IBM XL C/C++ Advanced Edition for Linux

コンパイラー・リファレンス

バージョン 7.0

SC88-9975-01
(英文原典：SC09-7942-01)

IBM XL C/C++ Advanced Edition for Linux

コンパイラー・リファレンス

バージョン 7.0

SC88-9975-01
(英文原典：SC09-7942-01)

ご注意

本書および本書で紹介する製品をご使用になる前に、 421 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM XL C/C++ Advanced Edition for Linux (プログラム番号 5724-K77) のバージョン 7 リリース 0 モディフィケーション 1、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： SC09-7942-01
IBM XL C/C++ Advanced Edition for Linux
Compiler Reference
Version 7.0

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2005.1

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

構文図の読み方

本節では、構文図の読み方について説明します。構文図のシンボル、図に含まれる項目 (キーワード、変数、区切り文字、演算子、フラグメント参照、オペランド) を定義し、これらの項目が含まれている構文例を示します。

構文図は、コマンド・ステートメントを構成する順序および部分 (オプションおよび引き数) を図で表します。水平線上のメインパスに沿って、右から左へ、また、上から下へ読んでいきます。

シンボル

以下は、構文図に出現するシンボルです。

シンボル	定義
▸▸──	構文図の先頭を表します。
──▸	構文図が次の行に続くことを表します。
▸──	構文が直前の行から続いていることを表します。
──▸◂	構文図の終わりを表します。

構文項目

構文図には、多くのさまざまな項目が含まれています。構文項目には、以下のものがあります。

- キーワード - コマンド名またはその他のリテラル情報です。
- 変数 - 変数は、小文字のイタリックで表され、ユーザーが指定可能な値の名前を表します。
- 区切り文字 - 区切り文字は、キーワード、変数、または演算子の始まりまたは終わりを表します。例えば、左括弧は区切り文字です。
- 演算子 - 演算子には、加算 (+)、減算 (-)、乗算 (*)、除法 (/)、等号 (=)、および実行する必要のあるその他の数学演算子が含まれます。
- フラグメント参照 - 構文図の一部で、より詳しく示すために図から分離したものです。
- 分離文字 - 分離文字は、キーワード、変数、または演算子を区切ります。例えば、コンマ (,) は分離文字です。

キーワード、変数、および演算子は、必須、オプショナル、またはデフォルトとして表示されます。フラグメント、区切り文字、および区切り文字は、必須またはオプショナルとして表示されます。

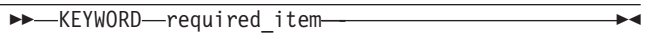
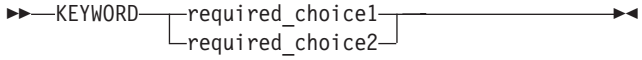
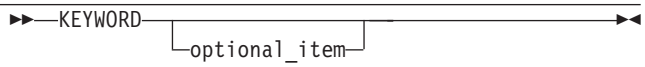
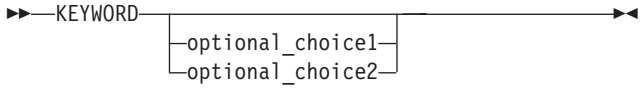
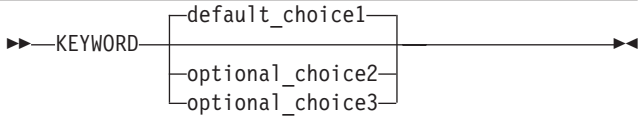
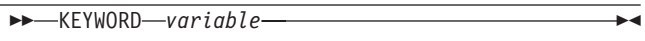
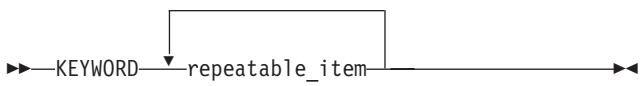
項目タイプ	定義
必須	必須項目は、水平線のメインパス上に表示されます。
オプショナル	オプショナル項目は、水平線のメインパスの下に表示されます。

デフォルト　　デフォルト項目は、水平線のメインパスの上に表示されます。

構文例

次の表では、構文例を示します。

表 1. 構文例

項目	構文例
必須項目 必須項目は、水平線のメインパス上に表示されます。下記の項目を指定しなければなりません。	
必須の選択項目 必須の選択項目 (2 つ以上の項目) は、水平線のメインパス上に縦に重ねて表示されます。重なった項目からいずれか 1 つを選択しなければなりません。	
オプショナル項目 オプショナル項目は、水平線のメインパスの下に表示されます。	
オプショナルの選択項目 オプショナルの選択項目 (2 つ以上の項目) は、水平線のメインパスの下に縦に重ねて表示されます。重なった項目からいずれか 1 つを選択できます。	
デフォルト デフォルト項目は、水平線のメインパスの上に表示されます。これ以外の項目 (必須またはオプショナル) は、水平線のメインパス上 (必須)、または水平線のメインパスの下 (オプショナル) に表示されます。下記の例は、オプショナル項目のあるデフォルトを表したものです。	
変数 変数は、小文字のイタリックで表されます。これらは、名前または値を示します。	
反復可能な項目 水平線のメインパスの上を左に戻る矢印は、繰り返しの可能な項目を示しています。 反復可能な項目のグループの上を左に戻る矢印は、そのいずれか 1 つの項目を選択できるか、または単一項目を繰り返すことができることを示しています。	

第 1 部 概要

XL C/C++

IBM XL C/C++ Advanced Edition for Linux を使用して、 C および C++ プログラム・ソース・ファイルを実行可能オブジェクト・モジュールにコンパイルすることができます。

注: このトピック全体において、 **xlc** および **xlc++** コマンド呼び出しは、コンパイラーの動作を説明するために使用します。ただし、ユーザーの特定の環境が必要とする場合は、他の形式のコンパイラー呼び出しコマンドで代用でき、特に指定しない限り、コンパイラー・オプションの使用法はそのままです。

XL C/C++ コンパイラーについて詳しくは、本節の下記のトピックを参照してください。

- 『コンパイラー・モード』
- 5 ページの『コンパイラー・オプション』
- 6 ページの『入力ファイルの種類』
- 8 ページの『出力ファイルの種類』
- 10 ページの『コンパイラー・メッセージおよびリスト情報』

コンパイラー・モード

XL C/C++ コンパイラー呼び出しコマンドのさまざまな形式が、さまざまなレベルの C および C++ 言語をサポートしています。

たいていの場合、C++ ソース・ファイルをコンパイルするには **xlc++** コマンドを、C ソース・ファイルをコンパイルするには **xlc** コマンドを使用する必要があります。 C および C++ オブジェクト・ファイルが両方ともある場合は、 **xlc++** を使用してリンクしてください。

ユーザーの環境で必要であれば、コマンドの別の形式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。

コンパイラー呼び出し	
基本	**特殊**
xlC	xlC_r
xlc++	xlc++_r
xlc	xlc_r
cc	cc_r
c99	c99_r
c89	c89_r
xlCcore	xlCcore_r

基本コンパイラー呼び出しコマンドは、上記の各行の最初の項目として現れます。以下の基準で、基本呼び出しを選択してください。

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++ である場合、適正なランタイム・ライブラリーとリンクさせるには、この呼び出しを使用しなければなりません。ソース・ファイルは、 **-qalias=ansi** セットを使用してコンパイルされます。 サフィックス **.c** を持つファイルは、**-+** コンパイラー・オプションを使用していないと見なし、**-qlanglvl=extc89** が有効なとき、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
c99	ISO C99 言語フィーチャーをサポートする、C ソース・ファイルのコンパイラーを呼び出します。 ISO C99 規格に完全に適合させるためは、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なければなりません。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__

c89	ISO C89 言語フィーチャーがサポートされる、C ソース・ファイル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__ ANSI 規格 (ISO/IEC 9899:1990) に厳密に準拠させる場合は、この呼び出しを使用してください。
xlCcore **xlc++core**	両方とも **xlC** および **xlc++** のために上記で説明したコンパイラーを呼び出しますが、このコンパイラーはランタイム・ライブラリーのコアにのみリンクします。 XL C/C++ によって提供されるランタイム・ライブラリー以外のランタイム・ライブラリーにアプリケーションをリンクしたい場合は、この呼び出しを使用します。

IBM XL C/C++ Advanced Edition for Linux は、下記のように、基本コンパイラー呼び出しで、**_r** バリエーションを提供します。

_r- サフィックス呼び出し すべての **_r** サフィックス呼び出しは、追加のマクロ名 **_THREAD_SAFE** および **__VACPP_MULTI__** を定義し、 **-lpthreads** 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。Posix スレッド化アプリケーションを作成したい場合は、これらのコマンドを使用してください。

関連タスク

23 ページの『コンパイラーの起動』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
287 ページの『汎用プラグマ』
355 ページの『並列処理を制御するプラグマ』
264 ページの『threaded』

コンパイラー・オプション

コンパイラー・オプションは、広範囲の種類の関数を実行します。例えば、コンパイラー特性の設定、オブジェクト・コードおよび作成されるコンパイラー出力の記述、いくつかのプリプロセッサー関数の実行などです。以下の 3 つの方法のいずれかを使用してコンパイラー・オプションを指定することができます。

- コマンド行で
- 構成ファイル (.cfg) で
- ソース・プログラム内で

上記の方法で明示的に設定されていないコンパイラー・オプションのほとんどについて、コンパイラーはデフォルト設定されているものと見なします。

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、一貫したやり方で以下のとおり解決します。

多くの場合、コンパイラーは、以下の順序で、対立するオプションまたは矛盾する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るコンパイラー・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。対立または競合するコンパイラー・オプションが、同じコマンド行コンパイラー呼び出しに指定されている場合、その呼び出しの後のほうに指定されているオプションが優先されます。
3. 構成ファイルに指定されるコンパイラー・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。

この優先順位に従わないオプション競合については、32 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。

関連タスク

23 ページの『コンパイラーの起動』
25 ページの『コンパイラー・オプションの指定』
32 ページの『矛盾するコンパイラー・オプションの解決』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
287 ページの『汎用プラグマ』
355 ページの『並列処理を制御するプラグマ』

入力ファイルの種類

コンパイラーは、現れた順にソース・ファイルを処理します。コンパイラーが指定されたソース・ファイルを見つけることができない場合は、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・エディターは実行せず、一時オブジェクト・ファイルは除去されます。

ユーザー・プログラムは、いくつかのソース・ファイルから構成することができます。これらのソース・ファイルはすべて、コンパイラーを一回だけ呼び出すことで、一度にコンパイルすることができます。複数ソース・ファイルを、コンパイラーの一回の呼び出しを使用してコンパイルすることは可能ですが、呼び出しごとにコマンド行で指定できるコンパイラー・オプションは、 1 セットのみになります。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。

デフォルトでは、コンパイラーは、すべての指定されたソース・ファイルをプリプロセスして、コンパイルします。通常このデフォルトを使用しますが、コンパイラ

ーを使用して、コンパイルせずにソース・ファイルをプリプロセスすることができます。その場合、**-E** または **-P** オプションのいずれかを指定します。 **-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。

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

受け入れられる入力ファイル・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 コンパイラーとして使用して 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、例えば mysource.c のように **.c** (小文字の c) サフィックスを付けなければなりません。

C++ コンパイラーを使用するには、ソース・ファイルに、**.C** (大文字 C)、**.cc**、**.cp**、**.cpp**、 **.cxx**、または **.c++** サフィックスを付けなければなりません。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。サフィックスが **.a**、**.o**、 **.s**、または **.so** 以外で、このオプションで指定され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、例えば *file_name***.i** のように、**.i** サフィックスが付いています。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 をコンパイラーへ送ります。そこで、そのファイルは **.c** または **.C** ファイルと同じ方法で再びプリプロセスされます。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・ディレクティブのチェックに役立ちます。

オブジェクト・ファイル
オブジェクト・ファイルには、サフィックス **.o** が必要です (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、実行可能なファイルを作成するため、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・ファイルには、サフィックス **.s** が必要です (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。

共用ライブラリー・ファイル
共用ライブラリー・ファイルには、**.so** サフィックスが必要です (例えば、*file_name***.so**)。

出力ファイルの種類

XL C/C++ コンパイラーを呼び出すときに、以下の種類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。実行可能ファイルに別の名前を付けたい場合は、 **-o***file_name* オプションを呼び出しコマンドで使用します。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。指定した名前は、実行可能ファイルの相対、または絶対パス名になります。

オブジェクト・ファイル コンパイラーはオブジェクト・ファイルのサフィックスを **.o** とします (例えば、異なるサフィックスを指定して、またはサフィックスを指定しないで **-o***file_name* オプションを指定しない限り、*file_name***.o** となります)。

-c オプションを指定した場合、出力オブジェクト・ファイル *file_name***.o** が、それぞれの入力ソース・ファイル *file_name***.***x* ごとに作成されます。ここで、*x* は認識される C または C++ ファイル名拡張子です。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配置されます。すべての処理は、コンパイルの完了時に停止します。

コンパイラーを呼び出すことによって、オブジェクト・ファイルを後で単一の実行可能ファイルに、リンク・エディット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・ファイルには、サフィックス **.s** が必要です (例えば、*file_name***.s**)。

アセンブラー・ファイルは、 **-S** オプションを指定して作成します。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、例えば *file_name***.i** のように、**.i** サフィックスが付いています。

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** オプションから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** は、それぞれのソース・ファイルに対して作成され、作成されたソース・ファイルと同じファイル名を持ちます (拡張子は **.i**)。

リスト・ファイル リスト・ファイルは、例えば *file_name*.**lst** のように、.lst サフィックスが付いています。

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** 拡張子) となります。

共用ライブラリー・ファイル 共用ライブラリー・ファイルには、**.so** サフィックスが付いています (例えば my_shrlib.so)。

ターゲット・ファイル **-M** または **-qmakedep** オプションに関連する出力ファイルは、例えば **conversion.d** のように、サフィックス **.d** となります。

ファイルは、**make** コマンド用記述ファイルに組み込むのに適したターゲットを含みます。それぞれの C または C++ 入力ファイルごとに **.d** ファイルが 1 つ作成され、特定の入力ファイルに変更が加えられた場合に、別の入力ファイルの再コンパイルが必要かどうかを判別するために、**make** コマンドによって使用されます。 **.d** ファイルは、その他のファイルには作成されません (**-+** オプションを使用して、その他のファイル・サフィックスを **.C** ファイルとして取り扱わない限り)。

関連概念

6 ページの『入力ファイルの種類』

関連参照

71 ページの『c』
99 ページの『E』
185 ページの『M』
188 ページの『makedep』
201 ページの『o』
203 ページの『P』
215 ページの『print』
230 ページの『S』

関連参照

71 ページの『c』
99 ページの『E』
185 ページの『M』
188 ページの『makedep』
201 ページの『o』
203 ページの『P』
215 ページの『print』
230 ページの『S』

コンパイラー・メッセージおよびリスト情報

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ム・エラーを検出すると、診断メッセージを標準エラー装置に発行し、該当するオプションが選択されている場合には、診断メッセージをリスト・ファイルに発行します。

コンパイラー・メッセージ

コンパイラーは C または C++ 言語特定のメッセージを出します。

C コンパイラー・オプション **-qsrcmsg** を指定し、エラーが特定のコード行にある場合は、再構成されたソース行、または一部のソース行が、エラー・メッセージとともに stderr ファイルに含まれます。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。

-qsource コンパイラー・オプションを指定する場合、コンパイラーはソース・リストにメッセージを置きます。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場合、現行ディレクトリーには **filename.lst** という名のファイルが見えます。

-qflag オプション、または **-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、 **-qinfo** オプションを使用します。

コンパイラー・リスト

コンパイラーによって作成されるリストは、デバッグに役立ちます。正しいオプションを指定すれば、コンパイルのすべての局面での情報を要求することができます。リストは、以下のセクションの組み合わせから構成されます。

- ソース・ファイル名およびコンパイル日時の他に、コンパイラー名、バージョン、およびリリースをリストする、ヘッダー・セクション。
- 入力ソース・コードに行番号を付けてリストする、ソース・セクション。行にエラーがある場合、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。
- コンパイル中に効力のあるオプションをリストする、オプション・セクション。
- コンパイル単位で使用される変数についての情報を提供する、属性および相互参照リスト・セクション。
- 各メイン・ソース・ファイルおよびインクルード・ファイルのファイル番号およびファイル名を示す、ファイル・テーブル・セクション。
- 診断メッセージを要約し、読み取られたソース行数をリストし、コンパイルが正常終了したかどうかを示す、コンパイル・エピローグ・セクション。
- オブジェクト・コードをリストする、オブジェクト・セクション。

ヘッダー・セクションを除く各セクションには、それを識別するセクション見出しがあります。セクション見出しは、二重かぎ括弧で囲まれています。

関連参照

379 ページの『メッセージ重大度レベルとコンパイラー応答』
381 ページの『コンパイラー・メッセージ・フォーマット』
109 ページの『flag』
134 ページの『info』
240 ページの『source』
244 ページの『srcmsg』
282 ページの『w』

プログラムの並列化

コンパイラーを使用すると、**OpenMP アプリケーション・プログラム・インターフェース**の仕様に準拠するプラグマ・ディレクティブを使用して並列化する C および C++ プログラム・コードのセクションを明示的に識別することができます。

プログラム・コードの並列領域は複数のスレッドによって実行され、複数のプロセッサー上で動作している可能性があります。作成されるスレッドの数は、ランタイム・オプションおよびライブラリー関数への呼び出しによって決まります。作業は、指定されたスケジューリング・アルゴリズムに従って使用可能なスレッドに分散されます。スレッドの作成および使用方法についての詳細は、「OpenMP Specification」のセクション 2.3『Parallel Construct』を参照してください。

XL C/C++ コンパイラーが提供する並列プログラミング・サポートについて詳しくは、本節の下記のトピックを参照してください。

- 『OpenMP ディレクティブ』
- 14 ページの『並列環境での共用変数および Private 変数』

OpenMP 仕様の完全な情報については、以下を参照してください。

- OpenMP Web サイト (www.openmp.org)
- OpenMP Specification (www.openmp.org/specs)

OpenMP ディレクティブ

C C++ OpenMP ディレクティブは、さまざまな型の*並列領域* を定義することによって、共用メモリーの並列性を活用します。並列領域は、プログラム・コードの反復セグメントおよび非反復セグメントの両方を含むことができます。

プラグマは、以下の 4 つの一般的なカテゴリーに分かれています。

1. 1 つ目のカテゴリーのプラグマは、作業がスレッドによって並列で行われる並列領域を定義できるようにするものです。ほとんどの OpenMP ディレクティブは、囲んでいる並列領域に、静的にバインドするか、または動的にバインドするかのいずれかです。
2. 2 つ目のカテゴリーは、作業がどのように並列領域のスレッド全体に分散および共用されるかを定義できるようにするものです。
3. 3 つ目のカテゴリーは、スレッド間の同期を制御できるようにするものです。
4. 4 つ目のカテゴリーは、スレッド全体のデータの可視性のスコープを定義できるようにするものです。

関連概念

14 ページの『並列環境での共用変数および Private 変数』

関連タスク

37 ページの『プラグマを使用した並列処理の制御』

関連参照

355 ページの『並列処理を制御するプラグマ』
387 ページの『並列処理のための OpenMP ランタイム・オプション』
389 ページの『並列処理に使用する組み込み関数』

OpenMP 仕様の完全な情報については、以下を参照してください。

- OpenMP Web サイト (www.openmp.org)
- OpenMP Specification (www.openmp.org/specs)

並列環境での共用変数および Private 変数

変数は、並列環境において共用コンテキストか private コンテキストのいずれかを持つことができます。

- 共用コンテキストでの変数は、関連する並列ループ内で実行されているすべてのスレッドに対して可視になります。
- private コンテキストでの変数は、他のスレッドから見えません。スレッドはそれぞれ独自に変数の private コピーを持っており、スレッドによってそのコピーに行われた変更内容は、他のスレッドには不可視です。

変数のデフォルトのコンテキストは、以下の規則によって決まります。

- **静的**ストレージの存在期間を持つ変数は共用である。
- 動的に割り振られたオブジェクトは共用である。
- 自動ストレージの存在期間を持つ変数は private である。
- ヒープが割り振られたメモリー内の変数は共用である。共用ヒープは 1 つしか存在できません。
- 並列構成の外部で定義されたすべての変数は、並列ループが現れると共用になる。
- ループ反復変数は、各自のループ内では private である。ループの後の反復変数の値は、ループが連続的に実行される場合と同じになる。
- **alloca** 関数によって並列ループ内に割り振られたメモリーは、そのループの反復 1 つ分の期間しか存在せず、各スレッドに対して private である。

以下のコード・セグメントは、これらのデフォルト・ルールの例を示したものです。

```
int E1;                          /* shared static     */

 void main (argvc,...) {          /* argvc is shared   */
   int i;                         /* shared automatic  */

   void *p = malloc(...);         /* memory allocated by malloc   */
                                  /* is accessible by all threads */
                                  /* and cannot be privatized     */

   #pragma omp parallel firstprivate (p)
   {
     int b;                       /* private automatic  */
     static int s;                /* shared static      */

     #pragma omp for
     for (i =0;...) {
       b = 1;                     /* b is still private here !    */
       foo (i);                   /* i is private here because it */
                                  /* is an iteration variable     */
     }

     #pragma omp parallel
     {
       b = 1;                     /* b is shared here because it  */
                                  /* is another parallel region   */
     }
   }
 }
```

```
int E2;                          /*shared static */

 void foo (int x) {               /* x is private for the parallel */
                                  /* region it was called from     */

   int c;                         /* the same */
 ... }
```

コンパイラーは、プログラムのセマンティクスの変更が必要でなければ、いくつかの共用変数を private にすることができます。例えば、それぞれのループ反復が共用変数の固有の値を使用する場合は、その変数を private にすることができます。private になった共用変数は、 **-qinfo=private** オプションによって報告されます。クリティカル・セクションを使用して、このレポートにリストされないすべての共用変数へのアクセスを同期化してください。

OpenMP プリプロセッサー・ディレクティブには、選択したデータ変数の可視性コンテキストを指定できるものがあります。データ・スコープ属性文節の概要を以下にリストします。

データ・スコープ属性文節	**説明**
private	**private** 文節は、リスト内の変数がチーム内のそれぞれのスレッドに対して private であることを宣言します。
firstprivate	**firstprivate** 文節は、private 文節によって提供される機能のスーパーセットを提供します。
lastprivate	**lastprivate** 文節は、private 文節によって提供される機能のスーパーセットを提供します。
shared	**shared** 文節は、チーム内のすべてのスレッド間でリストに表示される変数を共用します。チーム内のすべてのスレッドが、共用変数用の同じストレージ域にアクセスします。

データ・スコープ属性文節	説明
reduction	**reduction** 文節は、指定された演算子を使用して、リスト内に表示されたスカラー変数の縮小を実行します。
default	**default** 文節は、変数のデータ・スコープ属性をユーザーが操作できるようにします。

詳しくは、OpenMP ディレクティブの説明、または OpenMP C および 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。

関連概念

13 ページの『OpenMP ディレクティブ』

関連タスク

37 ページの『プラグマを使用した並列処理の制御』

関連参照

355 ページの『並列処理を制御するプラグマ』
387 ページの『並列処理のための OpenMP ランタイム・オプション』
389 ページの『並列処理に使用する組み込み関数』

OpenMP 仕様の完全な情報については、以下を参照してください。

- OpenMP Web サイト www.openmp.org
- OpenMP Specification www.openmp.org/specs

第 2 部 構成と使用

コンパイル環境のセットアップ

C または C++ プログラムをコンパイルする前に、ご使用のシステムに応じた環境変数と構成ファイルとを設定しておかなければなりません。

vac_configure プログラムは、コンパイラーの初期構成ファイルを作成し、GCC ライブラリーへのリンクをセットアップします。 **vac_configure** の使用について詳しくは、「*インストール・ガイド*」を参照してください。

XL C/C++ コンパイラーが使用する環境変数について詳しくは、本節の以下のトピックを参照してください。

- 『環境変数の設定』
- 20 ページの『他の環境変数の設定』

環境変数の設定

Linux システムの Bourne Again SHell (**bash**) は、 AIX® システムにある Bourne Shell (**bsh**) に似ています。 **bash** インターフェースを使用して、XL C/C++ コンパイラーの要求する環境変数を、コマンド行から、またはコマンド・ファイル・スクリプトによって設定します。

bashにおける環境変数の設定

以下のステートメントは、コマンド行に入力するか、またはコマンド・ファイル・スクリプトに挿入するもので、 Bourne Again SHell で環境変数を設定する方法を示しています。表示されたパスは、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していることを前提としています。

```
LANG=en_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/L/%N
export LANG NLSPATH
```

全ユーザーがアクセスできるような変数を設定するには、ファイル **/etc/profile** にコマンドを追加します。特定ユーザー専用の変数を設定するには、ユーザーのホーム・ディレクトリーのファイル **.profile** にコマンドを追加します。環境変数は、ユーザーがログインするたびに設定されます。

並列処理ランタイム・オプションの設定

XLSMPOPTS 環境変数は、ループ並列化を使用してプログラムのオプションを設定します。例えば、プログラム実行時に 4 つのスレッドを作成させ、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、以下に示すように、 XLSMPOPTS 環境変数を設定します。

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

追加の環境変数は、プログラム並列化のオプションを OpenMP 準拠のディレクティブを使用して設定します。

関連タスク

19 ページの『環境変数の設定』

関連参照

383 ページの『並列処理のためのランタイム・オプション』
387 ページの『並列処理のための OpenMP ランタイム・オプション』

他の環境変数の設定

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。

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/7.0/bin にあります。

MANPATH オプションで、man ページを見つけるためのディレクトリー検索パスを指定します。 MANPATH には、デフォルトの man パスの前に、 /opt/ibmcmp/vacpp/7.0/man/en_US が含まれていなければなりません。

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 メッセージおよびヘルプ・ファイル用の各国語を指定します。

LANG 環境変数は、システムに用意されている任意のロケールに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** を有効な任意のほかの各国語コードに置き換えることができます。

ご使用のシステムの各国語の現行設定値を判別するには、以下の両方の **echo** コマンドを使用してください。

```
echo $LANG
echo $NLSPATH
```

NLSPATH メッセージおよびヘルプ・ファイルのパス名を指定します。

PDFDIR オプションで、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指定します。デフォルト値は設定されず、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入れます。プロファイル指示フィードバックの場合は、この変数を絶対パスに設定することをお勧めします。

TMPDIR オプションで、一時ファイルが作成されるディレクトリーを指定します。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が非常に大きなディスク・スペースを必要とする場合があり、デフォルト・ロケーション /tmp は高水準の最適化には不十分である可能性があります。

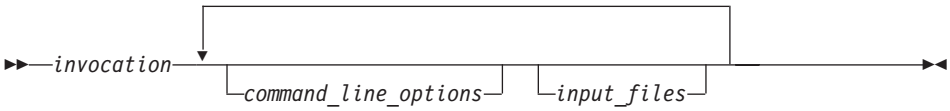
注: LANG および **NLSPATH** 環境変数はオペレーティング・システムのインストール時に初期設定され、使用したいものとは異なっていることがあります。

関連タスク

19 ページの『環境変数の設定』

コンパイラーの起動

IBM® XL C/C++ コンパイラーは、下の構文を使用して起動します。ここで、*invocation* は、有効な XL C/C++ 呼び出しコマンドに置き換えることができます。



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。 C++ コンパイルでは、munch ユーティリティーに渡すための munch オプションもパラメーターに含めることができます。

コンパイラー・オプションは、広範囲の種類の関数を実行します。例えば、コンパイラー特性の設定、オブジェクト・コードおよび作成されるコンパイラー出力の記述、いくつかのプリプロセッサー関数の実行などです。

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** オプションによって、コンパイル完了後にコンパイラーが停止され、**-o** オプションが使用され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・ファイルごとに出力としてオブジェクト・ファイル *file_name*.**o** が作成されます。リンケージ・エディターは起動されません。同じ呼び出しコマンドを使用し、**-c** オプションを付けずにオブジェクト・ファイルを指定することにより、そのオブジェクト・ファイルを後でリンク・エディットすることができます。

注: 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・エディターを*両方とも* 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・オプションの受け入れも行います。

リンケージ・エディターの起動

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。 -E、-P、-c、-S、-qsyntaxonly、または -# のコンパイラー・オプションの 1 つを指定しない限り、呼び出しコマンドの 1 つを指定してコンパイラーを起動すると、自動的にリンケージ・エディターが呼び出されます。

入力ファイル

オブジェクト・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 **.o** が必要です (例えば **year.o**)。静的ライブラリー・ファイル名には、サフィックス **.a** が付きます (例えば **libold.a**)。動的ライブラリー・ファイル名には、サフィックス **.so** が付きます (例えば **libold.so**)。

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。実行可能ファイルに明示的に名前を付けるには、コンパイラー呼び出しコマンドで **-o** *file_name* オプションを使用します。 *file_name* は、実行可能ファイルに指定する名前です。例えば、**myfile.c** をコンパイルし、**myfile** という名の実行可能ファイルを生成するには、次を入力します。

```
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** ファイル名拡張子を持ちます。

ld コマンドによって、明示的にリンケージ・エディターを起動することができます。コンパイラー呼び出しコマンドは、リンケージ・エディター・オプションをいくつか設定して、標準ファイルを実行可能出力にリンクします (デフォルト)。多くの場合、コンパイラー呼び出しコマンドの 1 つを使用して、オブジェクト・ファイルをリンク・エディットするようにしてください。

注: オブジェクト・ファイルのリンク・エディット時には、 **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。

関連概念

3 ページの『コンパイラー・モード』

関連タスク

25 ページの『コンパイラー・オプションの指定』
23 ページの『コンパイラーの起動』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
379 ページの『メッセージ重大度レベルとコンパイラー応答』
413 ページの『付録 C. XL C/C++ 内のライブラリー』

コンパイラー・オプションの指定

以下の方法のいずれかを使用してコンパイラー・オプションを指定することができます。

- コマンド行上に指定する (25 ページを参照してください)
- ソース・プログラム内に指定する (27 ページを参照してください)
- 構成 (.cfg) ファイル内に指定する (28 ページを参照してください)

上記の方法で明示的に設定されていないコンパイラー・オプションのほとんどについて、コンパイラーはデフォルト設定されているものと見なします。

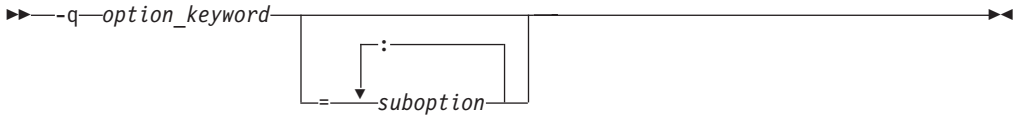
コマンド行におけるコンパイラー・オプションの指定

コマンド行で指定された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、構成ファイルのオプション・セットをオーバーライドします。同様に、コマンド行で指定されたほとんどのオプションは、プラグマ・ディレクティブによって逆にオーバーライドされ、ソース・ファイル中にコンパイラー・オプションを設定する手段を提供します。この方式に従わないオプションについては、『*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* 書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** オプションの場合、特定のオプションが複数回指定されると、そのオプションの中で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** により、コンパイラー・リストを作成するソース・オプションをオンにします。次に、**-qnosource** により、ソース・オプションをオフにすると、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。これは、(**-qsource**) と指定された最後の **source** オプションが優先されるからです。

同一コマンド行で、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けなければなりません。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。ファイル名の前または後に、任意の **-q***option_keyword* を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・オプションを省略することもできます。例えば、**-qopt** を指定するのは、コマンド行で **-qoptimize** を指定するのと同等です。

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

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

報告されるメッセージの重大度レベルを指定するために **-qflag** オプションを使用して、 C ソース・ファイルの file.c をコンパイルします。 **-qflag** サブオプション `w` (警告) はリストに関して報告される最低限の重大度レベルを設定し、サブオプション `e` (エラー) は端末に関して報告される最低限の重大度レベルを設定します。 *full* サブオプションを指定した **-qflag** オプション **-qattr** は、プログラム内のすべての ID の属性リストを作成します。

フラグ・オプション

Linux システムで使用できるコンパイラーは、多くの共通標準フラグ・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、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) に対す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** に対するこれらのフラグの受け渡しを行います。

フラグ・オプションの中には、フラグの一部を形成する引き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き数を指定し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
```

そして、C ソース・ファイル file.c を最適化を指定して (**-O**) コンパイルし、コンパイラーの進行について報告します (**-v**) が、リンケージ・エディターは呼び出しません (**-c**)。

引き数を指定するフラグ・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。しかし、引き数を指定する 1 フラグしか使用することができず、さらにこのフラグは指定される最後のオプションでなければなりません。例えば、(実行可能ファイルの名前を指定するため) ほかのフラグと一緒に **-o** フラグを使用することができるのは、 **-o** オプションとその引き数が最後に指定されている場合のみです。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・オプションは一文字ですが、中には二文字のものもあります。**-pg** (拡張プロファイル) を指定するのは **-p -g** (プロファイルの **-p**、およびデバッグ情報生成の **-g**) を指定するのと同じではないことに注意してください。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、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。

関連概念

5 ページの『コンパイラー・オプション』

関連タスク

23 ページの『コンパイラーの起動』
『プログラム・ソース・ファイル内のコンパイラー・オプションの指定』
28 ページの『構成ファイル内のコンパイラー・オプションの指定』
30 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) コンパイル用コンパイラー・オプションの指定』
32 ページの『矛盾するコンパイラー・オプションの解決』

関連参照

41 ページの『コンパイラーのコマンド行オプション』

プログラム・ソース・ファイル内のコンパイラー・オプションの指定

#pragma ディレクティブを使用すると、プログラム・ソース内でコンパイラー・オプションを指定できます。

プラグマは、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。プラグマには、以下の一般的な書式があり、*character_sequence* は、特定のコンパイラーの命令および引き数 (ある場合) を提供する一連の文字です。

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。複数のプラグマの構成は、1 つの #pragma ディレクティブで指定できます。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・ファイルでプラグマ・ディレクティブにより指定されたオプションは、ほかのプラグマ・ディレクティブを除く、ほかのすべてのオプション設定をオーバーライドします。 2 回以上同じプラグマ・ディレクティブを指定する効果はさまざまです。特定の情報についてはそれぞれのプラグマに関する説明を参照してください。

プラグマ設定は組み込まれたファイルに持ち越すことができます。プラグマ設定からの望ましくない副次作用の可能性を除くには、プログラム・ソースでプラグマ定義の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考慮する必要があります。一部のプラグマ・オプションは **reset** または **pop** サブオプションを提供し、これを実行するのを助けます。

これらの **#pragma** ディレクティブは、これらのディレクティブが適用されるオプションの詳細記述にリストされます。各種の **#pragma** プリプロセッサー・ディレクティブに関する詳細な説明については、*汎用プラグマ* を参照してください。

関連概念

5 ページの『コンパイラー・オプション』

関連タスク

23 ページの『コンパイラーの起動』
25 ページの『コマンド行におけるコンパイラー・オプションの指定』
『構成ファイル内のコンパイラー・オプションの指定』
30 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) コンパイル用コンパイラー・オプションの指定』
32 ページの『矛盾するコンパイラー・オプションの解決』

関連参照

287 ページの『汎用プラグマ』
355 ページの『並列処理を制御するプラグマ』

構成ファイル内のコンパイラー・オプションの指定

デフォルト構成ファイル (**/etc/opt/ibmcmp/vac/7.0/vac.cfg**) は、コンパイラー用の値およびコンパイラー・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・ファイルであり、特定のコンパイル要件をサポートするためか、または別の 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。

構成ファイルで指定したほとんどのオプションは、オプションのデフォルト設定をオーバーライドします。同様に、構成ファイルで指定されたほとんどのオプションは、逆に、ソース・ファイル内またはコマンド行上で設定されたオプションでオーバーライドされます。この方式に従わないオプションについては、『*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。

構成ファイルの調整

デフォルト構成ファイルのテンプレートのテンプレートは、**/opt/ibmcmp/vac/7.0/etc/vac.base.cfg** にインストールされます。

初めてコンパイラーを使用する前に、 **vac_configure** ユーティリティーを使用して、テンプレート・ファイルに基づいた独自の構成ファイルを作成する必要があります。デフォルトでは、 **vac_configure** ユーティリティーは、新しい構成ファイルを **/opt/ibmcmp/vac/7.0/etc/vac.base.cfg** に作成します。後で、新たに作成した構成ファイルを変更して、特定のコンパイル要件をサポートさせるか、または別の C または C++ コンパイル環境をサポートさせることができます。構成ファイル作成に関して詳しくは、「*インストール・ガイド*」を参照してください。

既存のスタンザを変更するか、または新しいスタンザを構成ファイルに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版の **xlC** スタンザに **-qnoro** を追加します。

さまざまな複数の名前に、コンパイラー呼び出しコマンドをリンクさせることができます。コンパイラーを起動するときに指定する名前により、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ほかのスタンザを追加して、ユーザー固有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で **-F** オプションを使用して、追加のスタンザを選択するか、または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。例を以下に示します。

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

これにより、ユーザーの作成した myconfig.cfg 構成ファイルの SPECIAL スタンザを使用して、 myfile.c をコンパイルします。

構成ファイルの属性

構成ファイルには、いくつかのスタンザが含まれます。以下は、構成ファイル内のスタンザによって定義される項目の一部です。

as アセンブラーで使用されるパス名です。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/7.0/exe/xlcentry** です。

code コンパイラーのコード生成フェーズで使用されるパス名です。デフォルトは **/opt/ibmcmp/vac/7.0/bin/xlCcode** です。

codeopt コンパイラーのコード生成フェーズ用のオプション・リストです。

crt リンケージ・エディターへの先頭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場合、**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/7.0/exe/dis** です。

gcrt リンケージ・エディターへの先頭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。デフォルトは **/usr/bin/ld** です。

ldopt コンパイラーのリンケージ・エディター部分に送られるオプション・リストです。これらは、コンパイラーによる標準処理のすべてをオーバーライドして、リンケージ・エディターに送られます。対応するフラグにパラメーターを指定する場合、ストリングは、1 つの文字の後にコロン **(:)** が続く形の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 コンマで区切られたライブラリー・オプションです。コンパイラーは、最後のパラメーターとしてこのオプションをリンケージ・エディターに渡します。**libraries2** は、リンケージ・エディターがプロファイルおよび非プロファイルの両方に応じてリンク・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・エディターへの先頭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 コンマで区切られたオプション・フラグのストリングです。このストリングは、コマンド行で入力されたかのように、コンパイラーにより処理されます。

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、名前付きのスタンザまたはローカル・スタンザから採用されます。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指定されていないと、 **use** スタンザの値が適用されます。コマンド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。デフォルトは **/opt/ibmcmp/vac/7.0/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/7.0/bin/xlC** です。

関連概念

5 ページの『コンパイラー・オプション』

関連タスク

23 ページの『コンパイラーの起動』
25 ページの『コマンド行におけるコンパイラー・オプションの指定』
27 ページの『プログラム・ソース・ファイル内のコンパイラー・オプションの指定』
『アーキテクチャー固有の (32 ビットまたは 64 ビットの) コンパイル用コンパイラー・オプションの指定』
32 ページの『矛盾するコンパイラー・オプションの解決』

関連参照

41 ページの『コンパイラーのコマンド行オプション』

「*Getting Started with XL C/C++ Advanced Edition for Linux*」の『*コンパイラーの構成*』セクションも参照してください。

アーキテクチャー固有の (32 ビットまたは 64 ビットの) コンパイル用コンパイラー・オプションの指定

IBM 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にコンパイラー出力を最適化することができます。32 ビットまたは 64 ビットのいずれかのモードでコンパイルするようにコンパイラーに指示することもできます。

コンパイラーは、コンパイラー・モードを判別する最後に検出された有効なコンパイラー・オプションによって、以下の順でコンパイラー・オプションを評価します。

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・オプションの設定値の組み合わせに応じて、以下の条件に従って決定されます。

- コンパイラー・モード は、 **-q32** または **-q64** 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ます。
- アーキテクチャー・ターゲット は、指定された **-qarch** の設定値がコンパイラー・モード の設定値と互換性がある場合は、 **-qarch** コンパイラー・オプションの最後に検出されたインスタンスに応じて設定されます。 **-qarch** オプションを設定しない場合は、コンパイラーは **-qarch** の設定値が **ppc64grsq** であると見なします。
- アーキテクチャー・ターゲットの調整は、 **-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 の設定値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ます。 **-qtune** オプションを設定しない場合は、コンパイラーは、使用中の **-qarch** の設定値に応じてデフォルトの **-qtune** の設定値であると見なします。 **-qarch** を指定しない場合は、コンパイラーは **-qtune** 設定値が **pwr4** であると見なします。

以下では、可能性のあるオプションの矛盾およびこ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** の設定値が、ユーザーが選択した **-qarch** オプションと互換性がない。

  解決: -q32 または **-q64** の設定値が **-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出し、**-qarch** をそのデフォルト設定値に設定して、**-qtune** オプションをそのデフォルト値に応じて設定します。
- **-q32** または **-q64** の設定値が、ユーザーが選択した **-qtune** オプションと互換性がない。

  解決: -q32 または **-q64** の設定値が **-qtune** 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出し、**-qtune** オプションを **-qarch** の設定値に対するデフォルトの **-qtune** 値に設定します。
- **-qarch** オプションが、ユーザーが選択した **-qtune** オプションと互換性がない。

  解決: コンパイラーは警告メッセージを出し、**-qtune** を **-qarch** の設定値に対するデフォルトの **-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出し、 **-qarch** および **-qtune** をデフォルトの設定値に設定します。コンパイラー・モード (32 ビットまたは 64 ビット) は、**-q32** または **-q64** コンパイラー設定によって判別されます。

関連概念

5 ページの『コンパイラー・オプション』

関連タスク

23 ページの『コンパイラーの起動』
25 ページの『コマンド行におけるコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
『矛盾するコンパイラー・オプションの解決』

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 (**xref** と **attr** 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値を使用します。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、対立するオプションまたは矛盾する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るコンパイラー・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。対立する、または互換性のないコンパイラー・オプションがコマンド行で指定されている場合、コマンド行で後のほうに指定されているオプションが優先されます。
3. 構成ファイルに指定されるコンパイラー・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。

すべてのオプション競合が上記の規則で解決するわけではありません。下のテーブルでは、例外およびコンパイラーがそれらの間の競合を処理する方法を要約しています。

オプション	矛盾するオプション	解決
-qhalt	**-qhalt** によって指定される複数の重大度	指定された最低の重大度
-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**

