# Oracle® Solaris Studio 12.3: C++
# ユーザーズガイド

Part No: E26438
2011 年 12 月

ORACLE®

120220@25097

# 目次

# 例目次

# はじめに

このガイドでは、Oracle Solaris Studio 12.3 C++ コンパイラについて説明します。

## サポートされるプラットフォーム

Oracle Solaris Studio のこのリリースは、Oracle Solaris オペレーティングシステムを実行している SPARC ファミリのプロセッサアーキテクチャーを使用するプラットフォームと、Oracle Solaris または特定の Linux システムを実行している x86 ファミリのプロセッサアーキテクチャーを使用するプラットフォームをサポートしています。

このマニュアルでは、次の用語を使用して x86 プラットフォームの違いを示しています。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品を指します。
- 「x64」は、特定の 64 ビット x86 互換 CPU を指します。
- 「32 ビット x86」は、x86 ベースシステムで特定の 32 ビット情報を指します。

Linux システムに固有の情報は、サポートされている Linux x86 プラットフォームだけに関連し、Oracle Solaris システムに固有の情報は、SPARC および x86 システムでサポートされている Oracle Solaris プラットフォームだけに関連します。

サポートされるハードウェアプラットフォームとオペレーティングシステムリリースの完全なリストについては、Oracle Solaris Studio 12.3 リリースノートを参照してください。

## Oracle Solaris Studio マニュアル

Oracle Solaris Studio ソフトウェアの完全なマニュアルは、次のように見つけることができます。

- 製品のマニュアルは、リリースノート、リファレンスマニュアル、ユーザーガイド、チュートリアルも含め、Oracle Solaris Studio Documentation Web サイトにあります。

- コードアナライザ、パフォーマンスアナライザ、スレッドアナライザ、dbxtool、DLight、および IDE のオンラインヘルプには、これらのツール内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログボックスにある「ヘルプ」ボタンを使用してアクセスできます。
- コマンド行ツールのマニュアルページには、ツールのコマンドオプションが説明されています。

# 関連するサードパーティの Web サイトリファレンス

このマニュアルには、詳細な関連情報を提供するサードパーティの URL が記載されています。

---

注 – このマニュアルで紹介するサードパーティ Web サイトが使用可能かどうかについては、Oracle は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Oracle は保証しておらず、法的責任を負いません。また、このようなサイトやリソースから直接あるいは経由することで利用できるコンテンツ、商品、サービスの使用または依存が直接のあるいは関連する要因となり実際に発生した、あるいは発生するとされる損害や損失についても、Oracle は一切の法的責任を負いません。

---

# 開発者向けのリソース

Oracle Technical Network Web サイトにアクセスすると、Oracle Solaris Studio を使用する開発者向けの次のようなリソースがあります。

- リソースは頻繁に更新されます。
- ソフトウェアの最近のリリースに関連する完全なマニュアルへのリンク
- サポートレベルに関する情報
- ユーザーディスカッションフォーラム.

# Oracle サポートへのアクセス

Oracle のお客様は、My Oracle Support にアクセスして電子サポートを受けることができます。詳細は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info`、聴覚に障害があるお客様は `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` を参照してください。

# 表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

| 字体または記号 | 意味 | 例 |
|---|---|---|
| AaBbCc123 | コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。 | .login ファイルを編集します。<br>ls -a を使用してすべてのファイルを表示します。<br>system% |
| **AaBbCc123** | ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。 | system% **su**<br>password: |
| *AaBbCc123* | 変数を示します。実際に使用する特定の名前または値で置き換えます。 | ファイルを削除するには、rm *filename* と入力します。 |
| 『』 | 参照する書名を示します。 | 『コードマネージャ・ユーザーズガイド』を参照してください。 |
| 「」 | 参照する章、節、ボタンやメニュー名、強調する単語を示します。 | 第 5 章「衝突の回避」を参照してください。<br>この操作ができるのは、「スーパーユーザー」だけです。 |
| \ | 枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。 | sun% **grep '^#define **<br>**XV_VERSION_STRING'** |

Oracle Solaris OS に含まれるシェルで使用する、UNIX のデフォルトのシステムプロンプトとスーパーユーザープロンプトを次に示します。コマンド例に示されるデフォルトのシステムプロンプトは、Oracle Solaris のリリースによって異なります。

- C シェル

  machine_name% **command y|n** [*filename*]

- C シェルのスーパーユーザー

  machine_name# **command y|n** [*filename*]

- Bash シェル、Korn シェル、および Bourne シェル

  $ **command y|n** [*filename*]

- Bash シェル、Korn シェル、および Bourne シェルのスーパーユーザー

```
# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: `Shift` キーを押します)。ただし、キーボードによっては `Enter` キーが `Return` キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、`Ctrl-D` は `Control` キーを押したまま `D` キーを押すことを意味します。

パート I

# C++ コンパイラ

◆ ◆ ◆ 第 1 章

# C++ コンパイラの紹介

この章では、最新の Oracle Solaris Studio C++ コンパイラの概要について説明します。

## 1.1 Oracle Solaris Studio 12.3 C++ 5.12 コンパイラの新機能

この節では、Oracle Solaris Studio 12.3 C++ 5.12 コンパイラリリースに導入された新機能と変更された機能の概要を一覧に示します。

- 新しい SPARC T4 プラットフォームのサポート: `–xtarget=T4`、`–xchip=T4`、`–xarch=sparc4`
- 新しい x86 プラットフォーム Sandy Bridge / AVX のサポート: `–xtarget=sandybridge –xchip=sandybridge –xarch=avx`
- 新しい x86 プラットフォーム Westmere / AES のサポート: `–xtarget=westmere –xchip=westmere –xarch=aes`
- 新規コンパイラオプション: `–g3` は、拡張されたデバッグシンボルテーブル情報を追加します。(205 ページの「A.2.31 -g3」)
- 新しいコンパイラオプション: `–Xlinker` *arg* は、arg をリンカー `ld`(1) に渡します。`–Wl,`*arg* と同等です。(237 ページの「A.2.98 -Xlinker *arg*」)
- OpenMP のデフォルトのスレッド数 `OMP_NUM_THREADS` が 2 になりました (1 でした)。(289 ページの「A.2.152 -xopenmp[= *i*]」)
- OpenMP 3.1 共有メモリー並列化指定のサポート。(289 ページの「A.2.152 -xopenmp[= *i*]」)
- 新規コンパイラオプション: `–xivdep` は `ivdep` プラグマの解釈を設定します。`ivdep` プラグマは、最適化の目的でループ内で検出された、配列参照へのループがもたらす依存関係の一部またはすべてを無視するようにコンパイラに指示します。これによってコンパイラは、マイクロベクトル化、分散、ソフトウェアパイプラインなど、それ以外の場合は不可能なさまざまなループ最適化を実行できます。こ

れは、依存関係が重要ではない、または依存関係が実際に発生しないことをユーザーが把握している場合に使用されます。(270 ページの「A.2.126 -xivdep[=*p*]」)

- Sun Performance Library にリンクするには、`–library=sunperf` を使用します。これによって `-xlic_lib=sunperf` は廃止されます。(214 ページの「A.2.49 -library=*l*[,*l*...]」)
- `–compat=4` サブオプション (「互換モード」) は削除されました。デフォルトは `–compat=5` になりました。さらに、Linux プラットフォーム上でのみ以前使用可能だった g++ ソースおよびバイナリ互換性のための `-compat=g` オプションが、Oracle Solaris/x86 にも拡張されました (179 ページの「A.2.6 -compat=[ 5|g]」)。
- 新規オプション `–features=cplusplus_redef` によって、通常は事前定義されているマクロ `__cplusplus` を、`–D` オプションによってコマンド行で再定義できるようになりました。`__cplusplus` をソースコード内の `#define` ディレクティブ経由で再定義しようとすることは、引き続き許可されません。また、`–features=%none` および `–features=%all` の使用は、このリリースで非推奨となりました (190 ページの「A.2.17 -features=*a*[ ,*a*...]」)。
- 新規オプション `–xanalyze={code| no}` は、Oracle Solaris コードアナライザを使用して表示可能なソースコードの静的分析を提供します。(240 ページの「A.2.102 -xanalyze={code| no}」)
- 新しいサブオプション `–xbuiltin=%default` は、`errno` を設定しない関数のみをインライン化します。`errno` の値はすべての最適化レベルで常に正しく、信頼できる方法で検査できます。(247 ページの「A.2.108 -xbuiltin[={ %all|%default| %none}]」)
- ユーザー提供のコンパイラオプションデフォルトのサポート。(58 ページの「3.4 ユーザー指定のデフォルトオプションファイル」)
- C99 ヘッダー `stdbool.h` および C++ の同等の `cstdbool` が使用できます。C++ では、ヘッダーは効果がなく、C99 との互換性のために提供されています。

## 1.2 x86 の特記事項

- x86 Oracle Solaris プラットフォーム用にコンパイルする場合、いくつかの重要な点に注意してください。
- `-xarch` を `sse`、`sse2`、`sse2a`、または `sse3` 以降に設定してコンパイルしたプログラムは、必ずこれらの拡張子と機能を提供するプラットフォームでのみ実行してください。
- x86 の 80 ビット浮動小数点レジスタが原因で、x86 での演算結果が SPARC の結果と異なる場合があります。この差を最小にするには、`--fstore` オプションを使用するか、ハードウェアが SSE2 をサポートしている場合は `-xarch=sse2` でコンパイルします。
- 組み込み数学ライブラリ (`sin(x)` など) が異なるため、Solaris と Linux の間でも数値結果が異なることがあります。

## 1.3 64 ビットプラットフォーム用のコンパイル

ILP32 32 ビットモデル用にコンパイルするには、`–m 32` オプションを使用します。ILP64 64 ビットモデル用にコンパイルするには、`–m64` オプションを使用します。

ILP 32 モデルでは、C++ 言語の int、long、およびポインタデータ型はすべて 32 ビット幅になります。long およびポインタデータ型を指定する LP64 モデルは、すべて 64 ビット拡張です。Oracle Solaris OS および Linux OS は、LP64 メモリーモデルの大きなファイルや配列もサポートします。

`-m64` を使用してコンパイルを行う場合、結果の実行可能ファイルは、64 ビットカーネルを実行する Oracle Solaris OS または Linux OS の 64 ビット UltraSPARC または x86 プロセッサでのみ動作します。コンパイル、リンク、および 64 ビットオブジェクトの実行は、64 ビット実行をサポートする Oracle Solaris OS または Linux OS でのみ行うことができます。

## 1.4 バイナリの互換性の妥当性検査

Oracle Solaris システムでは、Oracle Solaris Studio コンパイラによってコンパイルされたプログラムのバイナリには、そのコンパイル済みバイナリによって想定されている命令セットを示すアーキテクチャーハードウェアフラグが付いています。実行時にこれらのマーカーフラグがチェックされ、実行しようとしているハードウェアで、そのバイナリが実行できることが検証されます。

プログラムにこれらのアーキテクチャーハードウェアフラグが含まれない場合、またはプラットフォームが適切な機能または命令セット拡張に対応していない場合、プログラムを実行することによりセグメント例外、または明示的な警告メッセージなしの不正な結果が発生することがあります。

この警告は、`.il` インラインアセンブリ言語関数を使用しているプログラムや、SSE、SSE2、SSE2a、SSE3、およびより新しい命令と拡張機能を利用している `__asm()` アセンブラコードにも当てはまります。

## 1.5 準拠規格

この C++ コンパイラ (`CC`) は、ISO International Standard for C++, ISO IS 14882:2003, Programming Language - C++ に準拠しています。

SPARC プラットフォームでは、このコンパイラは、UltraSPARC の実装と SPARC V8 と SPARC V9 の「最適化活用」機能をサポートします。これらの機能は、Prentice-Hall から出版された SPARC International による『SPARC アーキテクチャ・マニュアル バージョン 8』(トッパン刊) と『SPARC Architecture Manual, Version 9』(ISBN 0-13-099227-5) (英語版のみ) に定義されています。

このマニュアルでは、「標準」は、前述の規格の各バージョンに準拠していることを意味します。「非標準」および「拡張」は、これらの規格のバージョンに準拠しない機能のことを指します。

これらの標準は、それぞれの標準を策定する組織によって改訂されることがあります。したがって、コンパイラが準拠するバージョンの規格が改訂されたり、書き換えられた場合、機能によっては、Oracle Solaris Studio C++ コンパイラの将来のリリースで前のリリースと互換性がなくなる場合があります。

## 1.6 リリース情報

『Oracle Solaris Studio 12.3 リリースの新機能』ガイドでは、このリリースのコンパイラに関する重要な情報の要旨を示し、次が含まれています。

- マニュアルの印刷後に判明した情報
- 新規および変更された機能
- ソフトウェアの修正事項
- 問題および解決方法
- 制限および互換性の問題
- 出荷可能なライブラリ
- 実装されていない規格

『新機能』ガイドには、このリリースのドキュメント索引 (http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation) からアクセスできます。

## 1.7 マニュアルページ

オンラインのマニュアルページ (man) では、コマンドや関数、サブルーチン、およびその機能に関する情報を簡単に参照できます。

マニュアルページを表示するには、次のように入力してください。

```
example% man topic
```

C++ のドキュメント全体を通して、マニュアルページのリファレンスは、トピック名とマニュアルページの節番号で表示されます。CC(1) を表示するには、man CC と入力します。1 以外の節 (ieee_flags(3M) など) には、次のように man コマンドで -s オプションを使用してアクセスできます。

```
example% man -s 3M ieee_flags
```

# 1.8 各国語のサポート

このリリースの C++ では、英語以外の言語を使用したアプリケーションの開発をサポートしています。対象としている言語は、ヨーロッパのほとんどの言語、中国語、日本語です。このため、アプリケーションをある言語から別の言語に簡単に置き換えることができます。この機能を国際化と呼びます。

通常 C++ コンパイラでは、次のように国際化を行なっています。

- どの国のキーボードから入力された ASCII 文字でも認識する。つまりキーボードに依存せず、8 ビット透過となっています。
- メッセージによっては現地語で出力できるものもある。
- 注釈、文字列、データに、現地語の文字を使用できる。
- C++ は、Extended UNIX Character (EUC) 準拠の文字セットのみサポートしています。この文字セットでは、文字列中のすべての NULL バイトが NULL 文字になります。また、文字列中で ASCII 値が / のバイトはすべて / 文字になります。

変数名は国際化できません。必ず英語の文字を使用してください。

アプリケーションをある国の言語から別の国の言語に変更するには、ロケールを設定します。言語の切り換えのサポートに関する情報については、オペレーティングシステムのドキュメントを参照してください。

◆ ◆ ◆ 第 2 章

# C++ コンパイラの使用方法

この章では、C++ コンパイラの使用方法を説明します。

コンパイラの主な目的は、C++ などの高水準言語で書かれたプログラムをコンピュータハードウェアで実行できるデータファイルに変換することです。C++ コンパイラを使用すると、次の作業を行うことができます。

- ソースファイルを再配置可能なバイナリ (.o) ファイルに変換する。これらのファイルはそのあと、実行可能ファイル、(-xar オプションで) 静的 (アーカイブ) ライブラリ (.a) ファイル、動的 (共有) ライブラリ (.so) ファイルなどにリンクされる。
- オブジェクトファイルとライブラリファイルのどちらか (または両方) をリンク (または再リンク) して実行可能ファイルを作成する。
- 実行時デバッグを (-g オプションで) 有効にして、実行可能ファイルをコンパイルする。
- 文レベルや手続きレベルの実行時プロファイルを (-pg オプションで) 有効にして、実行可能ファイルをコンパイルする。

## 2.1 はじめに

この節では、C++ コンパイラを使って C++ プログラムのコンパイルと実行をどのように行うかを簡単に説明します。コマンド行オプションの詳細なリファレンスについては、付録 A 「C++ コンパイラオプション」を参照してください。

---

注 – この章のコマンド行の例は、CC の使用方法を示すためのものです。実際に出力される内容はこれと多少異なる場合があります。

---

C++ プログラムを構築して実行するための基本的な手順には、次の作業が含まれます。

1. エディタを使用して、表 2–1 に一覧表示されている有効な接尾辞の 1 つを指定し、C++ ソースファイルを作成する。
2. コンパイラを起動して実行可能ファイルを作成する。
3. 実行可能ファイルの名前を入力してプログラムを実行する。

次のプログラムは、メッセージを画面に表示する例です。

```
example% cat greetings.cc
    #include <iostream>
    int main()  {
      std::cout << “Real programmers write C++!” << std::endl;
      return 0;
    }
example% CC greetings.cc
example% ./a.out
 Real programmers write C++!
example%
```

この例では、ソースファイル greetings.cc を CC でコンパイルしています。デフォルトでは、実行可能ファイルがファイル a.out として作成されます。プログラムを起動するには、コマンドプロンプトで実行可能ファイル名 a.out を入力します。

従来、UNIX コンパイラは実行可能ファイルに a.out という名前を付けていました。しかし、すべてのコンパイルで同じファイルを使用するのは不都合な場合があります。そのファイルがすでにあれば、コンパイラを実行したときに上書きされてしまうからです。次の例のように、コンパイラオプションに -o を使用すれば、実行可能出力ファイルの名前を指定できます。

```
example% CC– o greetings greetings.cc
```

この例では、-o オプションを指定することによって、実行可能なコードがファイル greetings に書き込まれます。(プログラムにソースファイルが 1 つだけしかない場合は、ソースファイル名から接尾辞を除いたものをそのプログラム名にすることが一般的な方法です。)

あるいは、コンパイルのあとに mv コマンドを使って、デフォルトの a.out ファイルを別の名前に変更することもできます。いずれの場合も、実行可能ファイルの名前を入力して、プログラムを実行します。

```
example% ./greetings
Real programmers write C++!
example%
```

# 2.2 コンパイラの起動

本章のこれ以降の節では、CC コマンドで使用する規約、コンパイラのソース行指令など、コンパイラの使用に関連する内容について説明します。

## 2.2.1 コマンド構文

コンパイラの一般的なコマンド行の構文を次に示します。

CC [*options*] [*source-files*] [*object-files*] [*libraries*]

*options* は、先頭にダッシュ (-) またはプラス記号 (+) の付いたキーワード (オプション) です。このオプションには、引数をとるものがあります。

通常、コンパイラオプションの処理は、左から右へと行われ、マクロオプション (ほかのオプションを含むオプション) は、条件に応じて内容が変更されます。ほとんどの場合、同じオプションを 2 回以上指定すると、最後に指定したものだけが有効になり、オプションの累積は行われません。次の点に注意してください。

- すべてのリンカーオプション、ならびに `-features`、`-I`、`l`、`L`、`-library`、`-pti`、`-R`、`-staticlib`、`-U`、`-verbose` および `-xprefetch` オプションで指定した内容は蓄積され、上書きはされません。
- `-U` オプションは、すべて `-D` オプションのあとに処理されます。

ソースファイル、オブジェクトファイル、およびライブラリは、コマンド行に指定した順にコンパイルとリンクが行われます。

次の例では、CC を使って 2 つのソースファイル (`growth.C` と `fft.C`) をコンパイルし、実行時デバッグを有効にして `growth` という名前の実行可能ファイルを作成します。

```
example% CC -g -o growth growth.C fft.C
```

## 2.2.2 ファイル名に関する規則

コンパイラがコマンド行に指定されたファイルをどのように処理するかは、ファイル名に付加された接尾辞で決まります。次の表以外の接尾辞を持つファイルや、接尾辞がないファイルはリンカーに渡されます。

表 2–1 C++ コンパイラが認識できるファイル名接尾辞

| 接尾辞 | 言語 | 処理 |
|---|---|---|
| .c | C++ | C++ ソースファイルとしてコンパイルし、オブジェクトファイルを現在のディレクトリに入れる。オブジェクトファイルのデフォルト名は、ソースファイル名に .o 接尾辞が付いたものになる。 |
| .C | C++ | .c 接尾辞と同じ処理。 |
| .cc | C++ | .c 接尾辞と同じ処理。 |
| .cpp | C++ | .c 接尾辞と同じ処理。 |
| .cxx | C++ | .c 接尾辞と同じ処理。 |
| .c++ | C++ | .c 接尾辞と同じ処理。 |
| .i | C++ | C++ ソースファイルとして扱われるプリプロセッサ出力ファイル。.c 接尾辞と同じ処理。 |
| .s | アセンブラ | ソースファイルをアセンブラを使ってアセンブルする。 |
| .S | アセンブラ | C 言語プリプロセッサとアセンブラを使ってソースファイルをアセンブルする。 |
| .il | インライン展開 | アセンブリ用のインラインテンプレートファイルを使ってインライン展開を行う。コンパイラはテンプレートを使って、選択されたルーチンのインライン呼び出しを展開します(インラインテンプレートファイルは、特殊なアセンブラファイルです。inline(1) のマニュアルページを参照してください)。 |
| .o | オブジェクトファイル | オブジェクトファイルをリンカーに渡す |
| .a | 静的 (アーカイブ) ライブラリ | オブジェクトライブラリの名前をリンカーに渡す。 |
| .so<br>.so.*n* | 動的 (共有) ライブラリ | 共有オブジェクトの名前をリンカーに渡す。 |

## 2.2.3 複数のソースファイルの使用

C++ コンパイラでは、複数のソースファイルをコマンド行に指定できます。コンパイラが直接または間接的にサポートするファイルも含めて、コンパイラによってコンパイルされる 1 つのソースファイルを「コンパイル単位」といいます。C++ では、それぞれのソースが別個のコンパイル単位として扱われます。

## 2.3 バージョンが異なるコンパイラでのコンパイル

このコンパイラは、デフォルトではキャッシュを使用しません。-instances=extern を指定した場合のみ、キャッシュが使用されます。コンパイラは、キャッシュを使用する場合、キャッシュディレクトリのバージョンを確認し、キャッシュのバージョンの問題を検出した場合は必ずエラーメッセージを発行します。将来の C++ コンパイラもキャッシュのバージョンを調べます。たとえば、将来のコンパイラは異なるテンプレートキャッシュのバージョン識別子を持っているため、現在のリリースで作成されたキャッシュディレクトリを処理しようとすると、次のようなエラーを出力します。

```
Template Database at ./SunWS_cache is incompatible with
this compiler
```

同様に、現在のリリースのコンパイラで以降のバージョンのコンパイラで作成されたキャッシュディレクトリを処理しようとすると、エラーが発行されます。

コンパイラをアップグレードするときは、キャッシュを消去するのが常によい方法です。テンプレートキャッシュディレクトリを含むすべてのディレクトリで CCadmin -clean を実行します。ほとんどの場合、テンプレートキャッシュディレクトリの名前は SunWS_cache です。CCadmin -clean の代わりに、rm -rf SunWS_cache と指定しても同様の結果が得られます。

## 2.4 コンパイルとリンク

この節では、プログラムのコンパイルとリンクについていくつかの側面から説明します。次の例では、CC を使って 3 つのソースファイルをコンパイルし、オブジェクトファイルをリンクして prgrm という実行可能ファイルを作成します。

```
example% CC file1.cc file2.cc file3.cc -o prgrm
```

### 2.4.1 コンパイルとリンクの流れ

前の例では、コンパイラがオブジェクトファイル (file1.o、file2.o、file3.o) を自動的に生成し、次にシステムリンカーを起動してファイル prgrm の実行可能プログラムを作成します。

コンパイル後も、オブジェクトファイル (file1.o、file2.o、および file3.o) はそのまま残ります。この規則により、ファイルの再リンクと再コンパイルを簡単に行えます。

注 – ソースファイルが 1 つだけであるプログラムに対してコンパイルとリンクを同時に行なった場合は、対応する .o ファイルが自動的に削除されます。複数のソースファイルをコンパイルする場合を除いて、すべての .o ファイルを残すためにはコンパイルとリンクを別々に行なってください。

コンパイルが失敗すると、エラーごとにメッセージが返されます。エラーがあったソースファイルの .o ファイルは生成されず、実行可能プログラムも作成されません。

## 2.4.2 コンパイルとリンクの分離

コンパイルとリンクは別々に行うことができます。-c オプションを指定すると、ソースファイルがコンパイルされて .o オブジェクトファイルが生成されますが、実行可能ファイルは作成されません。-c オプションを指定しないと、コンパイラはリンカーを起動します。コンパイルとリンクを分離すれば、1 つのファイルを修正するためにすべてのファイルを再コンパイルする必要はありません。次の例では、最初の手順で 1 つのファイルをコンパイルし、次の手順でそれをほかのファイルとリンクします。

```
example% CC -c file1.cc          Make new object file
example% CC -o prgrm file1.o file2.o file3.o          Make executable file
```

リンク時には (2 行目)、完全なプログラムを作成するのに必要なすべてのオブジェクトファイルを必ず 指定してください。オブジェクトファイルが足りないと、リンクは「undefined external reference (未定義の外部参照がある)」エラーで、ルーチンがないために失敗します。

## 2.4.3 コンパイルとリンクの整合性

コンパイルとリンクを別々に実行する場合で、48 ページの「3.3.3 コンパイル時とリンク時のオプション」に示すコンパイラオプションを使用する場合は、コンパイルとリンクの整合性を保つことは非常に重大な意味を持ちます。

これらのオプションのいずれかを使用してサブプログラムをコンパイルした場合は、リンクでも同じオプションを使用してください。

- -library オプションまたは -m64 / -m32 オプションを使用してコンパイルする場合、これらの同じオプションをすべての cc コマンドに含める必要があります。
- -p、-xpg、-xprofile オプションの場合、ある段階ではオプションを指定して別の段階では指定しないと、プログラムの正しさには影響はありませんが、プロファイル処理ができなくなります。

- -g、-g0 オプションの場合、ある段階ではオプションを指定して別の段階では指定しないと、プログラムの正しさには影響はありませんが、プログラムを正しくデバッグできなくなります。これらのオプションでコンパイルされず、-g または -g0 でリンクされるモジュールはどれも、デバッグ用に正しく準備されません。--g オプション (または -g0 オプション)付きの main 関数があるモジュールをコンパイルするには、通常デバッグする必要があります。

次の例では、-library=stlport4 コンパイラオプションを使用してプログラムをコンパイルしています。

```
example% CC -library=stlport4 sbr.cc -c
example% CC -library=stlport4 main.cc -c
example% CC -library=stlport4 sbr.o main.o -o myprogram
```

-library=stlport4 を一貫して使用しない場合は、プログラムの特定の部分はデフォルトの libCstd を使用し、ほかの部分はオプションの置換である STLport ライブラリを使用します。結果として得られたプログラムは正常にリンクできず、どのような状況でも正常に動作しません。

プログラムがテンプレートを使用する場合は、リンク時に一部のテンプレートがインスタンス化される可能性があります。その場合、インスタンス化されたテンプレートは最終行 (リンク行) のコマンド行オプションを使用してコンパイルされます。

## 2.4.4 64 ビットメモリーモデル用のコンパイル

-m64 オプションを使用して、ターゲットプラットフォームの 64 ビットメモリーモデルを指定します。64 ビットオブジェクトのコンパイル、リンク、および実行は、64 ビット実行をサポートする Oracle Solaris または Linux プラットフォームでのみ行うことができます。

## 2.4.5 コンパイラのコマンド行診断

-V オプションを指定すると、CC によって起動された各プログラムの名前とバージョン番号が表示されます。-v オプションを指定すると、CC によって起動されたコマンド行全体が表示されます。

–verbose=%all を指定すると、コンパイラに関する追加情報が表示されます。

コマンド行に指定された引数をコンパイラが認識できない場合には、それらはリンカーオプション、オブジェクトプログラムファイル名、ライブラリ名のいずれかとみなされます。

基本的には次のように区別されます。

- 認識できないオプション。これらの前にダッシュ (–) またはプラス記号 (+) が付けられ、警告が生成されます。

- 認識できない非オプション (先頭にダッシュかプラス符号 (+) が付いていないもの)には、警告が生成されません。ただし、リンカーへ渡されます。リンカーが認識しない場合は、リンカーエラーメッセージが生成されます。

次の例で、-bit は CC によって認識されないため、リンカー (ld) に渡されます。リンカーはこれを解釈しようとします。単一文字の ld オプションは連続して指定できるので、リンカーは -bit を -b、-i、-t とみなします。これらはすべて有効な ld オプションです。この結果は意図または期待したものとは異なる可能性があります。

```
example% CC -bit move.cc       -bit is not a recognized compiler option
CC: Warning: Option -bit passed to ld, if ld is invoked, ignored otherwise
```

次の例では、CC オプション -fast を指定しようとしましたが、先頭のダッシュ (-) を入力しませんでした。コンパイラはこの引数もリンカーに渡します。リンカーはこれをファイル名とみなします。

```
example% CC fast move.cc                    < - The user meant to type -fast
move.CC:
ld: fatal: file fast: cannot open file; errno=2
ld: fatal: File processing errors. No output written to a.out
```

## 2.4.6 コンパイラの構成

C++ コンパイラパッケージは、フロントエンド (CC コマンド本体)、オプティマイザ (最適化)、コードジェネレータ (コード生成)、アセンブラ、テンプレートのプリリンカー (リンクの前処理をするプログラム)、リンクエディタから構成されています。コマンド行オプションでほかの指定を行わないかぎり、CC コマンドはこれらの構成要素をそれぞれ起動します。

これらのコンポーネントはいずれもエラーを生成する可能性があり、コンポーネントはそれぞれ異なる処理を行うため、エラーを生成するコンポーネントを識別することが役立つことがあります。コンパイラ実行中に詳細を表示するには、-v および -dryrun オプションを使用します。

次の表に示すように、コンパイラの構成要素への入力ファイルには異なるファイル名接尾辞が付いています。どのようなコンパイルを行うかは、この接尾辞で決まります。ファイル名接尾辞の意味については、表 2–1 を参照してください。

表 2–2 C++ コンパイルシステムの構成要素

| コンポーネント | 説明 | 使用時の注意 |
|---|---|---|
| ccfe | フロントエンド (コンパイラプリプロセッサ (前処理系) とコンパイラ) | |
| iropt | コードオプティマイザ | -xO[2-5]、-fast |

表 2–2 C++ コンパイルシステムの構成要素 (続き)

| コンポーネント | 説明 | 使用時の注意 |
|---|---|---|
| ir2hf | x86: 中間言語トランスレータ | -xO[2-5]、-fast |
| inline | SPARC: アセンブリ言語テンプレートのインライン展開 | .il ファイルを指定 |
| fbe | アセンブラ | |
| cg | SPARC: コード生成、インライン機能、アセンブラ | |
| ube | x86: コードジェネレータ | -xO[2-5]、-fast |
| CClink | テンプレートのプリリンカー | -instances=extern オプションでのみ使用します |
| ld | リンクエディタ | |

## 2.5 指示および名前の前処理

この節では、C++ コンパイラ特有の前処理の指示について説明します。

### 2.5.1 プラグマ

プリプロセッサ指令 pragma は C++ 標準の一部ですが、書式、内容、および意味はコンパイラごとに異なります。C++ コンパイラが認識するプラグマ (指令) の詳細は、付録 B「プラグマ」を参照してください。

Oracle Solaris Studio C++ は、C99 のキーワードである _Pragma もサポートしています。次の 2 つの呼び出しは同等です。

```
#pragma dumpmacros(defs)
_Pragma("dumpmacros(defs)")
```

#pragma の代わりに _Pragma を使用するには、プラグマテキストをリテラル文字列として記述し、_Pragma キーワードの 1 つの引数として括弧で囲みます。

### 2.5.2 可変数の引数をとるマクロ

C++ コンパイラでは次の書式の #define プリプロセッサの指示を受け入れます。

```
#define identifier (...) replacement-list
#define identifier (identifier-list, ...) replacement-list
```

マクロパラメータリストの終わりが省略符号である場合、マクロパラメータより多くの引数をマクロの呼び出しで使用できます。追加の引数は、マクロ交換リストにおいて __VA_ARGS__ という名前で参照できる、コンマを含んだ単一文字列にまとめられます。

次の例は、変更可能な引数リストマクロの使い方を示しています。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                          printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n",x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

この結果は、次のようになります。

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):printf("x is %d but y is %d", x, y));
```

## 2.5.3 事前に定義されている名前

付録の 181 ページの「A.2.8 -D*name*[ =*def*]」 は、事前に定義されているマクロを示しています。これらの値は、`#ifdef` のようなプリプロセッサに対する条件式の中で使用できます。`+p` オプションを指定すると、`sun`、`unix`、`sparc`、および `i386` の事前定義マクロは自動的に定義されません。

## 2.5.4 警告とエラー

`#error` および `#warning` プリプロセッサディレクティブを使用すると、コンパイル時の診断を生成できます。

| | |
|---|---|
| `#error` *token-string* | エラー診断 *token-string* を発行して、コンパイルを終了します。 |
| `#warning` *token-string* | 警告診断 *token-string* を発行してコンパイルを続行します。 |

# 2.6 メモリー条件

コンパイルに必要なメモリー量は、次の要素によって異なります。

- 各手続きのサイズ
- 最適化のレベル
- 仮想メモリーに対して設定された限度
- ディスク上のスワップファイルのサイズ

SPARC プラットフォームでメモリーが足りなくなると、オプティマイザは最適化レベルを下げて現在の手続きを実行することでメモリー不足を補おうとします。それ以後のルーチンについては、コマンド行の -x0*level* オプションで指定した元のレベルに戻ります。

1 つのファイルに多数のルーチンが入っている場合、それをコンパイルすると、メモリーやスワップ領域が足りなくなることがあります。最適化のレベルを下げてみてください。あるいは、最大のプロシージャを、個別のファイルに分割してください。

## 2.6.1 スワップ領域のサイズ

現在のスワップ領域は swap -s コマンドで表示できます。詳細は、swap(1M) のマニュアルページを参照してください。

swap コマンドを使った例を次に示します。

```
example% swap -s
total: 40236k bytes allocated + 7280k reserved = 47516k used, 1058708k available
```

## 2.6.2 スワップ領域の増加

ワークステーションのスワップ領域を増やすには、mkfile(1M) と swap(1M) コマンドを使用します。そのためには、スーパーユーザーである必要があります。mkfile コマンドは特定サイズのファイルを作成し、swap -a はこのファイルをシステムのスワップ領域に追加します。

```
example# mkfile -v 90m /home/swapfile
/home/swapfile 94317840 bytes
example# /usr/sbin/swap -a  /home/swapfile
```

## 2.6.3 仮想メモリーの制御

1 つの手続きが数千行からなるような非常に大きなルーチンを -xO3 以上でコンパイルすると、大容量のメモリーが必要になることがあります。このようなときには、システムのパフォーマンスが低下します。メモリーフットプリントを制御するには、1 つのプロセスで使用できる仮想メモリーの量を制限します。

sh シェルで仮想メモリーを制限するには、ulimit コマンドを使用します。詳細は、sh(1) のマニュアルページを参照してください。

次の例では、仮想メモリーを 4G バイトに制限しています。

```
example$ ulimit -d 4000000
```

csh シェルで仮想メモリーを制限するには、limit コマンドを使用します。 詳細は、csh(1) のマニュアルページを参照してください。

次の例でも、仮想メモリーを 4G バイトに制限しています。

```
example% limit datasize 4G
```

どちらの例でも、オプティマイザは データ空間が 4G バイトになった時点でメモリー不足が発生しないような手段をとります。

仮想メモリーの限度は、システムの合計スワップ領域の範囲内です。さらに実際は、大きなコンパイルが行われているときにシステムが正常に動作できるだけの小さい値である必要があります。

スワップ領域の半分以上がコンパイルによって使用されることがないようにしてください。

8G バイトのスワップ領域のあるマシンでは、次のコマンドを使用します。

sh シェルの場合

```
example$ ulimit -d 4000000
```

csh の場合

```
example% limit datasize 4G
```

最適な設定は、必要な最適化レベルと使用可能な実メモリーと仮想メモリーの量によって異なります。

### 2.6.4 メモリー条件

ワークステーションには、少なくとも 2G バイトのメモリーを実装する必要があります。詳細な要件については、製品のリリースノートを参照してください。

## 2.7 C++ オブジェクトに対する strip コマンドの使用

UNIX の strip コマンドは、C++ のオブジェクトファイルに対して使用すべきではありません。それらのオブジェクトファイルが使用不可能になることがあります。

## 2.8 コマンドの簡略化

CCFLAGS 環境変数で特別なシェル別名を定義するか、または make を使用すると、複雑なコンパイラコマンドを簡略化できます。

### 2.8.1 C シェルでの別名の使用

次の例では、頻繁に使用するオプションをコマンドの別名として定義します。

```
example% alias CCfx "CC -fast -xnolibmil"
```

次に、この別名 CCfx を使用します。

```
example% CCfx any.C
```

コマンド CCfx は、次のコマンドと同じです。

```
example% CC -fast -xnolibmil any.C
```

### 2.8.2 CCFLAGS によるコンパイルオプションの指定

CCFLAGS 環境変数を設定すると、一度に特定のオプションを指定できます。

CCFLAGS 変数は、コマンド行に明示的に指定できます。次の例は、CCFLAGS の設定方法を示したものです (C シェル)。

```
example% setenv CCFLAGS '-xO2 -m64'
```

次の例では、CCFLAGS を明示的に使用しています。

```
example% CC $CCFLAGS any.cc
```

make を使用する場合、CCFLAGS 変数が前述の例のように設定され、メイクファイルのコンパイル規則が暗黙的に使用された状態で make を呼び出すと、次のコマンドと同等のコンパイルになります。

```
CC -xO2 -m64 files...
```

## 2.8.3 make の使用

make ユーティリティーは、Oracle Solaris Studio のすべてのコンパイラで簡単に使用できる非常に強力なプログラム開発ツールです。詳細については make(1S) のマニュアルページを参照してください。

### 2.8.3.1 make での CCFLAGS の使用

メイクファイルの暗黙のコンパイラ規則を使用する、つまり、*C++* コンパイルがない場合は、make プログラムによって CCFLAGS が自動的に使用されます。

◆ ◆ ◆ 第 3 章

# 3 C++ コンパイラオプションの使い方

この章では、コマンド行 C++ コンパイラオプションの使用方法について説明してから、機能別にその使用方法を要約します。オプションの詳細な説明は、176 ページの「A.2 オプションの一覧」を参照してください。

## 3.1 構文の概要

次の表は、一般的なオプション構文の形式の例です。

表 3–1 オプション構文形式の例

| 構文形式 | 例 |
|---|---|
| `-option` | `-E` |
| `-option`*value* | `-I`*pathname* |
| `-option=`*value* | `-xunroll=4` |
| `-option` *value* | `-o` *filename* |

括弧、中括弧、角括弧、パイプ文字、および省略符号は、オプションの説明で使用されているメタキャラクタです。これらは、オプションの一部ではありません。使用法の構文に関する詳細な説明は、「はじめに」の表記規則を参照してください。

## 3.2 一般的な注意事項

C++ コンパイラのオプションを使用する際の一般的な注意事項は次のとおりです。

- `-l`*lib* オプションは、ライブラリ `lib`*lib*`.a` (または `lib`*lib*`.so`) とリンクするときに使用します。ライブラリが正しい順序で検索されるように、`-l`*lib* オプションは、ソースやオブジェクトのファイル名のあとに指定する方が安全です。
- 一般にコンパイラオプションは左から右に処理され、マクロオプション (ほかのオプションを含むオプション) は条件に応じて内容が変更されます (ただし `-U` オプションだけは、すべての `-D` オプション後に処理されます)。この規則はリンカーのオプションには適用されません。
- `-features`、`-I-`、`-L`、`-library`、`-pti`、`-R`、`-staticlib`、`-U`、`-verbose` および `-xprefetch` オプションで指定した内容は蓄積され、上書きはされません。
- `-D` オプションは累積されます。同じ名前に複数の `-D` オプションがあるとお互いに上書きされます。

ソースファイル、オブジェクトファイル、ライブラリは、コマンド行に指定された順序でコンパイルおよびリンクされます。

## 3.3 機能別に見たオプションの要約

この節には、参照しやすいように、コンパイラオプションが機能別に分類されています。各オプションの詳細は、付録 A 「C++ コンパイラオプション」を参照してください。

オプションは、特に記載がないかぎりすべてのプラットフォームに適用されます。SPARC ベースシステム上の Oracle Solaris OS に特有の機能は *SPARC*、x86 ベースシステム上の Oracle Solaris OS に特有の機能は *x86* として識別されます。

### 3.3.1 コード生成オプション

表 3–2 コード生成オプション

| オプション | 処理 |
|---|---|
| `-compat` | コンパイラの主要リリースとの互換モードを設定します。 |
| `-g` | デバッグ用にコンパイルします。 |
| `-KPIC` | 位置に依存しないコードを生成します。 |
| `-Kpic` | 位置に依存しないコードを生成します。 |
| `-mt` | マルチスレッド化したコードのコンパイルとリンクを行います。 |

表 3–2 コード生成オプション　(続き)

| オプション | 処理 |
|---|---|
| -xaddr32 | コードを 32 ビットアドレス空間に制限します (x86/x64)。 |
| -xarch | ターゲットアーキテクチャーを指定します。 |
| -xcode=*a* | (SPARC) コードのアドレス空間を指定します。 |
| -xlinker | リンカーオプションを指定します。 |
| -xMerge | (SPARC) データセグメントとテキストセグメントをマージします。 |
| -xtarget | ターゲットシステムを指定します。 |
| -xmodel | 64 ビットオブジェクトの形式を Solaris x86 プラットフォーム用に変更します。 |
| +w | 意図しない結果が生じる可能性のあるコードを特定します。 |
| +w2 | +w で生成される警告に加え、おそらく問題がなくても、プログラムの移植性を低下させる可能性がある技術的な違反についての警告も生成します。 |
| -xregs | コンパイラは、一時記憶領域として使用できるレジスタ (一時レジスタ) が多ければ、それだけ高速なコードを生成します。このオプションは、利用できる一時レジスタを増やしますが、必ずしもそれが適切であるとはかぎりません。 |
| -z *arg* | リンカーオプション |

## 3.3.2 コンパイル時パフォーマンスオプション

表 3–3 コンパイル時パフォーマンスオプション

| オプション | 処理 |
|---|---|
| -instlib | 指定ライブラリにすでに存在しているテンプレートインスタンスの生成を禁止します。 |
| -m32\|-m64 | コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。 |
| -xinstrument | スレッドアナライザで分析するために、プログラムをコンパイルして計測します。 |
| -xjobs | コンパイラが処理を行うために作成するプロセスの数を設定します。 |

表 3–3 コンパイル時パフォーマンスオプション (続き)

| オプション | 処理 |
|---|---|
| -xpch | 共通の一連のインクルードファイルを共有するソースファイルを持つアプリケーションのコンパイル時間を短縮できることがあります。 |
| -xpchstop | -xpch でプリコンパイル済みヘッダーファイルを作成する際に考慮される最後のインクルードファイルを指定します。 |
| -xprofile_ircache | (SPARC) -xprofile=collect で保存されたコンパイルデータを再使用します。 |
| -xprofile_pathmap | (SPARC) 1 つのプロファイルディレクトリに存在する複数のプログラムや共有ライブラリをサポートします。 |

## 3.3.3 コンパイル時とリンク時のオプション

次の表は、リンク時とコンパイル時の両方に指定する必要があるオプションを一覧表示します。

表 3–4 コンパイル時とリンク時のオプション

| オプション | 処理 |
|---|---|
| -fast | 実行可能コードの速度を向上させるコンパイルオプションの組み合わせを選択します。 |
| -m32\|-m64 | コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。 |
| -mt | --D_REENTRANT --lthread に展開されるマクロオプションです。 |
| -xarch | 命令セットアーキテクチャーを指定します。 |
| -xautopar | 複数プロセッサ用の自動並列化を有効にします。 |
| -xhwcprof | (SPARC) コンパイラのハードウェアカウンタによるプロファイリングのサポートを有効にします。 |
| -xipo | 内部手続き解析パスを呼び出すことにより、プログラム全体の最適化を実行します。 |
| -xlinker | リンカーオプションを指定します |
| -xlinkopt | 再配置可能なオブジェクトファイルのリンク時の最適化を実行します。 |

表 3–4 コンパイル時とリンク時のオプション (続き)

| オプション | 処理 |
|---|---|
| -xmemalign | (SPARC) メモリーの予想される最大境界整列と境界整列していないデータアクセスの動作を指定します。 |
| -xopenmp | 明示的な並列化のための OpenMP インタフェースをサポートします。これには、ソースコード指令のセット、実行時ライブラリルーチン、環境変数などが含まれます。 |
| -xpagesize | スタックとヒープの優先ページサイズを設定します。 |
| -xpagesize_heap | ヒープの優先ページサイズを設定します。 |
| -xpagesize_stack | スタックの優先ページサイズを設定します。 |
| -xpg | gprof(1) でプロファイル処理するためのデータを収集するオブジェクトコードを用意します。 |
| -xprofile | プロファイルのデータを収集、または最適化のためにプロファイルを使用します。 |
| -xvector=lib | ベクトルライブラリ関数の呼び出しの自動生成を有効にします。 |

