedep	**-q***opt*	-	**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。
-qppline	**-q***opt*	**-qppline**	プリプロセスされた出力で `#line` ディレクティブの生成を使用可能にする。

表 11. 符号のオプション

オプション名	タイプ	デフォルト	説明
-qbitfields	**-q***opt*	**-qbitfields=signed**	ビット・フィールドを符号付きにするかどうかを指定する。
-qchars	**-q***opt*	**-qchars=unsigned**	`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ようコンパイラーに命令する。
C -qupconv	**-q***opt*	**-qnoupconv**	整数拡張を行うときに符号なしの指定を保持する。

表 12. データ・サイズと位置合わせのオプション

オプション名	タイプ	デフォルト	説明
-qalign	**-q***opt*	**-qalign=linuxppc**	コンパイラーがファイルのコンパイルに使用する集合体の位置合わせ規則を指定する。
-qenum	**-q***opt*	89 ページの『-qenum』を参照してください。	列挙が占有するストレージの量を指定する。

表 13. オブジェクト・コードの特性を制御するオプション

オプション名	タイプ	デフォルト	説明
-qenablevmx	**-q***opt*	**-qenablevmx**	サポートされるアーキテクチャー上で、VMX (Vector Multimedia Extension) の命令の生成を使用可能にする。
-qpic	**-q***opt*	**-qnopic**	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ようにコンパイラーに命令する。
C -qreserved_reg	**-q***opt*	-	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す。
C++ -qstaticinline	**-q***opt*	**-qnostaticinline**	インライン関数を静的として扱う。
-qstatsym	**-q***opt*	**-qnostatsym**	永続ストレージ・クラスを持つユーザー定義の非外部名を名前リストに追加する。
C++ -qvftable	**-q***opt*	**-qvftable**	仮想関数テーブルの生成を制御する。
-qvrsave	**-q***opt*	**-qvrsave**	VRSAVE レジスターの保守に必要な関数のプロローグおよびエピローグ・コードを制御する。
-qxcall	**-q***opt*	**-qnoxcall**	コンパイル単位内の静的ルーチンを、外部呼び出しのように扱うためのコードを生成する。

表 14. ストリングおよび定数データの配置を制御するオプション

オプション名	タイプ	デフォルト	説明
-qro	**-q***opt*	190 ページの『-qro』を参照してください。	ストリング・リテラルのストレージ・タイプを指定する。
-qroconst	**-q***opt*	191 ページの『-qroconst』を参照してください。	定数値のストレージ・ロケーションを指定する。

表 15. その他の出力オプション

オプション名	タイプ	デフォルト	説明
-# (ポンド記号)	**-***flag*	-	何も行わずにコンパイルをトレースする。
-c	**-***flag*	-	ソース・ファイルをリンケージ・エディターに送信せずにコンパイラーに渡すようコンパイラーに命令する。
-q32、-q64	**-q***opt*	**-q32**	32 ビットまたは 64 ビットのコンパイラー・モードを選択する。

表 15. その他の出力オプション (続き)

オプション名	タイプ	デフォルト	説明
C -qalloca	**-q***opt*	-	**#pragma alloca** ディレクティブがソース・コードにあるかのように、呼び出しのインライン・コードを関数 **alloca** に置換する。
C++ -qrtti	**-q***opt*	**-qrtti**	typeid 演算子および dynamic_cast 演算子に対する実行時型識別 (RTTI) 情報を生成する。
-qsaveopt	**-q***opt*	**-qnosaveopt**	コンパイラー・オプションをオブジェクト・ファイルに保管する。
-qthreaded	**-q***opt*	217 ページの『-qthreaded』を参照してください。	プログラムがマルチスレッド環境で稼働することを示す。

パフォーマンス最適化のオプション

表 16. 定義されている最適化レベルのオプション

オプション名	タイプ	デフォルト	説明
-O、-qoptimize、-qoptimize	**-***flag*、**-q***opt*	**-qnooptimize**	コンパイル中に、選択したレベルでコードを最適化する。

表 17. ABI パフォーマンス・チューニングのオプション

オプション名	タイプ	デフォルト	説明
-qdataimported	**-q***opt*	-	インポートとしてデータにマークを付ける。
-qdatalocal	**-q***opt*	-	ローカルとしてデータにマークを付ける。
-qlibansi	**-q***opt*	**-qnolibansi**	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。
-qminimaltoc	**-q***opt*	**-qnominimaltoc**	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 を入れることによって、64 ビット・コンパイルでの TOC オーバーフローを阻止する。
-qproclocal、-qprocimported、-qprocunknown	**-q***opt*	182 ページの『-qproclocal、-qprocimported、-qprocunknown』を参照してください。	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。
-qtocdata	**-q***opt*	**-qnotocdata**.	アプリケーションが使用するスレッド局在のストレージ・モデルを指定する。
-qunwind	**-q***opt*	**-qunwind**	アプリケーションがどのプログラム・スタックのアンワインド・メカニズムにも依存しないことをコンパイラーに伝える。

表 18. 最適化を制限するオプション

オプション名	タイプ	デフォルト	説明
-qprefetch	**-q***opt*	**-qprefetch**	コンパイル・コードでプリフェッチ命令の生成を使用可能にする。
-qsmallstack	**-q***opt*	**-qnosmallstack**	スタック・フレームのサイズを削減するようコンパイラーに命令する。
-qspill	**-q***opt*	**-qspill=512**	レジスター割り振り予備域のサイズを指定する。
-qstrict	**-q***opt*	207 ページの『-qstrict』を参照してください。	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** オプションによる積極的な最適化をオフにする。

表 19. プロセッサーおよびアーキテクチャー最適化のオプション

オプション名	タイプ	デフォルト	説明
-qarch	**-q***opt*	**-qarch=ppc64grsq**	実行可能プログラムを実行するアーキテクチャーを指定する。
-qcache	**-q***opt*	-	特定の実行マシンのキャッシュ構成を指定する。
-qdirectstorage	**-q***opt*	**-qnodirectstorage**。	ライトスルーが使用可能になっていること、あるいはキャッシュが抑制されたストレージが参照される可能性のあることをコンパイラーに通知する。
-qtune	**-q***opt*	223 ページの『-qtune』を参照してください。	実行可能プログラムの最適化対象とするアーキテクチャーを指定する。

表 20. ループ最適化のオプション

オプション名	タイプ	デフォルト	説明
-qhot	**-q***opt*	**-qnohot**	最適化中に高位ループ分析および変換を実行するようコンパイラーに命令する。
-qstrict_induction	**-q***opt*	208 ページの『-qstrict_induction』を参照してください。	プログラムのセマンティクスを変更する可能性のあるループ帰納変数の最適化を使用不可にする。
-qunroll	**-q***opt*	**-qunroll=auto**	プログラムの内部ループをアンロールする。

表 21. コード・サイズ縮小のオプション

オプション名	タイプ	デフォルト	説明
-qcompact	**-q***opt*	**-qnocompact**	最適化と共に使用すると、可能な場合に、実行速度を犠牲にしてコード・サイズを削減する。
C++ -qeh	**-q***opt*	**-qeh**	例外処理を制御する。

表 21. コード・サイズ縮小のオプション (続き)

オプション名	タイプ	デフォルト	説明
C++ -qkeepinlines	**-q***opt*	**-qnokeepinlines**	参照されていない外部インライン関数の定義を保持または廃棄するようにコンパイラーに命令する。

表 22. プログラム全体の分析のオプション

オプション名	タイプ	デフォルト	説明
-qipa	**-q***opt*	122 ページの『-qipa』を参照してください。	プロシージャー間分析 (IPA) と呼ばれる最適化のクラスをオンにしたりカスタマイズする。

表 23. 関数のインライン化のオプション

オプション名	タイプ	デフォルト	説明
-qinline	**-q***opt*	119 ページの『-qinline』を参照してください。	関数の呼び出しを生成する代わりに、関数のインライン化を試みる。
-Q	**-***flag*	185 ページの『-Q』を参照してください。	関数のインライン化を試行する。

表 24. パフォーマンス・データ収集のオプション

オプション名	タイプ	デフォルト	説明
-qpdf1、-qpdf2	**-q***opt*	**-qnopdf1**、**-qnopdf2**	プロファイル指示のフィードバックによって最適化を調整する。
-qshowpdf	**-q***opt*	**-qnoshowpdf**	**-qpdf1** および最小の **-O** と共に使用された場合、 追加の呼び出しとブロック・カウントのプロファイル情報を実行可能ファイルに追加する。

エラー検査とデバッグのオプション

表 25. デバッグのオプション

オプション名	タイプ	デフォルト	説明
-g	**-***flag*	-	デバッガーによって使用されるデバッグ情報を生成する。
C -qdbxextra	**-q***opt*	**-qnodbxextra**	すべての `typedef` 宣言、`struct` 型宣言、`union` 型宣言、および `enum` 型宣言をデバッガー処理のために組み込むことを指定する。
-qfullpath	**-q***opt*	**-qnofullpath**	**-g** オプションを使用したときにファイルに対して保管されるパス情報を指定する。
-qlinedebug	**-q***opt*	**-qnolinedebug**	デバッガーのために、省略された行番号およびソース・ファイル名の情報を生成する。

表 25. デバッグのオプション (続き)

オプション名	タイプ	デフォルト	説明
C -qsymtab	**-q***opt*	-	シンボル・テーブルに表示する情報を決定する。

表 26. プロファイルのオプション

オプション名	タイプ	デフォルト	説明
-p	**-***flag*	-	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設定する。
-pg	**-***flag*	-	プロファイル用にオブジェクト・ファイルを設定する。

表 27. その他のエラー検査とデバッグ・オプション

オプション名	タイプ	デフォルト	説明
C -qgenproto	**-q***opt*	**-qnogenproto**	K&R 関数定義から ANSI プロトタイプを生成する。
-qinitauto	**-q***opt*	**-qnoinitauto**	自動ストレージを指定された 2 桁の 16 進バイト値に初期化する。
-qkeepparm	**-q***opt*	**-qnokeepparm**	アプリケーションが最適化されていても、関数仮パラメーターがスタックに確実に保管されるようにする。
C -qproto	**-q***opt*	**-qnoproto**	すべての関数がプロトタイプ化されていると想定する。
-qtbtable	**-q***opt*	213 ページの『-qtbtable』を参照してください。	トレースバック・テーブルの特性を設定する。

リストとメッセージを制御するオプション

表 28. リストのオプション

オプション名	タイプ	デフォルト	説明
-qattr	**-q***opt*	**-qnoattr**	全 ID の属性リストを含むコンパイラー・リストを生成する。
-qdump_class_hierarchy	**-q***opt*	-	継承のクラス・レイアウトと構造を標準エラーに出力する。
-qlist	**-q***opt*	**-qnolist**	オブジェクト・リストを含むコンパイラー・リストを生成する。
-qlistopt	**-q***opt*	**-qnolistopt**	有効なオプションをすべて示すコンパイラー・リストを作成する。
-qprint	**-q***opt*	**-qprint**	**-qnoprint** はリストを抑制する。

表 28. リストのオプション (続き)

オプション名	タイプ	デフォルト	説明
-qshowinc	**-q***opt*	**-qnoshowinc**	**-qsource** と共に使用した場合に、プログラム・ソース・リストにユーザー・ヘッダー・ファイル (" " の使用を含む) またはシステム・ヘッダー・ファイル(< > の使用を含む) を選択して表示する。
-qsource	**-q***opt*	**-qnosource**	コンパイラー・リストを作成してソース・コードを組み込む。
-qtabsize	**-q***opt*	**-qtabsize=8**	コンパイラーによって認識されるタブの長さを変更する。
-qxref	**-q***opt*	**-qnoxref**	すべての ID の相互参照リストを含むコンパイラー・リストを生成する。

表 29. メッセージのオプション

オプション名	タイプ	デフォルト	説明
-qflag	**-q***opt*	**-qflag=i:i**	報告する診断メッセージの最小の重大度レベルを指定する。
-qformat	**-q***opt*	101 ページの『-qformat』を参照してください。	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。
-qhalt	**-q***opt*	**-qhalt=s**	指定した重大度 以上のエラーが検出された場合に、コンパイル・フェーズ後に停止するようにコンパイラーに命令する。
C++ -qhaltonmsg	**-q***opt*	-	特定のエラー・メッセージが検出された場合、コンパイル・フェーズ後に停止するようコンパイラーに命令する。
-qinfo	**-q***opt*	C **-qnoinfo** C++ **-qinfo=lan:trx**	通知メッセージを作成する。
-qphsinfo	**-q***opt*	**-qnophsinfo**	各コンパイル・フェーズでかかった時間を報告する。
-qreport	**-q***opt*	**-qnoreport**	プログラム・ループが並列化され、最適化される方法を示す変換レポートを作成するようコンパイラーに命令する。

表 29. メッセージのオプション (続き)

オプション名	タイプ	デフォルト	説明
C -qsrcmsg	**-q***opt*	**-qnosrcmsg**	対応するソース・コード行を stderr ファイル内の診断メッセージに追加する。
-qsuppress	**-q***opt*	209 ページの『-qsuppress』を参照してください。	抑制するコンパイラー・メッセージ番号を指定する。
-qversion	**-q***opt*	**-qnoversion**	呼び出しているコンパイラーのバージョンを表示する。
-V	**-***flag*	-	コンパイルの進行に関する情報をコマンドに似た形式で報告するようにコンパイラーに命令する。
-v	**-***flag*	-	コンパイルの進行に関する情報を報告するようにコンパイラーに命令する。
-w	**-***flag*	-	警告メッセージの抑止を要求する。

互換性のオプション

表 30. 互換性のオプション

オプション名	タイプ	デフォルト	説明
C++ -qabi_version	**-q***opt*	52 ページの『-qabi_version』を参照してください。	異なるレベルの GNU C++ とのバイナリー互換性のために C++ ABI バージョンを指定する。

整数および浮動小数点処理を制御するオプション

表 31. 整数および浮動小数点の制御のオプション

オプション名	タイプ	デフォルト	説明
-qfloat	**-q***opt*	96 ページの『-qfloat』を参照してください。	浮動小数点演算の速度や精度を上げるためにさまざまな浮動小数点オプションを指定する。
-qflttrap	**-q***opt*	**-qnoflttrap**	追加の命令を生成し、浮動小数点例外を検出してトラップする。
-qlonglit	**-q***opt*	**-qnolonglit**	サフィックスなしのリテラルを 64 ビット・モードの long 型にする。
-qlonglong	**-q***opt*	157 ページの『-qlonglong』を参照してください。	プログラムで long long 型を許可する。
-y	**-***flag*	**-yn**	浮動小数点定数式のコンパイル時丸めモードを指定する。

リンクを制御するオプション

表 32. リンカー入力制御のオプション

オプション名	タイプ	デフォルト	説明
-qbigdata	**-q***opt*	**-qnobigdata**	32 ビット・モードで、初期化されたデータのサイズが 16 MB を超えることを許可する。
-e	**-***flag*	-	共用オブジェクトの項目名を指定する。 **ld -e** *name* を使用する場合と同等。
-L	**-***flag*	136 ページの『-L』を参照してください。	指定されたディレクトリーで、リンク時に、**-l** オプションで指定されたライブラリー・ファイルを検索する。
-l	**-***flag*	136 ページの『-l』を参照してください。	リンクのために指定したライブラリーを検索する。
-qlib	**-q***opt*	**-qlib**	リンク時に標準システム・ライブラリーを使用するようコンパイラーに命令する。
-R	**-***flag*	188 ページの『-R』を参照してください。	共用ライブラリーについて、指定したディレクトリーを実行時に検索する。

表 33. リンカー出力制御のオプション

オプション名	タイプ	デフォルト	説明
-qmkshrobj	**-q***opt*	-	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。
-qstaticlink	**-q***opt*	**-qnostaticlink**	共用ライブラリーへのリンクを制御する。
-r	**-***flag*	-	再配置可能オブジェクトを作成する。

表 34. その他のリンカー・オプション

オプション名	タイプ	デフォルト	説明
-qcrt	**-q***opt*	**-qcrt**	リンク時に標準システム始動を使用するようリンケージ・エディターに命令する。
-qinlglue	**-q***opt*	**-qnoinlglue**	外部関数の呼び出しまたは関数ポインターを介した呼び出しに必要なポインター・グルー・コードをインライン化することにより、高速な外部結合を生成する。
C++ -qpriority	**-q***opt*	**-qpriority=65535**	静的オブジェクトの初期化の優先順位を指定する。

コンパイラーをカスタマイズするためのオプション

表 35. 一般のカスタマイズのオプション

オプション名	タイプ	デフォルト	説明
-B	**-***flag*	-	コンパイラー、アセンブラー、リンケージ・エディター、およびプリプロセッサーに対する代替パス名を判別する。
-F	**-***flag*	-	コンパイラーの代替構成ファイルを指定する。
-qasm_as	**-q***opt*	-	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。
-qmaxmem	**-q***opt*	**-qmaxmem=8192**	メモリーを大量に消費する特定の最適化のローカル・テーブルに使用するメモリーの量を制限する。
-qpath	**-q***opt*	-	代替プログラム名およびパス名を構成する。
-t	**-***flag*	211 ページの『-t』を参照してください。	**-B** オプションによって指定されたプレフィックスを、指定したプログラムに追加する。
-W	**-***flag*	-	リストされたオプションを指定したコンパイラー・プログラムに渡す。
-qtls	**-q***opt*	218 ページの『-qtls』を参照してください。	ローカルとしてデータにマークを付ける。

表 36. テンプレートに関連するオプション

オプション名	タイプ	デフォルト	説明
C++ -qtempinc	**-q***opt*	**-qnotempinc**	テンプレート関数とクラス宣言に対して別々の組み込みファイルを生成し、オプションで指定できるディレクトリーにそれらのファイルを配置する。
C++ -qtemplaterecompile	**-q***opt*	215 ページの『-qtemplaterecompile』を参照してください。	**-qtemplateregistry** コンパイラー・オプションを使用してコンパイルされたコンパイル単位間の依存性の管理に役立つ。
C++ -qtemplateregistry	**-q***opt*	**-qnotemplateregistry**	ソース内で検出されるすべてのテンプレートについてそのレコードを保守し、テンプレートごとに一度だけインスタンス化が行われることを保証する。
C++ -qtempmax	**-q***opt*	**-qtempmax=1**	各ヘッダー・ファイルごとに、**tempinc** オプションによって生成されるテンプレート組み込みファイルの最大数を指定する。
C++ -qtmplinst	**-q***opt*	**-qtmplinst=auto**	テンプレートの暗黙のインスタンス生成を管理する。

表 36. テンプレートに関連するオプション (続き)

オプション名	タイプ	デフォルト	説明
C++ -qtmplparse	**-q***opt*	**-qtmplparse=no**	構文解析とセマンティック検査がテンプレート定義のインプリメンテーションに適用されるかどうかを制御する。

個々のオプションの説明

この節では、XL C/C++ で使用可能な個々のコンパイラー・オプションについて説明します。

-+ (正符号)

C++

説明

任意のファイル *filename.nnn* を C++ 言語ファイルとしてコンパイルする。ここで、*nnn* は .a、.o、.so、.S または .s 以外のサフィックスです。

構文

注

-+ オプションを使用しない場合、ファイルを C++ ファイルとしてコンパイルするためには、.C (大文字の C)、.cc、.cp、.cpp、.cxx、または .c++ のサフィックスが必要です。.c (小文字の c) のサフィックスの付いたファイルを **-+** を指定せずにコンパイルすると、ファイルは C 言語ファイルとしてコンパイルされます。

-+ オプションは、**-qsourcetype** オプションと一緒に使用しないでください。

例

ファイル `myprogram.cplspls` を C++ ソース・ファイルとしてコンパイルするには、以下のように入力します。

```
xlc++ -+ myprogram.cplspls
```

関連情報

- 201 ページの『-qsourcetype』
- 入力を制御するオプション: その他の入力オプション

-# (ポンド記号)

説明

何も行わずにコンパイルをトレースする。このオプションは、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します。 このオプションを指定して **xlc++** コマンドが発行されると、呼び出されるプリプロセッサー、コンパイラー、および

リンケージ・エディター内のプログラムの名前と各プログラムに指定されるオプションが示されます。プリプロセッサー、コンパイラー、およびリンケージ・エディターは呼び出されません。

構文

注

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します。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。情報は標準出力に出力されます。

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。

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

- 228 ページの『-v』
- 出力を制御するオプション: その他の出力オプション

-q32、-q64

説明

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。

構文

注

このオプションがコマンド行に明示的に指定されていない場合、コンパイラーは 32 ビット出力モードをデフォルトとして使用します。

コンパイラー 64 ビット・モードで呼び出されると、__64BIT__ プリプロセッサー・マクロが定義されます。

-q32 および **-q64** オプションを **-qarch** および **-qtune** コンパイラー・オプションと共に使用して、コンパイラーの出力が使用されるアーキテクチャーに合わせてその出力を最適化します。

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
```

重要!

- 異なるソース・ファイルに 32 ビットと 64 ビットのコンパイル・モードを混用すると、 オブジェクトがバインドされません。完全に再コンパイルして、オブジェクトをすべて同じモードにしなければなりません。
- リンク・オプションは、リンクしているオブジェクトのタイプを反映していなければなりません。 64 ビット・オブジェクトをコンパイルした場合は、64 ビット・モードを使用してこれらのオブジェクトをリンクしなければなりません。

関連情報

- 58 ページの『-qarch』
- 223 ページの『-qtune』
- 234 ページの『コンパイラー・モードとプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ』
- 出力を制御するオプション: その他の出力オプション

-qabi_version

C++

説明

異なるレベルの GNU C++ とのバイナリー互換性のために C++ ABI バージョンを指定する。

構文

ここで、

1 GNU C++ 3.2 と同じ C++ ABI の振る舞いを指定します。

2 このバージョンが オペレーティング・システムによってサポートされている場合、GNU C++ 3.4 と同じ C++ ABI の振る舞いを指定します。

注

オプション **-qabi_version** は GNU C++ オプション -fabi-version=*n* との互換性のために提供されています。これによりユーザーはコンパイル中に使用される C++ 抽象バイナリー・インターフェースのバージョンを指定することができます。**-qabi_version** のデフォルト設定はコンパイル・マシン自体と XL C++ のインストール中に構成される GNU C++ のレベルに依存します。GNU C++ 3.2 または 3.3 がコンパイル・マシンにインストールされている場合、デフォルトは **-qabi_version=1** です。

通知メッセージ

-qabi_version の値は **-qlistopt** を有効にしてコンパイルを行うことで確かめることができます。

関連情報
- 155 ページの『-qlistopt』
- 互換性のオプション

-qaggrcopy

説明
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


デフォルト設定
-qlanglvl=extended または **-qlanglvl=classic** を有効にしたコンパイル時のこのオプションのデフォルト設定は **-qaggrcopy=overlap** です。それ以外の場合、デフォルトは **-qaggrcopy=nooverlap** です。

ソースと宛先の割り当てがオーバーラップしていないために ANSI C 標準に準拠しないプログラムは、**-qaggrcopy=overlap** コンパイラー・オプションを指定したコンパイルが必要な可能性があります。

注
-qaggrcopy=nooverlap コンパイラー・オプションが使用可能な場合、コンパイラーは、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します。この前提事項によって、コンパイラーはより速いコードを生成します。

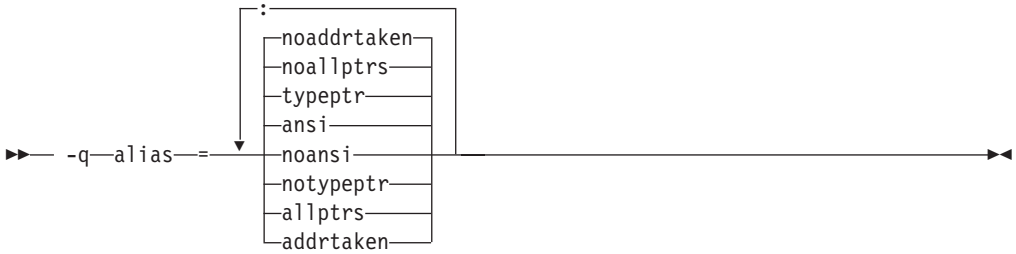
例
```
xlc myprogram.c -qaggrcopy=nooverlap
```

関連情報
- 137 ページの『-qlanglvl』
- コマンド行オプションの要約: 最適化フラグ

-qalias

説明
別名割り当て表明をコンパイル単位に適用するようコンパイラーに命令する。特に指定されない限り、コンパイラーは別名割り当て表明を利用して、可能な箇所で最適化の改善を行います。

構文



別名割り当ての使用可能オプションは、以下の通りです。

[no]typeptr
: **notypeptr** が指定されている場合、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。つまり、コンパイル単位内で異なる型の 2 つのポインターが同じストレージ・ロケーションを指すことはありません。

[no]allptrs
: **noallptrs** が指定されている場合、ポインターに別名が割り当てられることはありません (また、**-qalias=typeptr** を暗黙指定します)。したがって、コンパイル単位内で、2 つのポインターが同じストレージ・ロケーションを指すことはありません。

[no]addrtaken
: **noaddrtaken** が指定されている場合、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。

[no]ansi
: **ansi** が指定された場合、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することが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。これ (**ansi**) が **xlc**、**xlc++**、**xlC**、 **c89** および **c99** 呼び出しコマンドのデフォルトです。このオプションは、**-O** オプションも指定しない限り無効です。

  noansi を選択すると、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、またはアドレスがすで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。これが **cc** 呼び出しコマンドのデフォルトです。

注

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。

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

- 254 ページの『#pragma disjoint』
- パフォーマンス最適化のオプション: 別名割り当てのオプション

-qalign

説明

コンパイラーがファイルのコンパイルに使用する集合体の位置合わせ規則を指定する。このオプションを使用して、ソース・プログラム全体または特定の部分のいずれかについて、クラス型オブジェクトのマップ時に使用する最大の位置合わせを指定します。

構文

使用可能な位置合わせのオプションは、以下のとおりです。

linuxppc	コンパイラーはデフォルトの GNU C/C++ 位置合わせ規則を使用して、GNU C/C++ オブジェクトとの互換性を保持します。これがデフォルトです。
bit_packed	コンパイラーは、**bit_packed** の位置合わせ規則を使用します。 このサブオプションは、GCC の **-fpack-struct** オプションに似ています。

注

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。

#pragma align(*alignment_rule***)** を使用して **-qalign** コンパイラー・オプションの設定をオーバーライドすることにより、コードのサブセットの位置合わせを制御することができます。**#pragma align(reset)** を使用すると、直前の位置合わせ規則に戻すことができます。コンパイラーは位置合わせディレクティブをスタックします。そのため、**#pragma align(reset)** ディレクティブを指定すると、直前の位置合わせディレクティブが何であったかを知らなくても、その規則に戻して使用することができます。例えば、組み込みファイル内にクラス宣言があり、そのクラスに対して指定した位置合わせ規則をクラスの組み込み先に適用したくない場合に、このオプションを使用することができます。

例

例 1 - 集合体の定義にのみ影響

以下のコンパイラー呼び出しを使用します。

```
xlc++ file2.C /* <-- default alignment rule for file is                */
            /*     linuxppc because no alignment rule specified */
```

ここで、file2.C は以下を持ちます。

```
extern struct A A1;
typedef struct A A2;

#pragma options align=bit_packed /* <-- use bit_packed alignment rules*/
struct A {
```

```
  int a;
  char c;
};
#pragma options align=reset /* <-- Go back to default alignment rules */

struct A A1;  /* <-- aligned using bit_packed alignment rules since   */
A2 A3;        /*     this rule applied when struct A was defined      */
```

例 2 - 組み込みプラグマ

以下のコンパイラー呼び出しを使用します。

```
xlc -qalign=linuxppc file.c  /* <-- default alignment rule for file */
                            /*    is linuxppc                       */
```

ここで、file.c は以下を持ちます。

```
struct A {
  int a;
  struct B {
  char c;
    double d;
#pragma options align=bit_packed /* <-- B will be unaffected by this  */
                              /*     #pragma, unlike previous behavior; */
                              /*     linuxppc alignment rules still    */
                              /*     in effect                          */
  } BB;
#pragma options align=reset /* <-- A is unaffected by this #pragma;   */
} AA;                         /*     linuxppc alignment rules still    */
                              /*     in effect                          */
```

関連情報

- 246 ページの『#pragma align』
- 278 ページの『#pragma options』
- 284 ページの『#pragma pack』
- 出力を制御するオプション: データ・サイズと位置合わせのオプション
- 「*XL C/C++ 言語解説書*」の『__align 指定子』
- 「*XL C/C++ プログラミング・ガイド*」の『集合体でのデータの位置合わせ』
- 「*XL C/C++ 言語解説書*」の『位置合わせされた変数属性』
- 「*XL C/C++ 言語解説書*」の『パックされた変数属性』

-qalloca

C

説明

#pragma alloca ディレクティブがソース・コードにあるかのように、関数 alloca への呼び出しのインライン・コードを置換する。

構文

注

C **#pragma alloca** が指定解除されている場合、そして **-ma** を使用していない場合は、alloca は組み込み関数ではなく、ユーザー定義 ID として扱われます。

C++ C++ プログラムでは、__alloca 組み込み関数を使用してください。ソース・コードがすでに alloca を関数名として参照している場合は、コンパイラーを呼び出すときにコマンド行に以下のオプションを使用してください。

```
-Dalloca=__alloca
```

alloca の代わりに C99 可変長配列を使用することもできます。

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

- 247 ページの『#pragma alloca』
- 80 ページの『-D』
- 159 ページの『-ma』
- 出力を制御するオプション: その他の出力オプション

-qaltivec

説明

Vector データ型に対するコンパイラー・サポートを使用可能にする。

構文

注

このオプションはコンパイラーに vector データ型と演算子をサポートするよう命令し、**-qarch** が、VMX 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 (これは現在サポートされる Linux ディストリビューションではデフォルトで有効です)。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。

-qaltivec が有効な場合は、以下のマクロが定義されます。

- __ALTIVEC__ が 1 に定義される。
- __VEC__ が 10205 に定義される。

例

Vector 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  -qarch=ppc64v -qaltivec
```

関連情報

- 247 ページの『#pragma altivec_vrsave』
- 58 ページの『-qarch』
- 89 ページの『-qenablevmx』

- 「*XL C/C++ 言語解説書*」の『付録 C. Vector データ型およびリテラル』
- 「*AltiVec Technology Programming Interface Manual*」(http://www.freescale.com より入手可能)
- 入力を制御するオプション: 言語拡張のオプション

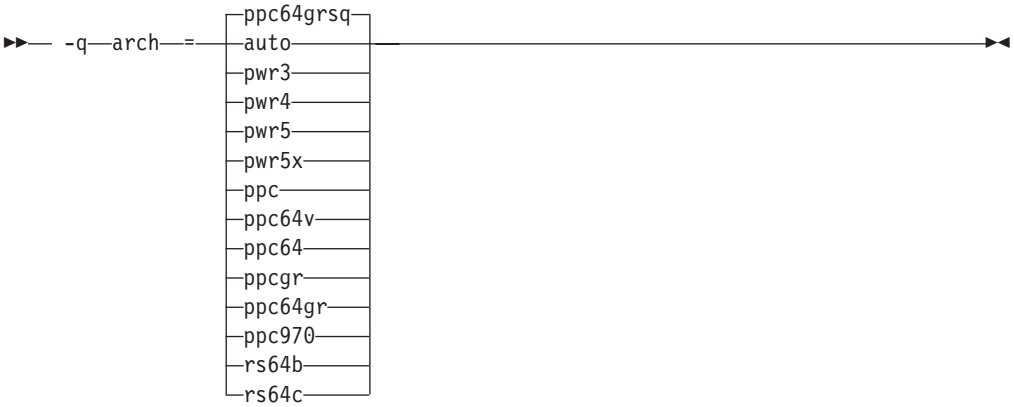
-qarch

説明

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。

一般に、**-qarch** オプションにより、コンパイルのための特定のアーキテクチャーをターゲットとして指定することができます。どの **-qarch** 設定についても、コンパイラーは特定のマッチングする **-qtune** 設定をデフォルトとして使用します。それによりさらにパフォーマンスが改善されます。結果コードは他のアーキテクチャーで稼働しない可能性がありますが、選択されたアーキテクチャーでは最良のパフォーマンスを提供します。複数のアーキテクチャー上で実行できるコードを生成するには、**ppc**、または **ppc64** などの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。こうすると、すべてのサポートされるアーキテクチャー、すべての PowerPC アーキテクチャー、またはすべての 64 ビット PowerPC アーキテクチャーでそれぞれ実行されるコードが生成されます。**-qarch** サブオプションをグループ引数と一緒に指定する場合は、**-qtune** を **auto** として指定するか、またはグループ内の特定のアーキテクチャーを提供することができます。**-qtune=auto** の場合、コンパイラーは **-qarch** サブオプションによって指定されたグループ内のすべてのアーキテクチャー上で実行されるコードを生成しますが、コンパイルに使用されるマシンのアーキテクチャー上で最高のパフォーマンスを提供する命令シーケンスを選択してください。あるいは、パフォーマンスのチューニングのために、特定のアーキテクチャーをターゲットにすることができます。

構文



ここで、使用可能なオプションは下記のようなプロセッサー・アーキテクチャーの幅広いファミリーまたはそれらのアーキテクチャー・ファミリーのサブグループを指定します。

auto
- **-O4** または **-O5** が設定または暗黙指定されている場合は、このオプションが暗黙指定されます。
- コンパイルが行われるハードウェア・プラットフォームで実行される命令を含むオブジェクト・コードを生成します。

pwr3
- どの POWER3™、POWER4™、POWER5™、POWER5+™ または PowerPC 970 ハードウェア・プラットフォームでも稼働する命令を含むオブジェクト・コードを作成します。
- _ARCH_PPC、 _ARCH_PPCGR、 _ARCH_PPC64、 _ARCH_PPC64GR、 _ARCH_PPC64GRSQ、および _ARCH_PWR3 マクロを定義します。

pwr4
- POWER4、POWER5、POWER5+、または PowerPC 970 ハードウェア・プラットフォームで稼働する命令を含むオブジェクト・コードを作成します。
- _ARCH_PPC、 _ARCH_PPCGR、 _ARCH_PPC64、 _ARCH_PPC64GR、 _ARCH_PPC64GRSQ、 _ARCH_PWR3、および _ARCH_PWR4 マクロを定義します。

pwr5
- POWER5 または POWER5+ ハードウェア・プラットフォームで稼働する命令を含むオブジェクト・コードを生成します。
- _ARCH_PPC、 _ARCH_PPCGR、 _ARCH_PPC64、 _ARCH_PPC64GR、 _ARCH_PPC64GRSQ、 _ARCH_PWR3、 _ARCH_PWR4、および _ARCH_PWR5 マクロを定義します。

pwr5x
- POWER5+ ハードウェア・プラットフォームで稼働する命令を含むオブジェクト・コードを作成します。
- _ARCH_PPC、 _ARCH_PPCGR、 _ARCH_PPC64、 _ARCH_PPC64GR、 _ARCH_PPC64GRSQ、 _ARCH_PWR3、 _ARCH_PWR4、 _ARCH_PWR5、および _ARCH_PWR5X マクロを定義します。

ppc
- 32 ビット・モードでは、32 ビット PowerPC ハードウェア・プラットフォームのすべてで実行される命令を含むオブジェクト・コードを生成します。このサブオプションによって、コンパイラーは単精度データで使用する単精度命令を生成します。
- _ARCH_PPC マクロを定義します。
- **-qarch=ppc** を **-q64** と共に指定すると、**-qarch=ppc64grsq** が暗黙指定されます。

ppc64
- どの 64 ビット PowerPC ハードウェア・プラットフォームでも稼働するオブジェクト・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。
- _ARCH_PPC、および _ARCH_PPC64 マクロを定義します。

ppcgr
- 32 ビット・モードでは、オプションのグラフィックス命令をサポートする PowerPC プロセッサー用のオブジェクト・コードを作成します。
- **-qarch=ppcgr** を **-q64** と一緒に指定すると、アーキテクチャーの設定が無音で **-qarch=ppc64grsq** にアップグレードされます。
- _ARCH_PPC、および _ARCH_PPCGR マクロを定義します。

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。
- _ARCH_PPC、_ARCH_PPCGR、_ARCH_PPC64、および _ARCH_PPC64GR マクロを定義します。

ppc64grsq
- オプションのグラフィックス命令と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。
- _ARCH_PPC、_ARCH_PPCGR、_ARCH_PPC64、 _ARCH_PPC64GR、および _ARCH_PPC64GRSQ マクロを定義します。

ppc64v
- VMX プロセッサー (PowerPC 970 など) を持つ汎用 PowerPC チップ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。
- _ARCH_PPC、_ARCH_PPCGR、_ARCH_PPC64、_ARCH_PPC64GR、_ARCH_PPC64GRSQ、_ARC マクロを定義します。

ppc970
- PowerPC 970 アーキテクチャーに特定の命令を生成します。
- _ARCH_PPC、_ARCH_PPC64V、_ARCH_PPCGR、_ARCH_PPC64、_ARCH_PPC970、_ARCH_PW および_ARCH_PPC64GRSQ マクロを定義します。

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。
- _ARCH_PPC、 _ARCH_PPCGR、 _ARCH_PPC64、 _ARCH_PPC64GR、 _ARCH_PPC64GRSQ、および _ARCH_RS64B マクロを定義します。

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。
- _ARCH_PPC、 _ARCH_PPCGR、 _ARCH_PPC64、 _ARCH_PPC64GR、 _ARCH_PPC64GRSQ、および _ARCH_RS64C マクロを定義します。

注

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。

-qarch=*suboption* は、 **-qtune=***suboption* と共に使用することができます。 **-qarch=***suboption* は、命令の生成対象とするアーキテクチャーを指定し、 **-qtune=***suboption* は、コードの最適化対象とするターゲット・プラットフォームを指定します。 **-qarch** が **-qtune** なしで指定されている場合、コンパイラーは指定されたアーキテクチャーのデフォルトのチューニング・オプションを使用し、リストには有効な **-qtune** 設定が表示されます。

例

`myprogram.c` からコンパイルされた実行可能プログラム testing が 32 ビット PowerPC アーキテクチャーのコンピューターで稼働することを指定するには、以下のように入力します。

```
xlc -o testing myprogram.c -q32 -qarch=ppc
```

関連情報
- 223 ページの『-qtune』
- 24 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』
- 234 ページの『コンパイラー・モードとプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ』
- パフォーマンス最適化のオプション: プロセッサーおよびアーキテクチャー最適化のオプション
- 「*XL C/C++ プログラミング・ガイド*」の『アプリケーションの最適化』

-qasm

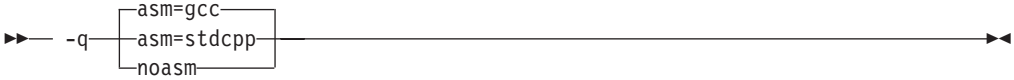
説明

asm アセンブリー・ステートメントのコードの解釈と以降の生成を制御する。

構文

C 言語レベルに関係なく、デフォルトは **-qasm=gcc** です。サブオプションなしで **-qasm** を指定した場合は、デフォルトを指定した場合と同等です。

C++ 言語レベルに関係なく、このデフォルトも **-qasm=gcc** です。



注

-qasm オプションとその否定フォームは、アセンブリー・ステートメントに対してコードが排出されるかどうかを制御します。オプションの肯定フォームは、ソース・コードでのアセンブリー・ステートメントのコードの生成をコンパイラーに命令します。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。例えば、**-qasm=gcc** を指定した場合、コンパイラーにアセンブリー・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命令します。

C トークン asm は C 言語のキーワードではありません。したがって、言語レベル **stdc89** と **stdc99**(それぞれ C89 および C99 標準への厳密な準拠を強制する) では、アセンブリー・コードを生成するソースをコンパイルするためにオプション **-qkeyword=asm** も指定されている必要があります。他のすべての言語レベルでは、オプション **-qnokeyword=asm** が有効になっていない限り、トークン asm はキーワードとして扱われます。C では、コンパイラー特定のバリアント __asm と __asm__ はすべての言語レベルでキーワードとなっており、使用不可にすることはできません。

C++ トークン asm、__asm、および __asm__ はすべての言語レベルでキーワードです。**-qnokeyword=***token* のサブオプションは、これらのそれぞれの予約語を使用不可にするために使用することができます。

事前定義マクロ

asm がキーワードとして扱われる場合、コンパイラーは指定されたアセンブリー言語構文に応じて、次の相互に排他的なマクロの 1 つを事前定義します。アセンブラー・コードが生成される場合、マクロは 1 の値を持ちます。そうでない場合は 0 になります。

__IBM_GCC_ASM

C++ __IBM_STDCPP_ASM

通知メッセージ

オプション **-qinfo=eff** も有効な場合は、アセンブリー・ステートメントに対してコードが生成されない場合、コンパイラーは通知メッセージを出します。

アセンブリー・ステートメントが有効な言語フィーチャーとして認識されたときには、オプション **-qinfo=por** は通知メッセージでそれを報告するようコンパイラーに命令します。

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。詳しくは、『-qasm_as』コンパイラー・オプションを参照してください。

例

以下のコードの断片は、インライン・ステートメントでの asm 構文に対する GCC 規則の例を示しています。

```
int a, b, c;
int main() {
    asm("add %0, %1, %2" : "=r"(a) : "r"(b), "r"(c) );
}
```

関連情報

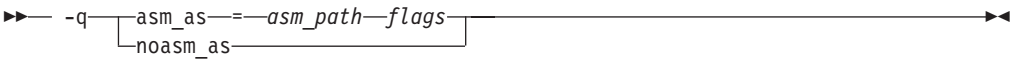
- 137 ページの『-qlanglvl』
- 114 ページの『-qinfo』
- 135 ページの『-qkeyword』
- 入力を制御するオプション: 言語拡張のオプション
- 「*XL C/C++ 言語解説書*」の『インライン・アセンブリー・ステートメント』
- 「*XL C/C++ 言語解説書*」の『言語拡張のキーワード』

-qasm_as

説明

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。

構文



ここで、

asm_path	アセンブリー・ステートメントのためにアセンブラーを呼び出すのに必要な、スペースで区切られたフラグのリスト
flags	使用されるアセンブラーの絶対パス名

コンパイラーはデフォルトで、コンパイラー構成ファイルから *asm_path* を読み取ります。

注

このオプションは、代替アセンブラー・プログラムとアセンブラーの呼び出しに必要なフラグを指定するために使用します。

このオプションは、コンパイラー構成ファイルに定義されているデフォルト設定の **as** コマンドをオーバーライドします。

例

`myprog.c` でインライン・アセンブラー・コードを検出したときに `/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、コマンド行に以下のように指定してください。

```
xlc myprog.c -qasm_as=/bin/as
```

関連情報

- 61 ページの『-qasm』
- コンパイラーのカスタマイズ
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qattr

説明

全 ID の属性リストを含むコンパイラー・リストを生成する。

構文

ここで、

-qnoattr	プログラムにある ID の属性リストを生成しません。
-qattr=full	プログラムにある ID をすべて報告します。
-qattr	使用されている ID のみを報告します。

278 ページの『#pragma options』も参照してください。

注

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。

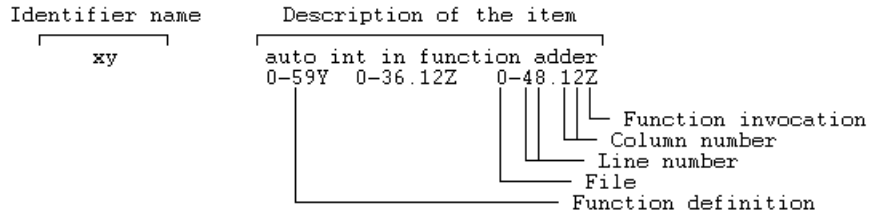
-qattr は、**-qattr=full** の後に指定すると、無効になります。完全なリストが生成されます。

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
```

一般的な相互参照リストの形式は、以下のとおりです。



関連情報
- 181 ページの『-qprint』
- 233 ページの『-qxref』
- リストとメッセージを制御するオプション: リストのオプション

-B

説明

コンパイラー、アセンブラー、リンケージ・エディター、およびプリプロセッサーなどのプログラムの代替パス名を決定する。

構文

ここで、*program* はコンパイラー・コンポーネント、または **-t** コンパイラー・オプションによって認識されるプログラム名です。

デフォルト

-B は指定されているが、*prefix* は指定されていない場合のデフォルト・プレフィックスは `/lib/o` です。 **-B***prefix* がまったく指定されていない場合、標準プログラム名のプレフィックスは `/lib/n` です。

-B は指定されているが、**-t***programs* が指定されていない場合は、デフォルトですべての標準プログラム名のパス名が構成されます。

注

prefix オプションは、新しいプログラムへのパス名の一部を定義します。コンパイラーは、プレフィックスとプログラム名の間に `/` を追加しません。

各プログラムごとに完全なパス名を形成するため、IBM XL C/C++ はコンパイラー、アセンブラー、エディター、およびプリプロセッサーの標準プログラム名にプレフィックスを追加します。

IBM XL C/C++ 実行可能ファイルの一部またはすべてについて複数のレベルを保持し、どれを使用するかを指定できるようにしたい場合、このオプションを使用します。

-B*prefix* が指定されていない場合は、デフォルトのパスが使用されます。

-B -t*programs* は、**-B** プレフィックス名を追加するプログラムを指定します。

-B*prefix* **-t***programs* オプションは、**-F***config_file* オプションをオーバーライドします。

例

代替 **xlc++** コンパイラーを使用して /lib/tmp/mine/ で 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 -B/lib/tmp/mine/ -tc
```

/lib/tmp/mine/ で代替エディターを使用して myprogram.C をコンパイルするには、以下のように入力します。

```
xlc++ myprogram.C -B/lib/tmp/mine/ -tl
```

関連情報

- 173 ページの『-qpath』
- 211 ページの『-t』
- 13 ページの『コンパイラーの呼び出し』
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qbigdata

説明

32 ビット・モードで、初期化されたデータのサイズが 16 MB を超えることを許可する。

構文

注

32 ビット・モードでは、初期化されたデータの GNU C/C++ サイズ制限は 16 MB です。共用ライブラリー内の初期化されたデータおよび呼び出しルーチン (open()、close()、printf() など) が 16 MB を超える 32 ビット・アプリケーションを作成するときには、このオプションを使用してください。

関連情報

- リンクを制御するオプション: リンカー入力制御のオプション

-qbitfields

説明

ビット・フィールドを符号付きにするかどうかを指定する。デフォルトでは、ビット・フィールドは符号付きです。

構文

ここで、オプションは以下の通りです。

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。

関連情報
- コマンド行オプションの要約: 符号のオプション

-C

説明
プリプロセスされた出力にコメントを保持する。

構文


注
-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、コメント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、コメントが出力ファイルに書き込まれます。

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
- 86 ページの『-E』
- 171 ページの『-P』
- コマンド行オプションの要約: その他の入力オプション

-c

説明
ソース・ファイルをコンパイラー・コンポーネントだけに渡すようコンパイラーに命令する。コンパイル済みのソース・ファイルは、リンケージ・エディターに送信されません。コンパイラーは、*file_name*.c、*file_name*.i、*file_name*.C、*file_name*.cpp など、それぞれ有効なソース・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。

構文


注
-c オプションは、**-E**、 **-P**、または **-qsyntaxonly** オプションのいずれかが指定されている場合にオーバーライドされます。

-c オプションを **-o** オプションと一緒に使用すると、コンパイラーによって作成されるオブジェクト・ファイルの明示的な名前を提供することができます。

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、以下のように入力します。

```
xlc++ myprogram.C -c -o new.o
```

関連情報

- 86 ページの『-E』
- 170 ページの『-o』
- 171 ページの『-P』
- 211 ページの『-qsyntaxonly』
- 出力を制御するオプション: その他の出力オプション

-qc_stdinc

C

説明

C ヘッダーの標準検索ロケーションを変更する。

構文

注

C ヘッダーの標準検索パスは、この (**-qc_stdinc**) コンパイラー・オプションと **-qgcc_c_stdinc** コンパイラー・オプションの両方によって指定される検索パスを、その順序で結合することによって決定します。このオプションのデフォルトの検索パスは、コンパイラーのデフォルト構成ファイルにあります。

これらのコンパイラー・オプションの一方が指定されていないか、一方が空ストリングを指定している場合は、標準検索ロケーションは他方のオプションによって指定されたパスになります。検索パスが **-qc_stdinc** と **-qgcc_c_stdinc** コンパイラー・オプションのどちらによっても指定されていない場合、デフォルトのヘッダー・ファイル検索パスが使用されます。

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。複数のディレクトリーを検索パスに指定するには、このオプションを一度指定し、**:** (コロン) で複数の検索ディレクトリーを区切ってください。

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。

例

mypath/headers1 および mypath/headers2 を標準検索パスの一部として指定するには、以下のように入力します。

```
xlc myprogram.c -qc_stdinc=mypath/headers1:mypath/headers2
```

関連情報

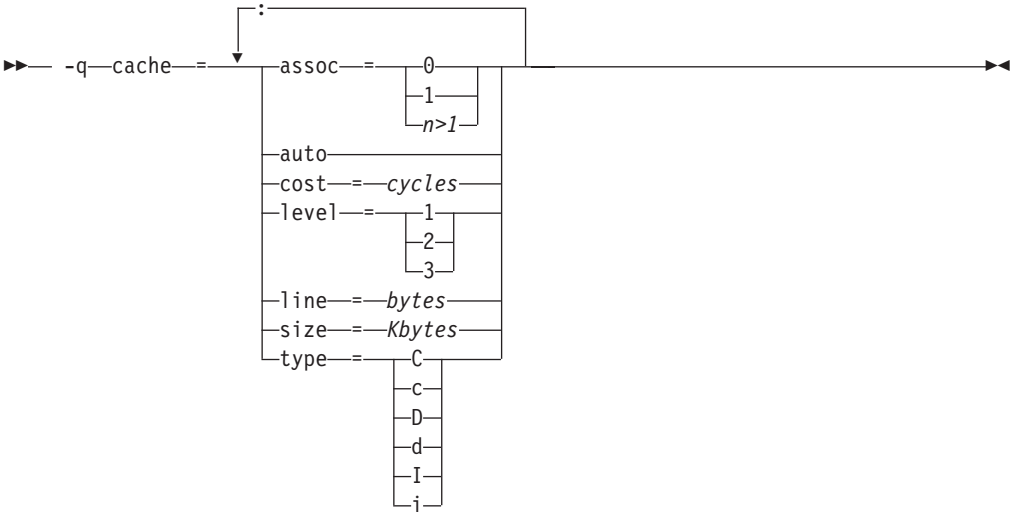
- 78 ページの『-qcpp_stdinc』
- 105 ページの『-qgcc_c_stdinc』
- 206 ページの『-qstdinc』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 22 ページの『構成ファイルでのコンパイラー・オプションの指定』
- 入力を制御するオプション: 検索パスのオプション

-qcache

説明

-qcache オプションは、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、キャッシュに関連する最適化の利点を計算します。

構文



ここで、使用可能なキャッシュ・オプションは以下の通りです。

assoc=*number*
: キャッシュの設定の結合順序を指定します。この場合、*number* 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、実行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=*cycles*
: 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。

level=*level*
: 影響されるキャッシュのレベルを指定します。この場合、*level* 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。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

  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。

line=*bytes*
: キャッシュの行サイズを指定します。

size=*Kbytes*
: キャッシュの合計サイズを指定します。

type=*cache_type*
: 指定されたタイプのキャッシュに設定を適用します。ここで、*cache_type* は、次のいずれかです。

  C または c
  : 結合されたデータおよび命令キャッシュ

  D または d
  : データ・キャッシュ

  I または i
  : 命令キャッシュ

注

-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。

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

- 166 ページの『-O、-qoptimize』
- 122 ページの『-qipa』
- パフォーマンス最適化のオプション: プロセッサーおよびアーキテクチャー最適化のオプション
- 「*XL C/C++ プログラミング・ガイド*」の『アプリケーションの最適化』

-qchars

説明

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ようコンパイラーに命令する。

構文

251 ページの『#pragma chars』および 278 ページの『#pragma options』も参照してください。

注

以下のプリプロセッサー・ディレクティブのいずれかを使用して、ソース・プログラムにおける符号のタイプを指定することもできます。

```
#pragma options chars=sign_type
```

```
#pragma chars (sign_type)
```

ここで、*sign_type* は signed または unsigned のいずれかです。

このオプションの設定に関係なく、char 型は、型の互換性検査 または C++ 多重定義のため、unsigned char 型および signed char 型とは異なると見なされます。

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
```

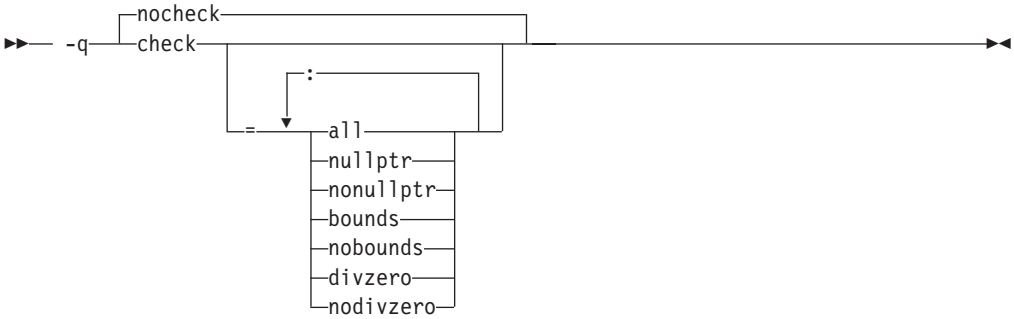
関連情報
- コマンド行オプションの要約: 符号のオプション

-qcheck

説明

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、ランタイム例外が出されます。

構文



ここで、

all
: 後続のサブオプションをすべてオンに切り替えます。**all** 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。

[no]nullptr
: 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。

[no]bounds
: 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。

[no]divzero
: 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。

278 ページの『#pragma options』も参照してください。

注

-qcheck オプションには前述のように幾つかのサブオプションがあります。複数のサブオプション を使用する場合は、コロン (:) でそれぞれを区切ってください。

サブオプションなし、および **-qcheck** のほかのバリエーションなしでコマンド行に **-qcheck** を指定すると、すべてのサブオプションがオンになります。

-qcheck オプションをサブオプションと共に使用すると、no プレフィックスがない場合は指定されたサブオプションがオンになり、no プレフィックスがある場合はオフになります。

-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。

-qcheck オプションは、アプリケーションのランタイム・パフォーマンスに影響を及ぼします。検査を有効にすると、ランタイム検査がアプリケーションに挿入されるため、実行が遅くなる場合があります。

例

1. **-qcheck=nullptr:bounds** の場合:

```
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 */
}
```

2. **-qcheck=divzero** の場合:

```
void func3(int a, int b) {
  a / b;            /* Traps if b=0  */
}
```

関連情報

- エラー検査とデバッグのオプション: エラー検査のオプション

-qcinc

C++

説明

extern "C" { } ラッパーを組み込みファイルのコンテンツのまわりに配置するようコンパイラーに命令する。

構文

ここで、

directory_prefix このオプションの影響を受けるファイルが検出されるディレクトリーを指定します。

注

指定されたディレクトリーの組み込みファイルは、組み込みファイルの最初のステートメントの前にトークン **extern "C" {** が挿入されて、組み込みファイルの最後のステートメントの後に **}** が付加されます。

例

アプリケーション `myprogram.C` にヘッダー・ファイル `foo.h` が組み込まれており、このファイルがディレクトリー `/usr/tmp` にあって、以下のコードが含まれていると想定します。

```
int foo();
```

以下のように入力してアプリケーションをコンパイルすると、

```
xlc++ myprogram.C -qcinc=/usr/tmp
```

以下のようにヘッダー・ファイル `foo.h` がアプリケーションに組み込まれます。

```
extern "C" {
int foo();
}
```

関連情報

- 入力を制御するオプション: 検索パスのオプション

-qcompact

説明

最適化と共に使用すると、可能な場合に、実行速度を犠牲にしてコード・サイズを削減する。

構文

278 ページの『#pragma options』も参照してください。

注

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。

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
```

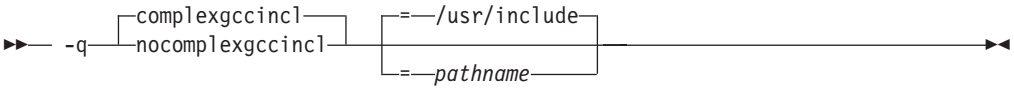
関連情報
- パフォーマンス最適化のオプション: コード・サイズ縮小のオプション

-qcomplexgccincl

説明

-qcomplexgccincl コンパイラー・オプションは、指定されたディレクトリーで検出される組み込みファイルのまわりの **#pragma complexgcc(on)** ディレクティブと **#pragma complexgcc(pop)** ディレクティブを内部でラップするようコンパイラーに命令する。

構文



ここで、

pathname 組み込みファイルの検索パスを指定します。 *pathname* が指定されていない場合、コンパイラーは `/usr/include` の *pathname* を想定します。

注

-qcomplexgccincl コンパイラー・オプションによって指定されたディレクトリー内で検出される組み込みファイルは、**#pragma complexgcc(on)** ディレクティブと **#pragma complexgcc(pop)** ディレクティブによって内部でラップされます。

-qnocomplexgccincl によって指定されたディレクトリーで検出された組み込みファイルは、これらのディレクティブによってラップされません。

デフォルト設定は **-qcomplexgccincl=/usr/include** です。

関連情報
- 253 ページの『#pragma complexgcc』
- 入力を制御するオプション: 検索パスのオプション

-qcpluscmt

C

説明

C++ のコメントを C ソース・ファイルで認識させるには、このオプションを使用します。

構文

デフォルト

デフォルト設定はさまざまです。

- **xlc** または **c99** および関連した **_r** 呼び出し を使用してコンパイラーを呼び出すと、**-qcpluscmt** が暗黙的に選択されます。
- **-qlanglvl** が **stdc99** または **extc99** に設定されている場合にも、**-qcpluscmt** が暗黙的に選択されます。これらの暗黙の選択は、コマンド行で **-qnocpluscmt** を **-qlanglvl** オプションの後に指定することによりオーバーライドできます。例えば次のようになります: **-qlanglvl=stdc99 -qnocpluscmt** または **-qlanglvl=extc99 -qnocpluscmt**。
- それ以外の場合、デフォルト設定は **-qnocpluscmt** になります。

注

__C99_CPLUSCMT コンパイラー・マクロは、**cpluscmt** が選択されたときに定義されます。

文字シーケンス // は、ヘッダー名、文字定数、ストリング・リテラル、またはコメント内を除き、 C++ のコメントを開始します。コメントはネストしません。また、マクロ置き換えは、コメント内では実行されません。以下の文字シーケンスは C++ コメント内では無視されます。

- //
- /*
- */

C++ のコメントの形式は、**//text** です。文字シーケンスの 2 つのスラッシュ (**//**) はその間に何も入らないで隣接している必要があります。スラッシュの右側から、改行文字で示された論理ソース行の終わりまでが、すべてコメントとして扱われます。 // 区切り文字は、行内の任意の位置に置くことができます。

// コメントは C89 の一部ではありません。 **-qcpluscmt** を指定すると、以下の有効な C89 プログラムの結果が誤りになります。

```
main() {
  int i = 2;
  printf("%i¥n", i //* 2 */
                 + 1);
}
```

正しい答えは 2 (2 / 1) です。 **-qcpluscmt** を指定すると、結果は 3 (2 + 1) になります。

プリプロセッサーは、以下の方法ですべてのコメントを処理します。

- **-C** オプションを指定しない 場合は、コメントがすべて除去され、単一のブランクで置換されます。
- **-C** オプションを指定した 場合は、コメントがプリプロセッサー・ディレクティブまたはマクロ引数に現れない限り、コメントが出力されます。
- **-E** を指定した場合は、継続シーケンスがすべてのコメントで認識されて出力されます。
- **-P** を指定した場合は、コメントが認識されて出力から除去され、連結された出力行が形成されます。

円記号 (¥) を使用して複数の物理ソース行が 1 つの論理ソース行に結合されている場合、コメントが複数の物理ソース行にわたることがあります。3 文字表記 (??/) によって円記号を表すことができます。

例

1. **C++ のコメントの例**

   以下の例は、C++ のコメントの使用を示したものです。

```
// A comment that spans two ¥
                  physical source lines

// A comment that spans two ??/
                  physical source lines
```

2. **プリプロセッサー出力の例 1**

   以下のソース・コード・フラグメントの場合:

```
int a;
int b;  // A comment that spans two ¥
                  physical source lines
int c;
        // This is a C++ comment
int d;
```

   -P オプションの出力は、以下の通りです。

```
int a;
int b;
int c;

int d;
```

   -P -C オプションの C89 モード出力は以下の通りです。

```
int a;
int b;  // A comment that spans two    physical source lines
int c;
        // This is a C++ comment
int d;
```

   -E オプションの出力は、以下の通りです。

```
int a;
int b;

int c;

int d;
```

   -E -C オプションの C89 モード出力は以下の通りです。

```
#line 1 "fred.c"
int a;
int b;  // a comment that spans two ¥
                  physical source lines
int c;
        // This is a C++ comment
int d;
```

   -P -C オプションまたは **-E -C** オプションの拡張モード出力は以下の通りです。

```
int a;
int b;  // A comment that spans two ¥
                physical source lines
int c;
        // This is a C++ comment
int d;
```

3. **プリプロセッサー出力の例 2 - ディレクティブ行**

   以下のソース・コード・フラグメントの場合:

   ```
   int a;
   #define mm 1   // This is a C++ comment on which spans two ¥
                     physical source lines
   int b;
           // This is a C++ comment
   int c;
   ```

   -P オプションの出力は、以下の通りです。

   ```
   int a;
   int b;

   int c;
   ```

   -P -C オプションの出力は以下の通りです。

   ```
   int a;
   int b;
           // This is a C++ comment
   int c;
   ```

   -E オプションの出力は、以下の通りです。

   ```
   #line 1 "fred.c"
   int a;
   #line 4
   int b;

   int c;
   ```

   -E -C オプションの出力は以下の通りです。

   ```
   #line 1 "fred.c"
   int a;
   #line 4
   int b;
           // This is a C++ comment
   int c;
   ```

4. **プリプロセッサー出力の例 3 - マクロ関数引数**

   以下のソース・コード・フラグメントの場合:

   ```
   #define mm(aa) aa
   int a;
   int b;  mm(// This is a C++ comment
                 int blah);
   int c;
           // This is a C++ comment
   int d;
   ```

   -P オプションの出力は、以下の通りです。

```
int a;
int b;  int blah;
int c;

int d;
```

-P -C オプションの出力は以下の通りです。

```
int a;
int b;  int blah;
int c;
        // This is a C++ comment
int d;
```

-E オプションの出力は、以下の通りです。

```
#line 1 "fred.c"
int a;
int b;
int blah;
int c;

int d;
```

-E -C オプションの出力は以下の通りです。

```
#line 1 "fred.c"
int a;
int b;
int blah;
int c;
        // This is a C++ comment
int d;
```

5. **コンパイル例**

   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、以下のように入力します。

   ```
   xlc myprogram.c -qcpluscmt
   ```

関連情報

- 66 ページの『-C』
- 86 ページの『-E』
- 137 ページの『-qlanglvl』
- 171 ページの『-P』
- 入力を制御するオプション: その他の入力オプション

-qcpp_stdinc

C++

説明

C++ ヘッダーの標準検索ロケーションを変更する。

構文

注

C++ ヘッダーの標準検索パスは、この (**-qcpp_stdinc**) コンパイラー・オプションと **-qgcc_cpp_stdinc** コンパイラー・オプションの両方によって指定される検索パスを、その順序で結合することによって決定します。このオプションのデフォルトの検索パスは、コンパイラーのデフォルト構成ファイルにあります。

これらのコンパイラー・オプションの一方が指定されていないか、一方が空ストリングを指定している場合は、標準検索ロケーションは他方のオプションによって指定されたパスになります。検索パスが **-qcpp_stdinc** と **-qgcc_cpp_stdinc** コンパイラー・オプションのどちらによっても指定されていない場合、デフォルトのヘッダー・ファイル検索パスが使用されます。

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。複数のディレクトリーを検索パスに指定するには、このオプションを一度指定し、**:** (コロン) で複数の検索ディレクトリーを区切ってください。

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。

例

`mypath/headers1` および `mypath/headers2` を標準の検索パスにするには、以下のように入力してください。

```
xlc++ myprogram.C -qcpp_stdinc=mypath/headers1:mypath/headers2
```

関連情報

- 67 ページの『-qc_stdinc』
- 106 ページの『-qgcc_cpp_stdinc』
- 206 ページの『-qstdinc』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 22 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』
- 入力を制御するオプション: 検索パスのオプション

-qcrt

説明

リンク時に標準システム始動を使用するようリンケージ・エディターに命令する。

構文

注

-qnocrt コンパイラー・オプションが指定されていると、コンパイラーはリンク時に標準始動ファイルを使用しません。

関連情報

- 152 ページの『-qlib』
- リンクを制御するオプション: その他のリンカー・オプション

-D

説明

マクロ *name* を #define プリプロセッサー・ディレクティブ内と同じに定義する。

構文

definition は、 *name* に割り当てるオプションの定義または値です。

注

-D コンパイラー・オプションによってマクロ名がまだ定義されていない場合、**#define** プリプロセッサー・ディレクティブを使用してソース・プログラムのマクロ名を定義することもできます。

-D*name*= は、 #define *name* と同等です。

-D*name* は、#define *name* 1 と同等です (これがデフォルトです)。

#define ディレクティブを使用して、すでに **-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、エラー状態の結果になります。

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。

-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。

例

1. 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、以下にように入力します。

   ```
   xlc myprogram.c -DCOUNT=100
   ```

   これは、ソース・ファイルの先頭に #define COUNT 100 を持つのと同じです。

関連情報

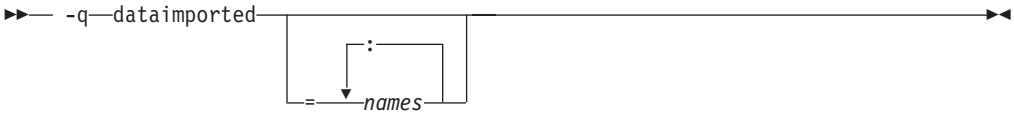
- 224 ページの『-U』
- 311 ページの『第 6 章 事前定義マクロ』
- コマンド行オプションの要約: その他の入力オプション

-qdataimported

説明

インポートとしてデータにマークを付ける。

構文



注

このオプションは 64 ビット・コンパイルにのみ適用されます。

このオプションが有効な場合、インポートされた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** を指定すると、すべての変数がインポートされたものであると想定するようコンパイラーに命令します。
- **-qdataimported=***names* を指定すると、名前付き変数がインポートされたものとしてマークを付けられます。ここで、*names* はコロン (**:**) で区切られた変数名のリストです。明示的に指定されていない変数は影響を受けません。

  注: C++ C++ プログラムでは、変数 *names* はそれらのマングル名を使用して指定する必要があります。例えば、以下のコード・セグメントを想定します。

  ```
  struct C{
          static int i;
  }
  ```

  変数 **C::i** をインポートされたものとして指定するには、以下の方法でコンパイラー・オプションを指定します。

  ```
  -qdataimported=i__1C
  ```

  オペレーティング・システムの **dump -tv** または **nm** ユーティリティーを使用して、オブジェクト・ファイルからマングル名を取得することができます。マングル名を検査するには、**c++filt** ユーティリティーを使用します。

-qdataimported および **-qdatalocal** データ・マーキング・オプションが矛盾する場合は、以下の方法で解決されます。

変数名のリスト・オプション:	特定の変数名に対する最後の明示的指定が使用されます。
デフォルトを変更するオプション:	この形式は、名前リストを指定しません。指定された最後のオプションが、名前リスト・フォームに明示的にリストされていない変数のデフォルトになります。

関連情報

- 82 ページの『-qdatalocal』
- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qdatalocal

説明

ローカルとしてデータにマークを付ける。

構文

注

このオプションは 64 ビット・コンパイルにのみ適用されます。

このオプションが有効な場合、ローカル変数はそれを使用する関数と静的にバインドされます。

- どの変数がローカルかを指定する必要があります。名前が指定されていないと、リンケージ・エディターはリンク時にリンクに失敗します。
- **-qdatalocal=***names* は、名前付き変数にローカルとしてマークを付けます。ここで、*names* は、コロン (**:**) によって区切られた ID のリストです。明示的に指定されていない変数は影響を受けません。

  注: C++ C++ プログラムでは、変数 *names* はそれらのマングル名を使用して指定する必要があります。例えば、以下のコード・セグメントを想定します。

  ```
  struct C{
          static int i;
  }
  ```

  変数 **C::i** をローカル・データとして指定するには、以下の方法でコンパイラー・オプションを指定します。

  ```
  -qdatalocal=i__1C
  ```

  オペレーティング・システムの **dump -tv** または **nm** ユーティリティーを使用して、オブジェクト・ファイルからマングル名を取得することができます。マングル名を検査するには、**c++filt** ユーティリティーを使用します。

インポートされた変数がローカルと想定された場合、パフォーマンスが低下する場合があります。

-qdataimported および **-qdatalocal** データ・マーキング・オプションが矛盾する場合は、以下の方法で解決されます。

変数名のリスト・オプション:	特定の変数名に対する最後の明示的指定が使用されます。
デフォルトを変更するオプション:	この形式は、名前リストを指定しません。指定された最後のオプションが、名前リスト・フォームに明示的にリストされていない変数のデフォルトになります。

関連情報

- 81 ページの『-qdataimported』
- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qdbxextra

C

説明

すべての typedef 宣言、struct、union、および enum 型定義がデバッグのために組み込まれることを指定する。

構文

注

-g オプションと一緒にこのオプションを使用すると、デバッガーで使用するための追加のデバッグ情報を生成します。

-g オプションを指定すると、デバッグ情報がオブジェクト・ファイルに組み込まれます。オブジェクトおよび実行可能ファイルのサイズを最小にするために、コンパイラーは、参照されるシンボルに関する情報しか組み込みません。デバッグ情報は、**-qdbxextra** を指定しない限り、参照されない配列、ポインター、またはファイル・スコープの変数に対しては生成されません。

-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。

例

デバッグのために myprogram.c 内のシンボルをすべて組み込むには、以下のように入力します。

```
xlc myprogram.c -g -qdbxextra
```

関連情報

- 103 ページの『-qfullpath』
- 153 ページの『-qlinedebug』
- 104 ページの『-g』
- 278 ページの『#pragma options』
- エラー検査とデバッグのオプション: デバッグのオプション

-qdigraph

説明

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を使用できるようにする。

構文

278 ページの『#pragma options』も参照してください。

デフォルト

- C **-qlanglvl** が**extc89**、**extended**、**extc99** または **stdc99** に設定されている場合、**-qdigraph**。
- C **-qlanglvl** が他のすべての言語レベルに設定されている場合、**-qnodigraph**。
- C++ **-qdigraph**

注

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。

連字キーの組み合わせは、以下のとおりです。

キーの組み合わせ	生成される文字
<%	{
%>	}
<:	[
:>	]
%%	#

C++ プログラムでのみ有効な追加のキーワードは、以下のとおりです。

キーワード	生成される文字
bitand	&
and	&&
bitor	|
or	||
xor	^
compl	~
and_eq	&=
or_eq	|=
xor_eq	^=
not	!
not_eq	!=

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

- 137 ページの『-qlanglvl』
- 222 ページの『-qtrigraph』
- 入力を制御するオプション: 言語拡張のオプション

-qdirectstorage

説明

ライトスルーが使用可能になっていること、あるいはキャッシュが抑制されたストレージが参照される可能性のあることをコンパイラーに通知する。

構文

注

-qdirectstorage コンパイラー・オプションは、コンパイラーに、ライトスルーが使用可能になっていたりキャッシュが禁止されているストレージが参照されている可能性があることと、適切なコンパイラー出力が生成されることを通知します。

PowerPC アーキテクチャーは、キャッシュ最適化の多くの異なるインプリメンテーションを許可します。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。

プログラムと実行される関数によって指定されたストレージ管理属性に応じて、プログラムは関数が正しく実行されることを保証するためにキャッシュ命令を使用する場合があります。

例えば、**dcbz** 命令はデータのブロックをキャッシュ内で割り振り、その後、それを一連のゼロに初期化します。これは大きなデータのブロックをゼロにしたときにパフォーマンスを改善するために使用できますが、**dcbz** 命令は以下のいずれかの条件において位置合わせエラーを引き起こすことがあるため、慎重に使用する必要があります。

- 命令によって指定されたキャッシュ・ブロックがキャッシュ禁止とマークの付けられたメモリー領域にある。
- キャッシュがライトスルー・モードである。
- L1 Dcache または L2 キャッシュが使用不可になっている。

-qdirectstorage を指定すると、**dcbz** 命令の生成が抑止され、前述の位置合わせエラーが回避されます。

関連情報

- パフォーマンス最適化のオプション: プロセッサーおよびアーキテクチャー最適化のオプション

-qdollar

説明

ID の名前に **$** シンボルを使用できるようにする。

構文

-qdollar 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。オプション **-qnodollar** と **-qlanglvl=ucs** の両方が有効な場合は、ドル記号は外字として扱われ、¥u0024 に変換されます。

例

$ をプログラム中の ID に使用できるように 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

- 278 ページの『#pragma options』
- 137 ページの『-qlanglvl』
- 入力を制御するオプション: 言語拡張のオプション

-qdump_class_hierarchy

C++

説明

それぞれのクラス・オブジェクトごとに、このオプションはその階層と仮想関数テーブルのレイアウトの表記をファイルにダンプする。ファイル名はソース・ファイル名に .class を付加して作成されます。例えば、myprogram.C を **-qdump_class_hierarchy** を使用してコンパイルすると、 myprogram.C.class という名前のファイルが作成されます。

構文

関連情報

- リストとメッセージを制御するオプション: リストのオプション

-E

説明

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、出力プリプロセス済みソース・ファイルを作成するようコンパイラーに命令する。

構文

注

-E および **-P** オプションの結果は異なります。**-E** オプションを指定すると、コンパイラーは、入力が C または C++ ファイルであり、出力が何らかの方法で再コンパイルまたは再処理されると見なします。これらの前提事項を以下に示します。

- 元のソースの位置が保持されます。これは、**#line** ディレクティブが生成されるためです。
- すべてのトークンは元のつづりで出力されます。この場合、継続シーケンスが含まれます。つまり、他のツールによる以後のコンパイルまたは再処理において、同じ位置 (例えば、エラー・メッセージの位置) が提供されます。

-P オプションは、汎用のプリプロセッシングに使用されます。入力または出力の使用目的に関する前提事項はありません。このモードは、C または C++ で作成されていない入力ファイルとともに使用します。このため、プリプロセス固有の構成は、ANSI C 標準の記述のとおりに処理されます。この場合は、継続シーケンスは、その標準の「変換フェーズ (Phases of Translation)」の記述通りに除去されます。プリプロセス固有のテキストでないテキストは、すべてそのまま出力されます。

-E を使用すると、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。ブランク行は除去されて、代わりに `#line` ディレクティブで置換されます。

-P オプションでは、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。**-E** オプションでは、トークンは、ソース位置を保持するために別個の行に出力されます。この場合は、継続シーケンスは除去されます。

-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て、任意のファイル名を受け入れます。

-M オプションと共に使用された場合、**-E** は .C、.cpp、.cc (すべて C++ ソース・ファイル)、.c (C ソース・ファイル)、または .i (プリプロセスされたソース・ファイル) ファイル名サフィックスを持つファイルに対してのみ機能します。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。

-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。コメントが複数のソース行にわたり、`-C` が指定されていない場合は、改行および **#line** ディレクティブが出されます。 マクロ関数引数にあるコメントは削除されます。

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
- 157 ページの『-M』
- 170 ページの『-o』
- 171 ページの『-P』
- 211 ページの『-qsyntaxonly』
- 出力を制御するオプション: ファイル出力のオプション

-e

説明

このオプションは、**-qmkshrobj** コンパイラー・オプションを指定した場合のみ使用される。詳しくは、165 ページの『-qmkshrobj』のコンパイラー・オプションの説明を参照してください。

構文

関連情報
- 165 ページの『-qmkshrobj』
- リンクを制御するオプション: リンカー入力制御のオプション

-qeh

C++

説明

例外処理がコンパイルされているモジュールで使用可能であるかどうかを制御する。

構文

ここで、

-qeh　例外処理が使用可能です。

-qnoeh　プログラムが C++ 構造化例外処理を使用しない場合は、 **-qnoeh** を指定してコンパイルし、アプリケーションに必要のないコードを生成しないようにしてください。

注

-qeh を指定すると、**-qrtti** も暗黙指定されます。 **-qeh** が **-qnortti** とともに指定されている場合、RTTI 情報は必要なものとして生成されます。

関連情報
- 192 ページの『-qrtti』
- パフォーマンス最適化のオプション: コード・サイズ縮小のオプション

-qenablevmx

説明

Vector Multimedia Extension (VMX) 命令の生成を使用可能にする。

構文

デフォルト

サポートされるすべての Linux ディストリビューションでは、 **-qenablevmx** がデフォルトで設定されています。

注

- 一部のプロセッサーは Vector Multimedia Extension (VMX) 命令をサポートすることができます。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、ハイパフォーマンスを提供することができます。
- **-qnoenablevmx** が有効な場合は、**-qaltivec**、および **-qhot=simd** は使用できません。

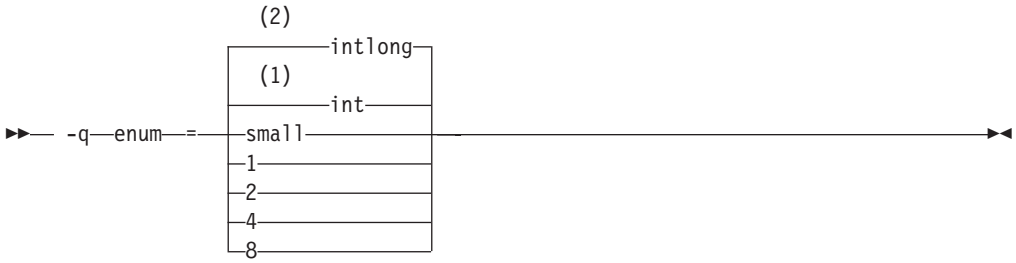
関連情報
- 57 ページの『-qaltivec』
- 58 ページの『-qarch』
- 109 ページの『-qhot』
- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-qenum

説明

列挙が占有するストレージの量を指定する。

構文



注:

1 C コンパイルのデフォルト

2 C++ コンパイルのデフォルト

ここで、有効な **enum** 設定は、以下のとおりです。

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、そうでない場合は 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、そうでない場合は unsigned short 型であると指定します。

4 列挙型がストレージの 4 バイトを占有し、列挙型の値の範囲が signed int の限度内の場合は int 型で、そうでない場合は unsigned int 型であると指定します。

8 列挙が 8 バイトのストレージを占有することを指定します。
32 ビットのコンパイル・モードでは、列挙型の値の範囲が signed long long の限度内の場合は long long 型、そうでない場合は unsigned long long 型となります。
64 ビットのコンパイル・モードでは、列挙型の値の範囲が signed long の場合は long 型、そうでない場合は unsigned long 型となります。

int 列挙が 4 バイトのストレージを占有して、int で表されることを指定します。C コンパイルでは、値は signed int の範囲を超えることはできません。

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。 89 ページの『-qenum』の説明を参照してください。
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。 89 ページの『-qenum』の説明を参照してください。

注

-qenum=small オプションは、enum 定数の範囲を表すことができる最小の事前定義の型に必要なストレージの量を enum 変数に割り振ります。デフォルトでは、符号なしの事前定義の型が使用されます。 enum 定数のいずれかが負の場合は、符号付きの事前定義の型が使用されます。

-qenum=1|2|4|8 オプションは、特定量のストレージを enum 変数に割り振ります。指定されたストレージ・サイズが enum 変数の範囲によって要求される量より少ない場合は、重大エラー・メッセージが発行されて、コンパイルが停止します。

ISO C 1989 および ISO C1999 標準では、列挙型の値が int の範囲を超えないことが要求されます。**-qlanglvl=stdc89** または **-qlanglvl=stdc99** を有効にしてコンパイルした場合、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。

- **-qenum=int** が有効な場合は、重大エラー・メッセージが発行されて、コンパイルが停止されます。
- **-qenum** の他のすべての設定の場合は、通知メッセージが発行されて、コンパイルは継続されます。

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。

関連情報

37 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』
256 ページの『#pragma enum』
278 ページの『#pragma options』

列挙型のサイズと型 - 特に明記されていない限り、すべての型は `signed` です。

	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
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[2b]	ERROR[1]	long[2b]	ERROR[1]	long[2b]	long long[2b]	long long[2b]	long[2b]	long[2b]
$-2^{63}..(2^{63}-1)$	ERROR[1]	long[2b]	ERROR[1]	long[2b]	ERROR[1]	long[2b]	long long[2b]	long long[2b]	long[2b]	long[2b]
$0..2^{64}$	ERROR[1]	unsigned long[2b]	ERROR[1]	unsigned long[2b]	ERROR[1]	unsigned long[2b]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]

注:

1. これらの列挙型は **-qenum=1|2|4** 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. C 列挙型は、C アプリケーションを ISO C 1989 および ISO C 1999 標準にコンパイルする場合、`int` の範囲を超えることはできません。
   -qlanglvl=stdc89 または **-qlanglvl=stdc99** を有効にしてコンパイルした場合、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。

列挙型のサイズと型 - 特に明記されていない限り、すべての型は `signed` です。										
	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[2a]	ERR[2a]	long long[2b]	long long[2b]	long[2b]	long[2b]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]
-2^{63}..(2^{63}-1)	ERR[2a]	ERR[2a]	long long[2b]	long long[2b]	long[2b]	long[2b]	long long[2b]	long long[2b]	long[2b]	long[2b]
0..2^{64}	ERR[2a]	ERR[2a]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]

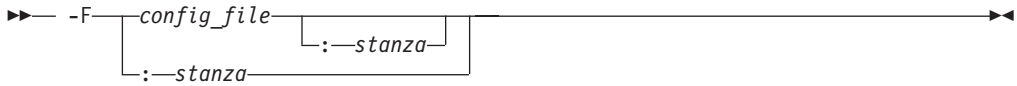
注:

1. これらの列挙型は **-qenum=1|2|4** 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 `enum` 設定を選択するか、 動的 `enum` 設定 (`small` または `intlong` など) を選択してください。
2. C 列挙型は、C アプリケーションを ISO C 1989 および ISO C 1999 標準にコンパイルする場合、`int` の範囲を超えることはできません。**-qlanglvl=stdc89** または **-qlanglvl=stdc99** を有効にしてコンパイルした場合、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。

関連情報
- 出力を制御するオプション: データ・サイズと位置合わせのオプション

-F

説明
コンパイラーの代替構成ファイル (.cfg) の名前を指定する。

構文


サブオプションは以下の通りです。

config_file コンパイラー構成ファイルの名前を指定します。
stanza コンパイラーの呼び出しに使用するコマンドの名前を指定します。これは、コンパイラーが、コンパイラー環境をセットアップするために、*config_file* 内の *stanza* の記入項目を使用するよう指示します。

注
デフォルトは、インストール時に構成される構成ファイルです。コマンド行またはソース・ファイル内に指定するファイル名またはスタンザは、構成ファイルに指定されたデフォルトをオーバーライドします。

-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。

例
`/usr/tmp/myvac.cfg` という構成ファイルを使用して `myprogram.c` をコンパイルするには、以下のように入力します。

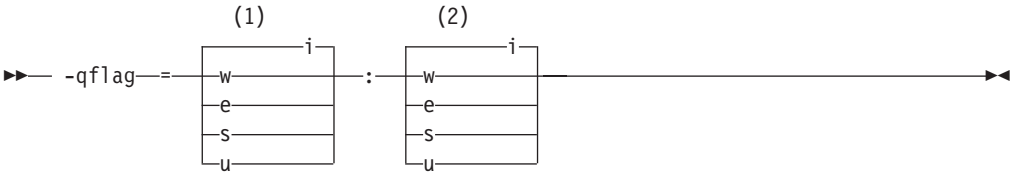
```
xlc myprogram.c -F/usr/tmp/myvac.cfg:xlc
```

関連情報
- 64 ページの『-B』
- 211 ページの『-t』
- 230 ページの『-W』
- 22 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』
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qflag

説明
リストで報告し、端末に表示する診断メッセージの最小の重大度レベルを指定する。診断メッセージは、関連するサブメッセージとともに表示されます。

構文



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

メッセージ重大度レベルは、以下の通りです。

重大度	**説明**
i	通知
w	警告
e	エラー
s	重大エラー
u	回復不能エラー

278 ページの『#pragma options』も参照してください。

注

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する必要があります。

通知メッセージ・レベルを指定しても、**-qinfo** オプションはオンになりません。

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、エラー以上のメッセージ (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

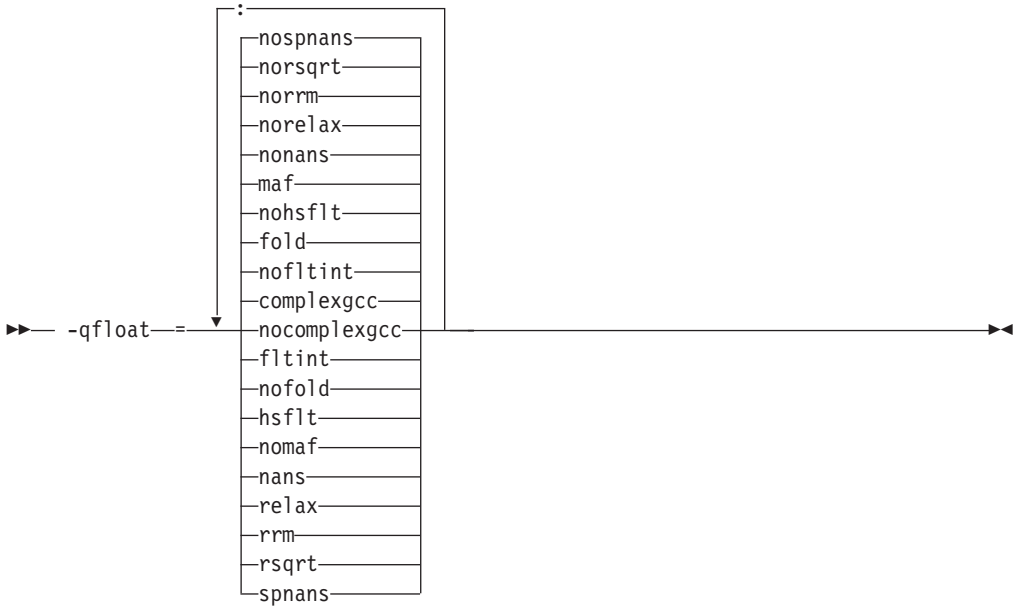
- 114 ページの『-qinfo』
- 231 ページの『-w』
- 30 ページの『コンパイラー・メッセージ』
- リストとメッセージを制御するオプション: メッセージのオプション

-qfloat

説明

さまざまな浮動小数点オプションを指定する。これらのオプションは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提供します。

構文



オプション選択については、以下の『**注**』セクションで説明します。 278 ページの『#pragma options』も参照してください。

注

デフォルト設定ではなく **float** サブオプションを使用すると、浮動小数点の計算でさまざまな結果が作成される可能性があります。指定されたサブオプションに必要な条件がすべて満たされていないと、誤った計算結果が作成される可能性があります。こういった理由により、IEEE 浮動小数点値に関連した浮動小数点の計算の経験があり、プログラムにエラーが混入する可能性を適切に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。

以下の 1 つ以上の **float** サブオプションを指定できます。

complexgcc nocomplexgcc	複合タイプの値を戻す場合とパラメーターを渡す場合に GCC との互換性を使用可能にします。デフォルトは、32 ビット・モードでコンパイルする場合が **float=complexgcc**で、64 ビット・モードでコンパイルする場合が **float=nocomplexgcc** です。

fltint
nofltint

オーバーフローを検査しない高速のインライン・コードを使用することによって、浮動小数点から整数への変換を高速化します。 デフォルトは **float=nofltint** であり、浮動小数点から整数への変換を検査して、範囲外の値がないかどうか調べます。

このサブオプションは、最適化オプションとともにのみ使用するようにしてください。

- **-O2** が有効な場合、**-qfloat=nofltint** が暗黙の設定です。
- **-O3** 以上が有効な場合、**-qfloat=fltint** が暗黙指定されます。

  -O3 オプションに小数点から整数への変換の範囲検査を組み込むには、**-qfloat=nofltint** を指定します。
- **-qnostrict** は、**-qfloat=fltint** を設定します。

fltint サブオプションが既に指定されている場合は、最適化レベルを変更しても、**fltint** の設定は変更されません。

-qstrict | **-qnostrict** オプションと **-qfloat=** オプションが矛盾する場合は、最後の設定が使用されます。

fold
nofold

浮動小数点定数式を、実行時ではなくコンパイル時に評価することを指定します。

hsflt
nohsflt

注: **hsflt** サブオプションは、浮動小数点計算の特性が既知である特定のアプリケーションのためのものです。他のアプリケーション・プログラムをコンパイルするときにこのオプションを使用すると、警告なしで誤った結果が作成される可能性があります。 **-qfloat=rndsngl**、**-q64**、または **-qarch=ppc**、あるいは他の PPC ファミリー・アーキテクチャー設定を指定してこのオプションを使用すると、rs64b またはそれ以降のシステムで間違った結果が生成される可能性があります。

hsflt オプションは、保管の前および浮動小数点から整数への変換時に、計算値を単精度に丸めるのではなく切り捨てることによって、計算を高速化します。 **nohsflt** サブオプションは、式の評価後に単精度式を丸め、範囲外の値かどうかについて浮動小数点から整数への変換を検査することを指定します。

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

-qfloat=hsflt オプションは、廃止された **-qhsflt** オプションに代わるものです。新しいアプリケーションでは、**-qfloat=hsflt** を使用してください。

このオプションは、**-qarch** オプションが **pwr**、**pwr2**、**pwrx**、 **pwr2s**、または **com** (32 ビット・モードの場合) に設定されていない場合にはほとんど効果がありません。PPC ファミリー・アーキテクチャーでは、すべての単精度 (float) 演算が丸められます。このオプションの影響を受けるのは、単精度 (float) への倍精度 (double) 式のキャストだけです。

maf
nomaf

適切な箇所で浮動小数点乗算・加算命令を使用することによって、より高速でより正確に浮動小数点計算を行います。 この結果は、コンパイル時に行われる類似の計算の結果または他のタイプのコンピューターでの結果と正確に同じにならない場合があります。 負のゼロの結果が作成される可能性があります。このオプションは、浮動小数点の中間結果の精度に影響を及ぼす場合があります。

nans
nonans

実行時に単精度から倍精度に変換しているときにシグナル NaN (Not-a-Number) を検出するよう追加の命令を生成します。オプション **nonans** は、この変換を検出する必要がないことを指定します。 **-qfloat=nans** は、IEEE 754 標準に完全に準拠するために必要です。

-qflttrap または **-qflttrap=invalid** オプションとともに使用すると、コンパイラーはオペランドのいずれかがシグナル NaN である場合に引き起こされる比較演算における無効な演算例外を検出します。

relax
norelax

厳密な IEEE への規格合致を、速度を速めるために若干緩めます。これは一般に、右方のゼロに関連した加算や減算など、いくつかのささいな浮動小数点の算術演算を除去することによって行います。

rrm
norrm

ランタイムの正および負の無限大方向への丸めモードと非互換の浮動小数点の最適化を回避する。浮動小数点の丸めモードが実行時に変更される可能性があること、または浮動小数点の丸めモードが実行時に*最近似値*へのモードでないことをコンパイラーに通知します。

浮動小数点状況レジスターおよび制御レジスターを実行時に変更する場合は、 **-qfloat=rrm** を指定しなければなりません (例外トラッピングの初期化についても同様です)。

rsqrt
norsqrt

平方根の逆数を計算して乗算することによって、平方根の結果による除算を含むコードのシーケンスを置換することができるかどうかを指定します。 この置換を可能にすると、コードの実行が高速化されます。

- **-O2** の場合、デフォルトは **-qfloat=norsqrt** です。
- **-O3** の場合、デフォルトは **-qfloat=rsqrt** です。このデフォルトをオーバーライドするには、**-qfloat=norsqrt** を使用します。
- **-qnostrict** は、**-qfloat=rsqrt** を設定します。 (**-qfloat=rsqrt** は、`errno` が `sqrt` 関数呼び出しに設定*されない* ことを意味するので注意してください。)
- **-qignerrno** も指定しない限り、**-qfloat=rsqrt** は無効です。

rsqrt がすでに指定されている場合、最適化レベルを変更しても **rsqrt** オプションの設定は変更されません。 **-qstrict** | **-qnostrict** オプションと **-qfloat=** オプション が矛盾する場合は、最後の設定が使用されます。

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。オプション **nospnans** は、この変換を検出する必要がないことを指定します。

例

コンパイル時に定数浮動小数点式が評価され、乗算・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、以下のように入力します。

```
xlc++ myprogram.C -qfloat=fold:nomaf
```

関連情報

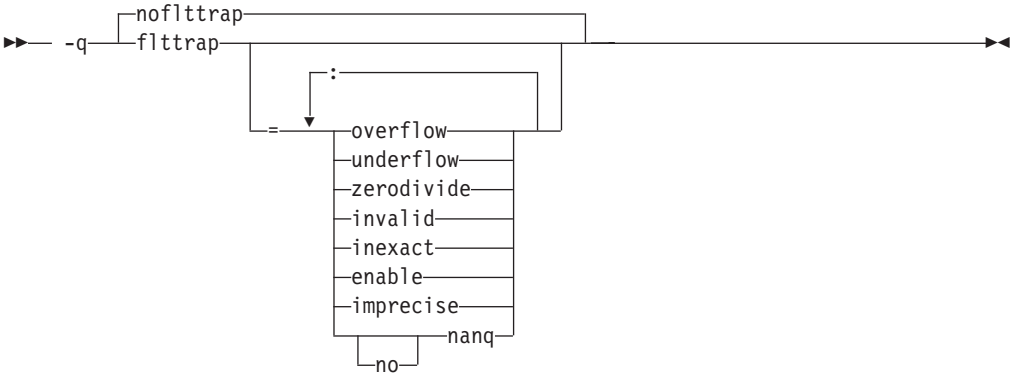
- 58 ページの『-qarch』
- 74 ページの『-qcomplexgccincl』
- 『-qflttrap』
- 207 ページの『-qstrict』
- 253 ページの『#pragma complexgcc』
- 整数および浮動小数点処理を制御するオプション

-qflttrap

説明

実行時の浮動小数点の例外を検出してトラップするために追加の命令を生成する。

構文



サブオプションは以下のことを行います。

enable	メインプログラムのプロローグに指定された例外を使用可能にします。 **nanq** (下記) は例外ですが、ソース・コードを変更せずに以下にリストされた例外トラッピングをオンにする場合は、このサブオプションが必要です。
overflow	浮動小数点のオーバーフローを検出およびトラップするためのコードを生成します。
underflow	浮動小数点のアンダーフローを検出およびトラップするためのコードを生成します。
zerodivide	ゼロによる浮動小数点除算を検出およびトラップするためのコードを生成します。
invalid	浮動小数点無効演算例外を検出およびトラップするためのコードを生成します。
inexact	浮動小数点精度低下例外を検出およびトラップするためのコードを生成します。
imprecise	指定された例外の不正確な検出のためのコードを生成します。 例外が引き起こされた場合は、例外が検出されますが、例外の正確な位置は判別されません。

nanq　浮動小数点演算子によって処理または生成される NaNQ (Not a Number Quiet) 例外を検出してトラップするためのコードを生成します。**nanq** および **nonanq** 設定は、**-qnoflttrap**、**-qflttrap**、または **-qflttrap=enable** による影響を受けません。

注

このオプションは、リンク時に認識されます。 **-qnoflttrap** は、これらの追加の命令を生成する必要がないことを指定します。

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。例外が自動的に使用可能にされることはなく、全浮動小数点演算が検査されて、正確な例外の位置に関する情報が提供されます。

#pragma options とともに指定する場合、 **-qnoflttrap** オプションは、指定する最初のオプションでなければなりません。

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** を使用してください。

-qflttrap オプションが **-qoptimize** 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。

例

浮動小数点のオーバーフローとアンダーフロー、およびゼロ除算を検出するように `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 -qflttrap=overflow:underflow:zerodivide:enable
```

関連情報

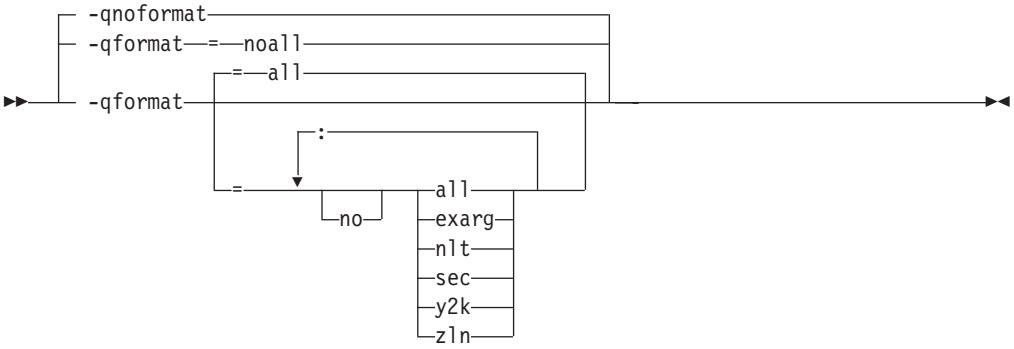
- 96 ページの『-qfloat』
- 278 ページの『#pragma options』
- 整数および浮動小数点処理を制御するオプション

-qformat

説明

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



サブオプションは以下の通りです。

all　すべてのフォーマット診断メッセージをオンにします。

exarg　`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　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　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　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　ゼロの長さのフォーマットについて警告を出します。

注: 前述のいずれかのサブオプションの前に **no** を指定すると、その診断メッセージのグループが使用不可になります。例えば、y2k 警告の診断メッセージをオフにするには、コマンド行に **-qformat=noy2k** と指定します。

注

-qformat がコマンド行に指定されていない 場合は、コンパイラーはデフォルト設定 **-qnoformat** を想定します。これは **-qformat=noall** と同等です。

-qformat がサブオプションなしでコマンド行に指定されていると、コンパイラーはデフォルト設定 **-qformat=all** を想定します。

例

1. 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
   ```

2. 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
   ```

関連情報

- リストとメッセージを制御するオプション: メッセージのオプション

-qfullpath

説明

-g コンパイラー・オプションを使用するときに、どのようなパス情報をファイル用に保管するかを指定する。

構文

注

-qfullpath を使用すると、コンパイラーは、 **-g** オプションで指定したソース・ファイルの絶対 (フル) パス名を保持します。

ファイルの相対パス名は、**-qnofullpath** を使用すると保持されます。

-qfullpath は、実行可能ファイルを他のディレクトリーに移動した場合に便利です。 **-qnofullpath** を指定すると、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。 **-qfullpath** を使用すると、ファイルを正常に見つけることができます。

関連情報
- 153 ページの『-qlinedebug』
- 104 ページの『-g』
- エラー検査とデバッグのオプション: デバッグのオプション

-qfuncsect

説明

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。デフォルトでは、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。

構文

注

複数の csect を使用すると、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ている関数を除去することをエディタ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。

このオプションが **#pragma options** に指定されている場合、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されている必要があります。

関連情報

- 278 ページの『#pragma options』
- 出力を制御するオプション: ファイル出力のオプション

-g

説明

GNU GDB Debugger などのデバッグ・ツールに使用される情報を生成する。

構文

注

-g を指定すると、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。例:

オプション	インライン化に対する効果
-g	インライン化なし
-O	宣言された関数をインライン化します。
-O -Q	宣言された関数をインライン化し、その他を自動インライン化します。
-g -O	宣言された関数をインライン化します。
-g -O -Q	宣言された関数をインライン化し、その他を自動インライン化します。

-g のデフォルトでは、参照されていないシンボルに関する情報はデバッグ情報に含まれません。

参照されているシンボルと参照されていないシンボルの両方に関する情報を組み込むには、**-qdbxextra** オプションを **-g** と共に使用します。

-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** 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。

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
```

`myprogram.c` をコンパイルして、参照されていないシンボルに関する追加情報を含む `testing_all` という名前の実行可能プログラムを作成し、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_all -g -qdbxextra
```

関連情報

- 83 ページの『-qdbxextra』
- 103 ページの『-qfullpath』
- 153 ページの『-qlinedebug』

- 166 ページの『-O、-qoptimize』
- 185 ページの『-Q』
- エラー検査とデバッグのオプション: デバッグのオプション

-qgcc_c_stdinc

C

説明

GCC ヘッダーの標準検索ロケーションを変更する。

構文

注

GCC ヘッダーの標準検索パスは、**-qc_stdinc** コンパイラー・オプションとこの (**-qgcc_c_stdinc**) コンパイラー・オプションの両方によって指定された検索パスを、その順序で結合することによって決定されます。このオプションのデフォルトの検索パスは、コンパイラーのデフォルト構成ファイルにあります。

これらのコンパイラー・オプションの一方が指定されていないか、一方が空ストリングを指定している場合は、標準検索ロケーションは他方のオプションによって指定されたパスになります。検索パスが **-qc_stdinc** と **-qgcc_c_stdinc** コンパイラー・オプションのどちらによっても指定されていない場合、デフォルトのヘッダー・ファイル検索パスが使用されます。

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。複数のディレクトリーを検索パスに指定するには、このオプションを一度指定し、**:** (コロン) で複数の検索ディレクトリーを区切ってください。

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。

例

`mypath/headers1` および `mypath/headers2` を標準検索パスの一部として指定するには、以下のように入力します。

```
xlc myprogram.c -qgcc_c_stdinc=mypath/headers1:mypath/headers2
```

関連情報

- 67 ページの『-qc_stdinc』
- 78 ページの『-qcpp_stdinc』
- 106 ページの『-qgcc_cpp_stdinc』
- 206 ページの『-qstdinc』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 22 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』
- 入力を制御するオプション: 検索パスのオプション

-qgcc_cpp_stdinc

C++

説明

g++ ヘッダーの標準検索ロケーションを変更する。

構文

注

g++ ヘッダーの標準検索パスは、**-qcpp_stdinc** コンパイラー・オプションとこの (**-qgcc_cpp_stdinc**) コンパイラー・オプションの両方によって指定される検索パスを、その順序で結合することによって判別します。このオプションのデフォルトの検索パスは、コンパイラーのデフォルト構成ファイルにあります。

これらのコンパイラー・オプションの一方が指定されていないか、一方が空ストリングを指定している場合は、標準検索ロケーションは他方のオプションによって指定されたパスになります。検索パスが **-qcpp_stdinc** と **-qgcc_cpp_stdinc** コンパイラー・オプションのどちらによっても指定されていない場合、デフォルトのヘッダー・ファイル検索パスが使用されます。

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。複数のディレクトリーを検索パスに指定するには、このオプションを一度指定し、**:** (コロン) で複数の検索ディレクトリーを区切ってください。

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。

例

`mypath/headers1` および `mypath/headers2` を標準検索パスの一部として指定するには、以下のように入力します。

```
xlc++ myprogram.C -qgcc_cpp_stdinc=mypath/headers1:mypath/headers2
```

関連情報

- 67 ページの『-qc_stdinc』
- 78 ページの『-qcpp_stdinc』
- 105 ページの『-qgcc_c_stdinc』
- 206 ページの『-qstdinc』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 22 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』
- 入力を制御するオプション: 検索パスのオプション

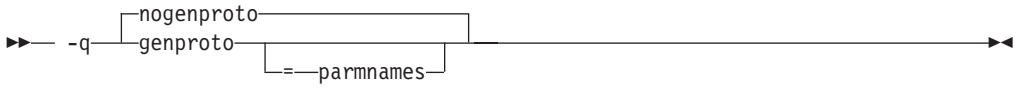
-qgenproto

C

説明

K&R 関数定義から ANSI プロトタイプを生成する。これにより、K&R から ANSI への遷移が容易になります。

構文



注

parmnames なしで **-qgenproto** を使用すると、パラメーター名なしでプロトタイプが生成されます。パラメーターは、**parmnames** が指定されたときにプロトタイプに組み込まれます。

例

以下の関数 foo.c について、

```
foo(a,b,c)
  float a;
  int *b;
  int c;
```

以下のように指定すると、

```
xlc -c -qgenproto foo.c
```

以下が生成されます。

```
int foo(double, int*, int);
```

パラメーター名は除去されます。一方、以下にように指定すると、

```
xlc -c -qgenproto=parmnames foo.c
```

以下が生成されます。

```
int foo(double a, int* b, int c);
```

この場合、パラメーター名は保持されます。

K&R 関数に渡される前にすべての狭い型の引数 (**char**、short、**float** など) が拡張されることが ANSI に規定されているため、`float a` は、プロトタイプでは `double` または `double a` として表されるので注意してください。

関連情報

- エラー検査とデバッグのオプション: その他のエラー検査とデバッグのオプション

-qhalt

説明

指定した*重大度* 以上のエラーが検出された場合に、コンパイル・フェーズ後に停止するようにコンパイラーに命令する。

構文

重大度レベルは以下の通りです (重大度レベルは上から下の順序で上がります)。

重大度	**説明**
i	通知
w	警告
e	エラー (C のみ)
s	重大エラー
u	回復不能エラー

278 ページの『#pragma options』も参照してください。

注

-qhalt 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ゼロ以外です。

-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。

-qhalt オプションは、 **-qmaxerr** オプションでオーバーライドすることができます。

診断メッセージは、**-qflag** オプションで制御することができます。

例

`myprogram.c` をコンパイルし、警告レベル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を停止するには、以下のように入力します。

```
xlc myprogram.c -qhalt=w
```

関連情報

- 95 ページの『-qflag』
- 162 ページの『-qmaxerr』
- リストとメッセージを制御するオプション: メッセージのオプション

-qhaltonmsg

C++

説明

指定した *msg_number* が検出された場合、コンパイル・フェーズ後に停止するようコンパイラーに命令する。

構文

注

-qhaltonmsg オプションの結果としてコンパイラーが停止した場合、コンパイラーの戻りコードはゼロ以外です。

-qhaltonmsg オプションを使用すると、複数のメッセージ番号を指定することができます。追加のメッセージ番号はコンマで分離する必要があります。

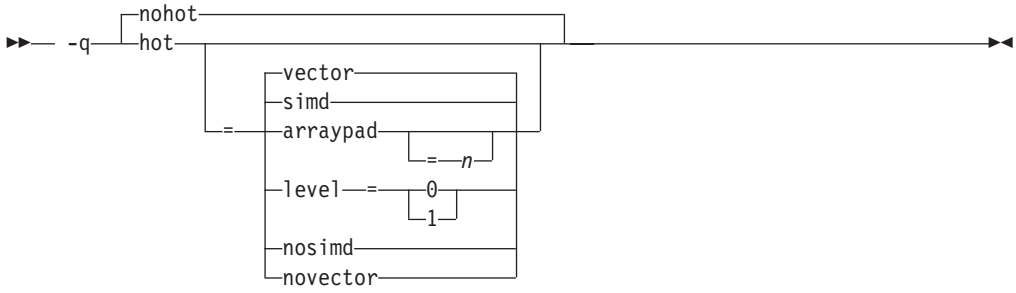
関連情報
- 30 ページの『コンパイラー・メッセージ』
- リストとメッセージを制御するオプション: メッセージのオプション

-qhot

説明

最適化中に高位ループ分析と変換を行うようコンパイラーに命令する。

構文



ここで、

arraypad　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により、 2 の累乗の配列次元はキャッシュの使用効率の低下を招く可能性があります。**arraypad** サブオプションにより、コンパイラーは配列の大きさを増加することができます。そうすることにより配列処理ループの効率が改善される可能性があります。 **arraypad** サブオプションを数値以外の値に使用すると、コンパイラーは利益があると推測する任意の配列に選択した量だけ埋め込みを行います。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。

arraypad=*n*　コンパイラーはコード内のすべての配列で埋め込みを行います。埋め込みの量は正整数の値でなければならず、それぞれの配列は整数のエレメントによって埋め込まれます。 *n* は整数値なので、埋め込み値は配列エレメントの最大サイズの倍数 (通常、4、8、または 16) にすることをお勧めします。

level=0　コンパイラーは高位変換のサブセットを実行します。

-qhot=level=0 を指定すると、デフォルトは **novector**、**nosimd** および **noarraypad** に設定されます。

-qhot=level=0 を **-O4** の前に指定すると、**level** は 1 に設定されます。
-qhot=level=0 を **-O4** の後に指定すると、 **level** は 0 に設定されます。

level=1　**-qhot=level=1** は **-qhot** と同等で、**-qhot** を暗黙指定するコンパイラー・オプションも **-qhot=level=0** が明示的に指定されている場合を除き、**-qhot=level=1** を暗黙指定します。

level 以外のすべての **-qhot** サブオプションのデフォルトのホット・レベルは 1 です。例えば、**-O3 -qhot=novector** を指定した場合、ホット・レベルは 1 に設定されます。

-O4 または **-qsmp** を指定すると、**-qhot=level=0** オプションが明示的に指定されている場合を除き、 **-qhot=level=1** が暗黙指定されます。

simd | nosimd　コンパイラーはループで配列の連続するエレメントに対して実行される特定の演算を、Vector Multimedia Extension (VMX) 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。

並列操作は 16 バイトのベクトル・レジスターで発生します。コンパイラーはレジスター長を超えるベクトルを 16 バイトの単位に分割して、最適化を可能にします。 16 バイトの単位には、次のいずれかのタイプのデータを入れることができます。

- 4 個の整数。
- 8 個の 2 バイト単位。
- 16 個の 1 バイト単位。

-qhot=simd 最適化を適用することは、大き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、VMX 命令をサポートするアーキテクチャー を指定した場合にのみ有効です。これらの条件では、**-qhot=simd** がデフォルトとして設定されます。

-qhot=nosimd を指定すると、コンパイラーはループと配列に対して最適化を実行しますが、特定のコードを VMX 命令への呼び出しに置換することは行いません。

vector | novector　**-qnostrict** および **-qignerrno**、または最適化レベル **-O3** 以上を使用して指定された場合、このベクトル・サブオプションによりコンパイラーは、ループ内で連続する配列のエレメント (例えば、平方根、逆平方根) に対して実行される特定の演算子を MASS ライブラリー・ルーチンへの呼び出しに変換させます。この呼び出しは同時に幾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高速です。コンパイラーはベクトル・サイズの制限のない標準レジスターを使用します。**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。

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。

デフォルト

デフォルトは **-qnohot** です。

-qhot をサブオプションなしで指定すると、 **-qhot=nosimd**、**-qhot=noarraypad**、**-qhot=vector**、および **-qhot=level=1** が暗黙指定されます。 **-qhot** オプションは、**-qsmp**、**-O4**、および **-O5** によっても暗黙指定されます。

注

コマンド行に **-qhot** を指定するときに少なくともレベル 2 の最適化も指定しないと、コンパイラーは **-O2** を想定します。

-qhot=arraypad と **-qhot=arraypad=***n* は両方とも危険なオプションです。これらは、埋め込みが行われた場合にコードの中断の原因となる可能性のある再形成や等価の検査を実行しません。

例

以下の例は、**-qhot=arraypad** オプションをオンにします。