オプション	矛盾するオプション	解決
-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** は **-B** および **-t** をオーバーライドする
-S	**-c**	**-S**

関連概念

5 ページの『コンパイラー・オプション』

関連タスク

23 ページの『コンパイラーの起動』
25 ページの『コマンド行におけるコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』

インクルード・ファイル用のパス名の指定

#include プリプロセッサー・ディレクティブを使用して、あるソース・ファイルを別のファイルに組み込むときには、組み込まれるファイル名を指定しなければなりません。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 とも呼ばれる*絶対パス名* 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、ユーザーが現在入って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) にかかわらず、指定されたファイルを探し出します。

  以下の例では、 John Doe のサブディレクトリー example_prog にあるファイル *mine.h* を指す絶対パスを指定します。

  ```
  /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インクルード・ファイルのディレクトリー検索順序

C および C++ 言語では、 **#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ます。 IBM XL C/C++は、これらを両方ともサポートします。 **#include** ディレクティブで、インクルード・ファイル名は区切り文字 **< >** または **“ ”** で囲まれています。

選択した区切り文字によって、特定のインクルード・ファイル名を探し出すのに使用される検索パスが決定されます。コンパイラーは、インクルード・ファイルが見つかるまで、以下のように検索パスの全ディレクトリーでそのインクルード・ファイルを検索します。

#include の型	**ディレクトリー検索順序**
#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、コマンド行に表示される順に *file_name* を検索します。 2. C++ コンパイルの場合、コンパイラーは、**-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. コンパイラーはまず、現行のソース・ファイルが常駐しているディレクトリーでインクルード・ファイルを検索します。現行のソース・ファイルは、ディレクティブ **#include "***file_name***"** を含むファイルです。 2. コンパイラーはそれから、**#include <***file_name***>** の上記の検索順序に従ってインクルード・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。一般的に、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み、ユーザーの作成するヘッダー・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みます。
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。

   ほかのディレクトリーを検索する前に、 **-I***directory* オプションで指定したディレクトリーを検索するには、 **#include "***file_name***"** ディレクティブと共に **-qidirfirst** オプションを使用します。

   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。

関連参照

129 ページの『I』
72 ページの『c_stdinc』
87 ページの『cpp_stdinc』
121 ページの『gcc_c_stdinc』
122 ページの『gcc_cpp_stdinc』
130 ページの『idirfirst』
248 ページの『stdinc』

プラグマを使用した並列処理の制御

並列処理操作は、ユーザーのプログラム・ソースでプラグマ・ディレクティブによって制御します。プラグマは、**-qsmp** コンパイラー・オプションで並列化が使用可能になっているときにのみ有効です。

OpenMP ディレクティブ

C C++

構文

```
#pragma omp pragma_name_and_args
statement_block
```

プラグマ・ディレクティブは、一般に、適用対象のコードのセクションの直前に記述します。 **omp parallel** ディレクティブは、並列化されるプログラム・コードの領域を定義する場合に使用します。その他の OpenMP ディレクティブは、定義済み並列領域内のデータ変数の可視性、およびその領域内の作業の共用方法と同期方法を定義します。

例えば、以下の例は、 **for** ループの反復が並列で実行できる並列領域を定義しています。

```
#pragma omp parallel
{
  #pragma omp for
    for (i=0; i<n; i++)
      ...
}
```

この例は、プログラム・コードの複数の非反復セクションが並列で実行できる並列領域を定義しています。

```
#pragma omp sections
{
  #pragma omp section
     structured_block_1
           ...
  #pragma omp section
     structured_block_2
           ...
      ....
}
```

関連概念

13 ページの『プログラムの並列化』

関連参照

355 ページの『並列処理を制御するプラグマ』
387 ページの『並列処理のための OpenMP ランタイム・オプション』
389 ページの『並列処理に使用する組み込み関数』

OpenMP 仕様の完全な情報については、以下を参照してください。

- OpenMP Web サイト
- OpenMP 仕様

第 3 部 解説

コンパイラー・オプション

本節では、XL C/C++ で使用できるコンパイラー・オプションについて説明します。オプションは、本節の以下のトピックに記載しているように、3 つの汎用グループに分類されます。

- 『コンパイラーのコマンド行オプション』
- 287 ページの『汎用プラグマ』
- 355 ページの『並列処理を制御するプラグマ』

コンパイラーのコマンド行オプション

本節では、XL C/C++ コマンド行オプションをリストして説明します。

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。

- 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 **#pragma** 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。
- オプションの目的およびその動きに関する追加情報。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。

関連概念

5 ページの『コンパイラー・オプション』

関連タスク

25 ページの『コマンド行におけるコンパイラー・オプションの指定』
27 ページの『プログラム・ソース・ファイル内のコンパイラー・オプションの指定』
28 ページの『構成ファイル内のコンパイラー・オプションの指定』
30 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) コンパイル用コンパイラー・オプションの指定』
32 ページの『矛盾するコンパイラー・オプションの解決』

関連参照

287 ページの『汎用プラグマ』
355 ページの『並列処理を制御するプラグマ』

コマンド行コンパイラー・オプションの要約

オプション名	型	デフォルト	説明
+ (正符号)	*-flag*	-	C++ 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**、および **.s** 以外の任意のサフィックスです。
# (ポンド記号)	*-flag*	-	何も行わずにコンパイルをトレースする。
32、64	**-q***opt*	**32**	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。
abi_version	**-q***opt*	**abi_version** を参照。	C++ 別のレベルの GNU C++ とのバイナリー互換性のための C++ ABI バージョンを指定する。
aggrcopy	**-q***opt*	**aggrcopy** を参照。	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。
alias	**-q***opt*	**alias** を参照。	最適化中に使用する型ベースの別名割り当てを指定する。
align	**-q***opt*	**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。
alloca	**-q***opt*	-	C **#pragma alloca** ディレクティブがソース・コードであるかのように、関数 **alloca** の呼び出しにインライン・コードを使用する。
altivec	**-q***opt*	**noaltivec**	AltiVec™ 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。
arch	**-q***opt*	**arch=ppcv**	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。
asm	**-q***opt*	**-qasm=gcc**	**asm** ステートメントの解釈および以降のコード生成を制御する。
attr	**-q***opt*	**noattr**	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。
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。
bigdata	**-q***opt*	**nobigdata**	32 ビット・モードで、初期化データを 16 MB より大きいサイズにすることができる。
bitfields	**-q***opt*	**bitfields=signed**	ビット・フィールドを符号付きとするかどうかを指定する。
C	*-flag*	-	プリプロセスされた出力にコメントを保持する。
c	*-flag*	-	コンパイラーに、コンパイラーのみにソース・ファイルを渡すように指示する。
c_stdinc	**-q***opt*	-	C C ヘッダーの標準検索ロケーションを変更する。
cache	**-q***opt*	-	特定の実行マシンのキャッシュ構成を指定する。

オプション名	型	デフォルト	説明
chars	**-q***opt*	**chars=unsigned**	コンパイラーに、 **char** 型の変数をすべて **signed** または **unsigned** のいずれかとして処理するように指示する。
check	**-q***opt*	**nocheck**	特定のタイプの実行時検査を行うコードを生成する。
cinc	**-q***opt*	**nocinc**	C++ コンパイラーに **extern "C" { }** ラッパーをインクルード・ファイルのコンテンツの周りに配置するように指示する。
compact	**-q***opt*	**nocompact**	最適化とともに使用すると、可能な場合に、実行速度を犠牲にしてコード・サイズが削減される。
complexgccincl	**-q***opt*	**complexgccincl =/usr/include**	指定されたディレクトリーにあるインクルード・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指示する。
cpluscmt	**-q***opt*	**cpluscmt** を参照。	C C++ のコメントをソース・ファイルで認識させたい場合は、このオプションを使用する。
cpp_stdinc	**-q***opt*	-	C++ C++ ヘッダーの標準検索ロケーションを指定する。
crt	**-q***opt*	**crt**	リンク時に標準システム始動ファイルを使用するようにリンカーに指示する。
D	*-flag*	-	**#define** プリプロセッサー・ディレクティブにあるのと同様に ID *name* を定義する。
dataimported	**-q***opt*	-	インポートされるものとしてデータにマークを付ける。
datalocal	**-q***opt*	-	ローカルとしてデータにマークを付ける。
mbcs、dbcs	**-q***opt*	**nodbcs**	プログラムにマルチバイト文字が含まれる場合には、**-qdbcs** オプションを使用する。
dbxextra	**-q***opt*	**nodbxextra**	C すべての **typedef** 宣言、 **struct** 型定義、**union** 型定義、および **enum** 型定義をデバッガー処理用に組み込むことを指定する。
digraph	**-q***opt*	**digraph** を参照。	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。
directstorage	**-q***opt*	**nodirectstorage**	ライトスルー対応ストレージまたはキャッシュ禁止ストレージが参照可能であることをコンパイラーに通知する。
dollar	**-q***opt*	**nodollar**	$ シンボルを ID の名前で使用できるようにする。
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指示する。
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。 **ld** オプションの追加情報については、システムの資料を参照してください。

オプション名	型	デフォルト	説明
eh	**-q***opt*	**eh**	C++ 例外処理を制御する。
enablevmx	**-q***opt*	**noenablevmx**	VMX (Vector Multimedia Extension) 命令の生成を使用可能にする。
enum	**-q***opt*	**enum** を参照。	列挙の占めるストレージの量を指定する。
F	*-flag*	-	コンパイラーの代替構成ファイルの名前を指定する。
flag	**-q***opt*	**flag=i:i**	報告させる診断メッセージの最低の重大度レベルを指定する。
float	**-q***opt*	**float** を参照。	浮動小数点演算を高速化するか正確度を向上させるために、各種浮動小数点オプションを指定する。
flttrap	**-q***opt*	**noflttrap**	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。
fullpath	**-q***opt*	**nofullpath**	**-g** 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。
funcsect	**-q***opt*	**nofuncsect**	別々のオブジェクト・ファイル、制御セクションまたは csect のそれぞれの関数ごとに命令を配置する。
g	*-flag*	-	デバッガーが使用するデバッグ情報を生成する。
gcc_c_stdinc	**-q***opt*	-	C gcc ヘッダーの標準検索ロケーションを変更する。
gcc_cpp_stdinc	**-q***opt*	-	C++ g++ ヘッダーの標準検索ロケーションを変更する。
genproto	**-q***opt*	**nogenproto**	C K&R 関数定義から ANSI プロトタイプを生成する。
halt	**-q***opt*	C **halt=s** C++ **halt=e**	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指示する。
haltonmsg	**-q***opt*	-	C++ 特定のエラー・メッセージが検出されたときは、コンパイル・フェーズ後に停止するようコンパイラーに指示する。
hot	**-q***opt*	**nohot**	最適化中に、高位ループ分析および変換の実行をコンパイラーに指示する。
I	*-flag*	-	絶対パスを指定しない **#include** ファイル名に追加の検索パスを指定する。
idirfirst	**-q***opt*	**noidirfirst**	**#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。
ignerrno	**-q***opt*	**noignerrno**	コンパイラーが、システム呼び出しによって **errno** が変更されないと想定して最適化を行うことを許可する。
ignprag	**-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指示する。
info	**-q***opt*	C **noinfo** C++ **info=lan**	通知メッセージを作成する。

オプション名	型	デフォルト	説明
initauto	**-q***opt*	**noinitauto**	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。
inlglue	**-q***opt*	**noinlglue**	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を行うため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って、高速な外部結合を生成する。
inline	**-q***opt*	**inline** を参照。	関数の呼び出しを生成する代わりに、関数のインラインを試みる。
ipa	**-q***opt*	**ipa** を参照。	プロシージャー間分析 (IPA) と呼ぶクラスの最適化をオンにしたりカスタマイズしたりする。
isolated_call	**-q***opt*	-	ソース・ファイル内の副次作用がない関数を指定する。
keepparm	-q*opt*	**nokeepparm**	C++ アプリケーションが最適化される場合でも関数仮パラメーターがスタックに保管されているか確認する。
keyword	-q*opt*	**keyword** を参照。	指定されたストリングが、キーワードとして処理されるのか、ID として処理されるのかを制御する。
L	*-flag*	**L** を参照。	リンク時に、**-l** オプションによって指定したライブラリー・ファイルについて、指定したディレクトリーを検索する。
l	*-flag*	**l** を参照。	リンクのために指定したライブラリーを検索する。
langlvl	**-q***opt*	**langlvl** を参照。	コンパイル用の C または C++ の言語レベルを選択する。
lib	**-q***opt*	**lib**	リンク時に標準システム・ライブラリーを使用するようにコンパイラーに指示する。
libansi	**-q***opt*	**nolibansi**	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。
linedebug	**-q***opt*	**nolinedebug**	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。
list	**-q***opt*	**nolist**	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。
listopt	**-q***opt*	**nolistopt**	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。
longlit	**-q***opt*	**nolonglit**	サフィックスをはずしたリテラルを 64 ビット・モードの long 型にする。
longlong	**-q***opt*	**longlong** を参照。	プログラムで **long long** 型を許可する。
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。
makedep	**-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。

オプション名	型	デフォルト	説明
maxerr	**-q***opt*	**nomaxerr**	指定した重大度以上のエラーの件数が指定した数に達した場合に、コンパイルを停止するようにコンパイラーに指示する。
maxmem	**-q***opt*	**maxmem=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。
mbcs、dbcs	**-q***opt*	**nombcs**	プログラムにマルチバイト文字が含まれる場合には、 **-qmbcs** オプションを使用する。
mkshrobj	**-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。
minimaltoc	**-q***opt*	**nominimaltoc**	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。
O、optimize	**-q***opt, -flag*	**nooptimize**	コンパイル中に、選択したレベルでコードを最適化する。
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時に名前を指定した C または C++ のソース・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・ファイルを入力ソース・ファイルごとに作成する。
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。
path	**-q***opt*	-	代替プログラム名およびパス名を構成する。
pdf1、pdf2	**-q***opt*	**nopdf1**、**nopdf2**	プロファイル指示のフィードバックを介して最適化を調整する。
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。
phsinfo	**-q***opt*	**nophsinfo**	各コンパイル・フェーズでかかった時間を報告する。
pic	**-q***opt*	**nopic**	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指示する。
prefetch	**-q***opt*	**prefetch**	コンパイルされたコード内でのプリフェッチ指示の生成を使用可能にする。
print	**-q***opt*	**print**	**-qnoprint** はリストを抑制する。
priority	**-q***opt*	-	C++ 静的オブジェクトを初期化する場合の優先順位を指定する。
proclocal、 procimported、 procunknown	**-q***opt*	**proclocal** を参照。	関数をローカル、インポートされるもの、または不明としてマークする。
proto	**-q***opt*	**noproto**	C すべての関数がプロトタイプ宣言されていると見なす。
Q	*-flag*	**Q** を参照。	関数のインラインを試行する。

オプション名	型	デフォルト	説明
R	*-flag*	**R** を参照。	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。
r	*-flag*	-	再配置可能オブジェクトを作成する。
report	**-q***opt*	**noreport**	プログラム・ループの並列化と最適化の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指示する。
ro	**-q***opt*	**ro** を参照。	ストリング・リテラルの保管型を指定する。
roconst	**-q***opt*	**roconst** を参照。	定数値の保管場所を指定する。
rtti	**-q***opt*	**rtti**	C++ typeid 演算子および dynamic_cast 演算子のための実行時識別 (RTTI) 情報を生成する。
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。
saveopt	**-q***opt*	**nosaveopt**	コマンド行コンパイラー・オプションをオブジェクト・ファイル内に保管する。
showinc	**-q***opt*	**noshowinc**	**-qsource** と共に使用して、プログラム・ソース・リストにユーザー・ヘッダー・ファイル (″ ″ を使用して組み込む) またはシステム・ヘッダー・ファイル (< > を使用して組み込む) を選択的に表示する。
showpdf	**-q***opt*	**noshowpdf**	**-qpdf1** と共に使用して、追加呼び出しおよびブロック数プロファイル情報を実行可能ファイルに追加する。
smallstack	**-q***opt*	**nosmallstack**	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指示する。
smp	**-q***opt*	**nosmp**	プログラム・コードの並列化を使用可能にする。
source	**-q***opt*	**nosource**	コンパイラー・リストを作成し、ソース・コードを組み込む。
sourcetype	**-q***opt*	**sourcetype=default**	実際のソース・ファイル名サフィックスとは関係なく、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・タイプであるかのように処理することをコンパイラーに指示する。
spill	**-q***opt*	**spill=512**	レジスター割り振り予備域のサイズを指定する。
srcmsg	**-q***opt*	**nosrcmsg**	C **stderr** ファイル内の診断メッセージに、対応するソース・コード行を追加する。
staticinline	**-q***opt*	**nostaticinline**	インライン関数を静的であるとして扱う。
staticlink	**-q***opt*	**nostaticlink**	共用ライブラリーへのリンクを制御する。
statsym	**-q***opt*	**nostatsym**	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を名前リストに追加する。

オプション名	型	デフォルト	説明
stdinc	**-q***opt*	**stdinc**	**#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。
strict	**-q***opt*	**strict** を参照。	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。
strict_induction	**-q***opt*	**strict_induction** を参照。	プログラムのセマンティクスを変更する可能性があるループ帰納変数の最適化を使用不可にする。
suppress	**-q***opt*	**suppress** を参照。	抑制するコンパイラー・メッセージ番号を指定する。
symtab	**-q***opt*	-	シンボル・テーブルに示される情報を決定する。
syntaxonly	**-q***opt*	-	C 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。
t	*-flag*	**t** を参照。	**-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。
tabsize	**-q***opt*	**tabsize=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。
tbtable	**-q***opt*	**tbtable** を参照。	トレースバック・テーブルの特性を設定する。
tempinc	**-q***opt*	**tempinc** を参照。	C++ テンプレート関数およびクラス宣言に対する別個のインクルード・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配置する。
templaterecompile	**-q***opt*	**templaterecompile** を参照。	C++ **-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。
templateregistry	**-q***opt*	**templateregistry**	C++ ソース内で検出されたときにすべてのテンプレートのレコードを保守し、各テンプレートのインスタンスが 1 つだけ生成されるようにする。
tempmax	**-q***opt*	**tempmax=1**	C++ **tempinc** オプションによって生成させるテンプレート・インクルード・ファイルの最大数をヘッダー・ファイルごとに指定する。
threaded	**-q***opt*	**threaded** を参照。	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。
tls	**-q***opt*	**tls** を参照。	ローカルとしてデータにマークを付ける。
tmplparse	**-q***opt*	**tmplparse=no**	C++ 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のインプリメンテーションに適用するかどうかを指定する。
tocdata	**-q***opt*	**notocdata**。	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。
trigraph	**-q***opt*	**trigraph** を参照。	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。
tune	**-q***opt*	**tune** を参照。	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。

オプション名	型	デフォルト	説明
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、指定したものを未定義にする。
unroll	**-q***opt*	**unroll=auto**	プログラムの内側のループをアンロールする。
unwind	**-q***opt*	**unwind**	アプリケーションがいずれのプログラム・スタック・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。
upconv	**-q***opt*	**noupconv**	C 整数拡張を行うときに **unsigned** の指定を保持する。
utf	**-q***opt*	**noutf**	UTF リテラル構文の認識を使用可能にする。
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指示する。
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指示する。
vftable	**-q***opt*	**vftable**	C++ 仮想関数テーブルの生成を制御する。
vrsave	**-q***opt*	**vrsave**	VRSAVE レジスターの保守に必要な関数 prolog および epilog コードを制御する。
W	*-flag*	-	リストしたワードを、指定したコンパイラー・プログラムに渡す。
w	*-flag*	-	警告メッセージの抑止を要求する。
warn64	**-q***opt*	**nowarn64**	32 ビットおよび 64 ビット・コンパイラー・モード間で起こりうるデータ変換問題の検査を使用可能にする。
xcall	**-q***opt*	**noxcall**	コンパイル単位内の静的ルーチンを扱うコードを、外部呼び出しであるかのように生成する。
xref	**-q***opt*	**noxref**	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。
y	*-flag*	-	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。

+ (正符号)

C++

目的

任意のファイル *filename.nnn* を C++ 言語としてコンパイルする。ここで、*nnn* は、**.a**、 **.o**、または **.s** 以外の任意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は、 **.C** (大文字の C)、**.cc**、 **.cp**、**.cpp**、**.cxx**、または **.c++** がなければ、C++ ファイルとしてコンパイルされません。 **.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** オプションと共に使用してはいけません。

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、以下を入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
241 ページの『sourcetype』

(ポンド記号)

C C++

目的

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。このオプションと一緒に **xlc++** コマンドを出すと、起動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・エディターは起動されません。

構文

注

このコマンドを使用して、特定のコンパイルに関連するコマンドおよび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、コンパイラーは起動しません。**-#** 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 をコンパイルするためのステップをプレビューさせるには、以下を入力します。

```
xlc myprogram.c -#
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
278 ページの『v』

32、64

C C++

目的

32 ビットまたは 64 ビットのコンパイラー・モードのいずれかを選択します。

構文

注

このオプションがコマンド行で明示的に指定されておらず、コンパイラーはデフォルトで 32 ビット出力モードになります。

コンパイラーを 64 ビット・モードで起動した場合は、__64BIT__ プリプロセッサー・マクロが定義されます。

コンパイラーの出力を使用するアーキテクチャーに合わせてその出力を最適化するには、**-q32** および **-q64** オプションを **-qarch** および **-qtune** コンパイラー・オプションとともに使用します。

例

32 ビットの PowerPC® アーキテクチャーのコンピューターで、myprogram.c からコンパイルした実行可能プログラムのテスト実行するように指定するには、次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・ファイルに対して、32 ビットおよび 64 ビットのコンパイル・モードを混在させると、 オブジェクトは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。

関連参照

- 「*XL C/C++ スタートアップ・ガイド*」の『32 ビット・アプリケーションおよび 64 ビット・アプリケーションの開発』のセクションを参照してください。
- 63 ページの『arch』
- 269 ページの『tune』

abi_version

C++

目的

別のレベルの GNU C++ とのバイナリー互換性を維持するための C++ ABI バージョンを指定する。

構文

ここで、

1 GNU C++ 3.2 におけるのと同じ C++ ABI 振る舞いを指定します。

2 このバージョンが Linux オペレーティング・システムでサポートされる場合、 GNU C++ 3.4 におけるのと同じ C++ ABI 振る舞いを指定します。

注

オプション **-qabi_version** は、GNU C++ オプションの -fabi-version=*n* との互換性を保つために提供されており、ユーザーはこれを使用して、コンパイル時に使用する 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・マシンそのものと、XL C++ のインストール時に構成された GNU C++ のレベルに応じて異なります。コンパイル・マシンに GNU C++ 3.2 または 3.3 が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することによって確認することができます。

関連参照

- 181 ページの『listopt』

aggrcopy

C C++

目的

構造体および共用体の破壊コピー操作を使用可能にします。

構文

デフォルト設定

有効な **-qlanglvl=extended** または **-qlanglvl=classic** でコンパイルする場合、このオプションのデフォルト設定は **-qaggrcopy=overlap** です。そうでない場合は、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 規格に適合しないプログラムは、**-qaggrcopy=overlap** コンパイラー・オプションでコンパイルする必要があります。

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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
160 ページの『langlvl』

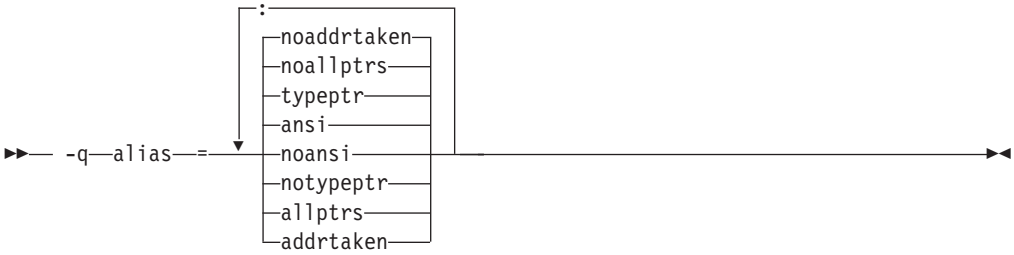
alias

C C++

目的

別名割り当てアサーションをコンパイル単位に適用するようにコンパイラーに指示する。コンパイラーは、他を指定しない限り、別名割り当てアサーションを利用して、可能な位置で最適化の対象を拡大します。

構文



別名割り当ての使用可能なオプションは、以下のとおりです。

[NO]TYPeptr
: **notypeptr** が指定された場合、さまざまな型に対するポインターに別名が割り当てられません。つまり、コンパイル単位では、異なる型の 2 つのポインターが同じ保管場所を指すことはありません。

[NO]ALLPtrs
: **noallptrs** が指定された場合、ポインターに別名が割り当てられません (これは **-qalias=typeptr** も暗黙指定します)。したがって、コンパイル単位では、2 つのポインターが同じ保管場所を指すことはありません。

[NO]ADDRtaken
: **noaddrtaken** が指定された場合、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、すべてポインターを介した間接アクセスでは結合されないと見なされます。

[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、ポインターが同じ型のオブジェクトを指すことしか できないと想定します。これ (**ansi**) は、**xlc xlC**、および **c89 コンパイラー**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、所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任意の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** コンパイラーのデフォルトです。

注

以下は、型ベースの別名割り当ての対象となりません。

- 符号付きまたは符号なしの型。例えば、**signed int** へのポインターは、**unsigned int** を指すことできます。
- 文字ポインター型は、任意の型を指すことができます。

- **volatile** または **const** として修飾された型。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 のコンパイル時に最悪の場合を考慮して別名割り当てを想定するように指定するには、以下を入力します。

```
xlc myprogram.c -O -qalias=noansi
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
299 ページの『#pragma disjoint』

align

C C++

目的

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 GCC 位置合わせ規則を使用して GCC オブジェクトとの互換性を維持します。これはデフォルトです。
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 **-fpack-struct** オプションとほぼ同じです。

290 ページの『#pragma align』および 330 ページの『#pragma options』も参照してください。

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できます。直前の位置合わせ規則に復帰するには、 **#pragma align=reset** を使用します。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、インクルード・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 - 集合体定義にのみ影響を与える

コンパイラー呼び出しを使用して、以下を実行します。

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

ここで、file2.C には以下が含まれます。

```
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 - #pragma の組み込み

コンパイラー呼び出しを使用して、以下を実行します。

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

ここで、file.c には以下が含まれます。

```
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                          */
```

__align 指定子の使用

__align 指定子を使用すると、データ項目を宣言または定義するときの位置合わせを明示的に指定することができます。

__align 指定子:

目的: **__align** 指定子を使用して、データ項目を宣言または定義する場合の位置合わせおよび埋め込みを明示的に指定する。

構文:

```
declarator __align (int_const) identifier;

__align (int_const) struct_or_union_specifier [identifier] {struct_decln_list}
```

ここで、

int_const は、バイト整合の境界を指定します。 *int_const* は、0 より大きく 2 の累乗と等しい整数定数でなければなりません。

注: **__align** 指定子は、第 1 レベルの変数および集合体定義の宣言でのみ使用することができます。これは、パラメーターおよび自動を無視します。

__align 指定子は、別の集合体定義の範囲内でネストされた集合体定義に使用することができます。

__align 指定子は、以下の状態で使用することはできません。

- 集合体定義内の個々のエレメント。

- 不完全型で宣言された変数。
- 定義なしで宣言された集合体。
- 1 つの配列の個々のエレメント。
- **typedef**、**function**、および **enum** などの、他の型の宣言または定義。
- 変数の位置合わせのサイズが、型の位置合わせのサイズよりも小さい場合。

すべての位置合わせがオブジェクト・ファイル内で表示可能であるとは限りません。

例: 以下は、**__align** を第 1 レベルの変数に適用しています。

```
int __align(1024) varA;        /* varA is aligned on a 1024-byte boundary
                                  and padded with 1020 bytes              */

static int __align(512) varB;  /* varB is aligned on a 512-byte boundary
                                  and padded with 508 bytes               */

int __align(128) functionB( ); /* An error                                */

typedef int __align(128) T;    /* An error                                */

__align enum C {a, b, c};      /* An error                                */
```

以下は、集合体メンバーに影響せずに **__align** を位置合わせおよび埋め込み集合体の各タグに適用しています。

```
__align(1024) struct structA {int i; int j;}; /* struct structA is aligned
                                                 on a 1024-byte boundary
                                                 with size including padding
                                                 of 1024 bytes            */

__align(1024) union unionA {int i; int j;}; /* union unionA is aligned
                                               on a 1024-byte boundary
                                               with size including padding
                                               of 1024 bytes            */
```

以下は、構造体または共用体を使用している集合体のサイズおよび位置合わせが影響を受けているところで、 **__align** を構造体または共用体に適用しています。

```
__align(128) struct S {int i;};     /* sizeof(struct S) == 128             */

struct S sarray[10];                /* sarray is aligned on 128-byte boundary
                                       with sizeof(sarray) == 1280         */

struct S __align(64) svar;          /* error - alignment of variable is
                                         smaller than alignment of type    */

struct S2 {struct S s1; int a;} s2; /* s2 is aligned on 128-byte boundary
                                         with sizeof(s2) == 256            */
```

以下は、**__align** を配列に適用しています。

```
AnyType __align(64) arrayA[10]; /* Only arrayA is aligned on a 64-byte
                                   boundary, and elements within that array
                                   are aligned according to the alignment
                                   of AnyType.  Padding is applied after the
                                   back of the array and does not affect
                                   the size of the array member itself.    */
```

以下は、変数の位置合わせのサイズが型の位置合わせのサイズと異なるところに **__align** を適用しています。

```
__align(64) struct S {int i;};

struct S __align(32) s1;          /* error, alignment of variable is smaller
                                     than alignment of type                */

struct S __align(128) s2;         /* s2 is aligned on 128-byte boundary     */

struct S __align(16) s3[10];      /* error                                  */

int __align(1) s4;                /* error                                  */

__align(1) struct S {int i;};     /* error                                  */
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
290 ページの『#pragma align』
337 ページの『#pragma pack』

また、「*XL C/C++ プログラミング・ガイド*」の『集合体内のデータの位置合わせ』も参照してください。

alloca

C

目的

#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 の呼び出しにインライン・コードを使用します。

構文

注

C **#pragma alloca** が未指定の場合で、**-ma** を使用しない場合、**alloca** は組み込み関数としてではなく、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する必要があります。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、コンパイラーの起動時に、コマンド行で次のオプションを使用します。

```
-Dalloca=__alloca
```

alloca の代わりに、C99 可変長配列を使用できます。

例

myprogram.c をコンパイルして、関数 **alloca** の呼び出しをインラインとして扱わせるには、次のように入力します。

```
xlc myprogram.c -qalloca
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
89 ページの『D』
187 ページの『ma』
291 ページの『#pragma alloca』

altivec

C C++

目的

AltiVec 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

AltiVec プログラミング・インターフェース仕様書には、 vector データ型および演算子の特殊なセットが記述されています。このオプションは、AltiVec データ型および演算子をサポートすることをコンパイラーに指示します。

-qaltivec の指定が有効となるのは、 **-qarch** が設定されている場合か、あるいは **ppc970** が暗黙指定されている場合に限られます。 **-qarch** の他の設定値が設定または暗黙指定されている場合、コンパイラーは **-qaltivec** を無視して警告を発します。

また、**-qaltivec** は、**-qnoenablevmx** が有効な場合、使用することができません。

-qaltivec が有効な場合、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

コンパイラーに AltiVec プログラミング・インターフェース仕様をサポートさせるには、次のように入力します。

```
xlc myprogram.c -qaltivec
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
63 ページの『arch』
269 ページの『tune』
292 ページの『#pragma altivec_vrsave』

arch

C C++

目的

コード (命令) の生成対象とする一般的なプロセッサーのアーキテクチャーを指定する。

構文

使用可能なオプションは、以下で説明するように、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、これは暗黙指定されます。
- 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んだオブジェクト・コードを生成します。

ppc970
- PowerPC 970 アーキテクチャーに固有の命令を生成します。
- _ARCH_PPC、_ARCH_PPCGR、_ARCH_PPC970 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になるようにしたい場合に、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* ととも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。 **-qtune** を指定せずに **-qarch** を指定した場合、コンパイラーは、指定されたアーキテクチャーに対応するデフォルトのチューニング・オプションを使用し、次のリストを示します: TUNE=DEFAULT。

例

32 ビットの PowerPCPPC970 アーキテクチャーのコンピューターで、myprogram.c からコンパイルされた実行可能プログラム testing を実行するように指定するには、以下を入力します。

```
xlc -o testing myprogram.c -q32 -qarch=ppc970
```

関連参照

- 269 ページの『tune』
- 30 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) コンパイル用コンパイラー・オプションの指定』
- 376 ページの『コンパイラー・モードおよびプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ』

asm

C C++

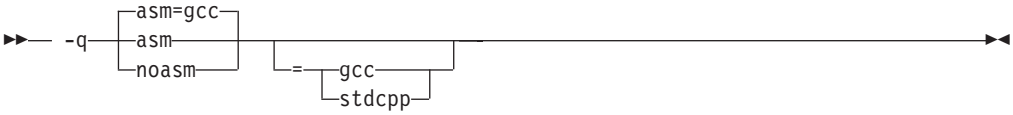
目的

asm ステートメントの解釈および以降のコード生成を制御する。

構文

C デフォルトは、言語レベルにかかわらず、**-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じになります。

C++ デフォルトは、言語レベルにかかわらず、**-qasm=gcc** です。



注

-qasm オプションとその否定の書式は、**asm** ステートメントに対応するコードが出されるかどうかを制御します。このオプションの肯定の書式は、ソース・コードの **asm** ステートメントに対応するコードを生成するように、コンパイラーに指示します。これらのサブオプションでは、**asm** ステートメントの内容を解釈するために使用される構文を指定します。たとえば、**-qasm=gcc** を指定すると、コンパイラーは、**asm** ステートメントについて拡張 gcc 構文およびセマンティクスを認識するようになります。

C トークン **asm** は C 言語キーワードではありません。したがって、言語レベル stdc89 および stdc99 (それぞれ C89 および C99 標準への厳密な準拠を指示します) では、オプション **-qkeyword=asm** も指定して、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** が有効な場合を除き、トークン **asm** はキーワードとして処理されます。C では、XL 固有のバリアント **__asm** および **__asm__** がすべての言語レベルでキーワードであり、これらを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* のサブオプションを使用すると、これらの予約語のそれぞれを個々に使用不可に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処理されるときは常に、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、以下の互い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と、マクロの値は 1 になります。生成されない場合、0 です。

__IBM_GCC_ASM
__IBM_STDCPP_ASM (C++ のみ)

通知メッセージ

オプション **-qinfo=eff** も有効なときには、**asm** ステートメントに対応するコードが生成されていない場合、コンパイラーは通知メッセージを出します。

asm 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るときは常に、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にコンパイラーに指示します。 C 言語レベル stdc89 または stdc99 では、オプション **-qkeyword=asm** も有効でなければなりません。

例

以下のコードの断片は、**-qasm** コンパイラー・オプションの簡単な使用法を示しています。

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

関連参照

- 160 ページの『langlvl』
- 134 ページの『info』
- 157 ページの『keyword』

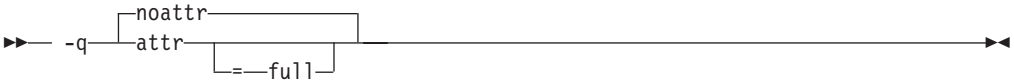
attr

C C++

目的

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。

330 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。

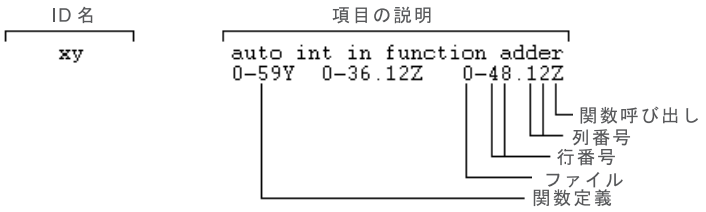
-qattr を **-qattr=full** の後に指定すると、無効になってしまいます。完全なリストが生成されます。

例

プログラム myprogram.C をコンパイルして、全 ID のコンパイラー・リストを生成するには、以下を入力します。

```
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。



関連参照

41 ページの『コンパイラーのコマンド行オプション』
215 ページの『print』
285 ページの『xref』
330 ページの『#pragma options』

B

C C++

目的

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

program は、 **-t** コンパイラー・オプションが認識する任意のプログラム名となります。プログラムの指定について詳しくは、 **t** の資料を参照してください。

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。

プログラムごとに完全パス名を形成するために、IBM XL C/C++ は、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用の標準のプログラム名にプレフィックスを追加します。

IBM XL C/C++ の実行可能ファイルのいくつかまたはすべてについて複数のレベルを保持し、使用するプログラムを指定できるようにしたい場合には、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

/lib/tmp/mine/ にある代替の **xlc++** コンパイラーを使用して myprogram.C をコンパイルするには、以下を入力します。

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

/lib/tmp/mine/ にある代替のリンケージ・エディターを使用して myprogram.C をコンパイルするには、次を入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
206 ページの『path』
255 ページの『t』

bigdata

C C++

目的

32 ビット・モードで、初期化データを 16 MB より大きいサイズにすることができます。

構文

注

32 ビット・モードで、初期化データの gcc サイズ限度は 16 MB です。共用ライブラリーの初期化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場合、このオプションを使用します。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

bitfields

C C++

目的

ビット・フィールドを符号付きとするかどうかを指定します。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、以下のとおりです。

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

C

C C++

目的

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** オプションでは、コメントが標準出力に書き込まれます。 **-P** オプションでは、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、コメントを含む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を生成させるには、以下を入力します。

```
xlc myprogram.c -P -C
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
99 ページの『E』
203 ページの『P』

-c

C C++

目的

コンパイラーに、コンパイラーのみにソース・ファイルを渡すように指示する。

構文

注

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** などの有効なソース・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。

-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** オプションと組み合わせて使用すると、コンパイラーによって作成されたオブジェクト・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、実行可能ファイルではなくオブジェクト・ファイル **myprogram.o** を生成させるには、次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、実行可能ファイルではなくオブジェクト・ファイル **new.o** を生成させるには、次のコマンドを入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
99 ページの『E』
201 ページの『o』
203 ページの『P』
254 ページの『syntaxonly』

c_stdinc

C

目的

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、以下を入力します。

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

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』
28 ページの『構成ファイル内のコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
87 ページの『cpp_stdinc』
121 ページの『gcc_c_stdinc』
248 ページの『stdinc』

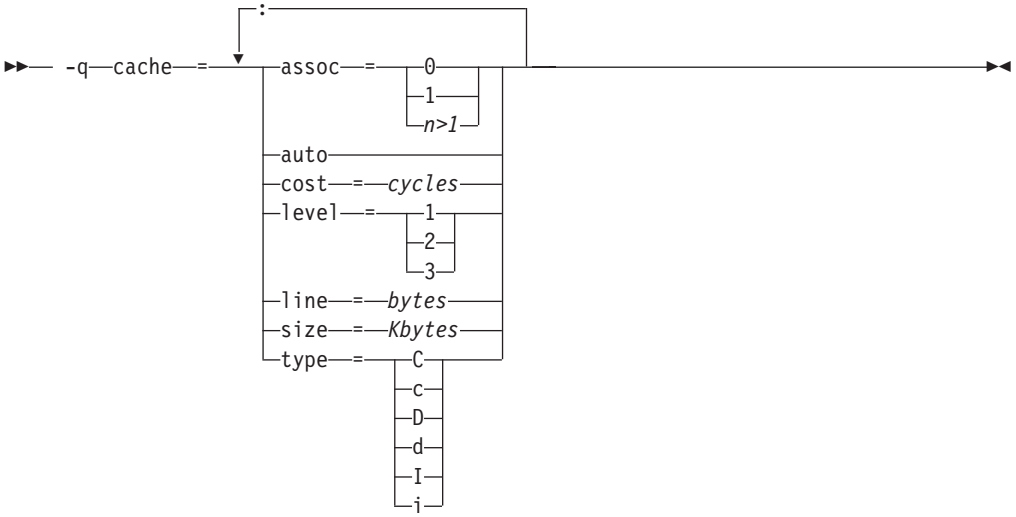
cache

C C++

目的

-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、以下のとおりです。

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 通りの集合の結合キャッシュ

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、または、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*
: 指定した型のキャッシュに、設定を適用します。この場合、*cache_type* は、以下のいずれかです。

  C または c
  : データおよび命令キャッシュの結合

  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用の、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 オプションで **-O4**、 **-O5**、または **-qipa** を指定しなければなりません。

-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。

- 可能な限り多くの構成パラメーターに対して情報を指定します。
- 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・レベルを記述します。
- 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、見積もった小さい方のキャッシュ・サイズを指定します。実際にあるキャッシュ・サイズより大きなサイズを指定して、キャッシュ・ミス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た方がよいでしょう。
- データ・キャッシュは、命令キャッシュよりもプログラム・パフォーマンスにおいて、より大きな影響を及ぼします。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、そのデータ・キャッシュに対して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、または異なる構成のマシン上でプログラムを実行した場合、プログラムのパフォーマンスが低下する場合がありますが、プログラム出力は期待どおりとなります。
- **-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します。 **-qcache** オプションを **-O4** または **-O5** オプションとともに指定する場合は、最後に指定されたオプションが優先されます。

例

結合された命令およびデータ・レベル 1 のキャッシュ (2 つの方法で結合されており、サイズが 8 KB で、64 バイトのキャッシュ行を持っているキャッシュ) でシステムのパフォーマンスを調整するには、以下を入力します。