## 3.3.4 デバッグオプション

表 3–5 デバッグオプション

| オプション | 処理 |
|---|---|
| -### | -dryrun と同等です |
| +d | C++ インライン関数を展開しません。 |
| -dryrun | コンパイルするすべてのコンポーネントに対してドライバが発行するすべてのコマンドを表示します。 |
| -E | C++ ソースファイルにプリプロセッサのみを実行し、結果を stdout に送信します。コンパイルはしません。 |
| -g | デバッグ用にコンパイルします。 |
| -g0 | デバッグ用にコンパイルしますが、インライン機能は無効にしません。 |
| -H | インクルードされるファイルのパス名を出力します。 |

表 3–5 デバッグオプション (続き)

| オプション | 処理 |
|---|---|
| `-keeptmp` | コンパイル中に作成されたすべての一時ファイルを残します。 |
| `-P` | ソースの前処理だけを行い、`.i` ファイルに出力します。 |
| `-Qoption` | オプションをコンパイル中の各処理に直接渡します。 |
| `-s` | 実行可能ファイルからシンボルテーブルを取り除きます。 |
| `-temp=`*dir* | 一時ファイルのディレクトリを定義します。 |
| `-verbose=`*vlst* | コンパイラの冗長性を制御します。 |
| `-xcheck` | スタックオーバーフローの実行時検査を追加します。 |
| `-xdumpmacros` | 定義内容、定義および解除された位置、使用されている場所に関する情報を出力します。 |
| `-xe` | 構文と意味のエラーのチェックだけを行います。 |
| `-xhelp=flags` | コンパイラオプションの要約を一覧表示します。 |
| `-xport64` | 32 ビットアーキテクチャーから 64 ビットアーキテクチャーへの移植中の一般障害について警告します。 |

## 3.3.5 浮動小数点オプション

表 3–6 浮動小数点オプション

| オプション | 処理 |
|---|---|
| `-fma` | (SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有効にします。 |
| `-fns[={no\|yes}]` | (SPARC) SPARC 非標準浮動小数点モードを有効または無効にします。 |
| `-fprecision=`*p* | *x86*: 浮動小数点精度モードを設定します。 |
| `-fround=`*r* | 起動時に IEEE 丸めモードを有効にします。 |
| `-fsimple=`*n* | 浮動小数点最適化の設定を行います。 |
| `-fstore` | *x86*: 浮動小数点式の精度を強制的に使用します。 |

表 3–6 浮動小数点オプション (続き)

| オプション | 処理 |
|---|---|
| -ftrap=*tlst* | 起動時に IEEE トラップモードを有効に設定します。 |
| -nofstore | *x86*: 強制された式の精度を無効にします。 |
| -xlibmieee | 例外時に libm が数学ルーチンに対し IEEE 754 値を返します。 |

## 3.3.6 言語オプション

表 3–7 言語オプション

| オプション | 処理 |
|---|---|
| -compat | コンパイラの主要リリースとの互換モードを設定します。 |
| -features=*alst* | C++ の各機能を有効化または無効化します。 |
| -xchar | 文字型が符号なしと定義されているシステムからのコードの移行を容易に行えるようにします。 |
| -xldscope | 共有ライブラリをより速くより安全に作成するため、変数と関数の定義のデフォルトリンカースコープを制御します。 |
| -xthreadvar | (SPARC) デフォルトのスレッドローカルな記憶装置へのアクセスモードを変更します。 |
| -xtrigraphs | 文字表記シーケンスを認識します。 |
| -xustr | 16 ビット文字で構成された文字リテラルを認識します。 |

## 3.3.7 ライブラリオプション

表 3–8 ライブラリオプション

| オプション | 処理 |
|---|---|
| -B*binding* | ライブラリのリンク形式を、シンボリック、動的、静的のいずれかから指定します。 |
| -d{y\|n} | 実行可能ファイル全体に対して動的ライブラリを使用できるかどうか指定します。 |
| -G | 実行可能ファイルではなく動的共有ライブラリを構築します。 |
| -h*name* | 生成される動的共有ライブラリに内部名を割り当てます。 |

表 3–8 ライブラリオプション (続き)

| オプション | 処理 |
|---|---|
| -i | ld(1) がどのような LD_LIBRARY_PATH 設定も無視します。 |
| -L*dir* | *dir* に指定したディレクトリを、ライブラリの検索に使用するディレクトリとして追加します。 |
| -l*lib* | リンカーのライブラリ検索リストに lib*lib*.a または lib*lib*.so を追加します。 |
| -library=*llst* | 特定のライブラリとそれに対応するファイルをコンパイルとリンクに強制的に組み込みます。 |
| -mt | マルチスレッド化したコードのコンパイルとリンクを行います。 |
| -norunpath | ライブラリのパスを実行可能ファイルに組み込みません。 |
| -R*plst* | 動的ライブラリの検索パスを実行可能ファイルに組み込みます。 |
| -staticlib=*llst* | 静的にリンクする C++ ライブラリを指定します。 |
| -xar | アーカイブライブラリを作成します。 |
| -xbuiltin[=*opt*] | 標準ライブラリ呼び出しの最適化を有効または無効にします。 |
| -xia | (Solaris) 適切な区間演算ライブラリをリンクし、浮動小数点環境を設定します。 |
| -xlang=*l*[,*l*] | 該当する実行時ライブラリをインクルードし、指定された言語に適切な実行時環境を用意します。 |
| -xlibmieee | 例外時に libm が数学ルーチンに対し IEEE 754 値を返します。 |
| -xlibmil | 最適化のために、選択された libm ライブラリルーチンをインライン展開します。 |
| -xlibmopt | 最適化された数学ルーチンライブラリを使用します。 |
| -xnolib | デフォルトのシステムライブラリとのリンクを無効にします。 |
| -xnolibmil | コマンド行の -xlibmil を取り消します。 |
| -xnolibmopt | 数学ルーチンのライブラリを使用しません。 |

### 3.3.8 廃止オプション

注 – 次のオプションは、現在は廃止されているためにコンパイラに受け入れられないか、将来のリリースではおそらく削除されます。

表 3–9 廃止オプション

| オプション | 処理 |
|---|---|
| -features=[%all\|%none] | %all と %none は廃止されたサブオプションです。 |
| -library=%all | 将来のリリースでおそらく削除される廃止されたサブオプションです。 |
| -xlic_lib=sunperf | Sun Performance Library にリンクするには、–library=sunperf を使用します。 |
| -xlicinfo | 非推奨。 |
| -xnativeconnect | 廃止。これに代わるオプションはありません。 |
| -xprefetch=yes | 代わりに - xprefetch=auto,explicit を使用します。 |
| -xprefetch=no | 代わりに -xprefetch=no%auto,no%explicit を使用します。 |
| -xvector=yes | 代わりに、--xvector=lib を使用します。 |
| -xvector=no | 代わりに、-xvector=none を使用します。 |

### 3.3.9 出力オプション

表 3–10 出力オプション

| オプション | 処理 |
|---|---|
| -c | コンパイルのみ。オブジェクト (.o) ファイルを作成しますが、リンクはしません。 |
| -dryrun | ドライバからコンパイラに対して発行されたコマンド行を表示しますが、コンパイルを行いません。 |
| -E | C++ ソースファイルにプリプロセッサのみを実行し、結果を stdout に送信します。コンパイルはしません。 |
| -erroff | コンパイラの警告メッセージを抑止します。 |
| -errtags | 各警告メッセージのメッセージタグを表示します。 |
| -errwarn | 指定の警告メッセージが出力されると、コンパイラはエラーステータスで終了します。 |

表 3–10 出力オプション (続き)

| オプション | 処理 |
|---|---|
| `-filt` | コンパイラがリンカーエラーメッセージに適用するフィルタリングを抑止します。 |
| `–G` | 実行可能ファイルではなく動的共有ライブラリを構築します。 |
| `–H` | インクルードされるファイルのパス名を出力します。 |
| `–migration` | 以前のコンパイラからの移行に関する情報の参照先を表示します。 |
| `-o` *filename* | 出力ファイルや実行可能ファイルの名前を *filename* にします。 |
| `-P` | ソースの前処理だけを行い、`.i` ファイルに出力します。 |
| `-Qproduce` *sourcetype* | `CC` ドライバに *sourcetype* (ソースタイプ) 型のソースコードを生成するよう指示します。 |
| `-s` | 実行可能ファイルからシンボルテーブルを取り除きます。 |
| `-verbose=`*vlst* | コンパイラの冗長性を制御します。 |
| `+w` | 必要に応じて追加の警告を出力します。 |
| `+w2` | 該当する場合は、より多くの警告を出力します。 |
| `–w` | 警告メッセージを抑止します。 |
| `-xdumpmacros` | 定義内容、定義および解除された位置、使用されている場所に関する情報を出力します。 |
| `-xe` | ソースファイルの構文と意味のチェックだけを行い、オブジェクトや実行可能コードは生成しません。 |
| `–xhelp=flags` | コンパイラオプションの要約を一覧表示します。 |
| `–xM` | メイクファイルの依存情報を出力します。 |
| `–xM1` | 依存情報を生成しますが、`/usr/include` は除きます。 |
| `–xtime` | コンパイル処理ごとの実行時間を報告します。 |
| `–xwe` | すべての警告をエラーに変換します。 |
| `-z` *arg* | リンカーオプション |

## 3.3.10 実行時パフォーマンスオプション

表 3–11 実行時パフォーマンスオプション

| オプション | 処理 |
|---|---|
| `-fast` | 一部のプログラムで最適な実行速度が得られるコンパイルオプションの組み合わせを選択します。 |
| `-fma` | (SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有 効にします。 |
| `-g` | パフォーマンスの解析 (およびデバッグ) に備えてプログラムを用意するようにコンパイラとリンカーの両方に指示します。 |
| `-s` | 実行可能ファイルからシンボルテーブルを取り除きます。 |
| `-m32`\|`-m64` | コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。 |
| `-xalias_level` | コンパイラで、型に基づく別名の解析および最適化を実行するように指定します。 |
| `-xarch=`*isa* | ターゲットのアーキテクチャー命令セットを指定します。 |
| `-xbinopt` | あとで最適化、変換、分析を行うために、バイナリを準備します。 |
| `-xbuiltin`[`=`*opt*] | 標準ライブラリ呼び出しの最適化を有効または無効にします。 |
| `-xcache=`*c* | (SPARC) オプティマイザのターゲットキャッシュプロパティーを定義します。 |
| `-xchip=`*c* | ターゲットのプロセッサチップを指定します。 |
| `-xF` | リンカーによる関数と変数の順序変更を有効にします。 |
| `-xinline=`*flst* | どのユーザーが作成したルーチンをオプティマイザでインライン化するかを指定します。 |
| `-xipo` | 内部手続きの最適化を実行します。 |
| `-xlibmil` | 最適化のために、選択された `libm` ライブラリルーチンをインライン展開します。 |
| `-xlibmopt` | 最適化された数学ルーチンライブラリを使用します。 |
| `-xlinkopt` | (SPARC) オブジェクトファイル内のあらゆる最適化のほかに、結果として出力される実行可能ファイルや動的ライブラリのリンク時最適化も行います。 |

表 3–11 実行時パフォーマンスオプション (続き)

| オプション | 処理 |
|---|---|
| -xmemalign=*ab* | (SPARC) メモリーの予想される最大境界整列と境界整列していないデータアクセスの動作を指定します。 |
| -xnolibmil | コマンド行の -xlibmil を取り消します。 |
| -xnolibmopt | 数学ルーチンのライブラリを使用しません。 |
| -x*Olevel* | 最適化レベルを *level* にします。 |
| -xpagesize | スタックとヒープの優先ページサイズを設定します。 |
| -xpagesize_heap | ヒープの優先ページサイズを設定します。 |
| -xpagesize_stack | スタックの優先ページサイズを設定します。 |
| -xprefetch[=*lst*] | 先読みをサポートするアーキテクチャーで先読み命令を有効にします。 |
| -xprefetch_level | -xprefetch=auto を設定したときの先読み命令の自動挿入を制御します。 |
| -xprofile | 実行時プロファイルデータを使って収集または最適化を実行します。 |
| -xregs=*rlst* | 一時レジスタの使用を制御します。 |
| -xsafe=mem | (SPARC) メモリーに関するトラップを起こさないものとします。 |
| -xspace | (SPARC) コードサイズが大きくなるような最適化は行いません。 |
| -xtarget=*t* | ターゲットの命令セットと最適化のシステムを指定します。 |
| -xthreadvar | デフォルトのスレッドローカル記憶装置アクセスモードを変更します。 |
| -xunroll=*n* | 可能な場合は、ループを展開します。 |
| -xvis | (SPARC) VIS 命令セットに定義されているアセンブリ言語テンプレートをコンパイラが認識します。 |

## 3.3.11 プリプロセッサオプション

表 3–12 プリプロセッサオプション

| オプション | 処理 |
|---|---|
| -D*name*[= *def*] | シンボル *name* をプリプロセッサに定義します。 |

表 3–12 プリプロセッサオプション (続き)

| オプション | 処理 |
|---|---|
| -E | C++ ソースファイルにプリプロセッサのみを実行し、結果を stdout に送信します。コンパイルはしません。 |
| -H | インクルードされるファイルのパス名を出力します。 |
| -P | ソースの前処理だけを行い、.i ファイルに出力します。 |
| -U*name* | プリプロセッサシンボル *name* の初期定義を削除します。 |
| -xM | メイクファイルの依存情報を出力します。 |
| -xM1 | 依存情報を生成しますが、/usr/include は除きます。 |

## 3.3.12 プロファイルオプション

表 3–13 プロファイルオプション

| オプション | 処理 |
|---|---|
| -p | prof でプロファイル処理するためのデータを収集するオブジェクトコードを用意します。 |
| -xpg | gprof プロファイラによるプロファイル処理用にコンパイルします。 |
| -xprofile | 実行時プロファイルデータを使って収集または最適化を実行します。 |

## 3.3.13 リファレンスオプション

表 3–14 リファレンスオプション

| オプション | 処理 |
|---|---|
| -xhelp=flags | コンパイラオプションの要約を一覧表示します。 |

## 3.3.14 ソースオプション

表 3–15 ソースオプション

| オプション | 処理 |
|---|---|
| -H | インクルードされるファイルのパス名を出力します。 |

表 3–15 ソースオプション (続き)

| オプション | 処理 |
|---|---|
| -I*pathname* | include ファイル検索パスに *pathname* を追加します。 |
| -I- | インクルードファイル検索規則を変更します。 |
| -xM | メイクファイルの依存情報を出力します。 |
| -xM1 | 依存情報を生成しますが、/usr/include は除きます。 |

### 3.3.15 テンプレートオプション

表 3–16 テンプレートオプション

| オプション | 処理 |
|---|---|
| -instances=*a* | テンプレートインスタンスの位置とリンケージを制御します。 |
| -template=*wlst* | さまざまなテンプレートオプションを有効または無効にします。 |

### 3.3.16 スレッドオプション

表 3–17 スレッドオプション

| オプション | 処理 |
|---|---|
| -mt | マルチスレッド化したコードのコンパイルとリンクを行います。 |
| -xsafe=mem | (SPARC) メモリーに関するトラップを起こさないものとします。 |
| -xthreadvar | (SPARC) デフォルトのスレッドローカルな記憶装置へのアクセスモードを変更します。 |

## 3.4 ユーザー指定のデフォルトオプションファイル

これらのデフォルトコンパイラオプションファイルは、ユーザーがすべてのコンパイルに適用される (ほかの方法で上書きされる場合を除く)、一連のデフォルトオプションを指定することを許可します。たとえば、ファイルがすべてのコンパイルを –xO2 をデフォルトで行うことを指定したり、ファイル setup.il を自動的にインクルードしたりできます。

コンパイラは起動時に、すべてのコンパイルに含めるべきデフォルトオプションがリストされているデフォルトオプションファイルを検索します。環境変数 SPRO_DEFAULTS_PATH は、デフォルトファイルを検索するディレクトリのコロン区切りリストを指定します。

環境変数が設定されていない場合、標準のデフォルトセットが使用されます。環境変数が設定されているが空の場合、デフォルトは使用されません。

デフォルトファイルの名前は compiler.defaults の形式である必要があります。compiler は cc、c89、c99、CC、ftn、または lint のいずれかです。たとえば、C++ コンパイラのデフォルトは CC.defaults です

SPRO_DEFAULTS_PATH にリストされたディレクトリにコンパイラ用のデフォルトファイルが見つかった場合、コンパイラはファイルを読み取り、コマンド行でオプションを処理する前にオプションを処理します。最初に見つかったデフォルトファイルが使用され、検索は終了します。

システム管理者は、システム全体のデフォルトファイルを *Studio-install-path*/prod/etc/config に作成してもかまいません。環境変数が設定されている場合、インストールされたデフォルトファイルは読み取られません。

デフォルトファイルの形式はコマンド行と同様です。ファイルの各行には、1 つ以上のコンパイラオプションを空白で区切って含めてもかまいません。ワイルドカードや置換などのシェル展開は、デフォルトファイル内のオプションには適用されません。

–#、–###、および –dryrun の各オプションによって生成される詳細出力では、SPRO_DEFAULTS_PATH の値と、完全展開されたコマンド行が表示されます。

コマンド行でユーザーが指定するオプションは通常、デフォルトファイルから読み取られるオプションより優先されます。たとえば、–xO4 でのコンパイルがデフォルトファイルで指定されており、ユーザーがコマンド行で –xO2 を指定した場合、–xO2 が使用されます。

デフォルトオプションファイルに記載されているオプションの一部は、コマンド行で指定されたオプションのあとに付加されます。これらは、プリプロセッサオプション –I、リンカーオプション –B、–L、–R、 –l、および、ソースファイル、オブジェクトファイル、アーカイブ、共有オブジェクトなどのすべてのファイル引数です。

次に示すのは、ユーザー指定のデフォルトコンパイラオプション起動ファイルがどのように使用される可能性があるかの例です。

```
demo% cat /project/defaults/CC.defaults
-I/project/src/hdrs –L/project/libs –llibproj –xvpara
demo% setenv SPRO_DEFAULTS_PATH  /project/defaults
demo% CC –c –I/local/hdrs –L/local/libs –lliblocal tst.c
```

現在、このコマンドは次と同義です。

```
CC -fast –xvpara –c –I/local/hdrs –L/local/libs –lliblocal tst.c \
     –I/project/src/hdrs –L/project/libs –llibproj
```

コンパイラデフォルトファイルはプロジェクト全体のデフォルトを設定するための便利な方法ですが、問題の診断を困難にする原因になる場合もあります。このような問題を回避するには、環境変数 SPRO_DEFAULTS_PATH を現在のディレクトリではなく絶対パスに設定します。

デフォルトオプションファイルのインタフェース安定性はコミットされていません。オプション処理の順序は、将来のリリースで変更される可能性があります。

パート II

# C++プログラムの作成

◆ ◆ ◆ 第 4 章

# 4 言語拡張

この章では、このコンパイラ特有の言語拡張について説明します。この章で扱っている機能のなかには、コマンド行でコンパイラオプションを指定しないかぎり、コンパイラが認識しないものがあります。関連するコンパイラオプションは、各セクションに適宜記載します。

`-features=extensions` オプションを使用すると、ほかの C++ コンパイラで一般的に認められている非標準コードをコンパイルすることができます。このオプションは、不正なコードをコンパイルする必要があり、そのコードを変更することが認められていない場合に使用することができます。

この章では、`-features=extensions` オプションを使用したときにコンパイラがサポートする言語拡張について説明します。

---

注 – 不正なコードは、どのコンパイラでも受け入れられる有効なコードに簡単に変更することができます。コードの変更が認められている場合は、このオプションを使用する代わりに、コードを有効なものに変更してください。`-features=extensions` オプションを使用すると、コンパイラによっては受け入れられない不正なコードが残ることになります。

---

## 4.1 リンカースコープ

次の宣言指定子を、外部シンボルの宣言や定義の制約のために使用します。静的なアーカイブやオブジェクトファイルに対して指定したスコープは、共有ライブラリや実行可能ファイルにリンクされるまで、適用されません。しかしながら、コンパイラは、与えられたリンカースコープ指定子に応じたいくつかの最適化を行うことができます。

これらの指示子を使うと、リンカースコープのマップファイルは使用しなくてすみます。 `-xldscope` をコマンド行で指定することによって、変数スコープのデフォルト設定を制御することもできます。

詳細は、274 ページの「A.2.130 -xldscope={*v*}」を参照してください。

表 4–1　リンカースコープ宣言指定子

| 値 | 意味 |
|---|---|
| `__global` | シンボル定義には大域リンカースコープとなります。これは、もっとも限定的でないリンカースコープです。シンボル参照はすべて、そのシンボルが定義されている最初の動的ロードモジュール内の定義と結合します。このリンカースコープは、extern シンボルの現在のリンカースコープです。 |
| `__symbolic` | シンボル定義は、シンボリックリンカースコープとなります。これは、大域リンカースコープより限定的です。リンク対象の動的ロードモジュール内からのシンボルへの参照はすべて、そのモジュール内に定義されているシンボルと結合します。モジュールの外側では、シンボルは大域と同じです。このリンカースコープはリンカーオプション `-Bsymbolic` に対応します。C++ ライブラリでは `-Bsymbolic` を使用できませんが、`__symbolic` 指定子は問題なく使用できます。リンカーの詳細については、`ld`(1) のマニュアルページを参照してください。 |
| `__hidden` | シンボル定義は、隠蔽リンカースコープとなります。隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも制限されたリンカースコープです。動的ロードモジュール内の参照はすべて、そのモジュール内の定義に結合します。シンボルはモジュールの外側では認識されません。 |

より限定的な指定子を使ってシンボル定義を宣言しなおすことはできますが、より限定的でない指定子を使って宣言しなおすことはできません。シンボルは一度定義したら、異なる指示子で宣言することはできません。

`__global` はもっとも制限の少ないスコープです。`__symbolic` はより制限されたスコープです。`__hidden` はもっとも制限の多いスコープです。

仮想関数の宣言は仮想テーブルの構造と解釈に影響を及ぼすので、あらゆる仮想関数は、クラス定義を含んでいるあらゆるコンパイル単位から認識される必要があります。

C++ クラスでは、仮想テーブルや実行時型情報といった暗黙の情報の生成が必要となることがあるため、構造体、クラス、および共用体の宣言と定義にリンカースコープ指定子を適用できます。その場合、指定子は、構造体、クラス、または共用体キーワードの直後に置きます。こういったアプリケーションでは、すべての暗黙のメンバーに対して 1 つのリンカースコーピングが適用されます。

### 4.1.1 Microsoft Windows との互換性

動的ライブラリに関して Microsoft Visual C++ (MSVC++) に含まれる類似のスコープ機能との互換性を保つため、次の構文もサポートされています。

`__declspec(dllexport)` は `__symbolic` と同一です。

`__declspec(dllimport)` は `__global` と同一です。

Oracle Solaris Studio C++ でこの構文の利点を活用するときは、`-xldscope=hidden` オプションを `CC` コマンド行に追加するようにしてください。結果は、MSVC++ を使用する場合と比較可能なものになります。MSVC++ を使用する場合は、定義ではなく外部シンボルの宣言のみに関して `__declspec(dllimport)` が使用されることが想定されます。例:

```
__declspec(dllimport) int foo(); // OK
__declspec(dllimport) int bar() { ... } // not OK
```

MSVC++ は、定義に対する `dllimport` の許容が緩やかであり、Oracle Solaris Studio C++ を使用する場合の結果と異なります。特に、Oracle Solaris Studio C++ を使用して定義に対して `dllimport` を使用すると、シンボルがシンボリックリンケージではなくグローバルリンケージを持つ結果になります。Microsoft Windows 上の動的ライブラリは、シンボルのグローバルリンケージをサポートしません。この問題がある場合は、定義に対して `dllimport` ではなく `dllexport` を使用するようにソースコードを変更できます。その後、MSVC++ と Oracle Solaris Studio C++ で同じ結果を得ることができます。

## 4.2 スレッドローカルな記憶装置

スレッドローカルの変数を宣言して、スレッドローカルな記憶領域を利用します。スレッドローカルな変数の宣言は、通常の変数宣言に宣言指定子 `__thread` を加えたものです。詳細については、320 ページの「A.2.174 -xthreadvar[=*o*]」 を参照してください。

`__thread` 指定子は、スレッド変数の最初の宣言部分に含める必要があります。`__thread` 指定子で宣言した変数は、`__thread` 指定子がない場合と同じように結合されます。

`__thread` 指定子で宣言できるのは、静的期間を持つ変数だけです。静的期間を持つ変数とは、ファイル内で大域なもの、ファイル内で静的なもの、関数内ローカルでかつ静的なもの、クラスの静的メンバーなどが含まれます。期間が動的または自動である変数を `__thread` 指定子を使って宣言することは避けてください。スレッド変数に静的な初期設定子を持たせることはできますが、動的な初期設定子あるいはデストラクタを持たせることはできません。たとえば、`__thread int x = 4;` を使用することはできますが、`__thread int x = f();` を使用することはできません。スレッド変数には、特殊なコンストラクタやデストラクタを持たせるべきではありません。とくに `std::string` 型をスレッド変数として持たせることはできません。

スレッド変数のアドレス演算子 (&) は、実行時に評価され、現在のスレッドの変数のアドレスが返されます。したがって、スレッド変数のアドレスは定数ではありません。

スレッド変数のアドレスは、対応するスレッドの有効期間の間は安定しています。変数の有効期間内は、プロセス内の任意のスレッドがスレッド変数のアドレスを自由に使用できます。スレッドが終了したあとは、スレッド変数のアドレスを使用できません。スレッドの変数のアドレスは、スレッドが終了するとすべて無効となります。

## 4.3 例外の制限の少ない仮想関数による置き換え

C++ 標準では、関数を仮想関数で置き換える場合に、置き換える側の仮想関数で、置き換えられる側の関数より制限の少ない例外を指定することはできません。置き換える側の関数の例外指定は、置き換えられる側の関数と同じか、それよりも制限されている必要があります。例外指定がないと、あらゆる例外が認められてしまうことに注意してください。

たとえば、基底クラスのポインタを使用して関数を呼び出す場合を考えてみましょう。その関数に例外指定が含まれていれば、それ以外の例外が送出されることはありません。置き換える側の関数で、それよりも制限の少ない例外指定が定義されている場合は、予期しない例外がスローされる可能性があり、予期しないプログラムの動作に続いてプログラムが異常終了する結果となることがあります。

`-features=extensions` オプションを使用すると、限定の少ない例外指定を含んだ関数による置き換えが認められます。

## 4.4 enum の型と変数の前方宣言の実行

`-features=extensions` オプションを使用すると、コンパイラにより `enum` の型と変数の前方宣言が認められます。さらに、不完全な `enum` 型による変数宣言も認められます。不完全な `enum` 型は、現行のプラットフォームの `int` 型と同じサイズと範囲を持つと想定されます。

次の 2 つの行は、`-features=extensions` オプションを使用した場合にコンパイルされる不正なコードの例です。

```
enum E; // invalid: forward declaration of enum not allowed
E e;    // invalid: type E is incomplete
```

`enum` 定義では、ほかの `enum` 定義を参照できず、ほかの型の相互参照もできないため、列挙型の前方宣言は必要ありません。コードを有効なものにするには、`enum` を使用する前に、その定義を完全なものにしておきます。

注 – 64 ビットアーキテクチャーでは、enum は int よりも大きなサイズが必要となる場合があります。その場合に、前方宣言と定義が同じコンパイルの中で見つかると、コンパイラエラーが発生します。実際のサイズが想定されたサイズと異なっていて、コンパイラがそのことを検出できない場合は、コードのコンパイルとリンクは行われますが、実際のプログラムが正しく動作する保証はありません。8 バイト値が 4 バイト変数に格納される場合は特に、予期しないプログラムの動作が発生する可能性があります。

## 4.5 不完全な enum 型の使用

-features=extensions オプションを使用した場合は、不完全な enum 型は前方宣言と見なされます。たとえば、-features=extensions オプションを使用すると、次の不正なコードがコンパイルされます。

```
typedef enum E F; // invalid, E is incomplete
```

前述したように、enum 型を使用する前に、その定義を記述しておくことができます。

## 4.6 enum 名のスコープ修飾子としての使用

enum 宣言ではスコープを指定できないため、enum 名をスコープ修飾子として使用することはできません。たとえば、次のコードは不正です。

```
enum E {e1, e2, e3};
int i = E::e1; // invalid: E is not a scope name
```

この不正なコードをコンパイルするには、-features=extensions オプションを使用します。enum 型の名前の場合に、-features=extensions オプションはコンパイラにスコープ修飾子を無視するよう命令します。

このコードを有効なものにするには、不正な修飾子 E:: を取り除きます。

注 – このオプションを使用すると、プログラムのタイプミスが検出されずにコンパイルされる可能性が高くなります。

# 4.7 名前のない struct 宣言の使用

名前のない構造体宣言は、構造体のタグも、オブジェクト名も、typedef 名も指定されていない宣言です。C++ では、名前のない構造体は認められていません。

-features=extensions オプションを使用すると、名前のない struct 宣言を使用できるようになりますが、これは共用体のメンバーとしてのみ使用できます。

次は、-features=extensions オプションを使用した場合にコンパイルが可能な、名前のない不正な struct 宣言の例です。

```
union U {
  struct {
    int a;
    double b;
  };  // invalid: anonymous struct
  struct {
    char* c;
    unsigned d;
  };  // invalid: anonymous struct
};
```

これらの構造体のメンバー名は、構造体名で修飾しなくても認識されます。たとえば、共用体 U が前述のコードのように定義されているとすると、次のような記述が可能です。

```
U u;
u.a = 1;
```

名前のない構造体は、名前のない共用体と同じ制約を受けます。

コードを有効にするには、次のようにそれぞれの struct に名前を付けます。

```
union U {
  struct {
    int a;
    double b;
  } A;
  struct {
    char* c;
    unsigned d;
  } B;
};
U u;
U.A.a = 1;
```

## 4.8 名前のないクラスインスタンスのアドレスの受け渡し

一時変数のアドレスは取得できません。たとえば、次のコードは不正です。コンストラクタ呼び出しによって作成された変数のアドレスが取得されてしまうからです。ただし、-features=extensions オプションを使用した場合は、この不正なコードもコンパイル可能になります。

```
class C {
  public:
    C(int);
    ...
};
void f1(C*);
int main()
{
  f1(&C(2)); // invalid
}
```

このコードを有効なものにするには、次のように明示的な変数を使用します。

```
C c(2);
f1(&c);
```

一時オブジェクトは、関数が終了したときに破棄されます。一時変数のアドレスを取得しないようにするのは、プログラムの作成者の責任になります。また、(f1 などで) 一時変数に格納されたデータは、その変数が破棄されたときに失われます。

## 4.9 静的名前空間スコープ関数のクラスフレンドとしての宣言

次のコードは不正です。

```
class A {
  friend static void foo(<args>);
  ...
};
```

クラス名に外部リンケージが含まれており、また、すべての定義が同一でなければならないため、フレンド関数にも外部リンケージが含まれている必要があります。しかし、-features=extensions オプションを使用すると、このコードもコンパイルできるようになります。

おそらく、この不正なコードの目的は、クラス A の実装ファイルに、メンバーではない「ヘルパー」関数を組み込むことでしょう。そうであれば、foo を静的メンバー関数にしても結果は同じです。クライアントから呼び出せないように、この関数を非公開にすることもできます。

注 – この拡張機能を使用すると、作成したクラスを任意のクライアントが「横取り」できるようになります。そのためには、任意のクライアントにこのクラスのヘッダーを組み込み、独自の静的関数 foo を定義します。この関数は、自動的にこのクラスのフレンド関数になります。その結果は、このクラスのメンバーをすべて公開にした場合と同じになります。

## 4.10 事前定義済み __func__ シンボルの関数名としての使用

コンパイラでは、それぞれの関数で __func__ 識別子が const char 型の静的配列として暗黙的に宣言されます。プログラムの中で、この識別子が使用されていると、コンパイラによって次の定義が追加されます。ここで、*function-name* は関数の単純名です。この名前には、クラスメンバーシップ、名前空間、多重定義の情報は反映されません。

```
static const char __func__[] = "function-name";
```

たとえば、次のコードを考えてみましょう。

```
#include <stdio.h>
void myfunc(void)
{
  printf("%s\n", __func__);
}
```

この関数が呼び出されるたびに、標準出力ストリームに次の情報が出力されます。

```
myfunc
```

識別子 __FUNCTION__ も定義され、__func__ と同等になります。

## 4.11 サポートされる属性

次の属性はサポートされます。__attribute__ (( *keyword*)) によって、あるいは [[ *keyword*]] によって呼び出される次の属性は、互換性のためにコンパイラによって実装されます。属性キーワードを二重アンダースコアの間に記載する __ *keyword*__ も受け入れられます。

| | |
|---|---|
| aligned | #pragma align とほぼ同等です。警告を生成し、可変長配列について使用される場合は無視されます。 |
| always_inline | #pragma inline および -xinline と同等です |
| const | #pragma no_side_effect と同等です |

| | |
|---|---|
| constructor | #pragma init と同等です |
| destructor | #pragma fini と同等です |
| malloc | #pragma returns_new_memory と同等です |
| mode | (相当するものはありません) |
| noinline | #pragma no_inline および -xinline と同等です |
| noreturn | #pragma does_not_return と同等です |
| pure | #pragma does_not_write_global_data と同等です |
| packed | #pragma pack() と同等です。次の詳細を参照してください。 |
| returns_twice | #pragma unknown_control_flow と同等です |
| strong | g++ との互換のために受け入れますが、効果はありません。g++ のドキュメントではこの属性を使用しないことが推奨されています。 |
| vector_size | 変数または (typedef を使用して作成された) 型の名前がベクトルを表していることを示します。 |
| visibility | リンカースコープを指定します。(274 ページの「A.2.130 -xldscope={*v*}」 を参照してください) 構文は、__attribute__((visibility("*visibility-type*"))) です。ここで、*visibility-type* は次のいずれかです。<br>default — __global リンカースコープと同じです<br>hidden — __hidden リンカースコープと同じです<br>internal — __symbolic リンカースコープと同じです |
| weak | #pragma weak と同等です |

## 4.11.1 __packed__ 属性の詳細

struct または union の型定義に添付されるこの属性は、必要なメモリーを最小限に抑えるために、構造体または共用体の各メンバー (幅が 0 のビットフィールドを除く) の配置を指定します。enum 定義に添付する場合は、__packed__ は最小の整数型を使用すべきことを示します。

struct および union 型に対してこの属性を指定する場合は、構造体または共用体の各メンバーに対して packed 属性を指定する場合と同じことを意味します。

次の例では、struct my_packed_struct のメンバーは互いに隣接してパックされますが、メンバーの内部レイアウトはパックされません。内部レイアウトをパックするには、struct my_unpacked_struct もパックする必要があります。

```
struct my_unpacked_struct
{
   char c;
   int i;
;

struct __attribute__ ((__packed__)) my_packed_struct
{
   char c;
   int  i;
   struct my_unpacked_struct s;
};
```

この属性を指定できるのは、enum、struct、または union の定義のみです。列挙型、構造体、共用体のいずれも定義しない typedef に対して、この属性は指定できません。

## 4.12 Intel MMX および拡張 x86 プラットフォーム組み込み関数のためのコンパイラサポート

mmintrin.h ヘッダーファイル内で宣言されたプロトタイプは Intel MMX 組み込み関数をサポートし、互換性のために提供されています。

次の表に示すように、特定のヘッダーファイルは、追加の拡張プラットフォーム組み込み関数のプロトタイプを提供します。

表 4–2 ヘッダーファイル

| x86 プラットフォーム | ヘッダーファイル |
|---|---|
| SSE | mmintrin.h |
| SSE2 | xmmintrin.h |
| SSE3 | pmmintrin.h |
| SSSE3 | tmmintrin.h |
| SSE4A | ammintrin.h |
| SSE4.1 | smmintrin.h |
| SSE4.2 | nmmintrin.h |
| AES 暗号化および PCLMULQDQ | wmmintrin.h |

表 4–2　ヘッダーファイル　　(続き)

| x86 プラットフォーム | ヘッダーファイル |
|---|---|
| AVX | immintrin.h |

各ヘッダーファイルは、表内でそれより前にあるプロトタイプをインクルードします。たとえば、SSE4.1 プラットフォーム上では、smmintrin.h は tmmintrin.h をインクルードし、これは pmmintrin.h をインクルードし、この関係が mmintrin.h まで続くため、smmintrin.h をユーザープログラムにインクルードすることで、SSE4.1、SSSE3、SSE3、SSE2、SSE、および MMX プラットフォームをサポートする組み込み関数名が宣言されます。

ammintrin.h は AMD が発行したものであり、Intel 組み込み関数ヘッダーからは一切インクルードされないことに注意してください。ammintrin.h は pmmintrin.h をインクルードするため、ammintrin.h をインクルードすれば、すべての AMD SSE4A だけでなく、Intel の SSE3、SSE2、SSE、および MMX 関数も宣言されます。

また、単一の Oracle Solaris Studio ヘッダーファイル sunmedia_intrin.h は、Intel ヘッダーファイルの宣言はすべてインクルードしますが、AMD ヘッダーファイル ammintrin.h はインクルードしません。

ホストプラットフォーム (SSE3 など) に配備され、任意のスーパーセット組み込み関数 (AVX についてのものなど) を呼び出すコードは、Oracle Solaris プラットフォームにロードされず、Linux プラットフォーム上で未定義の動作または正しくない結果が発生して失敗することがあることに注意してください。これらのプラットフォーム固有の組み込み関数を呼び出すプログラムは、それらの関数をサポートするプラットフォームにのみ配備してください。

これらはシステムヘッダーファイルであり、次の例に示すようにプログラムに現れるようにしてください。

```
#include <nmmintrin.h>
```

詳細については、最新の Intel C++ コンパイラリファレンスガイドを参照してください。

◆ ◆ ◆ 第 5 章

# 5 プログラムの編成

C++ プログラムのファイル編成は、C プログラムの場合よりも慎重に行う必要があります。この章では、ヘッダーファイルとテンプレート定義の設定方法について説明します。

## 5.1 ヘッダーファイル

有効なヘッダーファイルを簡単に作成できるとはかぎりません。場合によっては、C と C++ の複数のバージョンで使用可能なヘッダーファイルを作成する必要があります。また、テンプレートを使用するためには、複数回の包含 (べき等) が可能なヘッダーファイルが必要です。

### 5.1.1 言語に対応したヘッダーファイル

場合によっては、C と C++ の両方のプログラムにインクルード可能なヘッダーファイルを作成する必要があります。ただし、従来の C とも呼ばれる Kernighan & Ritchie C (K&R C) や、ANSI C、『Annotated Reference Manual』C++ (ARM C++)、および ISO C++ では、1 つのヘッダーファイル内の同一のプログラム要素について異なった宣言や定義が規定されていることがあります。言語とバージョンによる違いについての詳細は、『C++ 移行ガイド』を参照してください。これらのどの標準言語でもヘッダーファイルで使用できるようにするには、プリプロセッサマクロ `__STDC__` や `__cplusplus` の定義の有無またはその値に基づいた条件付きコンパイルを使用する必要があります。

`__STDC__` マクロは、K&R C では定義されていませんが、ANSI C や C++ では定義されています。このマクロが定義されているかどうかを使用して、K&R C のコードを ANSI C や C++ のコードから区別します。このマクロは、プロトタイプの関数定義とプロトタイプではない関数定義を分離するときに特に役立ちます。

```
#ifdef __STDC__
int function(char*,...);      // C++ & ANSI C declaration
```

```
#else
int function();               // K&R C
#endif
```

__cplusplus マクロは、C では定義されていませんが、C++ では定義されています。

---

注 – 旧バージョンの C++ では、__cplusplus の代わりに c_plusplus マクロが定義されていました。c_plusplus マクロは、現在のバージョンでは定義されていません。

---

__cplusplus マクロが定義されているかどうかを使用して、C と C++ を区別します。このマクロは、次のように関数宣言用の extern "C" インタフェースを保護するときに特に便利です。extern "C" の指定の一貫性を保つには、extern "C" のリンケージ指定のスコープ内には #include 指令を含めないでください。

```
#include “header.h”
...                       // ... other include files...
#if defined(__cplusplus)
extern “C” {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

ARM C++ では、__cplusplus マクロの値は 1 です。ISO C++ では、このマクロの値は 199711L (long 定数で表現した、規格の年と月) です。この値の違いを使用して、ARM C++ と ISO C++ を区別します。これらのマクロ値は、テンプレート構文の違いを保護するときに特に役立ちます。

```
// template function specialization
#if __cplusplus < 199711L
int power(int,int);                       // ARM C++
#else
template <> int power(int,int);           // ISO C++
#endif
```

## 5.1.2 べき等ヘッダーファイル

ヘッダーファイルはべき等にするようにしてください。すなわち、同じヘッダーファイルを何回インクルードしても、1 回だけインクルードした場合と効果が同じになるようにしてください。このことは、テンプレートでは特に重要です。べき等を実現するもっともよい方法は、プリプロセッサの条件を設定し、ヘッダーファイルの本体の重複を防止することです。

```
#ifndef HEADER_H
#define HEADER_H
```

```
/* contents of header file */
#endif
```

## 5.2 テンプレート定義

テンプレート定義は 2 通りの方法で編成することができます。すなわち、テンプレート定義を取り込む方法 (定義取り込み型編成) と、分離する方法 (定義分離型編成) があります。テンプレート定義を取り込んだほうが、テンプレートのコンパイルを制御しやすくなります。

### 5.2.1 テンプレート定義の取り込み

テンプレートの宣言と定義を、そのテンプレートを使用するファイルの中に含める場合、編成が定義の取り込みです。例:

main.cc

```
template <class Number> Number twice(Number original);
template <class Number> Number twice(Number original )
    { return original + original; }
int main()
    { return twice<int>(-3); }
```

テンプレートを使用するファイルに、テンプレートの宣言と定義の両方を含んだファイルをインクルードした場合も、定義取り込み型編成を使用したことになります。例:

twice.h

```
#ifndef TWICE_H
#define TWICE_H
template <class Number>
Number twice(Number original);
template <class Number> Number twice( Number original )
    { return original + original; }
#endif
```

main.cc

```
#include “twice.h”
int main()
    { return twice(-3); }
```

---

注 – テンプレートヘッダーをべき等にすることは非常に重要です。76 ページの「5.1.2 べき等ヘッダーファイル」を参照してください。

---

## 5.2.2 テンプレート定義の分離

テンプレート定義を編成するもう一つの方法は、テンプレートの定義をテンプレート定義ファイルに記述することです。この例を次に示します。

twice.h

```
#ifndef TWICE_H
#define TWICE_H
template <class Number>
Number twice(Number original);
#endif TWICE_H
```

twice.cc

```
template <class Number>
Number twice( Number original )
    { return original + original; }
```

main.cc

```
#include “twice.h”
int main( )
    { return twice<int>( -3 ); }
```

テンプレート定義ファイルには、べき等ではないヘッダーファイルをインクルードしてはいけません。また、通常はテンプレート定義ファイルにヘッダーファイルをインクルードする必要はありません。76 ページの「5.1.2 べき等ヘッダーファイル」を参照してください。なお、テンプレートの定義分離型編成は、すべてのコンパイラでサポートされているわけではありません。

独立した定義ファイルはヘッダーファイルなので、多数のファイルに暗黙のうちにインクルードされることがあります。そのため、テンプレート定義の一部でないかぎり関数と変数はこのファイルに含めないようにします。独立した定義ファイルには、typedef などの型定義を定義できます。

---

注 – 通常、テンプレート定義ファイルには、ソースファイルの拡張子 (.c、.C、.cc、.cpp、.cxx、.c++ のいずれか) を付けますが、このテンプレート定義ファイルはヘッダーファイルです。コンパイラは、これらのファイルを必要に応じて自動的に取り込みます。テンプレート定義ファイルの単独コンパイルは行わないでください。

---

このように、テンプレート宣言とテンプレート定義を別々のファイルに配置した場合は、定義ファイルの内容、その名前、配置先に特に注意する必要があります。さらに、定義ファイルの配置先をコンパイラに明示的に通知する必要もあります。テンプレート定義の検索規則については、105 ページの「7.5 テンプレート定義の検索」を参照してください。

-E オプションまたは -P オプションを使用してプリプロセッサ出力を生成する場合、定義分離ファイルの構成では、テンプレート定義を .i ファイルに含めることが許可されません。見つからない定義があるため、.i ファイルのコンパイルに失敗します。テンプレート定義ファイルをテンプレート宣言ヘッダー (次のコード例を参照) に条件付きで含めることで、コマンド行で -template=no%extdef を使用することによりテンプレート定義を使用可能にできます。libCtd ライブラリと STLport ライブラリは、この方法で実装されます。