```
xlc -qhot=arraypad myprogram.c
```

関連情報

- 58 ページの『-qarch』
- 66 ページの『-C』
- 166 ページの『-O、-qoptimize』
- 207 ページの『-qstrict』
- 198 ページの『-qsmp』
- パフォーマンス最適化のオプション: ループ最適化のオプション
- 「*XL C/C++ プログラミング・ガイド*」の『Mathematical Acceleration Subsystem (MASS) の使用法』

-I

説明

絶対パスを指定しない組み込みファイル名の追加の検索パスを指定する。

構文

注

directory の値は、有効なパス名 (例えば、`/u/golnaz`、`/tmp`、または `./subdir`) でなければなりません。コンパイラーはスラッシュ (*/*) をディレクトリーに付加し、検索を行う前にそれをファイル名に連結します。パス・ディレクトリーは、名前がスラッシュ (*/*) で始まっていない組み込みファイルについて、最初にコンパイラーが検索するディレクトリーです。ディレクトリーを指定しない場合は、デフォルトで、標準のディレクトリーが検索されます。

構成ファイルとコマンド行の両方に **-I** directory オプションが指定されている場合は、構成ファイにで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。

#include ディレクティブにフル (絶対) パス名を指定した場合、このオプションの影響はありません。

例

myprogram.C をコンパイルして、組み込みファイルを /usr/tmp で検索し、次に /oldstuff/history で検索するには、以下のように入力します。

```
xlc++ myprogram.C -I/usr/tmp -I/oldstuff/history
```

関連情報

- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 入力を制御するオプション: 検索パスのオプション

-qidirfirst

説明

#include "*file_name***"** ディレクティブで組み込むファイルの検索順序を指定する。

構文

278 ページの『#pragma options』も参照してください。

注

-qidirfirst は、**-I** オプションと一緒に使用してください。

idirfirst オプションが*指定されていない* 場合、(#include "*file_name*" ディレクティブによって組み込まれたファイルの) 標準検索順序は以下の通りです。

1. 現行のソース・ファイルがあるディレクトリーを検索します。
2. **-I** オプションを使用して指定されたディレクトリーを検索します。
3. 標準の組み込みディレクトリーを検索します。

-qidirfirst が指定されている場合は、現行ファイルが常駐するディレクトリーの前に、**-I** オプションで指定されたディレクトリーが検索されます。

-qidirfirst は、#include <*file_name*> ディレクティブの検索順序には影響ありません。

-qidirfirst は、#include "*file_name*" と #include <*file_name*> の両方の検索順序を変更する **-qnostdinc** オプションとは無関係です。

ファイルの検索順序については、 26 ページの『相対パス名を使用した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。

最後に有効な **#pragma options [no]idirfirst** は、後続の **#pragma options [no]idirfirst** によって置換されるまで有効なままとなります。

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前に `/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

- 111 ページの『-I』
- 206 ページの『-qstdinc』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 入力を制御するオプション: 検索パスのオプション

-qignerrno

説明

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。

構文

278 ページの『#pragma options』も参照してください。

注

例外が発生すると、幾つかのシステム・ライブラリー・ルーチンは `errno` を設定します。この設定以降の `errno` の副次作用は、**-qignerrno** を指定することによって無視することができます。

-O3 以上の最適化オプションを指定すると、**-qignerrno** も指定されます。両方の最適化と `errno` を設定する能力が必要な場合は、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。

関連情報

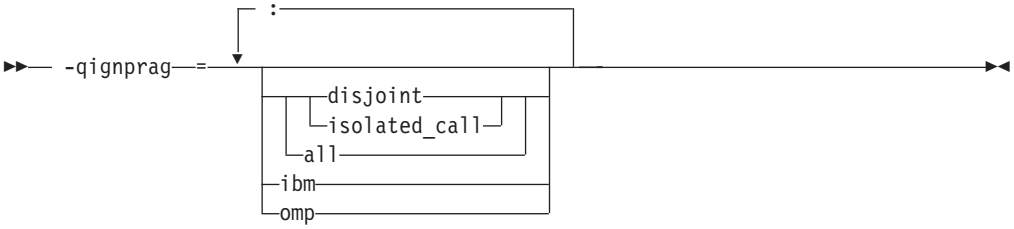
- 166 ページの『-O、-qoptimize』
- パフォーマンス最適化のオプション: 副次作用のオプション

-qignprag

説明

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。

構文



このオプションの影響を受けるプラグマ・ステートメントは、以下のとおりです。

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。
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。
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。
ibm	C ソース・ファイルですべての **#pragma ibm snapshot** 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。

278 ページの『#pragma options』も参照してください。

注

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 **-qignprag** を使用すると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。

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
```

関連情報

- 254 ページの『#pragma disjoint』
- 268 ページの『#pragma isolated_call』
- 263 ページの『#pragma ibm snapshot』
- 245 ページの『OpenMP プラグマ・ディレクティブの要約』
- 入力を制御するオプション: その他の入力オプション

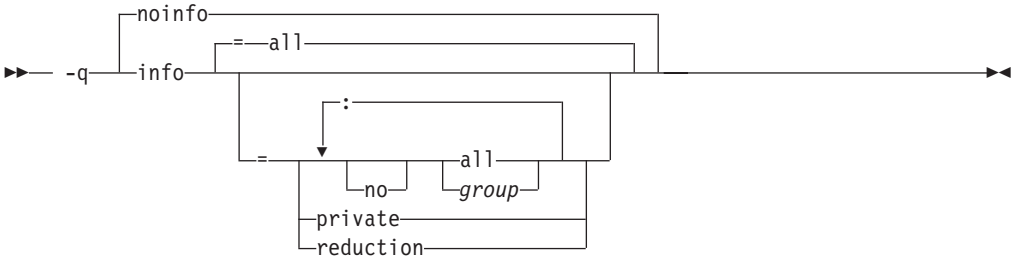
-qinfo

説明

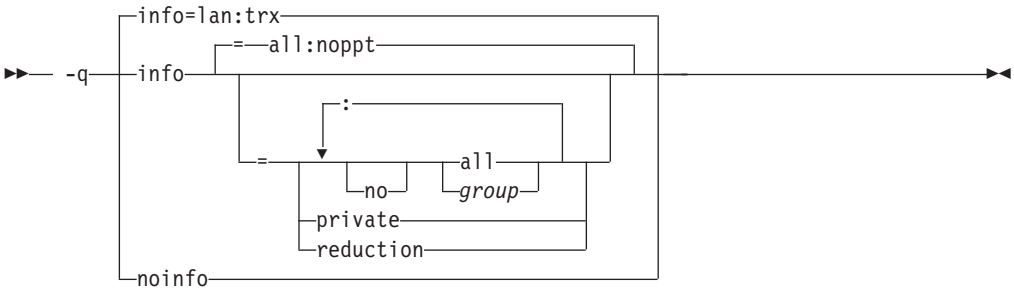
通知メッセージを作成する。

構文

C



C++



-qinfo オプションおよび診断メッセージ・グループについては、以下の『**注**』セクションで説明します。

264 ページの『#pragma info』および 278 ページの『#pragma options』も参照してください。

デフォルト

コマンド行に **-qinfo** を指定しないと、コンパイラーは以下を想定します。

1. C **-qnoinfo**
2. C++ **-qinfo=lan:trx**

コマンド行にサブオプションなしで **-qinfo** を指定すると、コンパイラーは以下を想定します。

1. C **-qinfo=all**
2. C++ **-qinfo=all:noppt**

注

サブオプションなしで **-qinfo=all** または **-qinfo** を指定すると、C++ コードの **ppt** (プリプロセッサー・トレース) グループを除き、すべてのグループに対するすべての診断メッセージがオンになります。

-qnoinfo または **-qinfo=noall** を指定すると、すべてのグループのすべての診断メッセージがオフになります。

このコンパイラー・オプションの **#pragma options info=***suboption*[**:***suboption* ...] または **#pragma options noinfo** 形式を使用して、プログラム・コードの 1 つ以上の特定のセクションで一時的にメッセージを使用可能または使用不可にすることができます。

-qinfo オプションの有効な形式は、以下のとおりです。

all
: すべてのグループのすべての診断メッセージをオンにします。

  C このオプションの **-qinfo** 形式と **-qinfo=all** 形式は同じ効果を持ちます。

  C++ このオプションの **-qinfo** 形式と **-qinfo=all** 形式は両方とも同じ効果を持ちますが、**ppt** グループ (プロセッサー・トレース) は含みません。

lan
: 言語レベルの影響を通知する診断メッセージを使用可能にします。これは C++ コンパイルのデフォルトです。

noall
: プログラムの特定の部分について、すべての診断メッセージをオフにします。

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 特定のメッセージのグループをオンまたはオフにします。ここで、*group* は次の 1 つ以上です。

グループ	戻される (抑制される) 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。
cls|nocls	C++ クラス。
cmp|nocmp	符号なしの比較において起こりうる冗長。
cnd|nocnd	条件式において起こりうる冗長または問題。
cns|nocns	定数が関係する演算。
cnv|nocnv	型変換。
dcl|nodcl	宣言の整合性。
eff|noeff	無効なステートメントおよびプラグマ。
enu|noenu	enum 変数の整合性。
ext|noext	未使用の外部定義。
gen|nogen	汎用診断メッセージ。
gnr|nognr	一時変数の生成。
got|nogot	goto 文の使用。
ini|noini	初期化で起こりうる問題。
inl|noinl	関数がインライン化されていない。
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。
vft|novft	C++ プログラムでの仮想関数テーブルの生成。
zea|nozea	ゼロ・エクステントの配列。

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き、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
```

関連情報

- 108 ページの『-qhaltonmsg』
- 209 ページの『-qsuppress』
- リストとメッセージを制御するオプション: メッセージのオプション

-qinitauto

説明

自動変数を 2 桁の 16 進バイト値 *hex_value* に初期化する。

構文

278 ページの『#pragma options』も参照してください。

注

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。

-qinitauto の初期値にはデフォルト設定がありません。明示的な値 (例えば、**-qinitauto=FA**) を設定する必要があります。

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
```

関連情報

- エラー検査とデバッグのオプション: その他のエラー検査とデバッグのオプション

-qinlglue

説明

外部関数の呼び出しまたは関数ポインターを介した呼び出しに必要なポインター・グルー・コードをインライン化することにより、高速な外部結合を生成する。

構文

278 ページの『#pragma options』も参照してください。

注

このオプションは 64 ビット・コンパイルにのみ適用されます。

グルー・コード はリンケージ・エディタ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たり、ポインターを介して関数を呼び出すときに使用します。したがって、**-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。

選択されたアーキテクチャーでのパフォーマンスの強化のために、グルー・コードのインライン化はハードウェア・チューニング・オプションの選択により自動化されるようになりました。**-qtune=pwr4**、**-qtune=pwr5**、**-qtune=ppc970**、または **-qtune=auto** を、これらのアーキテクチャーの 1 つを使用するシステム上で指定すると、**-qinlglue** オプションが自動的に使用可能になります。**-qtune** オプションをこのいずれかのサブオプションと共に使用して、グルー・コードのインライン化を使用不可にしたい場合は、必ず **-qnoinlglue** も指定してください。ただし、**-qcompact** は、他に指定されたオプションに関係なく、**-qinlglue** 設定をオーバーライドするため、**-qinlglue** を使用可能にしたい場合は、**-qcompact** を指定しないよう注意してください。

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。オプション **-qcompact** はコード・サイズを縮小しますが、**-qcompact** は他に指定されているオプションに関係なく、**-qinlglue** をオーバーライドするので注意してください。

関連情報

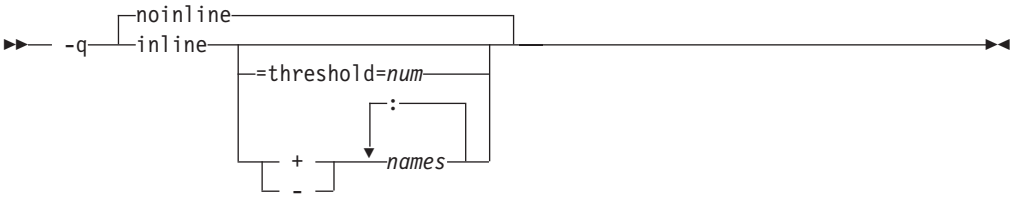
- 51 ページの『-q32、-q64』
- 73 ページの『-qcompact』
- 182 ページの『-qproclocal、-qprocimported、-qprocunknown』
- 223 ページの『-qtune』
- リンクを制御するオプション: その他のリンカー・オプション

-qinline

説明

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。インライン化は可能な場合に実行されますが、どの最適化が行われるに応じて、インライン化されない関数もあります。

構文



C C 言語では、以下の **-qinline** オプションが適用されます。

-qinline

コンパイラーは、**-qinline** オプションのサブオプションの他のすべての設定に従って、実行可能ソース・ステートメントが 20 以下のすべての適切な関数のインライン化を試行します。 **-qinline** が最後に指定された場合は、すべての関数がインライン化されます。

-qinline=threshold=*num*

インライン化する関数に対してサイズ制限を設定します。実行可能ステートメントの数は、インライン化する関数の *num* 以下でなければなりません。 *num* は正の整数でなければなりません。デフォルト値は 20 です。しきい値 **0** を指定すると、`inline` 関数指定子のサポートされる形式でマークされた関数を除き、関数はインライン化されません。

num 値は、論理 C ステートメントに適用されます。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 */
   }
}
```

-qinline-*names*

コンパイラーは、*names* にリストされた関数をインライン化しません。それぞれの *name* はコロン (**:**) で分離します。他のすべての適切な関数はインライン化されます。このオプションは、**-qinline** を暗黙指定します。

例を以下に示します。

```
-qinline-salary:taxes:expenses:benefits
```

これによって、`salary`、`taxes`、`expenses`、または `benefits` という名前の関数以外のすべての関数が、可能であればインライン化されます。

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。

-qinline+*names*

names にリストされた関数および他のすべての適切な関数をインライン化しようとします。それぞれの *name* は、コロン (**:**) で分離しなければなりません。このオプションは、**-qinline** を暗黙指定します。

例を以下に示します。

```
-qinline+food:clothes:vacation
```

これによって、インライン化に適格な他の関数とともに、可能な場合、`food`、`clothes`、または `vacation` という名前の関数がすべてインライン化されます。

ソース・ファイルに定義されていない関数、または定義はされているがインライン化できない関数については、警告メッセージが出されます。

このサブオプションは、*threshold* 値の設定をすべてオーバーライドします。 **-qinline+***names* と一緒に、ゼロのしきい値を使用して、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。

```
-qinline=threshold=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。

-qnoinline

関数を一切インライン化しません。 **-qnoinline** が最後に指定されている場合、関数は一切インライン化されません。

C++ C++ 言語では、以下の **-qinline** オプションが適用されます。

-qinline　コンパイラーは可能な関数をすべてインライン化します。
-qnoinline　コンパイラーは関数を一切インライン化しません。

デフォルト

デフォルトでは、インラインの指定はコンパイラーに対するヒントとして扱われ、選択された他のオプションに応じて結果は以下のようになります。

- **-O** コンパイラー・オプションの 1 つを使用してプログラムを最適化すると、コンパイラーはインラインとして宣言されているすべての関数のインライン化を試行します。そうでない場合、コンパイラーはインラインとして宣言されているいくつかの単純な関数だけをインライン化しようとします。

注

-qinline オプションは、**-Q** オプションと機能的に同等です。

(デバッグ情報を生成するために) **-g** オプションを指定すると、インライン化に影響する可能性があります。 104 ページの『-g』コンパイラー・オプションの情報を参照してください。

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。再帰的関数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。

通常、最適化を要求する (**-O** オプション) とアプリケーションのパフォーマンスが最適化され、最適化を要求しないとコンパイラーのパフォーマンスが最適化されます。

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** オプションを指定します。

XL C/C++ のキーワード `inline`、`__inline__`、`_inline`、`_Inline`、および `__inline` は、**-qnoinline** を除き、すべての **-qinline** オプションをオーバーライドします。コンパイラーは、他の **-qinline** オプション設定に関係なく、これらのキーワードでマークされた関数のインライン化を試行します。

詳しくは、「*XL C/C++ 言語解説書*」の『インライン関数指定子』を参照してください。

例

関数を一切インライン化しないように `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
```

コンパイラーが 12 行未満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=12
```

関連情報

- 166 ページの『-O、-qoptimize』
- 185 ページの『-Q』
- 104 ページの『-g』
- パフォーマンス最適化のオプション: 関数のインライン化のオプション

-qipa

説明

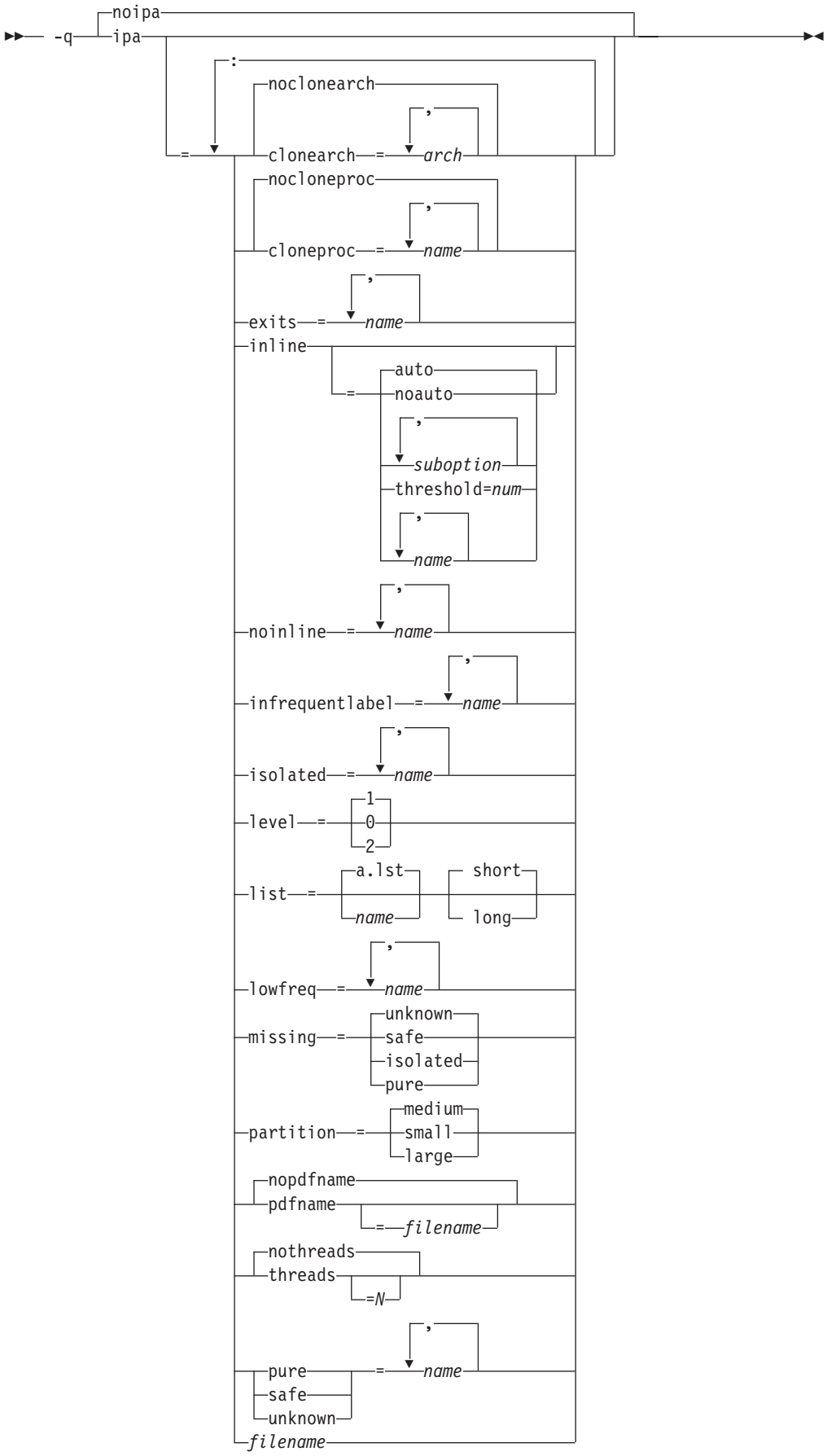
プロシージャー間分析 (IPA) と呼ばれる最適化のクラスをオンにしたりカスタマイズする。

コンパイル時の構文

ここで、

-qipa コンパイル時オプション	**説明**
-qipa	以下の **-qipa** サブオプション のデフォルトで、プロシージャー間分析を活動化します。 • **inline=auto** • **level=1** • **missing=unknown** • **partition=medium**
-qipa=object -qipa=noobject	標準のオブジェクト・コードをオブジェクト・ファイルに入れるかどうかを指定します。 **noobject** サブオプションを指定すると、最初の IPA フェーズ中にオブジェクト・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。 **-S** コンパイラー・オプションを **noobject** と一緒に指定した場合は、**noobject** は無視されます。 コンパイルおよびリンクを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していない場合は、デフォルトで **-qipa=noobject** が暗黙指定されます。 **-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。

リンク時の構文


-q
noipa
ipa
:
noclonearch
clonearch
arch
nocloneproc
cloneproc
name
exits
name
inline
auto
noauto
suboption
threshold=num
name
noinline
name
infrequentlabel
name
isolated
name
level
1
0
2
list
a.lst
name
short
long
lowfreq
name
missing
unknown
safe
isolated
pure
partition
medium
small
large
nopdfname
pdfname
filename
nothreads
threads
N
pure
safe
unknown
name
filename

ここで、

リンク時サブオプション	説明
-qnoipa	プロシージャー間分析を非活動化します。
-qipa	以下の **-qipa** サブオプション のデフォルトで、プロシージャー間分析を活動化します。 • **inline=auto** • **level=1** • **missing=unknown** • **partition=medium**

以下に示す 1 つ以上の形式をサブオプションに含めることもできます。

リンク時サブオプション	説明
clonearch=*arch*{,*arch*} 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。 IPA リンク・フェーズの間、コンパイラーはデフォルト・アーキテクチャー設定をターゲットとしたプロシージャーの汎用バージョンを生成し、その後、該当する場合は、指定されたアーキテクチャー用に最適化された別のバージョンを作成します。実行時、コンパイラーはプログラムがどのアーキテクチャー上で稼働しているかを動的に判別し、それに従って実行される関数の特定のバージョンを選択します。このオプションを使用すると、プログラムは異なる PowerPC アーキテクチャーの互換性を達成することができます。 *arch* は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 clonearch の値は **pwr4**、**pwr5**、および **ppc970** です。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。 **注:** 1. 複数のプラットフォーム間の互換性を保証するには、**-qarch** 値が、**-qarch=clonearch** によって指定されるアーキテクチャーのサブセットでなければなりません。 2. **-qcompact** が有効な場合、**-qarch=clonearch** は使用不可になります。 3. さまざまなアーキテクチャーで許可される **clonearch** 値について詳しくは、 129 ページの表 37を参照してください。 4. **-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。

リンク時サブオプション	説明
cloneproc=*name*{,*name*} nocloneproc=*name*{,*name*}	**clonearch** サブオプションによって指定されたアーキテクチャーの複製を行うための関数の名前を指定します。ここで、*name* はコンマで区切られた関数名のリストです。 **注: -qipa=clonearch** を指定しなかったり、**-qipa=noclonearch** を指定すると、**-qipa=cloneproc=***name*,{*name*} および **-qipa=nocloneproc=***name*,{*name*} は効果を持ちません。
exits=*name*{,*name*}	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プロシージャーを呼び出すことができない呼び出しのことです。
infrequentlabel=*name*{,*name*}	プログラムの実行中にまれに呼ばれる可能性の高いユーザー定義ラベルのリストを指定します。
inline=auto inline=noauto	自動インライン化だけを有効または無効にします。コンパイラーは、ユーザー指定関数をインライン化の候補として引き続き受け入れます。
inline[=*suboption*]	**-qinline** コンパイラー・オプションの指定と同じです。*suboption* は、任意の有効な **-qinline** サブオプションです。
inline=threshold=*num*	インライン化される関数の数の上限を指定します。ここで、*num* は負でない整数です。この引数は、 **inline=auto** がオンになっている場合にのみインプリメントされます。
inline=*name*{,*name*}	インライン化しようとする関数をコンマで区切ったリストを指定します。関数は *name* によって識別されます。
noinline=*name*{,*name*}	インライン化できない関数をコンマで区切ったリストを指定します。関数は *name* によって識別されます。
isolated=*name*,{*name*}	IPA でコンパイルされない*分離された* 関数のリストを指定します。分離された関数やそれらの呼び出しチェーン内の関数は、いずれもグローバル変数を参照することができません。
level=0 level=1 level=2	プロシージャー間分析の最適化レベルを指定します。デフォルトのレベルは 1 です。有効なレベルは以下のとおりです。 • Level 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。 • Level 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オンにします。 • Level 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。

リンク時サブオプション	説明
list list=[*name*] [shortllong]	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、区画ごとにバックエンドによって生成される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。このオプションを使用して、リスト・ファイルの名前を指定することもできます。 (**-qlist** か **-qipa=list** オプションのいずれかを使用して) リストを要求したときに *name* が指定されていない場合は、リスト・ファイルの名前はデフォルトで **a.lst** になります。 **long** および **short** サブオプションを使用して、リスト・ファイルの詳細化や簡略化を要求することができます。**short** サブオプションはデフォルトで、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** サブオプションでは、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=*name*{,*name*}	頻繁に呼び出されないと考えられる関数の名前を指定します。一般には、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小規模にすることによって、コンパイラーが、プログラムのその他の部分がより高速に実行されるようにできる場合があります。
missing=*attribute*	**-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プロシージャーのプロシージャー間の振る舞いを指定します。 以下の属性を使用して、この情報の詳細を指定することができます。 • safe - 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関数。 • isolated - 可視の関数からアクセスできるグローバル変数を直接参照しない関数。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 - *安全 (safe)* かつ*分離された (isolated)* 関数であり、可視の関数からアクセスできるストレージを間接的に変更しない関数。また、*純粋な (pure)* 関数には、取得できる内部状態はありません。 • unknown - デフォルトの設定。このオプションは、不明の (*unknown*) 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。欠落している関数が*安全 (safe)*、*分離された (isolated)*、または*純粋な (pure)* 関数として認識されないように指定します。

リンク時サブオプション	説明
partition=small partition=medium partition=large	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。
nopdfname pdfname pdfname=*filename*	PDF プロファイル情報を含むプロファイル・データ・ファイルの名前を指定します。*filename* を指定しない場合、デフォルト・ファイル名は **._pdf** になります。 プロファイルは、現行作業ディレクトリーか、PDFDIR 環境変数によって指定されたディレクトリーに配置されます。これにより、同じ PDFDIR を使用して複数の実行可能ファイルを同時に実行することが可能になります。これは動的ライブラリー上の PDF でのチューニングに役立ちます。
nothreads threads threads=*N*	コンパイラーがコード生成に割り当てるスレッド数を指定します。 **nothreads** を指定することは、1 つのシリアル処理を実行することと同等です。これがデフォルトです。 **threads** を指定すると、コンパイラーが、使用可能なプロセッサーの数に応じて、使用するスレッドの数を判別できるようになります。 **threads=***N* を指定すると、 *N* スレッドを使用するようプログラムに指示します。 *N* は 1 から MAXINT の範囲の整数値にすることができますが、 事実上 *N* は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。
pure=*name*{,*name*}	**-qipa** でコンパイルされない*純粋な* 関数のリストを指定します。*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=*name*{,*name*}	**-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)* 関数のリストを指定します。安全な関数は、グローバル変数を変更することができますが、 **-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。
unknown=*name*{,*name*}	**-qipa** でコンパイルされない*不明の (unknown)* 関数のリストを指定します。*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、グローバル変数やダミー引数を変更することができます。

リンク時サブオプション	説明

filename	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}`
	`missing = attribute{, attribute}`
	`exits = name{, name}`
	`lowfreq = name{, name}`
	`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`
	`prof [ = file-name ]`
	`noprof`
	`partition = small | medium | large | unsigned_int`
	
	ここで、*attribute* は以下のいずれかです。
	• clonearch
	• cloneproc
	• exits
	• lowfreq
	• unknown
	• safe
	• isolated
	• pure

以下のテーブルは、異なる **-qarch** 設定に対して許可される **clonearch** の値を示したものです。

表 37. 許可される clonearch 値

-qarch 設定	**許可される clonearch 値**
ppc、 pwr3、 ppc64、 ppcgr、 ppc64gr、 ppc64grsq	pwr4、pwr5、ppc970
pwr4	pwr5、ppc970
ppc64v	ppc970
pwr5、ppc970	N/A

注

IPA を使用するために必要なステップは、以下の通りです。

1. IPA 分析は、コンパイルとリンクの時間を増加させる 2 パス・メカニズムを使用するため、 **-qipa** オプションを指定してコンパイルする前に、予備のパフォーマンス分析とチューニングを行ってください。 **-qipa=noobject** オプションを使用することによって、コンパイルおよびリンクのオーバーヘッドをいくらか削減することができます。

2. **-qipa** オプションは、アプリケーション全体のコンパイルとリンクの両方のステップ、またはできるだけ多くのステップにおいて指定します。サブオプションを使用して、 **-qipa** でコンパイルされない プログラムの部分に関する前提を示します。コンパイル中に、コンパイラーはプロシージャー間分析情報を .o ファイルに保管します。リンク中には、 **-qipa** オプションによって、アプリケーション全体の完全な再コンパイルが行われます。

注: コンパイル中に重大エラーが発生すると、**-qipa** は RC=1 を戻して終了します。パフォーマンス分析も終了します。

- IPA は、**-qipa=noobject** オプションを指定した場合でもコンパイル時間を大幅に増加させる可能性があるので、IPA の使用は開発の最終のパフォーマンス・チューニング段階に限定してください。
- **-qipa** オプションは、アプリケーション全体のコンパイルとリンクの両方のステップ、またはできるだけ多くのステップにおいて指定します。少なくとも `main` を含むファイル、またはライブラリーをコンパイルしている場合はエントリー・ポイントの少なくとも 1 つをコンパイルしなければなりません。
- IPA によるプロシージャー間の最適化によって、プログラムのパフォーマンスを大幅に向上させることができますが、以前は誤っていても機能していたプログラムが失敗する可能性もあります。 以下に、積極的な最適化を行わなくても偶然に機能するが、IPA で明らかになるプログラミングの例をいくつかリストします。
  1. 割り振り順序または自動変数の位置に依存するもの。例えば、自動変数のアドレスを取得してから別のローカル変数のアドレスと後で比較して、スタックの拡大方向を判別するものです。C 言語では、自動変数が割り振られる位置や、他の自動変数に対する相対位置は保証されていません。そのような関数を IPA でコンパイルしないでください (機能しません)。
  2. 無効なポインターのアクセス、または配列の境界を超えた位置のアクセス。IPA はグローバル・データ構造を再編成する場合があります。 参照先が不正なポインターによって未使用のメモリーを以前に変更した可能性がある場合は、ユーザーが割り振ったストレージを破壊する場合があります。
- IPA でコンパイルするためのリソースが十分にあることを確認してください。IPA は従来のコンパイルよりもかなり大きなオブジェクト・ファイルを生成することができます。その結果、これらの中間ファイルの保持に使用される一時ストレージのロケーション (規則により、`/tmp`) が小さすぎる場合があります。大きなアプリケーションをコンパイルしている場合は、TMPDIR 環境変数を使用して一時記憶域の宛先を変更することを検討してください。
- IPA を実行するために十分なスワップ・スペース (大規模なプログラムの場合は少なくとも 200MB) があることを確認してください。ない場合は、オペレーティング・システムによってシグナル 9 で IPA が強制終了されます。これをトラップすることはできず、IPA はその一時ファイルをクリーンアップすることができません。
- 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。
- ソース・コードに明らかに参照されている、または設定されているいくつかのシンボルは、IPA によって最適化される場合があり、デバッグ、nm、またはダンプ

出力に対して失われる場合があります。 **-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。

次のサブオプションで *name* を指定するときに、正規表現式構文を使用できます。

- cloneproc、nocloneproc
- exits
- inline、noinline
- isolated
- lowfreq
- pure
- safe
- unknown

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***$**	*string* で指定されたパターンが行の終わりにある場合にのみ、そのパターンと一致します。
*str***.***ing*	ピリオド (.) は任意の 1 文字と一致します。例えば、t.st は、test、tast、tZst、および t1st と一致します。
*string***¥***special_char*	円記号 (¥) は、特殊文字をエスケープするために使用できます。例えば、ピリオドで終わる行を検索したい場合、式 .$ 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。

例

ファイルのセットをプロシージャー間分析でコンパイルするには、以下のように入力します。

```
xlc++ -c -O3 *.C -qipa
xlc++ -o product *.o -qipa
```

同じファイルのセットをコンパイルして、2 番目のコンパイルの最適化と最初のコンパイル・ステップの速度を改善する方法を以下に示します。ほとんど実行されることのない 2 つの関数 trace_error と debug_dump が存在すると想定します。

```
xlc++ -c -O3 *.C -qipa=noobject
xlc++ -c *.o -qipa=lowfreq=trace_error,debug_dump
```

関連情報

- 152 ページの『-qlibansi』
- 154 ページの『-qlist』
- 193 ページの『-S』
- パフォーマンス最適化のオプション: プログラム全体の分析のオプション
- 「*XL C/C++ 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

説明

ソース・ファイル内の副次作用がない関数を指定する。

構文

ここで、

function_name 副次作用がない関数 (ポインターまたは参照パラメーターが指す変数の値を変更する場合は除く)、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。

副次作用 とは、ランタイム環境の状態の変更です。このような変更の例としては、揮発オブジェクトへのアクセス、外部オブジェクトの変更、ファイルの変更、またはこれらのいずれかを行う別の関数の呼び出しなどが挙げられます。 副次作用のない関数は、外部変数および静的変数を変更しません。

function_name には、関数をコロン (:) で区切ってリストすることができます。

268 ページの『#pragma isolated_call』および 278 ページの『#pragma options』も参照してください。

注

関数に分離のマークを付けると、以下のことを最適化プログラムに示すことにより、最適化されたコードのランタイム・パフォーマンスを改善することができます。

- 外部変数および静的変数が呼び出し先関数によって変更されていない
- ループ不変パラメーター付きの関数呼び出しをループの外に移動できる
- 同じパラメーターでの複数の関数呼び出しを 1 つの呼び出しにまとめることができる
- 結果値が必要ない場合には、関数呼び出しを廃棄できる

#pragma options isolated_call ディレクティブは、最初の C または C++ ステートメントの前の、ファイルの先頭に指定する必要があります。**#pragma isolated_call** ディレクティブは、ソース・ファイルの任意の位置で使用することができます。

誤って副次作用を持っていないと関数が識別されると、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。

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
```

関連情報

- パフォーマンス最適化のオプション: 副次作用のオプション

-qkeepinlines

C++

説明

参照されていない extern インライン関数の定義を保持または廃棄するようにコンパイラーに命令する。

構文

注

デフォルトの **-qnokeepinlines** 設定は、参照されていない外部インライン関数の定義を廃棄するようコンパイラーに命令します。これによって、オブジェクト・ファイルのサイズを削減することができます。

-qkeepinlines 設定は、参照されていない外部インライン関数の定義を保持します。この設定は、v5.0.2.1 更新レベルより前の VisualAge C++ コンパイラーと同じ振る舞いを提供し、共用ライブラリーと旧リリースのコンパイラーで作成されたオブジェクト・ファイルとの間の互換性を保つようにします。

関連情報

- 119 ページの『-qinline』
- パフォーマンス最適化のオプション: コード・サイズ縮小のオプション

-qkeepparm

説明

アプリケーションが最適化されていても、関数仮パラメーターがスタックに確実に保管されるようにする。

構文

注

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 を指定すると、最適化時でもパラメーターが必ずスタックに保管されます。このコンパイラー・オプションは、これらの値をスタックに保存することにより、デバッガーなどのツールが着信パラメーターの値を使用できるようにします。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。

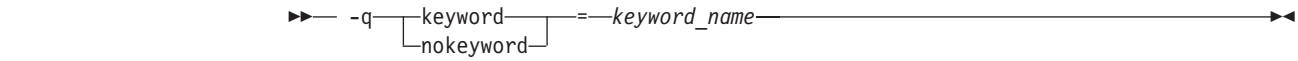
関連情報
- 166 ページの『-O、-qoptimize』
- エラー検査とデバッグのオプション: その他のエラー検査とデバッグのオプション

-qkeyword

説明

このオプションは、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



注

デフォルトでは、C および C++ 言語標準に定義されている組み込みキーワードはすべてキーワードとして予約されています。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* を使用して組み込みキーワードを使用不可にし、**-qkeyword=***keyword_name* を使用してこれらのキーワードを復元することができます。

このオプションは、すべての C++ 組み込みキーワードで使用できます。

このオプションは、以下の C キーワードにも使用することができます。

- asm
- inline
- restrict
- typeof

注: C asm は、**-qlanglvl** オプションが **ansi**、**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。

例

C++

以下の呼び出しを使用して、bool を復元することができます。

```
xlc++ -qkeyword=bool
```

C

以下の呼び出しを使用して、typeof を復元することができます。

```
xlc -qkeyword=typeof
```

関連情報
- 61 ページの『-qasm』
- 入力を制御するオプション: 言語拡張のオプション

-L

説明

-l*key* オプションによって指定されたライブラリー・ファイルのパス・ディレクトリーをリンク時に検索する。

構文

デフォルト

デフォルトでは、標準のディレクトリーしか検索されません。

注

LIBPATH 環境変数が設定されている場合、コンパイラーはまず LIBPATH によって指定されたディレクトリー・パスでライブラリーを検索し、次に **-L** コンパイラー・オプションによって指定されたディレクトリー・パスを検索します。

-L*directory* オプションが構成ファイルとコマンド行の両方に指定されている場合は、構成ファイルに指定された検索パスがリンク時に最初に検索されます。

-L コンパイラー・オプションで指定されたパスは、実行時に検索されません。

例

ディレクトリー /usr/tmp/old と、**-l** オプションによって指定された他のすべてのディレクトリーでライブラリー libspfiles.a を検索するよう 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 -lspfiles -L/usr/tmp/old
```

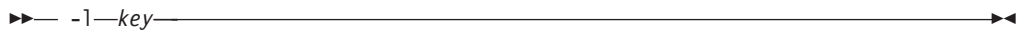
関連情報
- 『-l』
- 335 ページの『付録 A. 再配布可能ライブラリー』
- リンクを制御するオプション: リンカー入力制御のオプション

-l

説明

指定されたライブラリー・ファイル lib*key*.so 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 だけを検索する。

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。

注

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***directory* オプションを使用して変更することができます。

C および C++ ランタイム・ライブラリーは自動的に追加されます。

-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。

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)` にリンクするには、以下のように入力します。

```
xlc++ myprogram.C -lmylibrary -L/usr/mylibdir
```

関連情報

- 64 ページの『-B』
- 136 ページの『-L』
- 29 ページの『リンクの順序』
- 22 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』
- リンクを制御するオプション: リンカー入力制御のオプション

-qlanglvl

説明

コンパイルのための言語レベルと言語オプションを選択する。

構文

suboption の値は下記の『注』セクションに記載されています。

270 ページの『#pragma langlvl』および 278 ページの『#pragma options』も参照してください。

デフォルト

デフォルトの言語レベルは、コンパイラー呼び出しに使用するコマンドによって異なります。

呼び出し	デフォルト言語レベル
xlC/xlc++	extended
xlc	extc89
cc	extended
c89	stdc89
c99	stdc99

注

C 以下のいずれかのプラグマ・ディレクティブを使用して、C 言語ソース・プログラムの言語レベルを指定することもできます。

```
#pragma options langlvl=suboption
#pragma langlvl(suboption)
```

プラグマ・・ディレクティブは、ソース・コードのコメント以外のどの行よりも前に指定する必要があります。

C C プログラムでは、以下の **-qlanglvl** サブオプションを *suboption* に使用することができます。

classic	stdc89 以外の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。
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 標準に準拠するコンパイル。
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。

[no]ucs

言語レベル **stdc99** および **extc99** では、デフォルトは **-qlanglvl=ucs** です。

このオプションは、ユニコード文字が、プログラム・ソース・コードの ID、ストリング・リテラル、および文字リテラルで許可されるかどうかを制御します。

ユニコード文字セットは、C 標準でサポートされています。この文字セットには、北米および西ヨーロッパのすべての言語を含む、幅広い範囲の言語に使用される文字、数字、およびその他の文字の全セットが含まれています。ユニコード文字は、16 ビットまたは 32 ビットが可能です。 ASCII の 1 バイト文字は、ユニコード文字セットのサブセットです。

このオプションが、yes に設定されている場合は、ソース・ファイルに直接か、またはエスケープ・シーケンスに類似した表記を使用して、ユニコード文字を挿入することができます。ユニコード文字の多くが、画面に表示できない、またはキーボードから入力できないため、通常は、後者の方法が好まれています。ユニコード文字の表記形式は、16 ビット文字の場合は ¥u*hhhh*、 32 ビット文字の場合は ¥U*hhhhhhhh* です。*h* は、16 進数字を表します。文字の短縮 ID は、ISO/IEC 10646 によって指定されます。

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、**-qlanglvl=extended**、**-qlanglvl=extc99**、または **-qlanglvl=extc89** を使用してください。 **-qlanglvl** に他の値を指定すると、これらのサブオプションによって暗黙指定される関数は使用不可になります。

[no]gnu_assert GNU C 移植性オプション。
[no]gnu_explicitregvar GNU C 移植性オプション。
[no]gnu_include_next GNU C 移植性オプション。
[no]gnu_locallabel GNU C 移植性オプション。
[no]gnu_warning GNU C 移植性オプション。

C++ C++ プログラムでは、以下の 1 つ以上の **-qlanglvl** サブオプションを *suboption* に指定できます。

extended

コンパイルは ISO C++ 標準 を基にしますが、拡張言語フィーチャーの適応については若干の違いがあります。

[no]anonstruct

このサブオプションは、無名の構造体と無名のクラスが C++ ソースで許可されるかどうかを制御します。

デフォルトでは、コンパイラーは無名の構造体を許可します。これは C++ 標準の拡張で、Microsoft® Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

無名の構造体は、以下のコード・フラグメントにあるように、たいていの場合は共用体で使用されます。

```
union U {
   struct {
      int i:16;
      int j:16;
   };
   int k;
} u;
// ...
u.j=3;
```

このサブオプションが設定されていると、コードが無名の構造体を宣言し、**-qinfo=por** が指定されている場合は、警告が出ます。 **-qlanglvl=noanonstruct** でビルドすると、無名の構造体にエラーのフラグが付けられます。標準 C++ に準拠する場合は、**noanonstruct** を指定してください。

[no]anonunion

このサブオプションは、どのメンバーが無名の共用体で許可されるかを制御します。

このサブオプションが **anonunion** に設定されていると、無名の共用体は、標準 C++ が無名の共用体以外で許可するすべての型のメンバーを持つことができます。例えば、構造体、typedef、および列挙型などの非データ・メンバーは許可されます。

メンバー関数、仮想関数、または単純ではないデフォルト・コンストラクター、コピー・コンストラクター、またはデストラクターを持つクラスのオブジェクトは、このオプションの設定にかかわらず共用体のメンバーにはなれません。

デフォルトでは、コンパイラーは無名共用体の非データ・メンバーを許可します。これは標準 C++ の拡張で、VisualAge C++ の前のバージョンと先行製品、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

このオプションが **anonunion** に設定されていると、コードがその拡張子を使用した場合、 **-qsuppress** オプションで警告メッセージを抑止しない限り、警告が出ます。

標準 C++ に準拠する場合は、**noanonunion** を設定してください。

[no]ansifor

このサブオプションは、C++ 標準に定義されているスコープ規則を for-init ステートメントで宣言されている名前に適用するかどうかを制御します。

デフォルトでは、標準 C++ 規則が使用されます。例えば、以下のコードでは名前検索エラーが起こります。

```
{
   //...
   for (int i=1; i<5; i++) {
      cout << i * 2 << endl;
   }
   i = 10;  // error
}
```

エラーの理由は、i が、あるいは for-init ステートメント内で宣言されたいずれかの名前が、for ステートメント内でのみ可視であるからです。このエラーを訂正するには、i をループの外で宣言するか、ansiForStatementScopes を no に設定します。

noansifor を設定して、古い言語の振る舞いを許可します。これは、VisualAge C++ の旧版や先行製品、および Microsoft Visual C++ によって提供されたコンパイラーなど、他の製品を使用して開発されたコードに対して行う必要がある場合があります。

[no]ansisinit

このオプションは、**g++ -fuse-cxa-atexit** と同様に機能し、静的デコンストラクターの完全な標準に準拠した処理のために必要です。

[no]c99__func__

このサブオプションは、C99 __func__ ID を認識するようコンパイラーに命令します。__func__ ID は、以下の暗黙宣言が存在するかのように振る舞います。

```
static const char __func__[] = function_name;
```

ここで、*function_name* は __func__ ID が現れる関数の名前です。

__func__ ID の効果は、以下のコード・セグメントで見ることができます。

```
void this_function()
{
 printf("__func__ appears in %s", __func__);
}
```

これは実行されると、以下を出力します。

```
__func__ appears in this_function
```

c99__func__ サブオプションは、**-qlanglvl=extended** が有効な場合、デフォルトで使用可能です。これはどの言語レベルでも、**-qlanglvl=c99__func__** を指定して使用可能にすることができ、**-qlanglvl=noc99__func__** を指定して使用不可にすることができます。

__C99__FUNC__ マクロは、**c99__func__** が有効な場合は 1 に定義され、そうでない場合は定義されません。

[no]c99complex

このサブオプションは C99 複素数データ型と関連キーワードを認識するようコンパイラーに命令します。

注: 複素数データ型に対するサポートは各種 C++ コンパイラーごとに異なるため、潜在的な移植性の問題を作り出します。このコンパイラー・オプションを **-qinfo=por** と共に指定すると、コンパイラーが移植性の警告メッセージを発行します。

[no]c99compoundliteral

このサブオプションは C99 複合リテラル・フィーチャーをサポートするようコンパイラーに命令します。

[no]c99hexfloat

このオプションは C++ アプリケーションで C99 スタイルの 16 進浮動小数点定数に対するサポートを使用可能にします。**-qlanglvl=extended** の場合、このサブオプションはデフォルトでオンになります。これが有効な場合、コンパイラーはマクロ __C99_HEX_FLOAT_CONST を定義します。

[no]c99vla

c99vla が有効な場合、コンパイラーは C99 型可変長配列を C++ アプリケーションでサポートします。マクロ __C99_VARIABLE_LENGTH_ARRAY は 1 の値で定義されます。

注: C++ アプリケーションでは、可変長配列によって使用されるよう割り振られているストレージは、それらの配列が常駐する関数が実行を完了するまで解放されません。

[no]dependentbaselookup

デフォルトは **-qlanglvl=dependentbaselookup** です。

このサブオプションは、C++ 標準の TC1 の Issue 213 に準拠するコンパイルを指定する能力を提供します。

デフォルト設定は、従属型のテンプレート基底クラスの名前ルックアップに関して、以前の XL C/C++ コンパイラーの振る舞いを保存します。従属型である基底クラスのメンバーはテンプレート内で宣言されている名前、またはテンプレートのエンクロージング・スクープ内からの名前を隠します。

TC1 の準拠のために、**-qlanglvl=nodependentbaselookup** を指定します。

[no]gnu_assert

以下の GNU C システム識別表明のサポートを使用可能または使用不可にするための GNU C 移植性オプションです。

- #assert
- #unassert
- #cpu
- #machine
- #system

[no]gnu_complex

このサブオプションは GNU 複素数データ型と関連キーワードを認識するようコンパイラーに命令します。
注: 複素数データ型に対するサポートは各種 C++ コンパイラーごとに異なるため、潜在的な移植性の問題を作り出します。このコンパイラー・オプションを **-qinfo=por** と共に指定すると、コンパイラーが移植性の警告メッセージを発行します。

[no]gnu_computedgoto

計算された goto に対するサポートを使用可能にする GNU C 移植性オプション。このサブオプションは、 **-qlanglvl=extended** に対して使用可能になり、マクロ __IBM_COMPUTED_GOTO を定義します。

[no]gnu_externtemplate

このサブオプションは、extern テンプレートのインスタンス化を使用可能または使用不可にします。

デフォルト設定は、**extended** 言語レベルにコンパイルする場合は **nognu_externtemplate** です。

gnu_externtemplate が有効な場合、明示的 C++ テンプレート・インスタンス化の前にキーワード `extern` を追加することによってテンプレートのインスタンス化が extern であると宣言することができます。 `extern` キーワードは、宣言内の最初のキーワードでなければならず、`extern` キーワードは 1 つしか使用できません。

これは、クラスまたは関数のインスタンスを生成しません。クラスおよび関数の両方で、extern テンプレートのインスタンス化が、extern テンプレート・インスタンス化に先行するコードによってトリガーされておらず、明示的にインスタンスを生成されているのでも、明示的に特殊化されているのでもなければ、その extern テンプレートのインスタンス化はテンプレートのパーツのインスタンス化を妨げます。

クラスの場合、静的データ・メンバーおよびメンバー関数のインスタンスは生成されませんが、クラスをマップするために必要であれば、クラス自体のインスタンスは生成されます。必要コンパイラー生成関数 (例えば、デフォルト・コピー・コンストラクター) のインスタンスは生成されます。関数の場合、プロトタイプのインスタンスは生成されますが、テンプレート関数の本体のインスタンスは生成されません。

以下の例を参照してください。

```
template < class T > class C {
  static int i;
  void f(T) { }
};

template < class U > int C<U>::i = 0;
extern template class C<int>; // extern explicit
                                  // template
                                  // instantiation
C<int> c;  // does not cause instantiation of
           // C<int>::i  or C<int>::f(int) in
           // this file but class is
           // instantiated for mapping
C<char> d; // normal instantiations

==========================

template < class C > C foo(C c) { return c; }

extern template int foo<int>(int); // extern explicit
                                   // template
                                   // instantiation
int i = foo(1);    // does not cause instantiation
                   // of body of foo<int>
```

[no]gnu_include_next

GNU C `#include_next` プリプロセッサー・ディレクティブのサポートを使用可能または使用不可にするための GNU C 移植性オプションです。

[no]gnu_labelvalue

値としてのラベルに対するサポートを使用可能または使用不可にするための GNU C 移植性オプション。このサブオプションは、 **-qlanglvl=extended** の場合はデフォルトでオンで、マクロ __IBM_LABEL_VALUE を定義します。

[no]gnu_locallabel

ローカル宣言ラベルのサポートを使用可能または使用不可にするための GNU C 移植性オプションです。

gnu_membernamereuse

typedef としてメンバー・リスト内のテンプレート名の再使用を使用可能にする GNU C++ 移植性オプション。

[no]gnu_suffixij

GNU スタイルの複素数に対するサポートを使用可能または使用不可にするための GNU C 移植性オプション。**gnu_suffixij** が指定されている場合、複素数は i/I または j/J のサフィックスで終了することができます。

[no]gnu_varargmacros

このオプションは **-qlanglvl=varargmacros** に似ています。主な違いは以下の通りです。

- オプションの変数引数 ID が省略符号の前に指定され、ID をマクロ __VA_ARGS__ の代わりに使用できるようにする。ID と省略符号の間に空白が表示されることがあります。
- 変数引数を省略することができる。
- トークン貼り付け演算子 (**##**) がコンマと変数引数の間に現れると、変数引数が提供されていない場合、プリプロセッサーがコンマ (,)を除去する。
- macro __IBM_MACRO_WITH_VA_ARGS が 1 に定義される。

例 1 - 単純な置換:

```
#define debug(format, args...) printf(format, args)

debug("Hello %s¥n", "Chris");
```

次のようにプリプロセスされます。

```
printf("Hello %s¥n", "Chris");
```

例 2 - 変数引数の省略:

```
#define debug(format, args...) printf(format, args)

debug("Hello¥n");
```