```
xlc++ -O4 -qcache=type=c:level=1:size=8:line=64:assoc=2 file.C
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
144 ページの『ipa』
197 ページの『O、optimize』

chars

C C++

目的

コンパイラーに、**char** 型の変数をすべて **signed** または **unsigned** のいずれかとして処理するように指示する。

構文

294 ページの『#pragma chars』および 330 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** および **signed char** 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** として扱うには、次のように入力します。

```
xlc myprogram.c -qchars=signed
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
294 ページの『#pragma chars』
330 ページの『#pragma options』

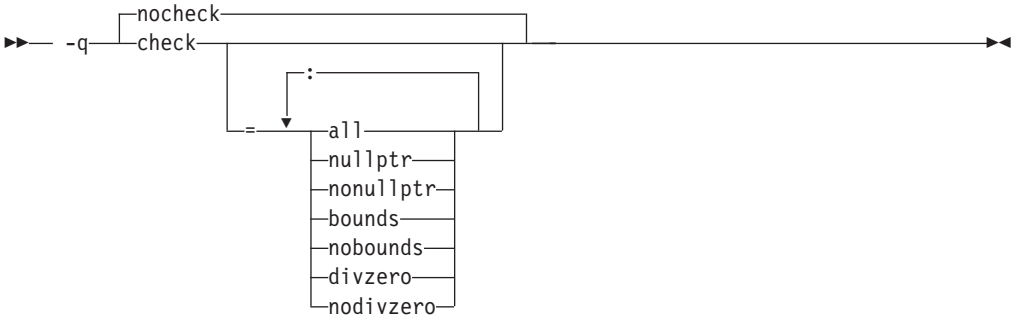
check

C C++

目的

特定のタイプの実行時検査を行うコードを生成する。違反が検出された場合は、**SIGTRAP** シグナルをプロセスに送信することによって、実行時例外が発生します。

構文



ここで、

all
以下のサブオプションをすべてオンに切り替えます。**no...** 形式の 1 つまたは複数の他のオプションとともに、 **all** オプションをフィルターとして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。

NULLptr | NONULLptr
ストレージの参照に使用するポインター変数に含まれるアドレスの実行時検査を行います。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。

bounds | nobounds
既知のサイズのオブジェクト内の添え字を指定するときに、アドレスの実行時検査を行います。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有効ではありません。

DIVzero | NODIVzero
整数除法の実行時検査を行います。 0 による除法を行おうとすると、トラップが起こります。

330 ページの『#pragma options』も参照してください。

注

-qcheck オプションには、上記のとおり、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、およびコマンド行の **-qcheck** 以外のバリエーションなしで **-qcheck** オプションを指定すると、すべてのサブオプションがオンになります。

サブオプションを指定して **-qcheck** オプションを使用すると、 no プレフィックスがない場合は指定し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、後のサブオプションによって前の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実行時の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、実行時検査がアプリケーションに組み込まれるため、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  */
}
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

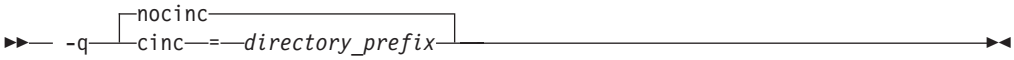
cinc

C++

目的

コンパイラーに **extern ″C″ { }** ラッパーをインクルード・ファイルのコンテンツの周りに配置するように指示します。

構文



ここで、

directory_prefix このオプションが作用するファイルが入ってい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からのインクルード・ファイルでは、トークン **extern ″C″ {** がインクルード・ファイルの最初のステートメントの前に挿入され、 **}** がインクルード・ファイルの最後のステートメントの後に付加されます。

例

アプリケーション myprogram.C がディレクトリー /usr/tmp に位置するヘッダー・ファイル foo.h を含んでおり、次のコードを含んでいるとします。

```
int foo();
```

以下を使用してアプリケーションをコンパイルします。

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

アプリケーションに以下のようなヘッダー・ファイル foo.h が含まれます。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』

compact

C C++

目的

最適化とともに使用すると、可能な場合に、実行速度を犠牲にしてコード・サイズが削減される。

構文

330 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、インラインでのコードの複製や展開による最適化を禁止することによって削減されます。実行時間が増加する可能性があります。

例

myprogram.C をコンパイルしてコード・サイズを削減するには、次のように入力します。

```
xlc++ myprogram.C -O -qcompact
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

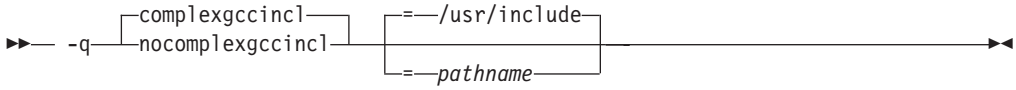
complexgccincl

C C++

目的

-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にあるインクルード・ファイルの前後で、**#pragma complexgcc(on)** および **#pragma complexgcc(pop)** ディレクティブを内部的に循環させるようにコンパイラーに命令します。

構文



ここで、

pathname インクルード・ファイルの検索パスを指定します。 *pathname* を指定しない場合は、コンパイラーは、 **/usr/include** の *pathname* であると見なします。

注

-qcomplexgccincl コンパイラー・オプションで指定されたディレクトリーで見つかったインクルード・ファイルは、**#pragma complexgcc(on)** および **#pragma complexgcc(pop)** ディレクティブによって、内部的に循環されます。

-qnocomplexgccincl で見つかったインクルード・ファイルは、これらのディレクティブによって循環されることはありません。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
111 ページの『float』
297 ページの『#pragma complexgcc』

cpluscmt

C

目的

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、以下のようにそれぞれ異なります。

- **xlc**、**xlc_r**、**cc**、または **cc_r** によっ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** に対して設定されていると、 **-qcpluscmt** も暗黙的に選択されます。コマンド行で **-qlanglvl** オプションの後に、 **-qnocpluscmt** を指定することによって、これらの暗黙選択をオーバーライドすることができます。例えば、次のようにします。 **-qlanglvl=stdc99 -qnocpluscmt** または **-qlanglvl=extc99 -qnocpluscmt**
- そうでない場合は、デフォルト設定は **-qnocpluscmt** です。

注

__C99_CPLUSCMT コンパイラー・マクロは、**cpluscmt** が選択される際に定義されます。

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、またはコメントを除いて、 C++ のコメントの開始となります。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、次の C++ コメント内で無視されます。

- //
- /*
- */

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、2 つの間に何もはさまずに隣接していなければなりません。スラッシュの右側から改行文字で示される論理ソース行の最後までが、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、以下の方法でコメントすべてを処理します。

- **-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き数に現れない限り、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 (¥) 文字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には、コメントは、それらの物理ソース行にわたることができます。 3 文字表記 (??/) によって円記号を表すこともできます。

例

1. **C++ のコメントの例**

   以下の例に、C++ のコメントの使用を示します。

```
// 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 オプションの場合の出力は、以下のとおりです。

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

int d;
```

   -P -C オプションの場合の C89 モード出力は、以下のとおりです。

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

   -E オプションの場合の出力は、以下のとおりです。

```
int a;
int b;

int c;

int d;
```

-E -C オプションの場合の C89 モード出力は、以下のとおりです。

```
#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** オプションの場合の拡張モード出力は、以下のとおりです。

```
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 オプションの場合の出力は、以下のとおりです。

```
int a;
int b;

int c;
```

   -P -C オプションの場合の出力は、以下のとおりです。

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

   -E オプションの場合の出力は、以下のとおりです。

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

int c;
```

   -E -C オプションの場合の出力は、以下のとおりです。

```
#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 - マクロ関数の引き数**

   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 オプションの場合の出力は、以下のとおりです。

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

int d;
```

   -P -C オプションの場合の出力は、以下のとおりです。

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

   -E オプションの場合の出力は、以下のとおりです。

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

int d;
```

   -E -C オプションの場合の出力は、以下のとおりです。

```
#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. so をコンパイルするには、次のように入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
70 ページの『C』

99 ページの『E』
160 ページの『langlvl』
203 ページの『P』

cpp_stdinc

C++

目的

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** 標準検索パスを作成するには、以下を入力します。

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

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』
28 ページの『構成ファイル内のコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
72 ページの『c_stdinc』
122 ページの『gcc_cpp_stdinc』
248 ページの『stdinc』

crt

C C++

目的

リンク時に標準システム始動ファイルを使用するようにリンカーに指示する。

構文

注

-qnocrt コンパイラー・オプションが指定されている場合には、コンパイラーは、リンク時に標準システム始動ファイルを使用しません。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
177 ページの『lib』

D

C C++

目的

#define プリプロセッサー・ディレクティブにあるのと同様にマクロ *name* を定義します。*definition* は、 *name* に割り当てるオプションの定義または値です。

構文

注

また、マクロ名がすでに **-D** コンパイラー・オプションによって定義されていない場合は、**#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ができます。

-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。)

-D オプションによってすでに定義されているマクロ名を定義するのに **#define** ディレクティブを使用すると、エラー状態になります。

プログラムの移植性および標準への適合性を援助するために、オペレーティング・システムは、 **-D** オプションで設定することができるマクロ名を参照するヘッダー・ファイルをいくつか提供します。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。

-U*name* オプションは、**-D** オプションによって定義されたマクロを未定義にするために使用され、その優先順位は **-D***name* オプションよりも上です。

例

1. `myprogram.c` において、 `COUNT` という名前のインスタンスをすべて `100` で置換するには、以下を入力します。

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

   これは、ソース・ファイルの先頭に **#define COUNT 100** があることと同等です。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
271 ページの『U』
395 ページの『付録 A. 事前定義マクロ』

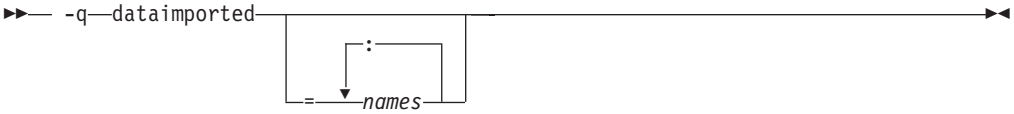
dataimported

C C++

目的

インポートされるもの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、インポートされる変数は、ライブラリーの共用部分と動的にバインドされます。

- **-qdataimported** を指定すると、変数をすべてインポートするものであると見なすようにコンパイラーに指示します。
- **-qdataimported=***names* は、指定された変数にインポートされるもの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、変数の*名前* は、マングルされた名前を使用して指定する必要があります。例えば、次のコード・セグメントを想定します。

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

  コンパイラー・オプションを次のように指定することによって、変数 **C::i** をインポートするように指定できます。

  ```
  -qdataimported=i__1C
  ```

  オペレーティング・システム **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された名前を取得することができます。マングルされた名前を検証するには、**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。

変数名をリストするオプション **:**	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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
92 ページの『datalocal』

datalocal

C C++

目的

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。

- ローカルにする変数を指定する必要があります。名前を指定しない場合、リンカーはリンク時にリンクできません。
- **-qdatalocal=***names* は、指定された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、変数の*名前* は、マングルされた名前を使用して指定する必要があります。例えば、次のコード・セグメントを想定します。

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

  コンパイラー・オプションを次のように指定することによって、変数 **C::i** をローカル・データとして指定できます。

  ```
  -qdatalocal=i__1C
  ```

  オペレーティング・システム **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された名前を取得することができます。マングルされた名前を検証するには、**c++filt** ユーティリティーを使用します。

インポートする変数がローカルであると見なされると、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。

変数名をリストするオプション:	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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
90 ページの『dataimported』

dbxextra

C

目的

すべての **typedef** 宣言、**struct** 型定義、 **union** 型定義、および **enum** 型定義をデバッグ用に組み込むことを指定する。

構文

330 ページの『#pragma options』も参照してください。

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

デバッグ用に myprogram.c 内のシンボルをすべて組み込むには、以下を入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』
330 ページの『#pragma options』

digraph

C C++

目的

キーボードにない文字を表すために、連字キーの組み合わせおよびキーワードを使用できるようにする。

構文

330 ページの『#pragma options』も参照してください。

デフォルト

- C **-qlanglvl** が、**extc99** または **stdc99** 以外の値に設定された場合、**-qnodigraph**。
- C **-qlanglvl** が、**extc99** または **stdc99** に設定された場合、**-qdigraph**。
- C++ **-qdigraph**

注

連字は、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、以下を入力します。

```
xlc++ myprogram.C -qnodigraph
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
160 ページの『langlvl』
268 ページの『trigraph』
330 ページの『#pragma options』

directstorage

C C++

目的

ライトスルー対応ストレージまたはキャッシュ禁止ストレージが参照可能で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、ライトスルー対応またはキャッシュ禁止のストレージが参照される可能性があること、および適切なコンパイラー出力を生成すべきであることを、コンパイラーに通知します。

PowerPC アーキテクチャーを使用すると、キャッシュ編成のさまざまなインプリメンテーションが可能になります。確実にアプリケーションをすべてのインプリメンテーションで正しく実行するためには、さまざまな命令およびデータ・キャッシュの存在を想定し、それに従ってアプリケーションをプログラムする必要があります。

関数を確実に正しく実行するため、プログラムで指定されたストレージ管理属性、および実行されている関数に応じて、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に割り振ってから、それを一連のゼロに初期化します。 **dcbz** 命令は、データの大きなブロックをゼロ化する場合にパフォーマンスを上げるために使用できますが、以下のいずれかの条件下で位置合わせエラーが発生するため、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さ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 か L2 キャッシュが使用不可である。

-qdirectstorage を指定すると、**dcbz** 命令の生成を抑制し、上記の位置合わせエラーが回避されます。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

dollar

C C++

目的

$ シンボルを ID の名前で使用できるようにする。

構文

-qdollar が有効である場合、ID 内のドル記号 **$** は基本文字として処理されます。オプション **-qnodollar** および **-qlanglvl=ucs** が両方とも有効である場合、ドル記号は外字として処理され、¥u0024 に変換されます。

例

プログラム内の ID で **$** を使用できるように myprogram.c をコンパイルするには、以下のように入力します。

```
xlc myprogram.c -qdollar
```

関連参照

- 330 ページの『#pragma options』

E

C C++

目的

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指示する。

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、汎用のプリプロセスのために使用します。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、同標準の “Phases of Translation” の記述のとおりに除去されます。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。

-E を **-M** オプションとともに使用すると、ファイル名のサフィックスが **.C**、**.cpp**、**.cc** (すべての 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) であるファイルのみが処理されます。認識されないファイル名サフィックスが付いたソース・ファイルは、C ファイルと見なされてプリプロセスされ、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、コメントは単一の空白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。マクロ関数の引き数にあるコメントは削除されます。

デフォルトでは、ソース・ファイルがプリプロセス、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生成されます。

例

myprogram.C をコンパイルしてプリプロセスされたソースを標準出力に送るには、以下を入力します。

```
xlc++ myprogram.C -E
```

myprogram.C に以下のようなコード・フラグメントがある場合は、

```
#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);
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
185 ページの『M』
201 ページの『o』
203 ページの『P』
254 ページの『syntaxonly』

e

C C++

目的

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、**-qmkshrobj** 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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
196 ページの『mkshrobj』

eh

C++

目的

コンパイル中のモジュールで例外処理が使用可能であるかどうかを制御する。

構文

ここで、

eh　例外処理が使用可能です。

noeh　プログラムで C++ 構造化例外処理を使用しない場合は、アプリケーションにとって必要のないコードが生成されないようにするために、**-qnoeh** を指定してコンパイルしてください。

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
229 ページの『rtti』

enablevmx

C C++

目的

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

構文

デフォルト

VMX 命令をサポートするオペレーティング・システム・バージョンでは、**-qenablevmx** がデフォルトで設定されています。その他の場合は、デフォルトは **-qnoenablevmx** です。

注

一部のプロセッサーは VMX (Vector Multimedia Extension) 命令をサポートできます。マルチメディア・アプリケーションなどのアルゴリズムに集約的なタスクで使用する場合、これらの命令はより高いハイパフォーマンスを提供します。

-qenablevmx コンパイラー・オプションは VMX 命令の生成を使用可能にしますが、オペレーティング・システム・バージョンが VMX 命令をサポートしていない場合、このオプションを使用すべきではありません。

また、**-qnoenablevmx** が有効な場合、**-qaltivec** は使用できません。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
62 ページの『altivec』
63 ページの『arch』
127 ページの『hot』

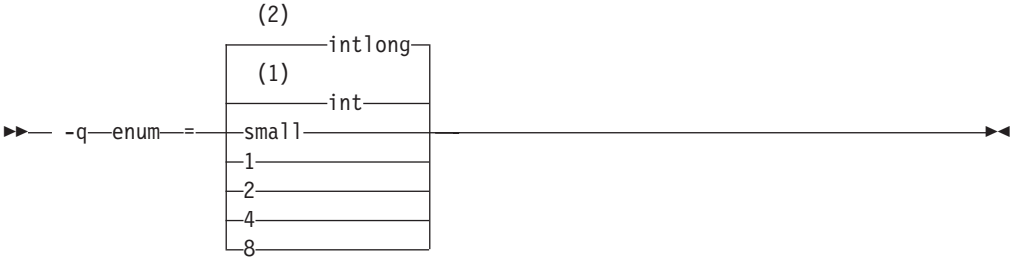
enum

C C++

目的

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
: 列挙が 1 バイトのストレージを占有すること、および、列挙の値の範囲が **signed char** の制限内である場合、型が **char** になること、それ以外の場合は **unsigned char** になることを指定します。

2
: 列挙が 2 バイトのストレージを占有すること、および、列挙の値の範囲が **signed short** の制限内である場合、型が **short** になり、それ以外の場合は **unsigned short** になることを指定します。

4
: 列挙が 4 バイトのストレージを占有すること、および、列挙の値の範囲が **signed int** の制限内である場合、型が **int** になり、それ以外の場合は **unsigned int** になることを指定します。

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・コンパイル・モードでは、列挙値の範囲が **signed long long** の制限内にある場合、その列挙は **long long** 型で、そうでない場合は **unsigned long long** です。
64 ビット・コンパイル・モードでは、列挙値の範囲が **signed long** の制限内にある場合は、その列挙は **long** 型で、そうでない場合は **unsigned long** です。

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。値は、 C コンパイルにおける **signed int** の範囲を超えることはできません。

intlong
: 列挙の値の範囲が **int** の限度を超える場合、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。 **-qenum=8** の説明を参照してください。
列挙の値の範囲が **int** の制限を超えない場合は、列挙は 4 バイトのストレージを占有し、**int** によって示されます。

small
: 列挙が列挙内の値の範囲を正確に表せる最小のスペース (1、2、4、または 8 バイトのストレージ) を占有するように指定します。値の範囲に負の値が含まれない限り、符号は *unsigned* です。
8 バイトの enum が結果として生じる場合、実際に使用される列挙型はコンパイル・モードに依存しています。**-qenum=8** の説明を参照してください。

302 ページの『#pragma enum』および 330 ページの『#pragma options』も参照してください。

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型が使用されます。

-qenum=1|2|4|8 オプションは、 **enum** 変数 に特定の量のストレージを割り振ります。指定されたストレージ・サイズが **enum** 変数の範囲が必要とするサイズよりも小さい場合は、重大エラー・メッセージが出され、コンパイルが停止します。

ISO C 1989 および ISO C1999 規格では、列挙範囲が **int** の範囲を超えないことが要求されます。有効な **-qlanglvl=stdc89** または **-qlanglvl=stdc99** でコンパイルすると、列挙の値が **int** の範囲を超える場合、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、重大エラー・メッセージが出されてコンパイルが停止する。
- **-qenum** のほかのすべての設定には、情報メッセージが出されてコンパイルが継続する。

以下の表に、事前定義型の選択のための優先順位を示します。この表には、事前定義型、対応する事前定義型の **enum** 定数の最大範囲、およびその事前定義型に必要なストレージの量、つまり、最小サイズの **enum** に適用した場合に、 **sizeof** 演算子によって得られる値も示します。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
302 ページの『#pragma enum』
330 ページの『#pragma options』

Enum サイズと型 - 特に断りがない限り、すべての型が signed です。

	enum=1		enum=2		enum=4		enum=8			
							32 ビット・コンパイル・モード		64 ビット・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. これらの列挙は **-qenum=1|2|4** 設定では大き過ぎます。重大エラーが出されてコンパイルが停止します。
   この状態を訂正するには、列挙の範囲を狭めて、より大きな **enum** 設定を選択するか、または **small** または **intlong** などの動的 **enum** 設定を選択する必要があります。
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 規格にコンパイルするときに、**int** の範囲を超えてはいけません。有効な **-qlanglvl=stdc89** または **-qlanglvl=stdc99** でコンパイルすると、列挙の値が **int** の範囲を超える場合、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、重大エラー・メッセージが出されてコンパイルが停止する。
   b. **-qenum** のほかのすべての設定には、情報メッセージが出されてコンパイルが継続する。

Enum サイズと型 - 特に断りがない限り、すべての型が signed です。

	enum=int		enum=intlong				enum=small			
			32 ビット・コンパイル・モード		64 ビット・コンパイル・モード		32 ビット・コンパイル・モード		64 ビット・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. これらの列挙は **-qenum=1|2|4** 設定では大き過ぎます。重大エラーが出されてコンパイルが停止します。この状態を訂正するには、列挙の範囲を狭めて、より大きな **enum** 設定を選択するか、または **small** または **intlong** などの動的 **enum** 設定を選択する必要があります。
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 規格にコンパイルするときに、**int** の範囲を超えてはいけません。有効な **-qlanglvl=stdc89** または **-qlanglvl=stdc99** でコンパイルすると、列挙の値が **int** の範囲を超える場合、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、重大エラー・メッセージが出されてコンパイルが停止する。
   b. **-qenum** のほかのすべての設定には、情報メッセージが出されてコンパイルが継続する。

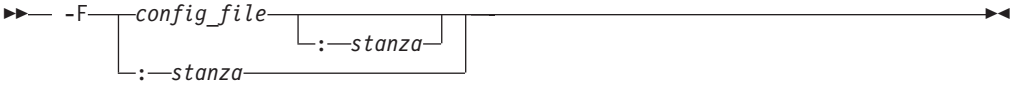
F

C C++

目的

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、以下のとおりです。

config_file コンパイラー構成ファイルの名前を指定します。
stanza コンパイラーを起動するため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・ファイル内で指定するファイル名またはスタンザは、すべて構成ファイルで指定したデフォルトをオーバーライドします。

構成ファイルの内容に関する詳細については、 28 ページの『構成ファイル内のコンパイラー・オプションの指定』を参照してください。

-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
```

関連タスク

28 ページの『構成ファイル内のコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
67 ページの『B』
255 ページの『t』
281 ページの『W』

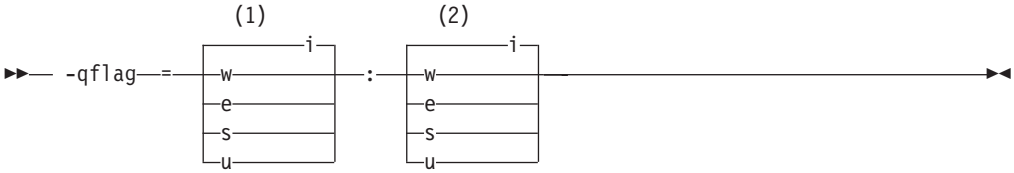
flag

C C++

目的

リストに報告させるとともに端末に表示させる診断メッセージの最低限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 リストに報告させる、最低限の重大度レベルのメッセージ

2 端末について報告させる、最低限の重大度レベルのメッセージ

ここで、メッセージ重大度レベルは、以下のとおりです。

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

330 ページの『#pragma options』も参照してください。

注

リスト報告および端末報告の両方について、最低限のメッセージ重大度レベルを指定しなければなりません。

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、その際に生成された全メッセージをリストに示し、ワークステーションにはエラー・メッセージ以上のメッセージ (およびエラーの修正に役立つ関連する通知メッセージ) のみが表示されるようにするには、次のように入力します。

```
xlc++ myprogram.C -qflag=i:e
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
134 ページの『info』
282 ページの『w』

330 ページの『#pragma options』
379 ページの『コンパイラー・メッセージ』

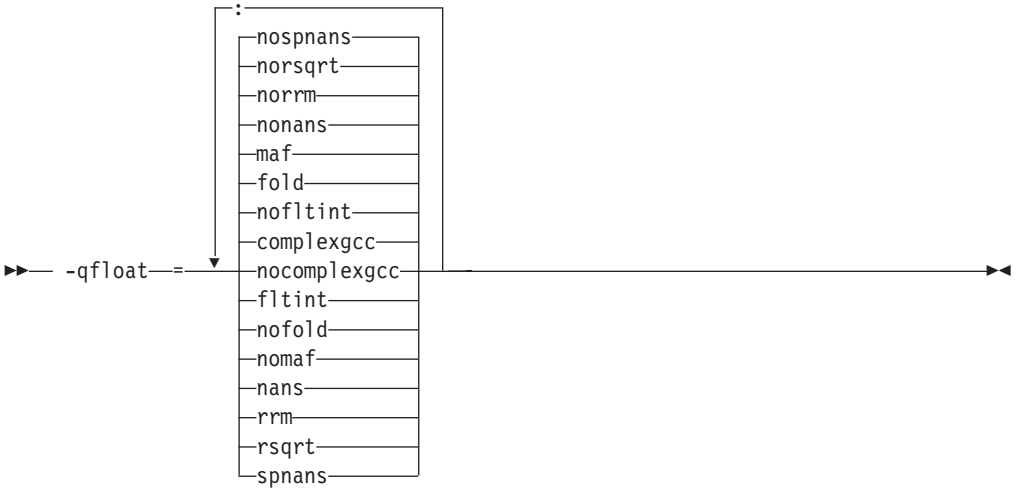
float

C C++

目的

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、より正確にしたりするための各種の技法を提供します。

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。 330 ページの『#pragma options』も参照してください。

注

デフォルト設定以外の **float** サブオプションを使用すると、浮動小数点計算でさまざまな結果を作成することになります。特定のサブオプションに対する必要な条件をすべて満たしていない場合は、誤った計算結果になる場合があります。このため、IEEE 浮動小数点値に関する浮動小数点計算の経験があり、プログラムにエラーが混入する可能性を適切に評価することができる場合でなければ、このオプションを使用してはなりません。

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
: **_complex** の受け渡しで、GCC との互換性を持たせます。 32 ビット・モードでコンパイルするときのデフォルトは **float=complexgcc** で、 64 ビット・モードでコンパイルするときのデフォルトは **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。

- 有効な **-O2** を使用すると、**-qfloat=nofltint** は暗黙設定です。
- 有効な **-O3** 以上を使用すると、**-qfloat=fltint** は暗黙指定されます。

  -O3 オプションによって浮動小数点から整数への変換に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。

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。負のゼロ結果が作成されることがあります。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

追加の命令を生成して、実行時に単精度から倍精度に変換するときにシグナル型 NaN (Not-a-Number) を検出します。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 規格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型 NaN である場合に引き起こされる比較演算における無効演算例外を検出します。

rrm
norrm

浮動小数点の最適化が、正および負の無限大への実行時丸めモードと非互換にならないようにします。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*最も近い値に丸める* 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法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** 関数呼び出しの場合に設定されない 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 がすでに指定してある場合は、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

追加の命令を生成して、単精度から倍精度への変換時にシグナル型 NaN を検出します。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を評価し、乗算・加算命令を生成しないように myprogram.C をコンパイルするには、以下を入力します。

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

関連参照

- 63 ページの『arch』
- 81 ページの『complexgccincl』
- 114 ページの『flttrap』
- 249 ページの『strict』
- 297 ページの『#pragma complexgcc』
- 330 ページの『#pragma options』

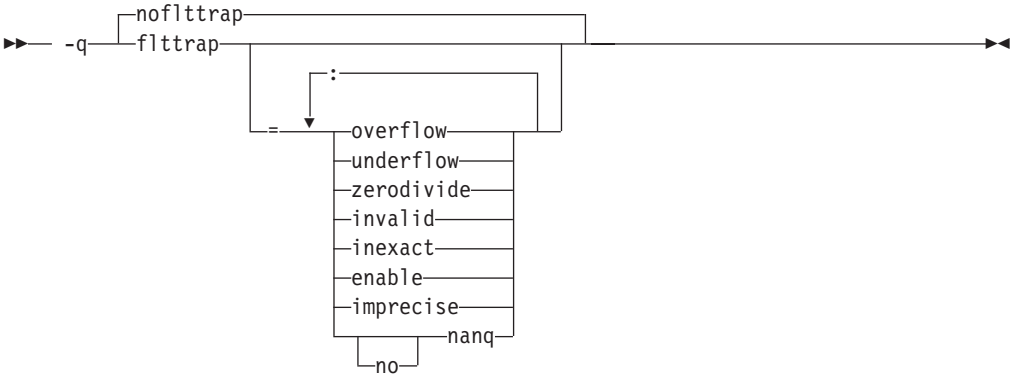
flttrap

C C++

目的

追加の命令を生成して、実行時の浮動小数点例外を検出およびトラップします。

構文



ここで、サブオプションでは以下が行われます。

ENable	メインプログラムの開始時点で指定された例外を使用可能にします。(以下で説明する) **nanq** を除き、ソース・コードを変更せずに、下記の例外トラッピング・オプション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法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	浮動小数点演算で処理または生成される NaNQ (Not a Number Quiet) 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。 **nanq** および **nonanq** 設定は、 **-qnoflttrap**、**-qflttrap**、または **-qflttrap=enable** の影響を受けません。

330 ページの『#pragma options』も参照してください。

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型 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** とともに **-qfloat=nans** を使用してください。

-qflttrap オプションを **-qoptimize** オプションとともに指定する場合、コンパイラーは、以下の例に示す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、直前の除法の結果によって乗算されたゼロに戻ります。

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 0 による除法を検出するように myprogram.c をコンパイルするには、以下を入力します。

```
xlc myprogram.c -qflttrap=overflow:underflow:zerodivide:enable
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
111 ページの『float』
197 ページの『O、optimize』

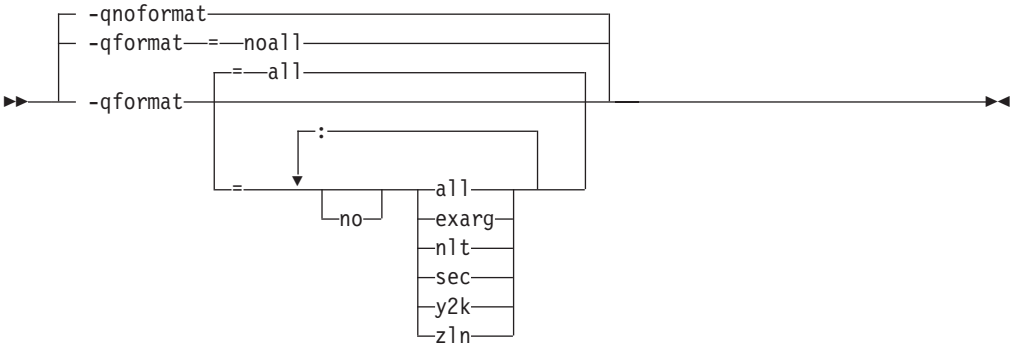
format

C C++

目的

ストリング入出力フォーマット指定で起こりうる問題の警告を出します。診断された関数は、printf、scanf、strftime、strfmon ファミリー関数、およびフォーマット属性でマークされた関数です。

構文



サブオプションは、以下のとおりです。

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　過剰な引き数が printf および scanf スタイル関数呼び出しに表示された場合に警告を出します。

nlt　フォーマット関数がそのフォーマット引き数を va_list として取らない限り、書式制御ストリングがストリング・リテラルではない場合に警告を出します。

sec　フォーマット関数の使用中に起こりうるセキュリティー問題の警告を出します。

y2k　2 桁の年数を作成する *strftime* フォーマットの警告を出します。

zln　長さがゼロのフォーマットの警告を出します。

注: 上記のサブオプションのいずれかの前に **no** を指定すると、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、コマンド行で **-qformat=noy2k** を指定します。

注

コマンド行で **-qformat** が指定されていない と、コンパイラーはデフォルト設定を **-qnoformat** と見なします。これは、**-qformat=noall** と同等です。

サブオプションなしで、コマンド行で **-qformat** が指定されると、コンパイラーはデフォルト設定を **-qformat=all** と見なします。

例

1. すべての書式制御ストリング診断を使用可能にするには、以下のいずれかを入力します。

```
xlc++ myprogram.C -qformat=all
xlc++ myprogram.C -qformat
```

2. y2k データ診断検査以外のすべてのフォーマット診断検査を使用可能にするには、以下を入力します。

```
xlc++ myprogram.C -qformat=all:noy2k
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

fullpath

C C++

目的

-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、ファイルを正常に探すことができます。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』

funcsect

C C++

目的

個別のオブジェクト・ファイル、制御セクション (csect) のそれぞれの関数ごとに命令を配置する。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・ファイル内に定義されたすべての関数を結合して、単一の制御セクションで構成されます。

構文

注

複数の csects を使用すると、リンケージ・エディターが、呼び出されていない関数を除去したり、または呼び出されるすべての場所で最適化プログラムがインラインした関数を除去できるようにすることによって、最終の実行可能ファイルのサイズが削減されることがあります。

このオプションが **#pragma options** で指定されているとき、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の前で指定しなければなりません。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

g

C C++

目的

GNU GDB デバッガーなどのデバッグ・ツールで使用する情報を生成します。

構文

注

-g を指定すると、明示的にそれを要求しない限り、すべてのインライン化がオフされます。例を以下に示します。

オプション	インライン化への影響
-g	インライン化なし。
-O	インライン宣言関数。
-O -Q	インライン宣言関数とそのほかの自動インライン関数。
-g -O	インライン宣言関数。
-g -O -Q	インライン宣言関数とそのほかの自動インライン関数。

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

参照されているシンボルおよび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** とともに使用します。

-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 を生成させ、デバッグできるようにするには、以下を入力します。

```
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実行可能プログラム testing_all を生成させ、デバッグできるようにするには、以下を入力します。

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

関連参照

- 197 ページの『O、optimize』
- 221 ページの『Q』

gcc_c_stdinc

C

目的

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 **-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、以下を入力します。

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

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』
28 ページの『構成ファイル内のコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
72 ページの『c_stdinc』
87 ページの『cpp_stdinc』
122 ページの『gcc_cpp_stdinc』
248 ページの『stdinc』

gcc_cpp_stdinc

C++

目的

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、以下を入力します。

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

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』
28 ページの『構成ファイル内のコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
72 ページの『c_stdinc』
87 ページの『cpp_stdinc』
121 ページの『gcc_c_stdinc』
248 ページの『stdinc』

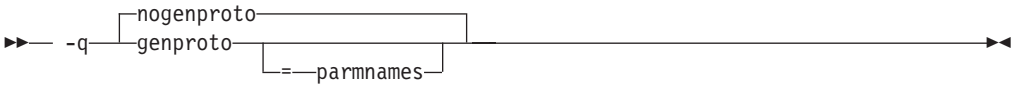
genproto

C

目的

K&R 関数定義から ANSI プロトタイプを生成する。これは、K&R から ANSI に簡単に変遷するのに役立ちます。

構文



注

parmnames を指定せずに **-qgenproto** を使用すると、プロトタイプはパラメーター名なしで生成されます。**parmnames** を指定すると、パラメーター名がプロトタイプに組み込まれます。

例

以下の関数が foo.c にある場合に、

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

以下を指定すると、

```
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、以下を指定すると、

```
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狭い型の引き数 (**char**、 **short**、**float** など) すべてが拡張されることが ANSI で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** として示されることに注意してください。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

halt

C C++

目的

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指示する。

構文

注:

1 C++ コンパイルのデフォルトです。

2 C コンパイルのデフォルトです。

ここで、重大度が増大する順での重大度レベルは、以下のとおりです。

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

330 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、コンパイラーが停止するときは、コンパイラーの戻りコードはゼロ以外です。

-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**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させるには、以下を入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
109 ページの『flag』
190 ページの『maxerr』
330 ページの『#pragma options』

haltonmsg

C++

目的

指定した *msg_number* が検出されたときに、コンパイル・フェーズ後に停止するよう、コンパイラーに指示する。

構文

注

-qhaltonmsg オプションの結果、コンパイラーが停止するときは、コンパイラーの戻りコードはゼロ以外です。

-qhaltonmsg オプションで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される必要があります。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
379 ページの『コンパイラー・メッセージ』

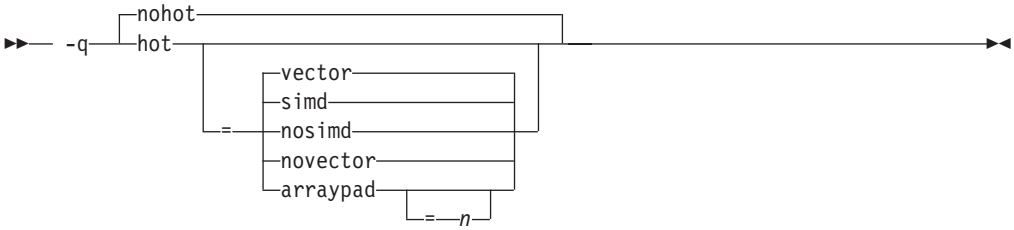
hot

C C++

目的

最適化中に、**高位**のループ分析と**変換**を実行するようコンパイラーに指示します。

構文



ここで、

arraypad
: コンパイラーは、利点があると考えられる個所に、任意の配列を適宜選択した分量ごとに埋め込みます。すべての配列が埋め込まれる必要はなく、いろいろな異なる配列が異なる分量ごとに埋め込まれる場合があります。

arraypad=*n*
: コンパイラーは、すべての配列をコードに埋め込みます。埋め込みの分量は、正整数の値でなければならず、各配列は、エレメントの整数値で埋め込まれます。 *n* は整数値であるため、埋め込み値を配列エレメントの最大サイズの倍数 (典型的な値は 4、8、または 16) にすることをお勧めします。

simd | nosimd
: コンパイラーは、連続する配列のエレメントに関してループ内で実行される特定の操作を、VMX (Vector Multimedia Extension) 命令に対する呼び出しに変換します。この呼び出しは、いくつかの結果を一度に計算するため、個々の結果を連続して計算するよりは速くなります。

  -qhot=nosimd を指定すると、コンパイラーはループおよび配列で最適化を実行しますが、VMX 命令に対する呼び出しで特定のコードを置き換えることを避けます。

  このサブオプションは、**-qarch=ppc970** が有効で、オペレーティング・システムが VMX 命令をサポートするバージョンの場合にのみ有効です。

vector | novector
: コンパイラーは、連続する配列のエレメント (例えば、平方根、逆数平方根) に関してループ内で実行される特定の操作をライブラリー・ルーチンの呼び出しに変換します。この呼び出しは、同時にいくつかの結果を計算します。これは、各結果を逐次計算するよりも高速です。

  -qhot=novector を指定すると、コンパイラーは、ループおよび配列について高位の変換を実行しますが、呼び出しで特定のコードをベクトル・ライブラリー・ルーチンに置き換える個所の最適化は行いません。
  -qhot=vector オプションは、使用中のプログラムの結果の精度に影響する場合があるため、精度の変更を受け入れられない場合は、 **-qhot=novector** または **-qstrict** のいずれかを指定してください。

デフォルト

-qhot、 **-O4**、または **-O5** オプションを指定したとき、**-qhot=simd** および **-qhot=vector** サブオプションはデフォルトでオンになります。

注

同様に、コマンド行で **-qhot** を指定するときに、最低でもレベル 2 の最適化を指定しなければ、コンパイラーは **-O2** と見なします。

キャッシュ・アーキテクチャーのインプリメンテーションにより、 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。オプショナルの **arraypad** サブオプションにより、コンパイラーは、配列の次元を増加できるようになります。そうすることにより、配列処理ループの効率が向上する可能性があります。 2 の累乗のいくつかの次元で大規模な配列 (特に最初の配列) がある場合、または配列処理プログラムがキャッシュ・ミスまたはページ不在によってスローダウンしている場合は、 **-qhot=arraypad** を指定することを考慮に入れてください。

simd サブオプションは配列データを最適化し、ターゲット・アーキテクチャーがこのような操作を許可する場合、並列して数学演算を実行します。並列操作は 16 バイトのベクター・レジスターで起こります。コンパイラーは、最適化を容易にするため、レジスター長を超えるベクターを 16 バイト単位に分割します。 16 バイト単位では、以下のデータ型の 1 つを含むことができます。

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

短いベクトル化は倍精度の浮動小数点数値をサポートしないため、**-qarch=ppc970** を指定する必要があります。**-qhot=simd** 最適化を適用すると、大きなイメージ処理要求があるアプリケーションに役立ちます。

vector サブオプションは配列データを最適化し、適用できる場合、並列して数学演算を実行します。コンパイラーは、ベクター・サイズの制限がない標準の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数値をサポートするため、大きな数値処理要求があるアプリケーションでは役立ちます。

-qhot=arraypad および **-qhot=arraypad=***n* は両方とも安全なオプションではありません。これらは、埋め込みが実行された場合にコードの中断の原因となりうる再シェーピングまたは同等性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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
70 ページの『C』
197 ページの『O、optimize』

I

C C++

目的

絶対パスを指定しない **#include** ファイル名に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 (**/u/golnaz**、 **/tmp**、 **./subdir** など) でなければなりません。コンパイラーは directory にスラッシュ (**/**) を付加し、検索前にファイル名と連結させます。パス・ディレクトリーは、名前がスラッシュ (**/**) で始まらない **#include** 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。

構成ファイルおよびコマンド行の両方で **-I** directory オプションを指定した場合は、構成ファイルで指定したパスが最初に検索されます。

-I directory オプションは、コマンド行で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。ディレクトリーの検索に関する詳細については、『*相対パス名を使用したインクルード・ファイルのディレクトリー検索順序*』を参照してください。

#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、インクルード・ファイルについて **/usr/tmp** と **/oldstuff/history** を順に検索するには、次のように入力します。

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

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』

関連参照

41 ページの『コンパイラーのコマンド行オプション』

idirfirst

C C++

目的

#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

330 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 オプションを*使用しない* 場合に、**#include "***file_name***"** ディレクティブで組み込まれるファイルの標準の検索順序は、以下のとおりです。

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** オプションで指定した 1 つ以上の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 を指定すると、 **-I** オプションで指定したディレクトリーが検索されてから、現行のファイルがあるディレクトリーが検索されるようになります。

-qidirfirst は、 **#include <file_name>** ディレクティブの検索順序には影響を及ぼしません。

-qidirfirst は、 **#include "file_name"** および **#include <file_name>** の両方について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、*相対パス名を使用したインクルード・ファイルのディレクトリー検索順序* で説明しています。

最後の有効な **#pragma options** [**no**]**idirfirst** は、それ以後の **#pragma options** [**no**]**idirfirst** によって置き換えられるまで、有効なままです。

例

myprogram.c をコンパイルして、インクルード・ファイルについて **/usr/tmp/myinclude** を検索させてから現行ディレクトリー (ソース・ファイルがある) を検索させるには、以下を入力します。

```
xlc myprogram.c -I/usr/tmp/myinclude -qidirfirst
```