```
// templace declaration file
template <class T> class foo { ... };
#ifdef _TEMPLATE_NO_EXTDEF
#include "foo.cc"  //template definition file
#endif
```

ただし、マクロ _TEMPLATE_NO_EXTDEF を自分で定義しないでください。–template=no%extdef オプションなしで定義すると、テンプレート定義ファイルが複数含められるためにコンパイルエラーが発生することがあります。

◆ ◆ ◆ 第 6 章

# 6 テンプレートの作成と使用

テンプレートによって、コード本体を 1 つ書くだけで、型が保証された方法で広範囲の型にそれを適用できます。この章では関数テンプレートに関連したテンプレートの概念と用語を紹介し、より複雑な (そして、より強力な) クラステンプレートと、テンプレートの使用方法について説明しています。また、テンプレートのインスタンス化、デフォルトのテンプレートパラメータ、およびテンプレートの特殊化についても説明しています。この章の最後には、テンプレートの潜在的な問題が挙げられています。

## 6.1 関数テンプレート

関数テンプレートは、引数または戻り値の型だけが異なった、関連する複数の関数を記述したものです。

### 6.1.1 関数テンプレートの宣言

テンプレートは使用する前に宣言する必要があります。次の例のように、宣言によってテンプレートを使用するのに十分な情報は提供されますが、テンプレートを実装するには十分ではありません。

```
template <class Number> Number twice( Number original );
```

この例では *Number* はテンプレートパラメータであり、テンプレートが記述する関数の範囲を指定します。つまり、*Number* はテンプレート型のパラメータです。テンプレート定義内で使用すると、型はテンプレートを使用するときに特定されることになります。

### 6.1.2 関数テンプレートの定義

テンプレートは宣言と定義の両方が必要になります。テンプレートを定義することで、実装に必要な情報が得られます。次の例は、前述の例で宣言されたテンプレートを定義しています。

```
template <class Number> Number twice( Number original )
    { return original + original; }
```

テンプレート定義は通常ヘッダーファイルで行われるので、テンプレート定義が複数のコンパイル単位で繰り返される可能性があります。しかし、すべての定義は同じである必要があります。この制限は「単一定義ルール」と呼ばれています。

### 6.1.3 関数テンプレートの使用

テンプレートは、いったん宣言するとほかのすべての関数と同様に使用することができます。テンプレートの使用は、そのテンプレートの命名と関数引数の提供で構成されます。コンパイラは、テンプレート型引数を、関数引数の型から推測します。たとえば、以前に宣言されたテンプレートを次のように使用できます。

```
double twicedouble( double item )
    { return twice( item ); }
```

テンプレート引数が関数の引数型から推測できない場合、その関数が呼び出される場所にその引数を指定する必要があります。例:

```
template<class T> T func(); // no function arguments
int k = func<int>(); // template argument supplied explicitly
```

## 6.2 クラステンプレート

クラステンプレートは、複数の関連するクラスまたはデータ型を記述します。クラステンプレートに記述されているクラスは、型、整数値、大域リンケージによる変数へのポインタや参照だけが互いに異なっています。クラステンプレートは、一般的ではあるが型が保証されているデータ構造を記述するのに特に便利です。

### 6.2.1 クラステンプレートの宣言

クラステンプレートの宣言では、クラスの名前とそのテンプレート引数だけを指定します。このような宣言は「不完全なクラステンプレート」と呼ばれます。

次の例は、任意の型の引数をとる `Array` というクラスに対するテンプレート宣言の例です。

```
template <class Elem> class Array;
```

次のテンプレートは、unsigned int の引数をとる String というクラスに対する宣言です。

```
template <unsigned Size> class String;
```

## 6.2.2 クラステンプレートの定義

クラステンプレートの定義では、次の例のようにクラスデータと関数メンバーを宣言する必要があります。

```
template <class Elem> class Array {
        Elem* data;
        int size;
    public:
        Array( int sz );
        int GetSize();
        Elem& operator[]( int idx );
};

template <unsigned Size> class String {
        char data[Size];
        static int overflows;
    public:
        String( char *initial );
        int length();
};
```

関数テンプレートとは違って、クラステンプレートには class Elem のような型パラメータと unsigned Size のような式パラメータの両方を指定できます。式パラメータには次の情報を指定できます。

- 整数型または列挙型を持つ値
- オブジェクトへのポインタまたは参照
- 関数へのポインタまたは参照
- クラスメンバー関数へのポインタ

## 6.2.3 クラステンプレートメンバーの定義

クラステンプレートを完全に定義するには、その関数メンバーと静的データメンバーを定義する必要があります。動的 (静的でない) データメンバーの定義は、クラステンプレート宣言で十分です。

### 6.2.3.1 関数メンバーの定義

テンプレート関数メンバーの定義は、テンプレートパラメータの指定と、それに続く関数定義から構成されます。関数識別子は、クラステンプレートのクラス名とそ

のテンプレートの引数で修飾されます。次の例は、template <class Elem> というテンプレートパラメータ指定を持つ Array クラステンプレートの 2 つの関数メンバー定義を示しています。それぞれの関数識別子は、テンプレートクラス名とテンプレート引数 Array<Elem> で修飾されています。

```
template <class Elem> Array<Elem>::Array( int sz )
    {size = sz; data = new Elem[size];}

template <class Elem> int Array<Elem>::GetSize()
    { return size; }
```

次の例は、String クラステンプレートの関数メンバーの定義を示しています。

```
#include <string.h>
template <unsigned Size> int String<Size>::length( )
    {int len = 0;
      while (len < Size && data[len]!= '\0') len++;
      return len;}

template <unsigned Size> String<Size>::String(char *initial)
    {strncpy(data, initial, Size);
      if (length( ) == Size) overflows++;}
```

### 6.2.3.2 静的データメンバーの定義

テンプレートの静的データメンバーの定義は、テンプレートパラメータの指定と、それに続く変数定義から構成されます。この場合、変数識別子は、クラステンプレート名とそのテンプレートの実引数で修飾されます。

```
template <unsigned Size> int String<Size>::overflows = 0;
```

## 6.2.4 クラステンプレートの使用

テンプレートクラスは、型が使用できる場所ならどこででも使用できます。テンプレートクラスを指定するには、テンプレート名と引数の値を設定します。次の宣言例では、Array テンプレートに基づいた変数 int_array を作成します。この変数のクラス宣言とその一連のメソッドは、Elem が int に置き換わっている点以外は、Array テンプレートとまったく同じです。85 ページの「6.3 テンプレートのインスタンス化」を参照してください。

```
Array<int> int_array(100);
```

次の宣言例は、String テンプレートを使用して short_string 変数を作成します。

```
String<8> short_string("hello");
```

テンプレートクラスのメンバー関数は、ほかのすべてのメンバー関数と同じように使用できます。

```
int x = int_array.GetSize( );
```

```
int x = short_string.length( );
.
```

## 6.3 テンプレートのインスタンス化

テンプレートのインスタンス化には、特定の組み合わせのテンプレート引数に対応した具体的なクラスまたは関数 (インスタンス) を生成することが含まれます。たとえば、コンパイラは `Array<int>` クラスと `Array<double>` に対応した別々のクラスを生成します。これらの新しいクラスの定義では、テンプレートクラスの定義の中のテンプレートパラメータがテンプレート引数に置き換えられます。82 ページの「6.2 クラステンプレート」に示す `Array<int>` の例では、`Elem` が表示されるたびに、コンパイラが `int` を置き換えます。

### 6.3.1 テンプレートの暗黙的インタンス化

テンプレート関数またはテンプレートクラスを使用すると、インスタンス化が必要になります。そのインスタンスがまだ存在していない場合には、コンパイラはテンプレート引数に対応したテンプレートを暗黙的にインスタンス化します。

### 6.3.2 テンプレートの明示的インスタンス化

コンパイラは、実際に使用されるテンプレート引数に対応したテンプレートだけを暗黙的にインスタンス化します。これは、テンプレートを持つライブラリの作成には適していない可能性があります。C++ には、次の例のように、テンプレートを明示的にインスタンス化するための手段が用意されています。

#### 6.3.2.1 テンプレート関数の明示的インスタンス化

テンプレート関数を明示的にインスタンス化するには、`template` キーワードに続けて関数の宣言 (定義ではない) を行います。関数の宣言では関数識別子のあとにテンプレート引数を指定します。

```
template float twice<float>(float original);
```

テンプレート引数は、コンパイラが推測できる場合は省略できます。

```
template int twice(int original);
```

#### 6.3.2.2 テンプレートクラスの明示的インスタンス化

テンプレートクラスを明示的にインスタンス化するには、`template` キーワードに続けてクラスの宣言 (定義ではない) を行います。クラスの宣言ではクラス識別子のあとにテンプレート引数を指定します。

```
template class Array<char>;

template class String<19>;
```

クラスを明示的にインスタンス化すると、そのメンバーもすべてインスタンス化されます。

#### 6.3.2.3 テンプレートクラス関数メンバーの明示的インスタンス化

テンプレート関数メンバーを明示的にインスタンス化するには、`template` キーワードに続けて関数の宣言 (定義ではない) を行います。関数の宣言ではテンプレートクラスで修飾した関数識別子のあとにテンプレート引数を指定します。

```
template int Array<char>::GetSize();

template int String<19>::length();
```

#### 6.3.2.4 テンプレートクラスの静的データメンバーの明示的インスタンス化

テンプレートの静的データメンバーを明示的にインスタンス化するには、`template` キーワードに続けてメンバーの宣言 (定義ではない) を行います。メンバーの宣言では、テンプレートクラスで修飾したメンバー識別子のあとにテンプレート引数を指定します。

```
template int String<19>::overflows;
```

## 6.4 テンプレートの編成

テンプレートは、 入れ子にして使用できます。これは、標準 C++ ライブラリで行う場合のように、一般的なデータ構造に関する汎用関数を定義する場合に特に便利です。たとえば、テンプレート配列クラスに関して、テンプレートのソート関数を次のように宣言することができます。

```
template <class Elem> void sort(Array<Elem>);
```

また、次のように定義することができます。

```
template <class Elem> void sort(Array<Elem> store)
    {int num_elems = store.GetSize();
      for (int i = 0; i < num_elems-1; i++)
          for (int j = i+1; j < num_elems; j++)
              if (store[j-1] > store[j])
                  {Elem temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp;}}
```

前述の例は、事前に宣言された Array クラステンプレートのオブジェクトに関するソート関数を定義しています。次の例はソート関数の実際の使用例を示しています。

```
Array<int> int_array(100);   // construct an array of ints
sort(int_array);             // sort it
```

## 6.5 デフォルトのテンプレートパラメータ

クラステンプレートのテンプレートパラメータには、デフォルトの値を指定できます (関数テンプレートは不可)。

```
template <class Elem = int> class Array;
template <unsigned Size = 100> class String;
```

テンプレートパラメータにデフォルト値を指定する場合、それに続くパラメータもすべてデフォルト値である必要があります。テンプレートパラメータに指定できるデフォルト値は 1 つです。

## 6.6 テンプレートの特殊化

この節の twice の例のように、テンプレート引数を例外的に特定の形式で組み合わせると、パフォーマンスがいくらか改善されることがあります。あるいは、この節の sort の例のように、ある引数の組み合わせに対してはテンプレート記述を適用できないこともあります。テンプレートの特殊化によって、実際のテンプレート引数の特定の組み合わせに対して代替実装を定義することが可能になります。テンプレートの特殊化はデフォルトのインスタンス化を無効にします。

### 6.6.1 テンプレートの特殊化宣言

前述のようなテンプレート引数の組み合わせを使用するには、その前に特殊化を宣言する必要があります。次の例は twice と sort の特殊化された実装を宣言しています。

```
template <> unsigned twice<unsigned>( unsigned original );

template <> sort<char*>(Array<char*> store);
```

コンパイラがテンプレート引数を明確に確認できる場合には、次の例のようにテンプレート引数を省略することができます。例:

```
template <> unsigned twice(unsigned original);

template <> sort(Array<char*> store);
```

## 6.6.2 テンプレートの特殊化定義

宣言するテンプレートの特殊化はすべて定義する必要があります。次の例は、前の節で宣言された関数を定義しています。

```
template <> unsigned twice<unsigned>(unsigned original)
    {return original << 1;}

#include <string.h>
template <> void sort<char*>(Array<char*> store)
    {int num_elems = store.GetSize();
      for (int i = 0; i < num_elems-1; i++)
          for (int j = i+1; j < num_elems; j++)
              if (strcmp(store[j-1], store[j]) > 0)
                  {char *temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp;}}
```

## 6.6.3 テンプレートの特殊化の使用とインスタンス化

特殊化されたテンプレートはほかのすべてのテンプレートと同様に使用され、インスタンス化されます。ただし、完全に特殊化されたテンプレートの定義はインスタンス化でもあります。

## 6.6.4 部分特殊化

前述の例では、テンプレートは完全に特殊化されています。つまり、このようなテンプレートは特定のテンプレート引数に対する実装を定義しています。テンプレートは部分的に特殊化することも可能です。これは、テンプレートパラメータの一部だけを指定する、または、1 つまたは複数のパラメータを特定のカテゴリの型に制限することを意味します。部分特殊化の結果、それ自身はまだテンプレートのままです。たとえば、次のコード例に、本来のテンプレートとそのテンプレートの完全特殊化を示します。

```
template<class T, class U> class A {...}; //primary template
template<> class A<int, double> {...};    //specialization
```

次のコード例に、本来のテンプレートの部分特殊化を示します。

```
template<class U> class A<int> {...};          // Example 1
template<class T, class U> class A<T*> {...};  // Example 2
template<class T> class A<T**, char> {...};    // Example 3
```

- 例 1 は、最初のテンプレートパラメータが `int` 型である特殊なテンプレート定義です。
- 例 2 は、最初のテンプレートパラメータが任意のポインタ型である、特殊なテンプレート定義です。

- 例 3 は、最初のテンプレートパラメータが任意の型のポインタへのポインタであり、2 番目のテンプレートパラメータが char 型である、特殊なテンプレート定義です。

## 6.7 テンプレートの問題

この節では、テンプレートを使用する場合の問題について説明しています。

### 6.7.1 非局所型名前の解決とインスタンス化

テンプレート定義で使用される名前の中には、テンプレート引数によって、またはそのテンプレート内で、定義されていないものがある可能性があります。そのような場合にはコンパイラが、定義の時点で、またはインスタンス化の時点で、テンプレートを取り囲むスコープから名前を解決します。1 つの名前が複数の場所で異なる意味を持つために解決の形式が異なることも考えられます。

名前の解決は複雑です。したがって、汎用性の高い大域環境で提供されているもの以外は、非局所型名前に依存するべきではありません。言い換えれば、どこでも同じように宣言され、定義されている非局所型名前だけを使用するようにしてください。この例では、テンプレート関数の converter が、非局所型名前である intermediary と temporary を使用しています。これらの名前は use1.cc と use2.cc では異なる定義を持っているため、コンパイラが異なれば結果は違うものになるでしょう。テンプレートが正しく機能するためには、すべての非局所型名前 (intermediary と temporary) がどこでも同じ定義を持つ必要があります。

```
use_common.h
// Common template definition
template <class Source, class Target>
Target converter(Source source)
       {temporary = (intermediary)source;
       return (Target)temporary;}
use1.cc
typedef int intermediary;
int temporary;

#include "use_common.h"
use2.cc
typedef double intermediary;
unsigned int temporary;

#include "use_common.h"
```

非局所型名前を使用する典型的な例として、1 つのテンプレート内で cin と cout のストリームの使用があります。ほとんどのプログラマは実際、ストリームをテンプレートパラメータとして渡すことは望まないので、1 つの大域変数を参照するようにします。しかし、cin と cout はどこでも同じ定義を持っている必要があります。

## 6.7.2 テンプレート引数としての局所型

テンプレートインスタンス化の際には、型と名前が一致することを目安に、どのテンプレートがインスタンス化または再インスタンス化される必要があるか決定されます。したがって、局所型がテンプレート引数として使用された場合には重大な問題が発生する可能性があります。自分のコードに同様の問題が生じないように注意してください。

例 6–1 テンプレート引数としての局所型の問題の例

```
array.h
template <class Type> class Array {
        Type* data;
        int   size;
    public:
        Array(int sz);
        int GetSize();
};

array.cc
template <class Type> Array<Type>::Array(int sz)
    {size = sz; data = new Type[size];}
template <class Type> int Array<Type>::GetSize()
    {return size;}

file1.cc
#include "array.h"
struct Foo {int data;};
Array<Foo> File1Data(10);

file2.cc
#include "array.h"
struct Foo {double data;};
Array<Foo> File2Data(20);
```

`file1.cc` に登録された `Foo` 型は、`file2.cc` に登録された `Foo` 型と同じではありません。局所型をこのように使用すると、エラーと予期しない結果が発生することがあります。

## 6.7.3 テンプレート関数のフレンド宣言

テンプレートは、使用前に宣言されている必要があります。フレンド宣言では、テンプレートを宣言するのではなく、テンプレートの使用を宣言します。フレンド宣言の前に、実際のテンプレートが宣言されている必要があります。次の例では、作成済みオブジェクトファイルをリンクしようとするときに、`operator<<` 関数が未定義であるというエラーが生成されます。その結果、*operator*<< 関数はインスタンス化されません。

例 6–2 フレンド宣言の問題の例

```
array.h
// generates undefined error for the operator<< function
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
        operator<<(std::ostream&, const array<T>&);
};
#endif

array.cc
#include <stdlib.h>
#include <iostream>

template<class T> array<T>::array() {size = 1024;}

template<class T>
std::ostream&
operator<<(std::ostream& out, const array<T>& rhs)
    {return out <<'[' << rhs.size <<']';}

main.cc
#include <iostream>
#include "array.h"

int main()
{
    std::cout
      << "creating an array of int... " << std::flush;
    array<int> foo;
    std::cout << "done\n";
    std::cout << foo << std::endl;
    return 0;
}
```

コンパイラは、次の行を array クラスの friend である正規関数の宣言として読み取っているので、コンパイル中にエラーメッセージは発行されません。

```
friend ostream& operator<<(ostream&, const array<T>&);
```

operator<< は実際にはテンプレート関数であるため、template class array を宣言する前にこの関数にテンプレート宣言を行う必要があります。ただし、operator<< はパラメータ type array<T> を持つため、関数宣言の前に array<T> を宣言する必要があります。ファイル array.h は、次の例のようになります。

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>
```

```
// the next two lines declare operator<< as a template function
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<< <T> (std::ostream&, const array<T>&);
};
#endif
```

## 6.7.4 テンプレート定義内での修飾名の使用

C++ 標準は、テンプレート引数に依存する修飾名を持つ型を、typename キーワードを使用して型名として明示的に示すことを規定しています。この要件は、コンパイラがそれを型として推定できる場合にも適用されます。次の例の各コメントは、それぞれの修飾名が typename キーワードを必要とするかどうかを示しています。

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // not a type
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;   // not a type
template <class T> struct example {
  static typename T::a_type variable1;             // dependent
  static typename parametric<T>::a_type variable2; // dependent
  static simple::a_type variable3;                 // not dependent
};
template <class T> typename T::a_type             // dependent
  example<T>::variable1 = 0;                      // not a type
template <class T> typename parametric<T>::a_type // dependent
  example<T>::variable2 = 0;                      // not a type
template <class T> simple::a_type   // not dependent
example<T>::variable3 = 0;          // not a type
```

## 6.7.5 テンプレート名の入れ子

「>>」という文字シーケンスは右シフト演算子と解釈されるため、あるテンプレート名を別のテンプレート名で使用する場合は注意が必要です。隣接する「>」文字を少なくとも 1 つの空白文字で区切ってください。

次に誤った書式の例を示します。

```
Array<String<10>> short_string_array(100); // >> = right-shift
```

これは次のように解釈されます。

```
Array<String<10 >> short_string_array(100);
```

正しい構文は次のとおりです。

```
Array<String<10> > short_string_array(100);
```

## 6.7.6 静的変数や静的関数の参照

テンプレート定義の内部では、大域スコープや名前空間で静的として宣言されたオブジェクトや関数の参照がサポートされません。複数のインスタンスが生成されると、それぞれのインスタンスが別々のオブジェクトを参照するため、単一定義規則 (C++ 標準の第 3.2 節) に違反します。通常、このエラーはリンク時にシンボルの不足の形で通知されます。

すべてのテンプレートのインスタンス化で同じオブジェクトを共有する場合は、そのオブジェクトを該当する名前空間の非静的メンバーにします。また、あるテンプレートクラスをインスタンス化するたびに、別々のオブジェクトを使用する場合は、そのオブジェクトを該当するテンプレートクラスの静的メンバーにします。同様に、あるテンプレート関数をインスタンス化するたびに、別々のオブジェクトを使用する場合は、そのオブジェクトを該当するテンプレート関数の局所メンバーにします。

## 6.7.7 テンプレートを使用して複数のプログラムを同一ディレクトリに構築する

`-instances=extern` を指定して複数のプログラムまたはライブラリを構築する場合は、それらを別のディレクトリに構築します。同一ディレクトリ内に構築する場合は、構築ごとにリポジトリを消去します。これにより、予期しないエラーが回避されます。詳細は、104 ページの「7.4.4 テンプレートリポジトリの共有」を参照してください。

メイクファイル `a.cc`、`b.cc`、`x.h`、および `x.cc`. を使用した次の例を考えてみますこの例は、`-instances=extern` を指定した場合にのみ有効です。

```
........
Makefile
........
CCC = CC

all: a b
```

```
a:
    $(CCC) -I. -instances=extern -c a.cc
    $(CCC) -instances=extern -o a a.o

b:
    $(CCC) -I. -instances=extern -c b.cc
    $(CCC) -instances=extern -o b b.o

clean:
    /bin/rm -rf SunWS_cache *.o a b
```

```
...
x.h
...
template <class T> class X {
public:
  int open();
  int create();
  static int variable;
};
```

```
...
x.cc
...
template <class T> int X<T>::create() {
  return variable;
}

template <class T> int X<T>::open() {
  return variable;
}

template <class T> int X<T>::variable = 1;
```

```
...
a.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.open();
  temp1.create();
}
```

```
...
b.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.create();
}
```

a と b の両方を構築する場合は、それらの 2 つの構築の間に make clean コマンドを追加します。次のコマンドでは、エラーが発生します。

```
example% make a
example% make b
```

次のコマンドでは、エラーは発生しません。

```
example% make a
example% make clean
example% make b
```

◆ ◆ ◆ 第 7 章

# 7 テンプレートのコンパイル

テンプレートをコンパイルするためには、C++ コンパイラは従来の UNIX コンパイラよりも多くのことを行う必要があります。C++ コンパイラは、必要に応じてテンプレートインスタンスのオブジェクトコードを生成します。コンパイラは、テンプレートリポジトリを使って、別々のコンパイル間でテンプレートインスタンスを共有することができます。また、テンプレートコンパイルのいくつかのオプションを使用できます。コンパイラは、別々のソースファイルにあるテンプレート定義を見つけ、テンプレートインスタンスと main コード行の整合性を維持する必要があります。

## 7.1 冗長コンパイル

フラグ `-verbose=template` が指定されている場合は、テンプレートコンパイル作業中の重要なイベントがユーザーに通知されます。逆に、デフォルトの `-verbose=no%template` が指定されている場合は、コンパイラは通知しません。そのほかに、`+w` オプション を指定すると、テンプレートのインスタンス化が行われたときに問題になりそうな内容が通知される場合があります。

## 7.2 リポジトリの管理

`CCadmin`(1) コマンドは、テンプレートリポジトリを管理します (`-instances=extern` オプションを使用する場合のみ)。たとえば、プログラムの変更によって、インスタンス化が不要になり、記憶領域が無駄になることがあります。`CCadmin` の `-clean` コマンド (以前のリリースの `ptclean`) を使用すれば、すべてのインスタンス化と関連データを整理できます。インスタンス化は、必要なときだけ再作成されます。

## 7.2.1 生成されるインスタンス

コンパイラは、テンプレートインスタンス生成のため、インラインテンプレート関数をインライン関数として扱います。コンパイラは、インラインテンプレート関数をほかのインライン関数と同じように管理します。この章の内容は、テンプレートインライン関数には適用されません。

## 7.2.2 全クラスインスタンス化

コンパイラは通常、テンプレートクラスのメンバーをほかのメンバーからは独立してインスタンス化するので、プログラム内で使用されるメンバーだけがインスタンス化されます。デバッガによる使用を目的としたメソッドは、通常はインスタンス化されません。

デバッグ中のメンバーを、デバッガから確実に利用できるようにするには、2 つの方法を使用します。

- 第 1 に、実際には使用されないテンプレートクラスインスタンスメンバーを使用する、非テンプレート関数を作成します。この関数は呼び出されないようにする必要があります。
- 第 2 に、`-template=wholeclass` コンパイラオプションを使用します。このオプションを指定すると、非テンプレートで非インラインのメンバーのうちのどれかがインスタンス化された場合に、ほかの非テンプレート、非インラインのメンバーもすべてインスタンス化されます。

ISO C++ 標準では、特定のテンプレート引数により、すべてのメンバーが正当であるとはかぎらないテンプレートクラスを開発者が作成することを許可しています。不正メンバーをインスタンス化しないかぎり、プログラムは依然として適正です。ISO C++ 標準ライブラリでは、この技法が使用されています。ただし、`-template=wholeclass` オプションはすべてのメンバーをインスタンス化するので、問題のあるテンプレート引数を使ってインスタンス化する場合には、この種のテンプレートクラスに使用できません。

## 7.2.3 コンパイル時のインスタンス化

インスタンス化とは、C++ コンパイラがテンプレートから使用可能な関数やオブジェクトを作成するプロセスをいいます。C++ コンパイラ ではコンパイル時にインスタンス化を行います。つまり、テンプレートへの参照がコンパイルされているときに、インスタンス化が行われます。

コンパイル時のインスタンス化の長所を次に示します。

- デバッグが非常に簡単である。エラーメッセージがコンテキストの中に発生するので、コンパイラが参照位置を完全に追跡することができる。

- テンプレートのインスタンス化が常に最新である。
- リンク段階を含めて全コンパイル時間が短縮される。

ソースファイルが異なるディレクトリに存在する場合、またはテンプレートシンボルを指定してライブラリを使用した場合には、テンプレートが複数回にわたってインスタンス化されることがあります。

### 7.2.4 テンプレートインスタンスの配置とリンケージ

デフォルトでは、インスタンスは特別なアドレスセクションに移動し、リンカーは重複を認識、および破棄します。コンパイラには、インスタンスの配置とリンケージの方法として、外部、静的、大域、明示的、半明示的のどれを使うかを指定できます。

- 外部 インスタンスは、次の条件が成立する場合に最大のパフォーマンスを達成します。
  - プログラム内のインスタンスのセットは小さいが、各コンパイル単位がそれぞれ参照するインスタンスのサブセットが大きい。
  - 2、3 個以上のコンパイル単位で参照されるインスタンスがほとんどない。

  静的インスタンスは非推奨です。
- デフォルトである大域インスタンスは、あらゆる開発に適していますが、さまざまなインスタンスをオブジェクトが参照する場合に最適です。
- 明示的インスタンスは、厳密に管理されたアプリケーションコンパイル環境に適しています。
- 半明示的インスタンスは、前述より多少管理の程度が緩やかなアプリケーションコンパイル環境に適しています。ただし、このインスタンスは明示的インスタンスより大きなオブジェクトファイルを生成し、用途はかぎられています。

この節では、5 つのインスタンスの配置とリンケージの方法について説明します。インスタンスの生成に関する詳細は、85 ページの「6.3 テンプレートのインスタンス化」にあります。

## 7.3 外部インスタンス

外部インスタンスの場合では、すべてのインスタンスがテンプレートリポジトリ内に置かれます。テンプレートインスタンスは 1 つしか存在できません。つまり、インスタンスが未定義であるとか、重複して定義されているということはありません。テンプレートは必要な場合にのみ再インスタンス化されます。非デバッグコードの場合、すべてのオブジェクトファイル (テンプレートキャッシュに入っているものを含む) の総サイズは、`-instances=extern` を指定したときの値が `-instances=global` を指定したときの値より小さくなることがあります。

テンプレートインスタンスは、リポジトリ内では大域リンケージを受け取ります。インスタンスは、現在のコンパイル単位からは、外部リンケージで参照されます。

---

注 – コンパイルとリンクの手順を別々に実行し、コンパイル処理で -instance=extern を指定する場合は、リンク処理でも -instance=extern を指定する必要があります。

---

この方法には、別のプログラムに変更したり、プログラムを大幅に変更したりした場合にはキャッシュを常にクリアーする必要があるという欠点があります。キャッシュへのアクセスを一度に 1 回だけに限定しなければならないため、キャッシュは、dmake を使用する場合と同じように、並列コンパイルにおけるボトルネックとなります。また、1 つのディレクトリ内に構築できるプログラムは 1 個だけです。

有効なテンプレートインスタンスがキャッシュ内に存在するかどうかを判断することは、単にインスタンスを本体のオブジェクトファイル内に作成してあとで必要に応じてそれを破棄することよりも時間がかかる可能性があります。

外部リンケージは、-instances=extern オプションによって指定します。

インスタンスはテンプレートリポジトリ内に保存されているので、外部インスタンスを使用する C++ オブジェクトをプログラムにリンクするには CC コマンドを使用しなければなりません。

使用するすべてのテンプレートインスタンスを含むライブラリを作成する場合には、-xar オプションでコンパイルしてください。ar コマンドは使用できません。例:

```
example% CC -xar -instances=extern -o libmain.a a.o b.o c.o
```

## 7.3.1 キャッシュの衝突の可能性

-instance=extern を指定する場合、キャッシュの衝突の可能性があるため、異なるバージョンのコンパイラを同一ディレクトリ内で実行しないでください。-instances=extern テンプレートモデルでコンパイルする場合は、次の点に注意してください。

- 同一ディレクトリ内に、無関係のバイナリを作成しないでください。同一ディレクトリ内で作成されるすべてのバイナリ (.o、.a、.so、実行可能プログラム) は関連しているべきです。これは、複数のオブジェクトファイルに共通のすべてのオブジェクト、関数、型の名前は、定義が同一であるためです。
- dmake を使用する場合などは、複数のコンパイルを同一ディレクトリで同時に実行しても問題はありません。ほかのリンク段階と同時にコンパイルまたはリンク段階を実行すると、問題が発生する場合があります。リンク段階とは、ライブラリまたは実行可能プログラムを作成する処理です。メイクファイル内での依存により、1 つのリンク段階でのコマンドの並列実行は許可されていません。

## 7.3.2 静的インスタンス

注 – `--instances=global` が `static` の利点をすべて提供し、かつ欠点がないので、`-instances=static` オプションは非推奨です。このオプションは、今はもう存在していない問題を克服するために、以前のバージョンで提供されました。

静的インスタンスの場合は、すべてのインスタンスが現在のコンパイル単位内に置かれます。その結果、テンプレートは各再コンパイル作業中に再インスタンス化されます。インスタンスはテンプレートリポジトリに保存されません。

この方法の欠点は、言語の意味解釈が規定どおりでないこと、かなり大きいオブジェクトと実行可能ファイルが作られることです。

インスタンスは静的リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位以外では認識することも使用することもできません。そのため、テンプレートの同じインスタンス化がいくつかのオブジェクトファイルに存在することがあります。複数のインスタンスによって不必要に大きなプログラムが生成されるので、静的インスタンスのリンケージは、テンプレートが重複してインスタンス化される可能性が低い小さなプログラムだけに適しています。

静的インスタンスは潜在的にコンパイル速度が速いため、修正継続機能を使用したデバッグにも適しています。『`dbx` コマンドによるデバッグ』を参照してください。

注 – プログラムがコンパイル単位間で、テンプレートクラスまたはテンプレート機能の静的データメンバーなどのテンプレートインスタンスの共有に依存している場合は、静的インスタンス方式は使用しないでください。プログラムが正しく動作しなくなります。

静的インスタンスリンケージは、`-instances=static` コンパイルオプションで指定します。

## 7.3.3 大域インスタンス

旧リリースのコンパイラとは異なり、新リリースでは、大域インスタンスの複数のコピーを防ぐ必要はありません。

この方法の利点は、ほかのコンパイラで通常受け入れられる正しくないソースコードを、このモードで受け入れられるようになったという点です。特に、テンプレートインスタンスの中からの静的変数への参照は正当なものではありませんが、通常は受け入れられるものです。

この方法の欠点は、テンプレートインスタンスが複数のファイルにコピーされることから、個々のオブジェクトファイルが通常より大きくなる可能性がある点です。デバッグを目的としてオブジェクトファイルの一部を -g オプションを使ってコンパイルし、ほかのオブジェクトファイルを -g オプションなしでコンパイルした場合、プログラムにリンクされるテンプレートインスタンスが、デバッグバージョンと非デバッグバージョンのどちらであるかを予測することは難しくなります。

テンプレートインスタンスは大域リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位の外でも認識でき、使用できます。

大域インスタンスは、`-instances=global` オプションで指定します (デフォルト)。

## 7.3.4 明示的インスタンス

明示的インスタンスの場合、インスタンスは、明示的にインスタンス化されたテンプレートに対してのみ生成されます。暗黙的なインスタンス化は行われません。インスタンスは現在のコンパイル単位に置かれます。

この方法の利点はテンプレートのコンパイル量もオブジェクトのサイズも、ほかのどの方法より小さくて済むことです。

欠点は、すべてのインスタンス化を手動で行う必要がある点です。

テンプレートインスタンスは大域リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位の外でも認識でき、使用できます。リンカーは、重複しているものを見つけ、破棄します。

明示的リンケージは、`-instances=explicit` オプションによって指定します。

## 7.3.5 半明示的インスタンス

半明示的インスタンスの場合、インスタンスは、明示的にインスタンス化されるテンプレートやテンプレート本体の中で暗黙的にインスタンス化されるテンプレートに対してのみ生成されます。明示的に作成されるインスタンスが必要とするインスタンスは自動的に生成されます。main コード行内で行う暗黙的なインスタンス化は不完全になります。インスタンスは現在のコンパイル単位に置かれます。したがって、テンプレートは再コンパイルごとに再インスタンス化されます。インスタンスが大域リンケージを受けることはなく、テンプレートリポジトリには保存されません。

半明示的インスタンスは、`-instances=semiexplicit` オプションで指定します。

# 7.4 テンプレートリポジトリ

必要なときだけテンプレートインスタンスがコンパイルされるよう、個別のコンパイル間のテンプレートインスタンスがテンプレートリポジトリに保存されます。テンプレートリポジトリには、外部インスタンスメソッドを使用するときにテンプレートのインスタンス化に必要となる非ソースファイルがすべて入っています。このリポジトリがほかの種類のインスタンスに使用されることはありません。

## 7.4.1 リポジトリの構造

テンプレートリポジトリは、デフォルトで、キャッシュディレクトリ (SunWS_cache) にあります。

キャッシュディレクトリは、オブジェクトファイルが置かれるのと同じディレクトリ内にあります。SUNWS_CACHE_NAME 環境変数を設定すれば、キャッシュディレクトリ名を変更できます。SUNWS_CACHE_NAME 変数の値は必ずディレクトリ名にし、パス名にしてはならない点に注意してください。コンパイラは、テンプレートキャッシュディレクトリをオブジェクトファイルディレクトリの下に自動的に置くため、コンパイラはすでにパスを持っています。

## 7.4.2 テンプレートリポジトリへの書き込み

コンパイラは、テンプレートインスタンスを格納しなければならないとき、出力ファイルに対応するテンプレートリポジトリにそれらを保存します。たとえば、次のコマンドでは、コンパイラはオブジェクトファイルを ./sub/a.o に、テンプレートインスタンスを ./sub/SunWS_cache 内のリポジトリにそれぞれ書き込みます。キャッシュディレクトリが存在せず、コンパイラがテンプレートをインスタンス化する必要がある場合、コンパイラはこのディレクトリを作成します。

```
example% CC -o sub/a.o a.cc
```

## 7.4.3 複数のテンプレートリポジトリからの読み取り

コンパイラは、読み込むオブジェクトファイルに対応するテンプレートリポジトリからテンプレートインスタンスを読み取ります。たとえば、次のコマンドは ./sub1/SunWS_cache と ./sub2/SunWS_cache を読み取り、必要な場合は ./SunWS_cache に書き込みます。

```
example% CC sub1/a.o sub2/b.o
```

### 7.4.4 テンプレートリポジトリの共有

リポジトリ内にあるテンプレートは、ISO/ANSI C++ 標準の単一定義規則に違反してはいけません。つまり、テンプレートは、どの用途に使用される場合でも、1 つのソースから派生したものでなければなりません。この規則に違反した場合の動作は定義されていません。

この規則に違反しないようにするための、もっとも保守的で、もっとも簡単な方法は、1 つのディレクトリ内では 1 つのプログラムまたはライブラリしか作成しないことです。無関係な 2 つのプログラムが同じ型名または外部名を使用して別のものを意味する場合があります。これらのプログラムがテンプレートリポジトリを共有すると、テンプレートの定義が競合し、予期せぬ結果が生じる可能性があります。

### 7.4.5 -instances=extern によるテンプレートインスタンスの自動一貫性

`-instances=extern` を指定すると、テンプレートリポジトリマネージャーは、リポジトリ中のインスタンスの状態をソースファイルと確実に一致させて最新の状態にします。

たとえば、ソースファイルが `-g` オプション (デバッグ付き) でコンパイルされる場合は、データベースの中の必要なファイルも `-g` でコンパイルされます。

さらに、テンプレートリポジトリはコンパイル時の変更を追跡します。たとえば、`-DDEBUG` フラグを指定して名前 `DEBUG` を定義すると、データベースがこれを追跡します。その次のコンパイルでこのフラグを省くと、コンパイラはこの依存性が設定されているテンプレートを再度インスタンス化します。

---

注 – テンプレートのソースコードを削除する場合や、テンプレートの使用を停止する場合も、テンプレートのインスタンスはキャッシュ内にとどまります。関数テンプレートの署名を変更する場合も、古い署名を使用しているインスタンスはキャッシュ内にとどまります。これらの点が原因でコンパイル時またはリンク時に予期しない動作が発生した場合は、テンプレートキャッシュをクリアし、プログラムを再構築してください。

---

# 7.5 テンプレート定義の検索

定義分離型テンプレートの編成、つまりテンプレートを使用するファイルの中にテンプレートの宣言だけがあって定義はないという編成を使用している場合には、現在のコンパイル単位にテンプレート定義が存在しないので、コンパイラが定義を検索しなければなりません。この節では、そうした検索について説明します。

定義の検索はかなり複雑で、エラーを発生しやすい傾向があります。このため、可能であれば、定義取り込み型のテンプレートファイルの編成を使用したほうがよいでしょう。こうすれば、定義検索をまったく行わなくて済みます。77 ページの「5.2.1 テンプレート定義の取り込み」を参照してください。

---

注 `--template=no%extdef` オプションを使用する場合、コンパイラは分離されたソースファイルを検索しません。

---

## 7.5.1 ソースファイルの位置規約

オプションファイルで提供されるような特定の指令がない場合には、コンパイラは `Cfront` 形式の方法でテンプレート定義ファイルを検出します。この方法の場合、テンプレート宣言ファイルと同じベース名がテンプレート定義ファイルに含まれている必要があります。またこの方法では、テンプレート定義ファイルが現在の `include` パス上にある必要もあります。たとえば、テンプレート関数 `foo()` が `foo.h` 内にある場合には、それと一致するテンプレート定義ファイルの名前を `foo.cc` か、またはほかの認識可能なソースファイル拡張子 (`.C`、`.c`、`.cc`、`.cpp`、`.cxx`、または `.c++`) にしなければなりません。テンプレート定義ファイルは、通常使用する `include` ディレクトリの 1 つか、またはそれと一致するヘッダーファイルと同じディレクトリの中に置かなければなりません。

## 7.5.2 定義検索パス

`-I` で設定する通常の検索パスの代わりに、`-pti`*directory* オプションでテンプレート定義ファイルの検索ディレクトリを指定することができます。複数の `-pti` フラグは、複数の検索ディレクトリ、つまり 1 つの検索パスを定義します。`-pti`*directory* を使用している場合には、コンパイラはこのパス上のテンプレート定義ファイルを探し、`-I` フラグを無視します。`-pti`*directory* フラグはソースファイルの検索規則を複雑にするので、`-pti`*directory* オプションの代わりに `-I` オプションを使用してください。

## 7.5.3 問題がある検索の回避

コンパイラが、コンパイル対象ではないファイルを検索することが原因で、紛らわしい警告またはエラーメッセージが生成されることがあります。通常、問題は、たとえば `foo.h` というファイルにテンプレート宣言が含まれていて、`foo.cc` などの別のファイルが暗黙的に取り込まれることにあります。

ヘッダーファイル `foo.h` の中にテンプレート宣言が存在する場合は、コンパイラはデフォルトで、`foo` という名前および C++ のファイル拡張子 (`.C`、`.c`、`.cc`、`.cpp`、`.cxx`、または `.c++`) を持つファイルをデフォルトで検索します。そうしたファイルを見つけた場合、コンパイラはそのファイルを自動的に取り込みます。こうした検索の詳細は、105 ページの「7.5 テンプレート定義の検索」を参照してください。

このように扱われるべきでないファイル `foo.cc` が存在する場合、選択肢は 2 つあります。

- `.h` または `.cc` の名前を変更して、名前が一致しないようにする。
- `-template=no%extdef` オプションを指定することによって、テンプレート定義ファイルの自動検索を無効にする。この場合は、すべてのテンプレート定義をコードに明示的に取り込む必要があります。このため、「定義分離」モデルは使用できなくなります。

◆ ◆ ◆ 第 8 章

# 8 例外処理

この章では、C++ コンパイラの例外処理の実装について説明します。118 ページの「10.2 マルチスレッドプログラムでの例外の使用」にも補足情報を掲載しています。例外処理の詳細については、『プログラミング言語 C++』(第 3 版、Bjarne Stroustrup 著、アスキー、1997 年) を参照してください。

## 8.1 同期例外と非同期例外

例外処理では、配列範囲のチェックといった同期例外だけをサポートすることが意図されています。同期例外とは、例外を `throw` 文からだけ生成できることを意味します。

C++ 標準でサポートされる同期例外処理は、終了モデルに基づいています。終了とは、いったん例外が送出されると、例外の送出元に制御が二度と戻らないことを意味します。

例外処理では、キーボード割り込みなどの非同期例外の直接処理は意図されていません。ただし、注意して使用すれば、非同期イベントが発生したときに、例外処理を行わせることができます。たとえば、シグナルに対する例外処理を行うには、大域変数を設定するシグナルハンドラと、この変数の値を定期的にチェックし、値が変化したときに例外を送出するルーチンを作成します。シグナルハンドラから例外をスローすることはできません。

## 8.2 実行時エラーの指定

例外に関する実行時エラーメッセージには、次の 5 種類があります。

- 例外のハンドラがありません
- 予期しない例外を送出
- ハンドラでは例外の再送出しかできません

- スタックの巻き戻し中は、デストラクタは独自の例外を処理しなければなりません
- メモリー不足

実行時にエラーが検出されると、現在の例外の種類と、前述の 5 つのメッセージのいずれかがエラーメッセージとして表示されます。デフォルト設定では、事前定義済みの `terminate()` 関数が呼び出され、さらにこの関数から `abort()` が呼び出されます。

コンパイラは、例外指定に含まれている情報に基づいて、コードの生成を最適化します。たとえば、例外を送出しない関数のテーブルエントリは抑止されます。また、関数の例外指定の実行時チェックは、できるかぎり省略されます。

## 8.3 例外の無効化

プログラムで例外を使用しないことが明らかであれば、`features=no%except` コンパイラオプションを使用して、例外処理用のコードの生成を抑止することができます。このオプションを使用すると、コードサイズが若干小さくなり、実行速度が多少高速になります。ただし、例外を無効にしてコンパイルしたファイルを、例外を使用するファイルにリンクすると、例外を無効にしてコンパイルしたファイルに含まれている局所オブジェクトが、例外が発生したときに破棄されずに残ってしまう可能性があります。デフォルト設定では、コンパイラは例外処理用のコードを生成します。時間と容量のオーバーヘッドが重要な場合を除いて、通常は例外を有効のままにしておいてください。

---

注 – C++ 標準ライブラリ、`dynamic_cast`、デフォルトの `new` 演算子では例外が必要です。そのため、標準モード (デフォルトモード) でコンパイルを行う場合は、例外を無効にしないでください。

---

## 8.4 実行時関数と事前定義済み例外の使用

標準ヘッダー `<exception>` には、C++ 標準で指定されるクラスと例外関連関数が含まれています。このヘッダーは、標準モード (コンパイラのデフォルトモード、すなわち `-compat=5` オプションを使用するモード) でコンパイルを行うときだけ使用されます。次は、`<exception>` ヘッダーファイル宣言を抜粋したものです。