コンマはそのままで、次のようにプリプロセスされます。

```
printf("Hello¥n",);
```

例 3 - 変数引数が省略されるときにコンマを除去するためのトークン貼り付け演算子の使用:

```
#define debug(format, args...) printf(format, ## args)

debug("Hello¥n");
```

次のようにプリプロセスされます。

```
printf("Hello¥n");
```

[no]gnu_warning

GNU C `#warning` プリプロセッサー・ディレクティブのサポートを使用可能または使用不可にするための GNU C 移植性オプションです。

[no]illptom

このサブオプションは、メンバーへのポインターを形成するのにどの式が使用できるかを制御します。 XL C++ コンパイラーは、共通に使用されるが、C++ 標準に準拠していない幾つかの形式を受け入れることができます。

コンパイラーはデフォルトでこれらの形式を許可します。これは標準 C++ の拡張で、VisualAge C++ の前のバージョン、その先行製品、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

このサブオプションが **illptom** に設定されていると、コードがその拡張子を使用した場合、 **-qsuppress** オプションで警告メッセージを抑止しない限り、警告が出ます。

例えば、以下のコードは、関数のメンバー `p` へのポインターを定義し、それを古いスタイルで `C::foo` のアドレスへ初期設定します。

```
struct C {
void foo(int);
};

void (C::*p) (int) = C::foo;
```

C++ 標準に準拠する場合は、**noillptom** を設定してください。上記のコード例では、& 演算子を使用するよう変更しなければなりません。

```
struct C {
void foo(int);
};

void (C::*p) (int) = &C::foo;
```

[no]implicitint

このサブオプションは、コンパイラーが欠落しているか部分的に指定されている型を暗黙指定の int として受け入れるかどうかを制御します。これは、標準では現在受け入れられてはいませんが、既存のコードには存在する場合があります。

サブオプション・セットが **noimplicitint** に設定されている場合は、すべての型が完全に指定されていなければなりません。

サブオプション・セットが **implicitint** に設定されている場合は、ネーム・スペース・スコープでの関数宣言、またはメンバー・リスト内の関数宣言は、`int` を戻すよう暗黙的に宣言されます。また、型を完全に指定しない宣言指定子のシーケンスは、いずれも、整数型を暗黙的に指定します。あたかも `int` 指定子が存在しているかのような効果がありますので、注意してください。これはつまり、指定子 `const` が自ら定数整数を指定することを意味します。

以下の指定子は、型を完全に指定しません。
- auto
- const
- extern
- extern “<literal>”
- inline
- mutable
- friend
- register
- static
- typedef
- virtual
- volatile
- プラットフォーム特定の型 (例えば、_cdecl)

型が指定されている状態は、いずれも、このサブオプションの影響を受けるので、注意してください。これには、例えば、テンプレート型およびパラメーター型、例外指定、式における型 (`casts`、`dynamic_cast`、`new` など)、および変換関数の型が含まれます。

コンパイラーはデフォルトで **-qlanglvl=implicitint** を設定します。これは C++ 標準の拡張で、VisualAge C++ の前のバージョンと先行製品、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

例えば、関数 `MyFunction` の戻りの型は、以下のコードで省略されたため、`int` です。

```
MyFunction()
{
  return 0;
}
```

標準 C++ に準拠する場合は、**-qlanglvl=noimplicitint** を設定してください。例えば、上記の関数宣言は以下のように変更する必要があります。

```
int MyFunction()
{
  return 0;
}
```

[no]offsetnonpod

このサブオプションは、`offsetof` マクロを、データだけでないクラスに適用できるかどうかを制御します。C++ プログラマーはよくデータだけのクラスを「Plain Old Data」(POD) クラスと呼びます。

コンパイラーはデフォルトで、`offsetof` を POD でないクラスに使用することを許可します。これは C++ 標準の拡張で、VisualAge C++ for OS/2® 3.0、VisualAge for C++ for Windows®、バージョン 3.5、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

このオプションが設定されていると、コードがその拡張子を使用した場合、**-qsuppress** オプションで警告メッセージを抑止しない限り、警告が出ます。

標準 C++ に準拠する場合は、**-qlanglvl=nooffsetnonpod** を設定してください。

コードが次のいずれか 1 つを含むクラスへ `offsetof` を適用する場合は、**-qlanglvl=offsetnonpod** を設定してください。

- ユーザー宣言のコンストラクターまたはデストラクター
- ユーザー宣言の代入演算子
- private または protected 非静的データ・メンバー
- 基底クラス
- 仮想関数
- メンバーへの型ポインターの非静的データ・メンバー
- 非データ・メンバーを持つ構造体または共用体
- 参照

[no]olddigraph

このオプションは、古いスタイルの連字が C++ ソースで許可されるかどうかを制御します。これは、**-qdigraph** も設定されている場合にのみ適用されます。

コンパイラーはデフォルトで、C++ 標準に指定された連字のみをサポートします。

以下の連字のうち少なくとも 1 つがコードに含まれている場合は、**-qlanglvl=olddigraph** を設定します。

連字	**結果の文字**
%:	# (ポンド記号)
%:%:	## (ダブル・ポンド記号、プリプロセッサー・マクロの連結演算子として使用される)

標準 C++ および VisualAge C++ の前のバージョンおよび先行製品によってサポートされている拡張 C++ 言語レベルとの互換性のために、**-qlanglvl=noolddigraph** を設定します。

[no]oldfriend

このオプションは、詳述されたクラス名なしでクラスを指定するフレンド宣言を C++ エラーとして取り扱うかどうかを制御します。

デフォルトで、コンパイラーはキーワード・クラスを持つクラスの名前を詳述せずにフレンド・クラスを宣言できるようになっています。これは C++ 標準の拡張で、VisualAge C++ の前のバージョンと先行製品、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

例えば、下記のステートメントは、クラス IFont がフレンド・クラスになるように宣言し、 **oldfriend** サブオプションが設定されている場合に有効です。

```
friend IFont;
```

標準 C++ に準拠する場合は、**nooldfriend** サブオプションを設定してください。上記の宣言例は、下記のステートメントに変更するか、または **-qsuppress** オプションで警告メッセージを抑止しない限り、警告を出します。

```
friend class IFont;
```

[no]oldtempacc

このサブオプションは、一時オブジェクトの作成が回避される場合でも、一時オブジェクトの作成のためのコピー・コンストラクターへのアクセスを常に検査するかどうかを制御します。

デフォルトで、コンパイラーはアクセス検査を抑止します。これは C++ 標準の拡張で、VisualAge C++ for OS/2 3.0、 VisualAge for C++ for Windows、バージョン 3.5、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

このサブオプションが yes に設定されていると、コードがその拡張子を使用した場合、 **-qsuppress** で警告メッセージを使用禁止にしない限り、警告が出ます。

標準 C++ に準拠する場合は、**-qlanglvl=nooldtempacc** を設定してください。例えば、以下のコードの throw ステートメントは、コピー・コンストラクターがクラス C の protected メンバーであるため、エラーの原因となります。

```
class C {
public:
   C(char *);
protected:
   C(const C&);
};

C foo() {return C("test");} // return copy of C object

void f()
{
// catch and throw both make implicit copies of
// the thrown object
   throw C("error");   // throw a copy of a C object
   const C& r = foo(); // use the copy of a C object
//                               created by foo()
}
```

上記のコード例では、3 個所にコピー・コンストラクター C(const C&) の誤った形式が使用されています。

[no]oldtmplalign

このサブオプションは、バージョン 5.0 より前のバージョンのコンパイラー (**xlC**) でインプリメントされている位置合わせ規則を指定します。これら初期のバージョンの xlC コンパイラーは、ネスト・テンプレート用に指定された位置合わせ規則を無視します。デフォルトでこれらの位置合わせ規則は、VisualAge C++ 4.0 以降では無視されません。例えば、次のテンプレートでは、`A<char>::B` のサイズは **-qlanglvl=nooldtmplalign** の場合は 5、 **-qlanglvl=oldtmplalign** の場合は 8 となります。

```
template <class T>
struct A {
#pragma options align=packed
 struct B {
  T m;
  int m2;
 };
#pragma options align=reset
};
```

[no]oldtmplspec

このサブオプションは、 C++ 標準に準拠していないテンプレートの特殊化を許可するかどうかを制御します。

デフォルトで、コンパイラーはこれらの特殊化を許可します (**-qlanglvl=nooldtmplspec**)。これは標準 C++ の拡張で、VisualAge C++ for OS/2 3.0、 VisualAge for C++ for Windows、バージョン 3.5、および Microsoft Visual C++ によって提供されている C++ コンパイラーとの互換性を持つ振る舞いを提供します。

-qlanglvl=oldtmplspec が設定されていると、コードがその拡張子を使用した場合、**-qsuppress** オプションで警告メッセージを抑止しない限り、警告が出ます。

例えば、以下の行を使用して、char 型のテンプレート・クラス・リボンを明示的に特殊化することができます。

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

標準 C++ に準拠する場合は、**-qlanglvl=nooldtmplspec** を設定してください。上記の例では、テンプレートの特殊化は以下のように変更する必要があります。

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

[no]redefmac

事前の `#undef` または `undefine()` 文なしでマクロを再定義できるかどうかを指定します。

[no]trailenum

このサブオプションは、`enum` 宣言で末尾のコンマを許可するかどうかを制御します。

デフォルトでは、コンパイラーは、列挙子リストの最後に 1 つまたはそれ以上の末尾のコンマを許可します。これは C++ 標準の拡張で、Microsoft Visual C++ との互換性を提供します。以下の `enum` 宣言はこの拡張を使用します。

```
enum grain { wheat, barley, rye,, };
```

標準 C++ 、または VisualAge C++ の以前のバージョンおよび先行の製品でサポートされている **stdc89** 言語レベルとの互換性のために、**-qlanglvl=notrailenum** を設定します。

[no]typedefclass

このサブオプションは、VisualAge C++ の前のバージョンおよび先行プロダクトとの後方互換性を提供します。

現在の C++ 標準は、クラス名を指定するはずの個所に typedef の名前を指定することを許可しません。このオプションは、この制限を緩和します。基本指定子リストおよびコンストラクター初期化指定子リストの typedef 名の使用を許可するには、**-qlanglvl=typedefclass** を設定します。

デフォルトでは、typedef 名はクラス名を指定するはずの個所に指定することはできません。

[no]ucs

このサブオプションは、ユニコード文字が、C++ ソースの 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。デフォルト設定は **-qlanglvl=noucs** です。

ユニコード文字セットは、C++ 標準でサポートされています。この文字セットには、北米および西ヨーロッパのすべての言語を含む、幅広い範囲の言語に使用される文字、数字、およびその他の文字の全セットが含まれています。ユニコード文字は、16 ビットまたは 32 ビットが可能です。 ASCII の 1 バイト文字は、ユニコード文字セットのサブセットです。

-qlanglvl=ucs が使用可能になっている場合は、ソース・ファイルに直接、あるいはエスケープ・シーケンスに似た表記を使用して、ユニコード文字を挿入することができます。ユニコード文字の多くが、画面に表示できない、またはキーボードから入力できないため、通常は、後者の方法が好まれています。ユニコード文字の表記形式は、16 ビット文字の場合は ¥u*hhhh*、32 ビット文字の場合は ¥U*hhhhhhhh* です。ここで、*h* は 16 進数字を表します。文字の短縮 ID は、ISO/IEC 10646 によって指定されます。

[no]varargmacros

この C99 フィーチャーは、C++ アプリケーションでの関数の似たマクロでの変数引数リストの使用を許可します。構文は変数引数関数に似ており、printf のマスキング・マクロとして使用することができます。

例:

```
#define debug(format, ...) printf(format, __VA_ARGS__)

debug("Hello %s¥n", "Chris");
```

次のようにプリプロセスされます:

```
printf("Hello %s¥n", "Chris");
```

置換リストの token __VA_ARGS__ は、パラメーターの省略符号に対応します。省略符号はマクロ呼び出しの可変引数を表します。

varargmacros を指定すると、マクロ __C99_MACRO_WITH_VA_ARGS が 1 の値に定義されます。

[no]zeroextarray

このサブオプションは、ゼロ範囲の配列をクラス定義で最後の非静的データ・メンバーとして許可するかどうかを制御します。

デフォルトでは、コンパイラーはゼロ・エレメントを持つ配列を許可します。これは C++ 標準の拡張で、Microsoft Visual C++ との互換性を提供します。下記の宣言例は、無次元配列 a および b を定義します。

```
struct S1 { char a[0]; };
struct S2 { char b[]; };
```

標準 C++ または VisualAge C++ の前のバージョンおよび先行製品によってサポートされる ANSI 言語レベルとの整合性のために、**nozeroextarray** を設定します。

classic によって示される **stdc89** モードに対する例外は以下のとおりです。

トークン化

マクロ展開により導入されるトークンは、場合によっては隣接するトークンと結合されることがあります。 歴史的には、これは旧式プリプロセッサーのテキスト・ベースのインプリメンテーションの成果物であり、旧式のインプリメンテーションでは、プリプロセッサーは別個のプログラムで、その出力がコンパイラーに渡されていたためです。

同様の理由から、コメントによってのみ分けられたトークンが、 1 つのトークンを形成するように結合されることもあります。 ここでは、 **classic** モードでコンパイルされたプログラムのトークンを行う方法の要約を示します。

1. ソース・ファイルの該当ポイントで、次のトークンが、トークンを形成することのできる可能性のある文字の最長シーケンスです。 例えば、 `i ++ + ++ j` は正しいプログラムになりますが、 `i+++++j` は `i ++ ++ + j` としてトークン化されます。
2. 形成されたトークンが ID およびマクロ名である場合は、そのマクロはその `#define` ディレクティブで指定されたトークンのテキストに置き換えられます。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成体は、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。

以下の構成体があるとします。

```
#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、¥ を使っ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、**classic** モードと **stdc89** モードの間の、プリプロセッサー・ディレクティブの相違点を示します。ここにリストされていないディレクティブは、両方のモードで同じように振る舞います。

#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 > のスペルと、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 ' は、< と > の中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 **classic** モードでは認識されません。

#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。

関連情報
- 209 ページの『-qsuppress』
- コマンド行オプションの要約: 標準の準拠
- 「*XL C/C++ 言語解説書*」の『IBM XL C 言語拡張』および『IBM XL C++ 言語拡張』

-qlib

説明

リンク時に標準システム・ライブラリーを使用するようコンパイラーに命令します。

構文

注

-qnolib コンパイラー・オプションが指定されている場合、標準システム・ライブラリーは使用されません。コマンド行に明示的に指定されたライブラリーだけがリンク時に使用されます。

関連情報
- 79 ページの『-qcrt』
- リンクを制御するオプション: リンカー入力制御のオプション

-qlibansi

説明

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。

構文

278 ページの『#pragma options』も参照してください。

注

これにより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。

関連情報

- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qlinedebug

説明

デバッガー用に行番号およびソース・ファイル名の情報を生成する。

構文

注

このオプション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。

このオプションは、**-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。

-qlinedebug オプションを指定した場合は、インライン・オプションがデフォルトで **-Q!** (関数をインライン化しない) になります。

-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 -qnolinedebug** を指定した場合は、**-qnolinedebug** が無視され、以下の警告が出されます。

```
1506-... (W) Option -qnolinedebug is incompatible with option -g and is ignored
```

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

- 278 ページの『#pragma options』

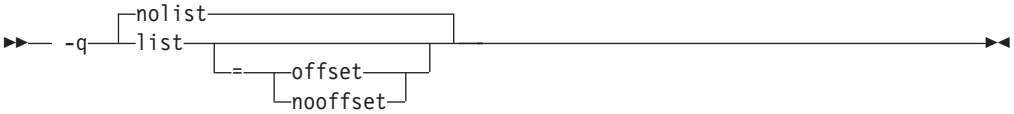
- 104 ページの『-g』
- 166 ページの『-O、-qoptimize』
- 185 ページの『-Q』
- エラー検査とデバッグのオプション: デバッグのオプション

-qlist

説明

オブジェクト・リストを含むコンパイラー・リストを生成する。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・リストを使用することができます。

構文



-qlist=offset を指定すると、`.lst` ファイルの命令のリストがプロシージャーの始めからのオフセットに変更されます。

注

-qlist=offset 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。

-qlist=offset を指定すると、PDEF ヘッダーのオフセットは `00000` ではなくなり、テキスト領域のはじめからのオフセットが含まれるようになります。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、**-qlist=offset** または **-qlist=nooffset** のどちらが指定されていても同じ値になります。

-qlist を指定すると、**-qlist=nooffset** が暗黙指定されます。

-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。

例

`myprogram.C` をコンパイルしてオブジェクト・リストを作成するには、以下のように入力します。

```
xlc++ myprogram.C -qlist
```

関連情報

- 278 ページの『#pragma options』
- 181 ページの『-qprint』
- リストとメッセージを制御するオプション: リストのオプション

-qlistopt

説明

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示すコンパイラー・リストを作成する。

構文

注

リストにはコンパイラー・デフォルト、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ている有効なオプションが表示されます。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・リストには表示されません。

-qnoprint を指定すると、このコンパイラー・オプションがオーバーライドされます。

例

`myprogram.C` をコンパイルして、有効なすべてのオプションを表示するコンパイラー・リストを作成するには、以下のように入力します。

```
xlc++ myprogram.C -qlistopt
```

関連情報

- 181 ページの『-qprint』
- 23 ページの『矛盾するコンパイラー・オプションの解決』
- リストとメッセージを制御するオプション: リストのオプション

-qlonglit

説明

サフィックスなしのリテラルを 64 ビット・モードの `long` 型にする。

構文

注

C

下記の表は、 **stdc89**、**extc89**、または **extended** 言語レベルでコンパイルするときの 64 ビット・モードでの定数の暗黙の型を示したものです。

	デフォルトの 64 ビット・モード	qlonglit での 64 ビット・モード
サフィックスなしの 10 進数	signed int signed long unsigned long	signed long unsigned long
サフィックスなしの 8 進数または 16 進数	signed int unsigned int signed long unsigned long	signed long unsigned long
サフィックス u/U 付き	unsigned int unsigned long	unsigned long
サフィックス l/L 付き	signed long unsigned long	signed long unsigned long
サフィックス ul/UL 付き	unsigned long	unsigned long

C++

下記の表は、**stdc99**、**extc99**、または **extended** 言語レベルでコンパイルするときの 64 ビット・モードでの定数の暗黙の型を示したものです。

	10 進定数	10 進定数への -qlonglit の効果
サフィックスなしの場合	int long int	long int
u または **U**	unsigned int unsigned long int	unsigned long int
l または **L**	long int	long int
u または **U**、および **l** または **L** の両方	unsigned long int	unsigned long int
ll または **LL**	long long int	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

	8 進数または 16 進定数	8 進数または 16 進定数への -qlonglit の効果
サフィックスなしの場合	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
l または **L**	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
ll または **LL**	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

関連情報
- 137 ページの『-qlanglvl』
- 整数および浮動小数点処理を制御するオプション

-qlonglong

説明
プログラムで long long 整数型を許可する。

構文


デフォルト
xlc、**xlC**、および **cc** のデフォルトは **-qlonglong** で、_LONG_LONG (long long 型がプログラムで機能する) を定義します。c89 のデフォルトは **-qnolonglong** です (long long 型はサポートされません)。

注
C このオプションは、選択した言語レベルが **stdc99** または **extc99** のときには指定できません。これは、C89 標準の拡張として提供されている long long のサポートを制御するために使用します。この拡張は、C99 標準の一部である long long のサポートとは若干異なります。

例
1. long long 整数を許可しないように 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 -qnolonglong
```

関連情報
- 整数および浮動小数点処理を制御するオプション

-M

説明
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。

構文


注
-M オプションは、 **-qmakedep** オプションと機能的に同じです。

.d ファイルは **make** ファイルではありません。.d ファイルは、**make** コマンドに使用する前に編集する必要があります。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:*file_name*.c
file_name.o:*include_file_name*

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、 26 ページの『相対パス名を使用した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。組み込みファイルは検出されない場合、.d ファイルに追加されません。

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。

例

-o オプションを指定しない場合、 **-M** によって生成される出力ファイルは、現行ディレクトリーに作成されます。これには .d サフィックスが付きます。例えば、以下のコマンドでは、

```
xlc -M person_years.c
```

出力ファイル person_years.d が生成されます。

.d ファイルは、.c、.C、.cpp、または .i サフィックスを持つすべての入力ファイルに対して作成されます。また **-+** コンパイラー・オプションを有効にして C++ プログラムをコンパイルした場合は、どのファイル・サフィックスも受け入れられ、.d ファイルが作成されます。そうでない場合は、出力 .d ファイルは他のどのファイルについても作成されません。

例えば、以下のコマンドでは、

```
xlc -M conversion.c filter.c /lib/libm.a
```

2 つの出力ファイル conversion.d と filter.d のほか、実行可能ファイルも作成されます。ライブラリーについては、.d ファイルは作成されません。

現行ディレクトリーが書き込み可能でない場合は、.d ファイルは作成されません。**-o** *file_name* を **-M** と一緒に指定すると、 .d ファイルが **-o** *file_name* によって暗黙指定されたディレクトリーに入れられます。例えば、以下の呼び出しでは、

```
xlc -M -c t.c -o /tmp/t.o
```

.d 出力ファイルが /tmp/t.d に入れられます。

関連情報

- 160 ページの『-qmakedep』
- -MF
- 50 ページの『-+ (正符号)』
- 170 ページの『-o』
- 201 ページの『-qsourcetype』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 出力を制御するオプション: ファイル出力のオプション

-ma

C

説明

組み込み関数 alloca への呼び出しのインライン・コードを置換する。

構文

注

#pragma alloca が指定解除されている場合や **-ma** を使用していない場合は、alloca は組み込み関数ではなく、ユーザー定義 ID として扱われます。

このオプションは、C++ プログラムには適用されません。 C++ プログラムでは、alloca 関数宣言を組み込むには、代わりにヘッダー malloc.h を組み込まなければなりません。

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 -ma
```

関連情報

- 56 ページの『-qalloca』
- 247 ページの『#pragma alloca』
- 出力を制御するオプション: その他の出力オプション

-MF

説明

-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。

構文

file はターゲット出力パスで、これはファイルまたはディレクトリーになります。

例

表 38.

コマンド行	生成される従属ファイル
xlc -c -qmakedep mysource.c	mysource.d
xlc -c -qmakedep foo_src.c -MF mysource.d	mysource.d
xlc -c -qmakedep foo_src.c -MF ../deps/mysource.d	../deps/mysource.d

表 38. (続き)

コマンド行	生成される従属ファイル
xlc -c -qmakedep foo_src.c -MF /tmp/mysource.d	/tmp/mysource.d
xlc -c -qmakedep foo_src.c -o foo_obj.o	foo_obj.d
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.d	mysource.d
xlc -c -qmakedep foo_src.c -MF mysource1.d -MF mysource2.d	mysource2.d
xlc -c -qmakedep foo_src1.c foo_src2.c -MF mysource.d	mysource.d (これには foo_src2.d source ファイルの規則が含まれます)
xlc -c -qmakedep foo_src1.c foo_src2.c -MF /tmp	/tmp/foo_src1.d /tmp/foo_src2.d

注

-MF は、**-qmakedep** または **-M** オプションと共に指定された場合にのみ効果を持ちます。

file がディレクトリーの名前の場合、コンパイラーによって生成される従属ファイルは指定されたディレクトリーに入れられます。それ以外の場合は、*file* にパスを指定しないと、従属ファイルは現行作業ディレクトリーに保管されます。

-MF オプションによって指定されたファイルがすでに存在する場合は、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** オプションを指定すると、単一の従属ファイルだけが生成され、それにはコマンド行に指定された最後のファイルの **make** 規則が含まれます。

関連情報

- 157 ページの『-M』
- 『-qmakedep』
- 170 ページの『-o』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 出力を制御するオプション: ファイル出力のオプション

-qmakedep

説明

コンパイルにおいてメイン・ソース・ファイルの依存関係を記述するための **make** コマンドの記述ファイルへ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。 **gcc** サブオプションが指定されている場合、記述ファイルにはすべての依存関係をリストした単一のターゲットが組み込まれます。そうでない場合は、記述ファイル内にそれぞれの依存関係ごとに別々の規則があります。

構文

gcc サブオプションは、GNU C/C++ フォーマットに一致するように、**make** 規則のフォーマットを制御します。

注

無効なサブオプションを指定すると、警告メッセージが発行され、オプションは無視されます。

-qmakedep をサブオプションなしで指定すると、機能的に **-M** オプションを指定することと同等です。

.d ファイルは **make** ファイルではありません。.d ファイルは、**make** コマンドに使用する前に編集する必要があります。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。

-o オプションを指定しない場合は、 **-qmakedep** オプションで生成される出力ファイルは、現行ディレクトリーに作成されます。これには .d サフィックスが付きます。例えば、以下のコマンドでは、

```
xlc++ -qmakedep person_years.C
```

出力ファイル `person_years.d` が生成されます。

.d ファイルは、.c、.C、.cpp、または .i サフィックスを持つすべての入力ファイルに対して作成されます。また **-+** コンパイラー・オプションを有効にして C++ プログラムをコンパイルした場合は、どのファイル・サフィックスも受け入れられ、.d ファイルが作成されます。そうでない場合は、出力 .d ファイルは他のどのファイルについても作成されません。

例えば、以下のコマンドでは、

```
xlc++ -qmakedep conversion.C filter.C /lib/libm.a
```

2 つの出力ファイル、`conversion.d` と `filter.d` (および実行可能ファイルも) 作成されます。ライブラリーについては、.d ファイルは作成されません。

現行ディレクトリーが書き込み可能でない場合は、.d ファイルは作成されません。**-o** *file_name* を **-qmakedep** と共に指定すると、.d ファイルが、**-o***file_name* によって暗黙指定されたディレクトリーに入れられます。例えば、以下の呼び出しでは、

```
xlc++ -qmakedep -c t.C -o /tmp/t.o
```

.d 出力ファイルが `/tmp/t.d` に入れられます。

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*.C

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、 26 ページの『相対パス名を使用した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。組み込みファイルは検出されない場合、.d ファイルに追加されません。

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。

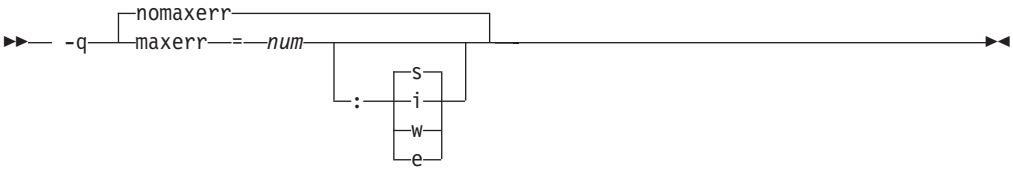
関連情報
- 157 ページの『-M』
- 159 ページの『-MF』
- 170 ページの『-o』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 出力を制御するオプション: ファイル出力のオプション

-qmaxerr

説明

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エラーの件数が *num* に達したときにコンパイルを停止するようコンパイラーに命令する。

構文



ここで、*num* は整数でなければなりません。重大度レベルは、以下のいずれかになります。

sev_level	説明
i	通知
w	警告
e	エラー (C のみ)
s	重大エラー

注

重大度レベルを指定していない場合は、**-qhalt** オプションの現行値が使用されます。

-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。

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ます。出されるエラー・メッセージは以下のようなものです。

```
1506-672 (U) The number of errors has reached the limit of ...
```

-qnomaxerr を指定すると、検出されたエラーの数に関係なく、ソース・ファイル全体がコンパイルされます。

診断メッセージは、**-qflag** オプションで制御することができます。

例

1. 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
   ```

2. 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=5
   ```

3. 3 件の通知メッセージを検出したら 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=3:i
   ```

   または

   ```
   xlc myprogram.c -qmaxerr=3 -qhalt=i
   ```

関連情報

- 95 ページの『-qflag』
- 107 ページの『-qhalt』
- 31 ページの『メッセージ重大度レベルとコンパイラー応答』
- エラー検査とデバッグのオプション: エラー検査のオプション

-qmaxmem

説明

メモリーを大量に消費する特定の最適化のローカル・テーブルのための最適化プログラムによって使用されるメモリーの量を制限する。メモリー・サイズの制限はキロバイト単位で指定されます。

構文

デフォルト

- -O2 最適化が有効な場合、maxmem=8192。
- -O3 以上の最適化が有効な場合、maxmem=-1。

注

- *size* の値を -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。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、この量が使用可能なシステム・リソースを超えてしまう場合があります。

- **-qmaxmem** によって設定される制限は、コンパイラー全体ではなく、特定の最適化のためのメモリーの量です。 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。
- 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。
- 最適化の範囲を制限しても、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ると、ページ・スペースがすべて使用されてしまう可能性があります。 特に、**-qmaxmem=-1** を指定すると、コンパイラーがストレージを無制限に使用できるようになるので、最悪の場合には、最も優れた装備を持つマシンのリソースでさえも使用し尽くしてしまうおそれがあります。

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
```

関連情報

- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qmbcs、-qdbcs

説明

プログラムにマルチバイト文字が含まれる場合に、**-qmbcs** オプションを使用する。**-qmbcs** オプションは **-qdbcs** と同等です。

構文

278 ページの『#pragma options』も参照してください。

注

マルチバイト文字は、中国語、日本語、韓国語などの特定の言語で使用されます。

-qmbcs または **-qdbcs** コンパイラー・オプションを指定すると、マルチバイト文字がコメントでも許可されます。

ソース・ファイルにマルチバイト文字リテラルが含まれており、デフォルトの **-qnombcs** または **-qnodbcs** コンパイラー・オプションが有効な場合、コンパイラーはすべてのリテラルを単一バイト・リテラルとして扱います。

例

`myprogram.c` にマルチバイト文字が含まれている場合にこのファイル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
- 入力を制御するオプション: その他の入力オプション

-qminimaltoc

説明

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 を入れることによって、 64 ビット・コンパイルでの toc オーバーフローを阻止する。

構文

注

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。

64 ビット・モードでコンパイルされたプログラムは、toc 入力が 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。**-qminimaltoc** オプションでコンパイルすることにより、このような toc オーバーフロー・エラーを避けることができます。

-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。

関連情報
- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qmkshrobj

説明

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。

構文

注

このオプションは下記の関連オプションと共に、共用オブジェクトの作成に使用されます。このオプションを使用する利点は、コンパイラーが自動的に tempinc ディレクトリー内のテンプレートのインスタンス化を組み込んでコンパイルすることです。

-qmkshrobj を指定すると、**-qpic** が暗黙指定されます。

また、以下の関連オプションは、**-qmkshrobj** コンパイラー・オプションと共に使用することができます。

-o *shared_file*　　共用ファイルの情報を保持するファイルの名前。デフォルトは a.outです。

-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。

-qmkshrobj を使用して共用ライブラリーを作成すると、コンパイラーとリンケージ・エディターが適切なオプションで呼び出されて、共用オブジェクトを構築します。

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.o を構成するには、以下のように入力します。

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

関連情報

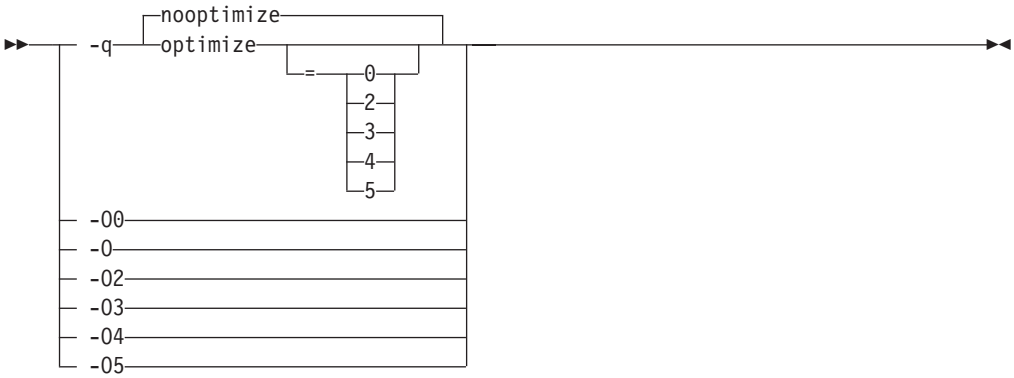
- 181 ページの『-qpriority』
- 286 ページの『#pragma priority』
- 179 ページの『-qpic』
- リンクを制御するオプション: リンカー出力制御のオプション
- 「*XL C/C++ プログラミング・ガイド*」の『ライブラリーの組み立て』

-O、-qoptimize

説明

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、最適化のレベルを指定する。

構文



最適化の設定は以下のとおりです。

-O0 -qnooptimize -qoptimize=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 -qoptimize	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。
-O2 -qoptimize=2	**-O** と同じ。
-O3 -qoptimize=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
-qoptimize=3 (*続き*)
	-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. XL C/C++ のバージョン 8 以降、**-O3** を指定すると、明示的に **-qhot** または **-qhot=level=1** オプションを指定しない限り、**-qhot=level=0** が暗黙指定されます。

-O3、-qoptimize=3 (続き)	注 • **-qfloat=rsqrt** は、**-O3** ではデフォルトで設定されます。 • **-qmaxmem=1** は、 **-O3** ではデフォルトで設定され、最適化を実行するときにコンパイラーが必要なだけ多くのメモリーを使用できるようにします。 • 組み込み関数は、-O3 では **errno** を変更しません。 • 整数除算命令の最適化は、 **-O3** であっても非常に危険であると見なされます。 • デフォルトの **-qmaxmem** 値は、 **-O3** では **-1** です。 • **optimize** オプションを **-qflttrap** オプションと一緒に指定したときのコンパイラーの振る舞いについては、100 ページの『-qflttrap』を参照してください。 • **-qstrict** および **-qstrict_induction** コンパイラー・オプションを使用して、プログラムのセマンティクスを変更する可能性がある **-O3** の影響をオフにすることができます。**-qstrict** を **-O3** と一緒に指定すると、**-O2** で実行されるすべての最適化のほか、ループの最適化も呼び出されます。**-qstrict** コンパイラー・オプションへの参照は、**-O3** オプションの前に指定することも、後に指定することもできます。 • **-O3** コンパイラー・オプションの後に **-O** オプションを指定すると、 **-qignerrno** がオンのままになります。 • **-O3** と **-qhot=level=1** が有効な場合、コンパイラーはソース・コード内のすべての呼び出しを、同等の MASS ライブラリー関数、および可能であれば、ベクトル・バージョンへの呼び出しを持つ標準の数学ライブラリー関数に置換します。
-O4 -qoptimize=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。 **Note: -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 -qoptimize=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。

注

-qoptimize... は、**-qopt...** に省略できます。例えば、**-qnoopt** は **-qnooptimize** と同等です。

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。

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

- パフォーマンス最適化のオプション: 定義済み最適化レベルのオプション
- 「*XL C/C++ プログラミング・ガイド*」の『アプリケーションの最適化』

-o

説明

コンパイラーによって作成されるオブジェクト・ファイル、アセンブラー・ファイル、または実行可能ファイルの出力位置を指定する。コンパイラー呼び出し中に **-o** オプションを使用するときには、*filespec* をファイルかディレクトリーのいずれかの名前にすることができます。リンケージ・エディターの直接呼び出し中に **-o** オプションを使用するときには、*filespec* はファイルの名前にしかすることができません。

構文

注

-o をコンパイラー呼び出しの一部として指定するときには、*filespec* をディレクトリーまたはファイルの相対パス名か絶対パス名にすることができます。

1. *filespec* が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。
2. *filespec* という名前のディレクトリーが存在しない場合は、**-o** オプションは、コンパイラーによって作成されるファイルの名前を *filespec* と指定します。 例えば、以下のコンパイラー呼び出しでは、

   ```
   xlc test.c -c -o new.o
   ```

   test.o の代わりにオブジェクト・ファイル new.o が作成されます。そして、以下のコンパイラー呼び出しでは、

   ```
   xlc test.c -o new
   ```

   new という同じ名前を持つディレクトリーが存在しなければ、a.out の代わりにオブジェクト・ファイル new が作成されます。存在する場合は、デフォルトのオブジェクト名 a.out が使用されて、new ディレクトリーに入れられます。

   C または C++ のソース・ファイル・サフィックス (.C、.c、.cpp、または .i) を持つ *filespec* (myprog.c や myprog.i など) はエラーの結果となり、コンパイラーもリンケージ・エディターも呼び出されません。

   -c と **-o** を一緒に使用して、 *filespec* がディレクトリーを指定していない場合は、一度に 1 つのソース・ファイルしかコンパイルすることはできません。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***filename* オプションをオーバーライドします。

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て、myprogram.c をコンパイルした結果、myaccount という名前の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 -o myaccount
```

ディレクトリー myaccount が存在する場合は、コンパイラーは実行可能ファイル a.out を作成して、それを myaccount ディレクトリーに入れます。

関連情報
- 66 ページの『-c』
- 86 ページの『-E』
- 『-P』
- 211 ページの『-qsyntaxonly』
- 出力を制御するオプション: ファイル出力のオプション

-P

説明

コンパイラー呼び出しに指定された C または C++ ソース・ファイルをプリプロセスし、プリプロセスされた出力ソース・ファイル *file_name*.i をそれぞれの入力ソース・ファイル *file_name*.c、*file_name*.C、または *file_name*.cpp ごとに作成する。デフォルトでは、 C または C++ のソース・ファイルがコンパイルおよびリンク・エディットされて、実行可能ファイルが作成されます。

構文

注

-P オプションはプリプロセッサーを直接呼び出します。

-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。

#line ディレクティブは出されません。

-P オプションは、*file_name***.i** など、プリプロセスされたソース・ファイルを入力として受け入れることはできません。コンパイラーはエラー・メッセージを発行します。

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。

拡張モードでは、プリプロセッサーは以下の箇所で円記号 (¥) 後に改行文字が続いている場合、その円記号 (¥) を行の継続と解釈します。

- マクロ置き換えテキスト
- マクロ引数
- プリプロセッサー・ディレクティブと同じ行にあるコメント

これ以外の箇所での行の継続は、**ANSI** モードでなければ処理されません。

-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。 **-C** オプションは、 **-E** および **-P** オプションの両方と一緒に使用することができます。

関連情報
- 66 ページの『-C』
- 66 ページの『-c』
- 86 ページの『-E』
- 170 ページの『-o』
- 211 ページの『-qsyntaxonly』
- 出力を制御するオプション: ファイル出力のオプション

-p

説明
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設定する。

構文

注
-qtbtable オプションを設定しない場合は、 **-p** オプションによって完全なトレースバック・テーブルが生成されます。

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 **-p** オプションを指定しなければなりません。

例
オペレーティング・システムの **gprof** コマンドと共に使用できるように `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 -p
```

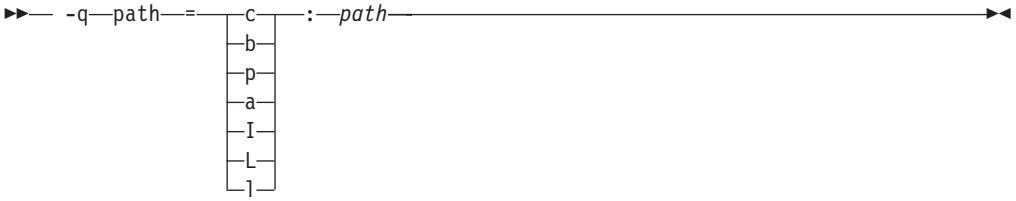
関連情報
- 213 ページの『-qtbtable』
- エラー検査とデバッグのオプション: プロファイルのオプション

-qpath

説明

コンパイラー・コンポーネントの代替プログラム名を構成する。このオプションによって指定されたプログラムおよびディレクトリー・パス は、通常のコンパイラー・コンポーネントまたはプログラムの代わりに使用されます。

構文



ここで、使用可能なコンパイラー・コンポーネントおよびプログラム名は以下の通りです。

プログラム	説明
c	コンパイラーのフロントエンド
b	コンパイラーのバックエンド
p	コンパイラー・プリプロセッサー
a	アセンブラー
I	プロシージャー間分析 (コンパイル・フェーズ)
L	プロシージャー間分析 (リンク・フェーズ)
l	リンケージ・エディター

注

-qpath オプションは、 **-F***config_file*、 **-t**、および **-B** オプションをオーバーライドします。

例

代替 **xlc++** コンパイラーを使用して `/lib/tmp/mine/` で `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 -qpath=c:/lib/tmp/mine/
```

代替リンカーを使用して `/lib/tmp/mine/` で `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 -qpath=l:/lib/tmp/mine/
```

関連情報

- 64 ページの『-B』
- 95 ページの『-F』
- 211 ページの『-t』
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qpdf1、-qpdf2

説明

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。

構文

注

PDF を使用するには、以下のステップに従ってください。

1. **-qpdf1** オプションを指定して、プログラム内の一部またはすべてのソース・ファイルをコンパイルする。少なくとも **-O2** 最適化オプションを指定する必要があり、また少なくとも有効な **-O2** にリンクする必要があります。後で同じオプションを使用する必要があるため、ファイルのコンパイルに使用するコンパイラー・オプションには特に注意してください。

   大きいアプリケーションでは、最適化によって得られる利益が最も大きいコードの領域に集中してください。アプリケーションのコードのすべてを **-qpdf1** オプションでコンパイルする必要はありません。
2. 一般的なデータ・セットを使用してプログラムを一通り実行する。プログラムは、終了時にプロファイル情報を記録します。 プログラムは、異なるデータ・セットで複数回実行することができます。また、プロファイル情報が累積されるため、分岐の頻度およびコードのブロックの実行頻度が正確にカウントされます。

   重要: 完了したプログラムの通常の実行中に使用されるデータのうち、代表的なデータを使用してください。
3. 前と同じコンパイラー・オプションを使用してプログラムを再リンクする。ただし、**-qpdf1** は **-qpdf2** に変更してください。 **-L**、**-l**、およびその他いくつかのオプションはリンカー・オプションであり、それらはこの時点で変更できます。この 2 番目のコンパイルでは、累積されたプロファイル情報を使用して最適化が微調整されます。結果として得られるプログラムにはプロファイルのオーバーヘッドが含まれないため、フルスピードで実行されます。

中間ステップとして、**-qpdf2** を使用して、 **-qpdf1** パスによって作成されたオブジェクト・ファイルを、 **-qpdf2** パスでソースを再コンパイルせずにリンクすることができます。これによりかなりの時間を節約し、大きなアプリケーションを最適化のために微調整することができます。 **-qpdf2** パスで異なるオプションを使用すると、PDF 最適化バイナリーの異なるフレーバーを作成してテストすることができます。

最高のパフォーマンスを得るには、PDF を使用するときに、すべてのコンパイルで **-O3**、**-O4**、または **-O5** オプションを使用します。

プロファイルは、現行作業ディレクトリーに配置されるか、PDFDIR 環境変数が設定されている場合はその変数によって指定されたディレクトリーに配置されます。

コンパイルと実行の時間を節約するために、PDFDIR 環境変数を必ず絶対パスに設定してください。そうしないと、誤ったディレクトリーからアプリケーションが実行され、プロファイル・データ・ファイルが見つからなくなる可能性があります。これが起こると、プログラムが正しく最適化されないか、またはセグメンテーション障害によって停止する場合があります。 PDF プロセスの終了前に PDFDIR 変数の値を変更してアプリケーションを実行した場合にも、セグメンテーション障害が起こる場合があります。

このオプションはアプリケーション全体を 2 回コンパイルすることを要求するため、他のデバッグとチューニングの終了後、アプリケーションを実稼働環境に入れる前の最終ステップの 1 つとして使用するよう意図されています。

制約事項

- PDF 最適化には、少なくとも **-O2** の最適化レベルが必要です。
- 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。
- プロファイル情報のセットを区別するために **-qipa=pdfname** サブオプションを使用していない限り、同じ PDFDIR ディレクトリーを使用する 2 つの異なるアプリケーションを同時にコンパイルまたは実行しないでください。
- 特定のプログラムのすべてのコンパイル・ステップで、同じコンパイラー・オプションのセットを使用しなければなりません。そうしないと、PDF がプログラムを正しく最適化できないだけでなく、プログラムの速度が遅くなる場合もあります。構成ファイルによって提供されるすべての設定も含めて、コンパイラー設定はすべて同じでなければなりません。
- 現行バージョン・レベルの XL C/C++ によって作成された PDF ファイルを他のバージョン・レベルのコンパイラーによって作成された PDF ファイルと混合しないでください。
- **-qipa** が直接あるいは別のオプションを介して呼び出されなかった場合、**-qpdf1** と **-qpdf2** が **-qipa=level=0** オプションを呼び出します。
- **-qpdf1** を指定してプログラムをコンパイルする場合は、実行するときにプロファイル情報が生成されるため、パフォーマンス上のオーバーヘッドがいくらか発生することを記憶しておいてください。このオーバーヘッドは、**-qpdf2** を指定して再コンパイルするか、PDF なしで再コンパイルするとなくなります。

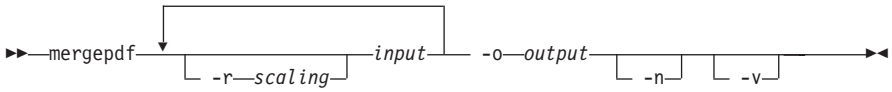
以下のユーティリティー・プログラム (`/opt/ibmcmp/vacpp/8.0/bin/` にある) は、PDFDIR ディレクトリーの管理に使用できます。

cleanpdf

pathname ディレクトリーからすべてのプロファイル情報を除去します。または、*pathname* が指定されていない場合は PDFDIR ディレクトリーから除去し、PDFDIR が設定されていない場合は現行ディレクトリーから除去します。プログラムを変更して PDF プロセスをもう一度実行する場合、プロファイル情報を除去するとランタイム・オーバーヘッドが減ります。

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、内部の位取りの比率とユーザー指定の位取りの比率を画面に表示します。

resetpdf

前述の **cleanpdf** と同じ。

showpdf

プログラム実行で実行されたすべてのプロシージャーの呼び出し数と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。

例

簡単な例を以下に示します。

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

/* 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
```

以下は、**-qpdf1** および **-qpdf2** オブジェクトの使用例です。

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

/* 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
```

関連情報

- 196 ページの『-qshowpdf』
- 122 ページの『-qipa』
- パフォーマンス最適化のオプション: パフォーマンス・データ割り振りのオプション

-pg

説明

プロファイル用にオブジェクト・ファイルを設定する。

-qtbtable オプションを設定しない場合は、 **-pg** オプションによって完全なトレースバック・テーブルが生成されます。

構文

例

ご使用のオペレーティング・システムの **gprof** コマンドと共に使用するように 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 -pg
```

コンパイルおよび リンクするには、必ず **-pg** オプションを指定してください。例を以下に示します。

```
xlc myprogram.c -pg -c
xlc myprogram.o -pg -o program
```

関連情報

- 213 ページの『-qtbtable』
- エラー検査とデバッグのオプション: プロファイルのオプション

-qphsinfo

説明

各コンパイル・フェーズでかかった時間を報告する。フェーズ情報は標準出力に送られます。

構文

注

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。

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。

```
Front End - Phase Ends;   0.004/  0.005
W-TRANS   - Phase Ends;   0.010/  0.010
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
```

-O4 を使用して同じプログラムをコンパイルすると、以下のようになります。

```
Front End - Phase Ends;   0.004/  0.006
IPA       - Phase Ends;   0.040/  0.040
IPA       - Phase Ends;   0.220/  0.280
W-TRANS   - Phase Ends;   0.030/  0.110
OPTIMIZ   - Phase Ends;   0.030/  0.030
REGALLO   - Phase Ends;   0.010/  0.050
AS        - Phase Ends;   0.000/  0.000
```

関連情報

- リストとメッセージを制御するオプション: メッセージのオプション

-qpic

説明

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ようにコンパイラーに命令する。

構文

ここで、

nopic　Position Independant Code を生成しないようコンパイラーに命令します。

pic　Position Independant Code を生成するようコンパイラーに命令します。

small　グローバル・オフセット・テーブルのサイズが 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 よりも大きいグローバル・オフセット・テーブルを許可します。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。

注

-qpic がサブオプションなしで指定された場合、**-qpic=small** が想定されます。

-qmkshrobj コンパイラー・オプションが指定されると、**-qpic** オプションが暗黙指定されます。

-q64 を指定すると、自動的に **-qpic** が暗黙指定されます。

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 -Wl, -shared, -soname="libmylib.so.1" -o libmylib.so.1
```

-shared と **-soname** オプションについては詳しくは、ご使用のオペレーティング・システムの資料で **ld** コマンドを参照してください。

関連情報

- 51 ページの『-q32、-q64』
- 165 ページの『-qmkshrobj』
- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-qppline

説明

プリプロセスされた出力で #line ディレクティブの生成を使用可能にする。

構文

注

-P オプションが使用された場合、デフォルトは **-qnoppline** です。

このオプションは **-E** および **-P** オプションの振る舞いをオーバーライドします。

例

-qppline オプションを使用して myprogram.C をコンパイルするには、以下のように入力します。

```
xlc++ myprogram.C -qppline
```

関連情報

- 出力を制御するオプション: ファイル出力のオプション

-qprefetch

説明

コンパイルされたコード内の dcbt や dcbz など、プリフェッチ命令の生成を使用可能にする。

構文

注

デフォルトで、コンパイラーはコンパイルされたコードにプリフェッチ命令を挿入する可能性があります。 **-qnoprefetch** オプションを使用すると、この機能を使用不可にすることができます。

-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。

関連情報

- パフォーマンス最適化のオプション: 最適化を制限するオプション

-qprint

説明

リストを使用可能にするか、または抑制する。 **-qnoprint** を指定すると、すべてのリスト作成オプションが、指定された位置に関係なくオーバーライドされ、リストが抑制されます。

構文

注

デフォルトの **-qprint** では、他のコンパイラー・オプションによって要求された場合にリストが使用可能になります。これらのオプションは以下のとおりです。

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

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
```

関連情報

- リストとメッセージを制御するオプション: リストのオプション

-qpriority

C++

説明

静的オブジェクトの初期化の優先順位を指定する。

構文

ここで、

number ファイル内の静的オブジェクトに割り当てられた初期化の優先順位、あるいは共用または非共用のファイルやライブラリーの優先順位です。

優先順位には、101 (最高の優先順位) から 65535 (最低の優先順位) を指定することができます。

指定されていない場合、デフォルトの優先順位は 65535 になります。

286 ページの『#pragma priority』および 278 ページの『#pragma options』も参照してください。

例

ファイル myprogram.C をコンパイルしてオブジェクト・ファイル myprogram.o を作成し、そのファイル内のオブジェクトの初期化の優先順位を 2000 にするには、次のように入力します。

```
xlc++ myprogram.C -c -qpriority=2000
```

異なる優先順位レベルを指定する **#pragma priority(***number***)** ディレクティブがソース・ファイルに含まれていない場合は、結果として得られるオブジェクト・ファイル内のすべてのオブジェクトの初期化の優先順位に 2000 が指定されます。

関連情報

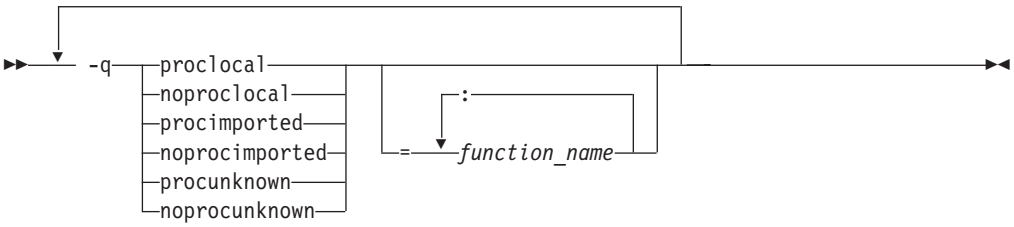
- リンクを制御するオプション: その他のリンカー・オプション

-qproclocal、-qprocimported、-qprocunknown

説明

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。

構文



278 ページの『#pragma options』も参照してください。

デフォルト

デフォルトでは、定義が現行コンパイル単位内にある関数はすべてローカル **proclocal** であり、その他の関数はすべて不明 **procunknown** であると想定されます。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、リンケージ・エディターは、エラーを検出して警告を出します。

注

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。

使用できるサブオプションは以下のとおりです。

ローカル関数　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。 **-qproclocal** を指定すると、すべての関数をローカルであると見なすようにデフォルトが変更されます。**-qproclocal=***names* は、指定された関数をローカルとしてマークします。ここで、*names* は、コロン (**:**) で区切った関数 ID のリストです。デフォルトは変更されません。

ローカルとしてマークされた関数への呼び出しに対しては、サイズがより小さくて高速なコードが生成されます。

インポートされる関数　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。 **-qprocimported** は、すべての関数がインポートされるものであると見なすようにデフォルトを変更します。 **-qprocimported=***names* を指定すると、指定された関数がインポートされるものとしてマークされます。ここで、*names* は、コロン (**:**) で区切った関数 ID のリストです。デフォルトは変更されません。

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。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。

不明の関数　不明の関数は、リンク・エディット中に、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。 **-qprocunknown** を指定すると、すべての関数を不明であると見なすようにデフォルトが変更されます。**-qprocunknown=***names* は、指定された関数を不明としてマークします。ここで、*names* は、コロン (**:**) で区切った関数 ID のリストです。デフォルトは変更されません。

C++ C++ プログラムでは、関数 *names* はそれらのマングル名を使用して指定する必要があります。

プロシージャーをマークするオプションの間で矛盾がある場合は、以下の方法で解決されます。

関数名をリストするオプション	特定の関数名に対する最後の明示的指定が使用される。
デフォルトを変更するオプション	この形式は、名前リストを指定しない。最後に指定されたオプションが、名前リスト・フォームに明示的にリストされていない関数に対するデフォルトとなります。

例

1. `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。さらに、

- 関数 venus を**不明**と指定する。

以下のように入力します。

```
xlc++ myprogram.c oldprogs.a -qprolocal=fun(int):sun()
  -qprocimported=moon():stars(float) -qprocunknown=venus()
```

2. 次の例は、ローカルとしてマークされた関数が、ローカルではなく共用ライブラリー関数に解決されたときに、結果として表示される一般的なエラー・メッセージです。

```
int main(void)
{
    printf("Just in function foo1()¥n");
    printf("Just in function foo1()¥n");
}
```

xlc -q64 -qproclocal -O -qlist t.c を指定してこのソース・コードをコンパイルすると、以下のような結果になります。

```
/usr/lib64: t.o(.text+0x10): unresolvable relocation ¥
  against symbol `.printf@@GLIBC_2.2.5'
    t.o: In function .main':
    t.o(.text+0x10): relocation truncated to fit: R_PPC64_REL24 .printf@@GLIBC_2.2.5
/usr/lib64: t.o(.text+0x18): unresolvable relocation ¥
  against symbol `.printf@@GLIBC_2.2.5'
    t.o(.text+0x18): relocation truncated to fit: R_PPC64_REL24 .printf@@GLIBC_2.2.5
```