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
129 ページの『I』

248 ページの『stdinc』
330 ページの『#pragma options』

ignerrno

C C++

目的

コンパイラーが、システム呼び出しによって **errno** が変更されないと想定して最適化を行うことを許可する。

構文

330 ページの『#pragma options』も参照してください。

注

例外が発生すると、一部のシステム・ライブラリー・ルーチンは **errno** を設定します。この設定および以後の **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** も設定します。最適化と **errno** を設定する機能の両方が必要な場合は、コマンド行で最適化オプションの後に **-qnoignerrno** を指定する必要があります。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
197 ページの『O、optimize』
330 ページの『#pragma options』

ignprag

C C++

目的

特定のプラグマ・ステートメントを無視するようにコンパイラーに指示する。

構文

ここで、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	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。
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。

330 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、診断が困難な実行時エラーが発生します。実行時エラーが発生するものの、**-O** オプションとともに **-qignprag** オプションを使用するとエラーが消失するときは、別名割り当てプラグマで指定し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させるには、以下を入力します。

```
xlc myprogram.c -qignprag=isolated
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
299 ページの『#pragma disjoint』
318 ページの『#pragma isolated_call』
330 ページの『#pragma options』
355 ページの『並列処理を制御するプラグマ』

info

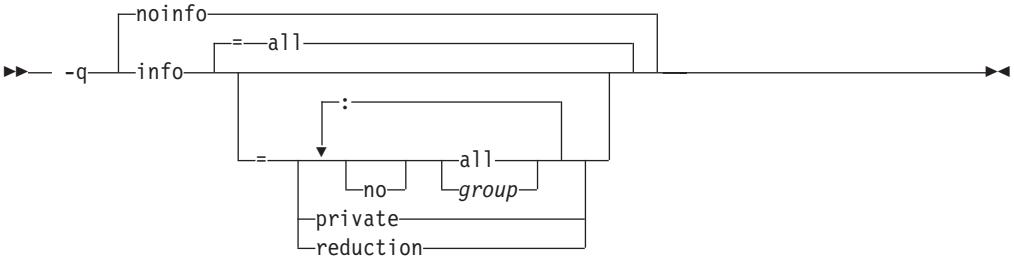
C C++

目的

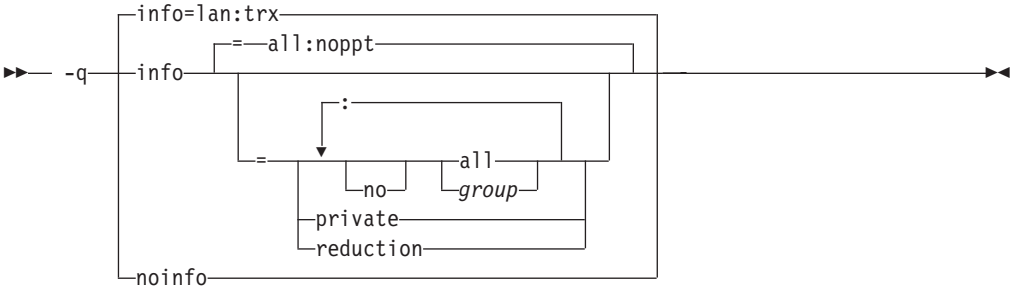
通知メッセージを作成する。

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。

312 ページの『#pragma info』および 330 ページの『#pragma options』も参照してください。

デフォルト

コマンド行で **-qinfo** を指定しなければ、コンパイラーは以下のことを想定します。

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

コマンド行で、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のことを想定します。

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、すべてのグループの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* ...] 形式、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にあるメッセージを一時的に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 **-qinfo** 形式と **-qinfo=all** 形式の効果は同じです。

  C++ オプションの **-qinfo** および **-qinfo=all** 形式は両方とも同じ効果がありますが、**ppt** グループ (プリプロセッサー・トレース) を組み込みません。

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約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* は以下の 1 つまたは複数です。

group	戻される、または抑止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動きをする場合がある C コード。
cls|nocls	C++ クラス。
cmp|nocmp	符号なし比較で生じうる冗長度。
cnd|nocnd	条件式で生じうる冗長度または問題。
cns|nocns	定数を含む命令。
cnv|nocnv	型変換。
dcl|nodcl	宣言の整合性。
eff|noeff	効果のない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	初期設定で起こりうる問題。
inl|noinl	インラインされていない関数。
lan|nolan	言語レベル効果。
obs|noobs	廃止されたフィーチャー。
ord|noord	評価の順序が指定されていない。
par|nopar	未使用パラメーター。
por|nopor	移送不能な言語構造体。
ppc|noppc	プリプロセッサーを使用した場合に起こりうる問題。
ppt|noppt	プリプロセッサー・アクションのトレース。
pro|nopro	欠落関数プロトタイプ。
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	データまたは精度の、起こりうる切り捨てまたは欠落。
tru|notru	コンパイラーで切り捨てられた変数名。
trx|notrx	16 進浮動小数点定数の丸め。
uni|nouni	初期化されていない変数。
upg|noupg	現在のコンパイラー・リリースの新しい振る舞いを、前のリリースと対比して示すメッセージを生成する。
use|nouse	未使用の自動変数および静的変数。
vft|novft	C++ プログラムでの仮想関数テーブルの生成。
zea|nozea	ゼロ範囲の配列。

例

myprogram.C をコンパイルして、変換および到達しないステートメントを除くすべての項目について通知メッセージを生成させるには、以下を入力します。

```
xlc++ myprogram.C -qinfo=all -qinfo=nocnv:norea
```

関連参照

- 126 ページの『haltonmsg』
- 252 ページの『suppress』

initauto

C C++

目的

自動変数を 2 桁の 16 進バイト値 *hex_value* に初期化します。

構文

330 ページの『#pragma options』も参照してください。

注

このオプションは、追加のコードを生成して、自動変数値を初期化します。これによって、プログラムの実行時のパフォーマンスが低下するので、デバッグ用にのみ使用してください。

-qinitauto の初期値にはデフォルト設定がありません。明示的な値を設定しなければなりません (**-qinitauto=FA** など)。

例

myprogram.c をコンパイルして、自動変数を 16 進値 FF (10 進値 255) に初期化させるには、以下を入力します。

```
xlc myprogram.c -qinitauto=FF
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

inlglue

C C++

目的

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を行うため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って、高速な外部結合を生成する。

構文

330 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 はリンカーによって生成され、 2 つの外部関数の間で制御を渡す目的で使用したり、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。外部関数の呼び出しの場合、**-qprocimported** オプションなどを使用することによって、関数がインポートされることを指定してください。

選択したアーキテクチャーでは、グルー・コードのインライン化は、ハードウェア・パフォーマンス・チューニング・オプションを選択することによって自動的に行われるようになります。これらのアーキテクチャーのいずれかを使用するシステムで、**-q64** および **-qtune=pwr4**、**-qtune=pwr5**、**-qtune=ppc970**、または **-qtune=auto** を指定すると、**-qinlglue** が自動的に使用可能になります。

グルー・コードをインライン化することによって、コードのサイズが増加する場合があります。オプション **-qcompact** を使用するとコードのサイズが削減されますが、他にどのようなオプションが指定されている場合であっても、 **-qcompact** は **-qinlglue** をオーバーライドするので、ご注意ください。

関連参照

- 330 ページの『#pragma options』

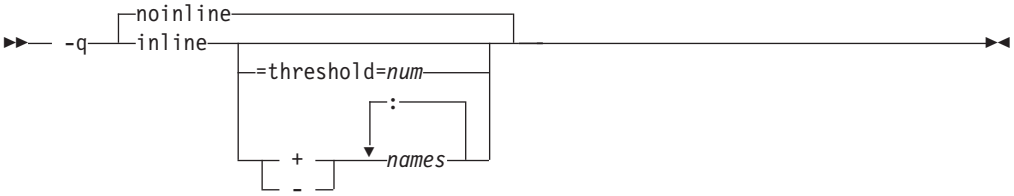
inline

C C++

目的

これらの関数の呼び出しを生成する代わりに、関数のインラインを試みます。インラインは可能であれば実行されますが、実行する最適化によってはインライン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** オプションに対するサブオプションのすべての他の設定に従って、実行可能なソース・ステートメントが 20 個以下の適切な関数をすべてインラインしようとします。 **-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されます。

-qinline=threshold=*num*
: インラインする関数に対してサイズの制限を設定します。実行可能ステートメントの数は、関数をインラインする場合は、*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。 threshold の値に **0** を指定すると、**__inline**、**_Inline**、または **_inline** キーワードでマークされた関数以外の関数はインラインされません。

num 値は、論理 C ステートメントに適用されます。以下の例に示すように、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しません。 *name* の各関数名は、それぞれコロン (**:**) 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されます。

ソース・ファイルで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すべての他の適切な関数をインラインしようとします。 *name* の各関数名は、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、インラインに適格なすべての他の関数とともに、可能な場合は、**food**、**clothes**、または **vacation** という名前の関数がすべてインラインされます。

ソース・ファイルで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 threshold 値にゼロを使用して、特定の関数をインライン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され、それ以外はインラインされません。

-qnoinline

関数を一切インラインしません。 **-qnoinline** が最後に指定された場合は、関数は一切インラインされません。

C++ 以下の **-qinline** オプションが C++ 言語に適用されます。

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

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手掛かりとして扱われます。結果は、ユーザーが選択した他のオプションに応じて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場合、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みます。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比較的簡単な関数宣言のいくつかのみ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。

-g オプションを (デバッグ情報を生成させるために) 指定した場合、インライン化は影響を受けることがあります。 **-g** コンパイラー・オプションの情報を参照してください。

インラインによって必ずしも実行時間が改善されるとは限らないため、コードに対するこのオプションの効果はユーザー自身がテストしなければなりません。再帰的な関数または相互に再帰的な関数はインラインしないでください。

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++ (**inline**、**_inline**、**_Inline**、および **__inline**) の C 言語キーワードは、**-qnoinline** 以外の **-qinline** オプションをすべてオーバーライドします。コンパイラーは、その他の **-qinline** 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しようとします。

inline、_Inline、_inline、および __inline 関数指定子: コンパイラーでは、コンパイラーにインラインさせたい関数を指定するために使用できるキーワード・セットを提供します。機能的に同等なキーワードは、以下のとおりです。

- inline (C99 および C++ のみ)
- _Inline (C のみ)
- _inline (C のみ)
- __inline (C および C++)

例を以下に示します。

```
_Inline int catherine(int a);
```

関数 `catherine` をインラインすることをコンパイラーに指示します。これは、関数呼び出しではなく関数に対してコードが生成されることを示します。

データとともにインライン指定子を使用する、または main() 関数を宣言すると、エラーが生成されます。

デフォルトでは、関数のインラインはオフであり、インライン指定子で修飾した関数は、単に extern 関数として扱われます。関数のインラインをオンにするには、**-qinline** または **-Q** コンパイラー・オプションのいずれかを指定します。**-O** または **-qoptimize** コンパイラー・オプションを指定して、最適化をオンにすると、インライン化もオンになります。

再帰的関数 (その関数そのものを呼び出す関数) は、最初のオカレンスでのみインラインされます。関数内部からのその関数への呼び出しはインラインされません。

指定したサイズよりも小さい関数すべてを自動的にインラインするには、 **-qinline** または **-Q** コンパイラー・オプションを指定することもできます。しかし、最高のパフォーマンスを得るには、自動インラインを使用するのではなく、インライン・キーワードを使ってインラインしたい関数を選ぶようにしてください。

インライン関数は、宣言と定義を同時に行うことができます。インライン・キーワードの 1 つを指定してインライン関数を宣言する場合、インライン・キーワードなしで定義することができます。以下のコードの断片では、インライン関数定義を示します。インライン・キーワードは関数定義および関数宣言の両方で指定されることに注意してください。

```
_inline int add(int i, int j) { return i + j; }

inline double fahr(double t);
```

注: インライン指定子の使用は関数の意味を変更しませんが、関数のインライン拡張は、実引き数の計算の順序を保存しないことがあります。

例

myprogram.C をコンパイルし、関数を一切インラインしないようにするには、以下を入力します。

```
xlc++ myprogram.C -O -qnoinline
```

myprogram.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 myprogram.c -O -qinline=12
```

関連参照

- 197 ページの『O、optimize』
- 221 ページの『Q』
- 120 ページの『g』

ipa

C C++

目的

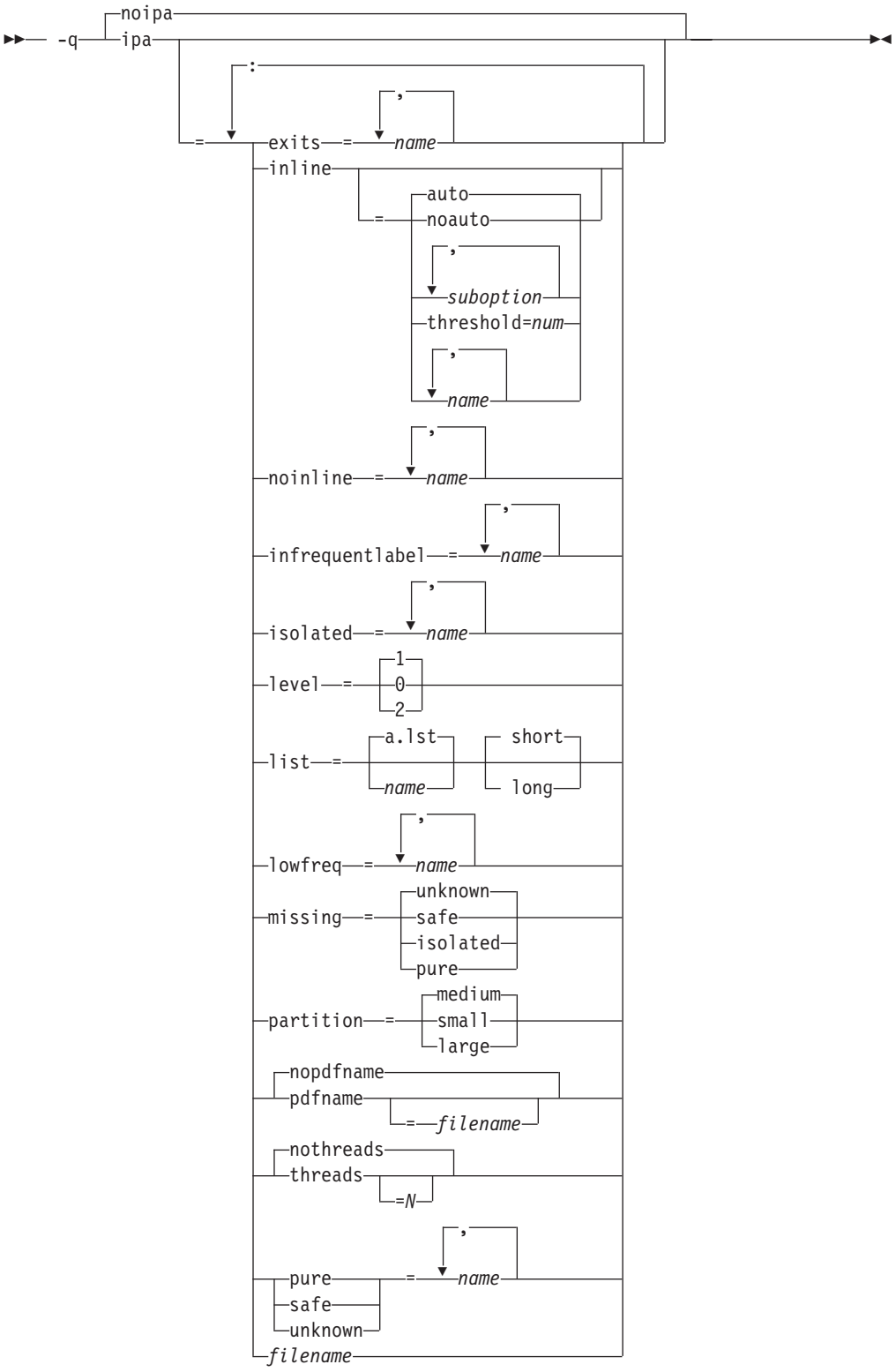
プロシージャー間分析 (IPA) と呼ぶクラスの最適化をオンにしたりカスタマイズしたりする。

コンパイル時の構文

ここで、

-qipa のコンパイル時のオプション	説明
-qipa	以下の **-qipa** *suboption* 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



ここで、

-qipa のリンク時のオプション	説明
-qnoipa	プロシージャー間分析を非活動化します。

-qipa のリンク時のオプション	説明
-qipa	以下の **-qipa** *suboption* 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。複数のサブオプションはコンマで区切ります。

リンク時のサブオプション	説明
exits=*name*{,*name*}	プログラム出口を表す関数の名前を指定します。プログラム出口は、戻ることができず、 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もできない呼び出しです。
inline=auto inline=noauto	自動インラインだけを有効にしたり無効にしたりします。コンパイラーは、ユーザーが指定した関数をインライン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*	インライン化する関数の数の上限を指定します。ただし、*num* は負でない整数です。この引き数は、**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*}	インラインしなければなら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。
infrequentlabel=*name*{,*name*}	プログラム実行中に頻繁に呼び出されないと思われるユーザー定義ラベルのリストを指定します。
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、限定された別名分析、および限定された呼び出し位置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の流れおよび別名の分析を実行します。

リンク時の サブオプション	説明
list list=[*name*] [short|long]	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) リストを要求した場合に *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あり、リストの Object File Map、Source File Map、および Global Symbols Map 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、 Object Resolution Warnings、Object Reference Map、Inliner Report、および Partition Map 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。

リンク時のサブオプション	説明
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動きを指定します。 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。
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、現在の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。

リンク時の サブオプション	説明
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。これはデフォルトです。 **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 ～ MAXINT の範囲の任意の整数値にすることができますが、効率上、ユーザーのシステム上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* として指定された関数は、すべて*分離された* および安全な 関数でなければならず、内部状態を変更したり、副次作用を持ったりしてはなりません。ここで、副次作用は、呼び出し元に対して可視のデータのいずれかを変更する可能性があることと定義します。
safe=*name*{,*name*}	**-qipa** でコンパイルされず、プログラムのほかのパートを呼び出さない 安全な 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* 関数のリストを指定します。*不明* として指定したすべての関数は、 **-qipa** でコンパイルされるプログラムの他の部分を呼び出したり、グローバル変数およびダミー引き数を変更したりすることができます。

リンク時のサブオプション	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}`
	`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。
	• exits
	• lowfreq
	• unknown
	• safe
	• isolated
	• pure

注

このオプションは、プロシージャー間分析 (IPA) と認識されているクラスの最適化をオンにしたりカスタマイズしたりします。

- 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場合があるので、 IPA の使用は、開発の最終的なパフォーマンス調整フェーズに限ってください。
- **-qipa** オプションは、アプリケーション全体、あるいはそのできるだけ多くの部分の、コンパイルおよびリンク・ステップで指定してください。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。以下のリストに、積極的な最適化を行わなくても偶然に機能するが、 IPA で判明するプログラミングの例をいくつか示します。
  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 でコンパイルしてはなりません (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場合があります。このため、これらの中間ファイルを保持するために使用する一時保管場所 (/tmp が使用されるきまりになっています) が小さすぎる場合があります。大規模なアプリケーションをコンパイルしている場合は、TMPDIR 環境変数による一時記憶域の宛先変更を考慮してください。
- IPA を実行するのに十分なスワップ・スペースがあることを確認してください (大きいプログラムの場合、少なくとも 200Mb)。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、正規表現構文を使用できます。

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

正規表現を指定するための構文規則については以下で説明します。

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* ～ *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。

IPA を使用するために必要なステップは、以下のとおりです。

1. IPA 分析は、コンパイルおよび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、予備パフォーマンス分析および調整を行ってから **-qipa** オプションでコンパイルします。 **-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起こった場合は、**-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。

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、以下を入力します。

```
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および最初のコンパイル・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* および *debug_dump* が存在するとしています。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
140 ページの『inline』
178 ページの『libansi』
180 ページの『list』
207 ページの『pdf1、pdf2』
230 ページの『S』

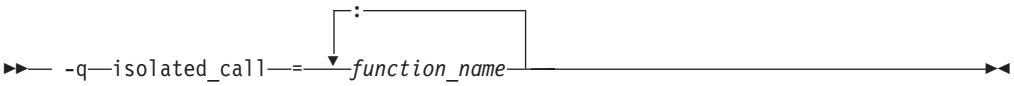
isolated_call

C C++

目的

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、実行時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。

318 ページの『#pragma isolated_call』および 330 ページの『#pragma options』も参照してください。

注

関数に分離としてマークを付けた場合は、次のことを最適化プログラムに示すことによって、最適化されたコードの実行時のパフォーマンスを向上させることができます。

- 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、ファイルの先頭で、最初の C または C++ ステートメントの前に指定しなければなりません。 **#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。

関数が、副次作用を持たないものとして誤って識別された場合、その結果得られるプログラムの振る舞いは、予期しないものになるか、または誤った結果を招く可能性があります。

例

myprogram.c をコンパイルして、関数 myfunction(int) および classfunction(double) に副次作用がないことを指定するには、以下を入力します。

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
318 ページの『#pragma isolated_call』
330 ページの『#pragma options』

keepparm

C C++

目的

アプリケーションが最適化される場合でも関数仮パラメーターがスタックに保管されているか確認します。

構文

注

関数は通常、エントリー・ポイントでスタックにその着信パラメーターを保管します。ただし、コードを使用可能な最適化オプションでコンパイルすると、コンパイラーがスタックからこれらのパラメーターを除去することで最適化に利点があると見なす場合、これらのパラメーターを除去することがあります。

-qkeepparm を指定すると、最適化時であっても確実にパラメーターをスタックに保管します。このコンパイラー・オプションは、スタックにこれらの値を保持することによって、着信パラメーターの値をデバッガーなどのツールに確実に使用可能にします。ただし、そうすることで、アプリケーション・パフォーマンスに悪影響が及ぶことがあります。

関連参照

- 197 ページの『O、optimize』

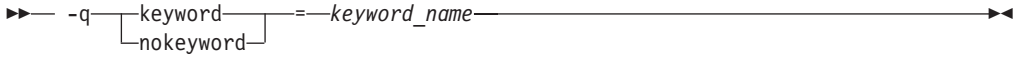
keyword

C C++

目的

このオプションは、指定された名前がプログラム・ソースに現れるたびに、キーワードとして処理するのか、ID として処理するのかを制御します。

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、また、以下の C 組み込みキーワードで使用することができます。

- asm
- __complex__ (C99)
- __imag__ (C99)
- inline
- __real__ (C99)
- restrict
- typeof

例

以下を起動することにより、 bool を復元することができます。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』

L

C C++

目的

-l*key* オプションによって指定したライブラリー・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 LIBPATH が指定したディレクトリー・パスでまずライブラリーを検索し、それから **-L** コンパイラー・オプションが指定したディレクトリー・パスで検索します。

構成ファイルとコマンド行の両方で **-L***directory* オプションを指定した場合は、構成ファイルで指定し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

myprogram.c をコンパイルして、ライブラリー **libspfiles.a** についてディレクトリー **/usr/tmp/old** および **-l** オプションで指定したすべてのディレクトリーを検索させるには、次のように入力します。

```
xlc myprogram.c -lspfiles -L/usr/tmp/old
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
159 ページの『l』
413 ページの『付録 C. XL C/C++ 内のライブラリー』

l

C C++

目的

ダイナミック・リンクの場合は指定したライブラリー・ファイル lib*key*.so を検索してから lib*key*.a を検索し、静的リンクの場合は単に lib*key*.a 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、いくつかのコンパイラー・ランタイム・ライブラリーのみが検索されます。デフォルトの構成ファイルは、**-l** コンパイラー・オプションで検索するためにデフォルトのライブラリー名を指定し、**-L** コンパイラー・オプションでデフォルトのライブラリー用検索パスを指定します。

注

また、デフォルトの検索パスにない追加のライブラリー用検索パス情報を提供する必要があります。検索パスは、**-L***directory* オプションで変更することができます。

C および C++ 実行時ライブラリーは自動的に追加されます。

-l オプションは、後から認識されたものが累積されます。コマンド行で後に指定された **-l** オプションは、前の **-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表示される順序で検索されるため、ライブラリーを指定する順序は、アプリケーション内のシンボル・レゾリューションに影響を与え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、/usr/mylibdir ディレクトリーにあるライブラリー mylibrary (libmylibrary.a) とリンクするには、以下を入力します。

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

関連タスク

28 ページの『構成ファイル内のコンパイラー・オプションの指定』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
67 ページの『B』
158 ページの『L』
413 ページの『付録 C. XL C/C++ 内のライブラリー』

langlvl

C C++

目的

コンパイル用の言語レベルと言語オプションを選択します。

構文

ここで、*suboption* については、以下の『**注**』セクションで説明します。

320 ページの『#pragma langlvl』および 330 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、コンパイラーの呼び出しに使用するコマンドに応じてそれぞれ異なります。

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

注

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

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

pragma ディレクティブは、ソース・コードのどの非コメント行よりも前に指定しなければなりません。

C C プログラムの場合、以下の **-qlanglvl** サブオプションを *suboption* に使用できます。

`classic`	非 stdc89 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適合させます。
`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 言語定義に適合するコンパイル。これは現在、SAA C レベル 2 です。
`saal2`	コンパイルは、SAA C レベル 2 CPI 言語定義に準拠します。ただし、いくつかの例外があります。
`stdc89`	ANSI C89 標準に適合するコンパイルで、ISO C90 としても知られています。

stdc99　　ISO C99 標準に適合するコンパイル。

extc89　　コンパイルは、ANSI C89 標準に適合しており、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 標準に適合しており、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、北アメリカおよび西欧のすべての言語を含む、幅広い範囲の言語で使用される文字の全セット、数字、およびその他の文字が含まれています。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。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。

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++ 規格に基づいて行われますが、拡張言語フィーチャーを適合させるために若干の相違があります。

[no]anonstruct

このサブオプションは、匿名の構造体および匿名のクラスが 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、コンパイラーは匿名の構造体を許可します。これは C++ 標準を拡張したもので、Microsoft® Visual C++ で提供されている C++ コンパイラーと互換性のある振る舞いをします。

匿名の構造体は、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、コードが匿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、匿名の構造体にエラーのフラグが付けられます。標準 C++ に適合している場合は、**noanonstruct** を指定してください。

[no]anonunion

このサブオプションは、どのメンバーが匿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、匿名の共用体は、標準 C++ が匿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、struct、typedef、および enumeration などの非データ・メンバーは許可されます。

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

デフォルトでは、コンパイラーは匿名の共用体の非データ・メンバーを許可します。これは、標準 C++ を拡張したものであり、VisualAge® C++ の以前のバージョンや以前に使用されていた製品、および Microsoft Visual C++ で提供されている C++ コンパイラーと互換性のある振る舞いを与えます。

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++ に適合している場合は、**noanonunion** を設定してください。

[no]ansifor

このサブオプションは、 C++ 標準で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。エラーを訂正するには、i をループの外で宣言するか、あるいは ansiForStatementScopes を no に設定します。

noansifor を設定して、古い言語の動きを許可してください。以前のバージョンの VisualAge C++ および以前に使用されていた製品で提供されたコンパイラー、および Microsoft Visual C++ などのその他の製品を使用して開発されたコードの場合、これを行う必要が生ずることがあります。

[no]ansisinit

このオプションは g++ -fuse-cxa-atexit と同じように働き、完全に静的デストラクターの標準に準拠した処理に必要です。

[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指示します。 **__func__** ID は、以下のような暗黙宣言があ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 が表示さ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__);
}
```

実行時に、以下を出力します。

```
__func__ appears in this_function
```

-qlanglvl=extended が有効な場合、**c99__func__** サブオプションはデフォルトで使用可能です。 **-qlanglvl=c99__func__** を指定することによってすべての言語レベルについて使用可能にし、 **-qlanglvl=noc99__func__** を指定することによってすべての言語レベルについて使用不可にできます。

c99__func__ が有効な場合、__C99__FUNC__ マクロは 1 に定義され、その他の場合は定義されません。

[no]c99complex

このサブオプションは、 C99 複合データ型および関連キーワードを認識するようにコンパイラーに指示します。

注: 複合データ型に対するサポートは、各種の C++ コンパイラー間でそれぞれ異なるため、移植性の問題が起きる場合があります。このコンパイラー・オプションを **-qinfo=por** と共に指定すると、コンパイラーは移植性に関する警告メッセージを出します。

[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指示します。

[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 16 進浮動小数点定数のサポートを使用可能にします。このサブオプションは、**-qlanglvl=extended** のためにデフォルトでオンです。有効な場合は、コンパイラーは macro __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 C++ アプリケーションで C99 型可変長配列の使用をサポートします。 macro __C99_VARIABLE_LENGTH_ARRAY は値 1 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するために割り振られるストレージは、常駐する関数が実行を完了するまでリリースされません。

[no]dependentbaselookup
デフォルトは **-qlanglvl=dependentbaselookup** です。

このサブオプションを使用すると、C++ Standard の TC1 の Issue 213 に準拠したコンパイルを指定することができます。

デフォルト設定では、従属型のテンプレート基底クラスの名前検索に関して、以前の XL C++ コンパイラーの振る舞いを保ちます。つまり、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た名前や、テンプレートの囲みスコープ内からのすべての名前を隠します。

TC1 との整合性のために、**-qlanglvl=nodependentbaselookup** を指定します。

[no]gnu_assert
以下の GNU C システム識別アサーション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 複合データ型および関連キーワードを認識するようにコンパイラーに指示します。
注: 複合データ型に対するサポートは、各種の C++ コンパイラー間でそれぞれ異なるため、移植性の問題が起きる場合があります。このコンパイラー・オプションを **-qinfo=por** と共に指定すると、コンパイラーは移植性に関する警告メッセージを出します。

[no]gnu_computedgoto
計算済み goto の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** に使用可能で、macro __IBM_COMPUTED_GOTO を定義します。

[no]gnu_explicitregvar
変数の明示的レジスターの指定をコンパイラーが受け入れて無視するかどうかを制御するための GNU C 移植性オプションです。

[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、**拡張**言語レベルへコンパイルするときは **gnu_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。このサブオプションは **-qlanglvl=extended** のためにデフォルトでオンであり、macro __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

メンバー・リストのテンプレート名を typedef として再使用可能にする GNU C 移植性オプション。

[no]gnu_suffixij

GCC スタイル複合メンバー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_suffixij** が指定される場合、複素数はサフィックス i/I または j/J で終わることができます。

[no]gnu_varargmacros

このオプションは **-qlanglvl=varargmacros** と同様です。主な違いは以下のとおりです。

- オプションの可変引き数 ID が省略符号の前に置かれ、macro __VA_ARGS__ の代わりに ID を使用可能にする。空白が ID と省略符号の間に表示されることがあります。
- 可変引き数を省略できる。
- トークン貼り付け演算子 (**##**) がコンマと可変引き数の間に表示される場合、可変引き数が提供されていないと、プリプロセッサーはぶら下がりコンマ (**,**) 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");
```

以下のようにプリプロセスします。

```
printf("Hello %s¥n", "Chris");
```

例 2 - 可変引き数の省略:

```
#define debug(format, args...) printf(format, args)

debug("Hello¥n");
```

以下のようにプリプロセスし、ぶら下がりコンマを残します。

```
printf("Hello¥n",);
```

例 3 - トークン貼り付け演算子を使用して、可変引き数が削除されたときにぶら下がりコンマを除去する:

```
#define debug(format, args...) printf(format, ## args)

debug("Hello¥n");
```

以下のようにプリプロセスします。

```
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、一般的に使用されているが、C++ Standard には準拠していない、いくつかの形式を受け入れることができます。

デフォルトで、コンパイラーはこれらの形式を許可します。これは、標準 C++ を拡張したものであり、以前のバージョンの VisualAge C++ やそれの前に使用されていた製品、および Microsoft Visual C++ で提供されている C++ コンパイラーと互換性のある振る舞いが行われるようにします。

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++ 標準に適合している場合は、**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、または部分的に指定されている型を int の暗黙的な指定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
(eg、casts、dynamic_cast、new)、および変換関数の型が含まれます。

デフォルトでは、コンパイラーは **-qlanglvl=implicitint** を設定します。これは、C++ 標準を拡張したものであり、以前のバージョンの VisualAge C++ および以前に使用されていた製品、および Microsoft Visual C++ で提供された C++ コンパイラーと互換性のある振る舞いが行われるようにします。

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++ に適合している場合は、**-qlanglvl=noimplicitint** を設定してください。例えば、上記の関数宣言は、以下のように変更されなければなりません。

```
int MyFunction()
{
  return 0;
}
```

[no]offsetnonpod

このサブオプションは、offsetof マクロをデータのみではないクラスへ適用できるかどうかを制御します。 C++ のプログラマーは、データのみのクラスを “プレーンな古いデータ” (POD) のクラスと普段よく呼んでいます。

デフォルトでは、コンパイラーは offsetof を non-POD クラスで使用することを許可します。これは、C++ 標準を拡張したものであり、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++ で提供された C++ コンパイラーと互換性のある振る舞いが行われるようにします。

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++ に適合している場合は、 **-qlanglvl=nooffsetnonpod** を設定してください。

コードが以下のいずれか 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 digraph 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。

デフォルトでは、コンパイラーは、C++ 標準で指定された digraph のみをサポートします。

以下の digraph 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。

Digraph	**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)

以前のバージョンの VisualAge C++ および以前に使用されていた製品でサポートされる標準 C++ および拡張 C++ 言語レベルとの互換性を維持するには、**-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。

デフォルトでは、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++ 標準を拡張したものであり、以前のバージョンの VisualAge C++ および以前に使用されていた製品、および Microsoft Visual C++ で提供された C++ コンパイラーと互換性のある振る舞いが行われるようにします。

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 specified に設定されている場合に有効です。

```
friend IFont;
```

標準 C++ に適合している場合は、**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。

デフォルトでは、コンパイラーはアクセス検査を抑制します。これは、C++ 標準を拡張したものであり、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++ で提供された C++ コンパイラーと互換性のある振る舞いが行われるようにします。

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不可にしない限り、警告が出ます。

標準 C++ に適合している場合は、**-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では、バージョン 5.0 以前のバージョンのコンパイラー (xlC) に実装されている位置合わせ規則を指定します。これらの以前のバージョンの 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。 VisualAge C++ 4.0 以降のデフォルトでは、これらの位置合わせ規則は無視されません。例えば、次のテンプレートでは、A<char>::B のサイズは **-qlanglvl=nooldtmplalign** の場合は 5、 **-qlanglvl=oldtmplalign** の場合は 8 となります。

```
template <class T>
struct A {
#pragma options align=packed
 struct B {
  T m;
  int m2;
 };
#pragma options align=reset
};
```

[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。

デフォルトでは、コンパイラーはこれらの古い特殊化を許可します (**-qlanglvl=nooldtmplspec**)。これは、標準 C++ を拡張したものであり、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++ で提供された C++ コンパイラーと互換性のある振る舞いが行われるようにします。

-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、型 char 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++ に適合している場合は、**-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、以下のように変更されなければなりません。

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

[no]redefmac

前の #undef または undefine() ステートメントなしでマクロを再定義できるかどうかを指定します。

[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++ 標準を拡張したものであり、Microsoft Visual C++ との互換性が維持されるようにします。以下の enum 宣言は、この拡張子を使用します。

```
enum grain { wheat, barley, rye,, };
```

標準 C++ に適合している場合、または VisualAge C++ の以前のバージョンおよび以前に使用されていた製品でサポートされている **stdc89** 言語レベルに適合している場合は、**-qlanglvl=notrailenum** を設定します。

[no]typedefclass

このサブオプションは、以前のバージョンの VisualAge C++ および以前に使用されていた製品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、北アメリカおよび西欧のすべての言語を含む、幅広い範囲の言語で使用される文字の全セット、数字、およびその他の文字が含まれています。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 が設定されている場合は、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。

[no]varargmacros

この C99 フィーチャーは、C++ アプリケーションの関数のようなマクロにある可変引き数リストの使用を許可します。構文は、可変引き数関数に似ており、printf のマスキング・マクロとして使用することができます。

例を以下に示します。

```
#define debug(format, ...) printf(format, __VA_ARGS__)

debug("Hello %s¥n", "Chris");
```

以下のようにプリプロセスします。

```
printf("Hello %s¥n", "Chris");
```

置換リストのトークン __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き数を示します。

varargmacros を指定すると、macro __C99_MACRO_WITH_VA_ARGS を値 1 に定義します。

[no]zeroextarray

このサブオプションは、クラス定義の最後の非静的データ・メンバーとして、範囲がゼロの配列を使用でき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++ 標準を拡張したものであり、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++ と適合している場合、または以前のバージョンの VisualAge C++ および以前に使用されていた製品でサポートされる ANSI 言語レベルに適合している場合、 **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。これは、従来は、旧式の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で意図的に行われていました。これは、従来のインプリメンテーションでは、プリプロセッサーが別個のプログラムであり、その出力がコンパイラーに渡されていたためです。

同様の理由から、コメントによってのみ分けられたトークンが、 1 つのトークンを形成するように結合されることもあります。ここでは、 **classic** モードでコンパイルされたプログラムのトークンを行う方法の要約を示します。

1. ソース・ファイルの該当ポイントで、次のトークンが、トークンを形成することのできる可能性のある文字の最長シーケンスです。例えば、 `i ++ + ++ j` は正しいプログラムになりますが、 `i+++++j` は `i ++ ++ + j` としてトークン化されます。
2. 形成されたトークンが ID およびマクロ名である場合は、そのマクロは、その **#define** ディレクティブで指定されたトークンのテキストで置き換えられます。各パラメーターは、対応する引き数のテキストで置き換えられます。コメントは、引き数とマクロ・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"
```

上記の構成体は、先頭が FALSE ブロック内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、および以下の例の (の場合にのみ、¥ 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であるかどうかに適用されます。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。ここにリストしていないディレクティブは、どちらのモードでも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
: 余計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
: 余計なトークンがあると、診断メッセージは生成されません。

#include
: < と > は別のトークンです。ヘッダーは、< と > のつづりと、これらの間のトークンを結合することにより、形成されます。そのため、/* と // はコメントとして認識されて (常に除去され)、" および ' が < および > 内のリテラルを開始します。 (C プログラムでは、**-qcpluscmt** を指定すると、C++ 形式のコメント // が認識されます。)

#line
: 行番号の一部ではないトークンすべてのつづりは、新しい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、ID なしで構成されるか、またはコンマで区切られた任意の数の 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。競合する場合は、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
: 余計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

- マクロ起動に関する引き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。
- 関数類似マクロのマクロ名の後に (トークンがある場合、これは引き数が少なすぎる (上記と同様)) 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き換えられます。
- 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。

関連参照

- 69 ページの『bitfields』
- 76 ページの『chars』
- 95 ページの『digraph』
- 109 ページの『flag』
- 134 ページの『info』
- 140 ページの『inline』
- 185 ページの『M』
- 227 ページの『ro』
- 252 ページの『suppress』
- 320 ページの『#pragma langlvl』
- 330 ページの『#pragma options』
- 「*XL C/C++ ランゲージ・リファレンス*」の『IBM C 言語拡張機能』および『IBM C++ 言語拡張機能』のセクションも参照してください。

lib

C C++

目的

リンク時に標準システム・ライブラリーを使用するようにコンパイラーに指示する。

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。コマンド行で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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
88 ページの『crt』

libansi

C C++

目的

ANSI C ライブラリー関数の名前が付いたすべての関数が実際はシステム関数であると見なす。

構文

330 ページの『#pragma options』も参照してください。

注

これによって、最適化プログラムは、指定された関数について副次作用があるかどうかなどの動きを認識するため、より優れたコードを生成することができます。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

linedebug

C C++

目的

デバッガー用に行番号およびソース・ファイル名の情報を生成する。

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のサイズは、 **-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・コードのステップを実行することはできますが、変数情報を表示させたり照会したり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!** (関数をインラインにしない) 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。コマンド行で **-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、デバッガーでステップ実行できるようにするには、次のように入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』
197 ページの『O、optimize』
221 ページの『Q』
330 ページの『#pragma options』

list

C C++

目的

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。

構文

330 ページの『#pragma options』も参照してください。

注

-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・リストを生成させるには、以下を入力します。

```
xlc++ myprogram.C -qlist
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
215 ページの『print』
330 ページの『#pragma options』

listopt

C C++

目的

コンパイラー呼び出し時に有効なすべてのオプションを示すコンパイラー・リストを作成する。

構文

注

リストには、コンパイラーのデフォルト、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た有効なオプションが示されます。プログラム・ソースにある **#pragma** ステートメントによるオプション設定は、コンパイラー・リストには示されません。

-qnoprint 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、有効なオプションをすべて示すコンパイラー・リストを作成させるには、以下を入力します。

```
xlc++ myprogram.C -qlistopt
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
215 ページの『print』
32 ページの『矛盾するコンパイラー・オプションの解決』

longlit

C C++

目的

サフィックスをはずしたリテラルを long 型に 64 ビット・モードで作成する。

構文

注

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
サフィックスがない 10 進数	signed int signed long unsigned long	signed long unsigned long
サフィックスをはずした 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

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
サフィックスをはずした場合	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 進または 16 進定数	8 進または 16 進定数への -qlonglit の効果
サフィックスを はずした場合	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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
160 ページの『langlvl』

longlong

C C++

目的

プログラムにおいて **long long** 整数型を許可する。

構文

デフォルト

xlc、 **xlC**、および **cc** でのデフォルトは **-qlonglong** であり、 **_LONG_LONG** を定義します (**long long** 型がプログラムで機能するようになります)。 **c89** でのデフォルトは、**-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 標準へ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 のサポートとはわずかに異なります。

例

1. myprogram.c をコンパイルして **long long int** を許可しないようにするには、次のように入力します。

```
xlc myprogram.c -qnolonglong
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

M

C C++

目的

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

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** ファイルを編集しなければ、**make** コマンドで使用することはできません。このコマンドに関する詳細については、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。

出力ファイルには、入力ファイルに対する行、および各インクルード・ファイルに対する項目が入ります。この一般形式は次のとおりです。

```
file_name.o:file_name.c
file_name.o:include_file_name
```

インクルード・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、『*相対パス名を使用したインクルード・ファイルのディレクトリー検索順序*』で説明しています。インクルード・ファイルが見つからない場合、**.d** ファイルには追加されません。

include 文がないファイルについては、入力ファイル名のみをリストする 1 行を含む出力ファイルが生成されます。

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。 サフィックスとして **.d** 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。