```
// standard header <exception>
namespace std {
    class exception {
           exception() throw();
           exception(const exception&) throw();
           exception& operator=(const exception&) throw();
           virtual ~exception() throw();
```

```
            virtual const char* what() const throw();
    };
    class bad_exception: public exception {...};
    // Unexpected exception handling
       typedef void (*unexpected_handler)();
       unexpected_handler
         set_unexpected(unexpected_handler) throw();
       void unexpected();
    // Termination handling
       typedef void (*terminate_handler)();
       terminate_handler set_terminate(terminate_handler) throw();
       void terminate();
       bool uncaught_exception() throw();
}
```

標準クラス exception は、構文要素や C++ 標準ライブラリから送出されるすべての例外のための基底クラスです。exception 型のオブジェクトは、例外を発生させることなく作成、複製、破棄することができます。仮想メンバー関数 what() は、例外についての情報を示す文字列を返します。

C++ release 4.2 で使用される例外との互換性について、ヘッダー <exception.h> は標準モードでの使用のため提供されます。このヘッダーは、C++ 標準のコードに移行するためのもので、C++ 標準には含まれていない宣言を含んでいます。開発スケジュールに余裕があれば、<exception.h> の代わりに <exception> を使用し、コードを C++ 標準に従って更新してください。

```
// header <exception.h>, used for transition
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

## 8.5 シグナルや Setjmp/Longjmp と例外との併用

同じプログラムの中で、setjmp/longjmp 関数と例外処理を併用できます。ただし、これらが相互に干渉しないことが条件になります。

その場合、例外と setjmp/longjmp のすべての使用規則が、それぞれ別々に適用されます。また、A 地点から B 地点への longjmp を使用できるのは、例外を A 地点から送出し、B 地点で捕獲した場合と効果が同じになる場合だけです。特に、try ブロック (または catch ブロック) への、または try ブロック (または catch ブロック) からの、直接的または間接的な longjmp や、自動変数や一時変数の初期化や明示的な破棄の前後にまたがる longjmp は行なってはいけません。

シグナルハンドラからは例外を送出できません。

## 8.6 例外のある共有ライブラリの構築

C++ コードが含まれているプログラムでは、`-Bsymbolic` を使用しないでください。リンカースコープオプションの代わりにリンカーのマップファイルを使用してください。63 ページの「4.1 リンカースコープ」を参照してください。`-Bsymbolic` を使用すると、異なるモジュール内の参照が、本来 1 つの大域オブジェクトの複数の異なる複製に結合されてしまう可能性があります。

例外メカニズムは、アドレスの比較によって機能します。オブジェクトの複製が 2 つある場合は、アドレスが同一であると評価されず、本来一意のアドレスを比較することで機能する例外メカニズムで問題が発生することがあります。

◆ ◆ ◆ 第 9 章

# 9 プログラムパフォーマンスの改善

C++ 関数のパフォーマンスを高めるには、コンパイラが C++ 関数を最適化しやすいように関数を記述することが必要です。多くの書籍に、ソフトウェアパフォーマンス全般について、および C++ について詳しく記載されています。この章では、そのような有益な情報を繰り返すのではなく、C++ コンパイラに強く影響を及ぼすパフォーマンス方針についてのみ説明します。

## 9.1 一時オブジェクトの回避

C++ 関数は、暗黙的に一時オブジェクトを多数生成することがよくあります。これらのオブジェクトは、生成後破棄する必要があります。しかし、そのようなクラスが多数ある場合は、この一時的なオブジェクトの作成と破棄が、処理時間とメモリー使用率という点でかなりの負担になります。C++ コンパイラは一時オブジェクトの一部を削除しますが、すべてを削除できるとはかぎりません。

プログラムの明瞭さを保ちつつ、一時オブジェクトの数が最少になるように関数を記述してください。このための手法としては、暗黙の一時オブジェクトに代わって明示的な変数を使用すること、値パラメータに代わって参照パラメータを使用することなどがあります。また、+ と = だけを実装して使用するのではなく、+= のような演算を実装および使用することもよい手法です。たとえば、次の例の最初の行は、a + b の結果に一時オブジェクトを使用していますが、2 行目は一時オブジェクトを使用していません。

```
T x = a + b;
T x(a); x += b;
```

# 9.2 インライン関数の使用

小さくて実行速度の速い関数を呼び出す場合は、通常どおりに呼び出すよりもインライン展開する方が効率が上がります。逆に言えば、大きいか実行速度の遅い関数を呼び出す場合は、分岐するよりもインライン展開する方が効率が悪くなります。また、インライン関数の呼び出しはすべて、関数定義が変更されるたびに再コンパイルする必要があります。このため、インライン関数を使用するかどうかは十分な検討が必要です。

関数定義を変更する可能性があり、呼び出し元をすべて再コンパイルするのに負荷が大きいと予測される場合は、インライン関数は使用しないでください。それ以外の場合は、関数をインライン展開するコードが関数を呼び出すコードよりも小さいか、あるいはアプリケーションの動作がインライン関数によって大幅に高速化される場合にのみインライン関数を使用してください。

コンパイラは、すべての関数呼び出しをインライン展開できるわけではありません。そのため、関数のインライン展開の効率を最高にするにはソースを変更しなければならない場合があります。どのような場合に関数がインライン展開されないかを知るには、+w オプションを使用してください。次のような状況では、コンパイラは関数をインライン展開しません。

- ループ、switch 文、try および catch 文のような難しい制御構造が関数に含まれる場合。実際には、これらの関数では、その難しい制御構造はごくまれにしか実行されません。このような関数をインライン展開するには、難しい制御構造が入った内側部分と、内側部分を呼び出す場合を決定する外側部分の 2 つに関数を分割します。コンパイラが関数全体をインライン展開できる場合でも、このようによく使用する部分とめったに使用しない部分を分けることで、パフォーマンスを高めることができます。
- インライン関数本体のサイズが大きいか、あるいは複雑な場合。見たところ単純な関数本体は、本体内でほかのインライン関数を呼び出していたり、あるいはコンストラクタやデストラクタを暗黙に呼び出していたりするために複雑な場合があります (派生クラスのコンストラクタとデストラクタでこのような状況がよく起きる)。このような関数ではインライン展開でパフォーマンスが大幅に向上することはめったにないため、インライン展開しないことをお勧めします。
- インライン関数呼び出しの引数が大きいか、あるいは複雑な場合。インラインメンバー関数を呼び出すためのオブジェクトが、そのインライン関数呼び出しの結果である場合は、パフォーマンスが大幅に下がります。複雑な引数を持つ関数をインライン展開するには、その関数引数を局所変数を使用して関数に渡してください。

## 9.3 デフォルト演算子の使用

クラス定義がパラメータのないコンストラクタ、コピーコンストラクタ、コピー代入演算子、またはデストラクタを宣言しない場合、コンパイラがそれらを暗黙的に宣言します。こうして宣言されたものはデフォルト演算子と呼ばれます。C のような構造体は、デフォルト演算子を持っています。デフォルト演算子は、優れたコードを生成するためにどのような作業が必要かを把握しています。この結果作成されるコードは、ユーザーが作成したコードよりもはるかに高速です。これは、プログラマが通常使用できないアセンブリレベルの機能をコンパイラが利用できるためです。そのため、デフォルト演算子が必要な作業をこなしてくれる場合は、プログラムでこれらの演算子をユーザー定義によって宣言する必要はありません。

デフォルト演算子はインライン関数であるため、インライン関数が適切でない場合にはデフォルト演算子を使用しないでください (前の節を参照)。それ以外の場合、次の状況ではデフォルト演算子が適しています。

- ユーザーが記述するパラメータのないコンストラクタが、その基底オブジェクトとメンバー変数に対してパラメータのないコンストラクタだけを呼び出す場合。基本の型は、「何も行わない」パラメータのないコンストラクタを効率よく受け入れます。
- ユーザーが記述するコピーコンストラクタが、すべての基底オブジェクトとメンバー変数をコピーする場合
- ユーザーが記述するコピー代入演算子が、すべての基底オブジェクトとメンバー変数をコピーする場合
- ユーザーが記述するデストラクタが空の場合

C++ のプログラミングを紹介する書籍の中には、コードを読んだ際にコードの作成者がデフォルト演算子の効果を考慮に入れていることがわかるように、常にすべての演算子を定義することを勧めているものもあります。しかし、そうすることは明らかに前述した最適化と相入れないものです。デフォルト演算子の使用について明示するには、クラスがデフォルト演算子を使用していることを説明したコメントをコードに入れることをお勧めします。

## 9.4 値クラスの使用

構造体や共用体などの C++ クラスは、値によって渡され、値によって返されます。POD (Plain-Old-Data) クラスの場合、C++ コンパイラは構造体を C コンパイラと同様に渡す必要があります。これらのクラスのオブジェクトは、直接的に渡されます。ユーザー定義のコピーコンストラクタを持つクラスのオブジェクトの場合、コンパイラは実際に、そのオブジェクトのコピーを構築し、コピーにポインタを渡し、返されたあとでコピーを破壊することが必要です。これらのクラスのオブジェクトは、間接的に渡されます。この 2 つの条件の中間に位置するクラスの場合

は、コンパイラによってどちらの扱いにするかが選択されます。しかし、そうすることでバイナリ互換性に影響が発生するため、コンパイラは各クラスに矛盾が出ないように選択する必要があります。

ほとんどのコンパイラでは、オブジェクトを直接渡すと実行速度が上がります。特に、複素数や確率値のような小さな値クラスの場合に、実行速度が大幅に上がります。そのためプログラムの効率は、間接的ではなく直接渡される可能性が高いクラスを設計することによって向上する場合があります。

クラスに次の要素が含まれる場合、クラスは間接的に渡されます。

- ユーザー定義のコピーコンストラクタ
- ユーザー定義のデストラクタ
- 間接的に渡される基底クラス
- 間接的に渡される非静的データメンバー

これらの要素が含まれない場合は、クラスは直接渡されます。

## 9.4.1 クラスを直接渡す

クラスが直接渡される可能性を最大にするには、次のようにしてください。

- 可能なかぎりデフォルトのコンストラクタ (特にデフォルトのコピーコンストラクタ) を使用する。
- 可能なかぎりデフォルトのデストラクタを使用する。デフォルトデストラクタは仮想ではないため、デフォルトデストラクタを使用したクラスは、通常は基底クラスにするべきではありません。
- 仮想関数と仮想基底クラスを使用しない。

## 9.4.2 各種のプロセッサでクラスを直接渡す

C++ コンパイラによって直接渡されるクラスおよび共用体は、C コンパイラが構造体または共用体を渡す場合とまったく同じように渡されます。しかし、C++ の構造体と共用体の渡し方は、アーキテクチャーによって異なります。

表 9–1 アーキテクチャー別の構造体と共用体の渡し方

| アーキテクチャー | 説明 |
|---|---|
| SPARC V7 および V8 | 構造体と共用体は、呼び出し元で記憶領域を割り当て、その記憶領域へのポインタを渡すことによって渡されます。つまり、構造体と共用体はすべて参照により渡されます。 |

表 9–1 アーキテクチャー別の構造体と共用体の渡し方 (続き)

| アーキテクチャー | 説明 |
|---|---|
| SPARC V9 | 16 バイト (32 バイト) 以下の構造体は、レジスタ中で渡され (返され) ます。共用体およびそのほかのすべての構造体は、呼び出し元で記憶領域を割り当て、その記憶領域へのポインタを渡すことによって渡されます。つまり、小さな構造体はレジスタで渡され、共用体と大きな構造体は参照により渡されます。この結果、小さな値のクラスは基本の型と同じ効率で渡されることになります。 |
| x86 プラットフォーム | 構造体と共用体を渡すには、スタックで領域を割り当て、引数をそのスタックにコピーします。構造体と共用体を返すには、呼び出し元のフレームに一時オブジェクトを割り当て、一時オブジェクトのアドレスを暗黙の最初のパラメータとして渡します。 |

# 9.5 メンバー変数のキャッシュ

C++ メンバー関数では、メンバー変数へのアクセスが頻繁に行われます。

そのため、コンパイラは、`this` ポインタを介してメモリーからメンバー変数を読み込まなければならないことがよくあります。値はポインタを介して読み込まれているため、次の読み込みをいつ行うべきか、あるいは先に読み込まれている値がまだ有効であるかどうかをコンパイラが決定できないことがあります。このような場合、コンパイラは安全な (しかし遅い) 手法を選択し、アクセスのたびにメンバー変数を再読み込みする必要があります。

不要なメモリー再読み込みが行われないようにするには、次のようにメンバー変数の値を局所変数に明示的にキャッシュしてください。

- 局所変数を宣言し、メンバー変数の値を使用して初期化する
- 関数全体で、メンバー変数の代わりに局所変数を使用する
- 局所変数が変わる場合は、局所変数の最終値をメンバー変数に代入する。しかし、メンバー関数がそのオブジェクトの別のメンバー関数を呼び出す場合には、この最適化のために意図しない結果が発生する場合があります。

この最適化は、基本の型の場合と同様に、値をレジスタに置くことができる場合にもっとも効果的です。また、別名の使用が減ることによりコンパイラの最適化が行われやすくなるため、記憶領域を使用する値にも効果があります。

この最適化は、メンバー変数が明示的に、あるいは暗黙的に頻繁に参照渡しされる場合には逆効果になる場合があります。

現在のオブジェクトとメンバー関数の引数の 1 つの間に別名が存在する可能性がある場合などには、クラスの意味を望ましいものにするために、メンバー変数を明示的にキャッシュしなければならないことがあります。例:

```
complex& operator*= (complex& left, complex& right)
{
  left.real = left.real * right.real + left.imag * right.imag;
  left.imag = left.real * right.imag + left.image * right.real;
}
```

前述のコードが次の指令で呼び出されると、意図しない結果になります。

```
x*=x;
```

◆ ◆ ◆ 第 10 章

# 10 マルチスレッドプログラムの構築

この章では、マルチスレッドプログラムの構築方法を説明します。さらに、例外の使用、C++ 標準ライブラリのオブジェクトをスレッド間で共有する方法、従来の (旧形式の) iostream をマルチスレッド環境で使用する方法についても取り上げます。

マルチスレッド化の詳細は、『Multithreaded Programming Guide』を参照してください。

OpenMP 共有メモリー並列化指令を使用してマルチスレッドプログラムを作成する方法の詳細は、『OpenMP API ユーザーガイド』も参照してください。

## 10.1 マルチスレッドプログラムの構築

C++ コンパイラに付属しているライブラリは、すべてマルチスレッドで使用しても安全です。マルチスレッドアプリケーションを作成したい場合や、アプリケーションをマルチスレッド化されたライブラリにリンクしたい場合は、`-mt` オプションを付けてプログラムのコンパイルとリンクを行う必要があります。このオプションを付けると、`–D_REENTRANT` がプリプロセッサに渡され、`–lthread` が `ld` に正しい順番で渡されます。デフォルトでは、`-mt` オプションは `libthread` が `ibCrun` の前にリンクされるようにします。マクロとライブラリを指定する代わりに、簡単でエラーの発生しにくい方法として `–mt` を使用することをお勧めします。

### 10.1.1 マルチスレッドコンパイルの確認

`ldd` コマンドを使用すると、アプリケーションが `libthread` にリンクされたかどうかを確認できます。

```
example% CC -mt myprog.cc
example% ldd a.out
libm.so.1 =>      /usr/lib/libm.so.1
libCrun.so.1 =>   /usr/lib/libCrun.so.1
```

```
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

### 10.1.2 C++ サポートライブラリの使用

C++ サポートライブラリ `libCrun`、 `ibiostream`、`libCstd` は、マルチスレッドに対しては安全ですが、`async` に対しては安全ではありません。したがって、マルチスレッドアプリケーションでは、サポートライブラリで使用できる関数をシグナルハンドラで使用しないでください。使用するとデッドロックが発生する可能性があります。

マルチスレッドアプリケーションのシグナルハンドラでは、次の機能を使用することは安全ではありません。

- iostream
- `new` 式と `delete` 式
- 例外

## 10.2 マルチスレッドプログラムでの例外の使用

現在の例外処理の実装は、マルチスレッドで使用しても安全です。これは、あるスレッドの例外によって、別のスレッドの例外が阻害されることがないためです。ただし、例外を使用して、スレッド間で情報を受け渡すことはできません。これは、あるスレッドからスローされた例外を、別のスレッドで捕獲できないためです。

それぞれのスレッドでは、独自の `terminate()` または `unexpected()` 関数を設定できます。あるスレッドで呼び出した `set_terminate()` 関数や `set_unexpected()` 関数は、そのスレッドの例外だけに影響します。デフォルトの `terminate()` 関数の内容は、すべてのスレッドで `abort()` になります。107 ページの「8.2 実行時エラーの指定」を参照してください。

### 10.2.1 スレッドの取り消し

`-noex` または `-features=no%except`、コンパイラオプションが指定されている場合を除き、`pthread_cancel`(3T) の呼び出しによってスレッドを取り消すと、スタック上の自動オブジェクト (静的ではない局所オブジェクト) が破棄されます。

`pthread_cancel`(3T) では、例外と同じ仕組みが使用されます。スレッドが取り消されると、局所デストラクタの実行中に、ユーザーが `pthread_cleanup_push()` を使用して登録したクリーンアップルーチンが実行されます。クリーンアップルーチンの登録後に呼び出した関数の局所オブジェクトは、そのクリーンアップルーチンが実行される前に破棄されます。

# 10.3 C++ 標準ライブラリのオブジェクトのスレッド間での共有

C++ 標準ライブラリ (libCstd -library=Cstd ) は一部のロケールを除き、マルチスレッドに対して安全です。これは、ライブラリの内部がマルチスレッド環境で適切に動作することを保証します。ただし、スレッド間で共有するライブラリオブジェクトには、ロックを配置する必要があります。setlocale(3C) および attributes(5) のマニュアルページを参照してください。

たとえば、文字列をインスタンス化し、この文字列を新しく生成したスレッドに参照で渡した場合は、この文字列への書き込みアクセスにロックを追加する必要があります。これは、同じ文字列オブジェクトを、プログラムが複数のスレッドで明示的に共有しているからです。この処理を行うために用意されたライブラリの機能については後述します。

これに対して、この文字列を新しいスレッドに値で渡した場合は、ロックについて考慮する必要はありません。このことは、Rogue Wave の「書き込み時コピー」機能により、2 つのスレッドの別々の文字列が同じ表現を共有している場合にも当てはまります。このような場合のロックは、ライブラリが自動的に処理します。プログラム自身でロックを行う必要があるのは、スレッド間での参照渡しや、大域オブジェクトや静的オブジェクトを使用して、同じオブジェクトを複数のスレッドから明示的に使用できるようにした場合だけです。

複数のスレッドが存在する場合の動作を保証するために、C++ 標準ライブラリの内部で使用されるロック (同期) メカニズムは、次のように説明することができます。

マルチスレッドでの安全性を実現する機能は、2 つの同期クラス、_RWSTDMutex と _RWSTDGuard によって提供されます。

_RWSTDMutex クラスは、プラットフォームに依存しないロック機能を提供します。このクラスには、次のメンバー関数があります。

- void acquire()– 自分自身に対するロックを獲得する。または、このロックを獲得できるまでブロックする。
- void release()– 自分自身に対するロックを解除する。

```
class _RWSTDMutex
{
public:
    _RWSTDMutex ();
    ~_RWSTDMutex ();
    void acquire ();
    void release ();
};
```

_RWSTDGuard クラスは、_RWSTDMutex クラスのオブジェクトをカプセル化するための便利なラッパークラスです。_RWSTDGuard クラスのオブジェクトは、自分自身のコンストラクタの中で、カプセル化された相互排他ロック (mutex) を獲得しようとしま

す。エラーが発生した場合は、このコンストラクタは std::exception から派生している ::thread_error 型の例外を送出します。獲得された相互排他ロックは、このオブジェクトのデストラクタの中で解除されます。このデストラクタは例外を送出しません。

```
class _RWSTDGuard
{
public:
    _RWSTDGuard (_RWSTDMutex&);
    ~_RWSTDGuard ();
};
```

さらに、_RWSTD_MT_GUARD(mutex) マクロ (従来の _STDGUARD) を使用すると、マルチスレッドの構築時にだけ _RWSTDGuard クラスのオブジェクトを生成できます。生成されたオブジェクトは、そのオブジェクトが定義されたコードブロックの残りの部分が、複数のスレッドで同時に実行されないようにします。単一スレッドの構築時には、マクロが空白の式に展開されます。

これらの機能は、次のように使用します。

```
#include <rw/stdmutex.h>

//
// An integer shared among multiple threads.
//
int I;

//
// A mutex used to synchronize updates to I.
//
_RWSTDMutex I_mutex;

//
// Increment I by one. Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire(); // Lock the mutex.
   I++;
   I_mutex.release(); // Unlock the mutex.
}

//
// Decrement I by one. Uses an _RWSTDGuard.
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex); // Acquire the lock on I_mutex.
   --I;
   //
   // The lock on I is released when destructor is called on guard.
   //
}
```

# 10.4 メモリーバリアー組み込み関数

コンパイラには、SPARC プロセッサと x86 プロセッサ用のさまざまなメモリーバリアー組み込み関数を定義するヘッダーファイル mbarrier.h が用意されています。これらの組み込み関数は、開発者が独自の同期プリミティブを使用してマルチスレッドコードを記述するために使用できます。これらの組み込み関数がいつ必要になるか、また、特定の状況で必要かどうかを判断するには、該当するプロセッサのドキュメントを参照してください。

mbarrier.h によりサポートされるメモリーオーダリング組み込み関数には、次のものが含まれます。

- __machine_r_barrier() - これは、*read* バリアーです。これにより、バリアー前のすべてのロード操作が、バリアー後のすべてのロード操作の前に完了します。
- __machine_w_barrier() - これは、*write* バリアーです。これにより、バリアー前のすべての格納操作が、バリアー後のすべての格納操作の前に完了します。
- __machine_rw_barrier() - これは、*read—write* バリアーです。これにより、バリアー前のすべてのロードおよび格納操作が、バリアー後のすべてのロードおよび格納操作の前に完了します。
- __machine_acq_barrier() - これは、*acquire* セマンティクスを持つバリアーです。これにより、バリアー前のすべてのロード操作が、バリアー後のすべてのロードおよび格納操作の前に完了します。
- __machine_rel_barrier() - これは、*release* セマンティクスを持つバリアーです。これにより、バリアー前のすべてのロードおよび格納操作が、バリアー後のすべての格納操作の前に完了します。
- __compiler_barrier() - コンパイラが、バリアーを越えてメモリーアクセスを移動するのを防ぎます。

__compiler_barrier() 組み込み関数を除くすべてのバリアー組み込み関数は、x86 でメモリーオーダリング命令を生成し、これらは mfence、sfence、または lfence 命令です。SPARC プラットフォームでは、これらは membar 命令です。

__compiler_barrier() 組み込み関数は、命令を生成せず、代わりに今後メモリー操作を開始する前にそれまでのメモリー操作をすべて完了する必要があることをコンパイラに通知します。この実際の結果として、ローカルでないすべての変数、および static 記憶クラス指定子を持つローカル変数が、バリアー前のメモリーに再度格納されてバリアー後に再ロードされ、コンパイラではバリアー前のメモリー操作とバリアー後のメモリー操作が混在することはありません。ほかのすべてのバリアーには、__compiler_barrier() 組み込み関数の動作が暗黙的に含まれています。

たとえば、次のコードでは、__compiler_barrier() 組み込み関数が存在しているためコンパイラによる 2 つのループのマージが止まります。

```
#include "mbarrier.h"
int thread_start[16];
```

```
void start_work()
{
/* Start all threads */
   for (int i=0; i<8; i++)
   {
     thread_start[i]=1;
   }
   __compiler_barrier();
/* Wait for all threads to complete */
   for (int i=0; i<8; i++)
   {
      while (thread_start[i]==1){}
   }
}
```

パート III

# ライブラリ

◆ ◆ ◆ 第 11 章

# 11 ライブラリの使用

ライブラリは、いくつかのアプリケーション間でコードを共有したり、非常に大規模なアプリケーションの複雑さを軽減する方法を提供します。C++ コンパイラでは、さまざまなライブラリを使用できます。この章では、これらのライブラリの使用方法を説明します。

## 11.1 C ライブラリ

Oracle Solaris オペレーティングシステムでは、いくつかのライブラリが `/usr/lib` にインストールされます。このライブラリのほとんどは C インタフェースを持っています。デフォルトでは `libc` および `libm` ライブラリが `CC` ドライバによってリンクされます。ライブラリ `libthread` は、`-mt` オプションを指定した場合にのみリンクされます。その他のシステムライブラリをリンクするには、`-l` オプションでリンク時に指定する必要があります。たとえば、`libdemangle` ライブラリをリンクするには、リンク時に `-ldemangle` を `CC` コマンド行に指定します。

```
example% CC text.c -ldemangle
```

C++ コンパイラには、独自の実行時ライブラリが複数あります。すべての C++ アプリケーションは、`CC` ドライバによってこれらのライブラリとリンクされます。C++ コンパイラには、次の節に示すようにこれ以外にも便利なライブラリがいくつかあります。

## 11.2 C++ コンパイラ付属のライブラリ

C++ コンパイラには、いくつかのライブラリが添付されています。

次の表に、C++ コンパイラに添付されるライブラリと、それらを使用できるモードを示します。

表 11–1　C++ コンパイラに添付されるライブラリ

| ライブラリ | 説明 |
|---|---|
| libstlport | 標準ライブラリの STLport 実装 |
| libstlport_dbg | デバッグモード用 STLport ライブラリ |
| libCrun | C++ 実行時 |
| libCstd | C++ 標準ライブラリ |
| libiostream | 従来の iostream |
| libcsunimath | -xia オプションをサポート |
| librwtool | Tools.h++ 7 |
| librwtool_dbg | デバッグ可能な Tools.h++ 7 |
| libgc | ガベージコレクション |
| libdemangle | 復号化 |
| sunperf | Sun Performance Library |

注 – STLport、Rogue Wave、または Oracle Solaris Studio C++ ライブラリの構成マクロを再定義したり変更したりしないでください。ライブラリは C++ コンパイラとともに動作するよう構成および構築されています。libCstd と Tools.h++ は互いに働き合うように構成されているので、その構成マクロを変更すると、プログラムのコンパイルやリンクが行われなくなったり、プログラムが正しく実行されなくなったりします。

### 11.2.1 C++ ライブラリの説明

この節では、各 C++ ライブラリについて簡単に説明します。

- libCrun – コンパイラがデフォルトの標準モード (-compat=5) で必要とする実行時サポートが含まれています。new と delete、例外、RTTI がサポートされます。

  libCstd – C++ 標準ライブラリ。特に、このライブラリには iostream が含まれています。既存のソースで従来の iostream を使用している場合には、ソースを新しいインタフェースに合わせて修正しないと、標準 iostream を使用できません。詳細は、オンラインマニュアルの『Standard C++ Library Class Reference』を参照してください。

- `libiostream` – `-compat=5` で構築した従来の iostreams ライブラリ。既存のソースで従来の iostream を使用している場合は、`libiostream` を使用すれば、ソースを修正しなくてもこれらのソースを標準モード (`-compat=5`) でコンパイルできます。このライブラリを使用するには、`-library=iostream` を使用します。

  注 – 標準ライブラリのほとんどの部分は、標準 iostream を使用することに依存しています。従来の iostream を同一のプログラム内で使用すると、問題が発生する可能性があります。

- `libstlport` – C++ 標準ライブラリの STLport 実装。このライブラリを使用するには、デフォルトの `libCstd` の代わりにオプション `-library=stlport4` を指定します。ただし、`libstlport` と `libCstd` の両方を同一プログラム内で使用することはできません。インポートしたライブラリを含むすべてを、どちらか一方のライブラリだけを使ってコンパイルしリンクする必要があります。
- `librwtool` (`Tools.h++`): RogueWave の C++ 基礎クラスライブラリです。Version 7 が提供されています。このライブラリは廃止され、新しいコードでこのライブラリを使用することは非推奨です。RW Tools.h++ を使用していた、C++ 4.2 用に作成されたプログラムに対応する目的で、このライブラリは提供されています。
- `libgc` – 展開モードまたはガベージコレクションモードで使用します。`libgc` ライブラリにリンクするだけで、プログラムのメモリーリークを自動的および永久的に修正できます。プログラムを `libgc` ライブラリとリンクする場合は、`free` や `delete` を呼び出さずに、それ以外は通常どおりにプログラムを記述できます。ガベージコレクションライブラリは、動的読み込みライブラリと依存関係があるため、プログラムをリンクするときは、`-lgc` および `-ldl` を指定します。

  詳細については、`gcFixPrematureFrees`(3) および `gcInitialize`(3) のマニュアルページを参照してください。
- `libdemangle` – C++ 符号化名を復号化するときに使用します。

## 11.2.2 C++ ライブラリのマニュアルページへのアクセス

この節で説明しているライブラリに関するマニュアルページは 1、3、3C++、および 3cc4 の各節にあります。

C++ ライブラリのマニュアルページにアクセスするには次のとおり入力してください。

```
example% man library-name
```

C++ ライブラリの Version 4.2 のマニュアルページにアクセスするには次のコマンドを入力してください。

```
example% man -s 3CC4 library-name
```

### 11.2.3 デフォルトの C++ ライブラリ

C++ ライブラリは実行可能プログラムをビルドするときにデフォルトでリンクされますが、共有ライブラリ (.so) をビルドするときにはリンクされません。共有ライブラリを構築するとき、必要なすべてのライブラリが明示的に指定される必要があります。-zdefs オプションによって、リンカーは必要なライブラリが省略されている場合に通知を出します。このオプションは実行可能プログラムをビルドする場合のデフォルトです。次のライブラリが CC ドライバによってデフォルトでリンクされます。

```
-lCstd -lCrun -lm -lc
```

詳細は、214 ページの「A.2.49 -library=*l*[ ,*l*...]」 を参照してください。

## 11.3 関連するライブラリオプション

CC ドライバには、ライブラリを使用するためのオプションがいくつかあります。

- リンクするライブラリを指定するには、-l オプションを使用します。
- ライブラリを検索するディレクトリを指定するには、-L オプションを使用します。
- マルチスレッド化コードをコンパイルしてリンクするには、-mt オプションを使用します。
- 区間演算ライブラリをリンクするには、-xia オプションを使用します。
- Fortran または C99 実行時ライブラリをリンクするには、-xlang オプションを使用します。
- Oracle Solaris Studio C++ コンパイラに付属する次のライブラリを指定するには、-library オプションを使用します。

```
libCrun
libCstd
libiostream
libC
libcomplex
libstlport, libstlport_dbg
librwtool, librwtool_dbg
libgc
sunperf
```

注 – librwtool の従来の iostream 形式を使用するには、-library=rwtools7 オプションを使用します。librwtool の標準 iostream 形式を使用するには、-library=rwtools7_std オプションを使用します。

-library オプションと -staticlib オプションの両方に指定されたライブラリは静的にリンクされます。例:

libstdcxx (Oracle Solaris OS の一部として配布)

次のコマンドでは Tools.h++ Version 7 の従来の iostream 形式と libiostream ライブラリが動的にリンクされます。

```
example% CC test.cc -library=rwtools7,iostream
```

次のコマンドでは libgc ライブラリが静的にリンクされます。

```
example% CC test.cc -library=gc -staticlib=gc
```

次のコマンドでは ライブラリ libCrunおよび libCstd がリンク対象から除外されます。指定しない場合は、これらのライブラリは自動的にリンクされます。

```
example% CC test.cc -library=no%Crun,no%Cstd
```

デフォルトでは、CC は、指定されたコマンド行オプションに従ってさまざまなシステムライブラリをリンクします。-xnolib (または -nolib) を指定した場合、CC は、-l オプションを使用してコマンド行で明示的に指定したライブラリだけをリンクします。-xnolib または -nolib を使用した場合、-library オプションが存在していても無視されます。

-R オプションは、動的ライブラリの検索パスを実行可能ファイルに組み込むときに使用します。実行時リンカーは、実行時にこれらのパスを使ってアプリケーションに必要な共有ライブラリを探します。CC ドライバは、コンパイラが標準の場所にインストールされている場合、デフォルトで –R<*install-directory*>/lib を ld に渡します。共有ライブラリのデフォルトパスが実行可能ファイルに組み込まれないようにするには、-norunpath を使用します。

デフォルトでは、リンカーは /lib および /usr/lib を検索します。-L オプションでこれらのディレクトリやコンパイラのインストールディレクトリを指定しないでください。

配備用に構築するプログラムは、コンパイラのディレクトリでライブラリを参照することを防止する -norunpath または -R オプションを使用して構築するようにしてください。133 ページの「11.6 共有ライブラリの使用」を参照してください。

# 11.4 クラスライブラリの使用

一般に、クラスライブラリを使用するには 2 つの手順が必要です。

1. ソースコードに適切なヘッダーをインクルードする。
2. プログラムをオブジェクトライブラリとリンクする。

## 11.4.1 iostream ライブラリ

C++ コンパイラには、2 通りの iostream が実装されています。

- 従来の **iostream**。この用語は、C++ 4.0、4.0.1、4.1、および 4.2 コンパイラに付属する iostream ライブラリを表し、以前は `cfront` ベースの 3.0.1 コンパイラに付属していました。このライブラリに標準はありません。これは `libiostream` で使用できます。
- 標準の **iostream**。これは C++ 標準ライブラリ `libCstd` の一部で、標準モードでのみ使用できます。これは、従来の iostream とはバイナリ互換性もソース互換性もありません。

すでに C++ のソースがある場合、そのコードは従来の iostream を使用しており、次の例のような形式になっていると思われます。

```
// file prog1.cc
#include <iostream.h>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

次の例では、標準 iostream が使用されています。

```
// file prog2.cc
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

次のコマンドは、`prog2.cc` をコンパイル、リンクして、`prog2` という実行可能なプログラムを生成します。プログラムは標準モードでコンパイルされます。このモードでは、標準の iostream ライブラリを含む `libCstd` がデフォルトでリンクされます。

```
example% CC prog2.cc -o prog2
```

#### 11.4.1.1 従来の iostream およびレガシー RogueWave ツールについての注意

いわゆる「従来型の」iostream は、iostream の 1986 オリジナルバージョンで、C++ 標準で置き換えられました。これは `-library=rwtools7,iostream` オプションで選択されます。「従来型の」iostream の実装はどれも同じでなく、廃止される場合は別として、これを使用するコードには移植性がありません。このライブラリとオプションは、将来の Oracle Solaris Studio リリースで中止されます。

レガシー Sun Studio および Oracle Studio で提供されている RW Tools.h++ ツールセットは 1990 年代に遡り、それ以来大きく更新されていません。その `time` および `date` クラスには、サマータイムに関する修正不能な重大な問題があります。(このツールセットの機能は、C++ 標準と、BOOST などのオープンソースライブラリで現在使用できます。)RW Tools.h++ は `-library=rwtools7` または `-library=rwtools7_std` オプションによって選択され、将来の Oracle Solaris Studio リリースで中止されます。

### 11.4.2 C++ ライブラリのリンク

次の表に、C++ ライブラリのリンクに関するコンパイラオプションを示します。詳細は、214 ページの「A.2.49 `-library=`*l*[ ,*l*...]」 を参照してください。

表 11–2 C++ ライブラリにリンクするためのコンパイラオプション

| ライブラリ | オプション |
|---|---|
| 従来の `iostream` | `-library=iostream` |
| `Tools.h++` Version 7 | `-library=rwtools7,iostream`<br>`-library=rwtools7_std` |
| `Tools.h++` Version 7 デバッグ | `-library=rwtools7_dbg,iostream`<br>`-library=rwtools7_std_dbg` |
| ガベージコレクション | `-library=gc` |
| STLport Version 4 | `-library=stlport4` |
| STLport Version 4 デバッグ | `-library=stlport4_dbg` |
| Apache `stdcxx` Version 4 | `-library=stdcxx4` |
| Sun Performance Library | `-library=sunperf` |

# 11.5 標準ライブラリの静的リンク

CC ドライバは、各デフォルトライブラリの -l*lib* オプションをリンカーに渡すことによって、libc や libm などのいくつかのライブラリの共有バージョンにデフォルトでリンクします。(デフォルトライブラリのリストについては、128 ページの「11.2.3 デフォルトの C++ ライブラリ」を参照してください。)

このようにデフォルトのライブラリを静的にリンクする場合、-library オプションと –staticlib オプションを一緒に使用できます。例:

```
example% CC test.c -staticlib=Crun
```

この例では、-library オプションが明示的にコマンドに指定されていません。この場合、標準モード (デフォルトのモード) では、-library のデフォルトの設定が Cstd,Crun であるため、-library オプションを明示的に指定する必要はありません。

あるいは、-xnolib コンパイラオプションも使用できます。-xnolib オプションを指定すると、ドライバは自動的には -l オプションを ld に渡しません。次の例は、libCrun と静的に、libm および libc と動的にリンクする方法を示します。

```
example% CC test.c -xnolib -lCstd -Bstatic -lCrun -Bdynamic -lm -lc
```

-l オプションの順序は重要です。-lCstd、-lCrun、および -lm オプションは、-lc の前に表示します。

---

注 – libCrun および libCstd を静的にリンクすることはお勧めできません。/usr/lib 内の動的バージョンは、インストール先の Oracle Solaris のバージョンで動作するよう構築します。

---

ほかのライブラリにリンクする CC オプションもあります。そうしたライブラリへのリンクも -xnolib によって行われないように設定できます。たとえば、-mt オプションを指定すると、CC ドライバは、-lthread を ld に渡します。これに対し、-mt と -xnolib の両方を使用すると、CC ドライバは ld に -lthread を渡しません。詳細は、284 ページの「A.2.147 -xnolib」を参照してください。ld については、Solaris に関するマニュアル『リンカーとライブラリ』を参照してください。

---

注 – /lib および /usr/lib にある Oracle Solaris ライブラリの静的バージョンは、もう使用できません。たとえば、libc を静的にリンクしようとすると失敗します。

```
CC hello.cc -xnolib -lCrun -lCstd -Bstatic -lc
```

---

# 11.6 共有ライブラリの使用

次の C++ 実行時共有ライブラリは、C++ コンパイラの一部として出荷されています。

- `libCCexcept.so.1` (SPARC Solaris のみ)
- `libcomplex.so.5` (Solaris のみ)
- `librwtool.so.2`
- `libstlport.so.1`

Linux では、次の追加ライブラリが C++ コンパイラの一部として出荷されています。

- `libCrun.so.1`
- `libCstd.so.1`
- `libdemangle.so`
- `libiostream.so.1`

最新の Oracle Solaris リリースでは、次の追加ライブラリがほかのライブラリとともに、Oracle Solaris C++ 実行時ライブラリパッケージである `SUNWlibC` の一部としてインストールされます。

アプリケーションが、C++ コンパイラの一部として出荷されている共有ライブラリのいずれかを使用している場合は、CC ドライバは *runpath* に調整を加え (`-R` オプションを参照)、実行可能ファイルの構築に使用するライブラリの場所を指すようにします。あとで、同じバージョンのコンパイラを同じ場所にインストールしていないコンピュータに実行可能ファイルを配備する場合は、必要な共有ライブラリが見つかりません。

プログラムの起動時に、ライブラリはまったく見つからない、あるいは誤ったバージョンのライブラリが使用される可能性があり、プログラムの正しくない動作につながります。このような状況では、必要なライブラリを実行可能ファイルとともに出荷し、それらのライブラリのインストール場所を指す *runpath* を指定して構築を行うべきです。

『Using and Redistributing Solaris Studio Libraries in an Application』の記事には、このトピックについての完全な説明と例が記載されています。これは `http://www.oracle.com/technetwork/articles/servers-storage-dev/redistrib-libs-344133.html` にあります。

# 11.7 C++ 標準ライブラリの置き換え

ただし、コンパイラに添付された標準ライブラリを置き換えることは危険で、必ずしもよい結果につながるわけではありません。基本的な操作は、コンパイラとともに提供される標準のヘッダーとライブラリを無効にして、新しいヘッダーファイルとライブラリがあるディレクトリと、ライブラリ自身の名前を指定することです。

コンパイラでは、標準ライブラリの STL ポートおよび Apache stdcxx 実装がサポートされます。詳細は、141 ページの「12.2 STLport」と143 ページの「12.3 Apache stdcxx 標準ライブラリ」を参照してください。

## 11.7.1 置き換え可能な対象

ほとんどの標準ライブラリおよびそれに関連するヘッダーは置き換え可能です。置き換えるライブラリが `libCstd` である場合は、次の関連するヘッダーも置き換える必要があります。

```
<algorithm> <bitset> <complex> <deque> <fstream <functional> <iomanip> <ios>
<iosfwd> <iostream> <istream> <iterator> <limits> <list> <locale> <map> <memory>
<numeric> <ostream> <queue> <set> <sstream> <stack> <stdexcept> <streambuf>
<string> <strstream> <utility> <valarray> <vector>
```

ライブラリの置き換え可能な部分は、いわゆる「STL」と呼ばれているもの、文字列クラス、iostream クラス、およびそれらの補助クラスです。このようなクラスとヘッダーは相互に依存しているため、それらの一部を置き換えるだけでは通常は機能しません。一部を変更する場合でも、すべてのヘッダーと `libCstd` のすべてを置き換える必要があります。

## 11.7.2 置き換え不可能な対象

標準ヘッダー `<exception>`、`<new>`、および `<typeinfo>` は、コンパイラ自身と `libCrun` に密接に関連しているため、これらを置き換えることは安全ではありません。ライブラリ `libCrun` は、コンパイラが依存している多くの「補助」関数が含まれているため置き換えることはできません。

C から派生した 17 個の標準ヘッダー (`<stdlib.h>`、`<stdio.h>`、`<string.h>` など) は、Oracle Solaris オペレーティングシステムと基本 Solaris 実行時ライブラリ `libc` に密接に関連しているため、これらを置き換えることは安全ではありません。これらのヘッダーの C++ バージョン (`<cstdlib>`、`<cstdio>`、`<cstring>` など) は基本の C バージョンのヘッダーに密接に関連しているため、これらを置き換えることは安全ではありません。

### 11.7.3 代替ライブラリのインストール

代替ライブラリをインストールするには、まず、代替ヘッダーの位置と libCstd の代わりに使用するライブラリを決定する必要があります。理解しやすくするために、ここでは、ヘッダーを /opt/mycstd/include にインストールし、ライブラリを /opt/mycstd/lib にインストールすると仮定します。ライブラリの名前は libmyCstd.a であると仮定します。(ライブラリ名は通常 "lib" で始まります。)

### 11.7.4 代替ライブラリの使用

コンパイルごとに -I オプションを指定して、ヘッダーがインストールされている位置を指示します。さらに、-library=no%Cstd オプションを指定して、コンパイラ独自のバージョンの libCstd ヘッダーが検出されないようにします。例:

```
example% CC -I/opt/mycstd/include -library=no%Cstd... (compile)
```

-library=no%Cstd オプションを指定しているため、コンパイル中、コンパイラ独自のバージョンのヘッダーがインストールされているディレクトリは検索されません。

プログラムまたはライブラリのリンクごとに -library=no%Cstd オプションを指定して、コンパイラ独自の libCstd が検出されないようにします。さらに、-L オプションを指定して、代替ライブラリがインストールされているディレクトリを指示します。さらに、-l オプションを指定して、代替ライブラリを指定します。例:

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd... (link)
```

あるいは、-L や -l オプションを使用せずに、ライブラリの絶対パス名を直接指定することもできます。例:

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a... (link)
```

-library=no%Cstd オプションを指定しているため、リンク中、コンパイラ独自のバージョンの libCstd はリンクされません。

### 11.7.5 標準ヘッダーの実装

C には、<stdio.h>、<string.h>、<stdlib.h> などの 17 個の標準ヘッダーがあります。これらのヘッダーは Oracle Solaris オペレーティングシステムの一部として提供され、/user/include にあります。C++ にも同様のヘッダーがありますが、さまざまな宣言の名前が大域名前空間と std 名前空間の両方に存在するという要件が付加されています。

また、C++ には、C 標準ヘッダー (<cstdio> 、<cstring>、<cstdlib> など) のそれぞれについても専用のバージョンがあります。C++ 版の C 標準ヘッダーでは、宣言名は std 名前空間にのみ存在します。C++ には、32 個の独自の標準ヘッダー (<string>、<utility>、<iostream> など) も追加されています。

標準ヘッダーの実装で、C++ ソースコード内の名前がインクルードするテキストファイル名として使用されているとしましょう。たとえば、標準ヘッダーの <string> (または <string.h>) が、あるディレクトリにある string (または string.h) というファイルを参照するものとします。この実装には、次の欠点があります。

- ヘッダーファイルにファイル名接尾辞 (拡張子) がない場合に、ヘッダーファイルのみを検索すること、またはヘッダーファイルに関する規則を示す makefile を作成することができない。
- string というディレクトリまたは実行可能プログラムがあると、そのディレクトリまたはプログラムが標準ヘッダーファイルの代わりに検出される可能性がある。

こうした問題を解決するため、コンパイラの include ディレクトリには、ヘッダーと同じ名前を持つファイルと、 一意の接尾辞 .SUNWCCh を持つ、そのファイルへのシンボリックリンクが含まれています。 SUNW はコンパイラに関係するあらゆるパッケージに対する接頭辞、CC は C++ コンパイラの意味、.h はヘッダーファイルの通常の接尾辞です。つまり <string> と指定された場合、コンパイラは <string.SUNWCCh> と書き換え、その名前を検索します。接尾辞付きの名前は、コンパイラ専用の include ディレクトリにだけ存在します。このようにして見つけられたファイルがシンボリックリンクの場合 (通常はそうである)、コンパイラは、エラーメッセージやデバッガの参照でそのリンクを 1 回だけ間接参照し、その参照結果 (この場合は string) をファイル名として使用します。ファイルの依存関係情報を送るときは、接尾辞付きの名前の方が使用されます。

この名前の書き換えは、2 つのバージョンがある 17 個の標準 C ヘッダーと 32 個の標準 C++ ヘッダーのいずれかを、パスを指定せずに山括弧 <> で囲んで指定した場合にだけ行われます。山括弧の代わりに引用符が使用されるか、パスが指定されるか、ほかのヘッダーが指定された場合、名前の書き換えは行われません。

次の表は、よくある書き換え例をまとめています。

表 11–3 ヘッダー検索の例

| ソースコード | コンパイラによる検索 | 注釈 |
|---|---|---|
| <string> | string.SUNWCCh | C++ の文字列テンプレート |
| <cstring> | cstring.SUNWCCh | C の string.h の C++ 版 |
| <string.h> | string.h.SUNWCCh | C の string.h |
| <fcntl.h> | fcntl.h | 標準 C および C++ ヘッダー以外 |

表 11–3 ヘッダー検索の例 (続き)

| ソースコード | コンパイラによる検索 | 注釈 |
|---|---|---|
| "string" | string | 山括弧ではなく、二重引用符 |
| <../string> | ../string | パス指定がある場合 |

コンパイラが *header*.SUNWCCh (header はヘッダー名) を見つけられない、コンパイラは、#include 指令で指定された名前で検索し直します。たとえば、#include <string> という指令を指定した場合、コンパイラは string.SUNWCCh という名前のファイルを見つけようとします。この検索が失敗した場合、コンパイラは string という名前のファイルを探します。

### 11.7.5.1 標準 C++ ヘッダーの置き換え

135 ページの「11.7.5 標準ヘッダーの実装」で説明している検索アルゴリズムのため、135 ページの「11.7.3 代替ライブラリのインストール」で説明している SUNWCCh バージョンの代替 ヘッダーを指定する必要はありません。ただし、記載されているいくつかの問題が発生した場合、推奨される解決方法は、接尾辞が付いていないヘッダーごとに、接尾辞 .SUNWCCh を持つシンボリックリンクを追加することです。つまり、ファイルが utility の場合、次のコマンドを実行します。

```
example% ln -s utility utility.SUNWCCh
```

utility.SUNWCCh というファイルを探すとき、コンパイラは 1 回目の検索でこのファイルを見つけます。そのため、utility という名前のほかのファイルやディレクトリを誤って検出してしまうことはありません。

### 11.7.5.2 標準 C ヘッダーの置き換え

標準 C ヘッダーの置き換えはサポートされていません。それでもなお、独自のバージョンの標準ヘッダーを使用する場合、推奨される手順は次のとおりです。

- すべての代替ヘッダーを 1 つのディレクトリに置きます。
- そのディレクトリ内にある代替ヘッダーごとに header.SUNWCCh (header はヘッダー名) へのシンボリックリンクを作成します。
- コンパイラを呼び出すごとに -I 指令を指定して、代替ヘッダーが置かれているディレクトリが検索されるようにします。

たとえば、<stdio.h> と <cstdio> の代替ヘッダーがあるとします。stdio.h と cstdio をディレクトリ /myproject/myhdr に置きます。このディレクトリ内で、次のコマンドを実行します。

```
example% ln -s stdio.h stdio.h.SUNWCCh
example% ln -s cstdio cstdio.SUNWCCh
```

コンパイルのたびに、オプション -I/myproject/mydir を使用します。

警告:

- C ヘッダーを置き換える場合は、対になっているもう一方のヘッダーを置き換える必要があります。たとえば、<time.h> を置き換えるときは、<ctime> も置き換える必要があります。
- 代替ヘッダーは、置き換える前のヘッダーと同じ効果を持っている必要があります。これは、さまざまな実行時ライブラリ (libCrun、libC、libCstd、libc、および librwtool) が標準ヘッダーの定義を使用して構築されているためです。同じ効果を持っていない場合、作成したプログラムはほとんどの場合、正しく動作しません。

◆ ◆ ◆ 第 12 章

# C++ 標準ライブラリの使用

デフォルトモード (標準モード) のコンパイルでは、コンパイラは C++ 標準で指定されている完全なライブラリにアクセスします。このライブラリコンポーネントには、非公式に「標準テンプレートライブラリ」 (STL) と呼ばれているものに加えて、次のコンポーネントが含まれています。

- 文字列クラス
- 数値クラス
- 標準ストリーム I/O クラス
- 基本的なメモリー割り当て
- 例外クラス
- 実行時の型情報 (RTTI)

用語 STL に公式な定義はありませんが、一般的にはコンテナ、反復子、およびアルゴリズムを含むとされています。標準ライブラリヘッダーの次のサブセットが、STL を構成すると考えることができます。

- `<algorithm>`
- `<deque>`
- `<iterator>`
- `<list>`
- `<map>`
- `<memory>`
- `<queue>`
- `<set>`
- `<stack>`
- `<utility>`
- `<vector>`

C++ 標準ライブラリ (`libCstd`) は、RogueWave 標準 C++ ライブラリ、Version 2 に基づいています。このライブラリはデフォルトです。

また、C++ コンパイラで、STLport の標準ライブラリの Version 4.5.3 がサポートされました。libCstd がデフォルトのライブラリですが、代わりに STLport の製品を使用できるようになりました。詳細は、141 ページの「12.2 STLport」を参照してください。

コンパイラに付属している C++ 標準ライブラリの代わりに、独自の C++ 標準ライブラリを使用できます。その場合は、-library=no%Cstd オプションを使用します。ただし、コンパイラに添付された標準ライブラリを置き換えることは危険で、必ずしもよい結果につながるわけではありません。詳細は、134 ページの「11.7 C++ 標準ライブラリの置き換え」を参照してください。

# 12.1 C++ 標準ライブラリのヘッダーファイル

完全な標準ライブラリのヘッダーとそれぞれの概要は表 12–1 に一覧表示します。

表 12–1 C++ 標準ライブラリのヘッダーファイル

| ヘッダーファイル | 説明 |
|---|---|
| <algorithm> | コンテナ操作のための標準アルゴリズム |
| <bitset> | 固定長のビットシーケンス |
| <complex> | 複素数を表す数値型 |
| <deque> | 先頭と末尾の両方で挿入と削除が可能なシーケンス |
| <exception> | 事前定義済み例外クラス |
| <fstream> | ファイルとのストリーム入出力 |
| <functional> | 関数オブジェクト |
| <iomanip> | iostream のマニピュレータ |
| <ios> | iostream の基底クラス |
| <iosfwd> | iostream クラスの先行宣言 |
| <iostream> | 基本的なストリーム入出力機能 |
| <istream> | 入力ストリーム |
| <iterator> | シーケンスの内容にくまなくアクセスするためのクラス |
| <limits> | 数値型の属性 |
| <list> | 順序付きシーケンス |
| <locale> | 国際化のサポート |
| <map> | キーと値を対にして使用する連想コンテナ |

表 12–1　C++ 標準ライブラリのヘッダーファイル　　(続き)

| ヘッダーファイル | 説明 |
|---|---|
| <memory> | 特殊なメモリーアロケータ |
| <new> | 基本的なメモリー割り当てと解放 |
| <numeric> | 汎用の数値演算 |
| <ostream> | 出力ストリーム |
| <queue> | 先頭への挿入と末尾からの削除が可能なシーケンス |
| <set> | 一意キーを使用する連想コンテナ |
| <sstream> | メモリー上の文字列との入出力ストリーム |
| <stack> | 先頭への挿入と先頭からの削除が可能なシーケンス |
| <stdexcept> | 追加標準例外クラス |
| <streambuf> | iostream 用のバッファークラス |
| <string> | 文字シーケンス |
| <typeinfo> | 実行時の型識別 |
| <utility> | 比較演算子 |
| <valarray> | 数値プログラミング用の値配列 |
| <vector> | ランダムアクセスが可能なシーケンス |

## 12.2 STLport

libCstd の代替ライブラリを使用する場合は、標準ライブラリの STLport 実装を使用します。 libCstd をオフにして、STLport ライブラリで代用するには、次のコンパイラオプションを使用します。

- -library=stlport4

詳細は、214 ページの「A.2.49 -library=*l*[ ,*l*...]」 を参照してください。

このリリースでは、libstlport.a という静的アーカイブおよび libstlport.so という動的ライブラリの両方が含まれています。

STLport 実装を使用するかどうかは、次のことを考慮して判断してください。

- STLport は、オープンソースの製品で、リリース間での互換性は保証されません。つまり、STLport の将来のバージョンでコンパイルすると、STLport 4.5.3 でコンパイルしたアプリケーションが破壊される可能性があります。また、STLport 4.5.3 を使用してコンパイルしたバイナリは、STLport の将来のバージョンを使用してコンパイルしたバイナリとリンクできない場合もあります。

- stlport4、Cstd、および iostream のライブラリは、固有の入出力ストリームを実装しています。これらのライブラリの 2 個以上を -library オプションを使って指定した場合、プログラム動作が予期しないものになる恐れがあります。
- コンパイラの将来のリリースには、STLport4 が含まれない可能性があります。STLport の新しいバージョンだけが含まれる可能性があります。コンパイラオプションの -library=stlport4 は、将来のリリースでは使用できず、STLport のそれ以降のバージョンを示すオプションに変更される可能性があります。
- STLport では、Tools.h++ はサポートされません。
- STLport は、デフォルトの libCstd とはバイナリ互換ではありません。STLport の標準ライブラリの実装を使用する場合は、-library=stlport4 オプションを指定してすべてのファイルのコンパイルおよびリンクを実行する必要があります。このことは、たとえば STLport 実装と C++ 区間演算ライブラリ libCsunimath を同時に使用できないことを意味します。その理由は、libCsunimath のコンパイルに使用されたのが、STLport ではなくデフォルトライブラリヘッダーであるためです。
- STLport 実装を使用する場合は、コードから暗黙に参照されるヘッダーファイルをインクルードしてください。標準のヘッダーは、実装の一部として相互にインクルードできます (必須ではありません)。

## 12.2.1 再配布とサポートされる STLport ライブラリ

エンドユーザーオブジェクトコードライセンスの条件に基づいて、実行可能ファイルまたはライブラリとともに再配布可能なライブラリおよびオブジェクトファイルの一覧は、再配布の README ファイルを参照してください。この README ファイルの C++ のセクションに、このリリースのコンパイラがサポートしている STLport.so のバージョンが記載されています。この README ファイルは、http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html にある、Oracle Solaris Studio ソフトウェアのこのリリースについての法的ページにあります。

次の例は、ライブラリの実装について移植性のない想定が行われているため、STLport を使用してコンパイルできません。特に、<vector> または <iostream> が <iterator> を自動的にインクルードすることを想定していますが、これは正しい想定ではありません。

```
#include <vector>
#include <iostream>

using namespace std;

int main ()
{
    vector <int> v1 (10);
    vector <int> v3 (v1.size());
    for (int i = 0; i < v1.size (); i++)
      {v1[i] = i; v3[i] = i;}
```

```
    vector <int> v2(v1.size ());
    copy_backward (v1.begin (), v1.end (), v2.end ());
    ostream_iterator<int> iter (cout, " ");
    copy (v2.begin (), v2.end (), iter);
    cout << endl;
    return 0;
}
```

問題を解決するには、ソースで <iterator> をインクルードします。

## 12.3 Apache stdcxx 標準ライブラリ

-library=stdcxx4 でコンパイルすることによって、デフォルトの libCstd の代わりに Apache stdcxx Version 4 C++ 標準ライブラリを Solaris で使用してください。このオプションにより、-mt オプションも暗黙的に設定されます。stdcxx ライブラリには、マルチスレッドモードが必要です。このオプションは、コンパイルのたびに、およびアプリケーション全体のリンクコマンドで一貫して使用する必要があります。-library=stdcxx4 を使用してコンパイルされたコードは、デフォルトの -library=Cstd または省略可能な -library=stlport4 を使用してコンパイルされたコードと同じプログラムでは使用できません。

Apache stdcxx ライブラリを使用するときは、次のことに注意してください。

- stdcxx および iostream のライブラリは、入出力ストリームの固有の実装を提供します。これらのライブラリの 2 個以上を -library オプションを使って指定した場合、プログラム動作が予期しないものになる恐れがあります。
- Tools.h++ は stdcxx でサポートされません。
- C++ 区間演算ライブラリ (libCsunimath) は stdcxx でサポートされません。
- stdcxx ライブラリは、デフォルトの libCstd および STLport と互換性のないバイナリです。標準ライブラリの stdcxx 実装を使用する場合は、-library=stdcxx4 オプションを指定して、他社製ライブラリを含むすべてのファイルのコンパイルおよびリンクを実行する必要があります。

◆ ◆ ◆ 第 13 章

# 13 従来の iostream ライブラリの使用

C++ も C と同様に組み込み型の入出力文はありません。その代わりに、出力機能はライブラリで提供されています。C++ コンパイラでは iostream クラスに対して、従来型の実装と ISO 標準の実装を両方とも提供しています。

- デフォルトでは、従来型の iostream クラスは libiostream に含まれています。従来型の libiostream クラスを使用したソースコードがあり、ソースを標準モードでコンパイルするときは、iostream を使用します。従来型の iostream の機能を標準モードで使用するには、iostream.h ヘッダーファイルをインクルードし、-library=iostream オプションを使用してコンパイルします。
- 標準の iostream クラスは標準モードだけで使用でき、C++ 標準ライブラリ libCstd に含まれています。

この章では、従来型の iostream ライブラリの概要と使用例を説明します。この章では、iostream ライブラリを完全に説明しているわけではありません。詳細は、iostream ライブラリのマニュアルページを参照してください。従来型の iostream マニュアルページにアクセスするには、次のコマンドを入力します。**man -s 3CC4** *name*

131 ページの「11.4.1.1 従来の iostream およびレガシー RogueWave ツールについての注意」を参照してください。

## 13.1 定義済みの iostream

定義済みの iostream には、次のものがあります。

- cin、 標準入力と結合しています。
- cout、標準出力と結合しています。
- cerr、 標準エラーと結合しています。
- clog、標準エラーと結合しています。

定義済み iostream は、cerr を除いて完全にバッファーされます。147 ページの「13.3.1 iostream を使用した出力」と 150 ページの「13.3.2 iostream を使用した入力」を参照してください。

# 13.2 iostream 操作の基本構造

iostream ライブラリを使用すると、プログラムで必要な数の入出力ストリームを使用できます。それぞれのストリームは、次のどれかを入力先または出力先とします。

- 標準入力
- 標準出力
- 標準エラー
- ファイル
- 文字型配列

ストリームは、入力のみまたは出力のみと制限して使用することも、入出力両方に使用することもできます。iostream ライブラリでは、次の 2 つの処理階層を使用してこのようなストリームを実現しています。

- 下層では、単なる文字ストリームであるシーケンスを実現します。シーケンスは、streambuf クラスか、その派生クラスで実現されています。
- 上層では、シーケンスに対してフォーマット操作を行います。フォーマット操作は istream と ostream の 2 つのクラスで実現されます。これらのクラスはメンバーに streambuf クラスから派生したオブジェクトを持っています。このほかに、入出力両方が実行されるストリームに対しては iostream クラスがあります。

標準入力、標準出力、標準エラーは、istream または ostream から派生した特殊なクラスオブジェクトで処理されます。

ifstream、ofstream、fstream の 3 つのクラスはそれぞれ istream、ostream、iostream から派生しており、ファイルへの入出力を処理します。

istrstream、ostrstream、strstream の 3 つのクラスはそれぞれ istream、ostream、および iostream から派生しており、文字型配列への入出力を処理します。

入力ストリームまたは出力ストリームをオープンする場合は、どれかの型のオブジェクトを生成し、そのストリームのメンバー streambuf をデバイスまたはファイルに関連付けます。通常、関連付けはストリームコンストラクタで行うので、ユーザーが直接 streambuf を操作することはありません。標準入力、標準出力、エラー出力に対しては、iostream ライブラリであらかじめストリームオブジェクトを定義してあるので、これらのストリームについてはユーザーが独自にオブジェクトを生成する必要はありません。

ストリームへのデータの挿入 (出力)、ストリームからのデータの抽出 (入力)、挿入または抽出したデータのフォーマット制御には、演算子または iostream のメンバー関数を使用します。

新たなデータ型 (ユーザー定義のクラス) を挿入したり抽出したりするときは一般に、挿入演算子と抽出演算子の多重定義をユーザーが行います。

## 13.3 従来の iostream ライブラリの使用

従来型の iostream ライブラリからルーチンを使用するには、必要なライブラリ部分のヘッダーファイルをインクルードする必要があります。 次の表で各ヘッダーファイルについて説明します。

表 13–1 iostream ルーチンのヘッダーファイル

| ヘッダーファイル | 説明 |
|---|---|
| iostream.h | iostream ライブラリの基本機能の宣言。 |
| fstream.h | ファイルに固有の iostream と streambuf の宣言。この中で iostream.h をインクルードします。 |
| strstream.h | 文字型配列に固有の iostream と streambuf の宣言。この中で iostream.h をインクルードします。 |
| iomanip.h | マニピュレータ値の宣言。マニピュレータ値とはiostream に挿入または iostream から抽出する値で、特別の効果を引き起こします。この中で iostream.h をインクルードします。 |
| stdiostream.h | (廃止) stdio FILE の使用に固有の iostream と streambuf の宣言。この中で iostream.h. をインクルードします。 |
| stream.h | (旧形式) この中で iostream.h、fstream.h、iomanip.h、stdiostream.h をインクルードします。C++ Version 1.2 の旧形式ストリームと互換性を保つため。 |

これらのヘッダーファイルすべてをプログラムにインクルードする必要はありません。自分のプログラムで必要な宣言の入ったものだけをインクルードします。デフォルトでは、従来型の libiostream ライブラリは iostream に含まれています。

### 13.3.1 iostream を使用した出力

iostream を使用した出力は、通常、左シフト演算子 (<<) を多重定義したもの (iostream の文脈では挿入演算子といいます) を使用します。ある値を標準出力に出力するには、その値を定義済みの出力ストリーム cout に挿入します。たとえば someValue を出力するには、次の文を標準出力に挿入します。

```
cout << someValue;
```

挿入演算子は、すべての組み込み型について多重定義されており、`someValue` の値は適当な出力形式に変換されます。たとえば `someValue` が `float` 型の場合、`<<` 演算子はその値を数字と小数点の組み合わせに変換します。`float` 型の値を出力ストリームに挿入するときは、`<<` を float 型挿入子といいます。一般に X 型の値を出力ストリームに挿入するときは、`<<` を X 型挿入子といいます。出力形式とその制御方法については、`ios`(3CC4) のマニュアルページを参照してください。

`iostream` ライブラリはユーザー定義型をサポートしません。出力する型を独自の方法で定義する場合、型を正しく処理するための挿入子を定義する (つまり、`<<` 演算子を多重定義する) 必要があります。

`<<` 演算子は反復使用できます。2 つの値を `cout` に挿入するには、次の例のような文を使用できます。

```
cout << someValue << anotherValue;
```

前述の例では、2 つの値の間に空白が入りません。空白を入れる場合は、次のようにします。

```
cout << someValue << " " << anotherValue;
```

`<<` 演算子は、組み込みの左シフト演算子と同じ優先順位を持ちます。ほかの演算子と同様に、括弧を使用して実行順序を指定できます。 必要に応じて、優先順位の問題を回避するために括弧を使用します。次の 4 つの文のうち、最初の 2 つは同じ結果になりますが、あとの 2 つは異なります。

```
cout << a+b;              // + has higher precedence than <<
cout << (a+b);
cout << (a&y);            // << has precedence higher than &
cout << a&y;            // probably an error: (cout << a) & y
```

### 13.3.1.1 ユーザー定義の挿入演算子

次のコーディング例では `string` クラスを定義しています。

```
#include <stdlib.h>
#include <iostream.h>


class string {
private:
    char* data;
    size_t size;

public:
    // (functions not relevant here)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

この例では、string クラスのデータ部が private であるため、挿入演算子と抽出演算子をフレンド定義しておく必要があります。

```
ostream& operator<< (ostream& ostr, const string& output)
{    return ostr << output.data;}
```

次の例は、string とともに使用される多重定義された operator<< の定義です。

```
cout << string1 << string2;
```

operator<< は、最初の引数として ostream& (ostream への参照) を受け取り、同じ ostream を返します。このため、次のように 1 つの文で挿入演算子を続けて使用できます。

### 13.3.1.2 出力エラーの処理

operator<< を多重定義するときは、iostream ライブラリからエラーが通知されることになるため、特にエラー検査を行う必要はありません。

エラーが起こると、エラーの起こった iostream はエラー状態になります。その iostream の状態の各ビットが、エラーの大きな分類に従ってセットされます。iostream で定義された挿入子がストリームにデータを挿入しようとしても、そのストリームがエラー状態の場合はデータが挿入されず、iostream の状態も変わりません。

一般的なエラー処理方法は、メインのどこかで定期的に出力ストリームの状態を検査する方法です。エラーが存在する場合は、何らかの方法でエラーを処理するようにしてください。この章では、文字列を取得してプログラムを異常終了する関数 error を定義したと仮定します。error は事前定義関数ではありません。error 関数の例については、153 ページの「13.3.9 入力エラーの処理」を参照してください。iostream の状態は、演算子 !, で確認でき、これは、iostream がエラー状態の場合にゼロ以外の値を返します。例:

```
if (!cout) error("output error");
```

エラーを調べるにはもう 1 つの方法があります。ios クラスでは、operator void *() が定義されており、エラーが起こった場合は NULL ポインタを返します。次の例のような文を使用できます。

```
if (cout << x) return; // return if successful
```

また、次のように ios クラスのメンバー関数 good を使用することもできます。

```
if (cout.good()) return; // return if successful
```

エラービットは次のような列挙型で宣言されています。

```
enum io_state {goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80};
```

エラー関数の詳細については、iostream のマニュアルページを参照してください。

#### 13.3.1.3 出力のフラッシュ

多くの入出力ライブラリと同様、iostream も出力データを蓄積し、より大きなブロックにまとめて効率よく出力します。出力バッファーをフラッシュする場合、次のように特殊な値 flush を挿入するだけでフラッシュできます。例:

```
cout << "This needs to get out immediately." << flush;
```

flush は、マニピュレータと呼ばれるタイプのオブジェクトの 1 つです。マニピュレータを iostream に挿入すると、その値が出力されるのではなく、何らかの効果が引き起こされます。これらの値は実際には関数で、引数 ostream& または istream& 引数を取り、何らかの動作を実行したあとにその引数を返します (157 ページの「13.7 マニピュレータ」を参照してください)。

#### 13.3.1.4 バイナリ出力

ある値をバイナリ形式のままで出力するには、次の例のようにメンバー関数 write を使用します。次の例では、x の値がバイナリ形式のまま出力されます。

```
cout.write((char*)&x, sizeof(x));
```

この例では、&x を char* に変換しており、型変換の規則に反します。通常このようにしても問題はありませんが、x の型が、ポインタまたは仮想メンバー関数を持つクラスであるか、重要なコンストラクタ動作を要求するクラスの場合、前述の例で出力した値を正しく読み込むことができません。

### 13.3.2 iostream を使用した入力

iostream を使用した入力は、出力と同じです。入力には、抽出演算子 >> を使用します。挿入演算子と同様に繰り返し指定できます。例:

```
cin >> a >> b;
```

この例では、標準入力から 2 つの値が取り出されます。多重されたほかの演算子のように、使用される抽出子は a および b の型に依存します。a と b が異なる型を持つ場合、2 つの異なる抽出子が使用されます。入力データのフォーマットとその制御方法についての詳細は、ios(3CC4) のマニュアルページを参照してください。通常は、先頭の空白文字 (スペース、改行、タブ、フォームフィードなど) は無視されます。

## 13.3.3 ユーザー定義の抽出演算子

ユーザーが新たに定義した型のデータを入力するには、出力のために挿入演算子を多重定義したのと同様に、その型に対する抽出演算子を多重定義します。

クラス string の抽出演算子は次のコーディング例のように定義します。

例 13–1 string の抽出演算子

```
istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
    istr.get(holder, maxline, ”\n’);
    input = holder;
    return istr;
}
```

get 関数は、入力ストリーム istr から文字列を読み取ります。読み取られた文字列は、maxline-1 バイトの文字が読み込まれる、新しい行に達する、EOF に達する、のうちのいずれかが発生するまで、holder に格納されます。データ holder は NULL で終わります。最後に、holder 内の文字列がターゲットの文字列にコピーされます。

規則に従って、抽出子は第 1 引数 (前述の例では istream& istr) から取り出した文字列を変換し、常に参照引数である第 2 引数に格納し、第 1 引数を返します。抽出子とは、入力値を第 2 引数に格納するためのものなので、第 2 引数は必ず参照引数である必要があります。

## 13.3.4 char* の抽出子

この定義済み抽出子は問題が起こる可能性があるため、注意して使用してください。この抽出子は次のように使用します。

```
char x[50];
cin >> x;
```

この抽出子は先頭の空白を読み飛ばし、次の空白文字に到達するまで、文字を抽出してそれを x にコピーします。これは終端ヌル (0) 文字で文字列を完了します。入力によって特定の配列がオーバーフローする可能性があるため、この抽出子は注意して使用してください。

さらに、ポインタが、割り当てられた記憶領域を指していることを確認する必要があります。次の例は一般的なエラーを示しています。

```
char * p; // not initialized
cin >> p;
```

入力データが格納される場所が不明確なため、プログラムは異常終了することがあります。

### 13.3.5 1 文字の読み込み

char 型の抽出子を使用することに加えて、次に示すいずれかの形式でメンバー関数 get を使用することによって、1 文字を読み取ることができます。例:

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

注 – ほかの抽出子とは異なり、char 型の抽出子は行頭の空白を読み飛ばしません。

次の例は、空白だけを読み飛ばし、タブや改行などそのほかの文字で停止する方法を示しています。

```
int a;
do {
    a = cin.get();
   }
while(a ==' ');
```

### 13.3.6 バイナリ入力

メンバー関数 write で出力したようなバイナリの値を読み込むには、メンバー関数 read を使用します。次の例では、メンバー関数 read を使用して x のバイナリ形式の値をそのまま入力します。次の例は、先に示した関数 write を使用した例と反対のことを行います。

```
cin.read((char*)&x, sizeof(x));
```

### 13.3.7 入力データの先読み

メンバー関数 peek を使用するとストリームから次の文字を抽出することなく、その文字を知ることができます。例:

```
if (cin.peek()!= c) return 0;
```

### 13.3.8 空白の抽出

デフォルトでは、iostream 抽出子は先頭の空白を読み飛ばします。 次の例では、cin の空白の読み飛ばしをオフにし、あとでオンに戻します。

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
...
cin.setf(ios::skipws); // turn it on again
```

iostream のマニピュレータ ws を使用すると、読み飛ばしが現在有効かどうかに関係なく、iostream から先頭の空白を取り除くことができます。次の例では、iostream istr から先頭の空白が取り除かれます。

```
istr >> ws;
```

## 13.3.9 入力エラーの処理

通常は、第 1 引数が非ゼロのエラー状態にある場合、抽出子は入力ストリームからのデータの抽出とエラービットのクリアを行わないでください。データの抽出に失敗した場合、抽出子は最低 1 つのエラービットを設定します。

出力エラーの場合と同様、エラー状態を定期的に検査し、非ゼロの状態の場合は処理の中止など何らかの動作を起こす必要があります。 ! 演算子は iostream のエラー状態をテストします。たとえば次のコーディング例では、英字を入力すると入力エラーが発生します。

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
     cerr << message << "\n";
     exit(1);
}
int main() {
     cout << "Enter some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
     return 0;
}
```

クラス ios には、エラー処理に使用できるメンバー関数があります。詳細はマニュアルページを参照してください。

## 13.3.10 iostream と stdio の併用

C++ プログラムでも stdio を使用できますが、プログラムで iostream と stdio とを標準ストリームとして併用すると、問題が起こる場合があります。たとえば stdout と cout の両方に書き込んだ場合、個別にバッファリングされるため出力結果が設計したとおりにならないことがあります。stdin と cin の両方から入力した場合、問題はさらに深刻です。個別にバッファリングされるため、入力データが使用できなくなっている可能性があります。

標準入力、標準出力、標準エラーに関するこのような問題を解決するためには、入出力を実行する前に次の命令を使用します。次の命令で、すべての定義済み iostream が、それぞれ対応する定義済み stdio FILE に結合されます。

```
ios::sync_with_stdio();
```

この種類の結合は、定義済みストリームが結合されたものの一部となってバッファリングされなくなってかなり効率が悪くなるため、デフォルトではありません。stdio および iostream の両方を、別のファイルに適用される同じプログラム内で使用できます。すなわち、stdio ルーチンを使用して stdout に書き込み、iostream に結合した別のファイルに書き込むことは可能です。また stdio FILE を入力用にオープンしても、stdin から入力しないかぎりは cin からも読み込むことができます。

# 13.4 iostream の作成

定義済みの iostream 以外のストリームを読み込む、あるいは書き込む場合は、ユーザーが自分で iostream を生成する必要があります。これは一般には、iostream ライブラリで定義されている型のオブジェクトを生成することになります。ここでは、使用できるさまざまな型について説明します。

## 13.4.1 クラス fstream を使用したファイル操作

ファイル操作は標準入出力の操作に似ています。ifstream、ofstrea m、fstream の 3 つのクラスはそれぞれ、istream、ostream、iostream の各クラスから派生しています。この 3 つのクラスは派生クラスなので、挿入演算と抽出演算、および、そのほかのメンバー関数を継承しており、ファイル使用のためのメンバーとコンストラクタも持っています。

fstream のいずれかを使用するときは、fstream.h をインクルードします。入力だけ行うときは ifstream、出力だけ行うときは ofstream、入出力を行うときは fstream を使用します。コンストラクタへの引数としてはファイル名を渡します。

thisFile というファイルから thatFile というファイルへのファイルコピーを行うときは、次のコーディング例のようになります。

```
ifstream fromFile("thisFile");
if     (!fromFile)
    error("unable to open 'thisFile' for input");
ofstream toFile ("thatFile");
if     (!toFile)
    error("unable to open 'thatFile' for output");
char c;
while (toFile && fromFile.get(c)) toFile.put(c);
```

このコードは次を実行します。

- fromFile という ifstream オブジェクトをデフォルトモード ios::in で生成し、それを thisFile に結合します。thisFile をオープンします。
- 新しい ifstream オブジェクトのエラー状態を調べ、エラーであれば関数 error を呼び出します。関数 error は、プログラムの別の場所で定義されている必要があります。
- toFile という ofstream オブジェクトをデフォルトモード ios::out で生成し、それを thatFile に結合します。
- 上記のように、toFile のエラー状態を検査します。
- データの受け渡しに使用する char 型変数を生成します。
- fromFile の内容を一度に 1 文字ずつ toFile にコピーします。

---

注 – ファイルをこのように、一度に 1 文字ずつコピーすることは望ましくありません。このコードは fstream の使用例として示したにすぎません。実際には、入力ストリームに関係付けられた streambuf を出力ストリームに挿入するのが一般的です。161 ページの「13.10 streambuf ストリームの操作」と、sbufpub(3CC4) のマニュアルページを参照してください。

---

### 13.4.1.1 オープンモード

このモードは、列挙型 open_mode の各ビットの or で構築されます。これは、ios クラスの公開部であり、次の定義を持ちます。

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

---

注 – UNIX では binary フラグは必要ありませんが、これを必要とするシステムとの互換性を保つために提供されています。移植可能なコードにするためには、バイナリファイルをオープンするときに binary フラグを使用する必要があります。

---

入出力両用のファイルをオープンできます。たとえば次のコードでは、someName という入出力ファイルをオープンして、fstream 変数 inoutFile に結合します。

```
fstream inoutFile("someName", ios::in|ios::out);
```

### 13.4.1.2 ファイルを指定しない fstream の宣言

ファイルを指定せずに fstream の宣言だけを行い、のちにファイルをオープンすることもできます。次の例では出力用の ofstream toFile を作成します。

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

### 13.4.1.3 ファイルのオープンとクローズ

fstream をいったんクローズし、また別のファイルでオープンすることができます。たとえば、コマンド行で与えられるファイルリストを処理するには次のようにします。

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

### 13.4.1.4 ファイル記述子を使用したファイルのオープン

標準出力の整数 1 などのようにファイル記述子がわかっている場合は、次のようにファイルをオープンできます。

```
ofstream outfile;
outfile.attach(1);
```

fstream コンストラクタの 1 つにファイル名を指定してファイルをオープンしたり、open 関数を使用してオープンしたファイルは、fstream が delete によって破壊されるか、スコープ外に出る時点で自動的にクローズされます。attach で fstream に結合したファイルは、自動的にはクローズされません。

### 13.4.1.5 ファイル内の位置の再設定

ファイル内の読み込み位置と書き込み位置を変更することができます。そのためには次のようなツールがあります。

- streampos は、iostream 内の位置を記憶しておくためのデータ型です。
- tellg (tellp) は istream (ostream) のメンバー関数で、現在のファイル内の位置を返します。istream と ostream は fstream の親クラスであるため、tellg と tellp も fstream クラスのメンバー関数として呼び出すことができます。
- seekg (seekp) は istream (ostream) のメンバー関数で、指定したファイル内の位置を探し出します。
- enum seek_dir は、seek での相対位置を指定します。

  ```
  enum seek_dir {beg=0, cur=1, end=2};
  ```

  fstream aFile の位置再設定の例を次に示します。

  ```
  streampos original = aFile.tellp();     //save current position
  aFile.seekp(0, ios::end); //reposition to end of file
  aFile << x;               //write a value to file
  aFile.seekp(original);    //return to original position
  ```

  seekg (seekp) は、1 つまたは 2 つの引数を受け取ります。引数を 2 つ受け取るときは、第 1 引数は、第 2 引数で指定した seek_dir 値が示す位置からの相対位置となります。例:

```
aFile.seekp(-10, ios::end);
```

この例では、ファイルの最後から 10 バイトの位置に設定されます。

```
aFile.seekp(10, ios::cur);
```

一方、次の例では現在位置から 10 バイト進められます。

---

注 – テキストストリーム上での任意位置へのシーク動作はマシン依存になります。ただし、以前に保存した streampos の値にいつでも戻ることができます。

---

## 13.5 iostream の代入

iostream では、あるストリームを別のストリームに代入することはできません。

ストリームオブジェクトのコピーでは、出力ファイル内の現在の書き込み位置ポインタなどの状態情報に 2 つのバージョンが存在するようになり、それを個別に変更できることが問題になります。その結果、問題が発生する可能性があります。

## 13.6 フォーマットの制御

フォーマットの制御については、ios(3CC4) のマニュアルページで詳しく説明しています。

## 13.7 マニピュレータ

マニピュレータとは、iostream に挿入したり、iostream から抽出したりする値で特別な効果があります。

引数付きマニピュレータとは、1 つ以上の追加の引数を持つマニピュレータのことです。

マニピュレータは通常の識別子であるため、マニピュレータの定義を多く行うと可能な名前を使いきってしまうので、iostream では考えられるすべての機能に対して定義されているわけではありません。マニピュレータの多くは、この章の別の箇所でメンバー関数とともに説明しています。

定義済みの 13 個のマニピュレータを次の表に示します。この表は、次のことを想定しています。

- i は long 型です。
- n は int 型です。
- c は char 型です。

- istr は入力ストリームです。
- ostr は入力ストリームです。

表 13–2 iostream の定義済みマニピュレータ

| | 定義済みマニピュレータ | 説明 |
|---|---|---|
| 1 | ostr << dec, istr >> dec | 10 を基数とする整数変換を行います。 |
| 2 | ostr << endl | 改行文字 ('\n') を挿入し、ostream::flush() を呼び出します。 |
| 3 | ostr << ends | null (0) 文字を挿入します。strstream 取引時に役に立ちます。 |
| 4 | ostr << flush | ostream::flush() を呼び出します。 |
| 5 | ostr << hex, istr >> hex | 16 を基数とする整数変換を行います。 |
| 6 | ostr << oct, istr >> oct | 8 を基数とする整数変換を行います。 |
| 7 | istr >> ws | 最初に空白以外の文字が見つかるまで (この文字以降は istr に残る)、空白を取り除きます (空白を読み飛ばす)。 |
| 8 | ostr << setbase(*n*), istr >> setbase(*n*) | 変換の基数を n (0、8、10、16 のみ) に設定します。 |
| 9 | ostr << setw(*n*), istr >> setw( *n*) | ios::width(n) を呼び出します。フィールド幅を n に設定します。 |
| 10 | ostr << resetiosflags(*i*), istr>> resetiosflags(*i*) | i のビットセットに従って、フラグのビットベクトルをクリアします。 |
| 11 | ostr << setiosflags(i), istr >> setiosflags(i) | i のビットセットに従って、フラグのビットベクトルを設定します。 |
| 12 | ostr << setfill(c), istr >> setfill(c) | 詰め合わせる文字 (フィールドのパディング用) を c に設定します。 |
| 13 | ostr << setprecision(*n*), istr >> setprecision(*n*) | 浮動小数点の精度を n 桁に設定します。 |

定義済みマニピュレータを使用するには、プログラムにヘッダーファイル iomanip.h をインクルードする必要があります。

ユーザーが独自のマニピュレータを定義することもできます。マニピュレータには次の 2 つの基本タイプがあります。

- 引数なしのマニピュレータ – istream&、ostream&、ios& のいずれかを引数として取り、ストリームで処理を行い、その引数を返します。

- 引数付きのマニピュレータ – istream&、ostream&、ios& のいずれかと、そのほかもう 1 つの引数 (追加の引数) を取り、ストリームで処理を行い、ストリーム引数を返します。

## 13.7.1 引数なしのマニピュレータの使用法

引数なしのマニピュレータは、次の動作を実行する関数です。

- ストリームの参照引数を受け取ります。
- そのストリームに何らかの処理を行います。
- その引数を返します。

iostream では、このような関数へのポインタを使用するシフト演算子がすでに定義されているので、関数を入出力演算子のシーケンスの中に入れることができます。シフト演算子は、値の入出力を行う代わりに、その関数を呼び出します。tab を ostream に挿入する tab マニピュレータの例を次に示します。

```
ostream& tab(ostream& os) {
             return os <<'\t';
           }
...
cout << x << tab << y;
```

この例は、次のコードを得るための複雑な方法です。

```
const char tab = '\t';
...
cout << x << tab << y;
```

次に示すのは別の例で、定数を使用してこれと同じことを簡単に実行することはできません。入力ストリームに対して、空白の読み飛ばしのオン、オフを設定すると仮定します。ios::setf と ios::unsetf を別々に呼び出して、skipws フラグをオンまたはオフに設定することもできますが、次の例のように 2 つのマニピュレータを定義して設定することもできます。

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
       is.setf(ios::skipws, ios::skipws);
       return is;
}
istream& skipoff(istream& is) {
       is.unsetf(ios::skipws);
       return is;
}
...
int main ()
{
      int x,y;
      cin >> skipon >> x >> skipoff >> y;
      return 1;
}
```

## 13.7.2 引数付きのマニピュレータ

iomanip.h に入っているマニピュレータの 1 つに setfill があります。setfill は、フィールド幅に詰め合わせる文字を設定するマニピュレータで、このマニピュレータは次の例に示すように実装されています。

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

引数付きマニピュレータは、2 つの部分から構成されます。

- マニピュレータ。これは引数を 1 つ追加します。この前の例では、int 型の第 2 引数があります。このような関数に対するシフト演算子は定義されていないので、このマニピュレータ関数を入出力演算子のシーケンスに入れることはできません。そこで、マニピュレータの代わりに補助関数 (適用子) を使用する必要があります。
- 適用子。これはマニピュレータを呼び出します。適用子は大域関数で、そのプロトタイプをヘッダーファイルに入れておきます。マニピュレータは通常、適用子の入っているソースコードファイル内に静的関数として作成します。マニピュレータは適用子によってのみ呼び出されます。これを静的にした場合、その名前は大域アドレス空間の外側に保持されます。

ヘッダーファイル iomanip.h には、さまざまなクラスが定義されています。各クラスには、マニピュレータ関数のアドレスと 1 つの引数の値が入っています。iomanip クラスについては manip (3CC4) マニュアルページで説明されています。前の例では smanip_int クラスが使用され、これは ios で使用できます。ios で使用できるということは、istream と ostream でも使用できるということです。この例ではまた、int 型の第 2 引数を使用しています。

適用子は、クラスオブジェクトを作成してそれを返します。この前の例では、smanip_int というクラスオブジェクトが作成され、そこにマニピュレータと、適用子の int 型引数が入っています。ヘッダーファイル iomanip.h では、このクラスに対するシフト演算子が定義されています。入出力演算子シーケンスの中に適用子関数 setfill があると、その適用子関数が呼び出され、クラスが返されます。シフト演算子はそのクラスに対して働き、クラス内に入っている引数値を使用してマニピュレータ関数が呼び出されます。

次の例では、マニピュレータ print_hex は次の動作を行います。

- 出力ストリームを 16 進モードにする
- long 型の値 をストリームに挿入する
- ストリームの変換モードを元に戻す

このコード例は出力のみに使用されるため、クラス omanip_long が使用されます。これは int でなく long で動作します。

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

## 13.8 strstream: 配列用の iostream

strstream(3CC4) のマニュアルページを参照してください。

## 13.9 stdiobuf: stdio ファイル用の iostream

stdiobuf(3CC4) のマニュアルページを参照してください。

## 13.10 streambuf ストリームの操作

入力や出力のシステムは、フォーマットを行う iostream と、システムのほかの部分は streambuf ストリームからなります。これはフォーマットなしの文字ストリームの入力または出力で動作します。

streambuf ストリームは通常、iostream 経由で使用するため、詳細に知っておく必要はありません。 streambuf ストリームを直接使用するように選択できます。たとえば、効率を高める場合や、iostream に組み込まれたエラー処理または整形を回避することが必要な場合があります。

### 13.10.1 streambuf ポインタ型

streambuf は文字シーケンス (文字ストリーム) と、シーケンス内を指す 1 つまたは 2 つのポインタとで構成されています。各ポインタは文字と文字の間を指しています。(ポインタは実際には文字と文字の間を指しているわけではありませんが、このように考えると理解しやすくなります。)streambuf ポインタには次の種類があります。

- *put* ポインタ – 次の文字が格納される直前を指します。
- *get* ポインタ - 取得される次の文字の直前を指します。

streambuf は、このどちらかのポインタ、または両方のポインタを持ちます。

ポインタ位置の操作とシーケンスの内容の操作にはさまざまな方法があります。操作時に両方のポインタが移動するかどうかは、使用される streambuf の種類によって違います。一般的に、キュー形式の streambuf ストリームの場合、get ポインタと put ポインタは別々に移動します。ファイル形式の streambuf ストリームの場合、get ポインタと put ポインタは同時に移動します。キュー形式ストリームの例としては strstream があり、ファイル形式ストリームの例としては fstream があります。

## 13.10.2 streambuf オブジェクトの使用

ユーザーは streambuf オブジェクト自体を作成することはなく、streambuf クラスから派生したクラスのオブジェクトを作成します。例としては、filebuf と strstreambuf があります。これらについては filebuf(3CC4) と ssbuf(3) のマニュアルページを参照してください。より高度な使い方として、独自のクラスを streambuf から派生させて特殊デバイスのインタフェースを提供したり、基本的なバッファリング以外のバッファリングを行なったりすることができます。sbufpub(3CC4) と sbufprot のマニュアルページ (3CC4) では、これを行う方法について説明します。

独自の特殊な streambuf を作成する以外にも、マニュアルページで説明しているように、iostream と結合した streambuf にアクセスして公開メンバー関数にアクセスする場合があります。また、各 iostream には、streambuf へのポインタを引数とする定義済みの挿入子と抽出子があります。streambuf を挿入したり抽出したりすると、ストリーム全体がコピーされます。

次の例では、以前説明したファイルコピーを実行する別の方法を示しています。簡潔にするためにエラーチェックは省略されています。

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

入力ファイルと出力ファイルは、前と同じようにオープンします。各 iostream クラスにはメンバー関数 rdbuf があり、それに結合した streambuf オブジェクトへのポインタを返します。fstream の場合、streambuf オブジェクトは filebuf 型です。fromFile に結合したファイル全体が toFile に結合したファイルにコピー (挿入) されます。最後の行は次のように書くこともできます。