実行可能ファイルは作成されますが、実行されません。エラー・メッセージには、オブジェクト・ファイル t.o 内の printf の呼び出しで問題が発生したことが示されます。呼び出されるルーチンが共用オブジェクトからインポートされるものであることを確認したら、警告の原因となったソース・ファイルを再コンパイルして、インポートされるものとして printf を明示的にマークしてください。例を以下に示します。

```
xlc -c -qprocimported=printf t.c
```

関連情報

- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qproto

C

説明

このオプションが設定されると、コンパイラーは、すべての関数がプロトタイプ化されているものと想定する。

構文

注

このオプションは、プロシージャーがプロトタイプ化されていない場合でも、プロシージャー呼び出し点がその宣言に一致することを表明します。

呼び出し元は、汎用レジスター (GPR) ではなく、浮動小数点レジスターのみで浮動小数点引数を渡すことができます。 コンパイラーは、プロシージャー呼び出し時の引数の型が、プロシージャー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。

コンパイラーはプロトタイプを持たない関数に対して警告を発行します。

例

すべての関数がプロトタイプ化されていると想定して `my_c_program.c` をコンパイルするには、以下のように入力します。

```
xlc my_c_program.c -qproto
```

関連情報

- エラー検査とデバッグのオプション: その他のエラー検査とデバッグのオプション

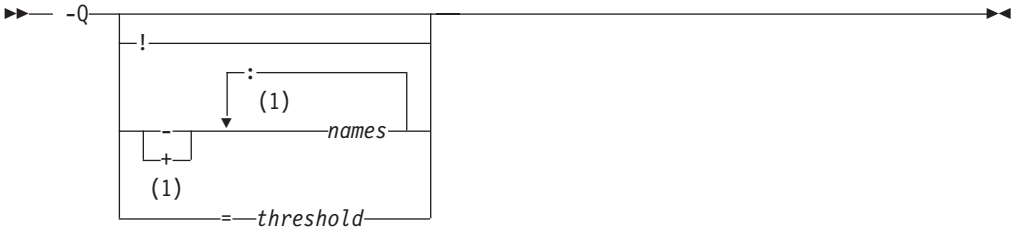
-Q

説明

C++ 言語アプリケーションでは、このオプションはコンパイラーに関数のインライン化を試行するよう命令する。インライン化は可能な場合に実行されますが、どの最適化が行われるに応じて、インライン化されない関数もあります。

C 言語アプリケーションでは、このオプションはコンパイラーがどの特定の関数のインライン化を試行するかを指定します。

構文



注:

1 C のみ

C++ C++ 言語では、以下の **-Q** オプションが適用されます。

-Q コンパイラーは可能な関数をすべてインライン化します。
-Q! コンパイラーは関数を一切インライン化しません。

C C 言語では、以下の **-Q** オプションが適用されます。

-Q **-Q** オプションのサブオプションの設定に従って、実行可能ソース・ステートメントが 20 以下の適切な関数をすべてインライン化しようとします。**-Q** が最後に指定されている場合は、すべての関数がインライン化されます。

-Q!
関数を一切インライン化しません。 **-Q!** が最後に指定されている場合、関数は一切インライン化されません。

-Q-*names*
names によってリストされた関数をインライン化しません。*names* の各関数名は、コロン (**:**) で区切ります。他のすべての適切な関数はインライン化されます。 このオプションは、**-Q** を暗黙指定します。

例を以下に示します。

```
-Q-salary:taxes:expenses:benefits
```

これによって、`salary`、`taxes`、`expenses`、または `benefits` という名前の関数以外のすべての関数が、可能であればインライン化されます。

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。

-Q+*names*
names にリストされた関数および他のすべての適切な関数をインライン化しようとします。 *names* の各関数名は、コロン (**:**) で区切る必要があります。このオプションは、**-Q** を暗黙指定します。

例を以下に示します。

```
    -Q+food:clothes:vacation
```

これによって、インライン化に適格な他の関数とともに、可能な場合、`food`、`clothes`、または `vacation` という名前の関数がすべてインライン化されます。

ソース・ファイルに定義されていない関数、または定義はされているがインライン化できない関数については、警告メッセージが出されます。

このサブオプションは、*threshold* 値の設定をすべてオーバーライドします。しきい値ゼロを **-Q+***names* と共に使用して、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。

```
-Q=0
```

これに以下を続けて指定します。

```
-Q+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。

-Q=*threshold*
インライン化する関数に対してサイズ制限を設定します。 実行可能ステートメントの数は、インライン化する関数の *threshold* 以下でなければなりません。 *threshold* は正の整数でなければなりません。デフォルト値は 20 です。しきい値 **0** を指定すると、`inline` 関数指定子のサポートされる形式でマークされた関数を除き、関数はインライン化されません。

threshold の値は、論理 C ステートメントに適用されます。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 */
  }
}
```

デフォルト

デフォルトでは、インライン指定はコンパイラーに対するヒントとして扱われます。インライン化が発生するかしないかは、選択される他のオプションにも依存する可能性があります。

- プログラムを最適化すると (**-O** オプションを指定)、コンパイラーはインラインとして宣言されている関数のインライン化を試行します。

注

-Q オプションは、 **-qinline** オプションと機能的に同等です。

(デバッグ情報を生成するために) **-g** オプションを指定すると、インライン化に影響する可能性があります。 104 ページの『-g』コンパイラー・オプションの情報を参照してください。

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。

再帰的関数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。

通常、最適化を要求する (**-O** オプション) とアプリケーションのパフォーマンスが最適化され、最適化を要求しないとコンパイラーのパフォーマンスが最適化されます。

`inline`、`_inline`、`_Inline`、`__inline__` および `__inline` 言語キーワードは、**-Q!** を除き、すべての **-Q** オプションをオーバーライドします。コンパイラーは、他の **-Q** オプションの設定に関係なく、これらのキーワードでマークされた関数をインライン化しようとします。

インライン化を最大限にするには、以下を行います。

- C プログラムの場合は、最適化 (**-O**) に加え、 C 言語用の適切な **-Q** オプションを指定します。
- C++ プログラムの場合は、最適化 (**-O**) は指定するが **-Q** オプションは指定しません。

例

関数が一切インライン化されないようにプログラム `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 -Q!
```

プログラム `my_c_program.c` をコンパイルして、12 行未満の関数のインライン化をコンパイラーに試行させるには、以下のように入力します。

```
xlc my_c_program.c -O -Q=12
```

関連情報

- 104 ページの『-g』
- パフォーマンス最適化のオプション: 関数のインライン化のオプション

-R

説明

実行時に、共用ライブラリーのパス・ディレクトリーを検索する。

構文

注

構成ファイルとコマンド行の両方に **-R***directory* オプションが指定されている場合は、実行時に構成ファイルに指定したパスが最初に検索されます。

-R コンパイラー・オプションは累積です。コマンド行で後に指定された **-R** オプションは、前の **-R** によって指定されたディレクトリー・パスを置換するのではなく、このパスへの追加を行います。

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。

例

ディレクトリー `/usr/tmp/old` が動的ライブラリー `libspfiles.so`の標準ディレクトリーと共に実行時に検索されるように `myprogram.c` をコンパイルするには、以下のように入力します。

```
xlc++ myprogram.C -lspfiles -R/usr/tmp/old
```

関連情報

- リンクを制御するオプション: リンカー入力制御のオプション

-r

説明

再配置可能オブジェクトを作成する。これにより未解決のシンボルが含まれる場合でも出力ファイルを生成することができます。

構文

注

このフラグを指定して作成されたファイルは、**xlc++** への別の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。

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
```

関連情報

- 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

-qreport

説明

プログラム・ループが並列化され、最適化される方法を示す変換レポートを作成するようコンパイラーに命令する。そして、**-qipa=clonearch** コンパイラー・オプションによって指定されたアーキテクチャーに対してクローン化されるプロシージャーを示す変換レポートも作成するよう命令する。変換レポートは、コンパイラー・リストの一部として組み込まれます。

構文

注

このオプションは **-qhot**、**-qsmp**、または **-qipa=clonearch** も有効でない限り無効です。

-qreport を、**-qhot** と一緒に指定すると、疑似 C のコード・リストとループの変換方法の要約を作成するようにコンパイラーに命令が出されます。この情報を使用して、プログラムでのループの実行を調整することができます。

-qreport を **-qsmp** と一緒に指定すると、プログラムがデータを処理する方法やプログラムでのループの自動並列化を示すレポートも作成するようコンパイラーに命令が出されます。この情報を使用して、プログラムでのループを並列化する方法、またはしない方法を決定することができます。

疑似 C のコード・リストは、コンパイルできるようにはなっていません。プログラムには疑似 C のコードは組み込まないでください。また、疑似 C のコード・リストに名前が表示される可能性のある内部ルーチンを、明示的に呼び出すことはしないでください。

例

コンパイラー・リストにループの最適化方法を示すレポートを組み込むように myprogram.C をコンパイルするには、以下のように入力します。

```
xlc++_r -qhot -O3 -qreport myprogram.C
```

myprogram.C をコンパイルし、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ているかを示すレポートも組み込むには、以下のように入力します。

```
xlc++_r -qsmp -O3 -qreport myprogram.C
```

関連情報

- 109 ページの『-qhot』
- 198 ページの『-qsmp』
- リストとメッセージを制御するオプション: メッセージのオプション

-qreserved_reg

C

説明

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

注

ターゲット・プラットフォームで有効なレジスター名を使用する必要があります。そうでないと、コンパイラーが警告メッセージを発行します。重複するレジスター名は、そのまま無視されます。

-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。有効なレジスター名は以下の通りです。

- r0-r31
- f0-f31
- v0-v31

例

```
xlc myprogram.c -qreserved_reg=r3:r4
```

r3 と r4 は、関数にパラメーターを渡して戻り値を受け取るという固定された役割以外では、生成されたコードで使用できないことを示しています。

関連情報

- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション
- 「*XL C/C++ 言語解説書*」の『指定されたレジスターのグローバル変数』

-qro

説明

ストリング・リテラルのストレージ・タイプを指定する。

構文

278 ページの『#pragma options』も参照してください。

デフォルト

cc とその派生物 **-qro** を除くすべてのコンパイラー呼び出しのデフォルト。**cc** コンパイラー呼び出しのデフォルトは **-qnoro** です。

注

-qro を指定した場合は、コンパイラーによってストリング・リテラルが読み取り専用ストレージに配置されます。 **-qnoro** を指定した場合は、ストリング・リテラルは読み取り/書き込みストレージに配置されます。

以下を使用して、ソース・プログラムにストレージ・タイプを指定することもできます。

```
#pragma strings storage_type
```

ここで、*storage_type* は、 **read-only** または **writable** です。

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が、読み取り専用のストリング・リテラルを変更しようと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。

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

- 292 ページの『#pragma strings』
- 『-qroconst』
- 出力を制御するオプション: ストリングおよび定数データの配置を制御するオプション

-qroconst

説明

定数値のストレージ・ロケーションを指定する。

構文

278 ページの『#pragma options』も参照してください。

デフォルト

xlc、**xlC**、および **c89** のデフォルトは **-qroconst** です。 **cc** でのデフォルトは **-qnoroconst** です。

注

-qroconst を指定した場合は、コンパイラーによって定数が読み取り専用ストレージに配置されます。 **-qnoroconst** を指定した場合は、定数値は読み取り/書き込みストレージに配置されます。

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。 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。以下の変数には、このオプションは適用されません。

- `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。

関連情報

- 190 ページの『-qro』
- 出力を制御するオプション: ストリングおよび定数データの配置を制御するオプション

-qrtti

C++

説明

このオプションを使用して、typeid 演算子と dynamic_cast 演算による例外処理および使用のための実行時型識別 (RTTI) 情報を生成する。

構文

ここで、使用できるサブオプションは以下のとおりです。

rtti	コンパイラーは、 RTTI の typeid および dynamic_cast 演算子に必要な情報を生成します。
nortti	コンパイラーは RTTI 情報を生成しません。

注

最良のランタイム・パフォーマンスを得るためには、デフォルトの **-qnortti** の設定により RTTI 情報の生成を抑止します。

C++ 言語は、実行時にオブジェクトのクラスを判別するための RTTI 機構を提供します。 これは、以下の 2 つの演算子から構成されます。

- オブジェクトの実行時の型を判別する演算子 (typeid)。
- 実行時に検査される型変換を行うための演算子 (dynamic_cast)。

type_info クラスには、使用可能な RTTI が記述されており、 typeid 演算子によって戻される型が定義されています。

-qrtti コンパイラー・オプションを指定する場合は、以下の影響に注意してください。

- **-qrtti** が指定されているときには、仮想関数テーブルの内容は異なります。
- オブジェクトをまとめてリンクするとき、対応するソース・ファイルはすべて、正しい **-qrtti** オプションを指定してコンパイルしなければなりません。
- ライブラリーを混合オブジェクト (いくつかのオブジェクトには **-qrtti** が指定されており、その他のオブジェクトには **-qnortti** が指定されている) でコンパイルすると、未定義シンボル・エラーとなる場合があります。

関連情報

- 88 ページの『-qeh』
- 出力を制御するオプション: その他の出力オプション

-S

説明

ソース・ファイルごとにアセンブラー言語ファイル (.s) を生成する。結果として得られる .s ファイルをアセンブルして、オブジェクトの .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

注

アセンブラーは、どの XL C/C++ 呼び出しコマンドで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、ローダ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
```

例

1. 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
   ```

2. 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
   ```

3. 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
- 86 ページの『-E』
- 104 ページの『-g』
- 122 ページの『-qipa』
- 170 ページの『-o』
- 171 ページの『-P』
- 213 ページの『-qtbtable』
- 出力を制御するオプション: ファイル出力のオプション

-s

説明

このオプションは、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。 **-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。

構文

注

strip コマンドを使用すると同じ効果があります。

関連情報
- 104 ページの『-g』
- 出力を制御するオプション: ファイル出力のオプション

-qsaveopt

説明

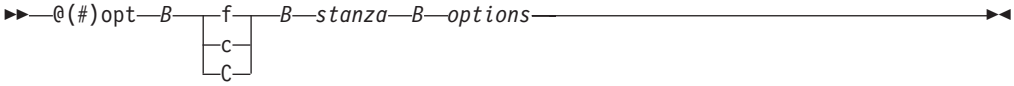
コンパイラー・オプションをオブジェクト・ファイルに保管する。

構文

注

このオプションを使用すると、コンパイラー・オプションをコンパイル中のオブジェクト・ファイルに保管することができます。このオプションは、オブジェクト (.o) ファイルにコンパイルしているときにのみ効果を持ちます。

ストリングは以下のフォーマットで保管されます。



ここで、

B スペースを示します。
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。
stanza コンパイルに使用されるドライバーを指定します。例えば、c89 または xlc++ など。
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。

関連情報

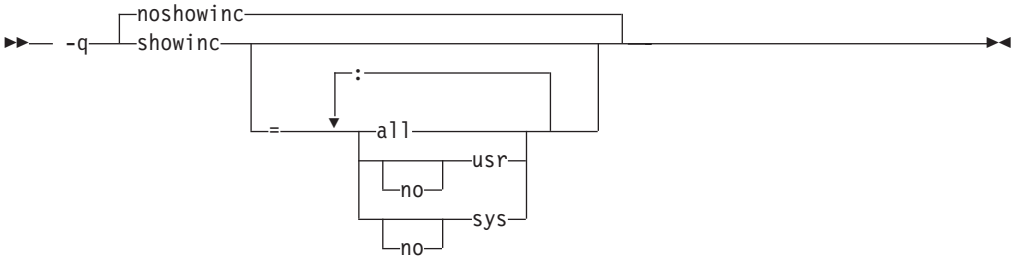
- 出力を制御するオプション: その他の出力オプション

-qshowinc

説明

-qsource と共に使用され、ユーザー・ヘッダー・ファイル (" " の使用を含む) またはシステム・ヘッダー・ファイル (< > の使用を含む) をプログラム・ソース・リストに選択して表示する。

構文



ここで、オプションは以下の通りです。

noshowinc 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どちらもプログラム・ソース・リストに表示しません。これは **-qshowinc=nousr:nosys** を指定した場合と同じです。

showinc	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。これは **-qshowinc=usr:sys** または **-qshowinc=all** を指定した場合と同じです。
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。これは **-qshowinc** または **-qshowinc=usr:sys** を指定した場合と同じです。
usr	ユーザー組み込みファイルをプログラム・ソース・リストに表示します。
sys	システム組み込みファイルをプログラム・ソース・リストに表示します。

278 ページの『#pragma options』も参照してください。

注

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。

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

- 200 ページの『-qsource』
- リストとメッセージを制御するオプション: リストのオプション

-qshowpdf

説明

-qpdf1 および最小の最適化レベル **-O** と共に使用された場合、追加の呼び出しとブロック・カウントのプロファイル情報を実行可能ファイルに追加する。

構文

注

このオプションは、**-qpdf1** コンパイラー・オプションと共に指定された場合にのみ有効です。

-qpdf1 と最小の最適化レベル **-O** と共に指定された場合、コンパイラーは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します。コンパイルされたアプリケーションを実行すると、呼び出しとブロックのカウントがファイル ._pdf に記録されます。

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、**showpdf** ユーティリティーを使用して ._pdf ファイルの内容を検索することができます。このユーティリティーについては、**-qpdf** ページに説明があります。

例

この例では、プログラム・ファイル hello.c に対して以下のソースを想定しています。

```
#include <stdio.h>

void HelloWorld()
{
 printf("Hello World");
}

main()
{
 HelloWorld();
}
```

以下を指定してソースをコンパイルします。

```
xlc -qpdf1 -O -qshowpdf hello.c
```

結果のプログラム実行可能ファイルを実行します。

```
a.out
```

showpdf ユーティリティーを使用して、この実行可能ファイルの呼び出しとブロックのカウントを表示します。

```
showpdf
```

以下のような出力が **showpdf** ユーティリティーによって戻されます。

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

Call Counters:
  5 | 1  printf(6)

Call coverage = 100% ( 1/1 )

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

Block coverage = 100% ( 2/2 )


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

Call Counters:
  10 | 1  HelloWorld(4)

Call coverage = 100% ( 1/1 )

Block Counters:
  8-11 | 1
  11 |

Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

関連情報

- 174 ページの『-qpdf1、-qpdf2』

- パフォーマンス最適化のオプション: パフォーマンス・データ割り振りのオプション

-qsmallstack

説明

スタック・フレームのサイズを削減するようコンパイラーに命令する。

構文

注

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。

例

`myprogram.c` をコンパイルして使用するスタック・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

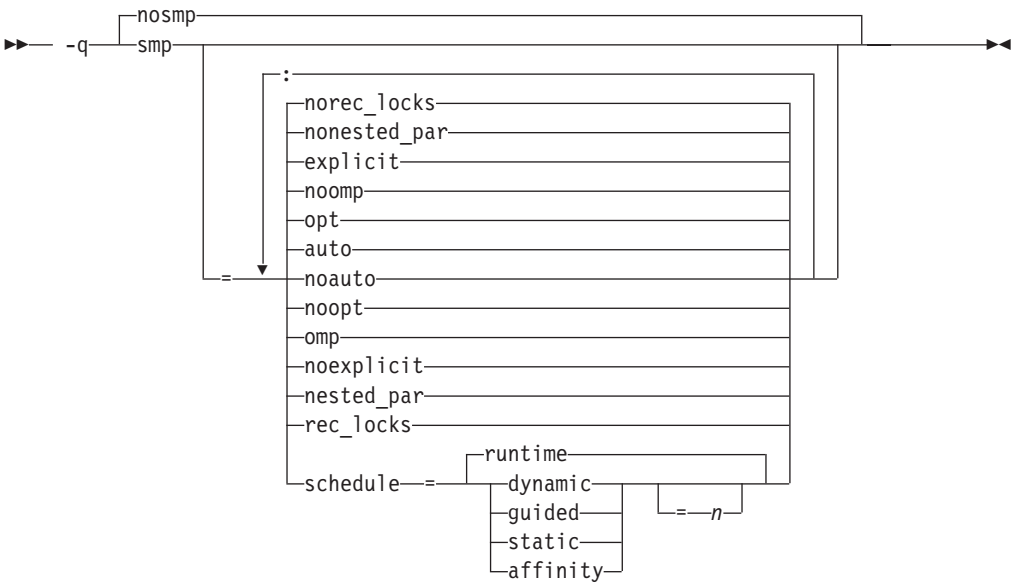
- 104 ページの『-g』
- 122 ページの『-qipa』
- パフォーマンス最適化のオプション: 最適化を制限するオプション

-qsmp

説明

プログラム・コードの並列化を使用可能にする。

構文



ここで、

auto
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にします。

noauto
プログラム・コードの自動並列化を使用不可にします。 OpenMP プラグマ・ステートメントと明示的に並列化されるプログラム・コードは最適化されます。

opt
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にします。

noopt
自動並列化は使用可能にしますが、並列化されたプログラム・コードの最適化は使用不可にします。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この設定を使用してください。

omp
OpenMP 標準への厳密な準拠を使用可能にします。自動並列化は使用不可になります。並列化されたプログラム・コードは最適化されます。OpenMP 並列化プラグマだけが認識されます。

noomp
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にします。

explicit
ループの明示的並列化を制御するプラグマを使用可能にします。

noexplicit
ループの明示的並列化を制御するプラグマを使用不可にします。

nested_par
これを指定すると、ネストされた並列構成は直列化されません。 **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。

このオプションを使用する場合は、注意が必要です。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。

nonested_par
ネストされた並列構成の並列化を使用不可にします。

rec_locks	これを指定すると、再帰的ロックが使用され、ネストされたクリティカル・セクションがデッドロックの原因となることはありません。
norec_locks	これを指定すると、再帰的ロックは使用されません。
schedule=*sched_type*[=*n*]	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。 *sched_type* が指定されていない場合、デフォルト設定として **schedule=runtime** が想定されます。

注

- **-qsmp** は、**xlc_r** など、スレッド・セーフのコンパイラー・モードの呼び出しにのみ使用する必要があります。これらの呼び出しによりすべてのデフォルト・ランタイム・ライブラリーの **Pthreads**、**xlsmp**、およびスレッド・セーフのバージョンが結果の実行可能ファイルに確実にリンクされます。
- **-qnosmp** のデフォルト・オプション設定では、構文検査は実行されますが、並列化ディレクティブのためのコードが生成されないよう指定されています。**-qignprag=omp** を使用すると、並列化ディレクティブが完全に無視されます。
- サブオプションなしで **-qsmp** を指定することは、**-qsmp=auto:explicit:noomp:norec_locks:nonested_par:schedule=runtime**、または **-qsmp=opt:explicit:noomp:norec_locks:nonested_par:schedule=runtime** を指定するのと同じです。
- **-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** を指定すると、**-qhot=level=1** が有効であることが暗黙指定されます。

関連情報
- 166 ページの『-O、-qoptimize』
- 217 ページの『-qthreaded』
- 298 ページの『並列処理のためのプラグマ・ディレクティブ』
- 332 ページの『並列処理のための組み込み関数』
- コマンド行オプションの要約: 最適化フラグ
- 245 ページの『OpenMP プラグマ・ディレクティブの要約』

-qsource

説明

コンパイラー・リストを作成してソース・コードを組み込む。

構文

278 ページの『#pragma options』も参照してください。

注

-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 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。

例

myprogram.C をコンパイルして、myprogram.C のソース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
```

コンパイラー・リストにプログラム・ソースの選択した部分だけを表示したい場合は、**-qsource** コンパイラー・オプションを使用しないでください。以下のコードでは、 **#pragma options source** ディレクティブと **#pragma options nosource** ディレクティブの間にあるソースがコンパイラー・リストに含まれます。

```
#pragma options source
   . . .
/* Source code to be included in the compiler listing
   is bracketed by #pragma options directives.
*/
   . . .
#pragma options nosource
```

関連情報

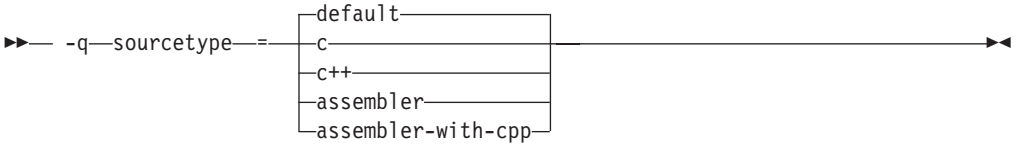
- 181 ページの『-qprint』
- リストとメッセージを制御するオプション: リストのオプション

-qsourcetype

説明

実際のソース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・ファイルをこのオプションによって指定されたソース・タイプとして扱うようコンパイラーに命令します。

構文



ここで、

ソース・タイプ	サフィックス	振る舞い
デフォルト	-	コンパイラーはソース・ファイルのプログラム言語がそのファイル名のサフィックスによって暗黙指定されることを想定します。
c	• .c • .i (プリプロセス済みファイル用)	コンパイラーはこのオプションの後のすべてのソース・ファイルを C 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。
c++	• .C、 .cc、 .cpp、 .cxx • .cp、.c++	コンパイラーはこのオプションの後のすべてのソース・ファイルを C++ 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。
assembler	• .s	コンパイラーはこのオプションの後のすべてのソース・ファイルを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。
assembler-with-cpp	• .S	コンパイラーはこのオプションの後のすべてのソース・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。

注

通常、コンパイラーはコマンド行に指定されたソース・ファイルのファイル名サフィックスを使用してソース・ファイルのタイプを判別します。例えば、**.c** のサフィックスは通常、 C ソース・コードを暗黙指定し、**.C** のサフィックスは通常 C++ ソース・コードを暗黙指定し、コンパイラーはこれらのファイルを以下のように扱います。

hello.c ファイルは C ファイルとしてコンパイルされます。

hello.C ファイルは C++ ファイルとしてコンパイルされます。

-qsourcetype オプションは、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。これはファイル・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。しかし、大/小文字を区別しないファイル・システムでは、前述の 2 つのコンパイルは同じ物理ファイルを参照します。つまり、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が、ファイル・システムからファイルを検索するときには大/小文字の違いを無視します。

このオプションはコマンド行でこのオプションより後に 指定されたファイルだけに影響し、このオプションよりも前のファイルには影響しない点に注意してください。したがって、以下の例では、

```
xlc goodbye.C -qsourcetype=c hello.C
```

hello.C は C ソース・ファイルとしてコンパイルされますが、goodbye.C は C++ ファイルとしてコンパイルされます。

-qsourcetype オプションは、**-+** オプションと一緒に使用しないでください。

例

ソース・ファイル hello.C を C 言語のソース・コードとして扱うには、以下のように入力します。

```
xlc -qsourcetype=c hello.C
```

関連情報

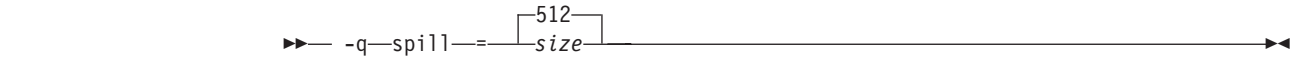
- 50 ページの『-+ (正符号)』
- 入力を制御するオプション: その他の入力オプション

-qspill

説明

レジスター割り振り予備域を *size* バイトに指定する。

構文



278 ページの『#pragma options』も参照してください。

注

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。

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したい場合には、以下のように入力します。

```
xlc myprogram.c -qspill=900
```

関連情報

- パフォーマンス最適化のオプション: 最適化を制限するオプション

-qsrcmsg

C

説明

対応するソース・コード行を stderr ファイル内の診断メッセージに追加する。

構文

278 ページの『#pragma options』も参照してください。

注

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。 **-qnosrcmsg** を指定すると、ソース行とフィンガー行の両方の生成が抑制され、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るようになります。

再構成されたソース行は、マクロ展開後の状態を表します。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。

デフォルト (**-qnosrcmsg**) では、解析可能な簡潔なメッセージが表示されます。エラーごとにソース行およびポインターが提供されない代わりに、単一の行が表示され、エラーがあるソース・ファイルの名前、エラーの行と文字の列位置、およびメッセージ自体を示す単一の行が示されます。

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
```

関連情報

- リストとメッセージを制御するオプション: メッセージのオプション

-qstaticinline

C++

説明

このオプションは、インライン関数を静的関数として扱うか、extern として扱うかを制御する。デフォルトでは、XL C/C++ は、インライン関数を extern として扱います。異なるソース・ファイルにいくつ関数の定義が現れるかに関係なく、インライン関数指定子によってマークされた 1 つの関数に対して生成される関数本体は 1 つのみです。

構文

例

-qstaticinline オプションを使用すると、以下の宣言における関数 **f** は、たとえ明示的に静的であると宣言していなくても、静的関数として扱われます。関数の定義ごとに別の関数本体が作成されます。これによりコード・サイズがかなり大きくなる可能性がある点に注意してください。

```
inline void f() {/*...*/};
```

デフォルトの **-qnostaticinline** を使用すると、 **f** に外部結合が与えられます。

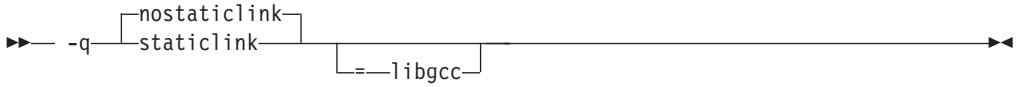
関連情報
- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-qstaticlink

説明

-qstaticlink コンパイラー・オプションは、共用および非共用ランタイム・ライブラリーをどのようにアプリケーションにリンクするかを制御します。このオプションは、単独または組み合わせて使用される GNU オプション **-static**、**-static-libgcc**、および **-shared-libgcc** によって暗黙指定されるものと同等のリンク規則を指定する能力を提供します。

構文



ここで、

nostaticlink　　`libgcc.a` と静的にリンクしないようコンパイラーに命令します。

staticlink　　このコンパイラー・オプションを有効にして生成されたオブジェクトは、静的ライブラリーにしかリンクしません。

libgcc　　このサブオプションが **nostaticlink** とともに指定されていると、コンパイラーは共用バージョンの `libgcc` にリンクします。

staticlink とともに指定されていると、コンパイラーは静的バージョンの `libgcc` にリンクします。

注

共用および非共用ライブラリーに対する GNU サポートは、以下のテーブルに示したオプションによって制御されます。

表 39. オプション・マッピング: Linux リンカーの制御

GNU オプション	意味	XL C/C++ オプション
-shared	共用オブジェクトをビルドします。	-qmkshrobj
-static	静的オブジェクトをビルドして、共用ライブラリーとのリンクを回避します。リンクされるすべてのライブラリーは、静的ライブラリーでなければなりません。**-shared** を使用して指定された場合は無視されます。	-qstaticlink
-shared-libgcc	`libgcc` の共用バージョンを使用します。**-static** を使用して指定された場合は無視されます。	-qnostaticlink=libgcc

表 39. オプション・マッピング: Linux リンカーの制御 (続き)

GNU オプション	意味	XL C/C++ オプション
-static-libgcc	libgcc の静的バージョンを使用します。	-qstaticlink=libgcc

関連情報
- 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

-qstatsym

説明

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、名前リスト (オブジェクトのシンボル・テーブル) に追加する。

構文

デフォルト

デフォルトでは、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。

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
```

関連情報
- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-qstdinc

説明

どのディレクトリーが **#include <***file_name***>** および **#include "***file_name***"** ディレクティブによって組み込まれたファイルに使用されるかを指定する。**-qnostdinc** オプションは、標準のインクルード・ディレクトリーを検索パスから除外します。

構文

278 ページの『#pragma options』も参照してください。

注

-qnostdinc を指定した場合、**-I** *directory* オプションを使用して明示的にデフォルト検索パス・ディレクトリーを追加していない限り、コンパイラーは、これらのディレクトリーを検索しません。

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。 しかし、相対パス名にはすべて影響します。

-qnostdinc は、**-qidirfirst** から独立しています (**-qidirfirst** は、現行ソース・ファイルが常駐するディレクトリーを検索する前に、**-I** *directory* で指定されたディレクトリーを検索します)。

ファイルの検索順序については、26 ページの『相対パス名を使用した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。

最後の有効な **#pragma options [NO]STDINC** は、それ以後の **#pragma options [NO]STDINC** によって置き換えられるまで、有効なままになります。

例

`#include "myinc.h"` ディレクティブによって `myprogram.c` に組み込まれるファイルをディレクトリー `/tmp/myfiles` で検索するように `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 -qnostdinc -I/tmp/myfiles
```

関連情報

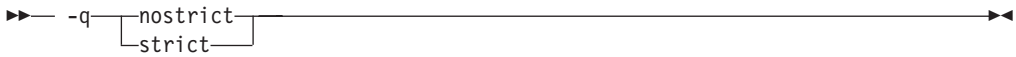
- 111 ページの『-I』
- 112 ページの『-qidirfirst』
- 26 ページの『相対パス名を使用した組み込みファイルのディレクトリー検索シーケンス』
- 入力を制御するオプション: 検索パスのオプション

-qstrict

説明

プログラムのセマンティクスを変更する可能性がある積極的な最適化をオフにする。

構文



278 ページの『#pragma options』も参照してください。

デフォルト

- 最適化レベル **-O3** 以上では **-qnostrict**。
- それ以外では **-qstrict**

注

-qstrict は、以下の最適化をオフにします。

- コード・モーションを実行し、例外をトリガーする可能性のあるロードや浮動小数点計算などの計算をスケジューリングする。
- IEEE 規則への準拠を緩和する。
- 浮動小数点式の再関連付けを行う。

このオプションは、**-O2** 以上の最適化レベルでのみ有効です。

-qstrict は **-qfloat=norsqrt** を設定します。

-qnostrict は **-qfloat=rsqrt** を設定します。

-qfloat=rsqrt を使用して **-qstrict** の設定値をオーバーライドできます。

例を以下に示します。

- **-O3 -qnostrict -qfloat=norsqrt** を使用すると、コンパイラーは、**-qfloat=rsqrt** 以外のすべての積極的な最適化を行います。

-qnostrict で設定されたオプションと、**-qfloat=***options* で設定されたオプションが矛盾する場合は、最後に指定されたオプションが認識されます。

例

-O3 の積極的な最適化がオフになり、 逆数による除算で平方根の結果による除算が置換される (**-qfloat=rsqrt**) ように `myprogram.C` をコンパイルするには、以下のように入力します。

```
xlc++ myprogram.C -O3 -qstrict -qfloat=rsqrt
```

関連情報

- 96 ページの『-qfloat』
- 166 ページの『-O、-qoptimize』
- パフォーマンス最適化のオプション: 最適化を制限するオプション

-qstrict_induction

説明

プログラムのセマンティクスを変更する可能性のあるループ帰納変数の最適化を使用不可にする。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、このような最適化はプログラムの結果を変更することがあります。

構文

デフォルト

- 最適化レベル 2 以上の場合、**-qnostrict_induction**。
- それ以外の場合、**-qstrict_induction**。

注

-O2 を指定すると、**-qnostrict_induction** が暗黙指定されます。両方を指定することは不要です。

-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。

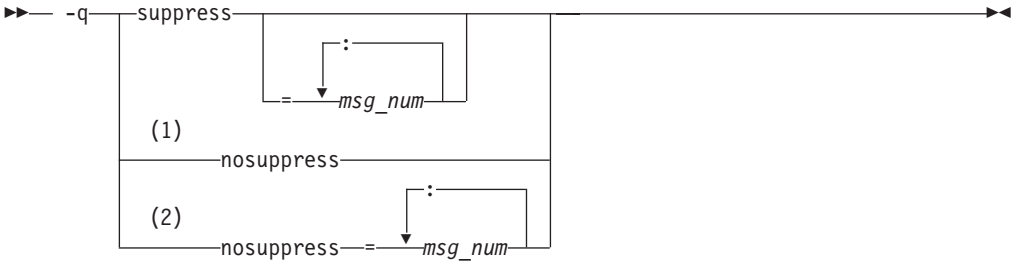
関連情報
- 166 ページの『-O、-qoptimize』
- パフォーマンス最適化のオプション: ループ最適化のオプション

-qsuppress

説明

指定したコンパイラー・メッセージやドライバー通知メッセージ、または警告メッセージが表示されたりリストに追加されるのを防ぐ。

構文



注:

1 C のみ

2 C++ のみ

注

このオプションは、コンパイラー・メッセージのみを抑制します。リンカー・メッセージまたはオペレーティング・システム・メッセージには影響しません。

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。

(S) および (U) レベルのメッセージなど、コンパイルを停止させる原因となるコンパイラー・メッセージ は抑止できません。

-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。

例

プログラムが、通常、結果として以下を出力する場合、

```
"myprogram.C", line 1.1:1506-224 (I) Incorrect #pragma ignored
```

以下のように指定してコンパイルすることにより、このメッセージを抑制することができます。

```
xlc++ myprogram.C -qsuppress=1506-224
```

関連情報

- 107 ページの『-qhalt』
- 122 ページの『-qipa』
- リストとメッセージを制御するオプション: メッセージのオプション

-qsymtab

C

説明

このオプションの設定値は、シンボル・テーブルに示される情報を決定する。

構文

ここで、

unref　すべての typedef 宣言、struct 型定義、union 型定義、および enum 型定義を、GNU GDB デバッガーによる処理用に組み込むことを指定します。

-g オプションと一緒にこのオプションを使用して、デバッガーで使用するための追加のデバッグ情報を生成します。

-g オプションを指定すると、デバッグ情報がオブジェクト・ファイルに組み込まれます。オブジェクトおよび実行可能ファイルのサイズを最小にするために、コンパイラーは、参照されるシンボルに関する情報しか組み込みません。デバッグ情報は、**-qsymtab=unref** を指定しない限り、参照されない配列、ポインター、またはファイル・スコープ変数については生成されません。

-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、名前リストに追加します。

デフォルトでは、静的変数はシンボル・テーブルに追加されません。

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
```

デバッガーで使用するために、myprogram.c 内のすべてのシンボルをシンボル・テーブルに組み込むには、次のように入力します。

```
xlc myprogram.c -g -qsymtab=unref
```

関連情報
- 104 ページの『-g』
- エラー検査とデバッグのオプション: デバッグのオプション

-qsyntaxonly

C

説明

コンパイラーに、オブジェクト・ファイルを生成せずに構文検査を行わせる。

構文

注

-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。

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
```

または

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

2 番目の例では、 **-qsyntaxonly** オプションによって **-o** オプションがオーバーライドされるため、オブジェクト・ファイルは作成されません。

関連情報
- 66 ページの『-C』
- 66 ページの『-c』
- 86 ページの『-E』
- 170 ページの『-o』
- 171 ページの『-P』
- 入力を制御するオプション: その他の入力オプション

-t

説明

-B オプションで指定したプレフィックスを、指定したプログラムに追加します。

構文

プログラムは以下の通りです。

プログラム	説明
c	コンパイラーのフロントエンド
b	コンパイラーのバックエンド
p	コンパイラー・プリプロセッサー
a	アセンブラー
I	プロシージャー間分析 (コンパイル・フェーズ)
L	プロシージャー間分析 (リンク・フェーズ)
l	リンケージ・エディター

デフォルト

-B は指定されているが、*prefix* は指定されていない場合のデフォルト・プレフィックスは `/lib/o` です。 **-B***prefix* がまったく指定されていない場合、標準プログラム名のプレフィックスは `/lib/n` です。

-B は指定されているが、**-t***programs* が指定されていない場合は、デフォルトですべての標準プログラム名のパス名が構成されます。

例

コンパイラーおよびアセンブラーのプログラム名に `/u/newones/compilers/` というプレフィックスが付くように `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 -B/u/newones/compilers/ -tca
```

関連情報

- 64 ページの『-B』
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qtabsize

説明

コンパイラーによって認識されるタブの長さを変更する。

構文

n は、ソース・プログラム内のタブを表す文字スペースの数です。

注

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。 例えば、**-qtabsize=1** を指定した場合、コンパイラーはタブの幅が 1 文字であると見なします。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。

関連情報

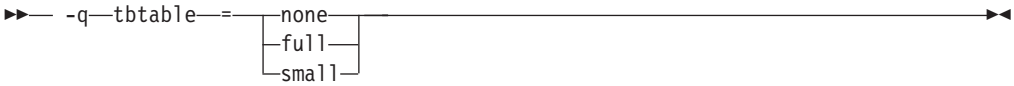
- リストとメッセージを制御するオプション: リストのオプション

-qtbtable

説明

関数の型、スタック・フレーム、およびレジスター情報を含め、各関数についての情報を含むトレースバック・テーブルを生成する。トレースバック・テーブルは、そのコードの終わりのテキスト・セグメントに配置されます。

構文



サブオプションは以下の通りです。

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。
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。 **-qnoopt** または **-g** を指定した場合には、これがデフォルトです。
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。 最適化は指定したが **-g** は指定していない場合には、これがデフォルトです。

278 ページの『#pragma options』も参照してください。

注

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。

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。コンパイラー構成ファイルには、この要件に合った項目が入っています。最適化されたコードに完全なトレースバック・テーブルが不要な場合は、コンパイラー構成ファイルを以下のように変更することによって、ファイル・スペースを節約することができます。

1. C または C++ コンパイル・スタンザの **options** 行から、 **-qtbtable=full** オプションを除去する。
2. **DFLT** スタンザの **xlCopt** 行から、 **-qtbtable=full** オプションを除去する。

これらの変更を行うと、 **tbtable** オプションのデフォルトが以下のようになります。

- 最適化オプションを設定してコンパイルするときは、**-qtbtable=small**
- 最適化オプションを設定せずにコンパイルするときは、**-qtbtable=full**

関連情報

- 104 ページの『-g』
- コマンド行オプションの要約: その他のエラー検査とデバッグ・オプション

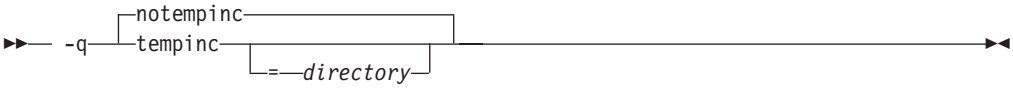
-qtempinc

C++

説明

テンプレート関数およびクラス宣言用に別個のテンプレート組み込みファイルを生成し、オプションで指定できるディレクトリーにこれらのファイルを入れます。

構文



注

-qtempinc コンパイラー・オプションと **-qtemplateregistry** コンパイラー・オプションは、相互に排他的です。**-qtempinc** を指定すると、 **-qnotemplateregistry** が暗黙指定されます。同様に、**-qtemplateregistry** を指定すると、**-qnotempinc** が暗黙指定されます。ただし、**-qnotempinc** を指定しても、 **-qtemplateregistry** が暗黙指定されるわけではありません。

-qtempinc または **-qtemplateregistry** のいずれかを指定すると、**-qtmplinst=auto** が暗黙指定されます。

-qtempinc を指定すると、コンパイラーは 1 の値を __TEMPINC__ マクロに割り当てます。この割り当ては、 **-qnotempinc** を指定した場合には行われません。

例

ファイル `myprogram.c` をコンパイルして、テンプレート関数について生成された組み込みファイルを `/tmp/mytemplates` ディレクトリーに配置するには、以下のように入力します。

```
xlc++ myprogram.C -qtempinc=/tmp/mytemplates
```

関連情報

- 218 ページの『-qtmplinst』
- 215 ページの『-qtemplateregistry』
- 215 ページの『-qtemplaterecompile』
- コンパイラーのカスタマイズのオプション: テンプレートに関連するオプション
- 「*XL C/C++ プログラミング・ガイド*」の『C++ テンプレートの使用法』

-qtemplaterecompile

C++

説明

-qtemplateregistry コンパイラー・オプションを使用してコンパイルされたコンパイル単位間の依存性の管理に役立つ。

構文

注

-qtemplaterecompile オプションは、**-qtemplateregistry** オプションと共に使用されることを意図しています。 複数のコンパイル単位が同一のテンプレート・インスタンス生成を参照するプログラムを指定すると、**-qtemplateregistry** オプションは、そのインスタンス生成を入れるための単一のコンパイル単位を指定します。他のコンパイル単位にはこのインスタンス生成は入れられず、オブジェクト・コードの重複が避けられます。

以前コンパイルされたソース・ファイルが再度コンパイルされる場合、**-qtemplaterecompile** オプションは、テンプレート・レジストリーに問い合わせて、このソース・ファイルへの変更が他のコンパイル単位の再コンパイルを必要とするかどうかを判断します。これは、指定されたインスタンス化およびそのインスタンス化を以前含んでいた対応オブジェクト・ファイルを最早参照しないような方法でソース・ファイルが変更された場合に起こります。この場合、影響を受けるコンパイル単位は自動的に再コンパイルされます。

-qtemplaterecompile オプションでは、コンパイラーによって生成されたオブジェクト・ファイルが、最初に書き込まれたサブディレクトリー内に残ることが必要となります。自動ビルド・プロセスがオブジェクト・ファイルを元のサブディレクトリーから移動した場合、 **-qtemplateregistry** が使用可能になっているときは、必ず **-qnotemplaterecompile** オプションを使用してください。

関連情報

- 218 ページの『-qtmplinst』
- 214 ページの『-qtempinc』
- 『-qtemplateregistry』
- コンパイラーのカスタマイズのオプション: テンプレートに関連するオプション
- 「*XL C/C++ プログラミング・ガイド*」の『C++ テンプレートの使用法』

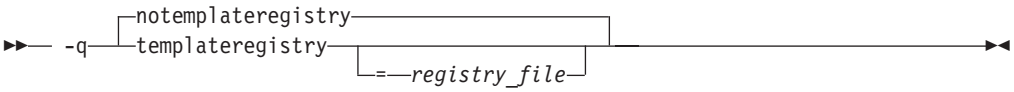
-qtemplateregistry

C++

説明

ソース内で検出されるすべてのテンプレートについてそのレコードを保守し、テンプレートごとに一度だけインスタンス化が行われることを保証する。

構文



注

コンパイラーが最初にテンプレートのインスタンス化の参照を検出すると、そのインスタンスが生成され、それに関連したオブジェクト・コードが現行オブジェクト・ファイルに配置されます。別のコンパイル単位で、同じテンプレートの同一のインスタンス化への参照がさらにある場合、それらの参照は記録はされますが、冗長のインスタンス化が生成されることはありません。 **-qtemplateregistry** オプションを使用するのに、特別なファイル編成は必要ありません。

場所を指定しない場合、コンパイラーは、現行作業ディレクトリーに保管されている **templateregistry** ファイルにすべてのテンプレート・レジストリー情報を保管します。テンプレート・レジストリー・ファイルを異なるプログラム間で共用することはできません。ソースが同じディレクトリーに入っている複数のプログラムがある場合、現行作業ディレクトリーに保管されているデフォルトのテンプレート・レジストリー・ファイルに依存すると、この状態になり、誤った結果が作成される可能性があります。

-qtempinc コンパイラー・オプションと **-qtemplateregistry** コンパイラー・オプションは、相互に排他的です。**-qtempinc** を指定すると、 **-qnotemplateregistry** が暗黙指定されます。同様に、**-qtemplateregistry** を指定すると、**-qnotempinc** が暗黙指定されます。ただし、**-qnotempinc** を指定しても、 **-qtemplateregistry** が暗黙指定されるわけではありません。

-qtempinc または **-qtemplateregistry** のいずれかを指定すると、**-qtmplinst=auto** が暗黙指定されます。

例

ファイル `myprogram.C` をコンパイルしてテンプレートのレジストリー情報を `/tmp/mytemplateregistry` ファイルに配置するには、以下のように入力します。

```
xlc++ myprogram.C -qtemplateregistry=/tmp/mytemplateregistry
```

関連情報

- 218 ページの『-qtmplinst』
- 214 ページの『-qtempinc』
- 215 ページの『-qtemplaterecompile』
- コンパイラーのカスタマイズのオプション: テンプレートに関連するオプション
- 「*XL C/C++ プログラミング・ガイド*」の『C++ テンプレートの使用法』

-qtempmax

C++

説明

-qtempinc オプションにより各ヘッダー・ファイルごとに生成されるテンプレート組み込みファイルの最大数を指定する。

構文

注

number に 1 から 99999 の値を指定して、テンプレート・ファイルの最大数を指定します。

インスタンス化は、複数のテンプレート組み込みファイルにわたって行われます。

このオプションは、 **-qtempinc** オプションによって生成されるファイルのサイズが大きくなり過ぎて、新しいインスタンスの作成時の再コンパイルに膨大な時間がかかるときに使用してください。

関連情報
- 214 ページの『-qtempinc』
- コンパイラーのカスタマイズのオプション: テンプレートに関連するオプション
- 「*XL C/C++ プログラミング・ガイド*」の『C++ テンプレートの使用法』

-qthreaded

説明

プログラムが複数のスレッドを使用することをコンパイラーに示す。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、すべての最適化がスレッド・セーフであることを保証します。

構文

デフォルト

デフォルトは、 **r** 呼び出しモードでコンパイルしたときは **-qthreaded**、その他の呼び出しモードでコンパイルしたときは **-qnothreaded** です。

注

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。

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、すでにスレッド・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。

関連情報
- 198 ページの『-qsmp』

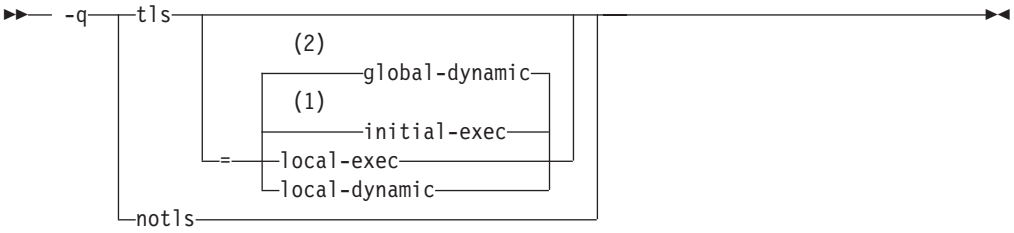
- 出力を制御するオプション: その他の出力オプション

-qtls

説明

アプリケーションが使用するスレッド局在のストレージ・モデルを指定する。

構文



注:

1 **-qnopic** コンパイラー・オプションが有効な場合はデフォルト。

2 **-qpic** コンパイラー・オプションが有効な場合はデフォルト。

注

このオプションでは、スレッド局在のストレージにアクセスするのに使用されるモデルを選択します。

GNU **__thread** キーワードをサポートするシステム上では、**vac_configure** ツールは **-qtls** をコンパイラー構成ファイルのデフォルト・オプションのセットに追加します。

-qtls がサブオプションを選択せずに指定されている場合は、コンパイラーは以下の設定を想定します。

- **-qnopic** が有効な場合は **-qtls=initial-exec**。
- **-qpic** が有効な場合は **-qtls=global-dynamic**。

関連情報

- 179 ページの『-qpic』
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-qtmplinst

C++

説明

テンプレートの暗黙のインスタンス生成を管理する。

構文

ここで、サブオプションは以下の通りです。

auto
: **-qtempinc** および **-qtemplateregistry** オプションに応じて暗黙のインスタンス生成を管理します。 **-qtempinc** と **-qtemplateregistry** の両方が使用不可の場合、暗黙のインスタンス生成は必ず実行されます。そうでない場合は、オプションの両方またはいずれかが使用可能な場合、使用可能になっているいずれかのオプションをコンパイラーが使用して暗黙のインスタンス生成を管理します。

always
: 暗黙のインスタンス生成を常に実行するようコンパイラーに命令します。これが指定されている場合は、**-qtempinc** と **-qtemplateregistry** コンパイラー・オプションは無視されます。

noinline
: 暗黙のインスタンス生成を実行しないようコンパイラーに命令します。これが指定されている場合は、**-qtempinc** と **-qtemplateregistry** コンパイラー・オプションは無視されます。

none
: インライン関数のインスタンスだけを生成するようコンパイラーに命令します。他の暗黙のインスタンス生成は実行されません。これが指定されている場合は、**-qtempinc** と **-qtemplateregistry** コンパイラー・オプションは無視されます。

注

- **-qtempinc** または **-qtemplateregistry** オプションは **-qtmplinst=auto** を暗黙指定します。
- **-qtempinc** と **-qtemplateregistry** オプションの両方を **-qtmplinst** と共に指定すると、最後に指定されたオプションが優先されます。例えば、**-qtmplinst** を指定し、次に **-qtemplateregistry** を指定すると、最終結果は **-qtmplinst=auto** と **-qtemplateregistry** になります。
- **-qtmplinst=auto** が指定されている場合は、 **-qtempinc** および **-qtemplateregistry** オプションと相対してそれがどの順序で表示されているかは関係ありません。

関連情報

- 215 ページの『-qtemplateregistry』
- 214 ページの『-qtempinc』
- 215 ページの『-qtemplaterecompile』
- コンパイラーのカスタマイズのオプション: テンプレートに関連するオプション

-qtmplparse

C++

説明

このオプションは構文解析およびセマンティック検査をテンプレート定義 (クラス・テンプレート定義、関数本体、メンバー関数本体、および静的データ・メンバー初期化指定子) に適用するか、またはテンプレートのインスタンス生成にのみ適用するかを制御する。コンパイラーは、テンプレート定義内の関数本体と変数初期化指定子を検査して、エラー・メッセージまたは警告メッセージを生成することができます。

構文

サブオプションは以下の通りです。

no	テンプレート定義を構文解析しません。これにより、前のバージョンの VisualAge C++ および先行製品用に作成されたコードで発生するエラーの数を減らすことができます。これがデフォルトです。
warn	テンプレート定義を構文解析し、意味エラーの場合に警告メッセージを発行します。
error	テンプレートのインスタンスが生成されない場合でも、テンプレート定義内の問題をエラーとして扱います。

注

このオプションは、テンプレートのインスタンス化ではなく、テンプレート定義に適用されます。このオプションの設定に関係なく、定義の外で問題が起こるとエラー・メッセージが生成されます。例えば、以下のように構成体の構文解析またはセマンティック検査中にエラーが見つかると、必ずエラー・メッセージが生成されます。

- 関数テンプレートの戻りの型
- 関数テンプレートのパラメーター・リスト

例

例 1:

以下の例では、テンプレート・クラスはインスタンス化されていません。したがってコンパイルによって構文解析されず、また **-qtmplparse=error** オプションを使用しなければ、コンパイラーは構文エラーを検出しません。ただし、**-qtmplparse=error** を使用すると、テンプレート・クラスは構文解析され、コンパイラーはエラー・メッセージにフラグを立てます。

```
template <class A> struct container
{
   A a1;
   A foo1()      //syntax error
   int _data;
};

xlC -c -qtmplparse=error myprogram.cpp
```

例 2:

以下の例では収容クラスのインスタンスが作成されないため、その行外 (out-of-line) メンバー定義は構文解析されません。**-qtmplparse=error** を使用しないと、コンパイラーは行外 (out-of-line) 定義とオリジナル定義の間のミスマッチに対してエラー・メッセージを発行しません。

```
template <class A> struct container
{
   void member(A a);
};
```

```
// error - this member is not declared in the struct container
template <class B> void container<B>::member()   {
}
xlC -c -qtmplparse=error myprogram.cpp
```

注: -qtmplparse=error を使用するかしないかに関係なく、クラスのインスタンスを作成しようとすると、コンパイラーがエラー・メッセージを発行します。

関連情報
- コンパイラーのカスタマイズのオプション: テンプレートに関連するオプション
- 「*XL C/C++ プログラミング・ガイド*」の『C++ テンプレートの使用法』

-qtocdata

説明
ローカルとしてデータにマークを付ける。

構文


注
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。

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。**-qtocdata** は、すべての変数がローカルであると想定するようコンパイラーに命令します。

インポートされる変数がローカル変数であると見なされた場合、不正なコードが生成され、パフォーマンスが低下することがあります。インポートされた変数は、ライブラリーの共用部分に動的にバインドされます。 **-qnotocdata** は、すべての変数がインポートされたものであると想定するようコンパイラーに命令します。

データにマークを付けるオプションが矛盾する場合は、以下の方法で解決されます。

変数名のリスト・オプション	特定の変数名に対する最後の明示的指定が使用されます。
デフォルトを変更するオプション	この形式は、名前リストを指定しません。指定された最後のオプションが、名前リスト・フォームに明示的にリストされていない変数のデフォルトになります。

関連情報
- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qtrigraph

説明

一部のキーボードにない文字を表すために使用される 3 文字表記のキーの組み合わせを認識するようコンパイラーに命令する。

構文

注

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。

3 文字表記の組み合わせは以下の通りです。

キーの組み合わせ	生成される文字
??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??!	|
??-	~

C デフォルトの **-qtrigraph** 設定は **-qnotrigraph** オプションを明示的にコマンド行に指定することによりオーバーライドすることができます。

コマンド行に **-qnotrigraph** を明示的に指定すると、**-qnotrigraph** がコマンド行のどこに指定されているかに関係なく、特定の **-qlanglvl** コンパイラー・オプションと関連付けられている **-qtrigraph** 設定より優先されます。

C++ 同じことが C++ プログラムでも言えます。

例

C プログラムをコンパイルするときに 3 文字表記の文字シーケンスを使用不可にするには、以下のように入力します。

```
xlc myprogram.c -qnotrigraph
```

1. C++ プログラムをコンパイルするときに 3 文字表記の文字シーケンスを使用不可にするには、以下のように入力します。

   ```
   xlc++ myprogram.C -qnotrigraph
   ```

関連情報

- 83 ページの『-qdigraph』
- 137 ページの『-qlanglvl』
- 入力を制御するオプション: 言語拡張のオプション

-qtune

説明

実行可能プログラムの最適化対象とするアーキテクチャー・システムを指定する。

構文

アーキテクチャーのサブオプションは以下の通りです。

auto	コンパイルに使用されているプラットフォーム用に最適化されたオブジェクト・コードを作成します。
ppc970	PowerPC 970 プロセッサー用に最適化されたオブジェクト・コードを作成します。
pwr3	POWER3 ハードウェア・プラットフォーム用に最適化されたオブジェクト・コードを作成します。
pwr4	POWER4 ハードウェア・プラットフォーム用に最適化されたオブジェクト・コードを作成します。
pwr5	POWER5 ハードウェア・プラットフォーム用に最適化されたオブジェクト・コードを作成します。
rs64b	RS64II プロセッサー用に最適化されたオブジェクト・コードを作成します。
rs64c	RS64III プロセッサー用に最適化されたオブジェクト・コードを作成します。

278 ページの『#pragma options』も参照してください。

デフォルト

-qtune オプションのデフォルト設定は、**-qarch** オプションの設定に応じて異なります。

- **-qtune** が **-qarch** なしで指定されると、コンパイラーは **-qarch** 設定を使用してコンパイル・モードを基に適切なデフォルトを指定します。
- **-qarch** が **-qtune** なしで指定されると、コンパイラー指定されたアーキテクチャーに対してデフォルトのチューニング・オプションを使用します。

特定の **-qarch** 設定でのデフォルトの **-qtune** 設定については、234 ページの『コンパイラー・モードとプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ』で説明されています。

注

-qtune=*suboption* は、 **-qarch=***suboption* と共に使用できます。

- **-qarch=***suboption* は、命令の生成対象のアーキテクチャーを指定します。
- **-qtune=***suboption* は、コードの最適化対象のターゲット・プラットフォームを指定します。

- 無効な **-qtune** オプションを有効な **-qarch** オプションに指定すると、**-qtune** が有効な **-qarch** オプションのデフォルトに設定されて、警告メッセージが発行されます。

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・プラットフォーム用に最適化されることを指定するには、以下のように入力します。

```
xlc++ -o testing myprogram.C -qtune=pwr3
```

関連情報

- 58 ページの『-qarch』
- 24 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』
- パフォーマンス最適化のオプション: プロセッサーおよびアーキテクチャー最適化のオプション
- 「*XL C/C++ プログラミング・ガイド*」の『アプリケーションの最適化』

-U

説明

コンパイラーまたは **-D***name* オプションによって定義された ID 名 を定義解除する。

構文

注

-U*name* 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***name* オプションによって定義された名前のみです。

#undef プリプロセッサー・ディレクティブを使用して、ソース・プログラムの中で ID 名を定義解除にすることもできます。

-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。

例

ご使用のオペレーティング・システムが名前 __unix を定義しているけれど、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で入力したくない場合を想定します。名前 __unix の定義が無効になるように 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  -U__unix
```

関連情報

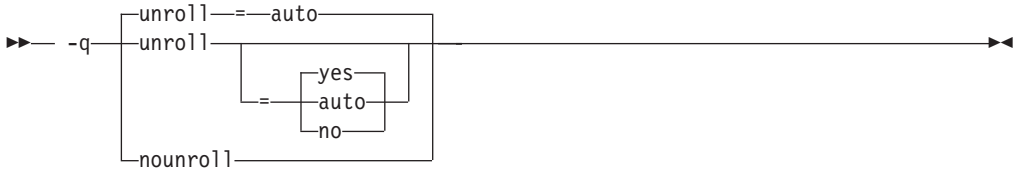
- 80 ページの『-D』
- コマンド行オプションの要約: その他の入力オプション

-qunroll

説明

プログラムの内部ループをアンロールする。これによりプログラムのパフォーマンスを改善することができます。

構文



ここで、

-qunroll=auto	ループをアンロールする決定をコンパイラーに任せます。これはコンパイラーのデフォルトです。
-qunroll または -qunroll=yes	ループをアンロールするようコンパイラーに提案します。
-qnounroll または -qunroll=no	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。

293 ページの『#pragma unroll』および 278 ページの『#pragma options』も参照してください。

注

このオプションのコンパイラー・デフォルトは、コマンド行に明示的に逆のことが指定されていない限り、**-qunroll=auto** です。

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。

-qunroll、**-qunroll=yes**、または **-qunroll=auto** が指定されていると、最適化プログラムにより、内部ループ本体がアンロールされるか、または複製されます。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。

unroll オプションによって、特定のアプリケーションのパフォーマンスが改善されるかどうかを確認するには、まず、通常オプションでプログラムをコンパイルしてから、それを代表的なワークロードで実行してください。次に、コマンド行 **-qunroll** オプションを指定するか、**unroll** プラグマを使用可能にして (あるいはその両方を行って)、プログラムを再コンパイルしてから、同じ条件下で再実行して、パフォーマンスが改善されたかどうか確認してください。

#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。このプラグマを設定すると、**-qunroll** コンパイラー・オプションの設定がオーバーライドされます。

例

1. 以下の例では、アンロールは使用不可です。

```
xlc++ -qnounroll file.C

xlc++ -qunroll=no file.C
```

2. 以下の例では、アンロールは使用可能です。

```
xlc++ -qunroll file.C

xlc++ -qunroll=yes file.C

xlc++ -qunroll=auto file.C
```

3. プログラム・コードがコンパイラーによってどのようにアンロールされるかの例については、 293 ページの『#pragma unroll』を参照してください。

関連情報

- パフォーマンス最適化のオプション: ループ最適化のオプション

-qunwind

説明

コンパイルのルーチンがアクティブである間、スタックをアンワインドできることをコンパイラーに通知する。

構文

注

-qnounwind を指定すると、不揮発性レジスターの保管と復元の最適化を改善できます。

C++ C++ プログラムの場合、**-qnounwind** を指定すると、**-qnoeh** も暗黙指定されます。

関連情報

- 88 ページの『-qeh』
- パフォーマンス最適化のオプション: ABI パフォーマンス・チューニングのオプション

-qupconv

C

説明

整数拡張を行うときに符号なしの指定を保持する。

構文

278 ページの『#pragma options』も参照してください。

注

-qupconv オプションは、intより小さい unsigned 型を int ではなく、unsigned int にプロモートします。

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。

デフォルト

デフォルトは **-qnoupconv** ですが、**-qlanglvl** が **classic** または **extended** に設定されているときは、デフォルトは **-qupconv** になります。コンパイラーは符号なしの指定を保存しません。

デフォルトのコンパイラー・アクションでは、整数拡張により、 char、short int、int ビット・フィールド、またはそれらのsigned または unsigned 型、あるいは enum 型が int に変換されます。それ以外の場合、型は unsigned int に変換されます。

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

- 137 ページの『-qlanglvl』
- コマンド行オプションの要約: 符号のオプション

-qutf

説明

UTF リテラル構文の認識を使用可能にする。

構文

注

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。

関連情報
- 入力を制御するオプション: 言語拡張のオプション
- 「*XL C/C++ 言語解説書*」の『UTF リテラル』

-V

説明
コンパイルの進行に関する情報、およびコンパイラー内で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報告するようにコンパイラーに命令する。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

注
-V オプションは、**-#** オプションによってオーバーライドされます。

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、呼び出しているプログラム、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
- 50 ページの『-# (ポンド記号)』
- 『-v』
- リストとメッセージを制御するオプション: メッセージのオプション

-v

説明
コンパイルの進行に関する情報、およびコンパイラー内で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報告するようにコンパイラーに命令する。情報はコンマで区切られたリストに表示されます。

構文

注
-v オプションは、**-#** オプションによってオーバーライドされます。

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
- 50 ページの『-# (ポンド記号)』
- 228 ページの『-V』
- リストとメッセージを制御するオプション: メッセージのオプション

-qversion

説明

呼び出しているコンパイラーのバージョンを表示する。正式なプロダクト名とシステムで検出されたコンパイラーのバージョンが出力されます。

構文

注

このオプションはコンパイラー・コマンドと共に単独で指定します。例:

```
xlC -qversion
```

関連情報
- リストとメッセージを制御するオプション: メッセージのオプション

-qvftable

C++

説明

仮想関数テーブルの生成を制御する。

構文

デフォルト

デフォルトでは、クラス・メンバー・リストで宣言されている最初の非インライン仮想メンバー関数の本体が現行のコンパイル単位に含まれる場合に、そのクラスに対する仮想関数テーブルが定義されます。

注

-qvftable を指定すると、現行のコンパイル単位で定義されている仮想関数を持つすべてのクラスについて仮想関数テーブルが生成されます。

-qnovftable を指定した場合、現行のコンパイル単位で仮想関数テーブルは生成されません。

例

仮想関数テーブルが生成されないようにファイル `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 -qnovftable
```

関連情報
- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-qvrsave

説明

VRSAVE レジスターを保守するための関数のプロローグとエピローグのコードを使用可能にする。

構文

ここで、

vrsave	コンパイル単位の関数のプロローグとエピローグに VRSAVE レジスターの保守に必要なコードが組み込まれます。
novrsave	コンパイル単位の関数のプロローグとエピローグに VRSAVE レジスターの保守に必要なコードが組み込まれません。

注

プログラム・ソース内の個々の関数に対するこのコンパイラー・オプションの現行設定をオーバーライドするには、**#pragma altivec_vrsave** を使用します。

関連情報
- 247 ページの『#pragma altivec_vrsave』
- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-W

説明

リストしたオプションを、指定したコンパイラー・コンポーネントに渡す。

構文

プログラムは以下の通りです。

プログラム	説明
a	アセンブラー

プログラム	説明
b	コンパイラーのバックエンド
c	コンパイラーのフロントエンド
I	プロシージャー間分析 (コンパイル・フェーズ)
L	プロシージャー間分析 (リンク・フェーズ)
l	リンケージ・エディター
p	コンパイラー・プリプロセッサー

注

-W オプションは、構成ファイルで使用される場合、パラメーター・ストリングでのコンマを表すための、エスケープ・シーケンスの円記号付きコンマ **(¥,)** を受け入れます。

例

1. *option* **-pg** をリンケージ・エディター **(l)** およびアセンブラー **(a)** に渡すように 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 -Wl,-pg -Wa,-pg
   ```

2. 構成ファイルでは、コンマ (,) を表すために **¥,** シーケンスを使用します。

   ```
   -Wl¥,-pg,-Wa¥,-pg
   ```

関連情報

- 13 ページの『コンパイラーの呼び出し』
- コンパイラーのカスタマイズのオプション: 一般のカスタマイズのオプション

-w

説明

警告および下位のメッセージを抑制するように要求する。このオプションを指定するのは、**-qflag=e:e** を指定するのと同等です。

構文

注

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。例えば、多重定義解決の問題が原因の重大エラーは通知メッセージも作成します。これらの通知メッセージは、 **-w** オプションを使用して使用不可になりません。

```
void func(int a){}
void func(int a, int b){}
int main(void)
{
func(1,2,3);
  return 0;
}
```

```
"x.cpp", line 6.4: 1540-0218 (S) The call does not match any parameter list for "func".
```

```
"x.cpp", line 1.6: 1540-1283 (I) "func(int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified for "func(int)".
"x.cpp", line 2.6: 1540-1283 (I) "func(int, int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified for "func(int, int)".
```

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

- 95 ページの『-qflag』
- リストとメッセージを制御するオプション: メッセージのオプション

-qwarn64

説明

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。

構文

注

生成されるメッセージは、すべて通知レベルです。

-qwarn64 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。

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
- ソース・ファイルとコマンド行でのプラグマ・オプション **arch** の矛盾

関連情報

- -q32、-q64
- 30 ページの『コンパイラー・メッセージ』
- エラー検査とデバッグのオプション: エラー検査のオプション

-qxcall

説明

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

注

-qxcall は、**-qnoxcall** よりも処理が遅いコードを生成します。

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
```

関連情報

- 出力を制御するオプション: オブジェクト・コードの特性を制御するオプション

-qxref

説明

すべての ID の相互参照リストを含むコンパイラー・リストを生成する。

構文

ここで、

noxref	プログラムにある ID を報告しません。
xref	使用されている ID のみを報告します。
xref=full	プログラムにある ID をすべて報告します。

278 ページの『#pragma options』も参照してください。

注

-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。

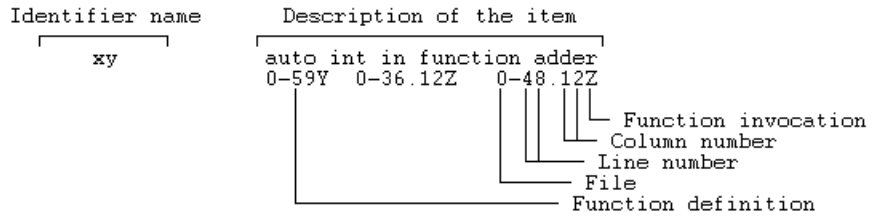
#pragma mc_func *function_name* ディレクティブに定義された関数は、いずれも **#pragma** ディレクティブの行に定義されているようにリストされます。

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 -qattr
```

一般的な相互参照リストの形式は、以下のとおりです。



関連情報
- 63 ページの『-qattr』
- 275 ページの『#pragma mc_func』
- リストとメッセージを制御するオプション: リストのオプション

-y

説明

浮動小数点定数式のコンパイル時丸めモードを指定する。

構文

サブオプションは以下の通りです。

n	表記可能な最近似値に丸める。これがデフォルトです。
m	負の無限大方向に丸める。
p	正の無限大方向に丸める。
z	ゼロ方向に丸める。

例

浮動小数点定数式をコンパイル時にゼロ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
```

関連情報
- 整数および浮動小数点処理を制御するオプション

コンパイラー・モードとプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ

-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。

すべての PowerPC マシンは、共通の命令セットを共用しますが、所定のプロセッサーまたはプロセッサー・ファミリーに固有の命令を追加して組み込むこともできます。

以下のテーブルは、選択された幾つかのプロセッサーとそれらがサポートする/サポートしないさまざまなフィーチャーを示したものです。

アーキテクチャー	グラフィックス・サポート	sqrt サポート	64 ビット・サポート	VMX サポート
rs64b	あり	あり	あり	なし
rs64c	あり	あり	あり	なし
pwr3	あり	あり	あり	なし
pwr4	あり	あり	あり	なし
pwr5	あり	あり	あり	なし
pwr5x	あり	あり	あり	なし
ppc	なし	なし	なし	なし
ppc64	なし	なし	あり	なし
ppc64gr	あり	なし	あり	なし
ppc64grsq	あり	あり	あり	なし
ppc64v	あり	あり	あり	あり
ppc970	あり	あり	あり	あり

さまざまなプロセッサー上で稼働するコードを生成したい場合は、以下のガイドラインを使用して、適切な **-qarch** および **-qtune** コンパイラー・オプション、またはそのいずれかを選択してください。以下をコンパイルに使用したコードの場合:

- **-qarch=pwr3** は、どの POWER3、POWER4、POWER5、POWER5+、および PowerPC 970 マシン上でも稼働する。
- **-qarch=pwr4** は、どの POWER4、POWER5、POWER5+、および PowerPC 970 マシン上でも稼働する。
- **-qarch=pwr5** は、POWER5、POWER5+、PowerPC 970 マシン上で稼働する。
- **-qarch=pwr5x** は、POWER5+ マシン上で稼働する。
- **-qarch=ppc** は、どの PowerPC システム上でも稼働する。

- **-qarch=ppcgr** は、グラフィックス・サポートがあるどの PowerPC システム上でも稼働する。
- **-qarch=ppc64** は、どの 64 ビット PowerPC システム上でも稼働する。
- **-qarch=ppc64gr** は、グラフィックス・サポートがあるどの 64 ビット PowerPC システム上でも稼働する。
- **-qarch=ppc64grsq** は、グラフィックス・サポートと平方根サポートのあるどの 64 ビット PowerPC システム上でも稼働する。
- **-qarch=ppc64v** は、VMX サポートのあるどの 64 ビット PowerPC システム上でも稼働する。
- **-q64** を使用したコードは、 64 ビットがサポートされている PowerPC マシン上でのみ稼働する。
- 特定のプロセッサーを参照するその他の **-qarch** オプションは、機能的に同等であればどの PowerPC マシンでも稼働する。例えば、以下の表は、**-qarch=pwr3** を指定してコンパイルされたコードが **rs64c** 上でも稼働することを示しています。。

特定のプロセッサー用に特に最適化されたコードを生成したい場合は、 **-q32**、**-q64**、**-qarch**、および **-qtune** コンパイラー・オプションの受け入れ可能な組み合わせを、以下のテーブルに示します。

表 40. 有効な **-qarch /-qtune** *の組み合わせ*

-qarch オプション	**事前定義マクロ**	**デフォルトの -qtune 設定**	**使用可能な -qtune 設定**
ppc	_ARCH_PPC	pwr4	auto pwr3 pwr4 pwr5 ppc970 rs64b rs64c
ppcgr	_ARCH_PPC _ARCH_PPCGR	pwr4	auto pwr3 pwr4 pwr5 ppc970 rs64b rs64c
ppc64	_ARCH_PPC _ARCH_PPC64	pwr4	auto pwr3 pwr4 pwr5 ppc970 rs64b rs64c
ppc64v	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PPC64GR _ARCH_PPC64GRSQ _ARCH_PPC64V	ppc970	auto ppc970
ppc64gr	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PPC64GR	pwr4	auto pwr3 pwr4 pwr5 ppc970 rs64b rs64c
ppc64grsq	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PPC64GR _ARCH_PPC64GRSQ	pwr4	auto pwr3 pwr4 pwr5 ppc970 rs64b rs64c
ppc970	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PWR3 _ARCH_PWR4 _ARCH_PPC970 _ARCH_PPC64GR _ARCH_PPC64GRSQ	ppc970	auto ppc970
rs64b	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_RS64B _ARCH_PPC64GR _ARCH_PPC64GRSQ	rs64b	auto rs64b
rs64c	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_RS64C _ARCH_PPC64GR _ARCH_PPC64GRSQ	rs64c	auto rs64c
pwr3	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PWR3 _ARCH_PPC64GR _ARCH_PPC64GRSQ	pwr3	auto pwr3 pwr4 pwr5 ppc970

表 40. 有効な **-qarch /-qtune** *の組み合わせ (続き)*

-qarch オプション	**事前定義マクロ**	**デフォルトの -qtune 設定**	**使用可能な -qtune 設定**
pwr4	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PWR3 _ARCH_PWR4 _ARCH_PPC64GR _ARCH_PPC64GRSQ	pwr4	auto pwr4 pwr5 ppc970
pwr5	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PWR3 _ARCH_PWR4 _ARCH_PWR5 _ARCH_PPC64GR _ARCH_PPC64GRSQ	pwr5	auto pwr5
pwr5x	_ARCH_PPC _ARCH_PPCGR _ARCH_PPC64 _ARCH_PWR3 _ARCH_PWR4 _ARCH_PWR5 _ARCH_PWR5X _ARCH_PPC64GR _ARCH_PPC64GRSQ	pwr5	auto pwr5

関連情報

- 24 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』
- 51 ページの『-q32、-q64』
- 58 ページの『-qarch』
- 223 ページの『-qtune』

第 4 章 glxc および glxc++ での GNU C/C++ コンパイラー・オプションの再使用

本章では、コンパイラー呼び出しユーティリティー **gxlc** および **gxlc++** の使用による XL C/C++ コンパイラーでの GNU コンパイラー・オプションの再使用方法について説明します。

gxlc および **gxlc++** ユーティリティーはそれぞれ GNU C または C++ コンパイラー・オプションを受け入れて、同等の XL C/C++ オプションに変換します。両ユーティリティーは XL C/C++ オプションを使用して **xlc** または **xlC** 呼び出しコマンドを作成し、それを使用してコンパイラーを呼び出します。これらのユーティリティーは、以前に GNU C/C++ を使用して開発されたアプリケーション用に作成された Make ファイルを再利用するために提供されています。しかし、XL C/C++ の機能を完全に活用するためには、XL C/C++ 呼び出しコマンドとそれらの関連オプションを使用することをお勧めします。

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

gxlc および **gxlc++** オプションのマッピングは変更可能です。**gxlc** および **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、 240 ページの『オプション・マッピングの構成』を参照してください。

例

GCC `-ansi` 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 -ansi hello.c
```

これは次のように変換されます。

```
xlc -F:c89 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。

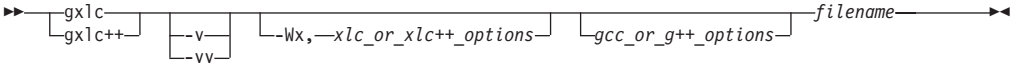
gxlc および gxlc++ の戻りコード

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

40 **gcc** または **g++** オプション・エラーまたは回復不能エラーが検出された。
255 プロセスの実行中にエラーが検出された。

glxc および glxc++ 構文

以下の図は **gxlc** および **gxlc++** 構文を示しています。



ここで、

filename
コンパイルするファイルの名前です。

-v XL C/C++ の呼び出しに使用されるコマンドの検証を許可します。**gxlc** または **gxlc++** は、作成した XL C/C++ 呼び出しコマンドを使用してコンパイラーを呼び出す前に、そのコマンドを表示します。

-vv シミュレーションの実行を許可します。**gxlc** または **gxlc++** は作成した XL C/C++ 呼び出しコマンドを表示しますが、コンパイラーを呼び出しません。

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

gcc_or_g++_options
xlc または **xlc++** オプションに変換される **gxlc** または **gxlc++** 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** または **gxlc++** によって認識される **gcc** および **g++** オプションは構成ファイル `gxlc.cfg` にリストされています。複数の *gcc_or_g++_options* はスペース文字で区切られます。

関連情報
- 『オプション・マッピングの構成』

オプション・マッピングの構成

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

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

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

以下の構文が gxlc.cfg 内の項目に使用されます。

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

ここで、

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

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

-fno-inline が指定されていると、**gxlc** はそれを -qnoinline に変換します。

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

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

そして、**gxlc** および **gxlc++** 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。値は以下の通りです。

- 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。**gxlc** および **gxlc++** 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は処理を停止するようユーティリティーに命令します。**gxlc** および **gxlc++** 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** および **gxlc++** によって無視されなければならない場合。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-"
```

gxlc および **gxlc++** はこのオプションを入力として検出すると、処理を行わず、警告を生成します。

d **gxlc** および **gxlc++** はコンパイラーのタイプを基にオプションを組み込んだり無視することができます。値は以下の通りです。

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

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

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

"gcc_or_g++_option"
: **gcc** or **g++** オプションを表すストリングです。

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

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

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

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

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

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

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

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

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

文字 `%` は GNU C または 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。これは、無視されるオプションと関連したパラメーターを、**gxlc** または **gxlc++** が無視することを保証するために使用されます。例えば、`-isystem` オプションはサポートされておらず、パラメーターを使用しています。両方ともアプリケーションによって無視されなければなりません。この場合、項目は以下のようになります。

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

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

http://www.ibm.com/support/search.wss?q=gxlcppoptionmaptabl&tc=SSJT9L&rs=2030&apar=include

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 5 章 コンパイラー・プラグマ参照

以下の節では、XL C/C++ (Linux プラットフォーム用) で使用可能なプラグマについて説明します。

- 『XL C/C++ プラグマの要約』
- 245 ページの『OpenMP プラグマ・ディレクティブの要約』
- 246 ページの『個々のプラグマの説明』

XL C/C++ プラグマの要約

以下にリストされたプラグマは、一般的なプログラミング使用に利用できます。

プラグマ	説明
#pragma align	構造体の中のデータ項目の位置を合わせる。
C #pragma alloca	alloca(size_t size) 関数のインライン・バージョンを提供する。
#pragma altivec_vrsave	VRSAVE レジスターを保守するために関数のプロローグとエピローグにコードを追加する。
#pragma block_loop	ループ・ネスト内の特定のループのブロッキング・ループを作成するようコンパイラーに命令する。
#pragma chars	文字データの sign 型を設定する。
#pragma comment	オブジェクト・ファイルにコメントを入れる。
#pragma complexgcc	関数を呼び出すときの複合タイプのパラメーターの受け渡し方法をコンパイラーに命令する。
#pragma STDC cx_limited_range	コンパイラーが制御するスコープ内では、 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だけ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。
C++ #pragma define	クラスのオブジェクトを実際に定義することなく、テンプレート・クラスの定義を強制的に行う。
#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。
C++ #pragma do_not_instantiate	指定されたテンプレート宣言のインスタンス化を抑制する。
#pragma enum	後続の enum 変数のサイズを指定する。
#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。
C++ #pragma hashome	指定されたクラスに **IsHome** プラグマで指定されるホーム・モジュールがあることをコンパイラーに通知する。
#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ユーザーが定義できるようにする。

プラグマ	説明
C++ #pragma implementation	プラグマが入っている組み込みファイル内のテンプレート宣言に対応する関数テンプレート定義を含むファイルの名前を、コンパイラーに伝える。
#pragma info	**info(...)** コンパイラー・オプションによって生成された診断メッセージを制御する。
C++ #pragma instantiate	指定されたテンプレート宣言の即時インスタンス化を実行する。
C++ #pragma ishome	指定されたクラスのホーム・モジュールが現行のコンパイル単位であることをコンパイラーに通知する。
#pragma isolated_call	パラメーターによって暗黙指定されている関数を除き、副次作用を持たなかったり副次作用に依存しない関数をマークする。
#pragma langlvl	コンパイルのために C または C++ 言語レベルを選択する。
#pragma leaves	関数名を取り、関数がその関数の呼び出し後に命令に戻らないことを指定する。
#pragma loop_id	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 map	ID への参照がすべて新しい名前に変換されることをコンパイラーに通知する。
#pragma mc_func	マシン・インストラクションの短いシーケンスを含む関数を定義できるようにする。
#pragma nosimd	このディレクティブのすぐ後に続くループで VMX (Vector Multimedia Extension) 命令を生成しない ようコンパイラーに命令する。
#pragma novector	次のループを自動ベクトル化しない ようコンパイラーに命令する。
#pragma options	使用しているソース・プログラムのコンパイラーへのオプションを指定する。
#pragma option_override	指定された関数に対して代替最適化オプションを指定する。
#pragma pack	このプラグマの後に続く構造体のメンバーに対する現行の位置合わせ規則を変更する。
C++ #pragma priority	静的オブジェクトが初期化される順序を指定する。
#pragma reachable	到達可能とマークされたルーチンへの呼び出し後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宣言する。
#pragma reg_killed_by	指定された関数によって値が破壊されるレジスターを指定する。これは **#pragma mc_func** と一緒に使用する必要があります。
C++ #pragma report	特定のメッセージの生成を管理する。
#pragma stream_unroll	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 strings	ストリングのストレージ・タイプを設定する。
#pragma unroll	プログラムの最内部と最外部のループをアンロールする。これによりプログラムのパフォーマンスを改善することができます。

プラグマ	説明
#pragma unrollandfuse	ネストされた for ループでアンロールおよびフューズ操作を試行するようコンパイラーに命令する。これによりプログラムのパフォーマンスを改善することができます。
#pragma weak	リンク・エディターがシンボルの定義を見つけなかった場合、またはリンク時に複数定義されたシンボルが見つかった場合に、リンク・エディターがエラー・メッセージを出さないようにする。

OpenMP プラグマ・ディレクティブの要約

このページに要約されているプラグマ・ディレクティブは、コンパイラーがプログラムの中で並列処理をどのように扱うかを制御します。

ディレクティブは、このディレクティブの直後に続くステートメントまたはステートメント・ブロックに対してのみ適用されます。

OpenMP プラグマ・ディレクティブ	説明
#pragma omp atomic	アトミックに更新されなければならず、しかも、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位置を識別する。
#pragma omp parallel	複数のスレッドによって並列で実行されるように並列領域を定義する。特定の例外はありますが、他のすべての OpenMP ディレクティブは、このディレクティブで定義された並列領域内で実行されます。
#pragma omp for	反復が並列で実行される反復 for ループを識別する作業共有構成。
#pragma omp parallel for	**omp parallel** と **omp for** プラグマ・ディレクティブのショートカット組み合わせで単一の **for** ディレクティブを含む並列領域の定義に使用される。
#pragma omp ordered	順番に実行しなければならないコードの構造化ブロックを識別する作業共有構成。
#pragma omp section、#pragma omp sections	並列で実行する必要があるコードの 1 つ以上のサブセクションを含む、コードの非反復セクションを識別する作業共有構成。
#pragma omp parallel sections	**omp parallel** と **omp sections** プラグマ・ディレクティブのショートカット組み合わせで単一の **sections** ディレクティブを含む並列領域の定義に使用される。
#pragma omp single	単一の使用可能スレッドによって実行しなければならないコードのセクションを識別する作業共有構成。
#pragma omp master	マスター・スレッドでしか実行してはならないコードのセクションを識別する同期構成。
#pragma omp critical	単一スレッドによって一度に実行しなければならないステートメント・ブロックを識別する同期構成。
#pragma omp barrier	並列領域内のすべてのスレッドを同期化する。

OpenMP プラグマ・ディレクティブ	説明
#pragma omp flush	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する同期構成体。
#pragma omp threadprivate	選択したファイル・スコープ・データ変数のスコープがスレッドに対して private であるが、そのスレッド内でファイル・スコープが可視であると定義する。

個々のプラグマの説明

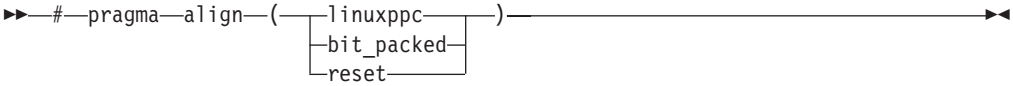
この節には XL C/C++ で使用可能な個々のプラグマの説明があります。

#pragma align

説明

#pragma align ディレクティブは、コンパイラーが構造体内でデータ項目を位置合わせする方法を指定します。

構文



278 ページの『#pragma options』も参照してください。

注

#pragma align(*suboption***)** 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。

コンパイラーは、位置合わせディレクティブをスタックします。このため、**#pragma align(reset)** ディレクティブを指定することによって、直前の位置合わせディレクティブの内容が不明であっても、その規則に戻して使用することができます。

例えば、組み込みファイル内にクラス宣言があり、そのクラスに対して指定した位置合わせ規則をクラスの組み込み先に適用したくない場合に、このオプションを使用することができます。ソース・ファイルで **#pragma align(reset)** をコーディングして、最後に指定した位置合わせオプションの前の指定内容に位置合わせオプションを変更することができます。直前の位置合わせ規則がファイルにない場合は、呼び出しコマンドで指定した位置合わせ規則が使用されます。

#pragma align の指定は、ソース・コードに **#pragma options align** を指定したのと同じ効果を持ちます。例えば、**#pragma align** と **#pragma options align** の使用法に関する詳細情報と例については、55 ページの『-qalign』を参照してください。

関連情報

- 「*XL C/C++ プログラミング・ガイド*」の
- 「*XL C/C++ 言語解説書*」のおよび『パックされた変数属性』

#pragma alloca

C

説明

#pragma alloca ディレクティブは、コンパイラーが alloca(size_t*size*) 関数のインライン・バージョンを提供することを指定します。alloca(size_t*size*) 関数は、オブジェクトのスペースを割り振るために使用することができます。割り振られるスペースの量は、*size* (バイト単位で測定) の値で決定されます。割り振られた空間はスタックに入れられます。

構文

注

コンパイラーに alloca のインライン・バージョンを提供させるためには、**#pragma alloca** ディレクティブ、または **-qalloca** コンパイラー・オプションを指定する必要があります。

一度指定されると、**#pragma alloca** はファイル全体に適用され、オフにすることはできません。**#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。

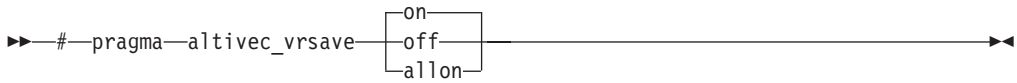
関連情報
- 56 ページの『-qalloca』

#pragma altivec_vrsave

説明

#pragma altivec_vrsave ディレクティブが使用可能の場合、関数のプロローグとエピローグに VRSAVE レジスターを保守するためのコードが組み込まれます。

構文



プラグマ設定は以下のことを行います。

on　関数のプロローグとエピローグに VRSAVE レジスターを保守するためのコードが組み込まれます。

off　関数のプロローグとエピローグに VRSAVE レジスターを保守するためのコードが組み込まれません。

allon　**altivec_vrsave** プラグマを含む関数は、VRSAVE レジスターのすべてのビットを 1 に設定することにより、すべてのベクトルが使用され、コンテキストの切り替えが発生した場合には保管されることを示します。

注

- **#pragma altivec_vrsave** は、**-qaltivec** オプションが有効な場合にのみサポートされます。
- VRSAVE レジスターのそれぞれのビットはベクトル・レジスターに対応し、1 に設定されている場合は、コンテキスト・スイッチが発生したときに保管されるデータが対応するベクトルに含まれることを示します。デフォルトの振る舞いでは、vrsave レジスターは常に保守されます。
- このプラグマは 1 つの関数内でのみ使用でき、その効果はそのプラグマが現れる関数に対してのみ適用されます。同じ関数の中で異なる設定を使用してこのプラグマを指定した場合は、エラー条件が作成されます。

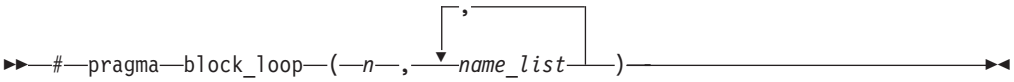
関連情報
- 57 ページの『-qaltivec』
- 230 ページの『-qvrsave』

#pragma block_loop

説明

スコープ固有の ID を使用してブロックにマークを付けます。

構文



ここで、

n 反復グループのサイズの整数式です。

name_list **#pragma loopid** ディレクティブを使用して作成できる固有 ID です。*name_list* を指定しないと、`#pragma block_loop` ディレクティブの後の最初の **for** ループ、または block_loop でブロッキングが発生します。

name は、スコープ単位内で固有の ID です。

注

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 **unroll**、**unroll_and_fuse**、または **stream_unroll** ディレクティブを指定すると、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、アンロールおよびフューズ、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 **unroll_and_fuse** または **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 **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。

#pragma block_loop を複数回指定したり、同じ for ループに対してこのディレクティブを **nounroll**、**unroll**、**nounrollandfuse**、**unrollandfuse**、または **stream_unroll** プラグマ・ディレクティブと結合してはいけません。また、単一のブロック・ループ・ディレクティブに複数の **unroll** ディレクティブを適用しないでください。

すべての **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。

ディレクティブの正確な使用例

例 1 - ループのタイル表示

```
#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++)
      {
         ...
      }
    }
  }
```

例 2 - ループのタイル表示

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
 for (i=0; i < n; n++)
 {
#pragma loopid(myfirdstloop)
  for (j=0; j < m; j++)
  {
#pragma loopid(mysecondloop)
   for (k=0; k < m; k++)
   {
    ...
   }
  }
 }
```

例 3 - ループの交換

```
 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++)
    {
     ...
    }
   }
  }
 }
```

例 4 - マルチレベル・メモリー階層のためのループのタイル表示

```
#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++)
        {
          ...
        }
      }
    }
  }
```

例 5 - ブロッキング・ループのアンロールとフューズ

```
#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持ちません。

例 6 - ブロックされたループのアンロール

```
#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、その発生後、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。

ディレクティブの誤った使用例

例 1 - 未定義ループ ID の Block_loop

```
#pragma block_loop(50, myloop)
  for (i=0; i < n; i++)
  {
  }

Referencing myloop is not allowed, since it is not in the nest
and may not be defined.
```

例 2 - 同一ループ・ネスト内にないループ ID の Block_loop

```
  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++)
  {
```

```
    ...
  }

 Referencing myloop is not allowed, since it is defined in a different
loop nest (nesting structure).
```

例 3 - ブロッキング・ループに指定された矛盾するアンロール・ディレクティブ

```
#pragma unrollandfuse(5)
#pragma unroll(2)
#pragma block_loop(10)
 for (i = 0; i < N; ++i) {
   }

This is not allowed since the unroll directives are conflicting
with each other.
```

例 4 - ブロックされたループに指定された矛盾するアンロール・ディレクティブ

```
#pragma block_loop(10)
#pragma unroll(5)
#pragma unroll(10)
  for (i = 0; i < N; ++i) {
  }

This is not allowed since there are two different unrolling factors
specified for the same loop, and therefore the diretives are conflicting.
```

関連情報

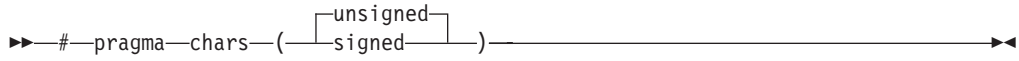
- 271 ページの『#pragma loop_id』
- 225 ページの『-qunroll』
- 293 ページの『#pragma unroll』
- 294 ページの『#pragma unrollandfuse』
- 291 ページの『#pragma stream_unroll』

#pragma chars

説明

#pragma chars ディレクティブは、char オブジェクトの符号の型を `signed` または `unsigned` のいずれかに設定します。

構文



注

有効にするためには、このプラグマがどのソース・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。

このプラグマは、一度指定するとファイル全体に適用され、オフにすることはできません。 **#pragma chars** 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。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。

注: デフォルトの character 型は、unsigned char と同様の振る舞いをします。

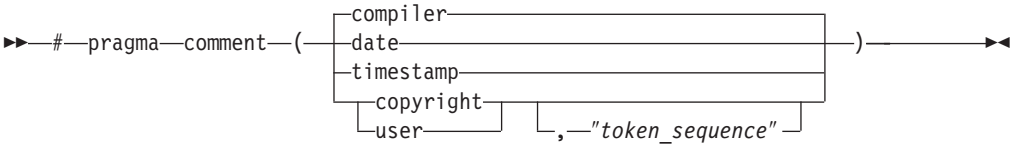
関連情報
- 70 ページの『-qchars』

#pragma comment

説明

#pragma comment ディレクティブはターゲットまたはオブジェクト・ファイルにコメント・ストリングを配置します。

構文



サブオプションは以下のことを行います。

compiler　コンパイラーの名前とバージョンが、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。

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　ソースを最後に変更した日付と時刻が、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。

copyright　*token_sequence* によって指定されたテキストが、生成されたオブジェクト・モジュール内にコンパイラーによって入れられ、プログラム実行時に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、生成されたオブジェクト内にコンパイラーによって入れられますが、プログラム実行時にメモリーにロードされません。

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;
}
```

オペレーティング・システムの **strings** コマンドを使用して、オブジェクトまたはバイナリー・ファイルでこれらのストリングやその他のストリングを検索することができます。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 10:28:09 2005
XL C/C++ for Linux Compiler Version 8.0
Mon Mar 1 10:28:13 2005
My copyright
```

注:

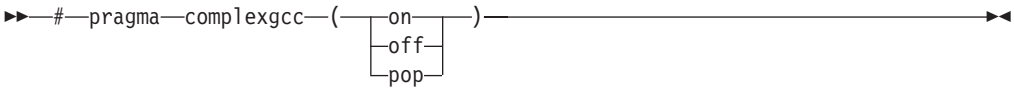
token_sequence に指定されたストリング・リテラルが 32767 バイトを超えると、通知メッセージが出され、プラグマが無視されます。

#pragma complexgcc

説明

#pragma complexgcc ディレクティブは、複合タイプのパラメーターのやり取りの方法をコンパイラーに命令する。

構文



サブオプションは以下のことを行います。

on　**-qfloat=complexgcc** をスタックにプッシュします。これは、汎用目的のレジスターを使用して、複合タイプのパラメーターのやり取りに、GCC 規則を使用するようコンパイラーに命令します。

off　**-qfloat=nocomplexgcc** をスタックにプッシュします。これは、浮動小数点レジスターを使用して、複合タイプのパラメーターのやり取りに AIX 規則を使用するようコンパイラーに命令します。

pop　スタックから現在の設定を除去し、直前の設定を復元します。スタックが空のときは、コンパイラーは **-qfloat=[no]complexgcc** 設定がコマンド行に指定されていることを想定し、指定されていないと、**-qfloat=[no]complexgcc** にはコンパイラーのデフォルトが使用されます。

注

このプラグマの現行設定は、この設定が有効な間に宣言または定義された関数だけに影響します。その他の関数には影響しません。

関数に対するポインターから関数を呼び出すと、**-qfloat=[no]complexgcc** コンパイラー・オプションによって、必ず規則設定が使用されます。コンパイラーを呼び出すときに、このオプションがコマンド行に明示的に設定されていないと、このオプションについてのコンパイラーのデフォルトが使用されます。複合値を渡す関数を値またはリターン複合値でミックス・アンド・マッチさせると、エラーが発生します。

例えば以下のコードが **-qfloat=nocomplexgcc** でコンパイルされるとします。

```
#pragma complexgcc(on)
void p (_Complex double x) {}
```

```
#pragma complexgcc(pop)
typedef void (*fcnptr) (_Complex double);

int main() {
    fcnptr ptr = p; /* error: function pointer is -qfloat=nocomplexgcc;
                              function is -qfloat=complexgcc */
}
```

関連情報

- 74 ページの『-qcomplexgccincl』
- 96 ページの『-qfloat』

#pragma define

C++

説明

#pragma define ディレクティブは、クラスのオブジェクトを実際に定義することなく、テンプレート・クラスの定義を強制的に行います。このプラグマは、後方互換性のためだけに提供されています。

構文

ここで、*template_classname* は、定義するテンプレートの名前です。

注

ユーザーは以下のフォームの構造体を使用して、クラス、関数、またはメンバー・テンプレート特殊化のインスタンスを明示的に生成することができます。

```
template declaration
```

例を以下に示します。

```
#pragma define(Array<char>)
```

は、以下と同等です。

```
template class Array<char>;
```

このプラグマはネームスペース・スコープ内に定義しなければなりません (つまり、関数/クラス本体に含めることはできません)。これはテンプレート関数の効率的または自動的生成のためにプログラムを編成するときに使用されます。

関連情報

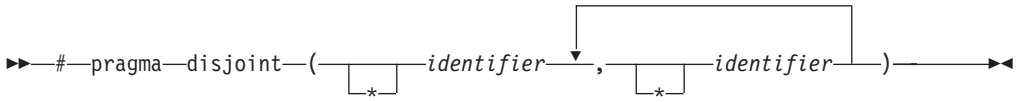
- 255 ページの『#pragma do_not_instantiate』
- 266 ページの『#pragma instantiate』

#pragma disjoint

説明

#pragma disjoint ディレクティブは、その使用スコープ内で互いに別名ではない ID をリストします。

構文



注

このディレクティブは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します。これにより最適化のための機会がより多く提供されます。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、このプラグマが原因でプログラムが間違った結果を出す可能性があります。

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。 disjoint 名リスト内の ID 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
- 型定義名
- ラベル

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

```
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
void one_function()
{
     b = 6;
     *ptr_a = 7; // Assignment does not alter the value of b
     another_function(b); // Argument "b" has the value 6
}
```

外部ポインター `ptr_a` は、外部関数 `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ないため、`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` とメモリー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。

関連情報

- 113 ページの『-qignprag』
- 53 ページの『-qalias』

#pragma do_not_instantiate

C++

説明

#pragma do_not_instantiate ディレクティブは、指定されたテンプレート宣言のインスタンスを生成しない ようコンパイラーに命令します。

構文

ここで、*template* はクラス・テンプレート ID です。例えば、以下のようになります。

```
#pragma do_not_instantiate Stack < int >
```

注

このプラグマを使用して、定義が提供されているテンプレートの暗黙のインスタンス生成を抑制します。

テンプレートのインスタンス生成を手動で処理しており (つまり、**-qnotempinc** と **-qnotemplateregistry** が指定されている)、指定されたテンプレートのインスタンス生成がすでに別のコンパイル単位に存在している場合は、**#pragma do_not_instantiate** を使用すると、リンク・エディット・ステップ中に複数のシンボル定義を取得しないことが保証されます。

関連情報

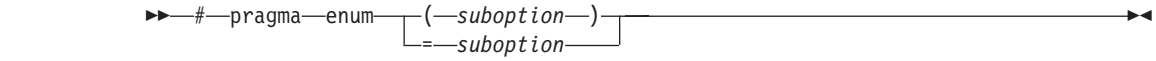
- 254 ページの『#pragma define』
- 266 ページの『#pragma instantiate』
- 214 ページの『-qtempinc』
- 215 ページの『-qtemplateregistry』

#pragma enum

説明

#pragma enum ディレクティブは、後続の enum 変数のサイズを指定します。宣言の左中括弧のサイズは、宣言内でさらに **enum** ディレクティブが出現するかどうかに関係なく、宣言に影響を与えます。このプラグマは、使用されるたびに値をスタックにプッシュし、リセット・オプションを使用すれば、直前にプッシュした値に戻ることができます。

構文



ここで、*suboption* は次のいずれかになります。

1	列挙型は 1 バイト長で、列挙型の値の範囲が signed char の限度内の場合は char 型、そうでない場合は unsigned char 型です。
2	列挙型は 2 バイト長で、列挙型の値の範囲が signed short の限度内の場合は short 型、そうでない場合は unsigned short 型です。
4	列挙型は 4 バイトの長さで、列挙型の値の範囲が signed int の限度内の場合は int 型、そうでない場合は unsigned int 型となります。

8	列挙型は 8 バイトの長さです。 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。
int	**#pragma enum=4** と同じ。
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。 256 ページの『#pragma enum』の説明を参照してください。 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	列挙型は、すべての変数を含めることができる最小の整数型です。 8 バイトの列挙型の結果の場合、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。 256 ページの『#pragma enum』の説明を参照してください。
pop	このサブオプションは、列挙型のサイズ設定を直前の **#pragma enum** 設定にリセットします。直前の設定がない場合は、**-qenum** のコマンド行設定が使用されます。
reset	**pop** と同じ。このオプションは、後方互換性のために用意されています。

注

空のスタックをポップすると、警告メッセージが生成され、`enum` 値は未変更のままとなります。

#pragma enum ディレクティブは、**-qenum** コンパイラー・オプションをオーバーライドします。

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。これは、1 つのファイルがそのファイルを **#include** する別のファイルの **enum** 設定を変更してしまう可能性を回避するための唯一の方法です。

#pragma options enum ディレクティブは、 **#pragma enum** の代わりに使用することができます。この 2 つのプラグマは交換可能です。

#pragma enum=reset ディレクティブに対応する **-qenum=reset** オプションは存在しません。**-qenum=reset** の使用を試行すると、警告メッセージが生成されて、オプションが無視されます。

例

1. 以下のコード・セグメントに **pop** および **reset** サブオプションの使用法を示します。

```
#pragma enum(1)
#pragma enum(2)
#pragma enum(4)
#pragma enum(pop)    /* will reset enum size to 2       */
#pragma enum(reset)  /* will reset enum size to 1       */
#pragma enum(pop)    /* will reset enum size to the -qenum setting,
```

```
                                assuming -qenum was specified on the command
                                line.  If -qenum was not specified on the
                                command line, the compiler default is used. */
```

2. **reset** サブオプションは、一般に、メインファイルのデフォルトと異なる列挙型のストレージを指定する組み込みファイルの最後で、設定された列挙型のサイズをリセットするために使用します。例えば、以下の組み込みファイル small_enum.h はさまざまな最小サイズの列挙を宣言し、組み込みファイルの最後に、その指定をオプション・スタックの最後の値にリセットしています。

```
#ifndef small_enum_h
#define small_enum_h 1
/*
 * File small_enum.h
 * This enum must fit within an unsigned char type
 */

#pragma options enum=small
enum e_tag {a, b=255};
enum e_tag u_char_e_var; /* occupies 1 byte of storage */


/* Reset the enumeration size to whatever it was before */
#pragma options enum=reset
#endif
```

以下のソース・ファイル int_file.c には、small_enum.h が組み込まれています。

```
/*
 * File int_file.c
 * Defines 4 byte enums
 */
#pragma options enum=int
enum testing {ONE, TWO, THREE};
enum testing test_enum;

/* various minimum-sized enums are declared */
#include "small_enum.h"

/* return to int-sized enums. small_enum.h has reset the
 * enum size
 */
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;
```

列挙型 test_enum と first_order の両方ともストレージの 4 バイトを占有し、int 型です。small_enum.h に定義された変数 u_char_e_var は 1 バイトのストレージを占有し、unsigned char データ型で表されます。

3. 以下の C コード・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。

4. 以下の C コード・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。

5. 以下のコマンドを使用してファイル myprogram.C をコンパイルした場合:

```
xlc++ myprogram.C -qenum=small
```

ファイル myprogram.C に **#pragma options=int** ステートメントが含まれていないと想定した場合、ソース・ファイル内のすべての enum 変数が占有するストレージの量は最小になります。

6. 以下の行が含まれたファイル yourfile.C をコンパイルする場合、

```
enum testing {ONE, TWO, THREE};
enum testing test_enum;
#pragma options enum=small
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;

#pragma options enum=int
enum music {ROCK, JAZZ, NEW_WAVE, CLASSICAL};
enum music listening_type;
```

以下のコマンドを使用すると、

```
xlc++ yourfile.C
```

enum 変数 first_order だけが最小サイズになります (つまり、enum 変数 first_order は 1 バイトのストレージのみを占有します)。他の 2 つの enum 変数 test_enum と listening_type は int 型となり、4 バイトのストレージを占有します。

以下の例は、無効な列挙型または **#pragma enum** の使用を示したものです。

- enum の宣言内で **#pragma enum** を使用して enum のストレージ割り振りを変更することはできません。以下のコード・セグメントは警告を生成し、 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
```

- enum 定数の範囲が unsigned int の範囲内ではありません。

```
#pragma options enum=small
enum e_tag { a=0,
             b=4294967296 /* error: larger than maximum int */
           } e_var;
#pragma options enum=reset
```

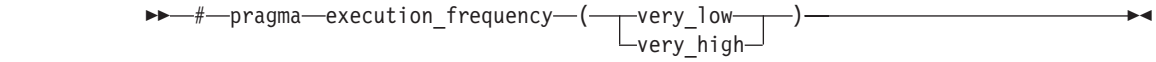
関連情報
- 89 ページの『-qenum』
- 278 ページの『#pragma options』

#pragma execution_frequency

説明

#pragma execution_frequency ディレクティブを使用すると、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ことができます。

構文



注

このプラグマは、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ために使用します。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。

このプラグマは、最適化プログラムに対するヒントとして使用します。最適化を選択していない場合、このプラグマは有効ではありません。

例

1. このプラグマは、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します。

   ```
   int *array = (int *) malloc(10000);

   if (array == NULL) {
       /* Block A */
       #pragma execution_frequency(very_low)
       error();
   }
   ```

   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();
   }
   ```

2. このプラグマは、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.   */
```

3. このプラグマは、ファイル・スコープでは使用できません。プラグマは、ブロック・スコープ内の任意の場所に配置することができ、最も近い分岐に影響します。

```
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)
```

1 番目と 4 番目のプラグマは無効ですが、2 番目と 3 番目は有効です。ただし、3 番目のプラグマだけが効果を持ち、`if (boo)` の決定の間にプログラム実行がブロック A に分岐するのか、またはブロック B に分岐するのかに影響します。2 番目のプラグマは、コンパイラーによって無視されます。

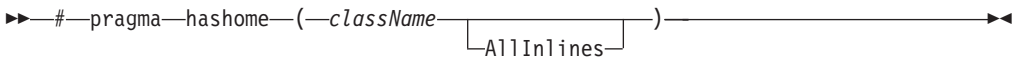
#pragma hashome

C++

説明

#pragma hashome ディレクティブは、**#pragma ishome** で指定されるホーム・モジュールが指定されたクラスにあることをコンパイラーに通知します。このクラスの仮想関数テーブルは、特定のインライン関数と共に、静的として生成されません。代わりにこれらは、**#pragma ishome** が指定されたクラスのコンパイル単位の中で外部として参照されます。

構文



ここで、

className 前述の外部参照を必要とするクラスの名前を指定します。 *className* はクラスでなければならず、定義されていなければなりません。

AllInlines *className* 内から、すべてのインライン関数を外部参照する必要があることを指定します。この引数では大/小文字の区別はありません。

注

一致する **#pragma hashome** を持たない **#pragma ishome** がある場合は、警告が出されます。

例

以下の例では、コード・サンプルをコンパイルすると仮想関数テーブルと、コンパイル単位 `a.o` のみの `S::foo()` の定義が生成されます (`b.o` 用はありません)。これによりアプリケーションに対して生成されるコードの量が削減されます。

```
// a.h
struct S
{
   virtual void foo() {}

   virtual void bar();
};



// a.C
#pragma ishome(S)
#pragma hashome (S)

#include "a.h"

int main()
{
   S s;
   s.foo();
   s.bar();
}



// b.C
#pragma hashome(S)
#include "a.h"

void S::bar() {}
```

関連情報

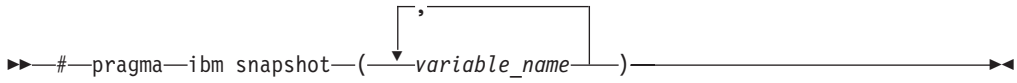
- 267 ページの『#pragma ishome』

#pragma ibm snapshot

説明

#pragma ibm snapshot を使用すると、ユーザーは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達したときに検査できる変数のリストを定義することができます。

構文



ここで、*variable_name* は変数の集合です。クラス、構造体、または共用体のメンバーは指定できません。

注

このプラグマは XL C/C++ コンパイラーに作成された最適化コードのデバッグを可能にするために提供されています。デバッグ・セッションの間、この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。プログラムが最適化付きで、オプション **-g** を含めてコンパイルされた場合、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。

スナップショットは、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。

#pragma ibm snapshot 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)
```

プログラムのこのポイントにデバッガーを通じてブレークポイントが設定されると、変数 `a`、`b`、および `c` の値が可視となります。

関連情報

- 104 ページの『-g』
- 166 ページの『-O、-qoptimize』

#pragma implementation

C++

説明

#pragma implementation ディレクティブは、関数テンプレート定義を含むテンプレート・インスタンス化ファイルの名前をコンパイラーに通知します。これらの定義はプラグマを含んでいる組み込みファイル内のテンプレート宣言に対応します。

構文

注

このプラグマは宣言が許可されているどこにでも現れることができます。これはテンプレート関数の効率的または自動的生成のためにプログラムを編成するときに使用されます。

関連情報

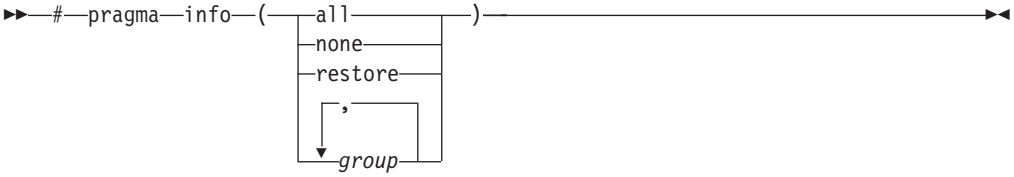
- 216 ページの『-qtempmax』

#pragma info

説明

#pragma info ディレクティブは、特定のグループのコンパイラー・メッセージを作成または抑制するようコンパイラーに命令します。

構文



ここで、

all　すべての診断検査をオンにします。
none　プログラムの特定部分に対するすべての診断サブオプションをオフにします。
restore　直前の **#pragma info** ディレクティブの前に有効だったオプションを復元します。

group 指定した診断グループ に関連したすべてのメッセージを生成または抑制します。以下のリストにある複数のグループ 名を指定できます。

グループ	戻される (抑制される) 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。
cls|nocls	C++ クラス。
cmp|nocmp	符号なしの比較において起こりうる冗長。
cnd|nocnd	条件式において起こりうる冗長または問題。
cns|nocns	定数が関係する演算。
cnv|nocnv	型変換。
dcl|nodcl	宣言の整合性。
eff|noeff	無効なステートメントおよびプラグマ。
enu|noenu	enum 変数の整合性。
ext|noext	未使用の外部定義。
gen|nogen	汎用診断メッセージ。
gnr|nognr	一時変数の生成。
got|nogot	goto 文の使用。
ini|noini	初期化で起こりうる問題。
inl|noinl	関数がインライン化されていない。
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。
vft|novft	C++ プログラムでの仮想関数テーブルの生成。
zea|nozea	ゼロ・エクステントの配列。

注

#pragma info ディレクティブを使用すると、コマンド行、構成ファイル、または以前の **#pragma info** ディレクティブの呼び出しに指定した、現行の **-qinfo** コンパイラー・オプション設定を一時的にオーバーライドすることができます。

例

例えば、下記のコード・セグメントで、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)** ディレクティブが呼び出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()
{
  .
  .
  .