.c、**.C**、**.cpp**、または **.i** サフィックスが付いた入力ファイルごとに、**.d** ファイルが作成されます。また、**-+** コンパイラー・オプションを有効にした状態で、C++ プログラムをコンパイルすると、どのファイル・サフィックスも受け入れられ、 **.d** ファイルが生成されます。そうでない場合は、出力の **.d** ファイルは、他のファイルについては一切作成されません。

例えば、以下のコマンドでは、

```
xlc -M conversion.c filter.c /lib/libm.a
```

conversion.d および **filter.d** の 2 つの出力ファイル、および実行可能ファイルが作成されます。 **.d** ファイルは、ライブラリーについては作成されません。

現行ディレクトリーが書き込み可能でない場合、 **.d** ファイルは作成されません。 **-o** *file_name* を **-M** とともに指定する場合、**.d** ファイルは、 **-o***file_name* で暗黙指定されるディレクトリーに置かれます。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** に配置されます。

関連タスク

35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
50 ページの『+ (正符号)』
188 ページの『makedep』
201 ページの『o』
241 ページの『sourcetype』

ma

C

目的

呼び出し用インライン・コードを組み込み関数 **alloca** に置換します。

構文

注

#pragma alloca が未指定の場合、あるいは **-ma** を使用しない場合、 **alloca** は組み込み関数としてではなく、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 **#include <malloc.h>** を指定しなければなりません。

例

myprogram.c をコンパイルして、関数 **alloca** の呼び出しをインラインとして扱わせるには、次のように入力します。

```
xlc myprogram.c -ma
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
61 ページの『alloca』
291 ページの『#pragma alloca』

makedep

C C++

目的

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

構文

注

-qmakedep オプションは、 **-M** 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** ファイルを編集しなければ、**make** コマンドで使用することはできません。このコマンドに関する詳細については、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。

-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。サフィックスとして **.d** 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。

.c、**.C**、**.cpp**、または **.i** サフィックスが付いた入力ファイルごとに、**.d** ファイルが作成されます。また、**-+** コンパイラー・オプションを有効にした状態で、C++ プログラムをコンパイルすると、どのファイル・サフィックスも受け入れられ、 **.d** ファイルが生成されます。そうでない場合は、出力の **.d** ファイルは、他のファイルについては一切作成されません。

例えば、以下のコマンドでは、

```
xlc++ -qmakedep conversion.C filter.C /lib/libm.a
```

conversion.d および **filter.d** の 2 つの出力ファイル (および実行可能ファイル) が作成されます。 **.d** ファイルは、ライブラリーについては作成されません。

現行ディレクトリーが書き込み可能でない場合、 **.d** ファイルは作成されません。**-o** *file_name* を **-qmakedep** とともに指定する場合、**.d** ファイルは、 **-o***file_name* で暗黙指定されるディレクトリーに置かれます。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** に配置されます。

出力ファイルには、入力ファイルに対する行、および各インクルード・ファイルに対する項目が入ります。この一般形式は次のとおりです。

```
file_name.o:include_file_name
file_name.o:file_name.C
```

インクルード・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、 35 ページの『相対パス名

を使用したインクルード・ファイルのディレクトリー検索順序』で説明しています。インクルード・ファイルが見つからない場合、**.d** ファイルには追加されません。

include 文がないファイルについては、入力ファイル名のみをリストする 1 行を含む出力ファイルが生成されます。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
185 ページの『M』
201 ページの『o』
35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』

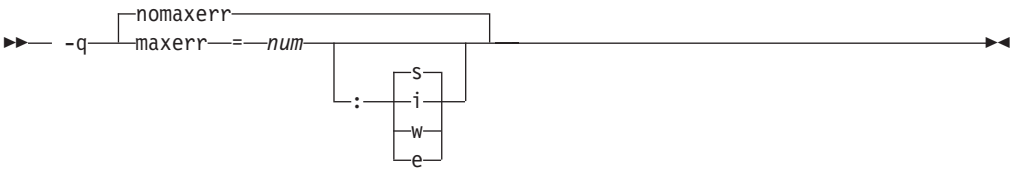
maxerr

C C++

目的

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、コンパイラーに指示する。

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、以下のいずれかを選択できます。

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** オプションの現行の値が使用されます。

-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** および **-qhalt** オプションの両方を指定した場合は、最後に指定した **-qmaxerr** または **-qhalt** オプションによって、**-qmaxerr** オプションで使用される重大度レベルが決まります。

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ときに myprogram.c のコンパイルを停止するには、以下を入力します。

   ```
   xlc myprogram.c -qmaxerr=10:w
   ```

2. 現行の **-qhalt** オプション値が **S** (重大) であるとして、 5 件の重大エラーが検出されたときに myprogram.c のコンパイルを停止するには、以下のコマンドを入力します。

```
xlc myprogram.c -qmaxerr=5
```

3. 3 件の通知メッセージが出された場合に myprogram.c のコンパイルを停止するには、以下のコマンドを入力します。

```
xlc myprogram.c -qmaxerr=3:i
```

または

```
xlc myprogram.c -qmaxerr=3 -qhalt=i
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
109 ページの『flag』
124 ページの『halt』
379 ページの『メッセージ重大度レベルとコンパイラー応答』

maxmem

C C++

目的

メモリーを大量に消費する特定の最適化のローカル・テーブルに最適化プログラムが使用するメモリーの量を制限します。メモリー・サイズの制限はキロバイトで指定されます。

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。
- 制限を増加させても、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、制限を高く設定しすぎるとページ・スペースがすべて使用されてしまう場合があります。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・テーブルに指定するメモリーが **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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

mbcs、dbcs

C C++

目的

プログラムにマルチバイト文字が含まれる場合には、 **-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

330 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・オプションを指定した場合、マルチバイト文字はコメントでも許可されます。

ソース・ファイルがマルチバイト文字リテラルを含み、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 がマルチバイト文字を含む場合、これをコンパイルするには、次のように入力します。

```
xlc myprogram.c -qmbcs
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

minimaltoc

C C++

目的

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
197 ページの『O、optimize』

mkshrobj

C C++

目的

生成されたオブジェクト・ファイルから共用オブジェクトを作成する。

構文

注

このオプションは、後述する関連オプションとともに、共用オブジェクトを作成するために 使用します。このオプションを使用する利点は、コンパイラーが tempinc ディレクトリー内のテンプレートのインスタンス生成を自動的に組み込んでコンパイルすることです。

-qmkshrobj を指定すると、**-qpic** を暗黙指定します。

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。

-qmkshrobj を使用して共用ライブラリーを作成する場合、コンパイラーおよびリンカーは適切なオプションで呼び出され、共用オブジェクトをビルドします。

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、次のように入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』
52 ページの『32、64』
101 ページの『e』
177 ページの『lib』
201 ページの『o』
206 ページの『path』
213 ページの『pic』
216 ページの『priority』
339 ページの『#pragma priority』

また、「*XL C/C++ プログラミング・ガイド*」のセクション『ライブラリーの作成』を参照してください。

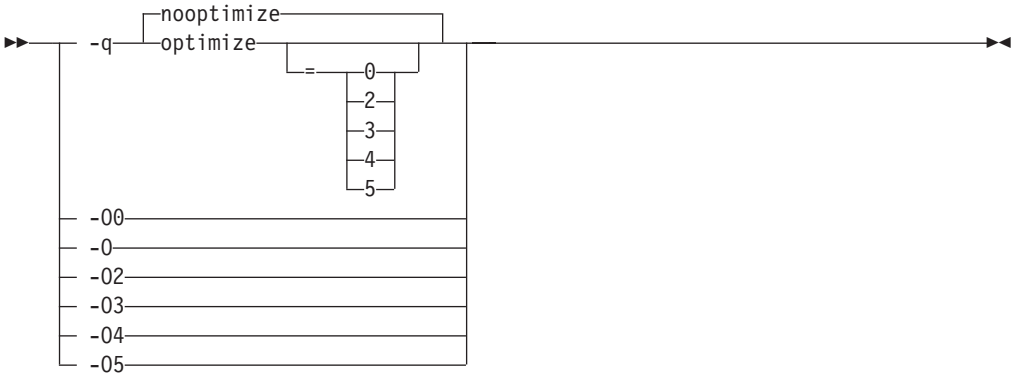
O、optimize

C C++

目的

コンパイル時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、最適化レベルを指定します。

構文



ここで、最適化の設定は以下のとおりです。

-O0 -qNOOPTimize -qOPTimize=0	定数の折り込みおよびローカルな共通部分副次式の除去などの、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	コンパイル速度と実行時のパフォーマンスに最適な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、製品のリリースによって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、**-qnostrict_induction** または **-qnostrict** によって明示的に否定されない限り、**-qstrict** と **-qstrict_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** レベルの最適化に適用しますが、時間およびメモリーが無制限です。**-O3** はまた、プログラムのセマンティクスを多少変更する可能性がある、より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を実行しません。 **-O3** を指定して **-qstrict** オプションを使用し、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してください。 **-qstrict** を **-O3** とともに指定すると、 **-O2** で実行されるすべての最適化に加えて、さらにループの最適化も行われます。**-qstrict** コンパイラー・オプションは、**-O3** オプションの後に指定しないと無視されます。

-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、以下のとおりです。
	1. 積極的なコードの移動、および例外を発生させる可能性がある計算に関する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。ロードの移動についても同じです。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、危険であると見なされる。
	• ループを通るすべてのパスで行われるわけではない。
	
	-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 規則に対する適合性が緩和されます。
	
	-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部分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、除法を逆数による乗算で置き換えることが挙げられます。

-O3 -qOPTimize=3 (*続き*)	注 • **-qfloat=rsqrt** は、デフォルトでは **-O3** で設定されます。 • **-qmaxmem=1** は、デフォルトでは **-O3** で設定され、コンパイラーは最適化を実行するときにメモリーを必要なだけ使うことができます。 • 組み込み関数は、 **-O3** では **errno** を変更しません。 • 整数除法命令の最適化は、 **-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 **flttrap** オプションと一緒に指定したときのコンパイラーの動きについては、**-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** コンパイラー・オプションへの参照は、**-O3** オプションの前後いずれに現れてもかまいません。 • **-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。
-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。 **注: -O**、**-qcache**、 **-qhot**、**-qipa**、**-qarch**、および **-qtune** オプションを後で設定すると、 **-O4** オプションで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の流れ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大限に最適化させるには、以下を入力します。

```
xlc++ myprogram.C -O3
```

関連参照

- 「*XL C/C++ スタートアップ・ガイド*」の『最適化入門』および「*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』のセクションも参照してください。

o

C C++

目的

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
   ```

   オブジェクト・ファイル **new.o** が、**test.o** の代わりに作成されます。また、以下の場合は、

   ```
   xlc test.c -o new
   ```

   new という名前のディレクトリーがなければ、オブジェクト・ファイル **new** が、**a.out** の代わりに作成されます。そうでない場合は、デフォルトのオブジェクト名 **a.out** が使用され、 **new** ディレクトリーに配置されます。

   C または C++ ソース・ファイルのサフィックス (**.C**、**.c**、 **.cpp**、または **.i**) が付いた *filespec* (myprog.c、myprog.i など) はエラーとなり、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すると、 myprogram.c をコンパイルした結果として **myaccount** というファイルを作成させるには、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、それを **myaccount** ディレクトリーに配置します。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
71 ページの『c』
99 ページの『E』
203 ページの『P』
254 ページの『syntaxonly』

P

C C++

目的

コンパイラーの呼び出し時に指定された C または C++ ソース・ファイルをプリプロセスして、プリプロセスされた出力ソース・ファイル *file_name*.i を、入力ソース・ファイル *file_name*.c、*file_name*.C、または *file_name*.cpp ごとに作成します。

構文

注

-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存します。

- コメントはすべて単一の空白に縮小される (**-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存されない。
- 関数形式のマクロに対する引き数の前後にある空白文字は保存されない。

#line ディレクティブは出されません。

-P オプションは、プリプロセスされたソース・ファイル *file_name*.**i** などを、入力として受け入れることはできません。コンパイラーがエラー・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の付いたソース・ファイルは、C ファイルと見なされてプリプロセスされ、エラー・メッセージは生成されません。

拡張モードでは、以下の場合には、プリプロセッサーは改行文字が後に続く円記号を行の継続として解釈します。

- マクロ置き換えテキスト
- マクロ引き数
- 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。

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
70 ページの『C』
71 ページの『c』
99 ページの『E』
201 ページの『o』

254 ページの『syntaxonly』

p

C C++

目的

コンパイラーが作成するオブジェクト・ファイルをプロファイル用に設定する。

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

myprogram.c をコンパイルして、オペレーティング・システムの **gprof** コマンドで使用できるようにするには、次のように入力します。

```
xlc++ myprogram.C -p
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
257 ページの『tbtable』

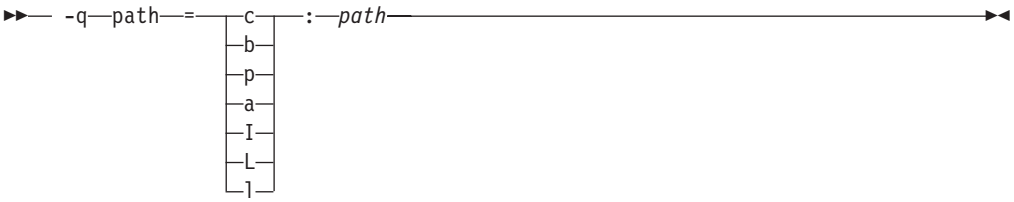
path

C C++

目的

コンパイラー・コンポーネントに代替プログラム名を構成します。このオプションで指定するプログラムとディレクトリー *path* は、正規のプログラムの代わりに使用されます。

構文



ここで、プログラム名は以下のとおりです。

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ツール (コンパイル・フェーズ)
L	プロシージャー間分析ツール (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

/lib/tmp/mine/ にある代替の **xlc++** コンパイラーを使用して myprogram.C をコンパイルするには、以下を入力します。

```
xlc++ myprogram.C -qpath=c:/lib/tmp/mine/
```

/lib/tmp/mine/ にある代替のリンケージ・エディターを使用して myprogram.C をコンパイルするには、次を入力します。

```
xlc++ myprogram.C -qpath=l:/lib/tmp/mine/
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
67 ページの『B』
108 ページの『F』
255 ページの『t』

pdf1、pdf2

C C++

目的

プロファイル指示フィードバック (PDF) を使用して最適化を調整する。ここでは、サンプル・プログラムの実行から得られた結果が使用されて、条件付き分岐の付近および頻繁に実行されるコード・セクションでの最適化が改善されます。

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、プログラム内のソース・ファイルの一部または全部をコンパイルする。少なくとも **-O2** 最適化オプションを指定する必要があり、また同様に、少なくとも有効な **-O2** でリンクする必要もあります。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。

   大規模なアプリケーションでは、最適化から最も利益を得ることのできる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: 終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。

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。そうでない場合、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。

このオプションではアプリケーション全体を 2 回コンパイルする必要があるため、他のデバッグおよび調整が終了した後の、アプリケーションを実動に供する前の最後のステップの 1 つとして使用するようにしてください。

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させるには、メイン・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。そうでない場合、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 ファイルと、別のバージョンのコンパイラーによって作成された PDF ファイルとを混合しないでください。
- **-qipa** が直接的にも別のオプションを介しても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** でプログラムをコンパイルすると、実行時にプロファイル情報が生成され、パフォーマンス上のオーバーヘッドをいくらか伴うので注意してください。このオーバーヘッドは、**-qpdf2** で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・プログラムは、**/usr/xlopt/bin** に存在し、**PDFDIR** ディレクトリーの管理に使用できます。

cleanpdf `cleanpdf [pathname]`

pathname ディレクトリーからプロファイル情報をすべて除去します。 *pathname* を指定しない場合は **PDFDIR** ディレクトリーから、 **PDFDIR** を設定し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から PDF 処理を再度行う場合は、プロファイル情報を除去すると実行時のオーバーヘッドが減少します。

特定のアプリケーション用 PDF 処理を終了した場合にのみ、**cleanpdf** を実行します。そうでない場合、そのアプリケーションで PDF の使用を再開したい場合は、**-qpdf1** で再度すべてのファイルを再コンパイルする必要があります。

mergepdf mergepdf [-r *scaling*] input {[-r *scaling*] input} ... —o *output* [-n] [-v]

2 つ以上の PDF レコードを単一 PDF 出力レコードにマージします。

-r *scaling* PDF レコード・ファイルのスケーリング率を指定します。この値は、ゼロ以上でなければならず、整数または浮動小数点値のいずれかにできます。指定されていない場合は、比率は 1.0 と見なします。

record PDF 入力レコード・ファイル、または PDF レコード・ファイルを含む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、またはマージされる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 指定されると、PDF レコード・ファイルは正規化されません。指定されない場合は、 **mergepdf** が、ユーザー定義の位取り係数を適用する前に、内部で計算した比率に基づいてレコードを正規化します。

-v 冗長モードを指定し、内部およびユーザー指定のスケーリング率をスクリーンに表示します。

resetpdf resetpdf [*pathname*]

上記の cleanpdf [*pathname*] と同様。

showpdf showpdf

プログラムの実行中に実行されたすべてのプロシージャーの呼び出しおよびブロック数を表示します。このコマンドを使用するには、最初にコマンド行で **-qpdf1** および **-qshowpdf** コンパイラー・オプションを両方とも指定して、使用している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.  */
```

具体的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 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++ file1.o file2.o file3.o file4.o
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
144 ページの『ipa』
197 ページの『O、optimize』
234 ページの『showpdf』

pg

C C++

目的

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** コマンドで使用するために 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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
257 ページの『tbtable』

phsinfo

C C++

目的

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、各フェーズごとに *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た 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 をコンパイルして、コンパイルの各フェーズでかかった時間を報告させるには、以下のように入力します。

```
xlc++ myprogram.C -qphsinfo
```

出力は以下のようになります。

```
Front End - Phase Ends;   0.004/  0.005
W-TRANS   - Phase Ends;   0.010/  0.010
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
```

-O4 で同じプログラムをコンパイルすると、次のようになります。

```
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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

pic

C C++

目的

共用ライブラリーでの使用に適した位置独立コードを生成するようにコンパイラーに指示する。

構文

ここで、

nopic　位置独立コードを生成しないようにコンパイラーに指示します。

pic　位置独立コードを生成するようにコンパイラーに指示します。

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指示します。

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た場合、**-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、次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 および **-soname** オプションについての詳細は、ご使用のオペレーティング・システムの資料の **ld** コマンドを参照してください。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
52 ページの『32、64』
196 ページの『mkshrobj』

prefetch

C C++

目的

コンパイルされたコード内で、dcbt および dcbz などのプリフェッチ命令の生成を使用可能にします。

構文

注

デフォルトでは、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ことがあります。 **-qnoprefetch** オプションを使用すると、このフィーチャー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ことを防ぎません。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

print

C C++

目的

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

いくつかのファイルに **#pragma options source** および類似したディレクティブがある場合でも myprogram.C をコンパイルしてすべてのリストを抑制するには、以下を入力します。

```
xlc myprogram.c -qnoprint
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
66 ページの『attr』
180 ページの『list』
181 ページの『listopt』
240 ページの『source』
285 ページの『xref』

priority

C++

目的

静的オブジェクトを初期化する場合の優先順位を指定します。

構文

339 ページの『#pragma priority』および 330 ページの『#pragma options』も参照してください。

注

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 (最も高い優先順位) から 65535 (最も低い優先順位) までを指定できます。

指定しない場合、デフォルト優先順位は 65535 になります。

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』
339 ページの『#pragma priority』

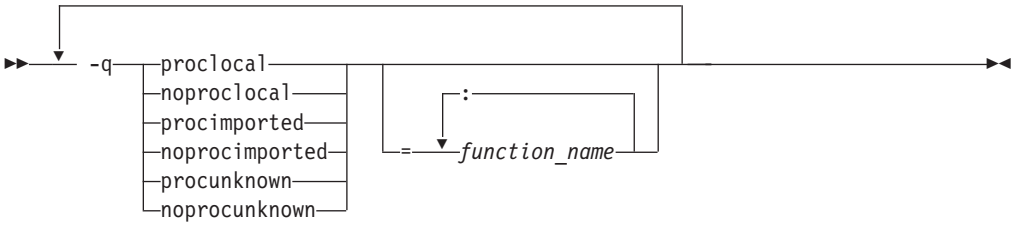
proclocal、 procimported、 procunknown

C C++

目的

64 ビット・コンパイルで、関数をローカル、インポートされるもの、または不明としてマークします。

構文



330 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、定義が現行の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** であると見なします。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。

ローカルな関数
: 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、関数の*名前* は、マングルされた名前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の矛盾は、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ません。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** とともにコンパイルして、以下のようにするには、
   - 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、**インポートされるもの**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。

```
/opt/cross/bin/powerpc64-linux-ld: 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
/opt/cross/bin/powerpc64-linux-ld: 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
```

実行可能ファイルが作成されますが、動作はしません。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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

proto

C

目的

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宣言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明示します。

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き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き数の型が、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ついては、警告を出します。

例

my_c_program.c をコンパイルして全関数がプロトタイプ宣言されていることを想定させるには、以下を入力します。

```
xlc my_c_program.c -qproto
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

Q

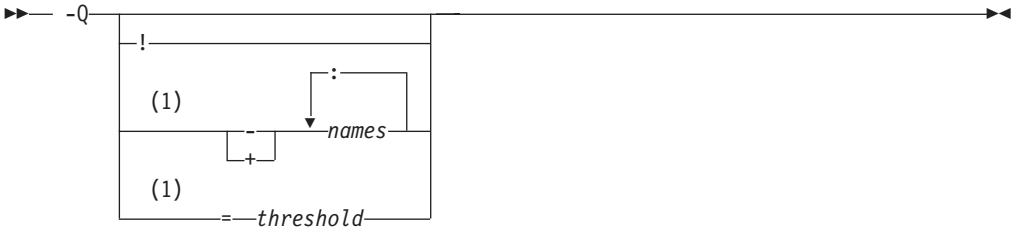
C C++

目的

C++ 言語アプリケーションでは、このオプションは関数のインラインを試行するようコンパイラーに指示します。インラインは可能であれば実行されますが、実行する最適化によってはインライン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、コンパイラーがインラインを試行する必要がある特定の関数を指定します。

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します。
-Q! コンパイラーは関数を一切インラインしません。

C C 言語では、以下の **-Q** オプションが適用されます。

-Q **-Q** オプションに対する任意のサブオプションの設定に従って、実行可能なソース・ステートメントが 20 個以下の該当する関数をすべてインラインしようとします。 **-Q** を最後に指定した場合は、すべての関数がインラインされます。

-Q! 関数を一切インラインしません。 **-Q!** を最後に指定した場合は、関数は一切インラインされません。

-Q-*names* *names* にリストされた関数をインラインしません。*names* の関数名は、それぞれコロン (**:**) 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関数が、可能であればインラインされます。

ソース・ファイルで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すべての他の適切な関数をインラインしようとします。 *names* の各関数名は、コロン (**:**) で分離しなければなりません。このオプションは、**-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、インラインに適格なすべての他の関数とともに、可能な場合は、**food**、**clothes**、または **vacation** という名前の関数がすべてインラインされます。

ソース・ファイルで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。 **-Q+***names* と一緒に threshold 値にゼロを使用して、特定の関数をインライン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され、それ以外はインラインされません。

-Q=*threshold*

インラインする関数に対してサイズの制限を設定します。実行可能ステートメントの数は、関数をインラインする場合は *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。 threshold の値に **0** を指定すると、**__inline**、**_Inline**、または **_inline** キーワードでマークされた関数以外の関数はインラインされません。

threshold の値は、論理 C ステートメントに適用されます。以下の例に示すように、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手掛かりとして扱われます。インライン化が行われるかどうかは、選択するほかのオプションにも依存することがあります。

- プログラムを最適化する場合、(**-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をインライン化しようとします。

注

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

-g オプションを (デバッグ情報を生成させるために) 指定した場合、インライン化は影響を受けることがあります。 **-g** コンパイラー・オプションの情報を参照してください。

インラインによって必ずしも実行時間が改善されるとは限らないため、コードに対するこのオプションの効果はユーザー自身がテストしなければなりません。

再帰的な関数または相互に再帰的な関数はインラインしないでください。

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**、および **__inline** 言語キーワードは、 **-Q!** 以外の **-Q** オプションをすべてオーバーライドします。コンパイラーは、その他の **-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しようとします。

最大限にインラインさせるには、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

プログラム myprogram.c をコンパイルし、関数を一切インラインしないようにするには、以下を入力します。

```
xlc myprogram.c -O -Q!
```

プログラム my_c_program.c をコンパイルして、 12 行以下の関数のインラインをコンパイラーに試行させるには、以下を入力します。

```
xlc my_c_program.c -O -Q=12
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』
140 ページの『inline』
197 ページの『O、optimize』
221 ページの『Q』
142 ページの『inline、_Inline、_inline、および __inline 関数指定子』

R

C C++

目的

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で **-R***directory* オプションを指定した場合は、実行時に、構成ファイルで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、後から認識されたものが累積されます。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とも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、次のように入力します。

```
xlc++ myprogram.C -lspfiles -R/usr/tmp/old
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
158 ページの『L』
159 ページの『l』

r

C C++

目的

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る。

構文

注

このフラグで作成されたファイルは、 **xlc++** への別の呼び出しでファイル・パラメーターとして使用されることを前提としています。

例

myprogram.c および myprog2.c を、単一のオブジェクト・ファイル **mytest.o** にコンパイルするには、以下のように入力します。

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

関連参照

41 ページの『コンパイラーのコマンド行オプション』

report

C C++

目的

プログラム・ループの並列化および/または最適化の方法を示す変換レポートを作成するようにコンパイラーに指示する。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** または **-qsmp** も有効でないと、有効ではありません。

-qreport を、**-qhot** と一緒に指定すると、コンパイラーに、疑似 C のコード・リストと、ループの変換方法の要約とを作成するように指示が出されます。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、コンパイラーに、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指示が出されます。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

myprogram.C をコンパイルして、コンパイラー・リストにループの最適化の過程を示すレポートを組み込むには、以下のように入力してください。

```
xlc++ -qhot -O3 -qreport myprogram.C
```

myprogram.C をコンパイルして、コンパイラー・リストに並列化されたループを変換する過程を示すレポートを組み込むには、以下のように入力してください。

```
xlc++ -qsmp -O3 -qreport myprogram.C
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
127 ページの『hot』
237 ページの『smp』

ro

C C++

目的

ストリング・リテラルの保管型を指定する。

構文

330 ページの『#pragma options』も参照してください。

デフォルト

cc およびその派生物を除く、すべてのコンパイラー呼び出しのデフォルトは **-qro** です。**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・プログラムで保管型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、実行時のパフォーマンスが向上し、ストレージを節約できますが、コードが読み取り専用のストリング・リテラルを変更しようとし、メモリー・エラーが生成されることがあります。

例

myprogram.c をコンパイルして保管型を **writable** にするには、以下のように入力します。

```
xlc myprogram.c -qnoro
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
228 ページの『roconst』
330 ページの『#pragma options』
347 ページの『#pragma strings』

roconst

C C++

目的

定数値の保管場所を指定する。

構文

330 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** でのデフォルトは **-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って、実行時のパフォーマンスを向上させ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 (**struct**、**union** 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または静的型
- 初期化されていない型
- **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**) の両方の保管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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
227 ページの『ro』
330 ページの『#pragma options』

rtti

C++

目的

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

実行時のパフォーマンスを最高にするには、デフォルトの **-qnortti** の設定によって 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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
102 ページの『eh』

S

C C++

目的

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 **.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、 **xlc++** コマンドで起動できます。例を以下に示します。

```
xlc++ myprogram.s
```

これによって、アセンブラーが起動され、成功した場合はローダーが起動され、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
   ```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
99 ページの『E』
120 ページの『g』
144 ページの『ipa』
201 ページの『o』
203 ページの『P』
257 ページの『tbtable』

s

C C++

目的

このオプションは、出力ファイルからのシンボル・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 コマンドの使用にも同じ影響があります。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』

saveopt

C C++

目的

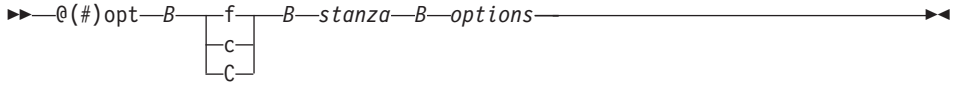
コマンド行コンパイラー・オプションをオブジェクト・ファイル内に保管する。

構文

注

このオプションにより、コンパイル中のオブジェクト・ファイルにコマンド行コンパイラー・オプションを保管できます。このオプションは、オブジェクト (**.o**) ファイルに対してコンパイルしている場合にのみ、有効です。

ストリングは、次の形式で保管されます。



ここで、

B スペースを示します。

f FORTRAN 言語コンパイルを示します。

c C 言語コンパイルを示します。

C C++ 言語コンパイルを示します。

stanza コンパイルに使用するドライバー、例えば、c89 または xlc++ を指定します。

options コマンド行で指定されるコマンド行オプションのリスト。個々のオプションをスペースで区切られています。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

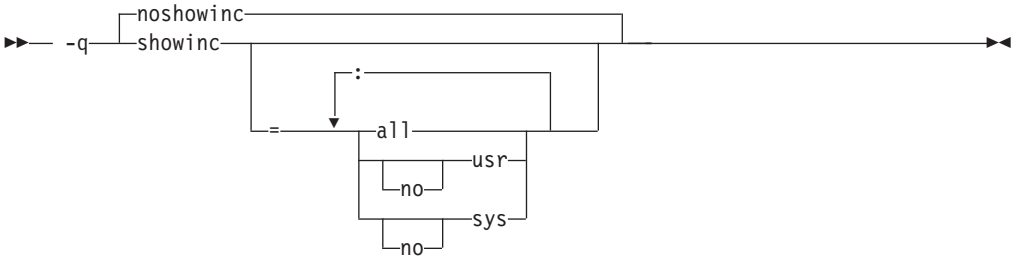
showinc

C C++

目的

-qsource と共に使用して、プログラム・ソース・リストにユーザー・ヘッダー・ファイル (" " を使用して組み込む) またはシステム・ヘッダー・ファイル (< > を使用して組み込む) を選択的に表示する。

構文



ここで、オプションは、以下のとおりです。

noshowinc	ユーザー・インクルード・ファイルとシステム・インクルード・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** の指定と同じです。
showinc	ユーザー・インクルード・ファイルとシステム・インクルード・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** の指定と同じです。
all	ユーザー・インクルード・ファイルとシステム・インクルード・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** の指定と同じです。
usr	ユーザー・インクルード・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・インクルード・ファイルをプログラム・ソース・リストに表示します。

330 ページの『#pragma options』も参照してください。

注

-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、このオプションは有効です。

例

myprogram.C をコンパイルしてすべてのインクルード・ファイルがソース・リストに出力されるようにするには、以下を入力します。

```
xlc++ myprogram.C -qsource -qshowinc
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
240 ページの『source』
330 ページの『#pragma options』

showpdf

C C++

目的

-qpdf1 および最小最適化レベル **-O** と共に使用して、追加呼び出しおよびブロック数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した場合にのみ有効です。

-qpdf1 および **-O** の最小最適化レベルと共に指定した場合、コンパイラーは、追加のプロファイル情報を、コンパイルされるアプリケーションに挿入して、アプリケーション内のすべてのプロシージャーに関する呼び出しおよびブロック数を収集します。コンパイルされたアプリケーションを実行すると、呼び出しおよびブロック数がファイル **._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、 **showpdf** ユーティリティーを使用して **._pdf** ファイルの内容を取り出すことができます。このユーティリティーは、 **-qpdf** ページに説明されています。

例

例では、プログラム・ファイル `hello.c` に対して次のソースを想定しています。

```
#include <stdio.h>

void HelloWorld()
{
 printf("Hello World");
}

main()
{
 HelloWorld();
}
```

以下を使用してソースをコンパイルします。

```
xlc -qpdf1 -O -qshowpdf hello.c
```

結果として得られる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実行して、実行可能ファイルに対する呼び出し数およびブロック数を表示します。

```
showpdf
```

showpdf ユーティリティーが、以下のようなデータを戻します。

```
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 )
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
207 ページの『pdf1、pdf2』

smallstack

C C++

目的

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指示する。

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。このオプションを使用すると、スタック・フレームのサイズを縮小して、オーバーフローを避けることができます。

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が低下する場合があります。

例

myprogram.c をコンパイルしてスタック・フレームを小さくするには、以下のように入力します。

```
xlc myprogram.c -qipa -qsmallstack
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』
144 ページの『ipa』
197 ページの『O、optimize』

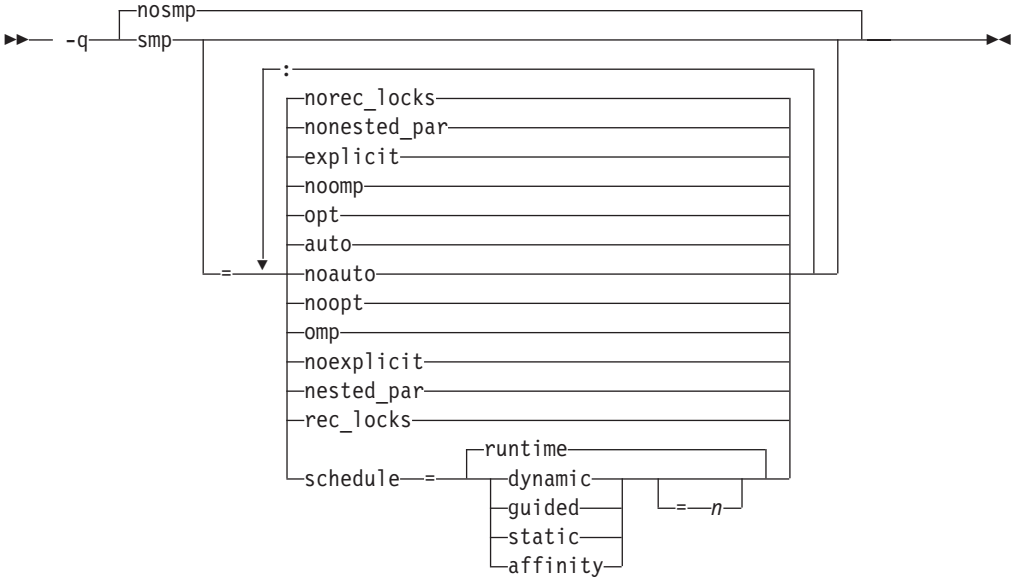
smp

C C++

目的

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。SMP または OpenMP プラグマ・ステートメントによって明示的に並列化されるプログラム・コードが最適化されます。

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は、この設定を使用してください。

omp OpenMP 2.0 標準に厳密に準拠できるようにします。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。認識されるのは、OpenMP 並列化プラグマのみです。

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シリアライズされません。 **nested_par** は、スレッドの新規のチームをネストされた並列領域用に作成させる原因とはならないため、真のネストされた並列性を提供しません。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によって相対位置変更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、**schedule=runtime** がデフォルト設定と見なされます。

注

- **-qsmp** の場合は、 **xlc_r** のようなスレッド・セーフ・コンパイラー・モードの起動でのみ使用しなければなりません。これらの起動により、**pthreads**、**xlsmp**、およびすべてのデフォルトのランタイム・ライブラリーのスレッド・セーフ・バージョンが、その結果生じる実行可能ファイルと確実にリンクされます。
- **-qnosmp** オプションのデフォルト設定では、構文検査が実行されることになっていても、並列化ディレクティブのためのコードを生成しないように指定されています。**-qignprag=omp:ibm** を使用して、並列化ディレクティブを完全に無視します。
- サブオプションを指定しない **-qsmp** の指定は、**-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、_IBMSMP プリプロセス・マクロが定義されます。

関連概念

13 ページの『プログラムの並列化』

関連タスク

37 ページの『プラグマを使用した並列処理の制御』

関連参照

41 ページの『コンパイラーのコマンド行オプション』
197 ページの『O、optimize』
264 ページの『threaded』
355 ページの『並列処理を制御するプラグマ』
383 ページの『並列処理のためのランタイム・オプション』
387 ページの『並列処理のための OpenMP ランタイム・オプション』
389 ページの『並列処理に使用する組み込み関数』

source

C C++

目的

コンパイラー・リストを作成し、ソース・コードを組み込む。

構文

330 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・プログラムにわたって使用することによって、ソースの一部を選択的に出力させることができます。 **#pragma options source** と **#pragma options nosource** の間にあるソースが出力されます。

例

myprogram.C をコンパイルして、ソースを **myprogram.C** に組み込むコンパイラー・リストを作成するには、以下を入力します。

```
xlc++ myprogram.C -qsource
```

コンパイラー・リストにプログラム・ソースの選択された部分のみ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。以下のコードによって、**#pragma options source** と **#pragma options nosource** ディレクティブの間にあるソースのみがコンパイラー・リストに入ります。

```
#pragma options source
   . . .
/* Source code to be included in the compiler listing
   is bracketed by #pragma options directives.
*/
   . . .
#pragma options nosource
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
215 ページの『print』
330 ページの『#pragma options』

sourcetype

C C++

目的

実際のソース・ファイル名サフィックスとは関係なく、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・タイプであるかのように処理することをコンパイラーに指示する。

構文

ここで、

default	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サフィックスによって暗黙指定されているものと見なします。
c	コンパイラーは、このオプションに従い、すべてのソース・ファイルを C 言語ソース・ファイルであるかのようにコンパイルします。
c++	コンパイラーは、このオプションに従い、すべてのソース・ファイルを C++ 言語ソース・ファイルであるかのようにコンパイルします。
assembler	コンパイラーは、このオプションに従い、すべてのソース・ファイルをアセンブラー言語ソース・ファイルであるかのようにコンパイルします。

注

-qsourcetype オプションは、 **-+** オプションと共に使用してはいけません。

-qsourcetype オプションはコンパイラーに対し、ファイル名サフィックスに依存するのではなく、ソース・タイプをオプションによって指定されたものと見なすように指示します。

通常、コンパイラーは、コマンド行で指定されたソース・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** サフィックスは通常 C ソース・コードを暗黙指定し、 **.C** サフィックスは通常 C++ ソース・コードを暗黙指定し、コンパイラーはそれらを以下のように処理します。

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。

これは、ファイル・システムが大/小文字を区別するかどうかにかかわらず適用されます。しかし、大/小文字を区別しないファイル・システムでは、上記の 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、まだコマンド行上のファイル名引き数の大/小文字の違いを認識し、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取り出すときに、大/小文字の区別を無視します。

例

ソース・ファイル **hello.C** を C 言語ソース・ファイルとして処理するには、以下を入力します。

```
xlc -qsourcetype=c hello.C
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
50 ページの『+ (正符号)』

spill

C C++

目的

レジスター割り振り予備域を *size* バイトに指定する。

構文

330 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には、この領域を増加させる必要がある場合があります。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。競合がある場合には、指定された最大の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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

srcmsg

C

目的

stderr ファイル内の診断メッセージに、対応するソース・コード行を追加する。

構文

330 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。時には、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

myprogram.c をコンパイルして、エラーの発生時に診断メッセージとともにソース行を表示させるには、以下を入力します。

```
xlc myprogram.c -qsrcmsg
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』

staticinline

C C++

目的

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。

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。

```
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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

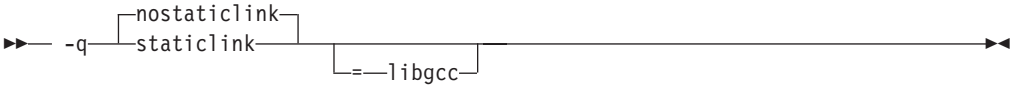
staticlink

C C++

目的

-qstaticlink コンパイラー・オプションは、共用および非共用のランタイム・ライブラリーをアプリケーションにリンクする方法を制御する。 XL オプションを使用すると、単独または組み合わせで使用される、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ことができます。

構文



ここで、

nostaticlink
: libgcc.a によって静的にリンクしないように、コンパイラーに指示します。

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のみリンクされます。

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

共用ライブラリーおよび非共用ライブラリーのための GNU サポートは、以下の表に示すオプションで制御されます。

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 オプション
-shared	共用オブジェクトを作成する。	-qmkshrobj
-static	静的オブジェクトを作成し、共用ライブラリーとのリンクを防止する。リンク先のライブラリーはすべて、静的ライブラリーでなければなりません。-shared とともに指定されたときは無視します。	-qstaticlink
-shared-libgcc	libgcc の共用バージョンを使用する。-static とともに指定されたときは無視します。	-qnostaticlink=libgcc
-static-libgcc	libgcc の静的バージョンを使用する。	-qstaticlink=libgcc

関連参照

41 ページの『コンパイラーのコマンド行オプション』

statsym

C C++

目的

永続的なストレージ・クラスを持つユーザー定義の非外部名 (初期化される静的変数または初期化されない静的変数など) を、名前リスト (オブジェクトのシンボル・テーブル) に追加します。

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

myprogram.C をコンパイルして静的シンボルがシンボル・テーブルに追加されるようにするには、以下を入力します。

```
xlc++ myprogram.C -qstatsym
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

stdinc

C C++

目的

#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るファイルに使用するディレクトリーを指定する。**-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

330 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、現行のソース・ファイルが置かれてい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、*相対パス名を使用したインクルード・ファイルのディレクトリー検索順序* で説明しています。

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

myprogram.c をコンパイルして、 **#include "myinc.h"** ディレクティブで myprogram.c に組み込まれるファイルについて、ディレクトリー **/tmp/myfiles** を検索させるには、以下のように入力します。