```
fromFile >> toFile.rdbuf();
```

前述の書き方では、ソースファイルが抽出されて目的のところに入ります。どちらの書き方をしても、結果はまったく同じになります。

# 13.11 iostream に関するマニュアルページ

C++ では、iostream ライブラリの詳細を説明する多くのマニュアルページがあります。次に、各マニュアルページの概要を示します。

従来型の iostream ライブラリのマニュアルページを表示するには、次のように入力します (name には、マニュアルページのトピック名を入力)。

```
example% man -s 3CC4 name
```

表 13–3 iostream に関するマニュアルページの概要

| マニュアルページ | 概要 |
|---|---|
| filebuf | streambuf から派生し、ファイル処理のために特殊化された filebuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承した機能の詳細については、sbufpub(3CC4) と sbufprot(3CC4) のマニュアルページを参照してください。filebuf クラスは、fstream クラスを通して使用します。 |
| fstream | istream、ostream、iostream をファイル処理用に特殊化した ifstream、ofstream、fstream の各クラスの特殊化されたメンバー関数を詳細に説明します。 |
| ios | iostream の基底クラスである ios クラスの各部を詳細に説明します。すべてのストリームに共通の状態データについても説明します。 |
| ios.intro | iostream を紹介し、概要を説明します。 |
| istream | 次の各項目を詳細に説明します。<br>■ streambuf から取り出した文字の解釈をサポートする、クラス istream のメンバー関数<br>■ 入力の書式設定<br>■ ostream の一部として記述されている位置決め関数<br>■ 一部の関連関数<br>■ 関連マニピュレータ |
| manip | iostream ライブラリで定義されている入出力マニピュレータを説明します。 |
| ostream | 次の各項目を詳細に説明します。<br>■ streambuf から取り出した文字の解釈をサポートする、クラス ostream のメンバー関数<br>■ 出力の書式設定<br>■ ostream の一部として記述されている位置決め関数<br>■ 一部の関連関数<br>■ 関連マニピュレータ |

表 13–3 iostream に関するマニュアルページの概要 (続き)

| マニュアルページ | 概要 |
|---|---|
| sbufprot | streambuf(3CC4) クラスから派生したクラスをコーディングするプログラマに必要なインタフェースを説明します。sbufprot(3CC4) のマニュアルページで説明されていない公開関数があるため、sbufpub(3CC4)のマニュアルページも参照してください。 |
| sbufpub | streambuf クラスの公開インタフェース、特に streambuf の公開メンバー関数について詳細に説明します。このマニュアルページには、streambuf 型のオブジェクトを直接操作したり、streambuf から派生したクラスが継承している関数を探し出したりするのに必要な情報が含まれています。streambuft からクラスを派生する場合は、sbufprot(3CC4) のマニュアルページも参照してください。 |
| ssbuf | streambuf から派生し、文字型配列処理用に特殊化された strstreambuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承する機能の詳細については、sbufpub(3CC4) のマニュアルページを参照してください。 |
| stdiobuf | streambuf から派生し、stdio FILE の処理用に特殊化されたクラス stdiobuf の最低限の説明が含まれています。streambuf クラスから継承する機能の詳細については、sbufpub(3CC4) のマニュアルページを参照してください。 |
| strstream | strstream の特殊化されたメンバー関数を詳細に説明します。これらの関数は、iostream クラスから派生した一連のクラスで実装され、文字型配列処理用に特殊化されています。 |

## 13.12 iostream の用語

iostream ライブラリの説明では、一般のプログラミングの用語と同じですが、特別な意味を持つ用語がよく使われます。次の表では、それらの用語が iostream ライブラリの説明で使用される場合の意味を定義します。

表 13–4　iostream の用語

| iostream 用語 | 定義 |
|---|---|
| バッファー | バッファーには、2 つの意味があります。1 つは iostream パッケージに固有のバッファーで、もう 1 つは入出力一般に適用されるバッファーです。<br><br>iostream ライブラリに固有のバッファーは、streambuf クラスで定義されたオブジェクトです。<br><br>一般にいうバッファーは、入出力データを効率よく転送するために使用するメモリーブロックを指します。バッファリングされた入出力の場合は、バッファーがいっぱいになるか、バッファーが強制的にフラッシュされるときまで、データの転送は行われません。<br><br>「バッファリングなしのバッファー」とは、前述で定義したように一般にいうバッファーがない streambuf を指します。この章では、streambuf を指すのにバッファーという用語を使用することを避けています。ただし、マニュアルページやほかの C++ のドキュメントでは、streambuf を意味するためにバッファーという用語を使用しています。 |
| 抽出 | iostream から入力データを取り出す操作を抽出といいます。 |
| Fstream | ファイル用に特殊化された入出力ストリームです。monospace フォントで印刷されている場合は、特に iostream クラスから派生したクラスを指します。 |
| 挿入 | iostream に出力データを送り込む操作を挿入といいます。 |
| iostream | 一般には、入力ストリームまたは出力ストリームです。 |
| iostream ライブラリ | インクルードファイル iostream.h、fstream.h、strstream.h、iomanip.h、および stdiostream.h によって実装されるライブラリを指します。iostream はオブジェクト指向のライブラリであるため、ユーザーが必要に応じて拡張できます。 |
| ストリーム | 一般に、iostream、fstream、strstream、またはユーザー定義のストリームをいいます。 |
| streambuf | 文字シーケンスの入ったバッファーで、put ポインタまたは get ポインタ、あるいはその両方を持ちます。monospace フォントで印刷されている場合は、特定のクラスを意味します。そのほかのフォントで印刷されている場合は一般に streambuf クラスのオブジェクト、または streambuf の派生クラスを意味します。ストリームオブジェクトは必ず、streambuf から派生した型のオブジェクト (またはそのオブジェクトへのポインタ) を持っています。 |

表 13–4　iostream の用語　　*(続き)*

| **iostream** 用語 | 定義 |
|---|---|
| `strstream` | 文字型配列処理用に特殊化した `iostream` です。`monospace` フォントで印刷されている場合は、特定のクラスを意味します。 |

◆ ◆ ◆ 第 14 章

# 14 ライブラリの構築

この章では、ライブラリの構築方法を説明します。

## 14.1 ライブラリとは

ライブラリには 2 つの利点があります。まず、ライブラリを使えば、コードをいくつかのアプリケーションで共有できます。共有するコードがある場合は、そのコードを含むライブラリを作成し、コードを必要とするアプリケーションとリンクできます。次に、ライブラリを使えば、非常に大きなアプリケーションの複雑さを軽減できます。アプリケーションの中の、比較的独立した部分をライブラリとして構築および保守することで、プログラマはほかの部分の作業により専念できるようになるためです。

ライブラリの構築とは、.o ファイルを作成し (コードを -c オプションでコンパイルし)、これらの .o ファイルを CC コマンドでライブラリに結合することです。ライブラリには、静的 (アーカイブ) ライブラリと動的 (共有) ライブラリがあります。

静的 (アーカイブ) ライブラリの場合は、のオブジェクトがリンク時にプログラムの実行可能ファイルにリンクされます。アプリケーションにとって必要な .o ファイルだけがライブラリから実行可能ファイルにリンクされます。静的 (アーカイブ) ライブラリの名前には、一般的に接尾辞として .a が付きます。

動的 (共有) ライブラリの場合は、 ライブラリのオブジェクトはプログラムの実行可能ファイルにリンクされません。その代わりに、プログラムがこのライブラリに依存することをリンカーが実行可能ファイルに記録します。プログラムが実行されるとき、システムは、プログラムに必要な動的ライブラリを読み込みます。同じ動的ライブラリを使用する 2 つのプログラムが同時に実行されると、ライブラリはこれらのプログラムによって共有されます。動的 (共有) ライブラリの名前には、接尾辞として .so が付きます。

共有ライブラリを動的にリンクすることは、アーカイブライブラリを静的にリンクすることに比べていくつかの利点があります。

- 実行可能ファイルのサイズが小さくなる
- 実行時にコードのかなりの部分をプログラム間で共有できるため、メモリーの使用量が少なくなる
- アプリケーションをリンクし直さずに、実行時にライブラリを置き換えることができる(これは、プログラムの再リンクおよび再配布の必要なしに、Oracle Solaris オペレーティングシステムの多くの改良をプログラムで利用できる主要機構である。)
- `dlopen()` 関数呼び出し を使えば、共有ライブラリを実行時に読み込むことができる。

ただし、動的ライブラリには短所もあります。

- 実行時のリンクに時間がかかる
- 動的ライブラリを使用するプログラムを配布する場合には、それらのライブラリも同時に配布しなければならないことがある
- 共有ライブラリを異なる場所に移動すると、システムがライブラリを見つけられず、プログラムを実行できないことがある(この問題は、環境変数 `LD_LIBRARY_PATH` で解決できる)。

# 14.2 静的(アーカイブ)ライブラリの構築

静的 (アーカイブ) ライブラリを構築する仕組みは、実行可能ファイルを構築することに似ています。一連のオブジェクト (`.o`) ファイルは、`CC` で `-xar` オプションを使うことで 1 つのライブラリに結合できます。

静的 (アーカイブ) ライブラリを構築する場合は、`ar` コマンドを直接使用せずに `CC -xar` を使用してください。C++ 言語では一般に、従来の `.o` ファイルに収容できる情報より多くの情報 (特に、テンプレートインスタンス) をコンパイラが持たなければなりません。`-xar` オプションを使用すると、テンプレートインスタンスを含め、すべての必要な情報がライブラリに組み込まれます。`make` ではどのテンプレートファイルが実際に作成され、参照されているのかが判別できないことがあるため、通常のプログラミング環境でこのようにすることは困難です。`CC -xar` を指定しないと、参照に必要なテンプレートインスタンスがライブラリに組み込まれないことがあります。例:

```
% CC -c foo.cc # Compile main file, templates objects are created.
% CC -xar -o foo.a foo.o # Gather all objects into a library.
```

`-xar` フラグによって、`CC` が静的 (アーカイブ) ライブラリを作成します。`-o` 命令は、新しく作成するライブラリの名前を指定するために必要です。コンパイラは、コマンド行のオブジェクトファイルを調べ、これらのオブジェクトファイルと、テンプレートリポジトリで認識されているオブジェクトファイルとを相互参照します。そして、ユーザーのオブジェクトファイルに必要なテンプレートを (本体のオブジェクトファイルとともに) アーカイブに追加します。

注 – 既存のアーカイブのみを作成または更新するには、-xar フラグを使用します。このフラグをアーカイブの保守に使用しないでください。-xar オプションは ar -cr を実行するのと同じことです。

各 .o ファイルに 1 つの関数だけを入れます。アーカイブとリンクする場合、特定の .o ファイルのシンボルが必要になると、.o ファイル全体がアーカイブからアプリケーションにリンクされます。.o ファイルに 1 つの関数を入れておけば、アプリケーションにとって必要なシンボルだけがアーカイブからリンクされます。

## 14.3 動的 (共有) ライブラリの構築

動的 (共有) ライブラリの構築方法は、コマンド行に -xar の代わりに -G を指定することを除けば、静的 (アーカイブ) ライブラリの場合と同じです。

ld は直接使用しないでください。静的ライブラリの場合と同じように、CC コマンドを使用すると、必要なすべてのテンプレートインスタンスがテンプレートリポジトリからライブラリに組み込まれます (テンプレートを使用している場合)。アプリケーションにリンクされている動的ライブラリでは、すべての静的コンストラクタは main() が実行される前に呼び出され、すべての静的デストラクタは main() が終了したあとに呼び出されます。dlopen() で共有ライブラリを開いた場合、すべての静的コンストラクタは dlopen() で実行され、すべての静的デストラクタは dlclose() で実行されます。

動的ライブラリを構築するには、必ず CC に -G を使用します。ld (リンクエディタ) または cc (C コンパイラ) を使用して動的ライブラリを構築すると、例外が機能しない場合があり、ライブラリに定義されている大域変数が初期化されません。

動的 (共有) ライブラリを構築するには、 CC の -Kpic や -KPIC オプションで各オブジェクトをコンパイルして、再配置可能なオブジェクトファイルを作成する必要があります。次に、これらの再配置可能オブジェクトファイルから動的ライブラリを構築します。予期しないリンクエラーが出る場合は、-Kpic や -KPIC でコンパイルしていないオブジェクトがある可能性があります。

ソースファイル lsrc1.cc と lsrc2.cc から作成するオブジェクトファイルから C++ 動的ライブラリ libfoo.so を構築するには、次のようにします。

```
% CC -G -o libfoo.so -h libfoo.so -Kpic lsrc1.cc lsrc2.cc
```

-G オプションは、動的ライブラリの構築を指定しています。-o オプションは、ライブラリのファイル名を指定しています。-h オプションは、共有ライブラリの内部名を指定しています。-Kpic オプションは、オブジェクトファイルが位置に依存しないことを指定しています。

CC -G コマンドは -l オプションをリンカー ld に渡しません。初期化順序が必ず正しくなるようにするには、共有ライブラリに自らが必要とするほかの各共有ライブラリとの明示的な依存関係を設定する必要があります。依存関係を作成するには、該当するライブラリごとに -l オプションを使用します。標準的な C++ 共有ライブラリは、次の一群のオプションのうち 1 つを使用します。

```
-lCstd -lCrun -lc
-library=stlport4 -lCrun -lc
```

必要とされるすべての依存関係をリストしたことを確認するには、-zdefs オプションを指定してライブラリを構築します。不明のシンボル定義ごとに、リンカーはエラーメッセージを生成します。不明の定義を指定するには、それらのライブラリに -l オプションを追加します。

不要な依存関係を含んでいるかどうか検索するには、次のコマンドを使用します

```
ldd -u -r mylib.so
ldd -U -r mylib.so
```

その後、不要な依存関係を除外し、mylib.so を再構築できます。

## 14.4 例外を含む共有ライブラリの構築

C++ コードが含まれているプログラムでは、-Bsymbolic を使用しないでください。代わりにリンカーのマップファイルを使用してください。-Bsymbolic を使用すると、異なるモジュール内の参照が、本来 1 つの大域オブジェクトの複数の異なる複製に結合されてしまう可能性があります。

例外メカニズムは、アドレスの比較によって機能します。オブジェクトの複製が 2 つある場合は、アドレスが同一であると評価されず、本来一意のアドレスを比較することで機能する例外メカニズムで問題が発生することがあります。

## 14.5 非公開ライブラリの構築

ある組織の内部でしか使用しないライブラリを構築する場合には、一般的な使用には適さないオプションを使ってライブラリを構築することもできます。具体的には、ライブラリはシステムのアプリケーションバイナリインタフェース (ABI) に準拠していなくてもかまいません。たとえば、ライブラリを -fast オプションでコンパイルして、特定のアーキテクチャー上でのパフォーマンスを向上させることができます。同じように、-xregs=float オプションでコンパイルして、パフォーマンスを向上させることもできます。

## 14.6 公開ライブラリの構築

ほかの組織からも使用できるライブラリを構築する場合は、ライブラリの管理やプラットフォームの汎用性などの問題が重要になります。ライブラリを公開にするかどうかを決める簡単な基準は、アプリケーションのプログラマがライブラリを簡単に再コンパイルできるかどうかということです。公開ライブラリは、システムの ABI に準拠して構築しなければなりません。一般に、これはプロセッサ固有のオプションを使用しないということを意味します。たとえば、`-fast` や `-xtarget` は使用しないようにします。

SPARC ABI では、いくつかのレジスタがアプリケーション専用で使用されます。SPARC V7 と V8 では、これらのレジスタは `%g2`、`%g3`、`%g4` です。SPARC V9 では、これらのレジスタは `%g2` と `%g3` です。ほとんどのコンパイルはアプリケーション用に行われるので、C++ コンパイラは、デフォルトでこれらのレジスタを一時レジスタに使用して、プログラムのパフォーマンスを向上させようとします。しかし、公開ライブラリでこれらのレジスタを使用することは、SPARC ABI に適合しないことになります。公開ライブラリを構築するときには、アプリケーションレジスタを使用しないようにするために、すべてのオブジェクトを `-xregs=no%appl` オプションでコンパイルしてください。

## 14.7 C API を持つライブラリの構築

C++ で作成されたライブラリを C プログラムから使用できるようにするには、C API (アプリケーションプログラミングインタフェース) を作成する必要があります。そのためには、エクスポートされるすべての関数を `extern "C"` にします。ただし、これができるのは大域関数だけで、メンバー関数にはできません。

C インタフェースライブラリで C++ の実行時サポートを必要とし、しかも `cc` とリンクしている場合は、C インタフェースライブラリを使用するときにアプリケーションも `libCrun` (標準モード) にリンクする必要があります。(C インタフェースライブラリで C++ 実行時サポートが不要の場合は、`libCrun` とリンクする必要はありません。)リンク手順は、アーカイブされたライブラリと共有ライブラリでは異なります。

アーカイブされた C インタフェースライブラリを提供するときは、ライブラリの使用方法を説明する必要があります。

- C インタフェースライブラリが `CC` を標準モード (デフォルト) で構築している場合は、C インタフェースライブラリを使用するときに `-lCrun` を `cc` コマンド行に追加します。
- C インタフェースライブラリが `CC` を互換モード (`-compat=4`) で構築している場合は、C インタフェースライブラリを使用するときに `-lC` を `cc` コマンド行に追加します。

共有 C インタフェースライブラリを提供するときは、ライブラリの構築時に libCrun と依存関係を作る必要があります。共有ライブラリの依存関係が正しければ、ライブラリを使用するときに -lCrun をコマンドに追加する必要はありません。

- C インタフェースライブラリをデフォルトの標準モードで構築している場合は、ライブラリの構築時に -lCrun を CC コマンドに追加します。

さらに、C++ 実行時ライブラリにもまったく依存しないようにするには、ライブラリソースに対して次のコーディング規則を適用する必要があります。

- どのような形式の new もしくは delete も使用しない (独自の new または delete を定義する場合は除く)
- 例外を使用しない
- 実行時の型識別機構 (RunTime Type Information、RTTI) を使用しない

## 14.8 dlopen を使って C プログラムから C++ ライブラリにアクセスする

C プログラムから dlopen() を使用して C++ 共有ライブラリを開く場合は、共有ライブラリが適切な C++ 実行時ライブラリ (-compat=5 の場合は libCrun.so.1) に依存していることを確認してください。

そのためには、共有ライブラリの構築時に、 -compat=5 の場合は -lCrun をコマンド行に追加します。例:

```
example% CC -G -compat=5... -lCrun
```

共有ライブラリが例外を使用している場合には、ライブラリが C++ 共有ライブラリに依存していないと、C プログラムが正しく動作しないことがあります。

パート IV

# 付録

◆ ◆ ◆ 付 録 A

# C++ コンパイラオプション

この付録では、C++ コンパイラのコマンド行オプションを詳しく説明します。ここで説明する機能は、特に記載がないかぎりすべてのプラットフォームに適用されます。SPARC ベースシステム上の Oracle Solaris OS に特有の機能は *SPARC*、x86 ベースシステム上の Oracle Solaris OS に特有の機能は *x86* として識別されます。Oracle Solaris OS のみに限定されている機能には *Solaris* というマークが付きます。Linux OS のみに限定されている機能には *Linux* というマークが付きます。

この節では、個別のオプションを説明するために、このマニュアルの「はじめに」に記載した表記上の規則を使用しています。

括弧、中括弧、角括弧、パイプ文字、および省略符号は、オプションの説明で使用されているメタキャラクタです。これらは、オプションの一部ではありません。

## A.1 オプション情報の構成

簡単に情報を検索できるように、次の見出しに分けてコンパイラオプションを説明しています。オプションがほかのオプションで置き換えられたり、ほかのオプションと同じである場合、詳細についてはほかのオプション説明を参照してください。

表 A–1 オプションの見出し

| 見出し | 内容 |
|---|---|
| オプションの定義 | 各オプションのすぐあとには短い定義があります (小見出しはありません)。 |
| 値 | オプションに値がある場合は、その値を示します。 |

表 A–1 オプションの見出し (続き)

| 見出し | 内容 |
|---|---|
| デフォルト | オプションに一次または二次のデフォルト値がある場合は、それを示します。<br><br>一次のデフォルトとは、オプションが指定されなかったときに有効になるオプションの値です。たとえば、-compat を指定しないと、デフォルトは -compat=5 になります。<br><br>二次のデフォルトとは、オプションは指定されたが、値が指定されなかったときに有効になるオプションの値です。たとえば、値を指定せずに -compat を指定すると、デフォルトは -compat=5 になります。 |
| 展開 | オプションにマクロ展開がある場合は、ここに示します。 |
| 例 | オプションの説明のために例が必要な場合は、ここに示します。 |
| 相互の関連性 | ほかのオプションとの相互の関連性がある場合は、その関係をここに示します。 |
| 警告 | オプションの使用についての注意はここに示します。予測できない動作の原因となる操作についてもここに示します。 |
| 関連項目 | ここには、参考情報が得られるほかのオプションや文書を示します。 |
| 「置き換え」、「同じ」 | そのオプションが廃止され、ほかのもので置き換えられていたり、そのオプションの代わりに別のオプションを使用する方がよい場合は、置き換えるオプションを「置き換え」や「同じ」という表記とともに示しています。このような指示のあるオプションは、将来のリリースでサポートされない可能性があります。<br><br>2 つのオプションの一般的な意味と目的が同じである場合は、望ましいオプションをここに示します。たとえば、「-x0 と同じです」は、-x0 が望ましいオプションであることを示します。 |

# A.2 オプションの一覧

次の節では、C++ コンパイラオプションのアルファベット順の一覧と、プラットフォームの制限を示しています。

## A.2.1 -#

冗長モードを有効にし、コマンドオプションがどのように展開されるかを表示します。要素が呼び出されるごとにその要素を表示します。

## A.2.2 -###

呼び出される各構成要素が表示されますが、実行はされません。また、コマンドオプションの展開内容を表示します。

## A.2.3 -Bbinding

ライブラリのリンク形式を、シンボリックか、動的 (共有)、静的 (共有でない) のいずれかから指定します。

-B オプションは同じコマンド行で何回も指定できます。このオプションはリンカー (ld) に渡されます。

---

注 – Oracle Solaris の 64 ビットコンパイル環境では、多くのシステムライブラリは動的ライブラリとしてのみ使用できます。このため、コマンド行の最後に -Bstatic を使用しないでください。

---

### A.2.3.1 値

*binding* は、次の表に示されている値のいずれかである必要があります。

| 値 | 意味 |
|---|---|
| dynamic | まず lib*lib*.so (共有) ファイルを検索するようにリンカーに指示します。これらのファイルが見つからないと、リンカーは lib*lib*.a (静的で共有されない) ファイルを検索します。ライブラリのリンク方式を共有にしたい場合は、このオプションを指定します。 |
| static | -Bstatic オプションを指定すると、リンカーは lib*lib*.a (静的で、共有されない) ファイルだけを検索します。ライブラリのリンク形式を非共有にしたい場合は、このオプションを指定します。 |
| symbolic | シンボルがほかですでに定義されている場合でも、可能であれば共有ライブラリ内でシンボル解決を実行します。<br>ld(1) のマニュアルページを参照してください。 |

-B と *binding* との間に空白があってはいけません。

#### デフォルト

-B を指定しないと、 -Bdynamic が使用されます。

### 相互の関連性

C++ のデフォルトのライブラリを静的にリンクするには、`-staticlib` オプションを使用します。

`-Bstatic` および `-Bdynamic` オプションは、デフォルトで使用されるライブラリのリンクにも影響します。デフォルトのライブラリを動的にリンクするには、最後に指定する `-B` を `-Bdynamic` にするべきです。

64 ビットの環境では、多くのシステムライブラリは共有の動的ライブラリとしてのみ利用できます。これらのシステムライブラリには、`libm.so` および `libc.so` があります。`libm.a` と `libc.a` は提供していません。その結果、`-Bstatic` と `-dn` を使用すると 64 ビットの Oracle Solaris オペレーティングシステム環境でリンクエラーが生じる可能性があります。この場合、アプリケーションを動的ライブラリとリンクさせる必要があります。

### 例

次の例では、lib*foo*.so があっても lib*foo*.a がリンクされます。ほかのすべてのライブラリは動的にリンクされます。

```
example% CC a.o –Bstatic –lfoo –Bdynamic
```

### 警告

C++ コードが含まれているプログラムでは、`-Bsymbolic` を使用せずに、リンカーのマップファイルを使用してください。

`-Bsymbolic` を使用すると、異なるモジュール内の参照が、本来 1 つの大域オブジェクトの複数の異なる複製に結合されてしまう可能性があります。

例外メカニズムは、アドレスの比較によって機能します。オブジェクトの複製が 2 つある場合は、アドレスが同一であると評価されず、本来一意のアドレスを比較することで機能する例外メカニズムで問題が発生することがあります。

コンパイルとリンクを別々に行う場合で、コンパイル時に `-B`*binding* オプションを使用した場合は、このオプションをリンク時にも指定する必要があります。

### 関連項目

`-nolib`、`-staticlib`、`ld`(1) のマニュアルページ、132 ページの「11.5 標準ライブラリの静的リンク」、『リンカーとライブラリ』

## A.2.4 -c

コンパイルのみ。オブジェクト .o ファイルを作成しますが、リンクはしません。

このオプションは ld によるリンクを抑止し、各ソースファイルに対する .o ファイルを 1 つずつ生成するように、CC ドライバに指示します。コマンド行にソースファイルを 1 つだけ指定する場合には、-o オプションでそのオブジェクトファイルに明示的に名前を付けることができます。

### A.2.4.1 例

**CC -c x.cc** と入力すると、x.o というオブジェクトファイルが生成されます。

**CC -c x.cc -o y.o** と入力すると、y.o というオブジェクトファイルが生成されます。

#### 警告

コンパイラは、入力ファイル (.c、.i) に対するオブジェクトコードを作成する際に、.o ファイルを作業ディレクトリに作成します。リンク手順を省略すると、この .o ファイルは削除されません。

#### 関連項目

-o *filename*, -xe

## A.2.5 -cg{89|92}

(SPARC) 非推奨、使用しないでください。現在の Oracle Solaris オペレーティングシステムソフトウェアは、SPARC V7 アーキテクチャーをサポートしません。このオプションでコンパイルすると、現在の SPARC プラットフォームでの実行速度が遅いコードが生成されます。代わりに -x0 を使用し、-xarch、-xchip、および -xcache のコンパイラのデフォルトを利用します。

## A.2.6 -compat={ 5|g}

コンパイラの主要リリースとの互換モードを設定します。このオプションは、__SUNPRO_CC_COMPAT プリプロセッサマクロを制御します。

C++ コンパイラには主要なモードが 2 つあります。デフォルトの -compat=5 は 2003 年に更新された ANSI/ISO 1998 C++ 標準に従った構造を受け入れ、-compat=5 モードで C++ 5.0 ~ 5.12 と互換性のあるコードを生成します。-compat=g オプションにより、Oracle Solaris x86 および Linux プラットフォーム上の gcc/g++ コンパイラとのソースおよびバイナリ互換性が追加されます。名前の符号化、クラスの配置、vtable の配置、その他の ABI の細かい点に互換性のない、大きな変更が行われているため、これらのモードには互換性がありません。

以前のリリースの 4.2 コンパイラで定義された意味と言語を受け入れていた互換モード (-compat=4) は、使用できなくなりました。

次の節に示すように、これらのモードは -compat オプションで区別されます。

### A.2.6.1 値

-compat オプションには、次の表に示されている値を指定できます。

| 値 | 意味 |
|---|---|
| -compat=5 | (標準モード) 言語とバイナリの互換性を ANSI/ISO 標準モード 5.0 コンパイラに合わせます。__SUNPRO_CC_COMPAT プリプロセッサマクロを 5 に設定します。 |
| -compat=g | (x86 のみ) g++ 言語拡張の認識を有効にし、コンパイラで Solaris および Linux プラットフォーム上の g++ とバイナリ互換のあるコードが生成されるようにします。__SUNPRO_CC_COMPAT プリプロセッサマクロを 'G' に設定します。 |

-compat=g を使用すると、バイナリ互換性は個々の .o ファイルまたはアーカイブ (.a) ライブラリではなく、共有 (動的または .so) ライブラリにのみ拡張されます。

次の例は、g++ 共有ライブラリの C++ メインプログラムへのリンクを示しています。

```
% g++ -shared -o libfoo.so -fpic a.cc b.cc c.cc
% CC -compat=g main.cc -L. -lfoo
```

次の例は、C++ 共有ライブラリの g++ メインプログラムへのリンクを示しています。

```
% CC -compat=g -G -o libfoo.so -Kpic a.cc b.cc c.cc
% g++ main.cc -L. -lfoo
```

#### デフォルト

-compat オプションを指定しないと、-compat=5 が使用されます。

#### 相互の関連性

詳細は、-features を参照してください。

#### 警告

共有ライブラリを構築するときは、-Bsymbolic を使用しないでください。

## A.2.7 +d

C++ インライン関数を展開しません。

C++ 言語の規則では、C++ インライン関数は、次の文のいずれかがあてはまる場合の関数です。

- 関数が `inline` キーワードを使用して定義されている
- 関数が、宣言されているだけでなく、クラス定義内で定義されている
- 関数がコンパイラで生成されたクラスメンバー関数である

C++ 言語の規則では、呼び出しを実際にインライン化するかどうかをコンパイラが選択します。C++ コンパイラは、次のいずれかがあてはまらないかぎり、呼び出しをインライン関数にインライン化します。

- 関数が複雑すぎる
- `+d` オプションが選択されている
- `–xO`*n* 最適化レベルが指定されずに `–g` オプションが選択されている

### A.2.7.1 例

デフォルトでは、コンパイラは次のコード例で関数 `f()` と `memf2()` をインライン化できます。また、クラスには、コンパイラによって生成されたデフォルトのコンストラクタとコンパイラでインライン化できるデストラクタがあります。`+d` を使用すると、コンパイラでコンストラクタ `f()` とデストラクタ `C::mf2()` はインライン化されません。

```
inline int f() {return 0;} // may be inlined
class C {
  int mf1(); // not inlined unless inline definition comes later
  int mf2() {return 0;} // may be inlined
};
```

#### 相互の関連性

このオプションは、最適化レベルも同時に指定されていないかぎり (`-O` または `-xO`)、デバッグオプション `-g` を指定すると自動的に有効になります。

`-g0` デバッグオプションでは、`+d` は有効になりません。

`+d` オプションは、`-xO4` または `-xO5` を使用するときに実行される自動インライン化に影響を与えません。

#### 関連項目

`-g0`、`-g`

## A.2.8 -Dname[ =*def*]

プリプロセッサに対してマクロシンボル名 *name* を def と定義します。

このオプションは、ソースファイルの先頭に `#define` 指令を記述するのと同じです。`-D` オプションは複数指定できます。

コンパイラの定義済みマクロのリストについては、CC(1) のマニュアルページを参照してください。

## A.2.9 -d{y|n}

実行可能ファイル全体に対して動的ライブラリを 使用できるかどうか指定します。

このオプションは ld に渡されます。

このオプションは、コマンド行では 1 度だけしか使用できません。

### A.2.9.1 値

| 値 | 意味 |
|---|---|
| -dy | リンカーで動的リンクを実行します。 |
| -dn | リンカーで静的リンクを実行します。 |

#### デフォルト

-d オプションを指定しないと、-dy が使用されます。

#### 相互の関連性

64 ビットの環境では、多くのシステムライブラリは共有の動的ライブラリとしてのみ利用できます。これらのシステムライブラリには、libm.so および libc.so があります。libm.a と libc.a は提供していません。その結果、-Bstatic と -dn を使用すると 64 ビットの Oracle Solaris オペレーティングシステムでリンクエラーが生じる可能性があります。この場合、アプリケーションを動的ライブラリとリンクさせる必要があります。

#### 警告

このオプションを動的ライブラリと組み合わせて使用すると、重大なエラーが発生します。ほとんどのシステムライブラリは、動的ライブラリでのみ使用できます。

#### 関連項目

ld(1) のマニュアルページ、『リンカーとライブラリ』

## A.2.10 -dalign

(SPARC) 廃止。使用しないでください。-xmemalign=8s を使用してください。詳細は、281 ページの「A.2.145 -xmemalign=*ab*」を参照してください。

x86 プラットフォームでは、このオプションはメッセージを表示せずに無視されます。

## A.2.11 -dryrun

ドライバによって作成されたコマンドを表示しますが、コンパイルはしません。

このオプションは、コンパイルドライバが作成したサブコマンドの表示のみを行い、実行はしないように CC ドライバに指示します。

## A.2.12 -E

ソースファイルに対してプリプロセッサを実行しますが、コンパイルはしません。

C++ のソースファイルに対してプリプロセッサだけを実行し、結果を stdout (標準出力) に出力するよう CC ドライバに指示します。コンパイルは行われません。したがって .o ファイルは生成されません。

このオプションを使用すると、プリプロセッサで作成されるような行番号情報が出力に含まれます。

ソースコードにテンプレートが含まれている場合に -E オプションの出力をコンパイルするには、-E オプションとともに -template=no%extdef オプションを使用する必要が生じることがあります。アプリケーションコードで定義分離テンプレートのソースコードモデルが使用されている場合でも、-E オプションの出力が引き続きコンパイルされない可能性があります。詳細は、テンプレートの章を参照してください。

### A.2.12.1 例

このオプションは、プリプロセッサの処理結果を知りたいときに便利です。たとえば、次に示すプログラムでは、foo.cc は、183 ページの「A.2.12.1 例」に示す出力を生成します。

例 A–1 プリプロセッサのプログラム例 foo.cc

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
```

例 A–1　プリプロセッサのプログラム例 foo.cc　　*(続き)*

```
.
```

例 A–2　-E オプションを使用したときの foo.cc のプリプロセッサ出力

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power (int, int);


int main () {
int x;
x = power (2, 10);
}
```

### 警告

コードの定義分離モデルにテンプレートが含まれている場合は、このオプションの出力を C++ コンパイルへの入力として使用できないことがあります。

### 関連項目

`-P`

## A.2.13　-erroff[= t]

このコマンドは、C++ コンパイラの警告メッセージを無効にします。エラーメッセージには影響しません。このオプションは、`-errwarn` によってゼロ以外の終了状態を発生させるように指定されているかどうかにかかわらず、すべての警告メッセージに適用されます。

### A.2.13.1　値

*t* には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、`no%`*tag*、`%all`、`%none`。指定順序によって実行内容が異なります。たとえば、「`%all,no%`*tag*」と指定すると、*tag* 以外のすべての警告メッセージを抑制します。次の表は、`-erroff` の値を示しています。

表 A–2　-erroff の値

| 値 | 意味 |
|---|---|
| *tag* | *tag* のリストに指定されているメッセージを抑制します。`-errtags=yes` オプションで、メッセージのタグを表示することができます。 |
| `no%`*tag* | *tag* 以外のすべての警告メッセージの抑制を解除します。 |

表 A–2 -erroffの値 (続き)

| 値 | 意味 |
|---|---|
| %all | すべての警告メッセージを抑制します。 |
| %none | すべてのメッセージの抑制を解除します (デフォルト)。 |

### デフォルト

デフォルトは -erroff=%none です。-erroff と指定すると、-erroff=%all を指定した場合と同じ結果が得られます。

### 例

たとえば、-erroff=*tag* は、この tag が指定する警告メッセージを抑止します。それに対して、-erroff=%all,no% *tag* は、*tag* で識別されるメッセージ以外の警告メッセージをすべて抑制します。

警告メッセージのタグを表示するには、-errtags=yes オプションを使用します。

### 警告

-erroff オプションで無効にできるのは、C++ コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

### 関連項目

-errtags、-errwarn

## A.2.14 -errtags[= a]

C++ コンパイラのフロントエンドで出力される警告メッセージのうち、-erroff オプションで無効にできる、または -errwarn オプションで重大な警告に変換できるメッセージのメッセージタグを表示します。

### A.2.14.1 値とデフォルト

*a* には yes または no のいずれかを指定します。デフォルトは -errtags=no です。-errtags だけを指定すると、-errtags=yes を指定するのと同じことになります。

### 警告

C++ コンパイラドライバや、コンパイルシステムのその他のコンポーネントからのメッセージには、エラータグは含まれていません。そのため、これらのメッセージを -erroff で抑制したり、-errwarn で致命的エラーにすることはできません。

### 関連項目

`-erroff`、`-errwarn`

## A.2.15 -errwarn[=t]

指定した警告メッセージが生成された場合に、重大なエラーを出力して C++ コンパイラを終了する場合は、`-errwarn` を使用します。

### A.2.15.1 値

*t* には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、`no%`*tag*、`%all`、`%none`。このとき、順序が重要になります。たとえば、`%all,no%`*tag* と指定すると、*tag* 以外のすべての警告メッセージが生成された場合に、重大なエラーを出力して `cc` を終了します。

`-errwarn` の値を次の表に示します。

表 A–3 -errwarn の値

| 値 | 意味 |
|---|---|
| *tag* | *tag* に指定されたメッセージが警告メッセージとして発行されると、`cc` は致命的エラーステータスを返して終了します。*tag* に指定されたメッセージが発行されない場合は無効です。 |
| `no%`*tag* | *tag* に指定されたメッセージが警告メッセージとしてのみ発行された場合に、`cc` が致命的なエラーステータスを返して終了しないようにします。*tag* に指定されたメッセージが発行されない場合は無効です。このオプションは、*tag* または `%all` を使用して以前に指定したメッセージが警告メッセージとして発行されても `cc` が致命的エラーステータスで終了しないようにする場合に使用してください。 |
| `%all` | 警告メッセージが 1 つでも発行されると `cc` は致命的ステータスを返して終了します。`%all` に続いて `no%`*tag* を使用して、特定の警告メッセージを対象から除外することもできます。 |
| `%none` | どの警告メッセージが発行されても `cc` が致命的エラーステータスを返して終了することがないようにします。 |

### デフォルト

デフォルトは `-errwarn=%none` です。`-errwarn` のみを指定することは、`-errwarn=%all` と同等です。

### 警告

-errwarn オプションを使用して、障害状態で C++ コンパイラを終了するように指定できるのは、C++ コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

C++ コンパイラで生成される警告メッセージは、コンパイラのエラーチェックの改善や機能追加に応じて、リリースごとに変更されます。-errwarn=%all を指定してエラーなしでコンパイルされるコードでも、コンパイラの次期リリースではエラーを出力してコンパイルされる可能性があります。

### 関連項目

-erroff、 -errtags、 -xwe

## A.2.16 -fast

このオプションは、 実行ファイルの実行時のパフォーマンスのチューニングで効果的に使用することができるマクロです。-fast は、コンパイラのリリースによって変更される可能性があるマクロで、ターゲットのプラットフォーム固有のオプションに展開されます。-dryrun オプションまたは -xdryrun を使用して -fast の展開を調べ、-fast の該当するオプションを使用して実行可能ファイルのチューニングを行なってください。

このオプションは、コードをコンパイルするマシン上でコンパイラオプションの最適な組み合わせを選択して実行速度を向上するマクロです。

### A.2.16.1 展開

このオプションは、次のコンパイラオプションを組み合わせて、多くのアプリケーションのパフォーマンスをほぼ最大にします。

表 A–4 -fast の展開

| オプション | SPARC | x86 |
|---|---|---|
| -fns | X | X |
| -fsimple=2 | X | X |
| -nofstore | - | X |
| -xbuiltin=%all | X | X |
| -xlibmil | X | X |
| -xlibmopt | X | X |

表 A–4 -fast の展開 (続き)

| オプション | SPARC | x86 |
|---|---|---|
| -xmemalign | X | - |
| -xO5 | X | X |
| -xregs=frameptr | - | X |
| -xtarget=native | X | X |

## 相互の関連性

-fast マクロから展開されるコンパイラオプションが、指定されたほかのオプションに影響を与えることがあります。たとえば、次のコマンドの -fast マクロの展開には -xtarget=native が含まれています。そのため、ターゲットのアーキテクチャーは -xarch に指定された SPARC-V9 ではなく、32 ビットアーキテクチャーのものに戻されます。

誤

```
example% CC -xarch=sparcvis2 -fast test.cc
```

正

```
example% CC -fast -xarch=sparcvis2 test.cc
```

個々の相互の関連性については、各オプションの説明を参照してください。

このコード生成オプション、最適化レベル、組み込み関数の最適化、インラインテンプレートファイルの使用よりも、そのあとで指定するフラグの方が優先されます (例を参照)。ユーザーの指定した最適化レベルは、以前に設定された最適化レベルを無効にします。

-fast オプションには -fns -ftrap=%none が含まれているため、このオプションによってすべてのトラップが無効になります。

x86 では、–fast オプションに –xregs=frameptr が含まれます。特に C、Fortran、および C++ の混合ソースコードをコンパイルする場合は、その詳細について、このオプションの説明を参照してください。

## 例

次のコンパイラコマンドでは、最適化レベルは -xO3 になります。

```
example% CC –fast –xO3
```

次のコンパイラコマンドでは、最適化レベルは -xO5 になります。

```
example% CC -xO3 –fast
```

## 警告

別々の手順でコンパイルしてリンクする場合は、-fast オプションをコンパイルコマンドとリンクコマンドの両方に表示する必要があります。

-fast オプションでコンパイルしたオブジェクトバイナリは移植できません。たとえば、UltraSPARC-III システムで次のコマンドを指定すると、生成されるバイナリは UltraSPARC-II システムでは動作しません。

```
example% CC -fast test.cc
```

IEEE 標準の浮動小数点演算に依存するプログラムでは、このオプションを使用しないでください。異なる数値の結果、プログラムの強制終了、または予期しない SIGFPE シグナルが発生する可能性があります。

-fast の展開には、-D_MATHERR_ERRNO_DONTCARE が含まれます。

-fast を使用すると、コンパイラは errno 変数を設定しない同等の最適化コードを使用して呼び出しを浮動小数点関数に自由に置き換えることができます。さらに、-fast を指定すると、マクロ __MATHERR_ERRNO_DONTCARE も定義されます。このマクロにより、コンパイラは errno や、浮動小数点関数呼び出しのあとに発生した浮動小数点例外の有効性の保証を無視できます。結果として、errno の値または浮動小数点関数呼び出しのあとに発生した適切な浮動小数点例外の値に依存するユーザーコードが、一貫性のない結果を生成する可能性があります。

この問題を解決する 1 つの方法は、-fast を使用してそのようなコードをコンパイルしないことです。ただし、-fast の最適化が必要であり、かつコードが正しく設定された errno の値、または浮動小数点ライブラリの呼び出しのあとに発生した浮動小数点例外に依存している場合は、コンパイラがこのようなライブラリ呼び出しを最適化することを抑制するために、コマンド行で -fast のあとに次のオプションを指定してコンパイルしてください。

```
-xbuiltin=%none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

任意のプラットフォームで -fast の展開を表示するには、次の例に示すように、コマンド CC -dryrun -fast を実行します。

```
>CC -dryrun -fast  |& grep ###
###     command line files and options (expanded):
### -dryrun -xO5 -xarch=sparcvis2 -xcache=64/32/4:1024/64/4 \
-xchip=ultra3i -xmemalign=8s -fsimple=2 -fns=yes -ftrap=%none \
-xlibmil -xlibmopt -xbuiltin=%all -D__MATHERR_ERRNO_DONTCARE
```