}
```

関連情報

- 114 ページの『-qinfo』

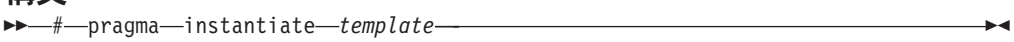
#pragma instantiate

C++

説明

#pragma instantiate ディレクティブは、指定されたテンプレート宣言のインスタンスをすぐに生成するように、コンパイラーに命令します。

構文



ここで、*template* はクラス・テンプレート ID です。例を以下に示します。

```
#pragma instantiate Stack < int >
```

注

既存のコードをマイグレーションする場合は、このプラグマを使用してください。新規コードは標準 C++ の明示的インスタンス生成を使用します。

テンプレートのインスタンス生成を手動で処理している (つまり、**-qnotempinc** と **-qnotemplateregistry** が指定されている) 場合は、**#pragma instantiate** を使用すると、指定されたテンプレートのインスタンス生成がコンパイル単位に表示されることが保証されます。

関連情報
- 254 ページの『#pragma define』
- 255 ページの『#pragma do_not_instantiate』
- 214 ページの『-qtempinc』
- 215 ページの『-qtemplateregistry』

#pragma ishome

C++

説明

#pragma ishome ディレクティブは、指定したクラスのホーム・モジュールが現行のコンパイル単位であることをコンパイラーに通知します。ホーム・モジュールは、仮想関数テーブルなどの項目が保管されている場所です。項目は、コンパイル単位の外側から参照されると、そのホームの外部には生成されません。これによりアプリケーションに対して生成されるコードの量を削減することができます。

構文

ここで、

className ホームが現行のコンパイル単位になるクラスのリテラル名です。

注

一致する **#pragma hashome** を持たない **#pragma ishome** がある場合は、警告が出されます。

例

以下の例では、コード・サンプルをコンパイルすると仮想関数テーブルと、コンパイル単位 `a.o` のみの `S::foo()` の定義が生成されます (`b.o` 用はありません)。これによりアプリケーションに対して生成されるコードの量が削減されます。

```
// a.h
struct S
{
   virtual void foo() {}

   virtual void bar();
};



// a.C
#pragma ishome(S)
#pragma hashome (S)

#include "a.h"
```

```
int main()
{
   S s;
   s.foo();
   s.bar();
}



// b.C
#pragma hashome(S)
#include "a.h"

void S::bar() {}
```

関連情報

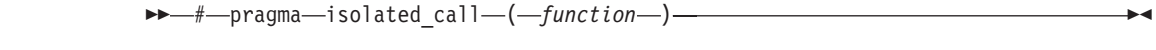
- 261 ページの『#pragma hashome』

#pragma isolated_call

説明

#pragma isolated_call ディレクティブは、パラメーターによって暗黙指定されるものを除き、副次作用を持たない、または副次作用に依存しない関数にマークを付けます。

構文



function は、1 次式であり、ID、演算子関数、変換関数、限定名のいずれかです。ID は、型関数または typedef 関数でなければなりません。名前により多重定義関数を参照する場合、この関数のすべての可変部は、孤立した呼び出しとしてマークされています。

注

-qisolated_call コンパイラー・オプションには、このプラグマと同じ効果があります。

このプラグマは、リストされた関数がそのパラメーターによって暗黙指定された以外の副次作用を持っておらず、また依存もしていないことをコンパイラーに通知します。以下のような場合、関数は副次作用を持つか、それに依存していると考えられます。

- 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

基本的には、ランタイム環境の状態での変更は副次作用と見なされます。 ポインターまたは参照によって渡された関数引数を変更することは、唯一許可される副次作用です。他の副次作用を持つ関数は **#pragma isolated_call** ディレクティブにリストされたときに誤った結果を与えることがあります。

関数を **isolated_call** 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。

指定された関数は、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。また、関数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) によって指示されるストレージを変更することもできます。 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関数は指定しないでください。このような関数を **#pragma isolated_call** ディレクティブにリストすると、予期しない結果になります。

-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。 この **-qignprag** コンパイラー・オプションを使用して、 **#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してください。

例

次の例は、**#pragma isolated_call** ディレクティブの使用法を示します。 `this_function` 関数には副次作用がないため、この関数を呼び出しても外部関数 `a` の値は変更されません。コンパイラーは、`other_function` への引数が値 `6` を持ち、メモリーから変数が再ロードされないことを想定することができます。

```
int a;

// Assumed to have no side effects
int this_function(int);

#pragma isolated_call(this_function)
that_function()
{
   a = 6;
   // Call does not change the value of "a"
   this_function(7);

   // Argument "a" has the value 6
   other_function(a);
}
```

関連情報

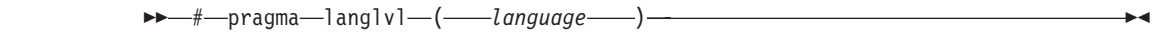
- 113 ページの『-qignprag』
- 133 ページの『-qisolated_call』

#pragma langlvl

説明

#pragma langlvl ディレクティブはコンパイルのために C 言語レベルを選択します。

構文



language の値は、以下の通りです。

C C プログラムの場合、以下のいずれかの値を *language* に指定できます。

classic	stdc89 以外の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。
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 標準に準拠するコンパイル。
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	コンパイルは **strict98** に基づいていますが、拡張言語フィーチャーを適合させるにはいくつかの相違があります。
strict98	C++ プログラムの ISO C++ 標準に準拠するコンパイル。

デフォルト

デフォルトの言語レベルは、コンパイラー呼び出しに使用するコマンドによって異なります。

呼び出し	デフォルト言語レベル
xlc	extc89
cc	extended
c89	stdc89
c99	stdc99

注

このプラグマは 1 つのソース・ファイルの中で 1 回しか指定できず、またソース・ファイルの中のコメント以外のステートメントの前に指定する必要があります。

コンパイラーは、ヘッダー・ファイル内の事前定義マクロを使用して、指定された言語レベルを定義するための宣言と定義を使用できるようにします。

この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することができます。その結果、**-E** コンパイラー・オプションを指定してコンパイルすると、**-E** オプションを指定してコンパイルしなかったときに作成されるものと異なる結果が作成される可能性があります。

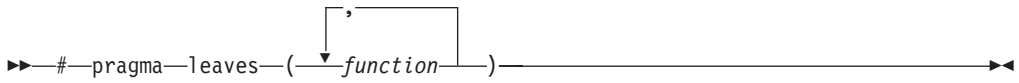
関連情報
- 137 ページの『-qlanglvl』
- 「*XL C/C++ 言語解説書*」の『IBM XL C 言語拡張』および『IBM XL C++ 言語拡張』

#pragma leaves

説明

#pragma leaves は関数名を取り、呼び出し後に関数がディレクティブに戻らないことを指定します。

構文



注

このプラグマは、*function* が呼び出し元に戻らないようにコンパイラーに命令します。

このプラグマの利点は、これを使用するとコンパイラーは *function* の後にあるコードをすべて無視することができるので、最適化プログラムはより効率的なコードを生成することができるという点です。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。 これと同様に振る舞う関数としては、**exit**、**longjmp**、および **terminate** があります。

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
 }
}
```

#pragma loop_id

説明

スコープ固有の ID を使用してブロックにマークを付けます。

構文

ここで、*name* はスコープ単位内で固有の ID です。

注

#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。

関連情報

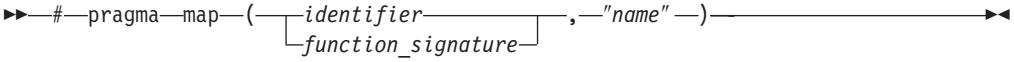
- 225 ページの『-qunroll』
- 248 ページの『#pragma block_loop』
- 293 ページの『#pragma unroll』
- 294 ページの『#pragma unrollandfuse』

#pragma map

説明

#pragma map ディレクティブは、ID に対するすべての参照を「*name*」に変換するようにコンパイラーに命令します。そして、*"name"* がオブジェクト・ファイルとアセンブリー・コードで使用されます。

構文



ここで、

identifier 外部結合を持つデータ・オブジェクトまたは非多重定義関数の名前。
C++ ID が多重定義の関数またはメンバー関数の名前である場合は、そのプラグマがコンパイラー生成の名前をオーバーライドする危険性があります。これにより、リンク時に問題が生じます。

function_signature 内部リンクを持つ関数または演算子の名前。 この名前は修飾できます。

name 指定したオブジェクト、関数、または演算子にバインドされる外部名。
C++ C++ 名 (C++ リンケージ・シグニチャーを持つ名前。これが C++ でのデフォルト・シグニチャーです) にリンクする場合はマングル名を指定してください。以下の『**例**』のセクションの『例 4』を参照。

注

ラベル名が以下と同じ場合、コンパイラーは重大エラー・メッセージを出します。

- 異なる変数または関数に指定されている既存のアセンブリー・ラベル名。
- **#pragma map** により異なる変数または関数に指定されている既存のマップ名。

#pragma map を使用して、以下をマップしないでください。

- C++ メンバー関数
- 多重定義関数
- テンプレートから生成されたオブジェクト
- 組み込みリンケージを持つ関数

このディレクティブは、プログラムのどこに指定してもかまいません。 ディレクティブ内に現れる ID は、プロトタイプ引数リストで使用されるすべての型名を含めて、実際の発生箇所とは無関係に、ディレクティブがファイル・スコープで現れた場合と同様に解決されます。

プラグマ・マップで指定された名前が 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。

例

例 1 C

```
int funcname1()
{
    return 1;
}

#pragma map(func , "funcname1") //maps func to funcname1

int main()
{
   return func(); // no function prototype needed in C
}
```

例 2 C++

```
extern "C" int funcname1()
{
  return 0;
}

extern "C" int func(); //function prototypes needed in C++

#pragma map(func , "funcname1") // maps ::func to funcname1

int main()
{
  return func();
}
```

例 3 C++

```
#pragma map(foo, "bar")

int foo();  //function prototypes needed in C++

int main()
{
   return foo();
}

extern "C" int bar() {return 7;}
```

以下の例は、**#pragma map** とアセンブリー・ラベルの間の対話がエラー・メッセージを生成する可能性のある幾つかのケースを示しています。

例 5

```
 #pragma map(a, "abc")

// error, since the label name is the same as a map name to a
// different identifier
 int cba asm("abc");
```

例 6

```
 int abc asm("myID");

//error, since the same label is used on two different variables
 int cba asm("myID");
```

プラグマ・マップで以前に指定されているラベル名と異なるラベル名を持つ宣言に asm ラベル指定が適用されると、コンパイラーがエラー・メッセージを生成します。

例 7

```
 #pragma map(a, "aaa")

// severe error, since "a" is already mapped by pragma map to a
// different name
 void a() asm("bbb");
```

例 8

```
 #pragma map(a, "aaa")

// Valid declaration, Since "a" is mapped to the same name
 int a asm("aaa");
```

プラグマ・マップが ID のマップ名を指定したとき、その名前が、異なる宣言にある以前のアセンブリー・ラベルのマップ名と矛盾している場合、**#pragma map** は無視され、警告メッセージが表示されます。

例 9

```
 int a asm("abc");

// Warning message,  since 'abc' is already used as a label name
 #pragma map(b, "abc")
```

#pragma map がすでにアセンブリー・ラベルを持っている ID をマップしようとすると、プラグマ・マップは無視され、警告メッセージが表示されます。

例 10

```
int a asm("abc");

 //Warning, since 'a' already has a label, pragma map is ignored
 #pragma map(a, "aaa")
```

例 11

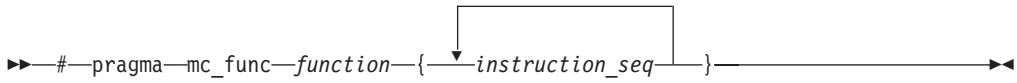
```
int a asm("abc");

// Valid declaration, Since "a" is mapped to the same name
 #pragma map(a, "abc")
```

#pragma mc_func

説明

#pragma mc_func ディレクティブを使用すると、短いシーケンスのマシン・インストラクションを含む関数を定義することができます。

構文



ここで、

function C または C++ プログラムで前に定義されている関数を指定します。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_seq ゼロ以上の 16 進数字のシーケンスを含むストリングです。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。

注

mc_func プラグマを使用すると、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ができます。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検出される `asm` キーワードに似ています。

mc_func 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だけに現れます。 **#pragma mc_func** によって定義される関数名は、事前に宣言されているかプロトタイプ化されている必要があります。

コンパイラーは他の関数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。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、 287 ページの『#pragma reg_killed_by』を参照してください。

instruction_seq 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。

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。

65535 バイトを超えるストリング・リテラルがプラグマ・マップに指定されると、通知メッセージが出され、プラグマは無視されます。

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 int 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。これはチェックサムの計算で頻繁に使用されます。

またこの例は、**#pragma reg_killed_by** を使用して、**#pragma mc_func** によって定義された関数によって変更できる特定の揮発性レジスターのセットをリストする方法も示しています。

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4           */
                /*   addze      r3 <- r3, carry bit    */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf("¥n¥nresult = %d¥n¥n",j);
}
```

関連情報

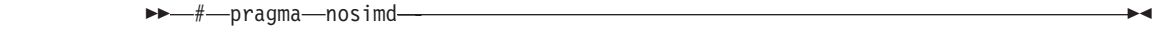
- 268 ページの『#pragma isolated_call』
- 287 ページの『#pragma reg_killed_by』
- 61 ページの『-qasm』

#pragma nosimd

説明

#pragma nosimd ディレクティブはコンパイラーに、このディレクティブのすぐ後に続くループで VMX (Vector Multimedia Extension) 命令を生成しない よう命令します。

構文



注

このディレクティブは VMX をサポートするアーキテクチャーに対してのみ、**-qhot=simd** オプションと共に使用された場合に効果を持ちます。これらのコンパイラー・オプションが有効な場合、コンパイラーは 1 つの配列の連続するエレメントのループで実行される特定の操作を、VMX (Vector Multimedia Extension) 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。

#pragma nosimd ディレクティブは、`while`、`do while`、および `for` ループにのみ適用されます。

#pragma nosimd ディレクティブはそのすぐ後に続くループにのみ適用されます。このディレクティブは、指定されたループ内にネストされている可能性のある他のループには何の効果も持ちません。

#pragma nosimd ディレクティブは、特定の最適化レベルを必要とせずに、ループ最適化および OpenMP ディレクティブと混用することができます。

関連情報

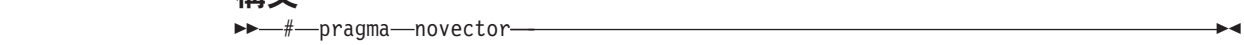
- 58 ページの『-qarch』
- 89 ページの『-qenablevmx』
- 109 ページの『-qhot』

#pragma novector

説明

#pragma novector ディレクティブは、このディレクティブのすぐ後に続くループを自動ベクトル化しない ようコンパイラーに命令します。

構文



注

このディレクティブはベクトル化をサポートするアーキテクチャー上で、**-qhot=vector** オプションと共に使用された場合にのみ効果を持ちます。**-qhot=vector** が有効な場合、コンパイラーはループ内で連続する配列のエレメント (平方根、逆平方根) に対して実行される特定の演算をベクトル・ライブラリー・ルーチンへ (MASS ライブラリー) の呼び出しに変換します。この呼び出しは同時に幾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高速です。

#pragma novector ディレクティブは、`while`、`do while`、および `for` ループにのみ適用されます。

#pragma novector ディレクティブはそのすぐ後に続くループにのみ適用されます。このディレクティブは、指定されたループ内にネストされている可能性のある他のループには何の効果も持ちません。

#pragma novector ディレクティブは、特定の最適化レベルを必要とせずに、ループ最適化および OpenMP ディレクティブと混用することができます。

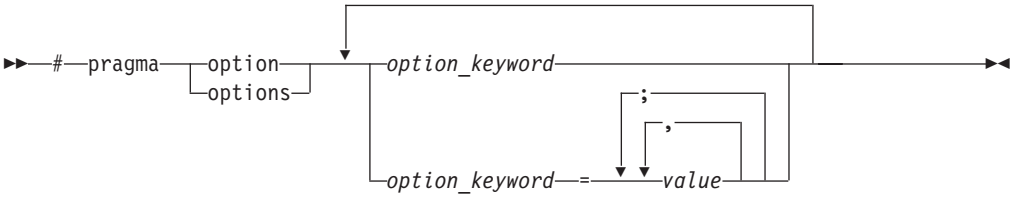
関連情報
- 109 ページの『-qhot』

#pragma options

説明

#pragma options ディレクティブは、ソース・プログラムのコンパイラー・オプションを指定します。

構文



注

一般にプラグマ・オプションはデフォルトでコンパイル単位全体に適用されます。

#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・スペースを使っ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
```

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、および他のプラグマの指定に限り、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 ... */
```

ソース・プログラム内のどのコードより前に指定されたオプションは、コンパイル単位全体に適用されます。プログラム全体で他のプラグマ・ディレクティブを使用すると、ソース・コードの選択済みブロックに対してオプションをオンにすることができます。例えば、以下のようにソース・コードの一部をコンパイラー・リストに組み込むように要求することができます。

```
#pragma options source
/*  Source code between the source and nosource #pragma
    options is included in the compiler listing               */
#pragma options nosource
```

下記のテーブルの設定は、**#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**	**説明**
align=*option*	-qalign	コンパイラーがファイルのコンパイルに使用する集合体の位置合わせ規則を指定する。
[no]attr attr=full	-qattr	すべての名前を含む属性のリストを作成する。
chars=*option*	-qchars #pragma chars も参照	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ようコンパイラーに命令する。
[no]check	-qcheck	特定タイプのランタイム検査を行うコードを生成する。
[no]compact	-qcompact	最適化と共に使用すると、可能な場合に、実行速度を犠牲にしてコード・サイズを削減する。
[no]dbcs	-qmbcs、-qdbcs	ストリング・リテラルとコメントには、DBCS 文字を含むことができる。
C [no]dbxextra	-qdbxextra	参照されない変数のためのシンボル・テーブル情報を生成する。
[no]digraph	-qdigraph	特定の連字とキーワード演算子を使用することができる。
[no]dollar	-qdollar	ID の名前に $ シンボルを使用できるようにする。
enum=*option*	-qenum #pragma enum も参照	列挙の占めるストレージの量を指定する。
flag=*option*	-qflag	報告する診断メッセージの最小の重大度レベルを指定する。 重大度レベルは、以下のように指定することもできます。 #pragma options flag=i => #pragma report (level,I) #pragma options flag=w => #pragma report (level,W) #pragma options flag=e,s,u => #pragma report (level,E)
float=[no]*option*	-qfloat	浮動小数点演算の速度や精度を上げるためにさまざまな浮動小数点オプションを指定する。
[no]flttrap=*option*	-qflttrap	追加の命令を生成し、浮動小数点例外を検出してトラップする。

#pragma options *option_keyword* に有効な設定	**同等のコンパイラー・オプション**	**説明**
[no]fullpath	-qfullpath	ファイルに保管されているパス情報を dbx スタブ・ストリングに指定する。
[no]funcsect	-qfuncsect	各関数ごとの命令を別個の csect に入れる。
halt	-qhalt	指定された重大度のエラーが検出されると、コンパイラーを停止する。
[no]idirfirst	-qidirfirst	ユーザー・組み込みファイルの検索順序を指定する。
[no]ignerrno	-qignerrno	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。
ignprag=*option*	-qignprag	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。
[no]info=*option*	-qinfo #pragma info も参照	通知メッセージを作成する。
initauto=*value*	-qinitauto	指定された 16 進バイト値に自動ストレージを初期化する。
[no]inlglue	-qinlglue	外部関数の呼び出しまたは関数ポインターを介した呼び出しに必要なポインター・グルー・コードをインライン化することにより、高速な外部結合を生成する。
isolated_call=*names*	-qisolated_call #pragma isolated_call も参照	ソース・ファイル内の副次作用がない関数を指定する。
C langlvl	-qlanglvl	異なる言語レベルを指定する。 この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することができます。その結果、**-E** コンパイラー・オプションを指定してコンパイルすると、**-E** オプションを指定してコンパイルしなかったときに作成されるものと異なる結果が作成される可能性があります。
[no]libansi	-qlibansi	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。

#pragma options *option_keyword* に有効な設定	**同等のコンパイラー・オプション**	**説明**
[no]list	-qlist	オブジェクト・リストを含むコンパイラー・リストを生成する。
[no]longlong	-qlonglong	プログラムで `long long` 型を許可する。
[no]maxmem=*number*	-qmaxmem	指定した重大度以上のエラーの件数が指定数に達したらコンパイルを停止するようコンパイラーに命令する。
[no]mbcs	-qmbcs、-qdbcs	ストリング・リテラルとコメントには、DBCS 文字を含むことができる。
[no]optimize optimize=*number*	-O、-qoptimize	プログラム・コードのセクションに適用される最適化レベルを指定する。 コンパイラーは *number* の値として以下を受け入れます。 • **0** - レベル 0 の最適化を設定します • **2** - レベル 2 の最適化を設定します • **3** - レベル 3 の最適化を設定します *number* に値が指定されなかった場合、コンパイラーはレベル 2 の最適化を想定します。
C++ priority=*number*	-qpriority 286 ページの『#pragma priority』 も参照	静的コンストラクターを初期化する場合の優先順位を指定する。
[no]proclocal、 [no]procimported、 [no]procunknown	-qproclocal、-qprocimported、-qprocunknown	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。
C [no]proto	-qproto	このオプションが設定されると、コンパイラーは、すべての関数がプロトタイプ化されているものと想定する。
[no]ro	-qro	ストリング・リテラルのストレージ・タイプを指定する。
[no]roconst	-qroconst	定数値のストレージ・ロケーションを指定する。
[no]showinc	-qshowinc	**-q-qsource** で使用された場合は、すべての組み込みファイルをソース・リストに組み込む。
[no]source	-qsource	ソース・リストを作成する。

#pragma options *option_keyword* に有効な設定	同等のコンパイラー・オプション	説明
spill=*number*	-qspill	レジスター割り振り予備域のサイズを指定する。
[no]stdinc	-qstdinc	`#include <file_name>` および `#include " file_name"` ディレクティブで組み込むファイルを指定する。
[no]strict	-qstrict	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** コンパイラー・オプションの積極的な最適化をオフにする。
tbtable=*option*	-qtbtable	コンパイラーによって認識されるタブの長さを変更する。
tune=*option*	-qtune	実行可能プログラムの最適化対象とするアーキテクチャーを指定する。
[no]unroll unroll=*number*	-qunroll	指定した係数によってプログラムの内部ループをアンロールする。
C [no]upconv	-qupconv	整数拡張を行うときに符号なしの指定を保持する。
C++ [no]vftable	-qvftable	仮想関数テーブルの生成を制御する。
[no]xref	-qxref	すべての ID の相互参照リストを含むコンパイラー・リストを生成する。

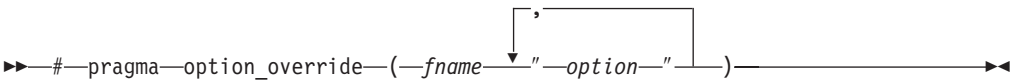
関連情報
- 86 ページの『-E』

#pragma option_override

説明

#pragma option_override ディレクティブを使用して、特定の関数に適用する代替の最適化オプションを指定することができます。

構文



option の有効な設定と構文、および対応するコマンド行オプションは以下に示す通りです。

#pragma option_override *option* **の設定と構文**	**コマンド行オプション**	**例**
opt(level,*number*)	-O, -O2, -O3, -O4, -O5	#pragma option_override (*fname*, ″opt(level, 3)″)
opt(registerSpillSize,*num*)	-qspill=*num*	#pragma option_override (*fname*, ″opt(registerSpillSize,512)″)
opt(size[,yes])	-qcompact	#pragma option_override (*fname*, ″opt(size)″) #pragma option_override (*fname*, ″opt(size,yes)″)
opt(size,no)	-qnocompact	#pragma option_override (*fname*, ″opt(size,no)″)
opt(strict)	-qstrict	#pragma option_override (*fname*, ″opt(strict)″)
opt(strict,no)	-qnostrict	#pragma option_override (*fname*, ″opt(strict,no)″)

注

デフォルトでは、コマンド行に指定した最適化オプションは、ソース・プログラム全体に適用されます。ただし、最適化がオンになっている場合だけ、特定タイプのランタイム・エラーが発生することがあります。このプラグマを使用してプログラム内の特定の関数 (*fname*) のコマンド行の最適化設定をオーバーライドすることができます。このことは、これらの関数のプログラミング・エラーを識別して訂正するのに役立つ場合があります。

関数単位の最適化は、コンパイル・オプションによって最適化がすでに使用可能になっている場合にのみ有効です。コンパイル中のプログラムの残りの部分に適用されるよりも低いレベルで関数ごとの最適化を要求することができます。このプラグマを介してオプションを選択すると、選択された特定の最適化オプションにのみ有効になり、関連オプションの暗黙設定には影響ありません。

オプションを指定するときは、二重引用符で囲むため、マクロ展開には影響されません。引用符で囲んで指定したオプションは、ビルド・オプションの構文に従わなければなりません。

このプラグマは多重定義のメンバー関数には使用できません。

このプラグマは、コンパイル単位に定義された関数にのみ影響し、例えば以下のように、コンパイル単位のどこにでも指定することができます。

- コンパイル単位の前または後
- 関数定義の前または後
- 関数宣言の前または後
- 参照された関数の前または後
- 関数定義の内側または外側

関連情報

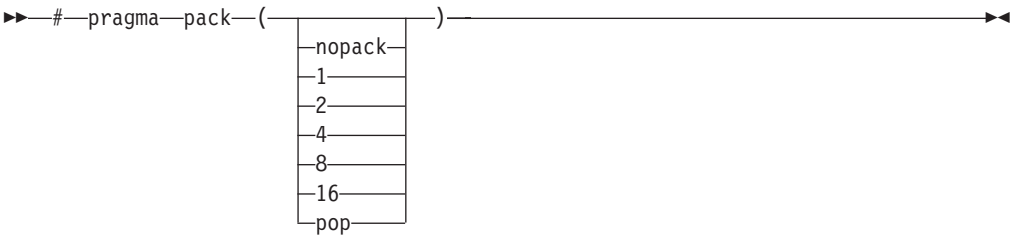
- 166 ページの『-O、-qoptimize』
- 73 ページの『-qcompact』
- 203 ページの『-qspill』
- 207 ページの『-qstrict』

#pragma pack

説明

#pragma pack ディレクティブは、そのディレクティブの後に続く構造体のメンバーの位置合わせ規則を変更します。

構文



ここで、

1 | 2 | 4 | 8 | 16	構造体のメンバーは指定されたバイト位置合わせか自然な位置合わせの境界のうち、いずれか少ない方に位置合わせされ、指定された値はスタックにプッシュされます。
nopack	パッキングは適用されず、パック・スタックに **nopack** がプッシュされます。
pop	プラグマ・パック・スタックの一番上のエレメントがポップされます。
(*引数が指定されていません*)	**#pragma pack()** を指定した場合、**#pragma pack(pop)** を指定したのと同じ効果があります。

注

#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、位置合わせ規則に従って、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、整数データ型 (int) のメンバーの場合、**#pragma pack(2)** ディレクティブでは、当該メンバーは構造体内で 2 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、スタックをベースにしています。すべてのパック値は、ソース・コードの構文解析時にスタックにプッシュされます。現行のプラグマ・パック・スタックの一番上にある値が、現行の位置合わせ規則のスコープ内の後続のすべての構造体のメンバーをパックするために使用されます。

#pragma pack スタックは、位置合わせ規則スタック内の現行エレメントに関連付けられます。位置合わせ規則は、**-qalign** コンパイラー・オプションまたは **#pragma options align** ディレクティブで指定します。新しい位置合わせ規則が指定されると、新規の **#pragma pack** スタックが作成されます。現行の位置合わせ規則

がポップ位置合わせ規則スタックからポップすると、現行の **#pragma pack** スタックが空になり、前の **#pragma pack** スタックが復元されます。スタック操作 (パック設定のプッシュおよびポップ) は、現行の **#pragma pack** スタックにのみ影響します。

#pragma pack ディレクティブ ビット・フィールドにビット・フィールド・コンテナーの境界をクロスさせます。

例

1. 以下に示すコードでは、構造体 s_t2 ではメンバーが 1 バイトにパックされますが、構造体 s_t1 は影響を受けません。この理由は、s_t1 の宣言がプラグマ・ディレクティブよりも前に開始されているためです。しかし、s_t2 はその宣言がプラグマ・ディレクティブよりも後に開始されているため影響を受けます。

```
struct s_t1 {
    char a;
    int b;
    #pragma pack(1)
    struct s_t2 {
            char x;
            int y;
    } S2;
    char c;
    int d;
} S1;
```

2. 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):**
sizeof s_t = 16	sizeof s_t = 11
offsetof a = 0	offsetof a = 0
offsetof b = 4	offsetof b = 1
offsetof c = 8	offsetof c = 5
offsetof d = 12	offsetof d = 7
align of a = 1	align of a = 1
align of b = 4	align of b = 1
align of c = 2	align of c = 1
align of d = 4	align of d = 1

関連情報

- 55 ページの『-qalign』
- 278 ページの『#pragma options』
- 「*XL C/C++ プログラミング・ガイド*」の『位置合わせ修飾子の使用法』

#pragma priority

C++

説明

#pragma priority ディレクティブは、静的オブジェクトが初期化される順序を指定します。

構文

注

n の値は、101 から 65535 の範囲の整数リテラルでなければなりません。デフォルト値は 65535 です。A の低い値は高い優先順位を示しています。高い値は低い優先順位を示しています。

優先順位の値は、明示的値が変数属性 init_priority によって提供されている場合や、別の **#pragma priority** ディレクティブを検出した場合を除き、**#pragma priority** ディレクティブに続くすべてのグローバルおよび静的オブジェクトに適用されます。

同じ優先順位の値を持つオブジェクトは、宣言順に構成されます。複数のファイル間のオブジェクトの構成順序を指定するには、**#pragma priority** を使用します。ただし、ソース・ファイルから実行可能ファイルまたは共用ライブラリー・ターゲットを作成している場合、コンパイラーは依存関係の順序付けをチェックし、それにより **#pragma priority** が上書きされる可能性があります。

例えば、オブジェクト A のコピーがパラメーターとしてオブジェクト B のコンストラクターに渡されると、コンパイラーは上から下の順序、または **#pragma priority** の順序付けに違反することになっても、A が最初に構成されるように調整します。このことは、コンパイラーが許可するオーダーレス・プログラミングに不可欠です。ターゲットが `.obj/.lib` の場合、依存性を検出するための情報が十分ではない可能性があるため、この処理は行われません。

注: C++ の変数属性 `init_priority` は、クラスの型の共用変数に優先順位を割り当てるためにも使用することができます。詳しくは、「*XL C/C++ 言語解説書*」の 『init_priority 変数属性』を参照してください。

例

```
#pragma priority(1001)
```

関連情報

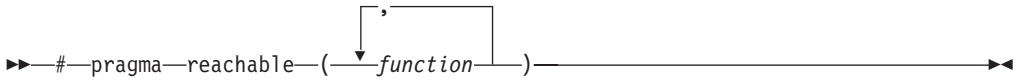
- 114 ページの『-qinfo』
- 「*XL C/C++ プログラミング・ガイド*」の『ライブラリー内の静的オブジェクトの初期化』

#pragma reachable

説明

#pragma reachable ディレクティブはルーチン、*function* を呼び出した後のポイントが、いくつかの不明のロケーションからの分岐の対象となる可能性があることを宣言します。このプラグマは `setjmp` マクロと共に使用します。

構文

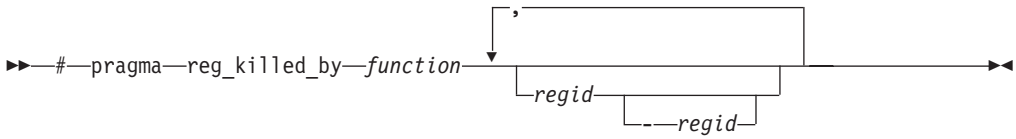


#pragma reg_killed_by

説明

#pragma reg_killed_by ディレクティブは指定された関数によって変更 (kill) される可能性のある揮発性レジスターのセットを指定します。このプラグマは、**#pragma mc_func** を使用して定義される関数でしか使用できません。

構文



ここで、

function 以前に **#pragma mc_func** を使って定義した関数。

regid 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。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されます。レジスターが指定されていない場合は、指定された *function* でどのレジスターも変更されません。

シンボル名は 2 つの部分からなります。最初の部分は、「a」から「z」または「A」から「Z」、あるいはその両方の範囲の 1 つ以上の文字のシーケンスを使用して指定されたレジスターのクラス名です。

2 番目の部分は `unsigned int` の範囲の整数価です。この数字はレジスター・クラスの中の特定のレジスター番号を識別します。一部のレジスター・クラスではレジスター番号の指定が必要なく、指定しようとした場合、エラーが発生します。

regid 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。

レジスター	
クラスと [レジスター番号]	**記述と使用法**
ctr	カウント・レジスター (CTR)

cr[0-7]	条件レジスター (CR) • このクラスのそれぞれのレジスターは、条件レジスターの中の 4 ビット・フィールドの 1 つです。 • 8 つの CR フィールドのうち、**cr0**、**cr1**、および **cr5**-**cr7** だけが **#pragma reg_killed_by** によって指定できます。
fp[0-31]	浮動小数点レジスター (FPR) • 32 のマシン・レジスターのうち、**fp0**-**fp13** だけが **#pragma reg_killed_by** によって指定できます。
fs	浮動小数点状況および制御レジスター (FPSCR)
lr	リンク・レジスター (LR)
gr[0-31]	汎用レジスター (GPR) • 32 のマシン・レジスターのうち、**gr0** と **gr3**-**gr12** だけが **#pragma reg_killed_by** によって指定できます。
vr[0-31]	ベクトル・レジスター (VMX プロセッサー専用)
xer	固定小数点例外 (XER)

注

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。

regid によって指定されるレジスターは、以下の要件を満たしている必要があります。

- レジスター名のクラス名の部分が有効でなければならない
- レジスター番号は必須または禁止である
- レジスター番号が必須の場合、その番号は有効範囲内になければならない

これらの要件のいずれかが満たされていない場合は、エラーが発行され、プラグマは無視されます。

例

以下の例は、**#pragma reg_killed_by** を使用して、**#pragma mc_func** で定義された関数によって使用される揮発性レジスターの特定セットをリストする方法を示しています。

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4           */
                /*   addze      r3 <- r3, carry bit    */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

      int i,j,k;

      i = 4;
```

```
    k = -4;
    j = add_logical(i,k);
    printf("¥n¥nresult = %d¥n¥n",j);
}
```

関連情報

- 275 ページの『#pragma mc_func』

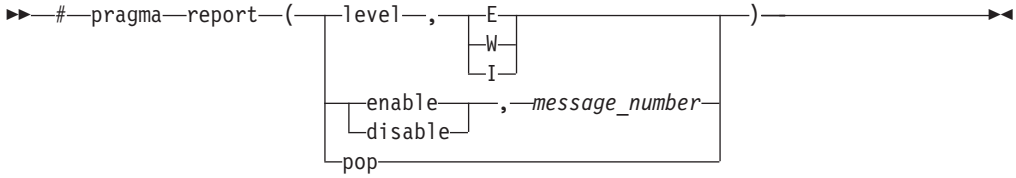
#pragma report

C++

説明

#pragma report ディレクティブは、特定のメッセージの生成を制御します。このプラグマは **#pragma info** に優先します。 **#pragma report(pop)** を指定すると、レポート・レベルは前のレベルに戻されます。前のレポート・レベルが指定されていない場合は、警告が表示され、レポート・レベルは変更されません。

構文



ここで、

レベル
E | W | I
: 表示する診断メッセージの最小の重大度レベルを示します。
表示する診断メッセージのタイプを決定するために **level** と共に使用されます。

E
: 「エラー」の最小のメッセージ重大度を示します。これは診断メッセージの最も重大なタイプと考えられます。「E」のレポート・レベルは、「エラー」メッセージだけ表示します。 レポート・レベルを「E」に設定する代わりの方法は、**-qflag=e:e** コンパイラー・オプションを指定することです。

W
: 「警告」の最小のメッセージ重大度を示します。「W」のレポート・レベルはフィルタリングして通知メッセージをすべて表示しないようにし、警告メッセージまたはエラー・メッセージに分類されたメッセージだけを表示します。 レポート・レベルを「W」に設定する代わりの方法は、**-qflag=w:w** コンパイラー・オプションを指定することです。

I
: 「通知」の最小のメッセージ重大度を示します。通知メッセージは、最も重大度の低い診断メッセージであると考えられます。レベル「I」はすべてのタイプのメッセージを表示します。コンパイラーはこれをデフォルト・オプションとして設定します。レポート・レベルを「I」に設定する代わりの方法は、**-qflag=i:i** コンパイラー・オプションを指定することです。

enable | disable
: 指定されたメッセージ番号を使用可能または使用不可にします。

message_number
: メッセージ番号のプレフィックスが含まれた ID で、この後にメッセージ番号が続きます。メッセージ番号の例: CPPC1004

pop　　レポート・レベルを前のレポート・レベルにリセットします。 ポップ操作が空のスタックに対して実行されると、レポート・レベルは未変更のままとなり、メッセージは生成されません。

例

1. **#pragma info** の指定は、コンパイラーにすべての通知診断をプリントするよう命令します。プラグマ・レポートはコンパイラーに、重大度「W」または警告メッセージのメッセージだけを表示するよう命令します。この場合、通知診断はまったく表示されません。

   ```
   1  #pragma info(all)
   2  #pragma report(level, W)
   ```

2. CPPC1000 がエラー・メッセージであった場合、表示されます。別のタイプの診断メッセージの場合は表示されません。

   ```
   1  #pragma report(enable, CPPC1000)  // enables message number CPPC1000
   2  #pragma report(level, E)   // display only error messages.
   ```

   次のようにコードの順序を変更すると、

   ```
   1  #pragma report(level, E)
   2  #pragma report(enable, CPPC1000)
   ```

   同じ結果になります。コードの 2 つの行が表示される順序は、結果に影響しません。ただし、メッセージが「使用不可」であった場合は、設定されているレポート・レベルやコードの行の表示順序に関係なく、診断メッセージは表示されません。

3. 下記の例の 1 行目で、初期のレポート・レベルは「I」に設定されており、分類されている診断メッセージのタイプに関係なく、メッセージ CPPC1000 が表示されています。 3 行目には、新しいレポート・レベル「E」が設定され、重大度レベル「E」のメッセージだけが表示されることを示しています。3 行目のすぐ後に、現行レベル「E」が「ポップ」して、「I」にリセットされています。

   ```
   1  #pragma report(level, I)
   2  #pragma report(enable, CPPC1000)
   3  #pragma report(level, E)
   4  #pragma report(pop)
   ```

関連情報

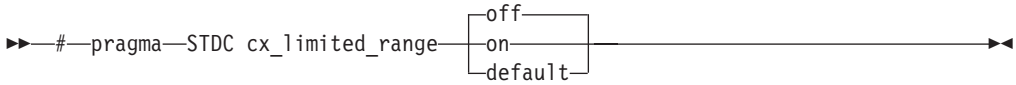
- 95 ページの『-qflag』

#pragma STDC cx_limited_range

説明

STDC cx_limited_range プラグマは、そのプラグマが制御するスコープ内では、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します。このプラグマのデフォルト設定は **off** です。

構文



注

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。

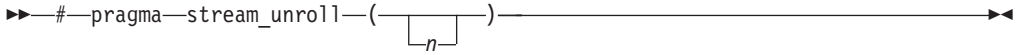
プラグマは最初に現れた位置から、次の **cx_limited_range** プラグマを検出するか、変換単位の終わりまで有効となります。プラグマが複合ステートメント内に現れると (ネストされた複合ステートメントを含む)、最初に現れた位置から、別の **cx_limited_range** プラグマを検出するか、その複合ステートメントの終わりまで有効となります。

#pragma stream_unroll

説明

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します。

構文



ここで、*n* はループのアンロール係数です。C プログラムでは、*n* の値は正の整数定数式です。C++ プログラムでは、*n* の値は正のスカラー整数またはコンパイル時の定数初期化式です。 1 のアンロール係数はアンロールを使用不可にします。*n* 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、最適化プログラムが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。

注

-O3 または **-qipa=level=2** のいずれもストリームのアンロールを使用可能にするのに十分ではありません。追加で **-qhot** 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。

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 **#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 (**unroll**、**nounroll**、**unrollandfuse**、**nounrollandfuse**) と結合した場合も、XL C から警告が発行されます。XL C++ は同じ **for** ループに指定された複数のループのアンロール・プラグマのうち最後のプラグマを除き、黙ってすべて無視します。

ストリームのアンロールは、特定の最適化オプションが指定されたコンパイルでも抑制されます。オプション **-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。したがって、**-qhot** オプションのみでストリームのアンロールを使用可能にしたい場合は、**-qnostrict** も指定する必要があります。

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 は反復の数を 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];
}
```

より多くの読み取り操作と保管操作がコンパイラーによって決定された多くのストリームに分散されたことにより、計算時間が削減され、パフォーマンスが改善されます。

関連情報

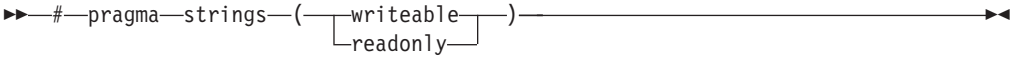
- 225 ページの『-qunroll』
- 293 ページの『#pragma unroll』
- 294 ページの『#pragma unrollandfuse』

#pragma strings

説明

#pragma strings ディレクティブはストリング・リテラルのストレージ・タイプを設定し、それらを読み取り専用メモリーに入れるか読み取り/書き込みメモリーに入れるかを指定します。

構文



注

C 任意の形式のコンパイラー呼び出し **xlc** が使用された場合、ストリングはデフォルトで読み取り専用です。

C++ 任意の形式のコンパイラー呼び出し **xlC** または **xlc++** が使用された場合、ストリングはデフォルトで読み取り専用です。

このプラグマは、すべてのソース・ステートメントの前になければ、有効にはなりません。

例

```
#pragma strings(writeable)
```

関連情報

- 191 ページの『-qroconst』

#pragma unroll

説明

#pragma unroll ディレクティブは、プログラム内の最内部または最外部の `for` ループをアンロールするために使用され、プログラムのパフォーマンス改善を支援します。

構文

ここで、*n* は、ループのアンロール係数です。 C プログラムでは、*n* の値は正の整数定数式です。C++ プログラムでは、*n* の値は正のスカラー整数またはコンパイル時の定数初期化式です。 アンロール係数 1 はアンロールを使用不可にします。*n* 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、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。

注

#pragma unroll および **#pragma nounroll** ディレクティブは、`for` ループまたは block_loop ディレクティブでのみ使用できます。これは `do while` および `while` ループには適用できません。

#pragma unroll および **#pragma nounroll** ディレクティブは、ループまたは影響を受ける block_loop ディレクティブの直前に現れなければなりません。

1 つの特定のループには、これらのディレクティブの 1 つだけを指定することができます。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。

ループに **#pragma nounroll** を指定すると、コンパイラーにそのループをアンロールしないよう命令します。 **#pragma unroll(1)** を指定しても同じ結果になります。

unroll オプションによって、特定のアプリケーションのパフォーマンスが改善されるかどうかを確認するには、まず、通常オプションでプログラムをコンパイルしてから、それを代表的なワークロードで実行してください。次に、コマンド行 **-qunroll** オプションを指定するか、**unroll** プラグマを使用可能にして (あるいはその両方を行って)、プログラムを再コンパイルしてから、同じ条件下で再実行して、パフォーマンスが改善されたかどうか確認してください。

例

- この例では、ループ制御が変更されます。

```
#pragma unroll(3)
for (i=0; i<n; i++) {
  a[i]=b[i] * c[i];
}
```

3 でアンロールすると、以下が生成されます。

```
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

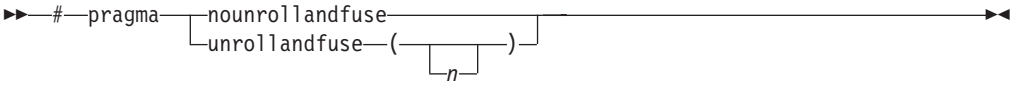
- 225 ページの『-qunroll』
- 『#pragma unrollandfuse』

#pragma unrollandfuse

説明

このプラグマはネストされた for ループに対してアンロールおよびフューズ操作を試行するようコンパイラーに命令します。

構文



ここで、*n* はループのアンロール係数です。C プログラムでは、*n* の値は正の整数定数式です。C++ プログラムでは、*n* の値は正のスカラー整数またはコンパイル時の定数初期化式です。 *n* 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。

注

#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 ループ構造体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 **nounrollandfuse**、**nounroll**、**unroll**、または **stream_unroll** ディレクティブと結合しないでください。

#pragma nounrollandfuse を指定すると、そのループをアンロールしないようコンパイラーに命令します。

例

1. 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、フューズしています。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構造体に適用した場合の可能な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];
    }
}
```

2. 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

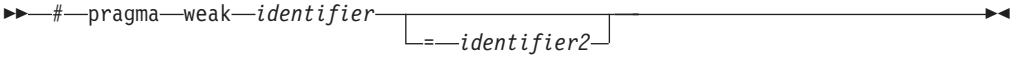
- 225 ページの『-qunroll』
- 293 ページの『#pragma unroll』

#pragma weak

説明

#pragma weak ディレクティブを実行すると、シンボルの定義が検出されないときやリンク中にシンボルが複数回定義されていることを検出したときに、リンケージ・エディターがエラー・メッセージを発行するのを阻止します。

構文



注

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。

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。

ダイナミック・リンカーはコマンド行に最初に表示されるオブジェクトに定義を使用します。 そのため、オブジェクト・ファイルがリンカーに示される順序は重要です。

プログラム・ソースには、2 つの形式の **#pragma weak** を指定できます。

#pragma weak *identifier*

この形式のプラグマは、*identifier* を弱いグローバル・シンボルとして定義します。*identifier* への参照は、定義されている場合は identifier 値を使用し、定義されていない場合は、*identifier* には 0 の値が割り当てられます。

Identifier が **#pragma weak** *identifier* と同じコンパイル単位に定義されている場合、*identifier* は弱い定義として扱われます。*identifier* を使用も宣言もしていないコンパイル単位に **#pragma weak** が存在する場合、プラグマは受け入れられますが無視されます。

identifier が C++ リンケージを持つ関数を指示するときは、*identifier* はその関数の C++ マングル名を使用して指定しなければなりません。また、C++ 関数がテンプレート関数の場合は、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。

#pragma weak *identifier*=*identifier2*

この形式のプラグマは、*identifier* を弱いグローバル・シンボルとして定義します。*identifier* を参照するときは、*identifier2* の値を使用します。

identifier2 はメンバー関数にできません。

identifier 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には定義しないでください。

identifier がこのコンパイル単位で宣言される場合、*identifier* の宣言は *identifier2* の宣言と互換性がなければなりません。例えば、*identifier2* が関数の場合、*identifier* は *identifier2* 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。

identifier2 は **#pragma weak** と同じコンパイル単位で宣言されていなければなりません。

identifier2 が C++ リンケージを持つ関数を示す場合は、その関数のマングル名を使用して *identifier* と *identifier2* の名前を指定する必要があります。C++ 関数がテンプレート関数の場合は、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。

以下の場合、コンパイラーは **#pragma weak** を無視して警告メッセージを出します。

- 指定された *identifier2* がコンパイル単位に定義されていない。
- 指定された *identifier2* がメンバー関数である。
- *identifier* は宣言されているが、その型が指定された *identifier2* の型と互換性がない。

弱い *identifier* が定義されているときには、コンパイラーは **#pragma weak** を無視し、重大エラー・メッセージを出します。

例

1. 以下はプラグマの **#pragma weak** *identifier* 形式の例です。

```
// Begin Compilation Unit 1
#include <stdio.h>
extern int foo;
#pragma weak foo

int main()
{
   int *ptr;
   ptr = &foo;
   if (ptr == 0)
      printf("foo has been assigned a value of 0¥n");
   else
      printf("foo was already defined¥n");
}
//End Compilation Unit 1
```

```
// Begin Compilation Unit 2
int foo = 1;
// End Compilation Unit 2
```

Compilation Unit 1 だけがコンパイルされて実行可能ファイルを作成する場合、identifier foo が定義され、値 0 を割り当てられます。実行からの出力は次のストリングになります: 「foo に値 0 が割り当てられました (foo has been assigned a value of 0)。」

2. 以下はプラグマの **#pragma weak** *identifier*=*identifier2* の形式の例です。

```
//Begin Compilation Unit
extern "C" void printf(char *,...);

void foo1(void)
{
   printf("Just in function foo1()¥n");
}


#pragma weak _Z3foov = _Z4foo1v

int main()
{
   foo();
}
//End Compilation Unit
```

並列処理のためのプラグマ・ディレクティブ

並列処理操作は、プログラム・ソースのプラグマ・ディレクティブによって制御されます。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。

#pragma omp atomic

説明

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。

statement	**条件**
x++	
++x	
x--	
--x	

注

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

説明

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

ここで、*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。

注

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。

ループの反復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。

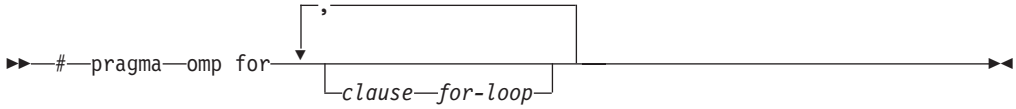
ネストされた並列領域は、常に直列化されています。

#pragma omp for

説明

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反復を配布するようコンパイラーに命令します。

構文



ここで、*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、最後の反復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 構成が **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 ループの反復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。

dynamic ループの反復はサイズ **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 ループの反復はサイズ **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* ループの反復がサイズ *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、ループの反復は 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反復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` ループ構成体です。