```
xlc myprogram.c -qnostdinc -I/tmp/myfiles
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
129 ページの『I』
130 ページの『idirfirst』
330 ページの『#pragma options』
35 ページの『相対パス名を使用したインクルード・ファイルのディレクトリー検索順序』

strict

C C++

目的

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

330 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。

- コードを移動させて、例外を起動する可能性が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 規則への適合性を緩和する。
- 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 (**-qfloat=rsqrt**) による乗算によって平方根の結果による除法が置換されるように、myprogram.C をコンパイルするには、次のように入力します。

```
xlc++ myprogram.C -O3 -qstrict -qfloat=rsqrt
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
111 ページの『float』

197 ページの『O、optimize』
330 ページの『#pragma options』

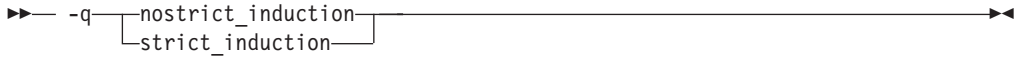
strict_induction

C C++

目的

プログラムのセマンティクスを変更する可能性が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できます。

構文



デフォルト

- 最適化レベル 2 以上では **-qnostrict_induction**
- それ以外は、**-qstrict_induction**

注

オプション **-O2** を指定すると、**-qnostrict_induction** が暗黙指定されます。両方とも指定する必要はありません。

パフォーマンスをかなり低下させる可能性があるため、一般には、オプション **-qstrict_induction** の使用はお勧めしません。

関連参照

- 197 ページの『O、optimize』

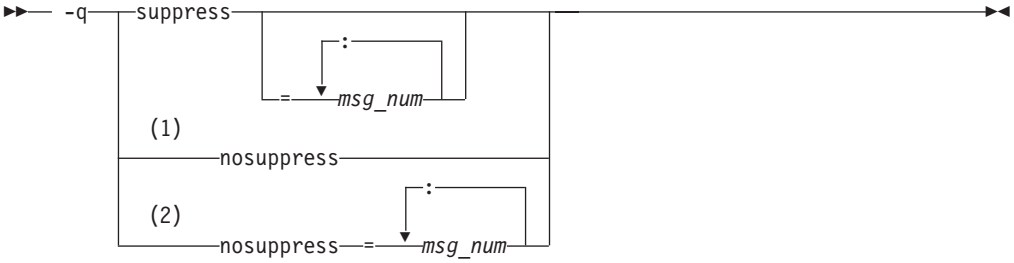
suppress

C C++

目的

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、またはリストに追加されたりするのを防ぐ。

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 メッセージを抑止するには、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の、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、または **-qhalt** コンパイラー・オプションの設定に依存するその他のメッセージを非表示にすることはできません。たとえば、**-qhalt=w** コンパイラー・オプションが設定された場合、**-qsuppress** コンパイラー・オプションによって警告メッセージを抑止することはできません。

-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
```

以下をコンパイルすることによって、このメッセージを抑止することができます。

```
xlc++ myprogram.C -qsuppress=1506-224
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
124 ページの『halt』
144 ページの『ipa』

symtab

C C++

目的

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

myprogram.c をコンパイルして静的シンボルがシンボル・テーブルに追加されるようにするには、次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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』

syntaxonly

C

目的

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。その他のすべてのファイル (リストなど) 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、以下を入力します。

```
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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
70 ページの『C』
71 ページの『c』
99 ページの『E』
201 ページの『o』
203 ページの『P』

t

C C++

目的

-B オプションで指定したプレフィックスを、指定したプログラムに追加します。

構文

プログラムは以下のとおりです。

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ツール (コンパイル・フェーズ)
L	プロシージャー間分析ツール (リンク・フェーズ)
l	リンケージ・エディター

注

このオプションは、必ず **-B** オプションと一緒に使用しなければなりません。

デフォルト

-B は指定しているものの、 *prefix* は指定し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* をまったく指定していない場合、標準プログラム名のプレフィックスは **/lib/n** です。

-B は指定しているものの、**-t***programs* は指定しない場合、デフォルトで、すべての標準プログラム名についてパス名が構成されます。

例

myprogram.c をコンパイルして、コンパイラーおよびアセンブラー・プログラム名に、 **/u/newones/compilers/** というプレフィックスを付けるには、以下のように入力します。

```
xlc myprogram.c -B/u/newones/compilers/ -tca
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
67 ページの『B』

tabsize

C C++

目的

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

tbtable

C C++

目的

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、以下のとおりです。

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。

330 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**

関連参照

41 ページの『コンパイラーのコマンド行オプション』
120 ページの『g』
197 ページの『O、optimize』
330 ページの『#pragma options』

tempinc

C++

目的

テンプレート関数およびクラス宣言用に別個の tempinc 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配置します。

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、同時に指定できません。**-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 を指定すると、コンパイラーは、**__TEMPINC__** マクロに値 1 を割り当てます。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インクルード・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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
261 ページの『templateregistry』

「*XL C/C++ プログラミング・ガイド*」の『*C++ テンプレートの使用*』のセクションも参照してください。

templaterecompile

C++

目的

-qtemplateregistry コンパイラー・オプションを使用してコンパイルされたコンパイル単位間の依存性の管理に役立つ。

構文

注

-qtemplaterecompile オプションは、 **-qtemplateregistry** 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ちます。複数のコンパイル単位が同一のテンプレート・インスタンス化を参照するプログラムを指定すると、**-qtemplateregistry** オプションは、インスタンス化を含めるための単一のコンパイル単位を指定します。このインスタンス化は、他のコンパイル単位には含まれず、オブジェクト・コードの重複が回避されます。

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・ファイルを今後参照しないなどの変更がソース・ファイルに行われたときに発生します。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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
261 ページの『templateregistry』
259 ページの『tempinc』

「*XL C/C++ プログラミング・ガイド*」の『*C++ テンプレートの使用*』のセクションも参照してください。

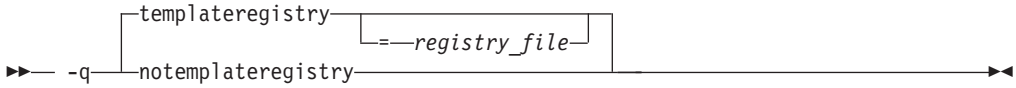
templateregistry

C++

目的

ソース内で検出されるすべてのテンプレートについて、そのレコードを保守し、テンプレートごとに、一度だけインスタンス化を行うようにする。

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、同時に指定できません。**-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。

-qtemplateregistry オプションは、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を行うようにします。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、インスタンスが重複して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・ファイルは、異なるプログラム間で共有してはいけません。ソースが同じディレクトリーにある 2 つ以上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、この状況が起こり、間違った結果を出すことになります。

例

ファイル myprogram.C をコンパイルして、テンプレート登録情報を **/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
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
260 ページの『templaterecompile』
259 ページの『tempinc』

「*XL C/C++ プログラミング・ガイド*」の『*C++ テンプレートの使用*』のセクションも参照してください。

tempmax

C++

目的

-qtempinc オプションによって各ヘッダー・ファイルに対して生成される、テンプレート・インクルード・ファイルの最大数を指定する。

構文

注

number に 1 ～ 99999 の値を指定することによって、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・インクルード・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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
259 ページの『tempinc』
311 ページの『#pragma implementation』

「*XL C/C++ プログラミング・ガイド*」の『*C++ テンプレートの使用*』のセクションも参照してください。

threaded

C C++

目的

プログラムが複数のスレッドを使用するようコンパイラーに指示します。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、コンパイルおよびリンケージ・エディターの両方の命令に適用します。

スレッド・セーフティーを保守するには、 **-qthreaded** オプションで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コンパイルされた場合でも、 **_r** コンパイラー呼び出しモードを選択して暗黙的にコンパイルされた場合でも、 **-qthreaded** オプションでリンクしなければなりません。

このオプションは、コードのスレッド・セーフの作成は行いませんが、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・セーフのまま残るようにします。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

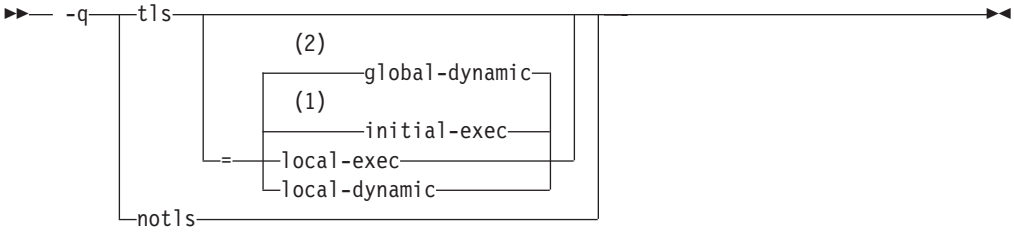
tls

C C++

目的

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。

gcc's __thread キーワードをサポートするシステムでは、**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・オプション群に追加します。

-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**。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
213 ページの『pic』

tmplparse

C++

目的

このオプションでは、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、テンプレートのインスタンス化にのみ適用するかを指定します。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、以下のとおりです。

なし	テンプレート定義を構文解析しません。これにより、以前のバージョンの VisualAge C++ および以前の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。これはデフォルトです。
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、定義の外側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、次のように構成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

関連参照

41 ページの『コンパイラーのコマンド行オプション』

「*XL C/C++ プログラミング・ガイド*」の『*C++ テンプレートの使用*』のセクションも参照してください。

tocdata

C C++

目的

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、それを使用する関数と静的にバインドされます。 **-qtocdata** によって、変数をすべてローカルであると見なすようにコンパイラーに指示します。

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。インポートされる変数は、ライブラリーの共用部分と動的にバインドされます。 **-qnotocdata** によって、変数をすべてインポートするものであると見なすようにコンパイラーに指示します。

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。

変数名をリストするオプション	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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

trigraph

C C++

目的

キーボードにない文字を表すために使用される、 3 文字表記キーの組み合わせを認識するようコンパイラーに指示します。

構文

注

3 文字表記は、すべてのキーボードで使用できるとは限らない文字を指定することができる 3 つのキー文字の組み合わせです。

3 文字表記の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 デフォルトの **-qtrigraph** 設定は、コマンド行で **-q[no]trigraph** オプションを明示的に設定することによってオーバーライドすることができます。

コマンド行で明示的に指定された **-q[no]trigraph** は、 **-q[no]trigraph** がコマンド行で指定された場所に関係なく、通常、指定された **-qlanglvl** コンパイラー・オプションに関連付けられている **-q[no]trigraph** 設定に優先します。

C++ C++ プログラムについても同じです。

例

1. 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できないようにするには、以下を入力します。

```
xlc++ myprogram.C -qnotrigraph
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
95 ページの『digraph』
160 ページの『langlvl』

tune

C C++

目的

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、以下のとおりです。

`auto` コンパイル先のプラットフォーム用に最適化されたオブジェクト・コードを生成します。

`ppc970` PowerPC 970 プロセッサー用に最適化されたオブジェクト・コードを生成します。

330 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。 **-q64** は **-qarch** のデフォルトに影響し、したがってまた **-qtune** のデフォルトにも影響することに注意してください。

- **-qarch** を指定せずに **-qtune** を指定した場合、コンパイラーは **-qarch=ppc970** を使用します。
- **-qtune** を指定せずに **-qarch** を指定した場合、コンパイラーは、指定されたアーキテクチャーに対応するデフォルトのチューニング・オプションを使用し、またリストには以下のように示されます: `TUNE=DEFAULT`。

特定の **-qarch** 設定でのデフォルトの **-qtune** 設定については、 376 ページの『コンパイラー・モードおよびプロセッサー・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* には、命令の生成対象とするアーキテクチャーを指定します。
- **-qtune=***suboption* には、コードの最適化対象であるターゲット・プラットフォームを指定します。

例

`myprogram.C` からコンパイルした実行可能プログラム testing を、PowerPC 970 ハードウェア・プラットフォーム用に最適化するように指定するには、以下のように入力します。

```
xlc++ -o testing myprogram.C
-qtune=ppc970
```

関連参照

- 30 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) コンパイル用コンパイラー・オプションの指定』
- 63 ページの『arch』
- 376 ページの『コンパイラー・モードおよびプロセッサー・アーキテクチャーの受け入れ可能な組み合わせ』

U

C C++

目的

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 を未定義にする。

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。このオプションでは、**#define** プリプロセッサー・ディレクティブによってソースで定義された名前を未定義にすることはできません。このオプションで未定義にできるのは、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・プログラムにおいて ID 名を未定義にすることもできます。

-U*name* オプションの優先順位は、 **-D***name* オプションよりも上です。

例

ご使用のオペレーティング・システムが名前 **__unix** を定義していますが、コンパイルによって定義されている名前にコード・セグメント条件を入力したくないとします。以下のように入力して、名前 **__unix** の定義が無効となるように myprogram.c をコンパイルします。

```
xlc myprogram.c  -U__unix
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
89 ページの『D』

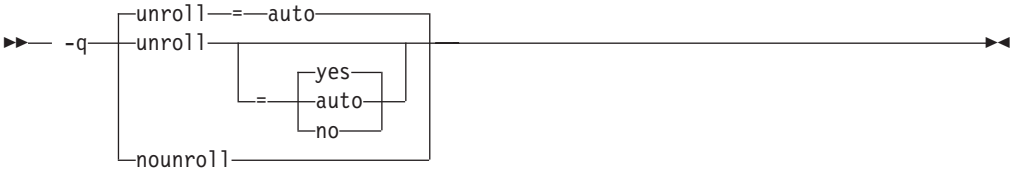
unroll

C C++

目的

プログラムの内側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向上させることができます。

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	コンパイラーにループのアンロールを提案します。
-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指示します。

348 ページの『#pragma unroll』および 330 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、コマンド行で特に明示的に指定されない限り、**-qunroll=auto** です。

サブオプションなしで **-qunroll** を指定するのは、**-qunroll=yes** を指定するのと同等です。

-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して、アンロールに関し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、アンロールは使用できません。

```
xlc++ -qnounroll file.C

xlc++ -qunroll=no file.C
```

2. 以下の例では、アンロールを使用できます。

```
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のかの例は、 348 ページの『#pragma unroll』を参照してください。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
330 ページの『#pragma options』
348 ページの『#pragma unroll』

unwind

C C++

目的

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。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
102 ページの『eh』

upconv

C

目的

整数拡張を行うときに **unsigned** の指定を保持する。

構文

330 ページの『#pragma options』も参照してください。

注

-qupconv オプションは、**int** よりも小さい任意の **unsigned** 型を、 **int** ではなく **unsigned int** に拡張します。

符号なしの保持は、古い形式の C と互換性を持たせるために提供されています。ANSI C 規格では、符号なしの保持とは対照的に値を保持するように要請しています。

デフォルト

デフォルトは **-qnoupconv** ですが、**-qlanglvl=extc89** のときのデフォルトは、**-qupconv** です。コンパイラーは **unsigned** の指定を保存しません。

デフォルトのコンパイラーの処置では、整数拡張によって、**char**、**short int**、**int** ビット・フィールド、もしくはこれらの **signed** 型、**unsigned** 型、または **enumeration** 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 -qupconv
```

次の短いリストで、**-qupconv** の効果を示します。

```
#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;
}
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
160 ページの『langlvl』

utf

C C++

目的

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かった場合、コンパイラーは **-qutf** オプションを無視して警告を出します。

関連参照

41 ページの『コンパイラーのコマンド行オプション』

V

C C++

目的

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起動中のプログラムおよび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指示する。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、起動中のプログラム、および指定されているオプションを説明するメッセージを参照したりすることができるようにするには、以下を入力します。

```
xlc++ myprogram.C -V
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
51 ページの『# (ポンド記号)』
278 ページの『v』

v

C C++

目的

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起動中のプログラムおよび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指示する。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、起動中のプログラム、および指定されているオプションを説明するメッセージを参照したりすることができるようにするには、以下を入力します。

```
xlc myprogram.c -v
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
51 ページの『# (ポンド記号)』
277 ページの『V』

vftable

C++

目的

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に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 myprogram.C 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 -qnovftable
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

vrsave

C C++

目的

関数 prolog および epilog 内のコードを使用可能にして、 VRSAVE レジスターを保守する。

構文

ここで、

vrsave	コンパイル単位内の関数 prolog および epilog には、 VRSAVE レジスターの保守に必要なコードが含まれています。
novrsave	コンパイル単位内の関数 prolog および epilog には、 VRSAVE レジスターの保守に必要なコードが含まれていません。

注

#pragma altivec_vrsave を使用して、使用しているプログラム・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します。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
292 ページの『#pragma altivec_vrsave』

W

C C++

目的

リストしたオプションを、指定したコンパイラー・プログラムに渡す。

構文

プログラムは以下のとおりです。

プログラム	説明
a	アセンブラー
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ツール (コンパイル・フェーズ)
L	プロシージャー間分析ツール (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するときには、パラメーター・ストリング内でのコンマを表すために、エスケープ・シーケンスの円記号付きコンマ **(¥,)** を使用します。

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
   ```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

w

C C++

目的

警告以下のレベル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情報および警告メッセージは、このオプションでは使用不可にできません。例えば、多重定義の解決に関する問題によって発生する重大エラーでは、情報メッセージも生成されます。これらの通知メッセージは、**-w** オプションで使用不可にできません。

```
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

myprogram.c をコンパイルして警告メッセージを表示させないようにするには、以下を入力します。

```
xlc++ myprogram.C -w
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』
109 ページの『flag』

warn64

C C++

目的

32 ビットと 64 ビットのコンパイラー・モード間で起こりうるデータ変換問題の検査を使用可能にします。

構文

注

生成されるメッセージは、すべて通知レベルになっています。

-qwarn64 オプションは、32 ビットと 64 ビットのどちらのコンパイラー・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 ビットへの移行時に起こり得る問題を事前に探す補助として機能します。

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** 型を **int** 型に明示的または暗黙的に変換したために切り捨てが行われる。
- **int** 型を **long** 型に明示的または暗黙的に変換したために予期しない結果が発生する。
- キャスト操作によって**ポインター**型から **int** 型に明示的に変換したために無効なメモリー参照が行われる。
- キャスト操作によって **int** 型から**ポインター**型に明示的に変換したために無効なメモリー参照が行われる。
- **定数**型から **long** 型に明示的または暗黙的に変換したために問題が発生する。
- キャスト操作によって**定数**型から**ポインター**型に明示的または暗黙的に変換したために問題が発生する。
- ソース・ファイル内とコマンド行でプラグマ・オプション **arch** が矛盾している。

関連参照

41 ページの『コンパイラーのコマンド行オプション』
52 ページの『32、64』

xcall

C C++

目的

コンパイル単位内の静的ルーチンを扱うコードを、外部ルーチンであるかのように生成します。

構文

注

-qxcall は、**-qnoxcall** よりも処理が遅いコードを生成します。

例

myprogram.c をコンパイルしてすべての静的ルーチンを外部ルーチンとしてコンパイルするには、以下を入力します。

```
xlc myprogram.c -qxcall
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

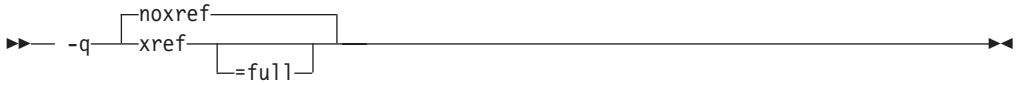
xref

C C++

目的

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。

330 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。

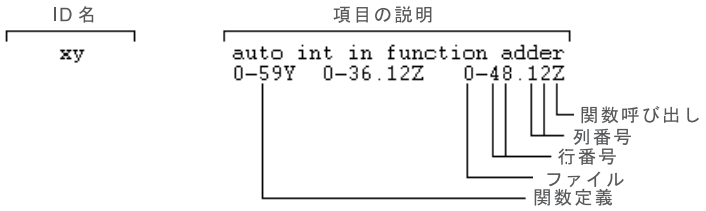
#pragma mc_func *function_name* ディレクティブで定義された関数は、いずれも **#pragma** ディレクティブの行で定義されているものとしてリストされます。

例

myprogram.C をコンパイルして、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生成させるには、以下を入力します。

```
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。



関連参照

41 ページの『コンパイラーのコマンド行オプション』
66 ページの『attr』
215 ページの『print』
326 ページの『#pragma mc_func』
330 ページの『#pragma options』

y

C C++

目的

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、以下のとおりです。

n	表現可能な近似値まで丸める。これはデフォルトです。
m	負の無限大の方向に丸める。
p	正の無限大の方向に丸める。
z	ゼロの方向に丸める。

例

myprogram.c をコンパイルして浮動小数点定数式をコンパイル時にゼロの方向に丸めるには、以下を入力します。

```
xlc myprogram.c -yz
```

関連参照

41 ページの『コンパイラーのコマンド行オプション』

汎用プラグマ

以下にリストしたプラグマは、汎用プログラミングで使用可能です。特に断りのない限り、プラグマは、C プログラムと C++ プログラムの両方で使用できます。

言語アプリケーション	#pragma	説明
C C++	#pragma align	構造体内のデータ項目の位置を合わせる。
C	#pragma alloca	関数 **alloca(size_t size)** のインライン・バージョンを提供する。
C C++	#pragma altivec_vrsave	関数 prolog および epilog にコードを追加して、 VRSAVE レジスターを保守する。
C C++	#pragma block_loop	ループ・ネストの特定ループ用のブロック化ループを作成するようにコンパイラーに指示を出す。
C C++	#pragma chars	文字データの符号型を設定する。
C C++	#pragma comment	オブジェクト・ファイルにコメントを入れる。
C C++	#pragma complexgcc	複雑な数学関数を呼び出すときのパラメーターの受け渡し方法をコンパイラーに指示する。
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。
C C++	#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。
C C++	#pragma enum	後続の enum 変数のサイズを指定する。
C C++	#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。
C C++	#pragma ibm snapshot	ユーザーはこれを使用して、ブレークポイントを設定できるロケーションを指定し、また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る。
C++	#pragma implementation	プラグマが入っているインクルード・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。
C C++	#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が生じる。

言語アプリケーション	#pragma	説明
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。
C C++	#pragma isolated_call	そのパラメーターによって暗黙指定される副次作用以外に、副次作用がない、または依存しない関数にマークを付ける。
C	#pragma langlvl	コンパイル用の C または C++ の言語レベルを選択する。
C C++	#pragma leaves	関数名を取って、関数の呼び出し後に命令に戻らない関数を指定する。
C C++	#pragma loop_id	ブロックにスコープ固有の ID でマークを付ける。
C C++	#pragma map	ID に対するすべてのリファレンスが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。
C C++	#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んだ関数を定義できるようにする。
C C++	#pragma nosimd	このディレクティブのすぐ後に続く **for** ループで VMX (Vector Multimedia Extension) 命令を生成しない ようにコンパイラーに指示する。
C C++	#pragma novector	次のループを自動的にベクトル化しない ようにコンパイラーに指示する。
C C++	#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。
C C++	#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。
C C++	#pragma pack	このプラグマの後に続く構造体のメンバーに適用され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設定される順序を指定する。
C C++	#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。
C C++	#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。
C C++	#pragma stream_unroll	ループに含まれるストリームを複数のストリームに分割する。
C C++	#pragma strings	ストリングのストレージ型を設定する。
C C++	#pragma unroll	プログラムの最内部および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向上させることができます。

言語アプリケーション	#pragma	説明
C C++	#pragma unrollandfuse	ネストされた **for** ループ上で、アンロールおよびフューズ操作を試行するようにコンパイラーに指示する。これにより、プログラムのパフォーマンスを向上させることができます。
C C++	#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。

関連タスク

27 ページの『プログラム・ソース・ファイル内のコンパイラー・オプションの指定』

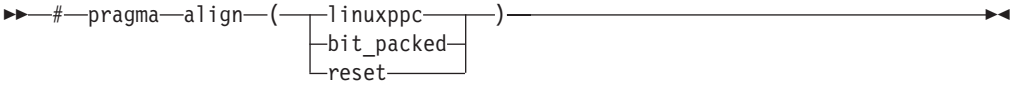
#pragma align

C C++

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



330 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の **-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、直前の位置合わせディレクティブの使用に戻ることができます。

例えば、インクルード・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 を指定すると、ご使用のソース・ファイルで **#pragma options align** を指定した場合と同じ影響があります。 **#pragma align** および **#pragma options align** の使用についての詳細および実例は、57 ページの『align』を参照してください。

関連参照

- 「*XL C/C++ プログラミング・ガイド*」の『集合体内のデータの位置合わせ』のセクションも参照してください。
- 「*XL C/C++ ランゲージ・リファレンス*」の『位置合わせ変数属性』および『パック変数属性』のセクションも参照してください。

#pragma alloca

C

説明

#pragma alloca ディレクティブは、コンパイラーが関数 **alloca(size_t***size***)** のインライン・バージョンを提供するように指定します。関数 **alloca(size_t***size***)** を使用して、オブジェクトにスペースを割り振ることができます。割り振られるスペースの大きさは、 *size* の値 (バイト単位) で決まります。割り振りスペースは、スタックに入ります。

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・オプションのいずれかを指定しなければなりません。

#pragma alloca 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 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。

関連参照

287 ページの『汎用プラグマ』
61 ページの『alloca』

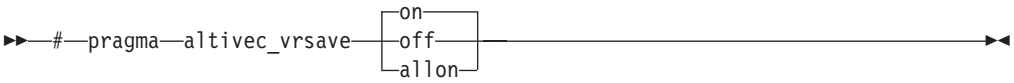
#pragma altivec_vrsave

C C++

説明

#pragma altivec_vrsave ディレクティブを使用可能にすると、関数 prolog および epilog は、VRSAVE レジスターを保守するためのコードを組み込みます。

構文



ここで、プラグマ設定では以下が行われます。

on　関数 prolog および epilog は、VRSAVE レジスターを保守するためのコードを組み込みます。

off　関数 prolog および epilog は、 VRSAVE レジスターを保守するためのコードを組み込みません。

allon　**altivec_vrsave** プラグマを含んだ関数は、 VRSAVE レジスターのすべてのビットを 1 に設定して、コンテキストの切り替えが発生した場合にすべての vector を使用して保管するように指示します。

注

VRSAVE レジスター内の各ビットは、vector レジスターに対応し、1 に設定されている場合は、対応する vector レジスターが、コンテキストの切り替えが発生した場合に保管すべきデータを含んでいることを示します。

このプラグマは関数内でのみ使用することができ、プラグマが出現する関数に対して機能します。同一の関数内で、異なる設定を使用してこのプラグマを指定すると、エラー状態が発生します。

関連参照

287 ページの『汎用プラグマ』
62 ページの『altivec』
280 ページの『vrsave』

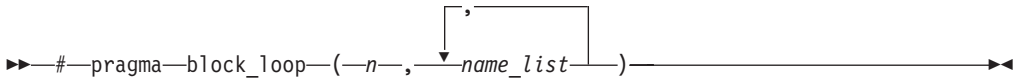
#pragma block_loop

C C++

説明

ブロックにスコープ固有の ID でマークを付ける。

構文



ここで、

n 反復グループのサイズを表す整数式です。

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* を指定しないと、**#pragma block_loop** ディレクティブの後に来る最初の **for** ループでブロック化が行われます。

ここで、*name* は、スコープ単位内で固有の ID です。

注

ループのブロック化を発生させるには、**for** ループの前に **#pragma block_loop** ディレクティブが配置されている必要があります。

#pragma blockloop を複数回指定したり、このディレクティブを同じ **for** ループの **nounroll**、**unroll**、**nounrollandfuse**、**unrollandfuse**、または **stream_unroll** #pragma ディレクティブと組み合わせて使用したりしてはなりません。

#pragma blockloop は、特定ループに対して複数回指定しないでください。

関連参照

287 ページの『汎用プラグマ』
272 ページの『unroll』
348 ページの『#pragma unroll』
350 ページの『#pragma unrollandfuse』
345 ページの『#pragma stream_unroll』

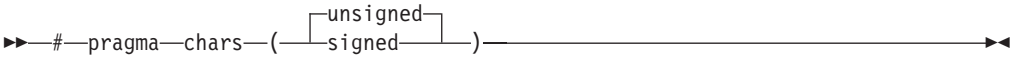
#pragma chars

C C++

説明

#pragma chars ディレクティブは、char オブジェクトの符号型を **signed** または **unsigned** のいずれか設定します。

構文



注

有効にするには、このプラグマがすべてのソース・ステートメントの前になければなりません。

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。プラグマをソース・ファイルで何度も指定した場合は、最初のプラグマが優先されます。

注: デフォルトの文字型は、unsigned char と同様です。

関連参照

287 ページの『汎用プラグマ』
76 ページの『chars』

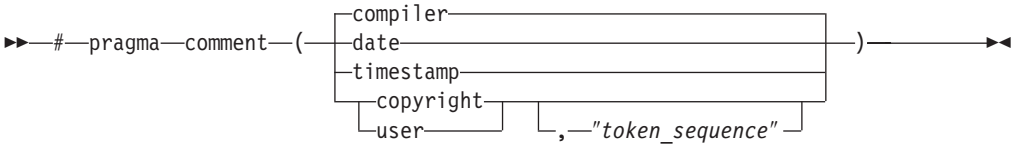
#pragma comment

C C++

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・ストリングを入れます。

構文



ここで、サブオプションでは以下が行われます。

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・コードが出力ファイル **a.out** を作成するようにコンパイルされていると想定します。

```
#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および他のストリングを検索することができます。コマンド

```
strings a.out
```

の発行により、**a.out** で検出される可能性がある他のすべてのストリングとともに、**a.out** に組み込まれたコメント情報が表示されます。例えば、上記のプログラム・コードを想定します。

```
Mon Mar 1 10:28:09 2004
XL C/C++ for Linux Compiler Version 7.0
Mon Mar 1 10:28:13 2004
My copyright
```

関連参照

287 ページの『汎用プラグマ』

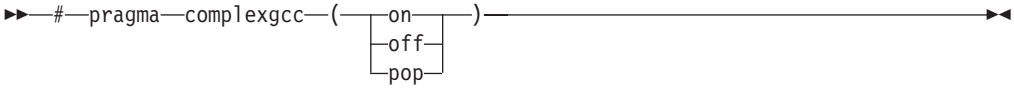
#pragma complexgcc

C C++

説明

#pragma complexgcc ディレクティブは、複雑な数学関数を呼び出すときに、コンパイラーに対してパラメーターの渡し方を命令します。

構文



ここで、サブオプションでは以下が行われます。

on
: **-qfloat=complexgcc** をスタックにプッシュします。このサブオプションは、複雑なパラメーターを渡したり、戻したりするときに、GCC 規則を使用するようにコンパイラーに指示します。

off
: **-qfloat=nocomplexgcc** をスタックにプッシュします。このサブオプションは、複雑なパラメーターを渡したり、戻したりするときに、AIX 規則を使用するようにコンパイラーに指示します。

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、設定が有効である間に宣言または定義された関数にのみ影響します。これは、それ以外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 */
}
```

関連参照

287 ページの『汎用プラグマ』
81 ページの『complexgccincl』
111 ページの『float』

#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、逆方向の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、フォームの構成を使用して、クラス、関数、またはメンバーの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・スコープ内で定義しなければなりません (つまり、関数/クラス本体の中に収めることはできません)。これは、テンプレート関数を効率的または自動的に生成するためにプログラムを再編成する場合に使用されます。

関連参照

287 ページの『汎用プラグマ』
301 ページの『#pragma do_not_instantiate』
315 ページの『#pragma instantiate』

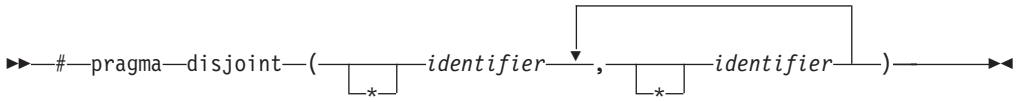
#pragma disjoint

C C++

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 ID はどれも、同じ物理ストレージを共用していない (これによって、さらに最適化の機会が提供されます) ことをコンパイラーに通知します。 ID のいずれかが実際に物理ストレージを共用している場合は、プラグマが原因でプログラムが間違った結果を出す場合があります。

ディレクティブの中の ID は、プログラム中のプラグマが現れる点では可視でなくてはなりません。disjoint 名前リスト内の 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
- 列挙型定数
- 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、外部変数 *b* とメモリーを共用せず、また外部変数を指し示すこともありません。したがって、*ptr_a* が指し示すオブジェクトに 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* とメモリーを共用せず、またそれを指し示すこともありません。コンパイラーは、*another_function* の引き数の値を 6 と見なすことができるので、メモリーから変数を再ロードしません。

関連参照

287 ページの『汎用プラグマ』
133 ページの『ignprag』
55 ページの『alias』

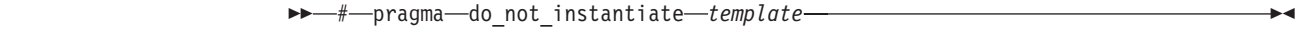
#pragma do_not_instantiate

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をインスタンス化しない ようにコンパイラーに指示します。

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 do_not_instantiate Stack < int >
```

注

定義を入力するテンプレートの暗黙的インスタンス化を抑制するには、このプラグマを使用します。

テンプレート・インスタンス化を手動で処理しており (つまり、**-qnotempinc** および **-qnotemplateregistry** が指定されている)、指定されたテンプレート・インスタンス化が別のコンパイル単位にすでに存在する場合、 **#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くなります。

関連参照

287 ページの『汎用プラグマ』
298 ページの『#pragma define』
315 ページの『#pragma instantiate』
259 ページの『tempinc』
261 ページの『templateregistry』

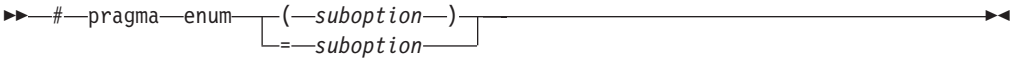
#pragma enum

C C++

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。宣言の左方中括弧のサイズは、宣言内で enum ディレクティブがさらに生じ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* は次のいずれかです。

1	列挙型は長さが 1 バイトで、列挙の値の範囲が **signed char** の制限内である場合、型は **char** で、そうでない場合は **unsigned char** です。
2	列挙型は長さが 2 バイトで、列挙の値の範囲が **signed short** の制限内である場合、型は **short** で、そうでない場合は **unsigned short** です。
4	列挙型は長さが 4 バイトで、列挙の値の範囲が **signed int** の制限内である場合、型は **int** で、そうでない場合は **unsigned int** です。
8	列挙型は長さが 8 バイトです。 32 ビット・コンパイル・モードでは、列挙値の範囲が **signed long long** の制限内にある場合、その列挙は **long long** 型で、そうでない場合は **unsigned long long** です。 64 ビット・コンパイル・モードでは、列挙値の範囲が **signed long** の制限内である場合は、その列挙は **long** 型で、そうでない場合は **unsigned long** です。
int	**#pragma enum=4** と同じ。
intlong	列挙の値の範囲が **int** の制限を越える場合、列挙がストレージの 8 バイトを占有するように指定します。 **#pragma enum=8** の説明を参照してください。 列挙の値の範囲が **int** の制限を超えない場合は、列挙は 4 バイトのストレージを占有し、 **int** によって示されます。
small	列挙型は、すべての変数を含むことができる最小の整数型です。 8 バイトの enum が結果として生じる場合、実際に使用される列挙型はコンパイル・モードに依存しています。 **#pragma enum=8** の説明を参照してください。
pop	このサブオプションは、enum サイズ設定を前の **#pragma enum** 設定にリセットします。前の設定がない場合、 **-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・ファイルに組み込んだ各 **#pragma enum** ディレクティブごとに、対応する **#pragma enum=reset** を該当するファイルの終わりの前に組み込むようにしてください。これは、 **#include** する他のファイルの **enum** 設定がファイルによって変更される可能性をなくす唯一の方法です。

#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** を使用しようとすると、警告メッセージが生成され、このオプションは無視されます。

例

1. **pop** および **reset** サブオプションの使用法は、以下のコード・セグメントで示されています。

```
#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インクルード・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、以下のインクルード・ファイル small_enum.h では、さまざまな最小サイズの列挙を宣言し、インクルード・ファイルの最後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 **test_enum** および **first_order** は、両方とも 4 バイトのストレージを占有し、 **int** 型です。small_enum.h で定義された変数 **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** オプションでコンパイルすると、

   ```
   enum e_tag {a, b, c} e_var;
   ```

   enum 定数の範囲は 0 から 2 までになります。この範囲は、上記の表で説明したいずれの範囲にも収まります。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** オプションでコンパイルすると、

   ```
   enum e_tag {a=-129, b, c} e_var;
   ```

   enum 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)** および **int (signed int)** の範囲にしか収まりません。 **short (signed short)** は小さ過ぎるため、 **enum** を表すのに使用されます。

5. 以下のコマンドを使用してファイル myprogram.C をコンパイルすると、

   ```
   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すると、ソース・ファイル内の **enum** 変数は、すべて最小量のストレージを占有します。

6. 以下の行を含むファイル 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;
   ```

   以下のコマンドの使用

   ```
   xlc++ yourfile.C
   ```

   enum 変数 **first_order** のみ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しか占有しません)。他の 2 つの enum 変数 **test_enum** および **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、無効な列挙または **#pragma enum** の使用です。

- enum の宣言内で **#pragma enum** を使用することによって 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・セグメントには警告が生成され、 2 番目に現れる **enum** オプションは無視されます。

```
#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)** のいずれかの範囲内でなければなりません。例えば、次のコード・セグメントにはエラーが含まれます。

```
#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** の範囲内に収まりません。

```
#pragma options enum=small
enum e_tag { a=0,
             b=4294967296 /* error: larger than maximum int */
           } e_var;
#pragma options enum=reset
```

関連参照

287 ページの『汎用プラグマ』
104 ページの『enum』
330 ページの『#pragma options』

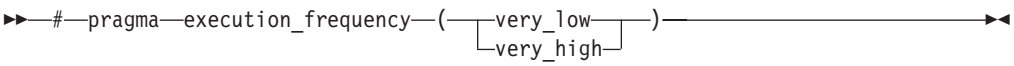
#pragma execution_frequency

C C++

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** ステートメント・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ていると、分岐の際には ″Block C″ が選択されることになります。

```
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** ステートメント・ブロックで使用します。

```
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 番目のプラグマのみが効果を有し、 ″**if (boo)**″ 判別内で、プログラム実行が Block A または Block B のいずれ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。

関連参照

287 ページの『汎用プラグマ』

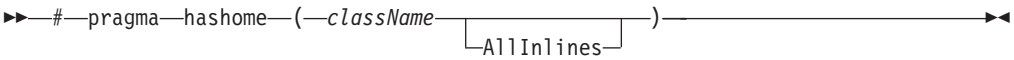
#pragma hashome

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であり、ある種のインライン関数とともに、静的には生成されません。代わりに、 **#pragma ishome** が指定されたクラスのコンパイル単位で外部参照されます。

構文



ここで、

className 上記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、またこれを定義しておか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。この引き数は大/小文字を区別しません。

注

#pragma hashome と一致しない **#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ことにより、仮想関数テーブル、およびコンパイル単位 b.o ではなく、a.o のみの S::foo() の定義が生成されます。これにより、アプリケーション用に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() {}
```

関連参照

287 ページの『汎用プラグマ』
316 ページの『#pragma ishome』

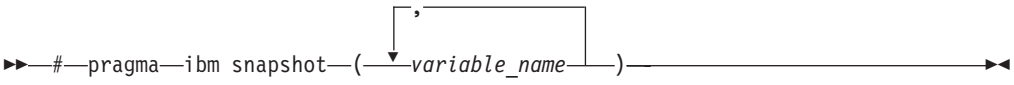
#pragma ibm snapshot

C C++

説明

ユーザーは **#pragma ibm snapshot** を使用して、ブレークポイントを設定できるロケーションを指定し、また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++ コンパイラーによって作成された最適化コードをデバッグしやすくするために提供されています。デバッグ・セッション中、この行にブレークポイントを置いて、名前付き変数の値を表示することができます。プログラムのコンパイル時に最適化を行うように指定し、またオプション **-g** を組み込んでおくと、名前付き変数をデバッガーから見えるようにすることができます。

#pragma ibm snapshot で指定した変数は、デバッガーでの監視中は読み取り専用と見なすべきであり、変更してはなりません。デバッガー内のこれらの変数を変更すると、予期しない動きをする場合があります。

例

```
#pragma ibm snapshot(a, b, c)
```

プログラム内のこのポイントでデバッガーによりブレークポイントが設定された場合、変数 a、b、および c の値は可視でなければなりません。

関連参照

- 120 ページの『g』
- 197 ページの『O、optimize』

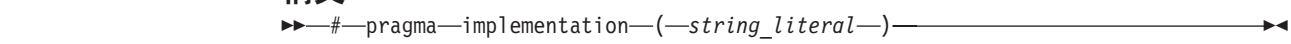
#pragma implementation

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、プラグマが入っている #include 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、宣言が使用できるところならばどこにでも入れることができます。これは、テンプレート関数を効率的または自動的に生成するためにプログラムを再編成する場合に使用されます。

関連参照

287 ページの『汎用プラグマ』
263 ページの『tempmax』

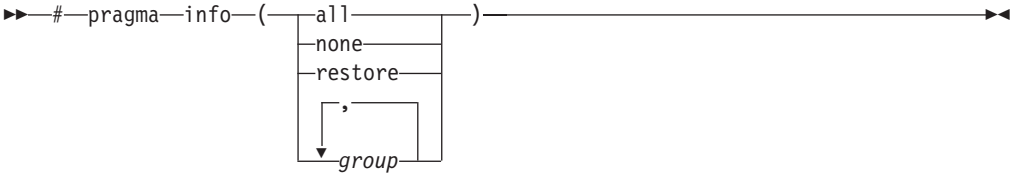
#pragma info

C C++

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指示します。

構文



ここで、

`all` すべての診断検査をオンにします。
`none` プログラムの特定の部分について、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であったオプションを復元します。

group 指定した診断 *group* に関連付けられる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 *group* 名を指定できます。

group	戻される、または抑止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動きをする場合がある C コード。
cls|nocls	C++ クラス。
cmp|nocmp	符号なし比較で生じうる冗長度。
cnd|nocnd	条件式で生じうる冗長度または問題。
cns|nocns	定数を含む命令。
cnv|nocnv	型変換。
dcl|nodcl	宣言の整合性。
eff|noeff	効果のない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	初期設定で起こりうる問題。
inl|noinl	インラインされていない関数。
lan|nolan	言語レベル効果。
obs|noobs	廃止されたフィーチャー。
ord|noord	評価の順序が指定されていない。
par|nopar	未使用パラメーター。
por|nopor	移送不能な言語構造体。
ppc|noppc	プリプロセッサーを使用した場合に起こりうる問題。
ppt|noppt	プリプロセッサー・アクションのトレース。
pro|nopro	欠落関数プロトタイプ。
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	データまたは精度の、起こりうる切り捨てまたは欠落。
tru|notru	コンパイラーで切り捨てられた変数名。
trx|notrx	16 進浮動小数点定数の丸め。
uni|nouni	初期化されていない変数。
upg|noupg	現在のコンパイラー・リリースの新しい振る舞いを、前のリリースと対比して示すメッセージを生成する。
use|nouse	未使用の自動変数および静的変数。
vft|novft	C++ プログラムでの仮想関数テーブルの生成。
zea|nozea	ゼロ範囲の配列。

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で指定した、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、効力のない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することと、初期化されていない変数を識別するメッセージを抑制することを、コンパイラーに指示します。 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指示します。

```
#pragma info(eff, nouni)
int MyFunction1()
{
  .
  .
  .

}

#pragma info(restore)
int MyFunction2()
{
  .
  .
  .

}
```