## 関連項目

-fns、 -fsimple、 -ftrap=%none、 -xlibmil、 -nofstore、 -xO5、 -xlibmopt、 -xtarget=native

## A.2.17 -features=*a*[ ,a...]

コンマで区切って指定された C++ 言語のさまざまな機能を、有効または無効にします。

### A.2.17.1 値

キーワード *a* には、次の表に示されている値を指定できます。no% 接頭辞によって、関連付けられたオプションが無効になります。

表 A–5 -features の値

| 値 | 意味 |
|---|---|
| %all | 非推奨。使用しないでください。ほぼすべての -features オプションを有効にします。結果は予測できない場合があります。 |
| [no%]altspell | トークンの代替スペル (たとえば、「&&」の代わりの「and」) を認識します。デフォルトは altspell です。 |
| [no%]anachronisms | 廃止されている構文を許可します。無効にした場合 (つまり、-features=no%anachronisms)、廃止されている構文は許可されません。デフォルトは anachronisms です。 |
| [no%]bool | bool 型とリテラルを許可します。有効にした場合は、マクロ _BOOL=1 です。有効にしないと、マクロは定義されません。デフォルトは bool です。 |
| [no%]conststrings | リテラル文字列を読み取り専用メモリーに配置します。デフォルトは conststrings です。 |
| cplusplus_redef | 通常は事前に定義されているマクロ __cplusplus を、コマンド行で -D オプションを使用して再定義できるようにします。ソースコードで #define 指令を使用して __cplusplus の再定義を試みることは許可されていません。例:<br>CC -features=cplusplus_redef -D__cplusplus=1 ...<br>g++ コンパイラは通常、__cplusplus マクロを 1 に事前に定義するため、一部のソースコードはこの非標準の値に依存している可能性があります。(標準の値は、1998 年の C++ 標準または 2003 年の更新を実装しているコンパイラを示す 199711L です。将来の標準では、このマクロにより大きな値が必要になります。)<br>g++ を目的に作成されたコードをコンパイルするために __cplusplus を 1 に再定義することが必要でないかぎり、このオプションを使用しないでください。 |

表 A–5 -features の値 (続き)

| 値 | 意味 |
|---|---|
| [no%]except | C++ 例外を許可します。C++ 例外を無効にした場合 (つまり、-features=no%except)、関数に指定された throw は受け入れられますが無視されます。つまり、コンパイラは例外コードを生成しません。キーワード try、throw、および catch は常に予約されています。108 ページの「8.3 例外の無効化」を参照してください。デフォルトは except です。 |
| explicit | キーワード explicit を認識します。オプション no%explicit は許可されていません。 |
| [no%]export | キーワード export を認識します。デフォルトは export です。 |
| [no%]extensions | ほかの C++ コンパイラでは一般に受け入れられる非標準コードを許可します。デフォルトは no%extensions です。 |
| [no%]iddollar | $ シンボルを最初以外の識別子の文字として許可します。デフォルトは no%iddollar です。 |
| [no%]localfor | for 文に対して標準準拠の局所スコープ規則を使用します。デフォルトは localfor です。 |
| [no%]mutable | キーワード mutable を認識します。デフォルトは mutable です。 |
| namespace | キーワード namespace を認識します。オプション no%namespace は許可されていません。 |
| [no%]nestedacess | ネストしたクラスが、包含するクラスの private メンバーにアクセスできるようにします。デフォルト: -features=nestedaccess |
| rtti | 実行時の型識別 (RTTI) を許可します。オプション no%rtti は許可されていません。 |
| [no%]rvalueref | const 以外の参照の右辺値または一時値へのバインドを許可します。デフォルト: -features=no%rvalueref<br><br>デフォルトでは、C++ コンパイラは const 以外の参照を一時値または右辺値にバインドできないという規則を適用します。この規則を上書きするには、-features=rvalueref オプションを使用します。 |
| [no%]split_init | ローカルではない静的オブジェクトに対応する初期化子を個別の関数に配置します。-features=no%split_init を使用すると、コンパイラではすべての初期設定子が 1 つの関数に入れられます。-features=no%split_init を使用すると、コンパイル時間を可能なかぎり費やしてコードサイズを最小化します。デフォルトは split_init です。 |

表 A–5 -features の値 (続き)

| 値 | 意味 |
|---|---|
| [no%]transitions | 標準 C++ で問題があり、しかもプログラムが予想とは違った動作をする可能性があるか、または将来のコンパイラで拒否される可能性のある ARM 言語構造を許可します。-features=no%transitions を使用すると、コンパイラではこれらの言語構造をエラーとして扱います。-features=transitions を使用すると、コンパイラはエラーメッセージの代わりに、これらの構造に関する警告を発行します。<br><br>次の構造は移行エラーとみなされます。テンプレートの使用後にテンプレートを再定義する、typename 指令をテンプレートの定義で必要なときに省略する、int 型を暗黙的に宣言する。一連の移行エラーは将来のリリースで変更される可能性があります。デフォルトは transitions です。 |
| [no%]strictdestrorder | 静的記憶領域の持続中にオブジェクトを破棄する順序に関する、C++ 標準の必要条件に従います。デフォルトは strictdestrorder です。 |
| [no%]tmplrefstatic | 関数テンプレートからの依存静的関数または静的関数テンプレートの参照を許可します。デフォルトは標準準拠の no%tmplrefstatic です。 |
| [no%]tmplife | ANSI/ISO C++ 標準で定義された完全な式の最後にある式によって作成された一時オブジェクトをクリーンアップします。-features=no%tmplife が有効である場合は、大多数の一時オブジェクトはそのブロックの終わりに整理されます。デフォルトは tmplife です。 |
| %none | 非推奨。使用しないでください。ほぼすべての機能を無効にします。結果は予測できない場合があります。 |

## 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

次の値の使用は、標準ライブラリやヘッダーと互換性がありません。

- no%bool
- no%except
- no%mutable

## 警告

-features=%all または -features=%none を使用しないでください。これらのキーワードは非推奨であり、将来のリリースで削除される可能性があります。結果は予測できない場合があります。

-features=tmplife オプションを使用すると、プログラムの動作が変わる場合があります。プログラムが -features=tmplife オプションを指定してもしなくても動作するかどうかをテストする方法は、プログラムの移植性をテストする方法の 1 つです。

### 関連項目

表 3–17 および『C++ 移行ガイド』

## A.2.18 -filt[= filter[,filter...]]

コンパイラによってリンカーとコンパイラのエラーメッセージに通常適用されるフィルタリングを制御します。

### A.2.18.1 値

*filter* は、次の表に示されている値のいずれかである必要があります。%no 接頭辞によって、関連付けられたサブオプションが無効になります。

表 A–6 -filt の値

| 値 | 意味 |
|---|---|
| [no%]errors | C++ のリンカーエラーメッセージの説明を表示します。説明の抑止は、リンカーの診断を別のツールに直接提供している場合に便利です。 |
| [no%]names | C++ で符号化されたリンカー名を復号化します。 |
| [no%]returns | 関数の戻り型を復号化します。この種の復号化を抑止すると、より迅速に関数名を識別できるようになりますが、共有の不変式の戻り値の場合、一部の関数は戻り型でのみ異なります。 |
| [no%]stdlib | リンカーとコンパイラの両方のエラーメッセージに出力される標準ライブラリからの名前を簡略化し、標準ライブラリテンプレート型の名前をより容易に認識するための方法を提供します。 |
| %all | -filt=errors,names,returns,stdlib と同等です。これはデフォルトの動作です。 |
| %none | -filt=no%errors,no%names,no%returns,no%stdlib と同等です。 |

### デフォルト

-filt オプションを指定しない場合、または -filt を値なしで指定した場合、コンパイラでは -filt=%all が使用されます。

### 例

次の例では、このコードを -filt オプションでコンパイルしたときの影響を示します。

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // no definition provided
};

int main()
{
    type t;
}
```

-filt オプションを指定しないでコードをコンパイルすると、コンパイラでは -filt=errors,names,returns,stdlib が使用され、標準出力が表示されます。

```
example% CC filt_demo.cc
Undefined             first referenced
 symbol                  in file
type::~type()         filt_demo.o
type::__vtbl          filt_demo.o
[Hint: try checking whether the first non-inlined, /
non-pure virtual function of class type is defined]

ld: fatal: Symbol referencing errors. No output written to a.out
```

次のコマンドでは、C++ で符号化されたリンカー名の復号化が抑止され、C++ のリンカーエラーの説明が抑止されます。

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

次のコードについて考えてみましょう。

```
#include <string>
#include <list>
int main()
{
    std::list<int> l;
    std::string s(l); // error here
}
```

-filt=no%stdlib を指定すると、次の出力が得られます。

```
Error: Cannot use std::list<int, std::allocator<int>> to initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

`-filt=stdlib` を指定すると、次の出力が得られます。

```
Error: Cannot use std::list<int> to initialize std::string .
```

### 相互の関連性

`no%names` を使用しても `returns` や `no%returns` に影響はありません。つまり、次のオプションは同じ効果を持ちます。

- `-filt=no%names`
- `-filt=no%names,no%returns`
- `-filt=no%names,returns`

## A.2.19 -flags

`-xhelp=flags` と同じです。

## A.2.20 -fma[={ none|fused}]

(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有効にします。`-fma=none` を指定すると、これらの命令の生成を無効にします。`-fma=fused` を指定すると、コンパイラは浮動小数点の積和演算 (FMA) 命令を使用して、コードのパフォーマンスを改善する機会を検出しようとします。

デフォルトは `-fma=none` です。

コンパイラが積和演算 (FMA) 命令を生成するための最小要件は、`-xarch=sparcfmaf` と、最適化レベルが `-xO2` 以上であることです。積和演算 (FMA) 命令をサポートしていないプラットフォームでプログラムが実行されないようにするため、コンパイラは積和演算 (FMA) 命令を生成する場合、バイナリプログラムにマーク付けをします。

積和演算 (FMA) 命令により、積と和の間で中間の丸め手順が排除されます。その結果、`-fma=fused` を指定してコンパイルしたプログラムは、精度は減少ではなく増加する傾向にありますが、異なる結果になることがあります。

## A.2.21 -fnonstd

これは、x86 上では `-ftrap=common`、また SPARC 上では `-fns -ftrap=common` に展開されるマクロです。

詳細は、`-fns` および `-ftrap=common` を参照してください。

## A.2.22 -fns[={yes|no}]

- SPARC: SPARC 非標準浮動小数点モードを有効または無効にします。

  -fns=yes (または -fns) を指定すると、プログラムが実行を開始するときに、非標準浮動小数点モードが有効になります。

  このオプションを使うと、-fns を含むほかのマクロオプション (-fast など) のあとで非標準と標準の浮動小数点モードを切り替えることができます。

  一部の SPARC アーキテクチャーでは、非標準浮動小数点モードで「段階的アンダーフロー」が無効になり、非正規の数値を生成する代わりに、小さい値がゼロにフラッシュされます。さらに、このモードでは、非正規のオペランドが報告なしにゼロに置き換えられます。

  段階的アンダーフローや、非正規の数値をハードウェアでサポートしない SPARC アーキテクチャーでは、-fns=yes (または -fns) を使用すると、プログラムによってはパフォーマンスが著しく向上することがあります。

- x86: SSE flush-to-zero モードを選択または選択解除します。利用可能な場合は、denormals-are-zero モードです。

  このオプションは、非正規数の結果をゼロにフラッシュします。また利用可能な場合には、非正規数オペランドもゼロとして扱われます。

  このオプションは、SSE や SSE2 命令セットを利用しない従来の x86 浮動小数点演算には影響しません。

### A.2.22.1 値

-fns オプションには、次の表に示されている値を指定できます。

表 A–7 -fns の値

| 値 | 意味 |
|---|---|
| yes | 非標準浮動小数点モードを選択します。 |
| no | 標準浮動小数点モードを選択します。 |

### デフォルト

-fns を指定しないと、非標準浮動小数点モードは自動的には有効にされません。標準の IEEE 754 浮動小数点計算が行われます。つまり、アンダーフローは段階的です。

-fns だけを指定すると、-fns=yes が想定されます。

### 例

次の例では、-fast は複数のオプションに展開され、その中には -fns=yes (非標準浮動小数点モードを選択する) も含まれます。ところが、そのあとに続く -fns=no が初期設定を変更するので、結果的には、標準の浮動小数点モードが使用されます。

```
example% CC foo.cc -fast -fns=no
```

### 警告

非標準モードが有効になっていると、浮動小数点演算によって、IEEE 754 規格の条件に合わない結果が出力されることがあります。

1 つのルーチンを -fns オプションを使用してコンパイルする場合は、そのプログラムのすべてのルーチンを -fns オプションを使用してコンパイルしてください。それ以外の場合、予期しない結果が生じることがあります。

このオプションは、メインプログラムをコンパイルする場合にのみ有効です。

-fns=yes (または -fns) オプションを使用すると、プログラムで通常は IEEE 浮動小数点トラップハンドラによって管理される浮動小数点エラーが発生した場合、警告メッセージが生成されることがあります。

### 関連項目

『数値計算ガイド』、ieee_sun(3M) のマニュアルページ

## A.2.23 -fprecision=*p*

x86: デフォルト以外の浮動小数点精度モードを設定します。

-fprecision オプションは、浮動小数点制御ワード (FPCW) の丸め精度モードのビットを設定します。これらのビットは、基本演算 (加算、減算、乗算、除算、平方根) の結果をどの精度に丸めるかを制御します。

### A.2.23.1 値

*p* は、次の表に示されている値のいずれかである必要があります。

表 A–8 -fprecision の値

| 値 | 意味 |
|---|---|
| single | IEEE 単精度値に丸めます。 |
| double | IEEE 倍精度値に丸めます。 |

表 A–8 -fprecision の値 (続き)

| 値 | 意味 |
|---|---|
| extended | 利用可能な最大の精度に丸めます。 |

*p* が single か double であれば、丸め精度モードは、プログラムの実行が始まるときに、それぞれ single か double 精度に設定されます。*p* が extended であるか、-fprecision フラグが使用されていなければ、丸め精度モードは extended 精度のままです。

single 精度の丸めモードでは、結果が 24 ビットの有効桁に丸められます。double 精度の丸めモードでは、結果が 53 ビットの有効桁に丸められます。デフォルトの extended 精度の丸めモードでは、結果が 64 ビットの有効桁に丸められます。このモードは、レジスタにある結果をどの精度に丸めるかを制御するだけであり、レジスタの値には影響を与えません。レジスタにあるすべての結果は、拡張倍精度形式の全範囲を使って丸められます。ただし、メモリーに格納される結果は、指定した形式の範囲と精度に合わせて丸められます。

float 型の公称精度は single です。long double 型の公称精度は extended です。

### デフォルト

-fprecision オプションを指定しないと、丸め精度モードは extended になります。

### 警告

このオプションは、x86 システムでメインプログラムのコンパイル時に使用する場合にのみ有効で、64 ビット (-m64) または SSE2 対応 (-xarch=sse2) プロセッサでコンパイルする場合は無視されます。SPARC システムでも無視されます。

## A.2.24 -fround=*r*

起動時に IEEE 丸めモードを有効にします。

このオプションは、コンパイラが定数式を評価するときに使用できる IEEE 754 丸めモードを設定します。丸めモードは、プログラム初期化中の実行時に確立されます。

内容は、ieee_flags サブルーチンと同じです。これは実行時のモードを変更するために使用します。

### A.2.24.1 値

*r* は、次の表に示されている値のいずれかである必要があります。

表 A–9 -fround の値

| 値 | 意味 |
|---|---|
| nearest | もっとも近い数値に丸め、中間値の場合は偶数にします。 |
| tozero | ゼロに丸めます。 |
| negative | 負の無限大に丸めます。 |
| positive | 正の無限大に丸めます。 |

### デフォルト

-fround オプションを指定しないと、丸めモードは -fround=nearest になります。

### 警告

1 つのルーチンを -fround=*r* を使用してコンパイルする場合は、そのプログラムのすべてのルーチンを同じ -fround=*r* オプションを使用してコンパイルする必要があります。それ以外の場合、予期しない結果が生じることがあります。

このオプションは、メインプログラムをコンパイルするときにだけ有効です。

-xvector または -xlibmopt でのコンパイルには、デフォルトの丸めが必要です。-xvector または -xlibmopt のどちらか、あるいはその両方を使用してコンパイルされたライブラリにリンクするプログラムは、デフォルトの丸めが有効になっていることを確認する必要があります。

## A.2.25 -fsimple[= *n*]

浮動小数点最適化の設定を選択します。

このオプションは、オプティマイザが浮動小数点演算に関する仮定を単純化できるようにします。

### A.2.25.1 値

*n* を指定する場合、0、1、2 のいずれかにしなければいけません。

表 A–10 -fsimple の値

| 値 | 意味 |
|---|---|
| 0 | 仮定の設定を許可しません。IEEE 754 に厳密に準拠します。 |

表 A–10 -fsimple の値 *(続き)*

| 値 | 意味 |
|---|---|
| 1 | 安全な簡略化を行います。生成されるコードは IEEE 754 に厳密には準拠していませんが、大半のプログラムの数値結果は変わりありません。<br>-fsimple=1 の場合、次に示す内容を前提とした最適化が行われます。<br>■ IEEE 754 のデフォルトの丸めとトラップモードが、プロセスの初期化以後も変わらない。<br>■ 起こり得る浮動小数点例外を除き、目に見えない結果を出す演算が削除される可能性がある。<br>■ 無限大数または非数をオペランドとする演算は、その結果に非数を伝える必要がある。x*0 は 0 によって置き換えられる可能性がある。<br>■ 演算はゼロの符号を区別しない。<br>-fsimple=1 の場合、四捨五入や例外を考慮せずに完全な最適化を行うことは許可されていません。特に浮動小数点演算は、丸めモードを保持した定数について実行時に異なった結果を出す演算に置き換えることはできません。 |
| 2 | -fsimple=1 のすべての機能に加えて、浮動小数点演算の最適化を積極的に行い、丸めモードの変更によって多くのプログラムが異なった数値結果を出すようになります。たとえば、あるループ内の x/y の演算をすべて x*z に置き換えるような最適化を許可します。この最適化では、x/y はループ z=1/y 内で少なくとも 1 回評価されることが保証されており、y と z にはループの実行中に定数値が割り当てられます。 |

## デフォルト

-fsimple を指定しないと、コンパイラーは -fsimple=0 を使用します。

-fsimple を指定しても *n* の値を指定しないと、-fsimple=1 が使用されます。

## 相互の関連性

-fast は -fsimple=2 を意味します。

## 警告

このオプションによって、IEEE 754 に対する適合性が損なわれることがあります。

## 関連項目

-fast

## A.2.26 -fstore

(x86) 浮動小数点式の精度を強制的に設定します。

このオプションを指定すると、コンパイラは、次の場合に浮動小数点の式や関数の値を代入式の左辺の型に変換します。つまり、その値はレジスタにそのままの型で残りません。

- 式や関数を変数に代入する。
- 式をそれより短い浮動小数点型にキャストする。

このオプションを解除するには、オプション `-nofstore` を使用してください。SPARC プラットフォームでは、`-fstore` と `-nofstore` のどちらも、警告が出力されて無視されます。

### A.2.26.1 警告

丸めや切り捨てによって、結果がレジスタの値から生成される値と異なることがあります。

### 関連項目

`-nofstore`

## A.2.27 -ftrap=*t*[,*t*...]

起動時の IEEE トラップ モードを設定します。ただし、`SIGFPE` ハンドラは組み込まれません。トラップの設定と SIGFPE ハンドラの組み込みを同時に行うには、`ieee_handler`(3M) か `fex_set_handling`(3M) を使用します。複数の値を指定すると、それらの値は左から右に処理されます。

### A.2.27.1 値

*t* には、次の表に示す値のいずれかにできます。

表 A–11 -ftrap の値

| 値 | 意味 |
|---|---|
| `[no%]division` | ゼロによる除算をトラップします。 |
| `[no%]inexact` | 正確でない結果をトラップします。 |
| `[no%]invalid` | 無効な演算をトラップします。 |
| `[no%]overflow` | オーバーフローをトラップします。 |
| `[no%]underflow` | アンダーフローをトラップします。 |

表 A–11 -ftrap の値 (続き)

| 値 | 意味 |
|---|---|
| %all | 前述のすべてをトラップします。 |
| %none | 前述のどれもトラップしません。 |
| common | 無効、ゼロ除算、オーバーフローをトラップします。 |

[no%] 形式のオプションは、下の例に示すように、%all や commonフラグの意味を変更するときだけ使用します。[no%] 形式のオプション自体は、特定のトラップを明示的に無効にするものではありません。

### デフォルト

-ftrap を指定しない場合、コンパイラは -ftrap=%none とみなします。

### 例

-ftrap=%all,no%inexact は、inexact を除くすべてのトラップが設定されます。

### 警告

1 つのルーチンを -ftrap=*t* を使用してコンパイルする場合は、そのプログラムのすべてのルーチンを同じ -ftrap=*t* オプションを使用してコンパイルしてください。それ以外の場合、予期しない結果が生じることがあります。

-ftrap=inexact のトラップは慎重に使用してください。-ftrap=inexact では、浮動小数点の値が正確でないとトラップが発生します。たとえば、次の文ではこの条件が発生します。

```
x = 1.0 / 3.0;
```

このオプションは、メインプログラムをコンパイルするときにだけ有効です。このオプションを使用する際には注意してください。IEEE トラップを有効にする場合は、 -ftrap=common を使用します。

### 関連項目

ieee_handler(3M) および fex_set_handling(3M) のマニュアルページ

## A.2.28 –G

実行可能ファイルではなく動的共有ライブラリを構築します。

コマンド行で指定したソースファイルはすべて、デフォルトで -xcode=pic13 オプションでコンパイルされます。

テンプレートが含まれていて、-instances=extern オプションを使用してコンパイルされたファイルから共有ライブラリを構築すると、.o ファイルにより参照されているテンプレートインスタンスがすべてテンプレートキャッシュから自動的に含められます。

コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションと -G オプションを組み合わせて共有ライブラリを作成した場合は、生成された共有オブジェクトとのリンクでも、必ず同じオプションを指定してください。

254 ページの「A.2.113 -xcode=*a*」で推奨されているように、共有オブジェクトを作成するときは、64 ビット SPARC アーキテクチャー用にコンパイルされるすべてのオブジェクトファイルもまた、明示的な -xcode 値を使用してコンパイルする必要があります。

### A.2.28.1 相互の関連性

-c (コンパイルのみのオプション) を指定しないと、次のオプションがリンカーに渡されます。

- -dy
- –G
- -R

#### 警告

共有ライブラリの構築には、ld -G ではなく、CC -G を使用してください。こうすると、CC ドライバによって C++ に必要ないくつかのオプションが ld に自動的に渡されます。

-G オプションを使用すると、コンパイラはデフォルトの -l オプションを ld に渡しません。共有ライブラリを別の共有ライブラリに依存させる場合は、必要な -l オプションをコマンド行に渡す必要があります。たとえば、共有ライブラリを libCrun に依存させる場合は、-lCrun をコマンド行に渡す必要があります。

#### 関連項目

-dy、-xcode=pic13、-ztext、ld(1) のマニュアルページ、169 ページの「14.3 動的 (共有) ライブラリの構築」

## A.2.29 -g

dbx(1) または Debugger によるデバッグおよびパフォーマンスアナライザ analyzer(1) による解析用のシンボルテーブル情報を追加生成します。

コンパイラとリンカーに、デバッグとパフォーマンス解析に備えてファイルとプログラムを用意するように指示します。

これには、次の処理が含まれています。

- オブジェクトファイルと実行可能ファイルのシンボルテーブル内に、詳細情報 (スタブ) を生成する。
- デバッガがその一部の機能を実装するために呼び出すことのできるヘルパー関数を生成します。
- 最適化レベルが指定されていない場合は、関数のインライン生成を無効にします。つまり、最適化レベルも同時に指定されていない場合は、このオプションを使用すると +d オプションが指定されていることになります。-O または -xO レベルが指定された -g では、インライン化は無効になりません。
- 特定のレベルの最適化を無効にする。

### A.2.29.1 相互の関連性

このオプションと -xO*level* (あるいは、同等の -O オプションなど) を一緒に使用した場合、デバッグ情報が限定されます。詳細は、286 ページの「A.2.151 -xO*level*」を参照してください。

このオプションを使用するとき、最適化レベルが -xO4 以上の場合、可能なかぎりのシンボリック情報と最高の最適化が得られます。最適化レベルを指定しないで –g を使用した場合、関数呼び出しのインライン化が無効になります (–g を使用して最適化レベルが指定されると、インラインが有効になります)。

このオプションを指定し、-O と -xO のどちらも指定していない場合は、+d +d オプションが自動的に指定されます。

パフォーマンスアナライザの機能を最大限に利用するには、-g オプションを指定してコンパイルします。一部のパフォーマンス分析機能は -g を必要としませんが、注釈付きのソースコード、一部の関数レベルの情報、およびコンパイラの注釈メッセージを確認するには、-g でコンパイルする必要があります。詳細は、analyzer(1) のマニュアルページおよびパフォーマンスアナライザのマニュアルを参照してください。

-g オプションで生成される注釈メッセージは、プログラムのコンパイル時にコンパイラが実行した最適化と変換について説明します。メッセージを表示するには、er_src(1) コマンドを使用します。これらのメッセージはソースコードでインタリーブされます。

#### 警告

プログラムを別々の手順でコンパイルしてリンクしてから、1 つの手順に -g オプションを取り込み、ほかの手順から -g オプションを除外すると、プログラムの正確さは損なわれませんが、プログラムをデバッグする機能に影響します。-g (または -g0) でコンパイルされず、-g (または -g0) とリンクされているモジュールは、デ

バッグ用に正しく作成されません。通常、main 関数の入っているモジュールをデバッグするには、-g オプション (または -g0 オプション) でコンパイルする必要があります。

### 関連項目

+d、-g0、-xs、analyzer(1) のマニュアルページ、er_src(1) のマニュアルページ、ld(1) のマニュアルページ、『dbx コマンドによるデバッグ』(スタブの詳細について)、『パフォーマンスアナライザ』の各マニュアル。

## A.2.30 -g0

デバッグ用にコンパイルとリンクを行いますが、インライン化は無効にしません。

このオプションは、+d が無効になり、dbx がインライン化された関数でステップイン機能を使用できない点を除き、-g と同じです。

-xO3 以下の最適化レベルで -g0 を指定すると、ほとんど完全な最適化と可能なかぎりのシンボル情報を取得することができます末尾呼び出しの最適化とバックエンドのインライン化は無効です。

### A.2.30.1 関連項目

+d、-g、『dbx コマンドによるデバッグ』

## A.2.31 -g3

追加のデバッグ情報を生成します。

-g3 オプションは、-g0 と同じですが、dbx がソースコード内のマクロの展開を表示できるようにするためのデバッグシンボルテーブル情報が追加されています。この追加のシンボルテーブル情報により、-g0 でのコンパイルに比べて、結果として得られる .o ファイルや実行可能ファイルのサイズが増加する場合があります。

## A.2.32 –H

インクルードされるファイルのパス名を出力します。

現在のコンパイルに含まれている #include ファイルのパス名を標準エラー出力 (stderr) に 1 行に 1 つずつ出力します。

## A.2.33 -hname

生成する動的共有ライブラリに名前 *name* を割り当てます。動的共有ライブラリ

これは ld に渡されるリンカーオプションです。通常、-h のあとに指定する name (名前) は、-o のあとに指定する名前と同じでなければいけません。-h と *name* の間には、空白文字を入れても入れなくてもかまいません。

コンパイルの時ローダーは、作成対象の共有動的ライブラリに、指定の名前を割り当てます。この名前は、ライブラリのイントリンシック名として、ライブラリファイルに記録されます。-h*name* (名前) オプションを指定しないと、イントリンシック名はライブラリファイルに記録されません。

実行可能ファイルはすべて、必要な共有ライブラリファイルのリストを持っています。実行時のリンカーは、ライブラリを実行可能ファイルにリンクするとき、ライブラリのイントリンシック名をこの共有ライブラリファイルのリストの中にコピーします。共有ライブラリにイントリンシック名がないと、リンカーは代わりにその共有ライブラリファイルのパス名を使用します。

-h オプションを指定せずに共有ライブラリを構築する場合は、実行時のローダーはライブラリのファイル名のみを検索します。ライブラリを、同じファイル名を持つほかのライブラリに置換することもできます。共有ライブラリにイントリンシック名がある場合は、ローダーはファイルを読み込むときにイントリンシック名を確認します。イントリンシック名が一致しない場合は、ローダーは置換ファイルを使用しません。

### A.2.33.1 例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

## A.2.34 -help

-xhelp=flags と同じです。

## A.2.35 -Ipathname

#include ファイル検索パスに *pathname* を追加します。

このオプションは、相対ファイル名 (スラッシュ以外の文字で始まるファイル名) を持つ #include ファイルを検索するためのディレクトリリストに、*pathname* (パス名) を追加します。

コンパイラは、引用符付きのインクルードファイル (形式は #include "foo.h") を次の順序で検索します。

1. ソースが存在するディレクトリ
2. -I オプションで指定したディレクトリ内 (存在する場合)
3. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊目的ファイルの include ディレクトリ内

4. /usr/include ディレクトリ内

コンパイラは、山括弧をインクルードしたファイル (形式は #include <foo.h>) を次の順序で検索します。

1. -I オプションで指定したディレクトリ内 (存在する場合)
2. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊目的ファイルの include ディレクトリ内
3. /usr/include ディレクトリ内

---

注 – スペルが標準ヘッダーファイルの名前と一致する場合は、135 ページの「11.7.5 標準ヘッダーの実装」も参照してください。

---

### A.2.35.1 相互の関連性

-I- オプションを指定すると、デフォルトの検索規則が無効になります。

-library=no%Cstd を指定すると、その検索パスに C++ 標準ライブラリに関連付けられたコンパイラで提供されるヘッダーファイルがコンパイラでインクルードされません。134 ページの「11.7 C++ 標準ライブラリの置き換え」を参照してください。

-pti*path* が使用されていないと、コンパイラは -I*pathname* でテンプレートファイルを検索します。

-pti*path*の代わりに -I*pathname* を使用します。

このオプションは、置き換えられる代わりに蓄積されます。

#### 警告

コンパイラがインストールされている位置の /usr/include、/lib、/usr/lib を検索ディレクトリに指定しないでください。

#### 関連項目

-I-

## A.2.36 -I-

インクルードファイル検索規則を変更します。

#include "foo.h" 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. -I オプションで指定されたディレクトリ内 (-I- の前後)

2. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊な目的のファイルのディレクトリ

3. /usr/include ディレクトリ

#include <foo.h> 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. -I- のあとに現れる -I オプションで指定されたディレクトリ。

2. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊な目的のファイルのディレクトリ

3. /usr/include ディレクトリ

---

注 – インクルードファイルの名前が標準ヘッダーの名前と一致する場合は、135 ページの「11.7.5 標準ヘッダーの実装」も参照してください。

---

### A.2.36.1 例

次の例は、prog.cc のコンパイル時に -I- を使用した結果を示します。

```
prog.cc
#include "a.h"
#include <b.h>
#include "c.h"
c.h
#ifndef _C_H_1
#define _C_H_1
int c1;
#endif
inc/a.h
#ifndef _A_H
#define _A_H
#include "c.h"
int a;
#endif
inc/b.h
#ifndef _B_H
#define _B_H
#include <c.h>
int b;
#endif
inc/c.h
#ifndef _C_H_2
#define _C_H_2
int c2;
#endif
```

次のコマンドでは、#include "foo.h" 形式のインクルード文のカレントディレクトリ (インクルードしているファイルのディレクトリ) のデフォルトの検索動作を示します。#include "c.h" ステートメントを inc/a.h で処理するときは、コンパイラで inc

サブディレクトリから c.h ヘッダーファイルがインクルードされます。#include "c.h" 文を prog.cc で処理するときは、コンパイラで prog.cc を含むディレクトリから c.h ファイルがインクルードされます。-H オプションがインクルードファイルのパスを印刷するようにコンパイラに指示していることに注意してください。

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

次のコマンドでは、-I- オプションの影響を示します。コンパイラでは、#include "foo.h" 形式の文を処理するときにインクルードしているディレクトリを最初に探しません。代わりに、コマンド行に配置されている順番で、-I オプションで命名されたディレクトリを検索します。#include "c.h" 文を inc/a.h で処理するときは、コンパイラで ./c.h ヘッダーファイルが、inc/c.h ヘッダーファイルの代わりにインクルードされます。

```
example% CC -c -I. -I- -Iinc -H prog.cc
inc/a.h
        ./c.h
inc/b.h
        inc/c.h
./c.h
```

### 相互の関連性

-I- がコマンド行に現れると、現在のディレクトリが -I 指令で明示的に指定されていないかぎり、コンパイラは現在のディレクトリを検索しません。この影響は #include "foo.h" 形式のインクルード文にも及びます。

### 警告

コマンド行の最初の -I- だけが、説明した動作を引き起こします。

コンパイラがインストールされている位置の /usr/include、/lib、/usr/lib を検索ディレクトリに指定しないでください。

## A.2.37 -i

リンカー ld に、LD_LIBRARY_PATH と LD_LIBRARY_PATH_64 の設定をすべて無視するよう指示します。

## A.2.38 -include filename

このオプションを指定すると、コンパイラは *filename* を、主要なソースファイルの 1 行目に記述されているかのように `#include` プリプロセッサ指令として処理します。ソースファイル `t.c` の考慮:

```
main()
{
   ...
}
```

`t.c` を **`cc -include t.h t.c`** コマンドを使用してコンパイルする場合は、ソースファイルに次の内容が含まれているかのようにコンパイルが進行します。

```
#include "t.h"
main()
{
   ...
}
```

コンパイラが *filename* を検索する最初のディレクトリは現在の作業ディレクトリであり、ファイルが明示的にインクルードされている場合のようにメインのソースファイルが存在するディレクトリになるわけではありません。たとえば、次のディレクトリ構造では、同じ名前を持つ 2 つのヘッダーファイルが異なる場所に存在しています。

```
foo/
   t.c
   t.h
   bar/
     u.c
     t.h
```

作業ディレクトリが `foo/bar` であり、**`cc ../t.c -include t.h`** コマンドを使用してコンパイルする場合は、コンパイラによって `foo/bar` ディレクトリから取得された `t.h` がインクルードされますが、ソースファイル `t.c` 内で `#include` 指令を使用した場合の `foo/` ディレクトリとは異なります。

`-include` で指定されたファイルをコンパイラが現在の作業ディレクトリ内で見つけることができない場合は、コンパイラは通常のディレクトリパスでこのファイルを検索します。複数の `-include` オプションを指定する場合は、コマンド行で表示された順にファイルがインクルードされます。

## A.2.39 -inline

`-xinline` と同じです。

## A.2.40 -instances=*a*

テンプレートインスタンスの位置とリンケージを制御します。

### A.2.40.1 値

*a* は、次の表に示されている値のいずれかである必要があります。

表 A–12 -instances の値

| 値 | 意味 |
|---|---|
| `extern` | 必要なすべてのインスタンスをテンプレートリポジトリのリンカー *comdat* セクション内に置き、それらに対して大域リンケージを行います。リポジトリのインスタンスが古い場合は、再びインスタンス化されます。<br><br>注: コンパイルとリンクを別々に行うとき、コンパイル処理で `-instance=extern` を指定した場合には、リンク処理でも -instance=extern を指定する必要があります。 |
| `explicit` | 明示的にインスタンス化されたインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。必要なインスタンスがほかにあっても生成しません。 |
| `global` | 必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。 |
| `semiexplicit` | 明示的にインスタンス化されたインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。明示的なインスタンスにとって必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。必要なインスタンスがほかにあっても生成しません。 |
| `static` | 注: `-instances=static` は非推奨です。現在では `-instances=global` によって static のすべての利点が得られ、欠点はなくなっているため、`-instances=static` を使用する必要はなくなりました。このオプションは、このバージョンのコンパイラには存在しない、旧リリースのコンパイラにあった問題を克服するために用意されていました。<br><br>必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して静的リンケージを行います。 |

### デフォルト

`-instances` を指定しないと、`-instances=global` が想定されます。

### 関連項目

99 ページの「7.2.4 テンプレートインスタンスの配置とリンケージ」

## A.2.41 -instlib= filename

このオプションを使用すると、ライブラリ (共有、静的) と現在のオブジェクトで重複するテンプレートインスタンスの生成が禁止されます。一般に、ライブラリを使用するプログラムが多数のインスタンスを共有する場合、`-instlib=`*filename* を指定して、コンパイル時間の短縮を試みることができます。

### A.2.41.1 値

現在のコンパイルで生成できるテンプレートインスタンスを含むライブラリを指定するには、*filename* 引数を使用します。ファイル名引数には、スラッシュ (/) 文字を含める必要があります。現在のディレクトリに関連するパスの場合には、ドットスラッシュ (./) を使用します。

#### デフォルト

`-instlib=`*filename* オプションにはデフォルト値はないので、値を指定する場合にのみ使用します。このオプションは複数回指定でき、指定内容は追加されていきます。

#### 例

`libfoo.a` ライブラリと `libbar.so` ライブラリが、ソースファイル `a.cc` と共有する多数のテンプレートインスタンスをインスタンス化すると仮定します。`-instlib=`*filename* を追加してライブラリを指定すると、冗長性が回避されコンパイル時間を短縮できます。

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

#### 相互の関連性

`-g` を使ってコンパイルするとき、`-instlib=`*file* で指定したライブラリが `-g` でコンパイルされていない場合には、テンプレートインスタンスがデバッグ不能となります。この問題の対策としては、`-g` を指定するときに `-instlib=`*}file* を使用しないようにします。

#### 警告

`-instlib` によってライブラリを指定する場合には、そのライブラリとのリンクを行う必要があります。

#### 関連項目

`-template`、`-instances`、`-pti`

## A.2.42 -KPIC

SPARC: (廃止) `-xcode=pic32` と同じです。

x86: `-Kpic` と同じです。

このオプションは、共有ライブラリを構築するためにソースファイルをコンパイルするときに使用します。大域データへの各参照は、大域オフセットテーブルにおけるポインタの間接参照として生成されます。各関数呼び出しは、手続きリンケージテーブルを通してプログラムカウンタ (PC) 相対アドレス指定モードで生成されます。

## A.2.43 -Kpic

SPARC: (廃止) `-xcode=pic13` と同じです。

x86: 位置に依存しないコードを使ってコンパイルします。

このオプションは、共有ライブラリを構築するためにソースファイルをコンパイルするときに使用します。大域データへの各参照は、大域オフセットテーブルにおけるポインタの間接参照として生成されます。各関数呼び出しは、手続きリンケージテーブルを通してプログラムカウンタ (PC) 相対アドレス指定モードで生成されます。

## A.2.44 -keeptmp

コンパイル中に作成されたすべての一時ファイルを残します。

このオプションを `-verbose=diags` と一緒に使用すると、デバッグに便利です。

### A.2.44.1 関連項目

`-v`、`-verbose`

## A.2.45 -L*path*

ライブラリを検索するディレクトリのリストに *path* を追加します。

このオプションは `ld` に渡されます。コンパイラが提供するディレクトリよりも *path* が先に検索されます。

### A.2.45.1 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

### 警告

コンパイラがインストールされている位置の /usr/include、/lib、/usr/lib を検索ディレクトリに指定しないでください。

## A.2.46 -l*lib*

ライブラリ lib*lib*.a または lib*lib*.so をリンカーの検索ライブラリに追加します。

このオプションは ld に渡されます。ライブラリは一般に、lib*lib*.a や lib*lib*.so などの名前を持っています。ここで、lib と .a または .so の部分は必須です。このオプションでは *lib* の部分を指定できます。ライブラリは 1 つのコマンド行にいくつでも置くことができます。ライブラリは、-L*dir* で指定された順序で検索されます。

このオプションはファイル名のあとに使用してください。

### A.2.46.1 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

ライブラリが正しい順序で確実に検索されるようにするために、ソースとオブジェクトのリストのあとに -l*x* を置いてください。

### 警告

ライブラリを正しい順序で確実にリンクするには、-lthread ではなく -mt を使用して libthread にリンクする必要があります。

### 関連項目

-L*dir* と -mt

## A.2.47 -libmieee

-xlibmieee と同じです。

## A.2.48 -libmil

-xlibmil と同じです。

## A.2.49 -library=/[ ,/...]

l に指定した、cc が提供するライブラリを、コンパイルとリンクに組み込みます。

### A.2.49.1 値

キーワード *l* は、次の表の値のいずれかである必要があります。no% 接頭辞によって、関連付けられたオプションが無効になります。

表 A–13 -library の値

| 値 | 意味 |
|---|---|
| [no%]f77 | 非推奨。-xlang=f77 を使用してください。 |
| [no%]f90 | 非推奨。-xlang=f90 を使用してください。 |
| [no%]f95 | 非推奨。-xlang=f95 を使用してください。 |
| [no%]rwtools7 | 古い iostream Tools.h++ Version 7 を使用します。 |
| [no%]rwtools7_dbg | デバッグが可能な古い iostream Tools.h++ Version 7 を使用します。 |
| [no%]rwtools7_std | 標準 iostream Tools.h++ Version 7 を使用します。 |
| [no%]rwtools7_std_dbg | デバッグが可能な標準 iostream Tools.h++ Version 7 を使用します。 |
| [no%]interval | 非推奨。使用しないでください。-xia を使用します。 |
| [no%]iostream | 古い iostream ライブラリ libiostream を使用します。 |
| [no%]Cstd | C++ 標準ライブラリ libCstd を使用します。コンパイラで提供される C++ 標準ライブラリヘッダーファイルをインクルードします。 |
| [no%]Crun | C++ 実行時ライブラリ libCrun を使用します。 |
| [no%]gc | ガベージコレクション libgc を使用します。 |
| [no%]stlport4 | デフォルトの libCstd の代わりに STLport の標準ライブラリ実装 Version 4.5.3 を使用します。STLport の実装の詳細は、141 ページの「12.2 STLport」 を参照してください。 |
| [no%]stlport4_dbg | STLport のデバッグ可能なライブラリを使用します。 |
| [no%]sunperf | Sun Performance Library を使用します。 |

表 A–13 -library の値 (続き)

| 値 | 意味 |
|---|---|
| [no%]stdcxx4 | デフォルトの libCstd の代わりに Solaris の Apache stdcxx Version 4 C++ 標準ライブラリを使用します。このオプションにより、-mt オプションも暗黙的に設定されます。stdcxx ライブラリには、マルチスレッドモードが必要です。このオプションは、コンパイルのたびに、およびアプリケーション全体のリンクコマンドで一貫して使用する必要があります。-library=stdcxx4 を使用してコンパイルされたコードは、デフォルトの -library=Cstd または省略可能な -library=stlport4 を使用してコンパイルされたコードと同じプログラムでは使用できません。 |
| %none | libCrun の場合を除いて C++ ライブラリを使用しません。 |

### A.2.49.2 デフォルト

- 標準モード (デフォルトモード)
  - libCstd ライブラリは、-library=%none または -library=no%Cstd、-library=stdcxx4 または -library=stlport4 を使用して明確に除外されないかぎり常に含まれます。
  - libCrun ライブラリは、-library=no%Crun を使用して明確に除外されないかぎり常に含まれます。

libm ライブラリは、-library=%none を指定した場合でも常に含まれます。

### A.2.49.3 例

標準モードでどの C++ ライブラリも使用せずに (libCrun を除く) リンクするには:

```
example% CC -library=%none
```

標準モードで従来の iostream と RogueWave tools.h++ ライブラリを使用するには、次のコマンドを使用します。

```
example% CC –library=rwtools7,iostream
```

標準モードで標準 の iostream と Rogue Wave tools.h++ ライブラリを使用するコマンドは次のとおりです。

```
example% CC -library=rwtools7_std
```

### A.2.49.4 相互の関連性

-library でライブラリを指定すると、適切な -I パスがコンパイルで設定されます。リンクでは、適切な -L、-Y P、および -R パスと、-l オプションが設定されます。

このオプションは、置き換えられる代わりに蓄積されます。

区間演算ライブラリを使用するときは、libC、libCstd、または libiostream のいずれかのライブラリを取り込む必要があります。

-library オプションを使用すると、指定されたライブラリに対する -l オプションが正しい順序で処理されることが保証されます。たとえば、-library=rwtools7,iostream および -lirabary=iostream,rwtools7 のどちらでも、-l オプションは、-lrwtool -liostream の順序で ld に渡されます。

指定したライブラリは、システムサポートライブラリよりも前にリンクされます。

-library=stdcxx4 の場合は、Apache stdcxx ライブラリが Oracle Solaris プラットフォーム上の /usr/include と /usr/lib にインストールされている必要があります。

-library=sunperf と -xlic_lib=sunperf は同じコマンド行で使用できません。

どのコマンド行でも、-library=stlport4、-library=stdcxx4、または -library=Cstd オプションのうち使用できるオプションは、多くても 1 つだけです。

同時に使用できる Rogue Wave ツールライブラリは 1 つだけです。また、-library=stlport4 または -library=stdcxx4 を指定して Rogue Wave ツールライブラリと併用することはできません。

従来の iostream RogueWave ツールライブラリを標準モード (デフォルトモード) で取り込む場合は、libiostream も取り込む必要があります (詳細は、『C++ 移行ガイド』を参照してください)。標準 iostream RogueWave ツールライブラリは、標準モードでのみ使用できます。次のコマンド例は、RogueWave tools.h++ ライブラリオプションの有効もしくは無効な使用法について示します。

```
% CC -library=rwtools7,iostream foo.cc        <-- valid, classic iostreams
% CC -library=rwtools7 foo.cc                 <-- invalid

% CC -library=rwtools7_std foo.cc             <-- valid, standard iostreams
% CC -library=rwtools7_std,iostream foo.cc    <-- invalid
```

libCstd と libiostream の両方を含めた場合は、プログラム内で古い形式と新しい形式の iostream を使用して同じファイルにアクセスしないように注意する必要があります (たとえば、cout と std::cout)。同じプログラム内に標準 iostream と従来の iostream が混在し、その両方のコードから同じファイルにアクセスすると、問題が発生する可能性があります。

Crun ライブラリも、Cstd と stlport4 のどちらのライブラリもリンクしない標準モードのプログラムは、C++ 言語のすべての機能は使用できません。

-xnolib を指定すると、-library は無視されます。

### A.2.49.5 警告

別々の手順でコンパイルしてリンクする場合は、コンパイルコマンドに表示される一連の -library オプションをリンクコマンドにも表示する必要があります。

stlport4、Cstd、および iostream のライブラリは、固有の入出力ストリームを実装しています。これらのライブラリの 2 個以上を -library オプションを使って指定した場合、プログラム動作が予期しないものになる恐れがあります。STLport の実装の詳細は、141 ページの「12.2 STLport」を参照してください。

これらのライブラリは安定したものではなく、リリースによって変わることがあります。

### A.2.49.6 関連項目

131 ページの「11.4.1.1 従来の iostream およびレガシー RogueWave ツールについての注意」を参照してください。

-I、-l、-R、-staticlib、-xia、-xlang、-xnolib、138 ページの「警告:」、142 ページの「12.2.1 再配布とサポートされる STLport ライブラリ」、『Tools.h++ User's Guide』

-library=no%cstd オプションを使用して独自の C++ 標準ライブラリの使用を有効にする方法については、134 ページの「11.7 C++ 標準ライブラリの置き換え」を参照してください。

## A.2.50 -m32|-m64

コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。

-m32 を使用すると、32 ビット実行可能ファイルと共有ライブラリが作成されます。-m64 を使用すると、64 ビット実行可能ファイルと共有ライブラリが作成されます。

ILP32 メモリーモデル (32 ビット int、long、ポインタデータ型) は、64 ビット対応ではないすべての Oracle Solaris プラットフォームおよび Linux プラットフォームでのデフォルトです。LP64 メモリーモデル (64 ビット long、ポインタデータ型) は 64 ビット対応の Linux プラットフォームのデフォルトです。-m64 は LP64 モデル対応のプラットフォームでのみ使用できます。

-m32 を使用してコンパイルされたオブジェクトファイルまたはライブラリを、-m64 を使用してコンパイルされたオブジェクトファイルまたはライブラリにリンクすることはできません。

-m32|-m64 を指定してコンパイルしたモジュールは、-m32 |-m64 を指定してリンクする必要があります。コンパイル時とリンク時の両方で指定する必要のあるコンパイラオプションの完全なリストについては、48 ページの「3.3.3 コンパイル時とリンク時のオプション」を参照してください。

64 ビットプラットフォーム上で大量の静的データを使用するアプリケーション (-m64) には、-xmodel=medium も必要になることがあります。一部の Linux プラットフォームは、ミディアムモデルをサポートしていません。

以前のコンパイラリリースでは、-xarch で命令セットを選択すると、メモリーモデル ILP32 または LP64 が使用されていました。ほとんどのプラットフォーム上では Solaris Studio 12 コンパイラで開始し、コマンド行に -m64 だけを追加することが、64 ビットオブジェクトを作成するための正しい方法です。

Oracle Solaris では、-m32 がデフォルトです。64 ビットプログラムをサポートする Linux システムでは、-m64 -xarch=sse2 がデフォルトです。

### A.2.50.1 関連項目

-xarch.