```
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
: 反復変数。反復変数は、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。反復変数は、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、反復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。

注

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反復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 ループの反復変数は、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、ループのすべての反復について同一でなければなりません。
- **omp for** ディレクティブが受け入れることができる **schedule** 節は、1 つのみです。
- *n* の値 (チャンク・サイズ) は、並列領域のすべてのスレッドについて同一でなければなりません。

#pragma omp ordered

説明

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

注

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。

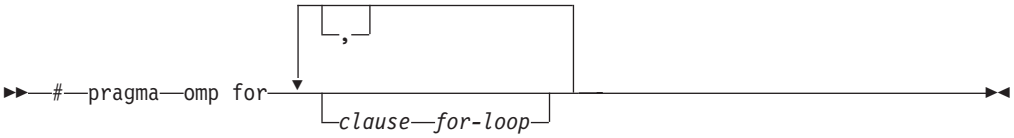
- **ordered** 節を含んでいる **omp for** または **omp parallel for** 構成の範囲内に表示されなければなりません。
- すぐ後に続くステートメント・ブロックに適用します。そのブロックのステートメントは、反復が順次ループ内で実行されるのと同じ順序で実行されます。
- ループの反復は、同一の **omp ordered** ディレクティブを 1 回以上実行してはなりません。
- ループの反復では、複数の特殊 **omp ordered** ディレクティブを実行してはなりません。

#pragma omp parallel for

説明

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



注

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

説明

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

ここで、*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。

注

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。

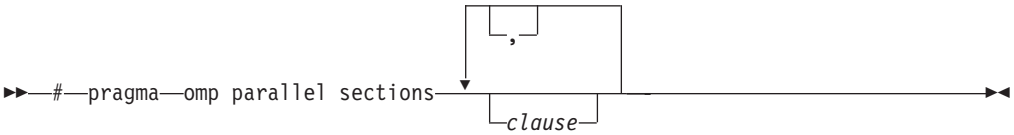
プログラム実行が **omp sections** ディレクティブに到達すると、後続の **omp section** ディレクティブによって定義されたプログラム・セグメントは、並列実行のために使用可能なスレッド間で配布されます。 **nowait** 節が指定されていない限り、バリアは **omp sections** ディレクティブに関連した、より広いプログラム領域の終了地点に暗黙的に定義されます。

#pragma omp parallel sections

説明

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



注

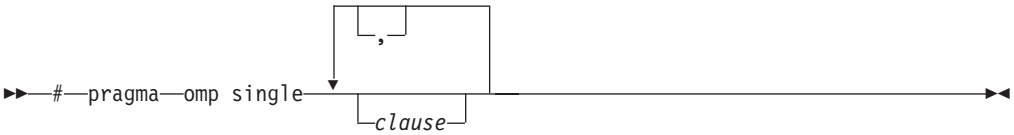
omp parallel および **omp sections** ディレクティブに記載されているすべての文節および制限は、 **omp parallel sections** ディレクティブに適用されます。

#pragma omp single

説明

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



ここで、*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成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。

注

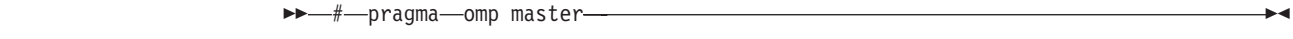
nowait 節が指定されていない限り、暗黙のバリアが並列化されたステートメント・ブロックの最後にあります。

#pragma omp master

説明

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



注

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

説明

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 には外部結合があり、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。

注

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

説明

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

注

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:

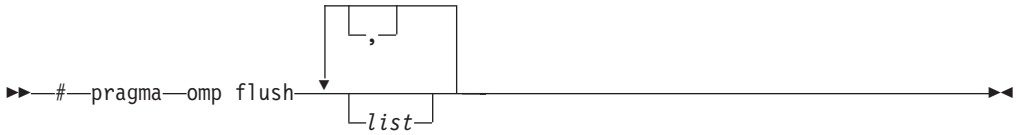
```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#pragma omp flush

説明

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。

注

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。

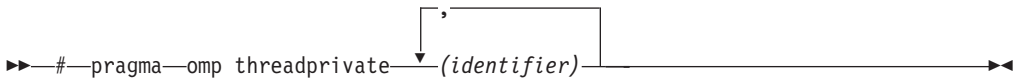
```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#pragma omp threadprivate

説明

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。

注

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。

第 6 章 事前定義マクロ

事前定義マクロは、幾つかのカテゴリーに分類されます。

- 言語機能に関連したマクロ
- XL C/C++ コンパイラーを示すマクロ
- Linux プラットフォームに関連したマクロ

言語機能に関連したマクロ

以下のマクロは、使用可能な C99 機能、GNU C または C++ に関連した機能、および他の IBM 言語拡張機能についてテストすることができます。リストされた機能が指定されたコンパイラー・オプションの下でサポートされている場合、マクロは 1 の値に定義されています。機能がサポートされていない場合、そのマクロは未定義です。事前定義のマクロはすべて保護されています。

表 41. 言語機能の事前定義マクロ

事前定義マクロ名	説明	コンパイラー・オプション:
__ALTIVEC__	ベクトル・データ型に対するサポート	-qaltivec
__C99_BOOL	`_Bool` データ型に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_COMPLEX	複素数データ型に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_COMPLEX_HEADERS__	C99 スタイル複素数ヘッダーに対するサポート	C++ -qlanglvl=extended
__C99_CPLUSCMT	C++ スタイル・コメントに対するサポート	C -qlanglvl=stdc99|extc99 -qcpluscmt
__C99_COMPOUND_LITERAL	複合リテラルに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_DESIGNATED_INITIALIZER	指定された初期化に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_DUP_TYPE_QUALIFIER	重複する型修飾子に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_EMPTY_MACRO_ARGUMENTS	空のマクロ引数に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended C++ -qlanglvl=extended
__C99_FLEXIBLE_ARRAY_MEMBER	フレキシブル配列メンバーに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99__FUNC__	`__func__` キーワードに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended C++ -qlanglvl=extended
__C99_HEX_FLOAT_CONST	16 進浮動小数点定数に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended C++ -qlanglvl=extended
__C99_INLINE	`inline` 関数指定子に対するサポート	C -qlanglvl=stdc99|extc99 -qkeyword=inline
__C99_LLONG	`long long` データ型に対するサポート	C -qlanglvl=stdc99|extc99

表 41. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	コンパイラー・オプション:
__C99_MACRO_WITH_VA_ARGS	変数引数を持つ関数のようなマクロに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended C++ -qlanglvl=extended
__C99_MAX_LINE_NUMBER	`#line` ディレクティブに対する新規制限	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_MIXED_DECL_AND_CODE	混合宣言およびコードに対するサポート	C -qlanglvl=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・リテラルの連結に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_NON_LVALUE_ARRAY_SUB	配列の非 lvalue 添え字に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_NON_CONST_AGGR_INITIALIZER	非定数集合体初期化指定子に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_PRAGMA_OPERATOR	`_Pragma` 演算子に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended C++ -qlanglvl=extended
__C99_REQUIRE_FUNC_DECL	サポートされない暗黙の関数宣言	C -qlanglvl=stdc99
__C99_RESTRICT	`restrict` 修飾子に対するサポート	C -qlanglvl=stdc99|extc99 -qkeyword=restrict C++ -qlanglvl=extended -qkeyword=restrict
__C99_STATIC_ARRAY_SIZE	関数に対する配列パラメーターの `static` キーワードに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_STD_PRAGMAS	標準プラグマに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_TGMATH	tgmath.h の型汎用マクロに対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_UCN	汎用文字名に対するサポート	C -qlanglvl=stdc99|extc99|ucs C++ -qlanglvl=ucs
__C99_VAR_LEN_ARRAY	可変長配列に対するサポート	C -qlanglvl=stdc99|extc99|extc89|extended
__C99_VARIABLE_LENGTH_ARRAY		C++ extended
__IBM__ALIGNOF__	`__alignof__` 演算子に対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_ALTERNATE_KEYWORDS	代替キーワードに対するサポート	C -qlanglvl=extc99|extc89|extended
__IBM_ATTRIBUTES	型、変数、および関数属性に対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_COMPUTED_GOTO	計算された `goto` 文に対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_EXTENSION_KEYWORD	`__extension__` キーワードに対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_GCC_ASM	GNU C インライン `asm` 文に対するサポート	C -qlanglvl=extc99|extc89|extended, -qkeyword=asm, -qasm=gcc C++ -qlanglvl=extended, -qasm=gcc

表 41. 言語機能の事前定義マクロ (続き)

事前定義マクロ名	説明	コンパイラー・オプション:
__IBM_GCC__INLINE__	GNU C __inline__ 指定子に対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_DOLLAR_IN_ID	ID のドル記号に対するサポート	C -qlanglvl=extc99|extc89|extended
__IBM_GENERALIZED_LVALUE	汎用の左辺値に対するサポート	C -qlanglvl=extc99|extc89|extended
__IBM_INCLUDE_NEXT	#include_next プリプロセス・ディレクティブに対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_LABEL_VALUE	値としてのラベルに対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_LOCAL_LABEL	ローカル・ラベルに対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
__IBM_MACRO_WITH_VA_ARGS	variadic マクロ拡張機能に対するサポート	C -qlanglvl=extc99|extc89|extended C++ -qlanglvl=extended
_IBM_NESTED_FUNCTION	ネストされた関数に対するサポート	C -qlanglvl=extc99|extc89|extended
__IBM_PP_PREDICATE	#assert、#unassert、**#cpu**、#machine、および #system プリプロセス・ディレクティブに対するサポート	C -qlanglvl=extc99|extc89|extended
__IBM_PP_WARNING	#warning プリプロセス・ディレクティブに対するサポート	C -qlanglvl=extc99|extc89|extended
__IBM_REGISTER_VARS	指定されたレジスターの変数に対するサポート	C
__IBM_STDCPP_ASM	asm 文に対するサポート。アセンブラー・コードが生成される場合、マクロは 1 の値を持ちます。そうでない場合は 0 になります。	C++
__IBM__TYPEOF__	__typeof__ キーワードに対するサポート	C -qlanglvl=extc99|extc89|extended, -qkeyword=typeof C++ -qlanglvl=extended、-qkeyword=typeof
__IBM_UTF_LITERAL	UTF-16 および UTF-32 ストリング・リテラルに対するサポート	-qutf

XL C/C++ コンパイラーを示すマクロ

XL C/C++ コンパイラーに関連した事前定義マクロは常に定義されています。

事前定義マクロ名	説明
C __IBMC__	XL C コンパイラーのレベルを、バージョン、リリース、およびモディフィケーション番号を表す整数定数として示します。
C++ __IBMCPP__	XL C ++ コンパイラーのレベルを、バージョン、リリース、およびモディフィケーション番号を表す整数定数として示します。
C __xlc__	XL C コンパイラーのレベルを、バージョン、リリース、モディフィケーション、および修正レベルを表示するストリングとして示します。

事前定義マクロ名	説明
__xlC__	XL C++ コンパイラーのレベルを、バージョン、リリース、およびモディフィケーション番号を表す 3 桁の 16 進定数として示します。 XL C コンパイラーを使用すると、このマクロも自動的に定義されます。

Linux 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。

事前定義マクロ名	説明
__BASE_FILE__	基本ソース・ファイルの完全修飾ファイル名に対して定義されます。
_BIG_ENDIAN	1 に定義されます。
__BIG_ENDIAN__	1 に定義されます。
_CALL_SYSV	1 に定義されます。
__CHAR_UNSIGNED__	オプション **-qchars=unsigned** または **#pragma chars(unsigned)** が有効な場合、1 に定義されます。このマクロは、オプション **-qchars=signed** または **#pragma chars(signed)** が有効な場合、未定義になります。
__ELF__	ELF オブジェクト・モデルが有効なことを示すために、このプラットフォームでは 1 に定義されます。
C++ __EXCEPTIONS	**-qeh** オプションが有効な場合、1 に定義されます。そうでない場合は、定義されません。
__GXX_WEAK__	C の場合は定義されません。C++ の場合、このマクロは gcc V3.3 の場合 0、g++ V3.5 の場合は 1 に定義されます。
__HOS_LINUX__	ホストのオペレーティング・システムが Linux の場合、1 に定義されます。そうでない場合は、定義されません。
__linux	1 に定義されます。
__linux__	1 に定義されます。
__OPTIMIZE__	最適化レベル **-O** または **-O2** では 2 に定義され、最適化レベル **-O3** またはそれ以上では 3 に定義されます。
__OPTIMIZE_SIZE__	オプション **-qcompact** と **-O** が設定されている場合、1 に定義されます。そうでない場合は、定義されません。
__powerpc	1 に定義されます。
__powerpc__	1 に定義されます。
__powerpc64__	64 ビット・モードでコンパイルするときは、1 に定義されます。そうでない場合は、定義されません。
__PPC	1 に定義されます。
__PPC__	1 に定義されます。
__PPC64__	64 ビット・モードでコンパイルするときは、1 に定義されます。そうでない場合は、定義されません。
__SIZE_TYPE__	このプラットフォームでの基礎となる型 `size_t` に定義されます。Linux 上で、32 ビット・モードでは、マクロは `unsigned int` として定義されます。64 ビット・モードでは、マクロは `unsigned long` として定義されます。
__TOS_LINUX__	ターゲットのオペレーティング・システムが Linux の場合、1 に定義されます。そうでない場合は、定義されません。
__unix	すべての UNIX 型のプラットフォームでは 1 に定義されます。そうでない場合は、定義されません。
__unix__	すべての UNIX 型のプラットフォームでは 1 に定義されます。そうでない場合は、定義されません。

第 7 章 POWER および PowerPC アーキテクチャーの組み込み関数

組み込み関数は C および C++ C++ のコーディング拡張で、これによりプログラマーは C 関数呼び出しと C 変数の構文を使用してコンパイラー・マシンのプロセッサーの命令セットにアクセスすることができます。IBM POWER および 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 POWER または PowerPC 命令へのアクセスは、C および C++ 言語の標準構成体を使用して生成できません。 他の命令は標準の構造体を通じて生成できますが、組み込み関数を使用すると生成されたコードを正確に制御することができます。これらの命令を直接使用するインライン・アセンブリー言語プログラミングは、XL C/C++ 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/C++ 組み込み関数はアセンブリー言語を通じてハードウェア・レジスターを管理する代わりに、最適化された POWER または 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。

C++ C++ で XL C/C++ 組み込み関数を呼び出すには、ヘッダー・ファイル `builtins.h` をソース・コードに組み込む必要があります。

以下のテーブルには、Linux プラットフォームで使用可能な組み込み関数が記述されています。

固定小数点組み込み関数

プロトタイプ	説明
int __assert1(int, int, int);	カーネル・デバッグのためのトラップ命令を生成します。
void __assert2(int);	カーネル・デバッグのためのトラップ命令を生成します。
unsigned int __cntlz4(unsigned int);	先行ゼロ・カウント。4 バイト整数。
unsigned int __cntlz8(unsigned long long);	先行ゼロ・カウント。8 バイト整数。
unsigned int __cnttz4(unsigned int);	後続ゼロ・カウント。4 バイト整数。
unsigned int __cnttz8(unsigned long long);	後続ゼロ・カウント。8 バイト整数。
signed long long __llabs (signed long long);	引数の絶対値を戻します。
unsigned short __load2r(unsigned short*);	Load Halfword Byte Reversed

プロトタイプ	説明
unsigned int __load4r(unsigned int*);	Load Word Byte Reversed
long long int __mulhd(long long int *ra*, long long int *rb*);	Multiply High Doubleword Signed オペランド *ra* と *rb* の 128 ビットの積から、高位の 64 ビットを戻します。 64 ビット・モードでのみサポートされます。
unsigned long long int __mulhdu(unsigned long long int *ra*, unsigned long long int *rb*);	Multiply High Doubleword Unsigned オペランド *ra* と *rb* の 128 ビットの積から、高位の 64 ビットを戻します。 64 ビット・モードでのみサポートされます。
int __mulhw(int *ra*, int *rb*);	Multiply High Word Signed オペランド *ra* と *rb* の 64 ビットの積から、高位の 32 ビットを戻します。
unsigned int __mulhwu(unsigned int *ra*, unsigned int *rb*);	Multiply High Word Unsigned オペランド *ra* と *rb* の 64 ビットの積から、高位の 32 ビットを戻します。
int __popcnt4(unsigned int);	32 ビット整数のビット・セットの数を戻します。
int __popcnt8(unsigned long long);	64 ビット整数のビット・セットの数を戻します。
unsigned long __popcntb(unsigned long);	ソース・オペランドの各バイトの 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。
int __poppar4(unsigned int);	32 ビット整数に奇数のビット・セットがあった場合、1 を戻します。それ以外は、0 を戻します。
int __poppar8(unsigned long long);	64 ビット整数に奇数のビット・セットがあった場合、1 を戻します。それ以外は、0 を戻します。
unsigned long long __rdlam(unsigned long long *rs*, unsigned int *shift*, unsigned long long *mask*);	Rotate Double Left and AND with Mask *rs* の内容を左に*shift* ビット回転し、回転されたデータをマスクと AND 演算します。*mask* は定数であり、隣接するビット・フィールドを表していなければなりません。
unsigned long long __rldimi(unsigned long long *rs*, unsigned long long *is*, unsigned int *shift*, unsigned long long *mask*);	Rotate Left Doubleword Immediate then Mask Insert *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。*shift* は定数で、<=*shift*<=63 でなければなりません。*mask* は定数であり、隣接するビット・フィールドを表していなければなりません。
unsigned int __rlwimi(unsigned int *rs*, unsigned int *is*, unsigned int *shift*, unsigned int *mask*);	Rotate Left Word Immediate then Mask Insert *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。*shift* は定数で、<=*shift*<=31 でなければなりません。*mask* は定数であり、隣接するビット・フィールドを表していなければなりません。

プロトタイプ	説明
unsigned int __rlwnm(unsigned int *rs*, unsigned int *shift*, unsigned int *mask*);	Rotate Left Word then AND with Mask *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。 *mask* は定数であり、隣接するビット・フィールドを表していなければなりません。
unsigned int __rotatel4(unsigned int *rs*, unsigned int *shift*);	Rotate Left Word *rs* を左方に *shift* ビットだけ回転します。
unsigned long long __rotatel8(unsigned long long *rs*, unsigned long long *shift*);	Rotate Left Doubleword *rs* を左方に *shift* ビットだけ回転します。
void __store2r(unsigned short, unsigned short *);	Store 2-byte Register
void __store4r(unsigned int, unsigned int *);	Store 4-byte Register
void __tdw(long long *a*, long long *b*, unsigned int *TO*);	Trap Doubleword オペランド *a* をオペランド *b* と比較します。この比較の結果、*TO* (これは 5 ビットの定数で、**0** から **31** (包括的) の値を含んでいる) 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。それぞれのビット位置が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 (高位ビット)** *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)。 64 ビット・モードでのみサポートされます。
void __trap(int);	Trap if the Parameter is not Zero
void __trapd (long long);	Trap if the Parameter is not Zero 64 ビット・モードでのみサポートされます。

プロトタイプ	説明
void __tw(int *a*, int *b*, unsigned int *TO*);	Trap Word オペランド *a* をオペランド *b* と比較します。この比較の結果、*TO* (これは 5 ビットの定数で、**0** から **31** (包括的) の値を含んでいる) 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。それぞれのビット位置が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 (高位ビット)** *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)。

浮動小数点組み込み関数

プロトタイプ	説明
double __exp(double);	指数の値を戻します。
double __fabs(double);	倍精度浮動小数点の絶対値を戻します。
float __fabss(float);	単精度浮動小数点の絶対値を戻します。
double __fcfid (double);	Floating Convert from Integer Doubleword 64 ビット符号付き固定小数点オペランドを、倍精度浮動小数点に変換します。
double __fctid (double);	Floating Convert to Integer Doubleword $FPSCR_{RN}$ (浮動小数点状況および制御レジスターの浮動小数点丸め制御フィールド) によって指定された丸めモードを使用して、浮動小数点オペランドを 64 ビットの符号付き固定小数点整数に変換します。
double __fctidz (double);	Floating Convert to Integer Doubleword with Rounding towards Zero 丸めモード「ゼロ方向への丸め」を使用して、浮動小数点オペランドを 64 ビットの符号付き固定小数点整数に変換します。
double __fctiw (double);	Floating Convert to Integer Word $FPSCR_{RN}$ (浮動小数点状況および制御レジスターの浮動小数点丸め制御フィールド) によって指定された丸めモードを使用して、浮動小数点オペランドを 32 ビットの符号付き固定小数点整数に変換します。

プロトタイプ	説明
double __fctiwz (double);	Floating Convert to Integer Word with Rounding towards Zero 丸めモード「ゼロの方向への丸め」を使用して、浮動小数点オペランドを 32 ビットの符号付き固定小数点整数に変換します。
double __fmadd (double, double, double);	Floating Point Multiply-Add
float __fmadds (float, float, float);	Floating Point Multiply-Add Short
double __fmsub(double, double, double);	Floating Point Multiply-Subtract
float __fmsubs (float, float, float);	Floating Point Multiply-Subtract
double __fmul (double, double);	Floating Point Multiply
float __fmuls (float, float);	Floating Point Multiply
double __fnabs(double);	Floating Point Negative Absolute
float __fnabss(float);	Floating Point Negative Absolute
double __fnmadd(double, double, double);	Floating Point Negative Multiply-Add
float __fnmadds (float, float, float);	Floating Point Negative Multiply-Add
double __fnmsub(double, double, double);	Floating Point Negative Multiply-Subtract __fnmsubs (*a*, *x*, *y*) = [- (*a* * *x* - *y*)]
float __fnmsubs (float, float, float);	Floating Point Negative Multiply-Subtract
float __fre (double);	Floating Point Reciprocal __fre (*x*) = [(概算) 1.0/*x*] POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。
float __fres (float);	Floating Point Reciprocal __fres (*x*) = [(概算) 1.0/*x*]
double __frim (double);	「負の無限大方向への丸め」モードを使用して倍精度の引数を整数に丸め、倍精度として値を戻します。 POWER5+ プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5x** に設定されている) 場合にのみサポートされます。
double __frin (double);	「最近似値への丸め」モードを使用して倍精度の引数を整数に丸め、倍精度として値を戻します。 POWER5+ プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5x** に設定されている) 場合にのみサポートされます。
double __frip (double);	「正の無限大方向への丸め」モードを使用して倍精度の引数を整数に丸め、倍精度として値を戻します。 POWER5+ プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5x** に設定されている) 場合にのみサポートされます。

プロトタイプ	説明
double __friz (double);	「ゼロ方向への丸め」モードを使用して倍精度の引数を整数に丸め、倍精度として値を戻します。 POWER5+ プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5x** に設定されている) 場合にのみサポートされます。
double __frsqrte (double);	Floating Point Reciprocal Square Root __frsqrte (*x*) = [(概算) 1.0/sqrt(*x*)]
float __frsqrtes (float);	Floating Point Reciprocal Square Root __frsqrtes (*x*) = [(概算) 1.0/sqrt(*x*)]。 POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。
double __fsel (double, double, double);	Floating Point Select if (*a* >= 0.0) then __fsel (*a*, *x*, *y*) = *x*; else __fsel (*a*, *x*, *y*) = *y*
float __fsels (float, float, float);	Floating point select if (*a* >= 0.0) then __fsels (*a*, *x*, *y*) = *x*; else __fsels (*a*, *x*, *y*) = *y*
double __fsqrt (double);	Floating Point Square Root __fsqrt (*x*) = *x* の平方根
float __fsqrts (float);	Floating Point Square Root __fsqrts (*x*) = *x* の平方根
signed long __labs (signed long);	長整数の絶対値を計算します。
void __mtfsb0(unsigned int *bt*);	Move to Floating Point Status/Control Register (FPSCR) Bit 0 FPSCR のビット *bt* を 0 に設定します。*bt* は定数であり、0<=*bt*<=31 でなければなりません。
void __mtfsb1(unsigned int *bt*);	Moves to FPSCR Bit 1 FPSCR のビット *bt* を 1 に設定します。*bt* は定数であり、0<=*bt*<=31 でなければなりません。
void __mtfsf(unsigned int *flm*, unsigned int *frb*);	Move to FPSCR Fields *frb* の内容を、*flm* によって指定されたフィールド・マスクの制御下の FPSCR に配置します。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。*flm* は定数の 8 ビットのマスクでなければなりません。
void __mtfsfi(unsigned int *bf*, unsigned int *u*);	Move to FPSCR Field Immediate *u* の値を *bf* によって指定された FPSCR フィールドに配置します。*bf* と *u* は定数で、0<=*bf*<=7 および 0<=*u*<=15 でなければなりません。
double __pow(double, double);	1 番目の引数を 2 番目の引数に累乗した値を計算します。
double __readflm();	FPSCR を読み取ります。

プロトタイプ	説明
double __setflm(double);	FPSCR を設定します。
double __setrnd(int);	丸めモードを設定します。 引数の許容値は以下の通りです。 • 0 -- ゼロ方向への丸め • 1 -- 最近似値への丸め • 2 -- 正の無限大方向への丸め • 3 -- 負の無限大方向への丸め
void __stfiw(const int* *addr*, double *value*);	Store Floating Point as Integer Word *value* の下位 32 ビットの内容を、変換せずに、*addr* によってアドレッシングされたストレージ内のワードに保管します。
double __swdiv_nochk(double, double);	`double` 型の浮動小数点の除算。範囲検査なし。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。 引数の制限事項: 無限大に等しい分子は許可されません。無限大に等しい、ゼロ、または正規化されない分母は許可されません。分子と分母の商は正または負の無限大と等しくならない場合があります。 **-qstrict** が有効な場合、その結果は IEEE の除算とビット単位で等しくなります。このシナリオで正しい操作が行われるためには、引数が次の追加の制限事項を満たしている必要があります。分子は 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。分母は 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。分子と分母の商は、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。
double __swdiv(double, double);	`double` 型の浮動小数点の除算。引数の制限はありません。
float __swdivs_nochk(float, float);	`float` 型の浮動小数点の除算。範囲検査なし。引数の制限事項: 無限大に等しい分子は許可されません。無限大に等しい、ゼロ、または正規化されない分母は許可されません。分子と分母の商は正または負の無限大と等しくならない場合があります。
float __swdivs(float, float);	`double` 型の浮動小数点の除算。引数の制限はありません。

同期およびアトミック組み込み関数

プロトタイプ	説明
unsigned int __check_lock_mp (const int* *addr*, int *old_value*, int *new_value*);	Check Lock on Multiprocessor Systems 条件付きで、シングル・ワード変数をアトミック更新します。*addr* は、シングル・ワード変数のアドレスを指定します。*old_value* は、シングル・ワード変数の値に照らし合わせて検査される元の値を指定します。 *new_value* は、シングル・ワード変数に条件付きで割り当てられる新規の値を指定します。ワード変数は、フルワード境界で位置合わせしておかなければなりません。 戻り値: 1. false (偽) の戻り値は、シングル・ワード変数が古い値と同じで、新しい値に設定されたことを示す。 2. true (真) の戻り値は、シングル・ワード変数が古い値と同じでなく、未変更のままであることを示す。
unsigned int __check_lockd_mp (const long long int* *addr*, long long int *old_value*, long long int *new_value*); 64 ビット・モードでのみサポートされます。	Check Lock Doubleword on Multiprocessor Systems 条件付きで、ダブルワード変数をアトミック更新します。*addr* は、ダブルワード変数のアドレスを指定します。*old_value* は、ダブルワード変数の値に照らし合わせて検査される元の値を指定します。 *new_value* は、ダブルワード変数に条件付きで割り当てられる新規の値を指定します。ダブルワード変数は、ダブルワード境界で位置合わせしておかなければなりません。 戻り値: 1. false (偽) の戻り値は、ダブルワード変数が古い値と同じで、新規の値に設定されたことを示す。 2. true (真) の戻り値は、ダブルワード変数が古い値と同じでなく、未変更のままであることを示す。
unsigned int __check_lock_up (const int* *addr*, int *old_value*, int *new_value*);	Check Lock on Uniprocessor Systems 条件付きで、シングル・ワード変数をアトミック更新します。*addr* は、シングル・ワード変数のアドレスを指定します。*old_value* は、シングル・ワード変数の値に照らし合わせて検査される元の値を指定します。 *new_value* は、シングル・ワード変数に条件付きで割り当てられる新規の値を指定します。ワード変数は、フルワード境界で位置合わせしておかなければなりません。 戻り値: • false (偽) の戻り値は、シングル・ワード変数が古い値と同じで、新規の値に設定されたことを示す。 • true (真) の戻り値は、シングル・ワード変数が古い値と同じでなく、未変更のままであることを示す。

プロトタイプ	説明
unsigned int __check_lockd_up (const long long int* *addr*, long long int *old_value*, int long long *new_value*);	Check Lock Doubleword on Uniprocessor systems 条件付きで、ダブルワード変数をアトミック更新します。*addr* は、ダブルワード変数のアドレスを指定します。*old_value* は、ダブルワード変数の値に照らし合わせて検査される元の値を指定します。 *new_value* は、ダブルワード変数に条件付きで割り当てられる新規の値を指定します。ダブルワード変数は、ダブルワード境界で位置合わせしておかなければなりません。 戻り値: • false (偽) の戻り値は、ダブルワード変数が古い値と同じで、新規の値に設定されたことを示す。 • true (真) の戻り値は、ダブルワード変数が古い値と同じでなく、未変更のままであることを示す。 64 ビット・モードでのみサポートされます。
void __clear_lock_mp (const int* *addr*, int *value*);	Clear Lock on Multiprocessor Systems アドレス *addr* にあるシングル・ワード変数に、*value* をアトミック保管します。ワード変数は、フルワード境界で位置合わせしておかなければなりません。
void __clear_lockd_mp (const long long int* *addr*, long long int *value*);	Clear Lock Doubleword on Multiprocessor Systems アドレス *addr* にあるダブルワード変数に、*value* をアトミック保管します。ダブルワード変数は、ダブルワード境界で位置合わせしておかなければなりません。 64 ビット・モードでのみサポートされます。
void __clear_lock_up (const int* *addr*, int *value*);	Clear Lock on Uniprocessor Systems アドレス *addr* にあるシングル・ワード変数に、*value* をアトミック保管します。ワード変数は、フルワード境界で位置合わせしておかなければなりません。
void __clear_lockd_up (const long long int* *addr*, long long int *value*);	Clear Lock Doubleword on Uniprocessor Systems アドレス *addr* にあるダブルワード変数に、*value* をアトミック保管します。ダブルワード変数は、ダブルワード境界で位置合わせしておかなければなりません。 64 ビット・モードでのみサポートされます。

プロトタイプ	説明
int __compare_and_swap(volatile int* *addr*, int* *old_val_addr*, int *new_val*);	シングル・ワード変数の内容を、保管されている古い値と比較するアトミック演算を実行します。値が等しい場合、新規値がシングル・ワード変数に保管されて、1 が戻されます。等しくない場合は、シングル・ワード変数は更新されず、0 が戻されます。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。 __compare_and_swap 関数は、シングル・ワード値が最後の読み取り以降変更されていないときにのみ更新を必要とする場合に役立ちます。入力パラメーター *addr* として取られるメモリー・ロケーションは 4 バイトの位置合わせでなければなりません。 __compare_and_swap がロック・プリミティブとして使用される場合は、__isync 組み込み関数をクリティカル・セクションの最初に挿入してください。
int __compare_and_swaplp(volatile long* *addr*, long* *old_val_addr*, long *new_val*);	ダブルワード変数の内容を、保管されている古い値と比較するアトミック演算を実行します。値が等しい場合、新規値がダブルワード変数に保管されて、1 が戻されます。等しくない場合は、ダブルワード変数は更新されず、0 が戻されます。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。入力パラメーター *addr* として取られるメモリー・ロケーションは 8 バイトの位置合わせでなければなりません。 この関数は、ダブルワードが最後の読み取り以降変更されていないときにのみ更新を必要とする場合に役立ちます。__compare_and_swaplp がロック・プリミティブとして使用される場合は、__isync 組み込み関数をクリティカル・セクションの最初に挿入してください。 64 ビット・モードでのみサポートされます。
void __eieio(void);	Enforce In-order Execution of Input/Output 主メモリーで __eioeio への呼び出しより前のすべての入出力ストレージ・アクセス命令が完了してからでないと、その関数呼び出しの後の入出力ストレージ・アクセス命令を実行できないことを保証します。 この組み込み関数は、ロード/保管アクセスの実行順序が重大なところで共用データ命令を管理するのに有用です。この関数は、他の同期命令と異なり、パフォーマンスを犠牲にすることなく、I/O 保管の制御に必要な機能を提供することができます。

プロトタイプ	説明
int __fetch_and_add(volatile int* *addr*, int *val*);	単一アトミック演算で *addr* によって指定されたシングル・ワードを、*val* によって指定された量だけ増分します。 戻り値はこのメモリー・ロケーションの元のコンテンツと等しくなります。*addr* によって指定されたアドレスは 4 バイトで位置合わせされている必要があります。 この演算は、カウンター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。
long __fetch_and_addlp(volatile long* *addr*, long *val*);	単一アトミック演算で *addr* によって指定されたダブルワードを、*val* によって指定された量だけ増分します。戻り値はこのメモリー・ロケーションの元のコンテンツと等しくなります。*addr* によって指定されたアドレスは 8 バイトで位置合わせされている必要があります。 この演算は、カウンター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。 64 ビット・モードでのみサポートされます。
unsigned int __fetch_and_and(volatile unsigned int* *addr*, unsigned int *val*);	単一アトミック演算で、*addr* によって指定されたシングル・ワードのビットを、その値を入力 *val* パラメーターと AND 演算することにより消去します。戻り値はこのメモリー・ロケーションの元のコンテンツと等しくなります。*addr* によって指定されたアドレスは 4 バイトで位置合わせされている必要があります。 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。
unsigned long __fetch_and_andlp(volatile unsigned long* *addr*, unsigned long *val*);	単一アトミック演算で、*addr* によって指定されたダブルワードのビットを、その値を入力 *val* パラメーターと AND 演算することにより消去します。戻り値はこのメモリー・ロケーションの元のコンテンツと等しくなります。*addr* によって指定されたアドレスは 8 バイトで位置合わせされている必要があります。 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。 64 ビット・モードでのみサポートされます。

プロトタイプ	説明
unsigned int __fetch_and_or(volatile unsigned int* *addr*, unsigned int*val*);	単一アトミック演算で、*addr* によって指定されたシングル・ワードのビットを、その値を入力 *val* パラメーターと OR 演算することにより設定します。戻り値はこのメモリー・ロケーションの元のコンテンツと等しくなります。*addr* によって指定されたアドレスは 4 バイトで位置合わせされている必要があります。 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。
unsigned long __fetch_and_orlp(volatile unsigned long* *addr*, unsigned long *val*;	単一アトミック演算で、*addr* によって指定されたダブルワードのビットを、その値を入力 *val* パラメーターと OR 演算することにより設定します。戻り値はこのメモリー・ロケーションの元のコンテンツと等しくなります。*addr* によって指定されたアドレスは 8 バイトで位置合わせされている必要があります。 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。 64 ビット・モードでのみサポートされます。
unsigned int __fetch_and_swap(volatile unsigned int* *addr*, unsigned int*val*);	単一アトミック演算で、*addr* によって指定されたシングル・ワードを値または入力 *val* パラメーターに設定し、メモリー・ロケーションのオリジナル・コンテンツを戻します。*addr* によって指定されるアドレスは 4 バイトで位置合わせでなければなりません。 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。
unsigned long __fetch_and_swaplp(volatile unsigned long* *addr*, unsigned long *val*);	単一アトミック演算で、*addr* によって指定されたダブルワードを値または入力 *val* パラメーターに設定し、メモリー・ロケーションのオリジナル・コンテンツを戻します。*addr* によって指定されたアドレスは 8 バイトで位置合わせされている必要があります。 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。 64 ビット・モードでのみサポートされます。
void __iospace_eieio(void);	`__eieio` 組み込み関数 (上記) の代替名。
void __iospace_lwsync(void);	`__lwsync` 組み込み関数 (下記) の代替名。
void __iospace_sync(void);)	`__sync` 組み込み関数 (下記) の代替名。

プロトタイプ	説明
void __isync(void);	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。
long __ldarx(volatile long* addr);	Load Doubleword and Reserve Indexed *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。 *addr* は 8 バイトで位置合わせされている必要があります。 後続の __stdcx 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。この 2 つの組み込み関数は一緒に機能し、保管が正常に行われた場合、__ldarx 関数が実行されてから __stdcx 関数が完了するまでの間に、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のと同じ効果を持ち、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 組み込み関数の説明については、 330 ページの『各種組み込み関数』を参照してください)。 64 ビット・モードでのみサポートされます。
int __lwarx(volatile int* addr);	Load Word and Reserve Indexed *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。64 ビット・モードでは、コンパイラーは符号拡張された結果を戻します。 *addr* は 4 バイトで位置合わせされている必要があります。 後続の __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。この 2 つの組み込み関数は一緒に機能し、保管が正常に行われた場合、__lwarx 関数が実行されてから __stwcx 関数が完了するまでの間に、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 __lwarx 組み込み関数の前後に挿入したのと同じ効果を持ち、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。
void __lwsync(void);	__lwsync への呼び出しの前のすべての保管命令が完了してからでないと、その関数を実行したプロセッサー上で新規命令を実行できないことを保証します。__lwsync は各プロセッサーからの確認を待たないため、こうすると、パフォーマンス上の影響を最小限にして複数のプロセッサー間の同期化を実行することができます。

プロトタイプ	説明
int __stdcx(volatile long* addr, long val);	Store Doubleword Conditional Indexed *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、指定されたメモリー・ロケーションの更新が成功すると 1 を戻し、失敗すると 0 を戻します。*addr* は 8 バイトで位置合わせされている必要があります。 先行する __ldarx 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。この 2 つの組み込み関数は一緒に機能し、保管が正常に行われた場合、__ldarx 関数が実行されてから __stdcx 関数が完了するまでの間に、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のと同じ効果を持ち、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。 64 ビット・モードでのみサポートされます。
int __stwcx(volatile int* addr, int val);	Store Word Conditional Indexed *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、指定されたメモリー・ロケーションの更新が成功すると 1 を戻し、失敗すると 0 を戻します。*addr* は 4 バイトで位置合わせされている必要があります。 先行する __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。この 2 つの組み込み関数は一緒に機能し、保管が正常に行われた場合、__lwarx 関数が実行されてから __stwcx 関数が完了するまでの間に、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 __stwcx 組み込み関数の前後に挿入したのと同じ効果を持ち、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。
void __sync(void);	関数呼び出し __sync より前のすべての命令が完了してからでないと、その関数呼び出しの後の命令を実行できないことを保証します。

キャッシュに関連した組み込み関数

プロトタイプ	説明
void __dcbt (void *);	Data Cache Block Touch 指定のアドレスを含むメモリーのブロックをデータ・キャッシュ内にロードします。
void __dcbz (void *);	Data Cache Block set to Zero 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。
void __prefetch_by_load(const void*);	明示的ロードを使用してメモリー・ロケーションにタッチします。
void __prefetch_by_stream(const int, const void*);	明示的ストリームを使用してメモリー・ロケーションにタッチします。
void __protected_stream_count(unsigned int *unit_cnt*, unsigned int *ID*);	ID *ID* の限定長さの protected ストリームに、*unit_cnt* 行のキャッシュ行を設定します。*unit_cnt* は、0 から 1023 の値を持つ整数でなければなりません。ストリーム *ID* は 0 から 15 の整数値を持っていなければなりません。 POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。
void __protected_stream_go();	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。 POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。
void __protected_stream_set(unsigned int *direction*, const void* *addr*, unsigned int *ID*);	ID *ID* を使用して限定長さの protected ストリームを確立します。これはキャッシュ行 (*addr*) から始まり、*direction* の値に応じて、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。 *direction* は 1 (前方) または 3 (後方) の値を持っていなければなりません。ストリーム *ID* は 0 から 15 の整数値を持っていなければなりません。 POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。

プロトタイプ	説明
void __protected_unlimited_stream_set_go (unsigned int *direction*, const void* *addr*, unsigned int *ID*);	ID *ID* を使用して非限定長さの protected ストリームを確立します。これはキャッシュ行 (*addr*) から始まり、*direction* の値に応じて、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。 *Direction* は 1 (前方) または 3 (後方) の値を持っていなければなりません。ストリーム *ID* は 0 から 15 の整数値を持っていなければなりません。 POWER5 または PowerPC 970 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5**、**pwr5x**、または **ppc970** に設定されている) 場合にのみサポートされます。
void __protected_stream_stop(unsigned int *ID*);	ID *ID* を持つ protected ストリームのプリフェッチを停止します。 POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。
void __protected_stream_stop_all();	すべての protected ストリームのプリフェッチを停止します。 POWER5 プロセッサーに対してターゲット・アーキテクチャーが指定されている (**-qarch** が **pwr5** または **pwr5x** に設定されている) 場合にのみサポートされます。

ブロックに関連した組み込み関数

プロトタイプ	説明
void __bcopy(char *, char *, size_t);	64 ビット・システムでのブロック・コピー
void __bzero(void *, size_t);	Block zero

各種組み込み関数

プロトタイプ	説明
void __alignx(int *alignment*, const void **address*);	指定された *address* 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します。*alignment* は、ゼロ以上で 2 乗の値を持つ正の定数整数でなければなりません。

プロトタイプ	説明
void __builtin_return_address (unsigned int *level*);	現行関数またはその呼び出し側の 1 つの戻りアドレスを戻します。ここで、*level* 引数は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戻りアドレスを戻します。 **注:** 1. スタックの先頭に到達すると、関数は 0 を戻します。 2. *level* は 0 から 63 の範囲でなければなりません。これ以外の場合は、警告メッセージが発行され、コンパイルが停止します。 3. 関数がインライン化されている場合、戻りアドレスは戻り先の関数の戻りアドレスに対応します。 4. コンパイラーの最適化は、インライン化などの最適化により、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。
void __builtin_frame_address (unsigned int *level*);	現行関数、またはその呼び出し側の 1 つの関数フレームのアドレスを戻します。ここで、*level* 引数は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・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・アドレスを戻します。 **注:** 1. スタックの先頭に到達すると、関数は 0 を戻します。 2. *level* は 0 から 63 の範囲でなければなりません。これ以外の場合は、警告メッセージが発行され、コンパイルが停止します。 3. 関数がインラインの場合、フレーム・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。 4. コンパイラーの最適化は、インライン化などの最適化により、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。
void __fence(void);	コード・モーション、またはマシン・インストラクションの再配列を行うコンパイラーの最適化に対するバリアとして機能します。コンパイラーの最適化は、`__fence` 呼び出しのロケーションより先までマシン・インストラクションを移動しません。この構成は、最適化が使用可能になっている場合に、コンパイラーによって生成されるオブジェクト・コードでの命令の順序を保証するために有用です。

プロトタイプ	説明
unsigned long __mftb();	Move from Time Base 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、__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、ダブルワード時間基準レジスター全体を戻します。 **注:** __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。
unsigned int __mftbu();	Move from Time Base Upper 32 ビット・コンパイル・モードでは、時間基準レジスターの上位のワードを戻し、__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。64 ビット・モードでは、 ダブルワード時間基準レジスター全体を戻します。したがって、__mftbu と別個に使用することは不要です。 **注:** __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。
unsigned long __mfmsr (void);	MSR の内容を、指定された GPR のビット 32 から 63 に移動します。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。
unsigned __mfspr(const int *registerNumber*);	指定された特殊目的レジスター *registerNumber* の値を戻します。*registerNumber* はコンパイル時に既知でなければなりません。
void __mtmsr (unsigned long);	指定された GPR のビット 32 から 63 の内容を、MSR に移動します。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。
void __mtspr(const int *registerNumber*, unsigned long *value*);	特殊目的レジスター *registerNumber* の値を、符号なし長整数値 *value* で設定します。両方の値がコンパイル時に既知でなければなりません。

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。 **omp_** 関数の関数定義は、omp.h ヘッダー・ファイルにあります。

プロトタイプ	説明
int omp_get_num_threads(void);	この関数が呼び出された並列領域を実行しているチームに現在存在するスレッドの数を戻します。

プロトタイプ	説明
void omp_set_num_threads(int *num_threads*);	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。値 *num_threads* は正整数でなければなりません。 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、omp_set_num_threads ライブラリー関数または 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。
int omp_get_max_threads(void);	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。
int omp_get_thread_num(void);	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。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。
int omp_get_num_procs(void);	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。
int omp_in_parallel(void);	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。
void omp_set_dynamic(int dynamic_threads);	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。
int omp_get_dynamic(void);	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。
void omp_set_nested(int nested);	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。
int omp_get_nested(void);	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。

void omp_init_lock(omp_lock_t **lock*);
 	
void omp_init_nest_lock(omp_nest_lock_t **lock*);
	これらの関数は、ロックを初期化する唯一の手段を提供します。各関数は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。

void omp_destroy_lock(omp_lock_t **lock*);
 	
void omp_destroy_nest_lock(omp_nest_lock_t **lock*);
	これらの関数は、指定されたロック変数 *lock* が未初期化であることを保証します。

void omp_set_lock(omp_lock_t *lock);
 	
void omp_set_nest_lock(omp_nest_lock_t *lock);
	これらの各関数は、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。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すでに所有されている場合は使用可能です。

void omp_unset_lock(omp_lock_t *lock);
 	
void omp_unset_nest_lock(omp_nest_lock_t *lock);
	これらの関数は、ロックの所有権を解放する手段を提供します。

int omp_test_lock(omp_lock_t *lock);
 	
int omp_test_nest_lock(omp_nest_lock_t *lock);
	これらの関数は、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。

double omp_get_wtime(void);	固定された開始時刻からの経過時間を戻します。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。

プロトタイプ	説明
double omp_get_wtick(void);	クロックの刻みの間の秒数を戻します。

注: 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 は影響力を持たず、omp_get_nested は常に 0 を戻します。

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

- 315 ページの『第 7 章 POWER および PowerPC アーキテクチャーの組み込み関数』

付録 A. 再配布可能ライブラリー

XL C/C++ を使用してアプリケーションを作成した場合、以下の 1 つ以上の再配布可能ライブラリーが使用される可能性があります。アプリケーションを出荷する場合は、そのアプリケーションのユーザーがライブラリーを含むパッケージを持っていることを確認してください。ユーザーが必要なライブラリーを確実に使用できるようにするには、次のいずれかを実行することができます。

- ライブラリーを含むパッケージをアプリケーションに同梱することができます。パッケージはインストール CD の該当する Linux ディストリビューション・ディレクトリーの下の `rpms/` ディレクトリーに保管されます。
- ユーザーがライブラリーを含むパッケージを、XL C/C++ のサポート Web サイトからダウンロードすることができます。

  http://www.ibm.com/software/awdtools/xlcpp/support/

以下のパッケージの配布に関連したライセンス交付要件については、CD 上の `LicAgree.pdf` を参照してください。

表 42. 再配布可能ライブラリー

パッケージ名	ライブラリー (およびデフォルト・インストール・パス)	説明
vacpp.rte	/opt/ibmcmp/lib/libibmc++.so.1 /opt/ibmcmp/lib64/libibmc++.so.1	C++ ランタイム・ライブラリー
xlsmp.rte	/opt/ibmcmp/lib/libxlomp_ser.so.1 /opt/ibmcmp/lib/libxlsmp.so.1 /opt/ibmcmp/lib64/libxlomp_ser.so.1 /opt/ibmcmp/lib64/libxlsmp.so.1	SMP (OMP) ランタイム・ライブラリー
xlsmp.msg.rte	/opt/ibmcmp/msg/en_US/smprt.cat /opt/ibmcmp/msg/en_US.utf8/smprt.cat	SMP メッセージ・カタログ (英語)
	/opt/ibmcmp/msg/ja_JP/smprt.cat /opt/ibmcmp/msg/ja_JP.eucjp/smprt.cat /opt/ibmcmp/msg/ja_JP.utf8/smprt.cat	SMP メッセージ・カタログ (日本語)
	/opt/ibmcmp/msg/zh_CN/smprt.cat /opt/ibmcmp/msg/zh_CN.gb18030/smprt.cat /opt/ibmcmp/msg/zh_CN.gb2312/smprt.cat /opt/ibmcmp/msg/zh_CN.gbk/smprt.cat /opt/ibmcmp/msg/zh_CN.utf8/smprt.cat	SMP メッセージ・カタログ (中国語)

.

付録 B. ASCII 文字セット

XL C/C++ は情報交換用米国標準コード (ASCII) 文字セットを使用します。

以下の表は、標準 ASCII 文字を昇順数値順序で、対応する 10 進値、8 進値、および 16 進値とともにリストしたものです。また、制御文字も **Ctrl-** 表記とともに示しています。例えば、復帰 (ASCII シンボルの **CR**) は **Ctrl-M** として示され、**Ctrl** キーと **M** キーを同時に押すことによって入力できます。

10 進数値	8 進数値	16 進数値	制御文字	ASCII シンボル	意味
0	0	00	Ctrl-@	NUL	ヌル
1	1	01	Ctrl-A	SOH	ヘッディング開始
2	2	02	Ctrl-B	STX	テキスト開始
3	3	03	Ctrl-C	ETX	テキスト終結
4	4	04	Ctrl-D	EOT	伝送終了
5	5	05	Ctrl-E	ENQ	照会
6	6	06	Ctrl-F	ACK	確認
7	7	07	Ctrl-G	BEL	ベル
8	10	08	Ctrl-H	BS	バックスペース
9	11	09	Ctrl-I	HT	水平タブ
10	12	0A	Ctrl-J	LF	改行
11	13	0B	Ctrl-K	VT	垂直タブ
12	14	OC	Ctrl-L	FF	用紙送り
13	15	0D	Ctrl-M	CR	復帰
14	16	0E	Ctrl-N	SO	シフトアウト
15	17	0F	Ctrl-O	SI	シフトイン
16	20	10	Ctrl-P	DLE	伝送制御拡張
17	21	11	Ctrl-Q	DC1	装置制御 1
18	22	12	Ctrl-R	DC2	装置制御 2
19	23	13	Ctrl-S	DC3	装置制御 3
20	24	14	Ctrl-T	DC4	装置制御 4
21	25	15	Ctrl-U	NAK	否定応答
22	26	16	Ctrl-V	SYN	同期信号
23	27	17	Ctrl-W	ETB	伝送ブロック終結
24	30	18	Ctrl-X	CAN	取り消し
25	31	19	Ctrl-Y	EM	メディア終端
26	32	1A	Ctrl-Z	SUB	置換
27	33	1B	Ctrl-[	ESC	エスケープ
28	34	1C	Ctrl-¥	FS	ファイル分離
29	35	1D	Ctrl-]	GS	グループ分離

10 進数値	8 進数値	16 進数値	制御文字	ASCII シンボル	意味
30	36	1E	Ctrl-^	RS	レコード分離
31	37	1F	Ctrl-_	US	ユニット分離
32	40	20		SP	ディジット選択
33	41	21		!	感嘆符
34	42	22		“	二重引用符
35	43	23		#	ポンド記号、番号記号
36	44	24		$	ドル記号
37	45	25		%	パーセント記号
38	46	26		&	アンパーサンド
39	47	27		’	アポストロフィ
40	50	28		(	左括弧
41	51	29		)	右括弧
42	52	2A		*	アスタリスク
43	53	2B		+	加算記号
44	54	2C		,	コンマ
45	55	2D		-	減算記号
46	56	2E		.	ピリオド
47	57	2F		/	右スラッシュ
48	60	30		0	
49	61	31		1	
50	62	32		2	
51	63	33		3	
52	64	34		4	
53	65	35		5	
54	66	36		6	
55	67	37		7	
56	70	38		8	
57	71	39		9	
58	72	3A		:	コロン
59	73	3B		;	セミコロン
60	74	3C		<	より小
61	75	3D		=	等号
62	76	3E		>	より大
63	77	3F		?	疑問符
64	100	40		@	アットマーク
65	101	41		A	
66	102	42		B	
67	103	43		C	
68	104	44		D	
69	105	45		E	

10 進数値	8 進数値	16 進数値	制御文字	ASCII シンボル	意味
70	106	46		F	
71	107	47		G	
72	110	48		H	
73	111	49		I	
74	112	4A		J	
75	113	4B		K	
76	114	4C		L	
77	115	4D		M	
78	116	4E		N	
79	117	4F		O	
80	120	50		P	
81	121	51		Q	
82	122	52		R	
83	123	53		S	
84	124	54		T	
85	125	55		U	
86	126	56		V	
87	127	57		W	
88	130	58		X	
89	131	59		Y	
90	132	5A		Z	
91	133	5B		[	左大括弧
92	134	5C		¥	円記号
93	135	5D		]	右大括弧
94	136	5E		^	ハット、曲折アクセント記号、挿入記号 (hat, circumflex, caret)
95	137	5F		_	下線
96	140	60		`	抑音符号
97	141	61		a	
98	142	62		b	
99	143	63		c	
100	144	64		d	
101	145	65		e	
102	146	66		f	
103	147	67		g	
104	150	68		h	
105	151	69		i	
106	152	6A		j	
107	153	6B		k	

10 進数値	8 進数値	16 進数値	制御文字	ASCII シンボル	意味
108	154	6C		l	
109	155	6D		m	
110	156	6E		n	
111	157	6F		o	
112	160	70		p	
113	161	71		q	
114	162	72		r	
115	163	73		s	
116	164	74		t	
117	165	75		u	
118	166	76		v	
119	167	77		w	
120	170	78		x	
121	171	79		y	
122	172	7A		z	
123	173	7B		{	左中括弧
124	174	7C		|	論理 OR、垂直バー
125	175	7D		}	右中括弧
126	176	7E		~	同様、波形記号
127	177	7F		DEL	削除

特記事項

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

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

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

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、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
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

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

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

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

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。

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

著作権使用許諾:

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

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

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. 1998, 2002. All rights reserved.

プログラミング・インターフェース情報

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

汎用プログラミング・インターフェースにより、お客様はこのプログラムのツールのサービスを得るアプリケーション・ソフトウェアを作成することができます。

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

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

商標

以下は、IBM Corporation の商標です。

AIX
IBM
PowerPC
pSeries
VisualAge

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

業界標準

以下の標準をサポートしています。

- C 言語は、International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)) との整合性があります。
- C++ 言語は、International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998) との整合性があります。
- C および C++ 言語は、OpenMP C および C++ アプリケーション・プログラミング・インターフェース、バージョン 1.0 に整合しています。

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

I

O

P

V

SD88-6729-00

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