関連参照

- 134 ページの『info』

#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を即時にインスタンス化するようにコンパイラーに指示します。

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ます。新規のコードは、標準 C++ 明示的インスタンス化を使用しなければなりません。

テンプレート・インスタンス化を手動で処理している (つまり、**-qnotempinc** および **-qnotemplateregistry** が指定されている) 場合、 **#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化がコンパイル単位内に表示されます。

関連参照

287 ページの『汎用プラグマ』
298 ページの『#pragma define』
301 ページの『#pragma do_not_instantiate』
259 ページの『tempinc』
261 ページの『templateregistry』

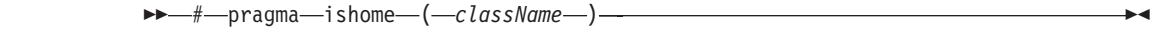
#pragma ishome

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、アプリケーション用に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

#pragma hashome と一致しない **#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ことにより、仮想関数テーブル、およびコンパイル単位 b.o ではなく、a.o のみの S::foo() の定義が生成されます。これにより、アプリケーション用に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() {}
```

関連参照

287 ページの『汎用プラグマ』
308 ページの『#pragma hashome』

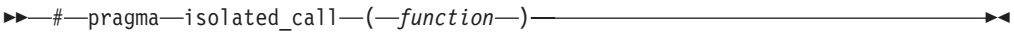
#pragma isolated_call

C C++

説明

#pragma isolated_call ディレクティブは、パラメーターが意味する副次作用の他には、副次作用を持つことも、また副次作用に依存することも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、そのパラメーターが意味する副次作用の他には副次作用がなく、それに依存もし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
- 同じ起動時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
- 上記のいずれかの行う関数を呼び出す場合

基本的には、実行時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り渡された関数引き数の変更だけが許可されている副次作用です。他の副次作用のある関数は、 **#pragma isolated_call** ディレクティブにリストされている場合、誤った結果を与える可能性があります。

関数を **isolated_call** としてマーク付けすると、呼び出し先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、および、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、実行時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、関数に渡されるポインター引き数、すなわち、参照体による呼び出しによって、ストレージを変更することもできます。それ自体を呼び出す関数、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** ディレクティブ内にリストすると、予想しない結果が出ます。

-qignprag コンパイラー・オプションによって、別名割り当てプラグマは無視されます。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** には副次作用がないので、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** への引き数には値 6 が指定されていて、メモリーから変数が再ロードされないことを前提と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);
}
```

関連参照

287 ページの『汎用プラグマ』
133 ページの『ignprag』
154 ページの『isolated_call』

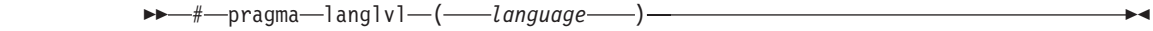
#pragma langlvl

C

説明

#pragma langlvl ディレクティブは、コンパイルで使用される C 言語レベルを選択します。

構文



ここで、*language* については、以下で説明します。

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	非 stdc89 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適合させます。
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 言語定義に適合するコンパイルです。これは現在は SAA C レベル 2 です。
saal2	SAA C レベル 2 CPI 言語定義に適合するコンパイルです。これにはいくつかの例外があります。
stdc89	ANSI C89 標準に適合するコンパイルで、ISO C90 としても知られています。
stdc99	ISO C99 標準に適合するコンパイルです。
extc89	コンパイルは、ANSI C89 標準に適合しており、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 標準に適合しており、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。

デフォルト

デフォルトの言語レベルは、コンパイラーの呼び出しに使用するコマンドに応じてそれぞれ異なります。

呼び出し	デフォルトの言語レベル
xlc	extc89
cc	extended
c89	stdc89
c99	stdc99

注

この pragma は、1 つのソース・ファイル内に 1 回だけ指定することができます。また、必ずソース・ファイル内のどの非コメント・ステートメントよりも前に置かなければなりません。

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宣言および定義を使用可能にします。

このディレクティブは、プリプロセッサーの動きを動的に変更することができます。結果として、**-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ないでコンパイルしたときに生成される結果とは異なる結果を生成する場合があります。

関連参照

287 ページの『汎用プラグマ』
99 ページの『E』
160 ページの『langlvl』

「*C/C++ ランゲージ・リファレンス*」の『*IBM C 言語拡張機能*』および『*IBM C++ 言語拡張機能*』のセクションも参照してください。

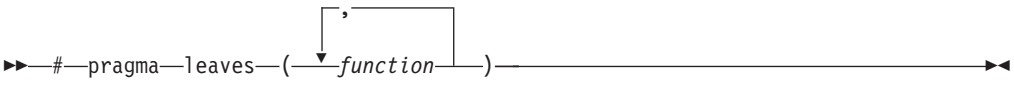
#pragma leaves

C C++

説明

#pragma leaves は関数名を取得し、呼び出し後に関数がディレクティブに戻ることがないように指定します。

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指示します。

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。これと同様の関数には、**exit**、**longjmp**、 **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
 }
}
```

関連参照

287 ページの『汎用プラグマ』

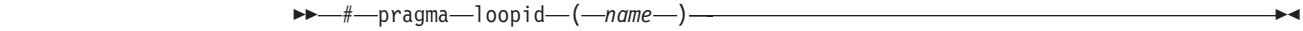
#pragma loop_id

C C++

説明

ブロックにスコープ固有の ID でマークを付ける。

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。指定した名前を **#pragma block_loop** で使用して、そのループ上での変換を制御することができます。これを使用して、**-qreport** コンパイラー・オプションの使用を通じてのループ変換に関する情報を提供することもできます。

#pragma loopid は、指定されたループに対して複数回指定しないでください。

関連参照

287 ページの『汎用プラグマ』
272 ページの『unroll』
293 ページの『#pragma block_loop』
348 ページの『#pragma unroll』
350 ページの『#pragma unrollandfuse』

#pragma map

C C++

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指示します。

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多重定義されていない関数の名前。

C++ ID が多重定義関数またはメンバー関数の名前である場合、プラグマはコンパイラーにより生成された名前をオーバーライドするおそれがあります。これにより、リンク時に問題が生じます。

function_signature
内部リンケージ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++ のデフォルト・シグニチャーである C++ リンケージ・シグニチャーを持つ名前) にリンクする場合は、マングルされた名前を指定します。下の**例**の、例 4 を参照してください。

注

#pragma map を使用して、以下のもの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
- リンケージにビルドされた関数

このディレクティブは、プログラムのどこに記述してもかまいません。ディレクティブ内に現れる ID は、プロトタイプ引き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。

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;}
```

関連参照

287 ページの『汎用プラグマ』

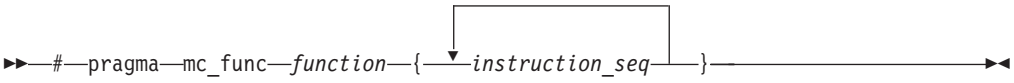
#pragma mc_func

C C++

説明

#pragma mc_func ディレクティブは、マシン・インストラクションの短いシーケンスを含んだ関数を定義できるようにします。

構文



ここで、

function C または C++ プログラムにおいて事前定義された関数を指定する必要があります。関数が事前定義されていない場合、コンパイラーはプラグマを関数定義として処理します。

instruction_seq ゼロ個以上の 16 進数字のシーケンスを含んだストリングです。数字の数は、32 ビットの整数倍で構成される必要があります。

注

mc_func プラグマによって、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・コード内に埋め込めます。このプラグマは、通常のリンケージ・コードの代わりに、指定された命令を生成するようにコンパイラーに指示します。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した外部関数への呼び出しに関連するパフォーマンス負荷を回避できます。このプラグマの機能は、このコンパイラーおよび他のコンパイラーにある **asm** キーワードに類似しています。

mc_func プラグマは、関数を定義し、関数の定義が通常行われるプログラム・ソース内にのみ出現しなければなりません。 **#pragma mc_func** によって定義される関数名は、事前に宣言またはプロトタイプ化される必要があります。

コンパイラーは、他の関数と同様にパラメーターを関数に渡します。例えば、整数型の引き数を取る関数の場合、1 番目のパラメーターは GPR3 に、 2 番目は GPR4 に、というように渡されます。関数によって戻される値は、integer 値の場合は GPR3 に、 float または double 値の場合は FPR1 となります。ご使用の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、 **#pragma reg_killed_by** を参照してください。

#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・セットをリストしていない限り、 *instruction_seq* から生成されるコードは、ご使用のシステムで使用可能な揮発性レジスターをどれでも使用できます。

インライン化オプションは、 **#pragma mc_func** によって定義された関数に影響を与えません。しかし、**#pragma isolated_call** を使用して、そのような関数のランタイム・パフォーマンスを改善できます。

例

次の例では、**add_logical** と呼ばれる関数を定義するために、**#pragma mc_func** が使用されています。関数は、マシン・インストラクションから構成されて、いわゆる*循環桁上げ* を使用して 2 つの int を加算しています。つまり、加算の結果、桁上げが発生する場合、桁上げが合計に加算されます。これは、チェックサム計算において頻繁に使用されます。

例では、 **#pragma mc_func** によって定義された関数による変更が可能な揮発性レジスターの特定のセットをリストする **#pragma reg_killed_by** の使用法も示しています。

```
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);
}
```

関連参照

287 ページの『汎用プラグマ』
318 ページの『#pragma isolated_call』
341 ページの『#pragma reg_killed_by』
64 ページの『asm』

#pragma nosimd

C C++

説明

#pragma nosimd ディレクティブは、このディレクティブのすぐ後に続く **for** ループで VMX (Vector Multimedia Extension) 命令を生成しない ようにコンパイラーに指示します。

構文

注

このディレクティブは、**-qhot=simd** が **-qarch=ppc970** とともに有効になっている場合のみ、効果があります。これらのコンパイラー・オプションが有効になっていると、コンパイラーは、連続する配列のエレメントに関してループ内で実行される特定の操作を VMX (Vector Multimedia Extension) 命令の呼び出しに変換します。この呼び出しは、いくつかの結果を一度に計算するため、個々の結果を連続して計算するよりは速くなります。

#pragma nosimd ディレクティブは、コンパイラーが、特定の **for** ループに対する VMX (Vector Multimedia Extension) 命令を生成することを防ぎます。このディレクティブの影響を受ける **for** ループは、このディレクティブの後に続く最初のプログラム・ステートメントでなければなりません。

#pragma nosimd ディレクティブは、その直後の **for** ループに対してのみ適用されます。このディレクティブは、指定されたループ内でネストされる可能性のある他の **for** ループ上では影響はありません。

#pragma nosimd は、ループ最適化および並列処理のディレクティブとともに使用することができます。

関連参照

287 ページの『汎用プラグマ』
63 ページの『arch』
103 ページの『enablevmx』
127 ページの『hot』

#pragma novector

C C++

説明

#pragma novector ディレクティブは、次のループを自動的にベクトル化しない ようにコンパイラーに指示します。

構文

注

このディレクティブは、**-qhot=vector** が有効になっている場合のみ、効果があります。**-qhot=vector** が有効になっていると、コンパイラーは、連続する配列のエレメント (例えば、平方根、逆数平方根) に関してループ内で実行される特定の操作をベクター・ライブラリー・ルーチンの呼び出しに変換します。この呼び出しは、同時にいくつかの結果を計算します。これは、各結果を逐次計算するよりも高速です。

#pragma novector ディレクティブは、コンパイラーが **for** ループに対する命令を自動的にベクトル化することを防ぎます。このディレクティブの影響を受ける **for** ループは、このディレクティブの後に続く最初のプログラム・ステートメントでなければなりません。

#pragma novector ディレクティブは、その直後の **for** ループに対してのみ適用されます。このディレクティブは、指定されたループ内でネストされる可能性のある他の **for** ループ上では影響はありません。

#pragma novector は、ループ最適化および並列処理のディレクティブとともに使用することができます。

関連参照

287 ページの『汎用プラグマ』
127 ページの『hot』

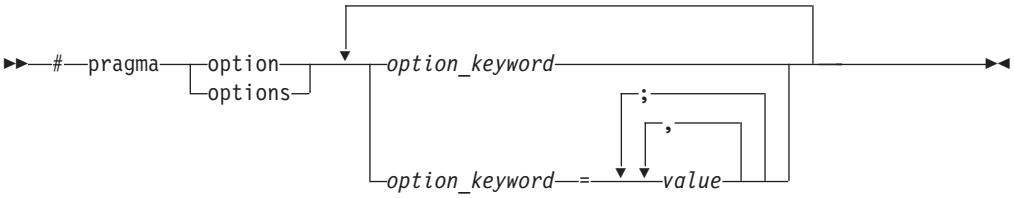
#pragma options

C C++

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

デフォルトでは、プラグマ・オプションは通常、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・プログラムのいずれのステートメントよりも前に指定しなければなりません。ただし、コメント、ブランク行、および他の **#pragma** の指定に限り、これらのディレクティブの前に置けます。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・プログラムのすべてのコードの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。プログラムの中で、他の **#pragma** 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、以下のとおり、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
```

下記の表の設定は、 **#pragma options** で有効なオプション 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。

言語 アプリケーション		#pragma options に有効な設定 *option_keyword*	同等のコンパイラー・オプション	説明
C	C++	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。
C	C++	[no]ansialias	-qalias	型ベースの別名割り当てを最適化中に使用するかどうかを指定する。
C	C++	assert=*option*	-qalias	別名割り当てアサーションをコンパイル単位に適用するようにコンパイラーに要求する。
C	C++	[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。
C	C++	chars=*option*	-qchars #pragma chars も参照	コンパイラーに、 char 型の変数をすべて signed または unsigned のいずれかとして処理するように指示する。
C	C++	[no]check	-qcheck	特定のタイプの実行時検査を行うコードを生成する。
C	C++	[no]compact	-qcompact	最適化とともに使用すると、可能な場合に、実行速度を犠牲にしてコード・サイズが削減される。
C	C++	[no]dbcs	-qmbcs、dbcs	ストリング・リテラルとコメントには、マルチバイト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。
C	C++	[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。
C	C++	[no]dollar	-qdollar	$ シンボルを ID の名前で使用できるようにする。
C	C++	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。
C	C++	flag=*option*	-qflag	報告させる診断メッセージの最低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)
C	C++	float=[no]*option*	-qfloat	浮動小数点演算を高速化するか正確度を向上させるために、各種浮動小数点オプションを指定する。

言語 アプリケーション		#pragma options に有効な設定 *option_keyword*	同等のコンパイラー・オプション	説明
C	C++	[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。
C	C++	[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。
C	C++	[no]funcsect	-qfuncsect	各関数ごとの命令を別個の cset に入れる。
C	C++	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。
C	C++	[no]idirfirst	-qidirfirst	ユーザー・インクルード・ファイルの検索順序を指定する。
C	C++	[no]ignerrno	-qignerrno	コンパイラーが、システム呼び出しによって **errno** が変更されないと想定して最適化を行うことを許可する。
C	C++	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指示する。
C	C++	[no]info=*option*	-qinfo #pragma info も参照	通知メッセージを作成する。
C	C++	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。
C	C++	[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を行うため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って、高速な外部結合を生成する。
C	C++	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動きを動的に変更することができます。結果として、**-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ないでコンパイルしたときに生成される結果とは異なる結果を生成する場合があります。
C	C++	[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。
C	C++	[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。
C	C++	[no]longlong	-qlonglong	プログラムで long long 型を許可する。

言語 アプリケーション		#pragma options に有効な設定 *option_keyword*	同等のコンパイラー・オプション	説明
C	C++	[no]maxmem=*number*	-qmaxmem	指定した重大度以上のエラーの件数が指定した数に達した場合に、コンパイルを停止するようにコンパイラーに指示する。
C	C++	[no]mbcs	-qmbcs、dbcs	ストリング・リテラルとコメントには、マルチバイト文字を含むことができる。
C	C++	[no]optimize optimize=*number*	-O、optimize	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 339 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。
C	C++	[no]proclocal、 [no]procimported、 [no]procunknown	-qproclocal、 procimported、 procunknown	ローカル、インポートされるもの、または不明として、関数にマークを付ける。
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。
C	C++	[no]ro	-qro	ストリング・リテラルの保管型を指定する。
C	C++	[no]roconst	-qroconst	定数値の保管場所を指定する。
C	C++	[no]showinc	-qshowinc	**-qsource** で使用された場合は、すべてのインクルード・ファイルをソース・リストに組み込む。
C	C++	[no]source	-qsource	ソース・リストを作成する。
C	C++	spill=*number*	-qspill	レジスター割り振り予備域のサイズを指定する。
C	C++	[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。
C	C++	[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。
C	C++	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。

言語 アプリケーション		#pragma options に有効な設定 *option_keyword*	同等のコンパイラー・オプション	説明
C	C++	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。
C	C++	[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 unsigned 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。
C	C++	[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。

関連参照

287 ページの『汎用プラグマ』
99 ページの『E』

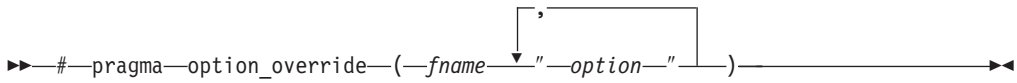
#pragma option_override

C C++

説明

#pragma option_override ディレクティブにより、特定の関数に対する代替最適化オプションを指定します。

構文



option の有効な設定と構文、およびそれらに対応するコマンド行オプションを下記に示します。

#pragma option_override *option* **の設定および構文**	**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、コマンド行で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のみ、特定のタイプのランタイム・エラーが発生する可能性があります。このプラグマにより、プログラム内の特定の関数 (*fname*) に対するコマンド行最適化設定をオーバーライドします。これは、これらの関数のプログラミング・エラーを識別したり修正したりする場合に役立ちます。

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。コンパイル中の残りのプログラムに適用されたレベルより下のレベルでの、関数単位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無効です。

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、多重定義されたメンバー関数とともに使用することはできません。

このプラグマは、コンパイル単位で定義された関数にのみ有効で、例えば、以下のように、コンパイル単位の任意の場所に置くことができます。

- コンパイル単位の前または後ろ
- 関数定義の前または後ろ
- 関数宣言の前または後ろ
- 参照された関数の前または後ろ
- 関数定義の内側または外側

関連参照

287 ページの『汎用プラグマ』
80 ページの『compact』
197 ページの『O、optimize』
243 ページの『spill』
249 ページの『strict』

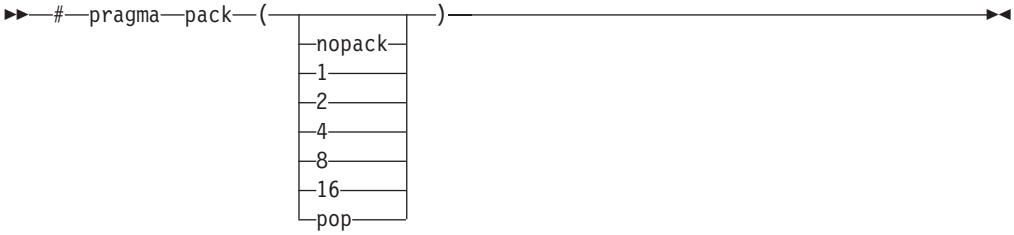
#pragma pack

C C++

説明

#pragma pack ディレクティブは、このディレクティブの後に続く構造体のメンバーに適用される現行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、指定したバイト位置合わせ、または自然な位置合わせ境界のいずれか少ない方で位置合わせされ、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・スタックには ″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。
(*引き数の指定なし*)	#pragma pack() の指定は、 #pragma pack(pop) の指定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けて宣言される構造体のメンバーについてのみ、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が直接、影響を受け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が生ずることにより、位置合わせ規則に従って構造体全体の位置合わせが影響を受ける場合があります。

#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。新規の位置合わせ規則が作成されると、新規 **#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されて、位置合わせ規則スタックから外された場合、現行の **#pragma pack** スタックは空になり、直前の **#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 ディレクティブは、ビット・フィールドがビット・フィールド・コンテナー境界をクロスする原因となります。

例

1. 以下に示すコードでは、構造体 s_t2 のメンバーは 1 バイトにパックされますが、構造体 s_t1 はその影響を受けません。これは、 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。ただし、s_t2 の宣言はプラグマ・ディレクティブより後に始まるため、s_t2 は影響を受けます。

```
struct s_t1 {
    char a;
    int b;
    #pragma pack(1)
    struct s_t2 {
            char x;
            int y;
    } S2;
    char c;
    int d;
} S1;
```

2. この例では、**#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に影響を与える可能性があるのかを示します。

```
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

関連参照

287 ページの『汎用プラグマ』
57 ページの『align』
330 ページの『#pragma options』

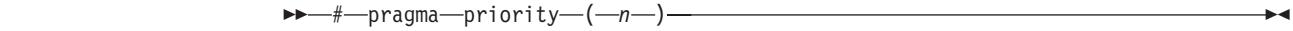
#pragma priority

C++

説明

#pragma priority ディレクティブは、静的オブジェクトを初期化する順序を指定します。

構文



注

n の値は、 101 から 65535 まで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。小さな値は、優先順位が高いことを示します。大きな値は、優先順位が低いことを示します。

明示的な値が変数属性 init_priority によって指定されたり、別の **#pragma priority** ディレクティブが検出されたりしない限り、優先度の値は **#pragma priority** ディレクティブに続くすべてのグローバル・オブジェクトおよび静的オブジェクトに適用されます。

同じ優先度の値を持つオブジェクトは、宣言の順序で構成されます。 **#pragma priority** を使用して、複数のファイルにまたがったオブジェクトの作成順序を指定します。ただし、ソース・ファイルから、実行可能な、または共用のライブラリー・ターゲットを作成する場合、コンパイラーは、 **#pragma priority** をオーバーライドする可能性のある、依存性の順序付けを検査します。

例えば、オブジェクト A のコピーがオブジェクト B コンストラクターにパラメーターとして渡されると、コンパイラーは、上から下、または **#pragma priority** の順序に違反することになっても、最初に構成される A に対する調整を行います。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。

変数属性 init_priority: C++ の変数属性 init_priority は、クラス型の共用変数に優先順位を割り当てるためにも使用できます。詳しくは、「*XL C/C++ ランゲージ・リファレンス*」を参照してください。

例

```
#pragma priority(1001)
```

関連参照

- 134 ページの『info』

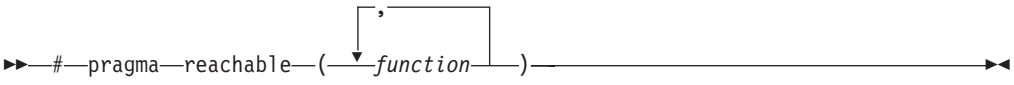
#pragma reachable

C C++

説明

#pragma reachable ディレクティブは、ルーチンである*関数* 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 マクロとともに使用する必要があります。

構文



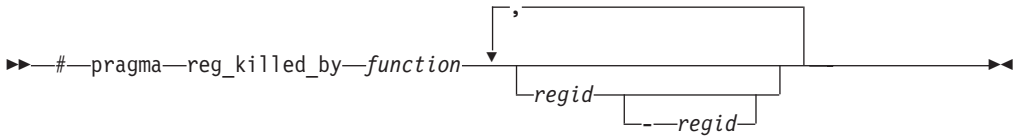
#pragma reg_killed_by

C C++

説明

#pragma reg_killed_by ディレクティブは、指定の関数によって変更 (強制終了) 可能な一連の揮発性レジスターを指定する。この pragma は、**#pragma mc_func** を使って定義された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、単一レジスターまたは一連のレジスターのシンボル名。一連のレジスターは、ダッシュで区切られた開始および終了の両レジスターのシンボル名を使用して識別されます。レジスターが指定されていない場合、どのレジスターも、指定された *function* によって変更されません。

シンボル名は、2 つの部分から構成されています。 1 つ目の部分は、レジスター・クラス名であり、 ″a″ から ″z″ および/または ″A″ から ″Z″ の範囲にある 1 つ以上の文字のシーケンスを使用して指定されます。

2 つ目の部分は、符号なし int の範囲にある整数です。この数値は、レジスター・クラス内の特定のレジスター番号を表します。一部のレジスター・クラスには、レジスター番号の指定が不要であり、レジスター番号を指定しようとすると、エラーが起こります。

regid が指定されていない場合、どの揮発性レジスターも、指定された *function* によって強制終了されません。

レジスター	
クラスおよび [レジスター番号]	**説明および使用法**
ctr	カウント・レジスター (CTR)
cr[0-7]	条件レジスター (CR) • このクラス内の各レジスターは、条件レジスター内の 4 ビット・フィールドの 1 つです。 • 8 CR フィールドのうち、**cr0**、**cr1**、および **cr5**-**cr7** のみが **#pragma reg_killed_by** によって指定されることができます。
fp[0-31]	浮動小数点レジスター (FPR) • 32 個のマシン・レジスターのうち、 **fp0** ～ **fp13** のみが **#pragma reg_killed_by** によって指定されることができます。
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 個のマシン・レジスターのうち、 **gr0**、および **gr3** ～ **gr12** のみが **#pragma reg_killed_by** によって指定されることができます。
vr[0-31]	ベクトル・レジスター (Altivec プロセッサーのみ)
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用に生成されたコードは、ご使用のシステムで使用可能な揮発性レジスターをどれでも変更できます。 **#pragma reg_killed_by** を使用して、そのような関数によって変更される揮発性レジスターの特定のセットを明示的にリストすることができます。このリストにないレジスターは、変更されません。

regid によって指定されたレジスターは、以下の要件を満たさなければなりません。

- 登録名のクラス名部分は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であるときは、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

次の例では、 **#pragma mc_func** によって定義された関数が使用する揮発性レジスターの特定のセットをリストする **#pragma reg_killed_by** の使用法を示しています。

```
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);
}
```

関連参照

287 ページの『汎用プラグマ』
326 ページの『#pragma mc_func』

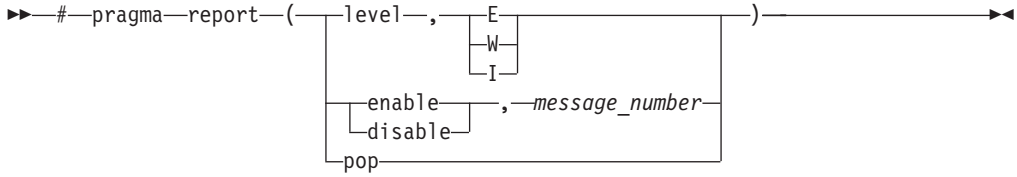
#pragma report

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。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・レベルは直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と、警告を出して、そのレポート・レベルがそのまま変更されません。

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型を決定するための**レベル**の論理積で使用します。

E
: 'error' 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厳密な型であると見なされます。 'E' のレポート・レベルは、'error' メッセージだけ表示します。別の方法で 'E' にレポート・レベルを設定するには、**-qflag=e:e** コンパイラー・オプションを指定します。

W
: 'warning' 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、すべての通知メッセージを遮り、警告メッセージまたはエラーのメッセージとして分類されたメッセージだけを表示します。別の方法で 'W' にレポート・レベルを設定するには、**-qflag=w:w** コンパイラー・オプションを指定します。

I
: 'information' 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、診断メッセージの最も厳密でない型と見なします。 'I' のレベルはすべてのメッセージ型を表示します。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。別の方法で 'I' にレポート・レベルを設定するには、 **-qflag=i:i** コンパイラー・オプションを指定します。

enable | disable
: 指定し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
: メッセージ番号が続く、メッセージ番号のプレフィックスを含む ID です。 メッセージ番号の例: CPPC1004

pop
: 直前のレポート・レベルにレポート・レベルがリセットされます。ポップ操作が空のスタックで実行されている場合、レポート・レベルは未変更のまま有効であり、メッセージも生成されません。

例

1. **#pragma info** を指定すると、すべての情報診断を印刷するようにコンパイラーに指示します。プラグマのレポートは、 'W' の重大度または警告メッセージを持つメッセージのみを表示するようにコンパイラーに指示します。この場合、情報診断は何も表示されません。

   ```
   1  #pragma info(all)
   2  #pragma report(level, W)
   ```

2. CPPC1000 がエラー・メッセージの場合、表示されます。別の診断メッセージ型の場合は、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 'disabled' の場合、設定されているレポート・レベルおよび表示されるコードの行の順序に関係なく、診断メッセージは表示されません。

3. 下記の例の行 1 で、初期レポート・レベルが 'I' に設定されているので、分類される診断メッセージの型に関係なくメッセージ CPPC1000 が表示されます。行 3 は、新しいレポート・レベル 'E' が設定され、重大度レベルが 'E' であるメッセージのみが表示されることを示します。行 3 の直後に、現行レベル 'E' が 'ポップ' であり、'I' にリセットされます。

   ```
   1  #pragma report(level, I)
   2  #pragma report(enable, CPPC1000)
   3  #pragma report(level, E)
   4  #pragma report(pop)
   ```

関連参照

287 ページの『汎用プラグマ』
109 ページの『flag』

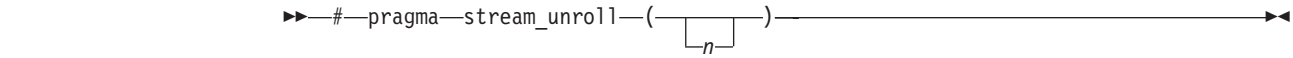
#pragma stream_unroll

C C++

説明

for ループに含まれるストリームを複数のストリームに分割します。

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、 値 *n* は正の整数定数式です。C++ プログラムでは、値 *n* は正のスカラー整数またはコンパイル時定数の初期設定式です。 アンロール係数として 1 を指定すると、アンロールが使用不可になります。 *n* を指定しない場合、および **-qhot**、**-qsmp**、または **-O4** 以上が指定された場合は、最適化プログラムにより、それぞれのネストされたループごとに該当するアンロール係数が決定されます。

注

ストリーム・アンロールを使用可能にするには、**-O3** でも **-qipa=level=2** でも十分ではありません。さらに追加して **-qhot** または **-qsmp** を指定するか、あるいは最適化レベル **-O4** 以上を使用する必要があります。

ストリーム・アンロールを発生させるには、**#pragma stream_unroll** ディレクティブが **for** ループより前に指定された最後のプラグマでなければなりません。同じ **for** に対して **#pragma stream_unroll** を複数回指定したり、このプラグマを他のループ・アンロール・プラグマ (**unroll**、**nounroll**、**unrollandfuse**、 **nounrollandfuse**) と組み合わせて使用した場合も、XL C から警告が出されます。 XL C++ は、同じ **for** ループに指定された複数のループ・アンロール・プラグマのうち、最後のものを除くすべてのプラグマを、警告を出すことなく無視します。

#pragma stream_unroll の後に **#pragma block_loop** がある場合、ストリーム・アンロールは行われません。この状態でも、XL C はまた重大エラーを出します。XL C++ では診断は出されませんが、最適化プログラムがストリーム・アンロール最適化を適用しない可能性があります。

ストリーム・アンロールは、特定の最適化オプションを指定してコンパイルを行った場合にも抑止されます。オプション **-qstrict** が有効な場合、ストリーム・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・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

以下に、**#pragma stream_unroll** を使用してパフォーマンスを向上させる方法の例を示します。

```
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];
}
```

アンロール係数 4 は、以下のように、反復の数を n から n/4 まで削減します。

```
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];
}
```

読み取りおよび保管操作は数が増えると、コンパイラーによって決定された多数のストリーム間で分散され、計算時間が削減され、パフォーマンスが向上します。

関連参照

- 272 ページの『unroll』
- 348 ページの『#pragma unroll』
- 350 ページの『#pragma unrollandfuse』

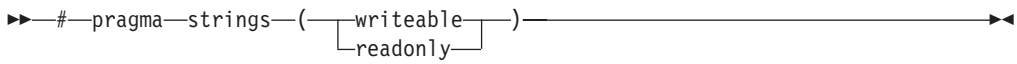
#pragma strings

C C++

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型を設定し、それらのストリングを読み取り専用メモリーまたは読み取り/書き込みメモリーに配置できるかどうかを指定します。

構文



注

C なんらかの書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、デフォルトでは読み取り専用になります。

C++ なんらかの書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、デフォルトでは読み取り専用になります。

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)
```

関連参照

- 228 ページの『roconst』

#pragma unroll

C C++

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ループをアンロールするために使用し、プログラムのパフォーマンスを向上させるのに役立つものです。

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、 値 *n* は正の整数定数式です。C++ プログラムでは、値 *n* は正のスカラー整数またはコンパイル時定数の初期設定式です。 アンロール係数 1 はアンロールを使用不可にします。*n* を指定しないで、**-qhot**、**-qsmp**、または **-O4** 以上を指定した場合、最適化プログラムにより、それぞれのネストされたループごとに、該当するアンロール係数が決定されます。

注

#pragma unroll および **#pragma nounroll** ディレクティブは、影響されるループの直前になければなりません。

特定ループに対して、これらのディレクティブのうちの 1 つのみ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なければなりません。

- ループ・カウンター変数は 1 つのみで、その変数に対する増分ポイントは 1 つ、終了変数は 1 つでなければなりません。これらは、ループ・ネストの任意のポイントで変更することはできません。
- ループは、複数のエントリーおよびエグジット・ポイントを持つことはできません。ループ終了は、ループを終了するための唯一の手段である必要があります。
- ループの依存関係は、「後方重視」であってはなりません。例えば、 A[i][j] = A**[i -1]**[j + 1] + 4) などのステートメントは、ループ内に表示されてはなりません。

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に指示します。 **#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

1. 以下の例では、ループ制御は変更されません。

```
#pragma unroll(2)
while (*s != 0)
{
  *p++ = *s++;
}
```

これを係数 2 でアンロールすると、以下が生成されます。

```
while (*s)
{
  *p++ = *s++;
  if (*s == 0) break;
  *p++ = *s++;
}
```

2. 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];
  }
}
```

関連参照

- 272 ページの『unroll』
- 350 ページの『#pragma unrollandfuse』

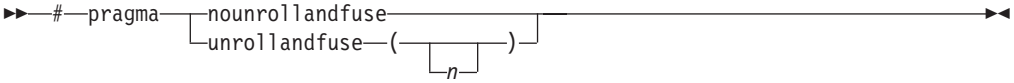
#pragma unrollandfuse

C C++

説明

このプラグマは、ネストされた **for** ループ上で、アンロールおよびフューズ操作を試行するようにコンパイラーに指示します。

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、 値 *n* 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定数の初期設定式です。 *n* を指定しない場合、および **-qhot**、**-qsmp**、または **-O4** 以上を指定した場合、最適化プログラムにより、それぞれのネストされたループごとに該当するアンロール係数が決定されます。

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にのみ適用されます。

- ループ・カウンター変数は 1 つのみで、その変数に対する増分ポイントは 1 つ、終了変数は 1 つでなければなりません。これらは、ループ・ネストの任意のポイントで変更することはできません。
- ループは、複数のエントリーおよびエグジット・ポイントを持つことはできません。ループ終了は、ループを終了するための唯一の手段である必要があります。
- ループの依存関係は、「後方重視」であってはなりません。例えば、 A[i][j] = A**[i -1]**[j + 1] + 4) などのステートメントは、ループ内に表示されてはなりません。

ループのアンロールを発生させるには、**for** ループの前に **#pragma unrollandfuse** ディレクティブが配置されている必要があります。 **#pragma unrollandfuse** は、最内部の **for** ループに対して指定してはなりません。

#pragma unrollandfuse を複数回指定したり、このディレクティブを同じ **for** ループの **block_loop**、**nounrollandfuse**、**nounroll**、**unroll**、または **stream_unroll** ディレクティブと組み合わせて使用しないでください。

#pragma nounrollandfuse 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に指示します。

例

1. 以下の例では、**#pragma unrollandfuse** ディレクティブがループの本体を複製し、フューズします。これにより、配列 *b* のキャッシュ・ミス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];
    }
}
```

次の **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に考えられる結果を示します。

```
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. ネストされたループ構造体で、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];
        }
    }
}
```

関連参照

- 272 ページの『unroll』
- 348 ページの『#pragma unroll』

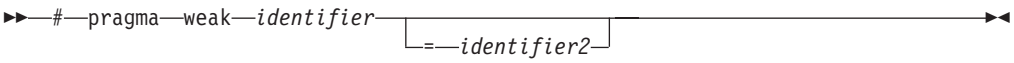
#pragma weak

C C++

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索されないときや、リンク中に複数のシンボルが定義されていることが発見されたときに、リンケージ・エディターはエラー・メッセージを発行できません。

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、未初期化状態のグローバル・データと共に、あるいは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は使用しないようにしてください。

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 *ID*

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* に対する参照では定義された ID 値が使用され、定義されていない場合、*identifier* には値 0 が割り当てられます。

Identifier が **#pragma weak** *identifier* と同じコンパイル単位で定義されている場合、*identifier* は weak 定義として処理されます。*identifier* を使用も宣言もし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ても無視されます。

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である場合、このテンプレート関数を明示的にインスタンス化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に宣言できることとできないことがありますが、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に宣言されると、*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* が関数のときは、*identifier* は *identifier2* と同じ戻りまたは引き数の型がなければなりません。

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指示するときは、*identifier* と *identifier2* はその関数のマングル名を使用して指定しなければなりません。 C++ 関数がテンプレート関数である場合、このテンプレート関数を明示的にインスタンス化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。

コンパイラーは **#pragma weak** を無視し、弱い *identifier* が定義されているときには重大エラー・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
```

実行可能ファイルを作成するために Compilation Unit 1 のみがコンパイルされる場合、 ID `foo` が定義され、値 `0` が割り当てられます。実行から得られる出力は、次のストリングになります: `"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
```

並列処理を制御するプラグマ

このページの #pragma ディレクティブによって、ユーザー・プログラムでの並列処理をコンパイラーが処理する方法を制御することができます。

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 C C++	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。
#pragma omp flush	並列領域内のすべてのスレッドがメモリー内の同じビューの指定したオブジェクト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。
#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。

関連概念

13 ページの『プログラムの並列化』

関連タスク

37 ページの『プラグマを使用した並列処理の制御』

関連参照

387 ページの『並列処理のための OpenMP ランタイム・オプション』
389 ページの『並列処理に使用する組み込み関数』

OpenMP 仕様の完全な情報については、以下を参照してください。

- OpenMP Web サイト
- OpenMP 仕様

#pragma omp atomic

C C++

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

```
#pragma omp atomic
    <statement_block>
```

ここで、*statement* は、以下に続く形式のいずれかを採るスカラー型の式ステートメントです。

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

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 atomic です。 *expr* の評価は、atomic 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。

並列で更新できる、また、競合状態の対象となり得るオブジェクトは、 **omp atomic** ディレクティブで保護されているはずです。

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;
```

関連参照

355 ページの『並列処理を制御するプラグマ』

#pragma omp parallel

説明

omp parallel ディレクティブは、選択したコードのセグメントを並列化するようにコンパイラーに明示的に指示します。

構文

```
#pragma omp parallel [clause[[,] clause] ...]
<statement_block>
```

ここで、*clause* は、次のいずれかです。

if (*exp*)
: **if** 引き数を指定したときは、 *exp* が表したスカラー式が実行時に非ゼロ値と評価された場合に限り、プログラム・コードが並列に実行されます。 **if** 文節は 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、並列領域用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である場合、 *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* 内のデータ変数のスコープがすべてのスレッドに渡って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** 文節は、 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*)** 文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・スコープ文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 文節内で指定するデータ変数は、それぞれ、 **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 文節で指定する変数は、以下のとおりでなければなりません。

- 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。

関連参照

355 ページの『並列処理を制御するプラグマ』
360 ページの『#pragma omp for』
365 ページの『#pragma omp parallel for』
368 ページの『#pragma omp parallel sections』

#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指示します。

構文

```
#pragma omp for [clause[[,] clause] ...]
<for_loop>
```

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** 文節で指定する変数は、以下のとおりでなければなりません。 • 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 ループの反復が、サイズが **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 実行時にスケジューリングの方針が判別されます。OMP_SCHEDULE 環境変数を使用して、スケジューリング型およびチャンク・サイズを設定します。

static ループの反復が、サイズが **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* としても知られています。

static,*n* ループの反復が、サイズが *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* としても知られています。

static,1 ループの反復が、サイズが 1 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。

このスケジューリング方針は、*巡回スケジューリング* としても知られています。

nowait

この文節は、**for** ディレクティブ終了時の暗黙の **barrier** 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** 文節が、所定の **for** ディレクティブ上に現れるのは、1 回のみです。

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	は、次の形式を取ります。	*iv = b* *integer-type iv = b*
exit_cond	は、次の形式を取ります。	*iv <= ub* *iv < ub* *iv >= ub* *iv > ub*
incr_expr	は、次の形式を取ります。	*++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** ループ内のいずれの個所でも変更されてい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。これらの式の評価中は、同期は実行されません。また、評価済みの副次作用は、結果として、不確定値になる場合があります。

注

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 文節が指定されていない限り、 **for** ループの終了時に暗黙の barrier 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りません。また、**break** ステートメントで終了してはなりません。
- 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** 文節は、1 つのみです。
- *n* の値 (チャンク・サイズ) は、並列領域のすべてのスレッドについて同一でなければなりません。

関連参照

355 ページの『並列処理を制御するプラグマ』
365 ページの『#pragma omp parallel for』

#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

```
#pragma omp ordered
    statement_block
```

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** 文節を含んでいる **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 2 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。

関連参照

355 ページの『並列処理を制御するプラグマ』
360 ページの『#pragma omp for』
365 ページの『#pragma omp parallel for』

#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

```
#pragma omp parallel for [clause[[,] clause] ...]
<for_loop>
```

注

nowait 文節を除き、 **omp parallel** および **omp for** 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 for** ディレクティブに適用されます。

関連参照

355 ページの『並列処理を制御するプラグマ』
360 ページの『#pragma omp for』
358 ページの『#pragma omp parallel』

#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

```
#pragma omp sections [clause[ clause] ...]
   {
     [#pragma omp section]
          statement-block
     [#pragma omp section]
          statement-block
       .
       .
       .
   }
```

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。

num_threads (*int_exp*)
: *int_exp* の値は、並列領域用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である場合、*int_exp* は使用されるスレッドの最大数を指定します。

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 文節で指定する変数は、以下のとおりでなければなりません。

  - 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** ディレクティブ終了時の暗黙の **barrier** 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** 文節が、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** 文節が指定されていない限り、barrier 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。

関連参照

355 ページの『並列処理を制御するプラグマ』
368 ページの『#pragma omp parallel sections』

#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

```
#pragma omp parallel sections [clause[[,] clause] ...]
   {
     [#pragma omp section]
          statement-block
     [#pragma omp section]
          statement-block
       .
       .
       .
     ]
   }
```

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。

関連参照

355 ページの『並列処理を制御するプラグマ』
358 ページの『#pragma omp parallel』
366 ページの『#pragma omp section、#pragma omp sections』

#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

```
#pragma omp single [clause[[,] clause] ...]
   statement_block
```

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。

また、**private** 文節内の変数は、同じ **omp single** ディレクティブ用の **copyprivate** 文節内に出現することはできません。

copyprivate (*list*)
list 内で指定された変数の値を、チーム内のある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** ディレクティブに関連付けられた構造化ブロックの実行の後、かつ、構成の終了時にすべてのスレッドがバリアから離れる前に行われます。チーム内の他のすべてのスレッドの場合、*list* 内の各変数は、構造化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。この文節に対する使用制限は、以下のとおりです。

- **copyprivate** 文節内の変数は、同じ **omp single** ディレクティブ用の **private** または **firstprivate** 文節内に出現することはできません。
- **copyprivate** 文節を持つ **omp single** ディレクティブが並列領域の動的範囲内で検出された場合、 **copyprivate** 文節内で指定された変数はすべて、囲んでいるコンテキスト内で private でなければなりません。
- 並列領域の動的範囲内の **copyprivate** 文節内で指定された変数は、囲んでいるコンテキスト内で private でなければなりません。
- **copyprivate** 文節内で指定された変数には、アクセス可能かつ、あいまいさのないコピー割り当て演算子がなければなりません。
- **copyprivate** 文節は、 **nowait** 文節と共に使用してはいけません。

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 文節内の変数は、同じ **omp single** ディレクティブ用の **copyprivate** 文節内に出現することはできません。

nowait
この文節は、**single** ディレクティブ終了時の暗黙の **barrier** を回避するために使用します。**nowait** 文節が、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** 文節は、 **copyprivate** 文節と共に使用してはいけません。

注

nowait 文節が指定されていない限り、暗黙の barrier 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。

関連参照

355 ページの『並列処理を制御するプラグマ』

#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

```
#pragma omp master
   statement_block
```

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。

関連参照

355 ページの『並列処理を制御するプラグマ』

#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

```
#pragma omp critical [(name)]
   statement_block
```

ここで、*name* は、オプションでクリティカル領域を識別するのに使用することができます。クリティカル領域を命名している ID には、外部結合があり、通常 ID が使用しているネーム・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クリティカル領域を実行しなくなるまで、指定された名前で識別されたクリティカル領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、未指定の同じ名前にマップされます。

関連参照

355 ページの『並列処理を制御するプラグマ』

#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

```
#pragma omp barrier
```

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。例を以下に示します。

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

関連参照

355 ページの『並列処理を制御するプラグマ』

#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の同じビューの指定したオブジェクト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

```
#pragma omp flush [ (list) ]
```

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表示されます。

- **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  */
```

関連参照

355 ページの『並列処理を制御するプラグマ』
373 ページの『#pragma omp barrier』
372 ページの『#pragma omp critical』
360 ページの『#pragma omp for』
358 ページの『#pragma omp parallel』
365 ページの『#pragma omp parallel for』
368 ページの『#pragma omp parallel sections』
366 ページの『#pragma omp section、#pragma omp sections』
369 ページの『#pragma omp single』

#pragma omp threadprivate

説明

omp threadprivate ディレクティブは、スレッドに private になる名前付きのファイル・スコープ、名前スペース・スコープ、または静的ブロック・スコープ変数を作成します。

構文

```
#pragma omp threadprivate (list)
```

ここで、*list* は、変数のコンマ区切りのリストです。

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、オブジェクトが使用前に変更された場合、振る舞いは指定解除されます。

スレッドは、**omp threadprivate** データ変数の別のスレッドのコピーを参照してはなりません。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・スコープ変数に対して適用でき、字句ブロック内に出現してそれらのブロック・スコープ変数を参照することができる。このディレクティブは、ネストされたスコープ内ではなく、変数のスコープ内にあり、かつそのリスト内の変数に対するすべての参照の前に出現しなければなりません。
- 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* の字句は、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* 内で指定されたデータ変数は、 **copyin**、**copyprivate**、**if**、**num_threads**、および **schedule** 文節以外の文節に出現してはならない。
- **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。

関連参照

355 ページの『並列処理を制御するプラグマ』

コンパイラー・モードおよびプロセッサー・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に適合するように最適化することができます。

- ターゲット・プロセッサーの非常に広範囲に渡る可能な選択
- 指定されたプロセッサー・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

一般的に、オプションでは以下が行われます。

- **-q32** は、32 ビット実行モードを選択します。
- **-q64** は、64 ビット実行モードを選択します。
- **-qarch** は、命令コードの生成対象として、一般的なファミリーのプロセッサー・アーキテクチャーを選択します。特定の **-qarch** 設定は、選択した **-qarch** 設定に応じてコンパイラーが生成するすべての 命令をサポートするシステム上でのみ 稼働す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します。 **-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動作している場合にコードのパフォーマンスにのみ影響しますが、コードがどこで動作するかの判別はしません。

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。

以下のテーブルに、選択されたいくつかのプロセッサー、およびさまざまなフィーチャーを、サポートされるもの、またはサポートされないものも含めて示します。

プロセッサー	グラフィックス・サポート	sqrt サポート	64 ビット・サポート
rs64b	あり	あり	あり
rs64c	あり	あり	あり
pwr3	あり	あり	あり
pwr4	あり	あり	あり
pwr5	あり	あり	あり

多様な種類の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/または **-qtune** コンパイラー・オプションを選択します。以下をコンパイルに使用したコードの場合:

- **-qarch=pwr4** を使用したコードは、POWER4 マシン上でのみ稼働する。
- **-qarch=pwr5** を使用したコードは、POWER5 マシン上でのみ稼働する。
- **-qarch=ppc** を使用したコードは、いずれの 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、機能的に同等の PowerPC マシンであればいずれにおいても稼働する。以下の表にある例では、**-qarch=pwr3** を使用してコンパイルされたコードも **rs64c**。

特定のプロセッサー用に最適化されたコードを生成したい場合の、**-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表 で示します。

-qarch と -qtune の受け入れ可能な組み合わせ			
-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**
ppc64v	_ARCH_PPCV	ppc970	auto ppc970

関連参照
- 30 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) コンパイル用コンパイラー・オプションの指定』
- 52 ページの『32、64』
- 63 ページの『arch』
- 269 ページの『tune』

コンパイラー・メッセージ

このセクションでは、コンパイラーがコンパイル・エラーを記述するのに使用するいくつかの基本的レポート・メカニズムについて概説します。

- 『メッセージ重大度レベルとコンパイラー応答』
- 380 ページの『コンパイラー戻りコード』
- 381 ページの『コンパイラー・メッセージ・フォーマット』

メッセージ重大度レベルとコンパイラー応答

以下の表では、各レベルのメッセージ重大度に関連したコンパイラー応答を示します。

文字	重大度	コンパイラーの応答
I	通知	コンパイルは継続します。このメッセージは、コンパイルと途中で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。このメッセージは、疑わしい条件および意図したものではない条件を報告します。プログラムは、書かれたとおりに実行します。
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することのできるエラー条件が存在しますが、プログラムは正しく実行でき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	C 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。 • 回復不能エラーが解決されない場合は、IBM サービス技術員にメッセージを報告してください。

関連概念

10 ページの『コンパイラー・メッセージおよびリスト情報』

関連参照

『コンパイラー戻りコード』
381 ページの『コンパイラー・メッセージ・フォーマット』
124 ページの『halt』
190 ページの『maxerr』
126 ページの『haltonmsg』

コンパイラー戻りコード

コンパイルの終了時に、コンパイラーは、以下の任意の条件のもとで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最高重大度レベルが、 **-qhalt** コンパイラー・オプションの設定値よりも小さい、さらに、エラーの数が **-qmaxerr** コンパイラー・オプションで設定した限界値に達していない。
- **-qhaltonmsg** コンパイラー・オプションで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	halt コンパイラー・オプションの設定値よりも高い重大度レベルのエラーが検出されました。
40	オプション・エラーまたは回復不能エラーが検出されました。
41	構成ファイル・エラーが検出されました。
250	メモリー不足のエラーが検出されました。 **xlc++** コマンドで、使用するためのメモリーをさらに割り振ることはできません。
251	シグナルで受信されたエラーが検出されました。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	ファイルを見つけられないエラーが検出されました。
253	入出力エラーが検出されました。ファイルを読み取ったり、ファイルに書き込むことができません。
254	fork エラーが検出されました。新規プロセスを作成することができません。
255	プロセスの実行中にエラーが検出されました。

注: 実行時エラーの戻りコードも表示される可能性があります。

関連概念

10 ページの『コンパイラー・メッセージおよびリスト情報』

関連参照

379 ページの『メッセージ重大度レベルとコンパイラー応答』
381 ページの『コンパイラー・メッセージ・フォーマット』

124 ページの『halt』
190 ページの『maxerr』
126 ページの『haltonmsg』

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) のとき、診断メッセージのフォーマットは次のとおりです。

```
"file", line line_number.column_number: 15dd-nnn (severity) text.
```

ここで、

file	エラーのある C または C++ ソース・ファイル名です。
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 コンパイラーに固有のメッセージ **06** - C コンパイラーに固有のメッセージ **40** - C++ コンパイラーに固有のメッセージ **86** - プロシージャー間分析 (IPA) に固有のメッセージ
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	エラーの内容を説明するメッセージです。

-qsrcmsg オプションが指定されているとき、診断メッセージのフォーマットは次のとおりです。

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

ここで、**x** はフィンガー行のフィンガーを示す文字です。

関連概念

10 ページの『コンパイラー・メッセージおよびリスト情報』

関連参照

379 ページの『メッセージ重大度レベルとコンパイラー応答』
380 ページの『コンパイラー戻りコード』
124 ページの『halt』
190 ページの『maxerr』
126 ページの『haltonmsg』

並列処理のサポート

本節では、並列処理の制御に使用する環境変数と組み込み関数についての情報を記載します。本節には以下のトピックがあります。

並列処理のためのランタイム・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しなければならず、以下の形式の基本構文を使用します。

並列化ランタイム・オプションは、OMP 環境変数を使用して指定することもできます。 OMP および XLSMPOPTS に固有の環境変数で指定されるランタイム・オプションが競合する場合は、 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・モードを使用して起動しなければなりません。

XLSMPOPTS 環境変数に応じた実行時オプション設定を、カテゴリー別に分類して以下に示します。

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、スケジューリング・アルゴリズムを明示的に割り当てられていないループ用に使用するスケジューリング・アルゴリズムを指定します。 *algorithm* の有効なオプションは、以下のとおりです。 • 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は、存在するプロセッサー数より多くのスレッドを使用するとパフォーマンスがかなり向上します。このオプションでは、プログラムを実行するために使用するユーザー・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。この場合は、作成するスレッドの数に *num* を設定してください。 *num* のデフォルト値は 0 です。
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。

パフォーマンス調整オプション

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、処理を完了すると、新しい処理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、作業待ち行列の完全スキャンが 1 回実行されます。拡張作業待ち状態によって特定のアプリケーションの応答性を高くすることができますが、定期的にスキャンして他のアプリケーションからの要求に譲歩するようにスレッドに指示しない限り、システム全体としての応答が低下する場合もあります。 **spins** および **yields** の両方を 0 に設定することによって、ベンチマークを目的として完全な作業待ち状態にすることができます。 *num* のデフォルト値は 100 です。
yields=*num*	*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。これによって、システムの使用効率は向上しますが、アプリケーションに余分なシステム・オーバーヘッドが加わります。 *num* のデフォルト値は 100 です。
delays=*num*	*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、次回はループを順次実行するべきか並列に実行する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。32 を超える *num* の値は 32 に変更されます。
parthreshold=*mSec*	*mSec* は、ループを順次実行しなければならないとする実行時間の上限をミリ秒単位で指定します。 *mSec* は、小数部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、小数部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。

関連概念

13 ページの『プログラムの並列化』
13 ページの『OpenMP ディレクティブ』

関連参照

387 ページの『並列処理のための OpenMP ランタイム・オプション』
389 ページの『並列処理に使用する組み込み関数』

OpenMP 仕様の完全な情報については、以下を参照してください。
OpenMP Web サイト www.openmp.org
OpenMP Specification www.openmp.org/specs

並列処理のための OpenMP ランタイム・オプション

並列処理に影響を及ぼす OpenMP ランタイム・オプションは、OMP 環境変数を指定して設定します。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ない場合、デフォルト設定が使用されます。

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・モードを使用して起動しなければなりません。

OpenMP ランタイム・オプションは、以下で説明する別々の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** ディレクティブでスケジューリング・アルゴリズムを明示的に割り当てられないループに対して使用するスケジューリング・アルゴリズムを指定します。例を以下に示します。

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

algorithm の有効なオプションは、以下のとおりです。

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

チャンク・サイズを *n* で指定する場合、 *n* の値は 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は、存在するプロセッサー数より多くのスレッドを使用するとパフォーマンスがかなり向上します。このオプションでは、プログラムを実行するために使用するユーザー・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。

いくつかの **#pragma omp** ディレクティブで使用可能な **num_threads** 文節を使用して、指定された parallel section の OMP_NUM_THREADS の設定を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の場合、ネストされた並列領域は直列化され、現在のスレッドで稼働します。

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 は何の影響も及ぼさず、**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 (非精密 OMP モードでのみ) は、ネストされた並列領域は、追加の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。

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。

関連概念

13 ページの『プログラムの並列化』
13 ページの『OpenMP ディレクティブ』

関連参照

355 ページの『並列処理を制御するプラグマ』
『並列処理に使用する組み込み関数』

OpenMP 仕様の完全な情報については、以下を参照してください。

- OpenMP Web サイト (www.openmp.org)
- OpenMP Specification (www.openmp.org/specs)

並列処理に使用する組み込み関数

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。

関数プロトタイプ	説明
int omp_get_num_threads(void);	この関数は、この関数が呼び出されている並列領域を実行しているチームに現在あるスレッドの数を戻します。
void omp_set_num_threads(int *num_threads*);	この関数は 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するスレッド数を指定します。値 num_threads は正の整数でなければなりません。 num_threads 文節が存在し、並列領域のために適用される場合、これは、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るスレッド数を置き換えます。後続の並列領域は、この影響を受けません。
int omp_get_max_threads(void);	この関数は、 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。

関数プロトタイプ	説明
int omp_get_thread_num(void);	この関数は、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 以上 omp_get_num_threads()-1 以下となります。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);	この関数は、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);	この関数は、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);	この関数は、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。
int omp_get_dynamic(void);	この関数は、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);	この関数は、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。
int omp_get_nested(void);	この関数は、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手段のみを提供します。それぞれの関数は、以降の呼び出しで使用するために、パラメーター *lock* に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_);
	これらの関数では、指定されたロック変数 *lock* が初期化されません。

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。

関数プロトタイプ	説明
double omp_get_wtime(void);	固定された開始時刻から経過した時間を戻します。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、プログラム実行中は一定に保たれます。
double omp_get_wtick(void);	時計が音を刻む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、何の影響力も持たず、**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、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。

関連概念

13 ページの『プログラムの並列化』

関連タスク

37 ページの『プラグマを使用した並列処理の制御』

関連参照

355 ページの『並列処理を制御するプラグマ』
387 ページの『並列処理のための OpenMP ランタイム・オプション』
397 ページの『付録 B. 組み込み関数』

第 4 部 付録

付録 A. 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かれています。言語フィーチャーに関連したマクロ、コンパイラーに関連したマクロ、および Linux プラットフォームに関連するマクロがあります。

言語仕様で必須とされているものを含む、言語フィーチャーに関連したマクロについては、「*XL C/C++ ランゲージ・リファレンス*」の『プリプロセッサー・ディレクティブ』のセクションを参照してください。

XL コンパイラーを指示するマクロ

XL 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ます。

事前定義マクロ名	説明
__IBMC__	C 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 XL C コンパイラーのレベルを指示します。
__IBMCPP__	C++ 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 XL C++ コンパイラーのレベルを指示します。
__xlc__	C バージョン、リリース、モディフィケーション、およびフィックス・レベルを表すストリングとして XL C コンパイラーのレベルを指示します。
__xlC__	C++ 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 XL C++ コンパイラーのレベルを指示します。 XL C コンパイラーの使用により、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するマクロ

プラットフォーム間でのアプリケーションの移植を容易にするために、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
_ARCH_PPC	プロセッサー・アーキテクチャーが PowerPC である場合、1 に定義されます。
_ARCH_PPC64	プロセッサー・アーキテクチャーが PowerPC である場合、1 に定義されます。
__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)** が有効である場合、1 に定義されます。オプション **-qchars=signed** または **#pragma chars(signed)** が有効な場合、このマクロは未定義となります。
__ELF__	ELF オブジェクト・モデルが有効であることを指示するために、このプラットフォームで 1 に定義されます。
__EXCEPTIONS	C++ **-qeh** オプションが有効な場合、 1 に定義されます。それ以外の場合、これは定義されません。

事前定義マクロ名	説明
__GXX_WEAK__	C の場合は未定義。 C++ の場合、このマクロは g++ V3.3 では 0 に定義され、g++ V3.5 では 1 に定義されます。
__HOS_LINUX__	ホスト・オペレーティング・システムが Linux である場合、1 に定義されます。それ以外の場合、これは定義されません。
__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** および **-O** が設定されている場合、1 に定義されます。それ以外の場合、これは定義されません。
__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。それ以外の場合、これは定義されません。
__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。それ以外の場合、これは定義されません。
__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。このプラットフォームでは、マクロは **long unsigned int** として定義されます。 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 **long int** として定義されます。コンパイル・モードは、**-q32** および **-q64** オプションによって制御されます。
__TOS_LINUX__	ターゲット・オペレーティング・システムが Linux である場合、1 に定義されます。それ以外の場合、これは定義されません。
__unix	すべての UNIX 型のプラットフォームで 1 に定義されます。それ以外の場合、これは定義されません。
__unix__	すべての UNIX 型のプラットフォームで 1 に定義されます。それ以外の場合、これは定義されません。

付録 B. 組み込み関数

コンパイラーは、より効率的なプログラムの作成を援助するため、ユーザーが組み込み関数を選択できるようにします。本節では、使用可能な各種の組み込み関数を要約します。

389 ページの『並列処理に使用する組み込み関数』で説明している追加の組み込み関数を検索することもできます。

名前	プロトタイプ	説明
__alignx	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知らせます。*位置合わせ* は、ゼロより大きい値を持つ正の定数整数で、2 の累乗でなければなりません。
__bcopy	void __bcopy(char *, char *, int);	ブロック・コピー
__bzero	void __bzero(void *, size_t);	ブロック・ゼロ
__check_lock_mp	unsigned int __check_lock_mp (const int* *addr*, int *old_value*, int *new_value*);	マルチプロセッサー・システムでのロックを検査します。 条件付きで、シングル・ワード変数をアトミックに更新します。*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) が戻された場合は、シングル・ワード変数が元の値と同じであったこと、さらにこの変数が新規の値に設定されたことを示す。 2. true (真) が戻された場合は、シングル・ワード変数が元の値と同じでなかったこと、さらにこの変数が変更されないままになっていることを示す。

名前	プロトタイプ	説明
__check_lockd_mp	unsigned int __check_lockd_mp (const long long int* *addr*, long long int *old_value*, long long int *new_value*);	マルチプロセッサー・システムでのロック・ダブルワードを検査します。 条件付きで、ダブルワード変数をアトミックに更新します。*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) が戻された場合は、ダブルワード変数が元の値と同じであったこと、さらにこの変数が新規の値に設定されたことを示す。 2. true (真) が戻された場合は、ダブルワード変数が元の値と同じでなかったこと、さらにこの変数が変更されないままになっていることを示す。

名前	プロトタイプ	説明
__check_lock_up	unsigned int __check_lock_up (const int* *addr*, int *old_value*, int *new_value*);	単一プロセッサー・システムでのロックを検査します。 条件付きで、シングル・ワード変数をアトミックに更新します。*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) が戻された場合は、シングル・ワード変数が元の値と同じであったこと、さらにこの変数が新規の値に設定されたことを示す。 • true (真) が戻された場合は、シングル・ワード変数が元の値と同じでなかったこと、さらにこの変数が変更されないままになっていることを示す。

名前	プロトタイプ	説明
__check_lockd_up	unsigned int __check_lockd_up (const long long int* *addr*, long long int *old_value*, int long long *new_value*);	単一プロセッサー・システムでのロック・ダブルワードを検査します。 条件付きで、ダブルワード変数をアトミックに更新します。*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) が戻された場合は、ダブルワード変数が元の値と同じであったこと、さらにこの変数が新規の値に設定されたことを示す。 • true (真) が戻された場合は、ダブルワード変数が元の値と同じでなかったこと、さらにこの変数が変更されないままになっていることを示す。
__clear_lock_mp	void __clear_lock_mp (const int* *addr*, int *value*);	マルチプロセッサー・システムでのロックをクリアします。 アドレス *addr* にあるシングル・ワード変数に、 *value* をアトミックに保管します。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。
__clear_lockd_mp	void __clear_lockd_mp (const long long int* *addr*, long long int *value*);	マルチプロセッサー・システムでのロック・ダブルワードをクリアします。 アドレス *addr* にあるダブルワード変数に、 *value* をアトミックに保管します。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。

名前	プロトタイプ	説明
__clear_lock_up	void __clear_lock_up (const int* *addr*, int *value*);	単一プロセッサー・システムでのロックをクリアします。 アドレス *addr* にあるシングル・ワード変数に、 *value* をアトミックに保管します。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。
__clear_lockd_up	void __clear_lockd_up (const long long int* *addr*, long long int *value*);	ユニプロセッサー・システムでのロック・ダブルワードをクリアします。 アドレス *addr* にあるダブルワード変数に、 *value* をアトミックに保管します。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。
__cntlz4	unsigned int __cntlz4(unsigned int);	先行ゼロ・カウント。4 バイト整数。
__cntlz8	unsigned int __cntlz8(unsigned long long);	先行ゼロ・カウント。8 バイト整数。
__cnttz4	unsigned int __cnttz4(unsigned int);	後続ゼロ・カウント。4 バイト整数。
__cnttz8	unsigned int __cnttz8(unsigned long long);	後続ゼロ・カウント。8 バイト整数。
__dcbt( )	void __dcbt (void *);	データ・キャッシュ・ブロック・タッチ。 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。
__dcbz( )	void __dcbz (void *);	ゼロに設定されたデータ・キャッシュ・ブロック。 データ・キャッシュ内の指定のアドレスをゼロ (0) に設定します。
__eieio	void __eieio(void);	すでに組み込まれている **__iospace_eieio** のもう 1 つの名前です。 コンパイラーは、**__eieio** を組み込まれているものとして認識します。名前以外は、すべてが **__iospace_eieio** とまったく同じです。 **__eieio** には、対応する PowerPC の命令名との一貫性があります。
__exp	double __exp(double);	指数値を戻します。
__fabs	double __fabs(double);	絶対値を戻します。

名前	プロトタイプ	説明
__fabss	float __fabss(float);	短精度浮動小数点の絶対値を戻します。
__fcfid	double __fcfid (double);	整数ダブルワードからの浮動変換です。 64 ビット符号付き固定小数点オペランドが、倍精度浮動小数点に変換されます。
__fctid	double __fctid (double);	整数ダブルワードへの浮動変換です。 $FPSCR_{RN}$ (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) で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が、64 ビット符号付き固定小数点整数に変換されます。
__fctidz	double __fctidz (double);	「ゼロの方向に丸める」による、整数ダブルワードへの浮動変換です。 浮動小数点オペランドが、丸めモード「ゼロの方向に丸める」を使用して、 64 ビット符号付き固定小数点整数に変換されます。
__fctiw	double __fctiw (double);	整数ワードへの浮動変換です。 $FPSCR_{RN}$ (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) で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が、32 ビット符号付き固定小数点整数に変換されます。
__fctiwz	double __fctiwz (double);	「ゼロの方向に丸める」による、整数ワードへの浮動変換です。 浮動小数点オペランドが、丸めモード「ゼロの方向に丸める」を使用して、 32 ビット符号付き固定小数点整数に変換されます。
__fmadd	double __fmadd (double, double, double);	浮動小数点の乗算と加算
__fmadds	float __fmadds (float, float, float);	浮動小数点の乗算と加算 short
__fmsub	double __fmsub(double, double, double);	浮動小数点の乗算と減算
__fmsubs	float __fmsubs (float, float, float);	浮動小数点の乗算と減算
__fnabs	double __fnabs(double);	負の浮動小数点の絶対値
__fnabss	float __fnabss(float);	負の浮動小数点の絶対値
__fnmadd	double __fnmadd(double, double, double);	負の浮動小数点の乗算と加算
__fnmadds	float __fnmadds (float, float, float);	負の浮動小数点の乗算と加算

名前	プロトタイプ	説明
__fnmsub	double __fnmsub(double, double, double);	負の浮動小数点の乗算と減算
__fnmsubs	float __fnmsubs (float, float, float);	__fnmsubs (*a*, *x*, *y*) = [- (*a* * *x* - *y*)]
__fre	double fre (double);	浮動小数点の逆数 __fre (*x*) = [(estimate of) 1.0/*x*] (POWER5 プロセッサーのみ)
__fres	float __fres (float);	浮動小数点の逆数 __fres (*x*) = [1.0/*x* (の概算)]
__frsqrte	double __frsqrte (double);	浮動小数点の逆数平方根 __frsqrte (*x*) = [1.0/sqrt(*x*) (の概算)]
__frsqrtes	float __frsqrtes (float);	浮動小数点の逆数平方根 __frsqrtes (*x*) = [(estimate of) 1.0/sqrt(*x*)] (POWER5 プロセッサーのみ)
__fsel	double __fsel (double, double, double);	浮動小数点の選択 if (*a* >= 0.0) then __fsel (*a*, *x*, *y*) = *x*; それ以外の場合 __fsel (*a*, *x*, *y*) = *y*
__fsels	float __fsels (float, float, float);	浮動小数点の選択 if (*a* >= 0.0) then __fsels (*a*, *x*, *y*) = *x*; それ以外の場合 __fsels (*a*, *x*, *y*) = *y*
__fsqrt	double __fsqrt (double);	浮動小数点の平方根 __fsqrt (*x*) = *x* の平方根
__fsqrts	float __fsqrts (float);	浮動小数点の平方根 __fsqrts (*x*) = *x* の平方根
__iospace_eieio	void __iospace_eieio(void);	EIEIO 命令を生成します。
__iospace_lwsync	(equivalent to: void __iospace_lwsync(void);)	lwsync 命令を生成します。
__iospace_sync	(equivalent to: void __iospace_sync(void);)	同期命令を生成します。
__isync	void __isync(void);	前のすべての命令が完了するのを待機した後、プリフェッチされたすべての命令を廃棄して、以降の命令が、前の命令によって確立されたコンテキストの中でフェッチ (または再フェッチ) および実行されるようにします。
__load2r	unsigned short __load2r(unsigned short*);	逆のハーフワード・バイトをロードします。
__load4r	unsigned int __load4r(unsigned int*);	逆のワード・バイトをロードします。

名前	プロトタイプ	説明
__lwsync	(コンパイラーは、**__lwsync** 組み込み関数を認識します。)	すでに組み込まれている **__iospace_lwsync** のもう 1 つの名前です。 コンパイラーは、**__lwsync** 組み込み関数を認識します。名前以外は、すべてが **__iospace_lwsync** とまったく同じです。 **__lwsync** には、対応する PowerPC の命令名との一貫性があります。この関数は、PowerPC 970 プロセッサーによってのみサポートされます。 値は、コンパイル時に既知でなければなりません。
__mfdcr	unsigned __mfdcr(const int);	装置制御レジスターの値を戻します。これは、PowerPC 440 の場合のみ有効です。
__mtdcr	void __mtdcr(const int, unsigned long);	unsigned long 値を使用して、装置制御レジスターの値を設定します。値は、コンパイル時に既知でなければなりません。これは、PowerPC 440 の場合にのみ有効です。
__mfspr	unsigned __mfspr(const int);	特殊目的レジスターの値を戻します。"const int" の値は、コンパイル時に既知でなければなりません。
__mtspr	void __mtspr(const int, unsigned long);	unsigned long 値を使用して、特殊目的レジスターの値を設定します。値は、コンパイル時に既知でなければなりません。
__mtfsb0	void __mtfsb0(unsigned int *bt*);	FPSCR のビット 0 に移動します。 FPSCR のビット *bt* は 0 に設定されます。 *bt* は定数で、0<=*bt*<=31 でなければなりません。
__mtfsb1	void __mtfsb1(unsigned int *bt*);	FPSCR Bit 1 に移動します。 FPSCR のビット *bt* は 1 に設定されます。 *bt* は定数で、0<=*bt*<=31 でなければなりません。

名前	プロトタイプ	説明
__mtfsf	void __mtfsf(unsigned int *flm*, unsigned int *frb*);	FPSCR フィールドに移動します。 *frb* の内容が、 *flm* で指定するフィールド・マスクの制御下の FPSCR に入れられます。フィールド・マスク *flm* は、該当する 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。
__mtfsfi	void __mtfsfi(unsigned int *bf*, unsigned int *u*);	FPSCR フィールドに即時移動します。 *u* の値が、*bf* で指定する FPSCR フィールドに入れられます。 *bf* と *u* は、 0<=*bf*<=7 および 0<=*u*<=15 の定数でなければなりません。
__mulhd	long long int __mulhd(long long int *ra*, long long int *rb*);	高位の符号付きダブルワードの乗算です。 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。
__mulhdu	unsigned long long int __mulhdu(unsigned long long int *ra*, unsigned long long int *rb*);	高位の符号なしダブルワードの乗算です。 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。
__mulhw	int __mulhw(int *ra*, int *rb*);	高位の符号付きワードの乗算です。 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。
__mulhwu	unsigned int __mulhwu(unsigned int *ra*, unsigned int *rb*);	高位の符号なしワードの乗算です。 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。

名前	プロトタイプ	説明
__parthds	int __parthds(void);	parthds ランタイム・オプションの値を戻します。 parthds オプションがユーザーによって明示的に設定されていない場合、関数は、ランタイム・ライブラリーによって設定されたデフォルト値を戻します。プログラムのコンパイル時に -qsmp コンパイラー・オプションが指定されなかった場合、この関数は、選択されたランタイム・オプションに関係なく 1 を戻します。
__popcnt4	int __popcnt4(unsigned int);	32 ビット整数用に設定されたビットの数を戻します。
__popcnt8	int __popcnt8(unsigned long long);	64 ビット整数用に設定されたビットの数を戻します。
__poppar4	int __poppar4(unsigned int);	32 ビット整数内に奇数のビットが設定されている場合、1 を戻します。それ以外の場合は、0 を戻します。
__poppar8	int __poppar8(unsigned long long);	64 ビット整数内に奇数のビットが設定されている場合、1 を戻します。それ以外の場合は、0 を戻します。
__pow	double __pow(double, double);	
__prefetch_by_load	void __prefetch_by_load(const void*);	明示的ロードによりメモリー・ロケーションをタッチします。
__prefetch_by_stream	void __prefetch_by_stream(const int, const void*);	明示的ストリームによりメモリー・ロケーションをタッチします。
__protected_stream_count	void __protected_stream_count(unsigned int *unit_cnt*, unsigned int *ID*);	*ID* という ID を持つ、限られた長さの protected ストリームに対して、*unit_cnt* 本のキャッシュ・ラインを設定します。*Unit_cnt* は、0 から 1023 までの値を持つ整数でなければなりません。ストリーム *ID* には、0 から 15 までの整数値がなければなりません。 (POWER5 プロセッサーのみ)
__protected_stream_go	void __protected_stream_go();	すべての限られた長さの protected ストリームの事前取り出しを開始します。 (POWER5 プロセッサーのみ)

名前	プロトタイプ	説明
__protected_stream_set	void __protected_stream_set(unsigned int *direction*, const void* *addr*, unsigned int *ID*);	*ID* という ID を使用する限られた長さの protected ストリームを確立します。これは、*addr* にあるキャッシュ・ラインから開始し、その後 *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) の方向でメモリー・アドレスから取り出します。このストリームは、ハードウェアで検出されるストリームによって置き換えられないように保護されています。 *Direction* には、値 1 (前方) または 3 (後方) が必要です。ストリーム *ID* には、0 から 15 までの整数値がなければなりません。 (POWER5 プロセッサーのみ)
__protected_unlimited_stream_set_go	void __protected_unlimited_stream_set_go (unsigned int *direction*, const void* *addr*, unsigned int *ID*);	*ID* という ID を使用して無制限の長さの protected ストリームを確立します。これは、*addr* にあるキャッシュ・ラインから開始し、その後 *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) の方向でメモリー・アドレスから取り出します。このストリームは、ハードウェアで検出されるどのようなストリームによっても置き換えられないように保護されています。 *Direction* には、値 1 (前方) または 3 (後方) が必要です。ストリーム *ID* には、0 から 15 までの整数値がなければなりません。 (PowerPC 970 および POWER5 プロセッサーのみ)
__protected_stream_stop	void __protected_stream_stop(unsigned int *ID*);	ID *ID* を持つ protected ストリームの事前取り出しを停止します。 (POWER5 プロセッサーのみ)
__protected_stream_stop_all	void __protected_stream_stop_all();	すべての protected ストリームの事前取り出しを停止します。 (POWER5 プロセッサーのみ)

名前	プロトタイプ	説明
__rdlam	unsigned long long __rdlam(unsigned long long *rs*, unsigned int *shift*, unsigned long long *mask*);	ダブルワードを左方に回転し、マスクに AND します。 *rs* の内容を、左方に *shift* ビットだけ回転します。回転したデータは、マスクに AND され、結果として戻されます。 *mask* は定数で、隣接するビット・フィールドを示していなければなりません。
__readflm	double __readflm();	浮動小数点状況/制御レジスターを読み取ります。
__rldimi	unsigned long long __rldimi(unsigned long long *rs*, unsigned long long *is*, unsigned int *shift*, unsigned long long *mask*);	ダブルワードを左方に即時回転してから、挿入をマスクします。 *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、 0<=*shift*<=63 でなければなりません。*mask* は定数で、隣接するビット・フィールドを示していなければなりません。
__rlwimi	unsigned int __rlwimi(unsigned int *rs*, unsigned int *is*, unsigned int *shift*, unsigned int *mask*);	ワードを左方に即時回転してから、挿入をマスクします。 *rs* を左方に *shift* ビットだけ回転してから、 *rs* をビット・マスク *mask* の下の *is* に挿入します。 shift は定数で、 0<=*shift*<=31 でなければなりません。*mask* は定数で、隣接するビット・フィールドを示していなければなりません。
__rlwnm	unsigned int __rlwnm(unsigned int *rs*, unsigned int *shift*, unsigned int *mask*);	ワードを左方に回転し、マスクに AND します。 *rs* を左方に *shift* ビットだけ回転し、ビット・マスク *mask* に *rs* を AND します。 *mask* は定数で、隣接するビット・フィールドを示していなければなりません。
__rotatel4	unsigned int __rotatel4(unsigned int *rs*, unsigned int *shift*);	ワードを左方に回転します。 *rs* を左方に *shift* ビットだけ回転します。
__setflm	double __setflm(double);	浮動小数点状況/制御レジスターを設定します。
__setrnd	double __setrnd(int);	丸めモードを設定します。

名前	プロトタイプ	説明
__stfiw	void __stfiw(const int* *addr*, double *value*);	浮動小数点を整数ワードとして保管します。 *value* の下位 32 ビットの内容が、*addr* でアドレッシングするストレージのワードに、変換されずに保管されます。
__store2r	void __store2r(unsigned short, unsigned short *);	2 バイト・レジスターを保管します。
__store4r	void __store4r(unsigned int, unsigned int *);	4 バイト・レジスターを保管します。
__swdiv_nochk	double __swdiv_nochk(double, double);	**double** 型の、浮動小数点除法です。範囲チェックは行いません。ループ内で繰り返し除法をする場合、引き数が許容範囲内にあると分かっているときには、この関数を使用することにより、通常の除法演算子や __swdiv 組み込み関数を使用した場合よりも高いパフォーマンスを得ることができます。 引き数の制約事項: 無限大に等しい分子、または、無限大、ゼロ、または正規化解除に等しい分母は許可されません。分子と分母の商が正または負の無限大であってはなりません。 **-qstrict** が有効である場合は、結果は IEEE 除法とビット単位で同等となります。このシナリオで正しい演算を行うには、引き数が以下の追加の制約事項を満たしている必要があります。分子の絶対値は、2 ^ (-970) より大で、かつ無限大より小でなければなりません。分母の絶対値は、2 ^ (-1022) より大で、かつ 2 ^ 1021 より小でなければなりません。分子と分母の商の絶対値は、2 ^ (-1021) より大で、かつ 2 ^ 1023 より小のでなければなりません。
__swdiv	double __swdiv(double, double);	**double** 型の浮動小数点除法。引き数の制限はありません。

名前	プロトタイプ	説明
__swdivs_nochk	float __swdivs_nochk(float, float);	**float** 型の浮動小数点除法です。範囲チェックは行いません。引き数の制約事項: 無限大に等しい分子、または、無限大、ゼロ、または正規化解除に等しい分母は許可されません。分子と分母の商が正または負の無限大であってはなりません。
__swdivs	float __swdivs(float, float);	**double** 型の浮動小数点除法。引き数の制限はありません。
__sync	void __sync(void);	すでに組み込まれている **__iospace_sync** のもう 1 つの名前です。 コンパイラーは、**__sync** を組み込まれているものとして認識します。名前以外は、すべてが **__iospace_sync** とまったく同じです。 **__sync** には、対応する PowerPC の命令名との一貫性があります。
__tdw	void __tdw(long long *a*, long long *b*, unsigned int *TO*);	ダブルワードをトラップします。 オペランド *a* が、オペランド *b* と比較されます。この比較の結果、5 ビットの定数 *TO* (**0** から **31** の範囲内の値を含む) に AND 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。各ビットの位置 (設定されていれば) は、以下の起こりうる条件を 1 つ以上示します。 **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)。
__trap	void __trap(int);	トラップ
__trapd	void __trapd (longlong);	パラメーターがゼロでない場合にトラップします。

名前	プロトタイプ	説明
__tw	void __tw(int *a*, int *b*, unsigned int *TO*);	ワードをトラップします。 オペランド ***a*** が、オペランド ***b*** と比較されます。この比較の結果、5 ビットの定数 *TO* (**0** から **31** の範囲内の値を含む) に AND 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。各ビットの位置 (設定されていれば) は、以下の起こりうる条件を 1 つ以上示します。 **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)。
__usrthds	int __usrthds(void);	**usrthds** ランタイム・オプションの値を戻します。 **usrthds** オプションがユーザーによって明示的に設定されていないか、プログラムのコンパイル時に -qsmp コンパイラー・オプションが指定されなかった場合、この関数は、選択されたランタイム・オプションに関係なく 0 を戻します。

付録 C. XL C/C++ 内のライブラリー

再配布可能ライブラリー

XL C/C++ は、次の再配布可能ライブラリーを提供しています。ご使用のアプリケーションによっては、これらのライブラリーの 1 つ以上を、XL C/C++ で作成したアプリケーションとともに配送しなければならない場合があります。

libibmc++.so

C++ プログラムにのみ使用されます。

libxlsmp.so libxlsmp_ser.so libxlsmpdebug.so

-qsmp または **-qsmp=omp** オプションが有効な場合にのみ必要です。

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の構成ファイルについては、28 ページの『構成ファイル内のコンパイラー・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 引き数	xldriver 属性
ld	gcc, g++	collect2	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	

ld コマンド・コンポーネント	オプション	ld 引き数	xldriver 属性
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	/opt/cross/powerpc64-linux/lib/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	/opt/cross/powerpc64-linux/lib/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	/opt/cross/powerpc64-linux/lib/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
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
		/opt/cross/powerpc64-linux/lib/crtn.o	crte_64

付録 D. 問題解決

本節には以下のトピックがあります。

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前に、あらかじめメッセージ・カタログをインストールし、メッセージ・カタログのインストール言語に環境変数の **LANG** と **NLSPATH** を設定しておかなければなりません。

以下のメッセージをコンパイルの途中で検出した場合、該当するメッセージ・カタログをオープンすることはできません。

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

ここで、*message_file* は、コンパイラーが開くことのできないメッセージ・カタログ名です。このメッセージは英語でのみ発行されます。

それから、メッセージ・カタログと環境変数が、適切な場所にあり、正しいことを検証してください。メッセージ・カタログまたは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、診断メッセージは抑止され、代わりに以下のメッセージが発行されます。

```
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でのみ発行されます。

ご使用のシステムにインストールされているメッセージ・カタログを判別するには、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したと想定して、以下のコマンドを使用して、カタログ用のファイル名すべてをリストすることができます。

```
ls /usr/lib/nls/msg/%L/*.cat
```

ここで、**%L** は、現行の 1 次言語環境 (ロケール) 設定です。デフォルト・ロケールは、**C** です。米国英語の場合、ロケールは **en_US** です。

/opt/ibmcmp/vacpp/7.0/msg のデフォルトのメッセージ・カタログは、

- コンパイラーが **%L** で指定されたロケール用のメッセージ・カタログを検出できないとき。
- ロケールがデフォルトの **C** から変更されたことがないとき。

NLSPATH および **LANG** 環境変数についての詳細は、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。

関連タスク

19 ページの『環境変数の設定』
20 ページの『他の環境変数の設定』

コンパイル中のページ・スペース・エラーの訂正

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・サイズを減らす。
- 最適化を使用しないでプログラムをコンパイルする。
- システム・ページ・スペースを競合するプロセスの数を減らす。
- システム・ページ・スペースを増やす。

ページング・スペースおよび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。

付録 E. 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 進値	8 進値	16 進値	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	改ページ
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 進値	8 進値	16 進値	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 進値	8 進値	16 進値	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 進値	8 進値	16 進値	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。

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

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

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

商標

以下は、IBM Corporation の商標です。

AIX
IBM
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

以下の規格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)) に準拠しています。
- C++ 言語は、International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998) に準拠しています。
- C および C++ 言語は、OpenMP C and C++ Application Programming Interface バージョン 1.0 に準拠しています。

Printed in Japan

SC88-9975-01

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