## A.2.51 -mc

オブジェクトファイルの ELF .comment セクションから重複した文字列を削除します。-mc オプションを使用すると、mcs -c コマンドが呼び出されます。詳細は、mcs(1) のマニュアルページを参照してください。

## A.2.52 -misalign

SPARC: 廃止。このオプションは使用しないでください。代わりに -xmemalign=2i を使用してください。

## A.2.53 -mr[, string]

オブジェクトファイルの .comment セクションからすべての文字列を削除します。*string* が与えられた場合、そのセクションに *string* を埋め込みます。文字列に空白が含まれている場合は、文字列を引用符で囲む必要があります。このオプションを使用すると、mcs -d [-a *string*] が呼び出されます。

## A.2.54 -mt[={yes|no}]

このオプションを使用して、Oracle Solaris スレッドまたは POSIX スレッドの API を使用しているマルチスレッド化コードをコンパイルおよびリンクします。-mt=yes オプションにより、ライブラリが適切な順序でリンクされることが保証されます。

このオプションは -D_REENTRANT をプリプロセッサに渡します。

Oracle Solaris スレッドを使用するには、thread.h ヘッダーファイルをインクルードし、–mt=yes オプション付きでコンパイルします。Oracle Solaris プラットフォーム上で POSIX スレッドを使用するには、pthread.h ヘッダーファイルをインクルードし、-mt=yes -lpthread オプションを使用してコンパイルします。

Linux プラットフォーム上では、POSIX スレッドの API のみが使用できます (Linux プラットフォームには libthread はありません)。したがって、Linux プラットフォームで –mt=yes を使用すると、–lthread の代わりに –lpthread が追加されます。Linux プラットフォームで POSIX スレッドを使用するには、–mt=yes を使用してコンパイルします。

–G を使用してコンパイルする場合は、–mt=yes を指定しても、–lthread と –lpthread のどちらも自動的には含められません。共有ライブラリを構築する場合は、これらのライブラリを明示的にリストする必要があります。

(OpenMP 共有メモリー並列化 API を使用するための) –xopenmp オプションには、–mt=yes が自動的に含まれます。

-mt=yes を指定してコンパイルを実行し、リンクを個別の手順でリンクする場合は、コンパイル手順と同様にリンク手順でも -mt=yes オプションを使用する必要があります。-mt を使用して 1 つの変換ユニットをコンパイルおよびリンクする場合は、-mt を指定してプログラムのすべてのユニットをコンパイルおよびリンクする必要があります。

-mt=yes は、コンパイラのデフォルトの動作です。この動作が望ましくない場合は、-mt=no でコンパイルします。

オプション –mt は –mt=yes と同じです。

### A.2.54.1 関連項目

-xnolib、Oracle Solaris『Multithreaded Programming Guide』および『リンカーとライブラリ』

## A.2.55 -native

-xtarget=native と同じです。

## A.2.56 -noex

`-features=no%except` と同じです。

## A.2.57 -nofstore

x86: コマンド行の `-fstore` を取り消します。

式を強制的に `-fstore` によって呼び出される代入先の変数の精度にすることを取り消します。`-nofstore` は、`-fast` によって呼び出されます。通常は、`-fstore` がデフォルトです。

### A.2.57.1 関連項目

`-fstore`

## A.2.58 -nolib

`-xnolib` と同じです。

## A.2.59 -nolibmil

`-xnolibmil` と同じです。

## A.2.60 -norunpath

実行可能ファイルに共有ライブラリへの実行時検索パスを組み込みません。

実行可能ファイルが共有ライブラリを使用する場合、コンパイラは通常、実行時のリンカーに対して共有ライブラリの場所を伝えるために構築を行なったパス名を知らせます。これは、`ld` に対して `-R` オプションを渡すことによって行われます。このパスはコンパイラのインストール先によって決まります。

このオプションは、プログラムで参照される共有ライブラリの異なるパスを使用する可能性のある顧客に出荷される実行可能ファイルの構築に推奨されます。
133 ページの「11.6 共有ライブラリの使用」 を参照してください。

### A.2.60.1 相互の関連性

共有ライブラリをコンパイラのインストールされている位置で使用し、かつ `-norunpath` を使用する場合は、リンク時に `-R` オプションを使うか、または実行時に環境変数 `LD_LIBRARY_PATH` を設定して共有ライブラリの位置を明示しなければいけません。そうすることにより、実行時リンカーはそれらの共有ライブラリを検索できるようになります。

## A.2.61 -O

-O マクロは -xO3 に展開されます。(以前の一部のリリースでは、-O を -xO2 に展開していました)。

このデフォルトの変更によって、実行時のパフォーマンスが向上します。ただし、あらゆる変数を自動的に volatile と見なすことを前提にするプログラムの場合、-xO3 は不適切なことがあります。この前提を持つ典型的なプログラムは、独自の同期処理プリミティブを実装するデバイスドライバや古いマルチスレッドアプリケーションです。回避策は、-O ではなく、-xO2 でコンパイルすることです。

## A.2.62 -Olevel

-xO*level* と同じです。

## A.2.63 -o filename

出力ファイルまたは実行可能ファイルの名前を *filename* (ファイル名) に指定します。

### A.2.63.1 相互の関連性

コンパイラは、テンプレートインスタンスを格納する必要がある場合には、出力ファイルのディレクトリにあるテンプレートリポジトリに格納します。たとえば、次のコマンドでは、コンパイラはオブジェクトファイルを ./sub/a.o に、テンプレートインスタンスを ./sub/SunWS_cache 内のリポジトリにそれぞれ書き込みます。

```
example% CC -instances=extern -o sub/a.o a.cc
```

コンパイラは、読み込むオブジェクトファイルに対応するテンプレートリポジトリからテンプレートインスタンスを読み取ります。たとえば、次のコマンドは ./sub1/SunWS_Cache と ./sub2/SunWS_cache を読み取り、必要な場合は ./SunWS_cache に書き込みます。

```
example% CC -instances=extern sub1/a.o sub2/b.o
```

詳細は、103 ページの「7.4 テンプレートリポジトリ」を参照してください。

#### 警告

この *filename* は、コンパイラが作成するファイルの型に合った接尾辞を含むする必要があります。-c とともに使用されると、*filename* はターゲットの .o オブジェクトファイルを指定します。-G とともに使用されると、ターゲットの .so ライブラリファイルを指定します。このオプションとその引数は ld に渡されます。

CC ドライバはソースファイルを上書きしないため、*filename* をソースファイルと同じファイルにすることはできません。

## A.2.64 +p

標準に従っていないプリプロセッサの表明を無視します。

### A.2.64.1 デフォルト

+p を指定しないと、コンパイラは非標準のプリプロセッサの表明を認識します。

### 相互の関連性

+p を指定している場合は、次のマクロは定義されません。

- sun
- unix
- sparc
- i386

## A.2.65 -P

ソースの前処理だけでコンパイルはしません (接尾辞 .i のファイルを出力します)。

このオプションを指定すると、プリプロセッサが出力するような行番号情報はファイルに出力されません。

### A.2.65.1 関連項目

-E

## A.2.66 -p

廃止。298 ページの「A.2.159 -xpg」を参照してください。

## A.2.67 -pentium

x86: -xtarget=pentium と置き換えられています。

## A.2.68 -pg

廃止。-xpg を使用します。

## A.2.69 -PIC

SPARC: `-xcode=pic32` と同じです。

x86: `-Kpic` と同じです。

## A.2.70 -pic

SPARC: `-xcode=pic13` と同じです。

x86: `-Kpic` と同じです。

## A.2.71 -pta

`-template=wholeclass` と同じです。

## A.2.72 -ptipath

テンプレートソース用の検索ディレクトリを追加指定します。

このオプションは `-I`*pathname* (パス名) によって設定された通常の検索パスに代わるものです。`-pti`*path* (パス) フラグを使用した場合、コンパイラはこのパス上にあるテンプレート定義ファイルを検索し、`-I`*pathname* フラグを無視します。

`-pti`*path* よりも `-I`*pathname* を使用すると混乱が起きにくくなります。

### A.2.72.1 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

### A.2.72.2 関連項目

–I*pathname* および105 ページの「7.5.2 定義検索パス」

## A.2.73 -pto

`-instances=static` と同じです。

## A.2.74 -ptv

`-verbose=template` と同じです。

## A.2.75 -Qoption phase option[,option...]

*option* (オプション) を *phase* (コンパイル段階) に渡します。

複数のオプションを渡すには、コンマで区切って指定します。`-Qoption` でコンポーネントに渡されるオプションは、順序が変更されることがあります。ドライバが認識するオプションは、正しい順序に保持されます。ドライバがすでに認識しているオプションに、`-Qoption` は使わないでください。たとえば C++ コンパイラは、リンカー (`ld`) に対する `-z` オプションを認識します。次の例のようなコマンドを発行すると、`-z` オプションが適切な順序でリンカーに渡されます。

```
CC -G -zallextract mylib.a -zdefaultextract ... // correct
```

ただし、次の例のようなコマンドを指定した場合は、`-z` オプションを並べ替えることはできますが、正しくない結果になります。

```
CC -G -Qoption ld -zallextract mylib.a -Qoption ld -zdefaultextract ... // error
```

### A.2.75.1 値

*phase* には、次の表に示されている値のいずれかを指定する必要があります。

表 A–14 -Qoption の値

| SPARC | x86 |
|---|---|
| ccfe | ccfe |
| iropt | iropt |
| cg | ube |
| CClink | CClink |
| ld | ld |

### A.2.75.2 例

次のコマンドでは、`ld` が `CC` ドライバによって呼び出されると、`-Qoption` は `ld` に `-i` オプションと `-m` オプションを渡します。

```
example% CC -Qoption ld -i,-m test.c
```

### A.2.75.3 警告

意図しない結果にならないように注意してください。たとえば、次のオプションのシーケンス

```
-Qoption ccfe -features=bool,iddollar
```

は次のように解釈されます。

```
-Qoption ccfe -features=bool -Qoption ccfe iddollar
```

正しい指定は次のとおりです。

```
-Qoption ccfe -features=bool,-features=iddollar
```

これらの機能は `-Qoption` を必要とせず、例としてのみ使用されています。

## A.2.76 -qoption phase option

`-Qoption` と同じです。

## A.2.77 -qp

`-p` と同じです。

## A.2.78 -Qproduce sourcetype

`CC` ドライバに *sourcetype* (ソースタイプ) 型のソースコードを生成するよう指示します。

*Sourcetype* 接尾辞は、次の表で定義されています。

表 A–15 -Qproduce の値

| 接尾辞 | 意味 |
|---|---|
| `.i` | `ccfe` が作成する前処理済みの C++ のソースコード |
| `.o` | 生成されるオブジェクトコード |
| `.s` | `cg` が作成するアセンブラソース |

## A.2.79 -qproduce sourcetype

`-Qproduce` と同じです。

## A.2.80 -Rpathname[ :pathname...]

動的ライブラリの検索パスを実行可能ファイルに組み込みます。

このオプションは `ld` に渡されます。

### A.2.80.1 デフォルト

-R オプションが存在しない場合、出力オブジェクトに記録され、実行時リンカーに渡されるライブラリ検索パスは、-xarch オプションで指定されたターゲットアーキテクチャー命令によって異なります。-xarch が存在しない場合は、-xarch=generic が使用されます。

コンパイラが想定するデフォルトのパスを表示するには、–dryrun と –R の各オプションをリンカー ld に渡して出力を検査します。

### A.2.80.2 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

LD_RUN_PATH 環境変数が設定されている場合に、-R オプションを指定すると、-R に指定したパスが検索され、LD_RUN_PATH のパスは無視されます。

### A.2.80.3 関連項目

-norunpath、『リンカーとライブラリ』

## A.2.81 -S

コンパイルしてアセンブリコードだけを生成します。

CC ドライバはプログラムをコンパイルして、アセンブリソースファイルを作成します。しかし、プログラムのアセンブルは行いません。このアセンブリソースファイル名には、.s という接尾辞が付きます。

## A.2.82 -s

実行可能ファイルからシンボルテーブルを取り除きます。

出力する実行可能ファイルからシンボリック情報をすべて削除します。このオプションは ld に渡されます。

## A.2.83 -staticlib=l[ ,l...]

-library オプションで指定されたライブラリ (そのデフォルトを含む)、-xlang オプションで指定されたライブラリ、および -xia オプションで指定されたライブラリのうち、どの C++ ライブラリが静的にリンクされるかを示します。

### A.2.83.1 値

*l* は、次の表に示されている値のいずれかである必要があります。

表 A–16 -staticlib の値

| 値 | 意味 |
|---|---|
| [no%]*library* | *library* を静的にリンクします。*library* に有効な値は、-library で有効なすべての値 (%all と %none を除く)、-xlang で有効なすべての値、および (-xia とともに使用される) interval です。 |
| %all | -library オプション で指定されているすべてのライブラリと、-xlang オプションで指定されているすべてのライブラリ、-xia をコマンド行で指定している場合は区間ライブラリを静的にリンクします。 |
| %none | -library オプションと -xlang オプションに静的に指定されているライブラリをリンクしません。-xia をコマンド行に指定した場合は、区間ライブラリを静的にリンクしません。 |

### A.2.83.2 デフォルト

-staticlib を指定しないと、-staticlib=%none が想定されます。

### A.2.83.3 例

Crun は -library のデフォルト値であるため、次のコマンドは libCrun を静的にリンクします。

```
example% CC –staticlib=Crun (correct)
```

ただし、libgc は -library オプションで明示的に指定されないかぎりリンクされないため、次のコマンドは libgc をリンクしません。

```
example% CC –staticlib=gc (incorrect)
```

libgc を静的にリンクするには、次のコマンドを使用します。

```
example% CC -library=gc -staticlib=gc (correct)
```

次のコマンドは、librwtool ライブラリを動的にリンクします。librwtool はデフォルトのライブラリでもなく、-library オプションでも選択されていないため、-staticlib の影響はありません。

```
example% CC -lrwtool -library=iostream \
-staticlib=rwtools7 (incorrect)
```

次のコマンドは、librwtool ライブラリを静的にリンクします。

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 (correct)
```

次のコマンドは、Sun Performance Library を動的にリンクします。それは、これらのライブラリのリンクで -staticlib オプションを有効にするには、-library=sunperf を -staticlib=sunperf と組み合わせて使用する必要があるためです。

```
example% CC -xlic_lib=sunperf -staticlib=sunperf (incorrect)
```

このコマンドは、Sun Performance Library を静的にリンクします。

```
example% CC -library=sunperf -staticlib=sunperf (correct)
```

### A.2.83.4 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

-staticlib オプションは、デフォルトで暗黙的に選択された C++ ライブラリに加えて、-xia オプション、-xlang オプション、および -library オプションで明示的に選択された C++ ライブラリに対してのみ機能します。デフォルトでは、Cstd と Crun が選択されます。

### A.2.83.5 警告

*library* に使用できる値は安定したものではないため、リリースによって変わることがあります。

Oracle Solaris プラットフォーム上では、システムライブラリは静的ライブラリとして使用できません。

### A.2.83.6 関連項目

-library、132 ページの「11.5 標準ライブラリの静的リンク」

## A.2.84 -sync_stdio=[yes| no]

このオプションは、C++ の iostream と C の stdio の間の同期のために実行時のパフォーマンスが低下する場合に使用します。同期が必要なのは、同じプログラム内で iostream を使って cout に書き込み、stdio を使って stdout に書き込みを行う場合だけです。C++ 規格では同期が求められており、このため C++ コンパイラはデフォルトで同期を有効にします。ただし、しばしば、アプリケーションのパフォーマンスは同期なしの方が良くなることがあります。cout と stdout の一方にしか書き込みを行わない場合は、-sync_stdio=no オプションを使って同期を無効にすることができます。

### A.2.84.1 デフォルト

-sync_stdio を指定しなかった場合は、-sync_stdio=yes が設定されます。

### A.2.84.2 例

次の例を考えてみましょう。

```
#include <stdio.h>
#include <iostream>
int main()
{
   std::cout << "Hello ";
   printf("beautiful ");
   std::cout << "world!";
   printf("\n");
}
```

同期が有効な場合は、1 行だけ出力されます。

```
Hello beautiful world!
:
```

同期なしの場合、出力が混乱します。

### A.2.84.3 警告

このオプションは、ライブラリではなく実行可能ファイルのリンクでのみ有効です。

## A.2.85 -temp=*path*

一時ファイルのディレクトリを定義します。

このオプションは、コンパイル中に生成される一時ファイルを格納するディレクトリのパス名を指定します。コンパイラは `-temp` によって設定された値を、`TMPDIR` の値より優先します。

### A.2.85.1 関連項目

`-keeptmp`

## A.2.86 -template=opt[,opt...]

さまざまなテンプレートオプションを有効/無効にします。

### A.2.86.1 値

*opt* は、次の表に示されている値のいずれかである必要があります。

表 A–17 -template の値

| 値 | 意味 |
|---|---|
| [no%]extdef | 別のソースファイル内のテンプレート定義を検索します。no%extdef を使用すると、コンパイラは _TEMPLATE_NO_EXTDEF を事前定義します。 |
| [no%]geninlinefuncs | 明示的にインスタンス化されたクラステンプレートのための非参照インラインメンバー関数を生成します。 |
| [no%]wholeclass | 使用されている関数だけでなく、テンプレートクラス全体をインスタンス化します。クラスの少なくとも 1 つのメンバーを参照する必要があります。それ以外の場合、コンパイラはそのクラスのどのメンバーもインスタンス化しません。 |

### A.2.86.2 デフォルト

-template オプションを指定しないと、-template=no%wholeclass,extdef が使用されます。

### A.2.86.3 例

次のコードについて考えてみましょう。

```
example% cat Example.cc
     template <class T> struct S {
               void imf() {}
               static void smf() {}
     };

     template class S <int>;

     int main() {
     }
example%
```

-template=geninlinefuncs を指定した場合、S の 2 つのメンバー関数は、プログラム内で呼び出されなくてもオブジェクトファイルに生成されます。

```
example% CC -c -template=geninlinefuncs Example.cc
example% nm -C Example.o

Example.o:

[Index] Value Size Type  Bind  Other Shndx Name
[5]       0   0    NOTY  GLOB  0     ABS   __fsr_init_value
[1]       0   0    FILE  LOCL  0     ABS   b.c
[4]      16   32   FUNC  GLOB  0     2     main
[3]     104   24   FUNC  LOCL  0     2     void S<int>::imf()
                                           [__1cBS4Ci_Dimf6M_v_]
[2]     64    20   FUNC  LOCL  0     2     void S<int>::smf()
                                           [__1cBS4Ci_Dsmf6F_v_]
```

### A.2.86.4 関連項目

98 ページの「7.2.2 全クラスインスタンス化」、105 ページの「7.5 テンプレート定義の検索」

## A.2.87 -time

`-xtime` と同じです。

## A.2.88 -traceback[={ %none|common|signals_list}]

実行時に重大エラーが発生した場合にスタックトレースを発行します。

`-traceback` オプションを指定すると、プログラムによって特定のシグナルが生成された場合に、実行可能ファイルで `stderr` へのスタックトレースが発行されて、コアダンプが実行され、終了します。複数のスレッドが 1 つのシグナルを生成すると、スタックトレースは最初のスレッドに対してのみ生成されます。

追跡表示を使用するには、リンク時に `-traceback` オプションをコンパイラコマンド行に追加します。このオプションはコンパイル時にも使用できますが、実行可能バイナリが生成されない場合無視されます。`-traceback` を `-G` とともに使用して共有ライブラリを作成しないでください。

表 A–18 -traceback オプション

| オプション | 意味 |
|---|---|
| `common` | `sigill`、`sigfpe`、`sigbus`、`sigsegv`、または `sigabrt` の共通シグナルのいずれかのセットが発生した場合にスタックトレースを発行するべきであることを指定します。 |
| *signals_list* | スタックトレースを生成するシグナルの名前 (小文字) のコンマ区切りリストを指定します。`sigquit`、`sigill`、`sigtrap`、`sigabrt`、`sigemt`、`sigfpe`、`sigbus`、`sigsegv`、`sigsys`、`sigxcpu`、`sigxfsz` のシグナル (コアファイルが生成されるシグナル) をキャッチできます。<br>これらのシグナルの前に `no%` を指定すると、そのシグナルのキャッチが無効になります。<br>たとえば、**`-traceback=sigsegv,sigfpe`** と指定すると、`sigsegv` または `sigfpe` が発生した場合にスタックトレースとコアダンプが生成されます。 |
| `%none` または `none` | 追跡表示を無効にします |

このオプションを指定しない場合、デフォルトは `-traceback=%none` になります。

= 記号なしで、-traceback だけを指定すると、-traceback=common の意味になります。

注: コアダンプが必要ない場合は、次のコマンドを使用してコアダンプのサイズ制限を 0 に設定できます。

```
% limit coredumpsize 0
```

-traceback オプションは、実行時のパフォーマンスに影響しません。

## A.2.89 -U*name*

プリプロセッサシンボル *name* の初期定義を削除します。

このオプションは、コマンド行に指定された (CC ドライバによって暗黙的に挿入されるものも含む) -D オプションによって作成されるマクロシンボル *name* の初期定義を削除します。ほかの定義済みマクロや、ソースファイル内のマクロ定義が影響を受けることはありません。

CC ドライバにより定義される -D オプションを表示するには、コマンド行に -dryrun オプションを追加します。

### A.2.89.1 例

次のコマンドでは、事前に定義されているシンボル __sun を未定義にします。#ifdef (__sun) のような foo.cc 中のプリプロセッサ文では、シンボルが未定義であると検出されます。

```
example% CC -U__sun foo.cc
```

### A.2.89.2 相互の関連性

コマンド行には複数の -U オプションを指定できます。

すべての -U オプションは、存在している任意の -D オプションのあとに処理されます。つまり、同じ *name* がコマンド行上の -D と -U の両方に指定されている場合は、オプションが表示される順序にかかわらず *name* は未定義になります。

### A.2.89.3 関連項目

-D

## A.2.90 -unroll=*n*

-xunroll=*n* と同じです。

## A.2.91 -V

-verbose=version と同じです。

## A.2.92 -v

-verbose=diags と同じです。

## A.2.93 -verbose=v[,v...]

コンパイラメッセージの詳細度を制御します。

### A.2.93.1 値

*v* は、次の表に示されている値のいずれかである必要があります。no% 接頭辞によって、関連付けられたオプションが無効になります。

表 A–19 -verbose の値

| 値 | 意味 |
|---|---|
| [no%]diags | コンパイルパスごとにコマンド行を出力します。 |
| [no%]template | テンプレートインスタンス化の verbose モード (「検証」モードとも呼ばれる) を有効にします。verboseモードはコンパイル中にインスタンス化の各段階の進行を表示します。 |
| [no%]version | CC ドライバに、起動するプログラムの名前とバージョン番号を出力するよう指示します。 |
| %all | ほかのすべてのオプションを起動します。 |
| %none | -verbose=%none は -verbose=no%template,no%diags,no%version を指定することと同じです。 |

#### デフォルト

-verbose を指定されない場合、-verbose=%none が想定されます。

#### 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

## A.2.94 -Wc,arg

指定されたコンポーネント *c* に引数 *arg* を渡します。

引数は前の引数からコンマでのみ区切る必要があります。すべての `-W` 引数は、残りのコマンド行引数のあとに渡されます。コンマを引数の一部として含めるには、コンマの直前にエスケープ文字 \ (バックスラッシュ) を使用します。すべての `-W` arg は、通常のコマンド行引数のあとに渡されます。

たとえば、`-Wa,-o,objfile` は、`-o` と `objfile` をこの順番でアセンブラに渡します。また、`-Wl,-I,`*name;* を指定すると、リンク段階で動的リンカー `/usr/lib/ld.so.1` のデフォルト名が無効になります。

引数がツールに渡される順序は、ほかに指定されるコマンド行オプションとの関係で、今後のコンパイラリリースで変更される可能性があります。

*c* に指定可能な値を次の表に示します。

表 A–20 -W のフラグ

| フラグ | 意味 |
|---|---|
| `a` | アセンブラ : (`fbe`); (`gas`) |
| `c` | C++ コードジェネレータ: (`cg`) *(SPARC)*; |
| `d` | CC ドライバ |
| `l` | リンクエディタ (`ld`) |
| `m` | mcs |
| `O` (大文字の o) | 相互手続きオプティマイザ |
| `o` (小文字 o) | ポストオプティマイザ |
| `p` | プリプロセッサ (`cpp`) |
| `0` (ゼロ) | コンパイラ (`ccfe`) |
| `2` | オプティマイザ: (`iropt`) |

注: `-Wd` を使用して `CC` オプションを C++ コンパイラに渡すことはできません。

## A.2.95 +w

意図しない結果が生じる可能性のあるコードを特定します。`+w` オプションは、関数が大きすぎてインライン化できない場合、および宣言されたプログラム要素が未使用の場合に警告を生成しません。これらの警告は、ソース中の実際の問題を特定す

るものではないため、開発環境によっては不適切です。そのような環境では、+w でこれらの警告を生成しないようにすることで、+w をより効果的に使用することができます。これらの警告は、+w2 オプションの場合は生成されます。

このオプションを指定すると、次のような問題のある構造に関する追加の警告が生成されます。

- 移植性がない
- 間違っていると考えられる
- 効率が悪い

### A.2.95.1 デフォルト

+w オプションを指定しない場合、コンパイラは必ず問題となる構造についてのみ警告を出力します。

### A.2.95.2 関連項目

-w、+w2

## A.2.96 +w2

+w で発行されるすべての警告に加えて、おそらく害はないが、プログラムの最大の移植性を損なう可能性のある技術的な違反に関する警告を発行します。

+w2 オプションは、システムのヘッダーファイル中で実装に依存する構造が使用されている場合をレポートしなくなりました。システムヘッダーファイルが実装であるため、これらの警告は不適切でした。+w2 でこれらの警告を生成しないようにすることで、+w2 をより効果的に使用することができます。

### A.2.96.1 関連項目

+w

## A.2.97 –w

ほとんどの警告メッセージを抑止します。

このオプションは、コンパイラが警告を出力しない原因となります。ただし、一部の警告、特に旧式の構文に関する重要な警告は抑制できません。

### A.2.97.1 関連項目

+w

## A.2.98 -Xlinker arg

*arg* をリンカー ld(1) に渡します。–z *arg* と同等

## A.2.99 -Xm

-features=iddollar と同じです。

## A.2.100 -xaddr32

(*Solaris x86/x64* のみ) コンパイラフラグ -xaddr32=yes は、結果として生成される実行可能ファイルまたは共有オブジェクトを 32 ビットアドレス空間に制限します。

この方法でコンパイルする実行可能ファイルは、32 ビットアドレス空間に制限されるプロセスを作成する結果になります。

-xaddr32=no が指定されている場合は、通常の 64 ビットバイナリが生成されます。

-xaddr32 オプションを指定しないと、 -xaddr32=no が想定されます。

-xaddr32 だけを指定すると、 -xaddr32=yes が想定されます。

このオプションは、 -m64 コンパイルのみ、および SF1_SUNW_ADDR32 ソフトウェア機能をサポートしている Oracle Solaris プラットフォームのみに適用できます。Linux カーネルはアドレス空間制限をサポートしていないので、Linux ではこのオプションは使用できません。

単一オブジェクトファイルが -xaddr32=yes を指定してコンパイルされた場合、リンク時には、出力ファイル全体が -xaddr32=yes を指定してコンパイルされたものと見なされます。

32 ビットアドレス空間に制限される共有オブジェクトは、制限された 32 ビットモードのアドレス空間内で実行されるプロセスから読み込む必要があります。

詳細は、『リンカーとライブラリ』で説明されている SF1_SUNW_ADDR32 ソフトウェア機能の定義を参照してください。

## A.2.101 -xalias_level[= n]

次のコマンドを指定すると、C++ コンパイラは、型に基づく別名の解析および最適化を実行できます。

-xalias_level[=*n* ]

ここで、*n* は any、simple、compatible のいずれかです。

### A.2.101.1 -xalias_level=any

このレベルの解析では、ある型を別名で定義できるとものとして処理されます。ただしこの場合でも、一部の最適化が可能です。

### A.2.101.2 -xalias_level=simple

単純型は別名で定義されていないものとして処理されます。記憶オブジェクトは、次のいずれかの単純型である動的な型を持っている必要があります。

char, signed char, unsigned char wchar_t, データポインタ型
short int, unsigned short int, int unsigned int, 関数ポインタ型
long int, unsigned long int, long long int, unsigned long long int, データメンバーポインタ型
float, double long double, 列挙型関数メンバーポインタ型

記憶オブジェクトには、次の型の左辺値を使用してのみアクセスするべきです。

- オブジェクトの動的な型
- オブジェクトの動的な型を constant または volatile で修飾したもの。つまり、オブジェクトの動的な型に相当する符号付きまたは符号なしの型。
- オブジェクトの動的な型を constant または volatile で修飾したものに相当する、符号付きまたは符号なしの型。
- 前述の型のいずれかがメンバーに含まれる集合体または共用体 (再帰的に、その下位の集合体またはそれに含まれる共用体のメンバーについても該当します)。
- char 型または unsigned char 型

### A.2.101.3 -xalias_level=compatible

配置非互換の型は、別名で定義されていないものとして処理されます。記憶オブジェクトは、次の型の lvalue を使用してだけアクセスされます。

- オブジェクトの動的な型
- オブジェクトの動的な型を constant または volatile で修飾したもの。つまり、オブジェクトの動的な型に相当する符号付きまたは符号なしの型。
- オブジェクトの動的な型を constant または volatile で修飾したものに相当する、符号付きまたは符号なしの型。
- 前述の型のいずれかがメンバーに含まれる集合体または共用体 (再帰的に、その下位の集合体またはそれに含まれる共用体のメンバーについても該当します)。
- オブジェクトの動的な型の (多くの場合は constant または volatile で修飾した) 基本クラス型。

- char 型または unsigned char 型

コンパイラでは、すべての参照の型が、対応する記憶オブジェクトの動的な型と配置の互換性があると想定します。2 つの型は、次の条件の場合に配置互換になります。

- 2 つの型が同じ型である場合
- 2 つの型が constant と volatile のどちらで修飾されたかという点だけが異なる場合
- 符号付き整数型のそれぞれについて、対応する (ただし、異なる) 符号なし整数型が存在する場合、これらの対応する型には配置の互換性があります。
- 2 つの列挙型は、基礎の型が同一の場合に配置互換になります。
- 2 つの Plain Old Data (POD) 構造体型は、メンバーの数が同じであり、かつ順序で対応するメンバーの型に配置の互換性がある場合に配置の互換性があります。
- 2 つの POD 共用体型は、メンバー数が同一で、対応するメンバー (順番は任意) が配置互換である場合に配置互換になります。

参照は、一部の場合に、記憶オブジェクトの動的な型と配置非互換になります。

- POD 共用体に、開始シーケンスが共通の POD 構造体が複数含まれていて、その POD 共用体オブジェクトにそれらの POD 構造体のいずれかが含まれている場合は、任意の POD 構造体の共通の開始部分を調べることができます。2 つの POD 構造体が共通の開始シーケンスを共有していて、対応するメンバーの型が配置互換であり、開始メンバーのシーケンスでビットフィールドの幅が同一の場合に、2 つの POD 構造体は開始シーケンスが共通になります。
- `reinterpret_cast` を使用して適切に変換された POD 構造体オブジェクトへのポインタは、その最初のメンバーか、またはそのメンバーがビットフィールドである場合はそのビットフィールドが存在するユニットを指します。

### A.2.101.4 デフォルト

`-xalias_level` を指定しない場合は、コンパイラでは `-xalias_level=any` が指定されます。`-xalias_level` を値なしで指定した場合は、コンパイラでは `-xalias_level=compatible` が指定されます。

### A.2.101.5 相互の関連性

コンパイラは、`-xO2` 以下の最適化レベルでは、型に基づく別名の解析および最適化を実行しません。

### A.2.101.6 警告

`reinterpret_cast` またはこれに相当する旧形式のキャストを使用している場合には、解析の前提にプログラムが違反することがあります。また、次の例に示す共用体の型のパンニングも、解析の前提に違反します。

```
union bitbucket{
  int i;
  float f;
};

int bitsof(float f){
bitbucket var;
var.f=3.6;
return var.i;
}
```

## A.2.102 -xanalyze={code|no}

ソースコードの静的分析を生成します。Oracle Solaris Studio コードアナライザを使用して表示できます。

–xanalyze=code を指定してコンパイルし、別の手順でリンクするときは、リンク手順でも –xanalyze=code を含めてください。

デフォルトは –xanalyze=no です。詳細は、Oracle Solaris Studio コードアナライザのドキュメントを参照してください。

## A.2.103 -xannotate[=yes|no]

(*Solaris* のみ) あとで最適化および可観測性ツール binopt(1)、code-analyzer(1)、discover(1)、collect(1)、および uncover(1) で使用できるバイナリを作成します。

デフォルトは -xannotate=yes です。値なしで -xannotate を指定することは、-xannotate=yes と同義です。

最適化および監視ツールを最適に使用するためには、コンパイル時とリンク時の両方で -xannotate=yes が有効である必要があります。最適化および監視ツールを使用しないときは、-xannotate=no を指定してコンパイルおよびリンクすると、バイナリとライブラリが若干小さくなります。

Linux システムでは、このオプションはありません。

## A.2.104 -xar

アーカイブライブラリを作成します。

テンプレートを使用する C++ のアーカイブを構築する場合は、テンプレートリポジトリ内でインスタンス化されたテンプレート関数をそのアーカイブに含めます。テンプレートリポジトリは、少なくとも 1 つのオブジェクトファイルを

-instances=extern オプションでコンパイルしたときにのみ使用されます。-xar を使用してコンパイルすると、それらのテンプレートが必要に応じてアーカイブに自動的に追加されます。

ただし、コンパイラのデフォルトではテンプレートキャッシュが使用されないため、多くの場合、-xar オプションは必要ありません。一部のコードが -instances=extern を使用してコンパイルされていないかぎり、通常の ar(1) コマンドを使用して C++ コードのアーカイブ (.a ファイル) を作成できます。その場合、または確信がない場合は、ar コマンドの代わりに CC -xar を使用します。

### A.2.104.1 値

-xar を指定すると、ar -c -r が起動され、アーカイブがゼロから作成されます。

#### 例

次のコマンド行は、ライブラリファイルとオブジェクトファイルに含まれるテンプレート関数をアーカイブします。

```
example% CC -xar -o libmain.a a.o b.o c.o
```

#### 警告

テンプレートデータベースの .o ファイルをコマンド行に追加しないでください。

アーカイブを構築するときは、ar コマンドを使用しないでください。CC -xar を使用して、テンプレートのインスタンス化情報が自動的にアーカイブに含まれるようにしてください。

#### 関連項目

ar(1) のマニュアルページ

## A.2.105 –xarch=*isa*

対象となる命令セットアーキテクチャー (*ISA*) を指定します。

このオプションは、コンパイラが生成するコードを、指定した命令セットアーキテクチャーの命令だけに制限します。このオプションは、すべてのターゲットを対象とするような命令としての使用は保証しません。ただし、このオプションを使用するとバイナリプログラムの移植性に影響を与える可能性があります。

注 – 意図するメモリーモデルとして LP64 (64 ビット) または ILP32 (32 ビット) を指定するには、それぞれ -m64 または -m32 オプションを使用してください。次に示すように、以前のリリースとの互換性を除き、-xarch オプションでメモリーモデルを指定できなくなりました。

アーキテクチャー固有の命令を使用する _asm 文またはインラインテンプレート (.il ファイル) を使用したコードには、コンパイルエラーを回避するために、適切な -xarch 値でのコンパイルが必要になることがあります。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ -xarch の値を指定してください。コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧については、48 ページの「3.3.3 コンパイル時とリンク時のオプション」を参照してください。

### A.2.105.1 SPARC および x86 用の -xarch フラグ

次の表は、SPARC プラットフォームと x86 プラットフォームの両方に共通する -xarch キーワードの一覧です。

表 A–21 SPARC および x86 での -xarch フラグ

| フラグ | 意味 |
|---|---|
| generic | ほとんどのプロセッサに共通の命令セットを使用します。これはデフォルト値です。 |
| generic64 | ほとんどの 64 ビットプラットフォームで良好なパフォーマンスになるようにコンパイルします。<br>このオプションは -m64 -xarch=generic と同義で、以前のリリースとの互換性のために提供されています。 |
| native | このシステムで良好なパフォーマンスを得られるようにコンパイルします。現在コンパイルしているシステムプロセッサにもっとも適した設定を選択します。 |
| native64 | このシステムで良好なパフォーマンスになるようにコンパイルします。<br>このオプションは -m64 -xarch=native と同義で、以前のリリースとの互換性のために提供されています。 |

### A.2.105.2 SPARC での -xarch のフラグ

次の表に、SPARC プラットフォームでの各 -xarch キーワードの詳細を示します。

表 A–22 SPARC プラットフォーム用の -xarch フラグ

| フラグ | 意味 |
|---|---|
| sparc | VIS (Visual Instruction Set) を使用せず、その他の実装に固有の ISA 拡張機能も使用せずに、SPARC-V9 ISA 用にコンパイルします。このオプションを使用して、コンパイラは、V9 ISA で良好なパフォーマンスが得られるようにコードを生成できます。 |
| sparcvis | SPARC-V9 + VIS (Visual Instruction Set) version 1.0 + UltraSPARC 拡張機能用のコンパイルを実行します。このオプションを使用すると、コンパイラは、UltraSPARC アーキテクチャー上で良好なパフォーマンスが得られるようにコードを生成することができます。 |
| sparcvis2 | UltraSPARC アーキテクチャー + VIS (Visual Instruction Set) version 2.0 + UltraSPARC-III 拡張機能用のオブジェクトコードを生成します。 |
| sparcvis3 | SPARC VIS version 3 の SPARC-V9 ISA 用にコンパイルします。SPARC-V9 命令セット、VIS (Visual Instruction Set) Version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) Version 2.0、積和演算 (FMA) 命令、および VIS (Visual Instruction Set) Version 3.0 を含む UltraSPARC-III 拡張機能の命令をコンパイラが使用できるようになります。 |
| sparcfmaf | SPARC-V9 命令セット、VIS (Visual Instruction Set) version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) version 2.0 を含む UltraSPARC-III 拡張機能、および浮動小数点積和演算用の SPARC64 VI 拡張機能の命令をコンパイラが使用できるようになります。<br><br>-xarch=sparcfmaf は fma=fused と組み合わせて使用し、ある程度の最適化レベルを指定することで、コンパイラが自動的に積和命令の使用を試みるようにする必要があります。 |
| sparcima | SPARC IMA バージョンの SPARC-V9 ISA 用にコンパイルします。コンパイラが、SPARC-V9 命令セットに加えて、VIS (Visual Instruction Set) Version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) Version 2.0 を含む UltraSPARC-III 拡張機能、浮動小数点の積和演算用の SPARC64 VI 拡張機能、整数の積和演算用の SPARC64 VII 拡張機能からの命令を使用できるようにします。 |
| sparc4 | SPARC4 バージョンの SPARC-V9 ISA 用にコンパイルします。コンパイラが SPARC-V9 命令セットからの命令、さらに拡張機能 (VIS 1.0 を含む)、UltraSPARC-III 拡張機能 (VIS 2.0 を含む)、浮動小数点積和演算 (FMA) 命令、VIS 3.0、および SPARC4 命令を使用できるようになります。 |
| v9 | -m64 -xarch=sparc と同等です。64 ビットメモリーモデルを得るために -xarch=v9 を使用する古いメイクファイルとスクリプトでは、-m64 だけを使用すれば十分です。 |
| v9a | -m64 -xarch=sparcvis と同等で、以前のリリースとの互換性のために提供されています。 |

表 A–22 SPARC プラットフォーム用の -xarch フラグ (続き)

| フラグ | 意味 |
|---|---|
| v9b | -m64 -xarch=sparcvis2 と同等で、以前のリリースとの互換性のために提供されています。 |

また、次のことにも注意してください。

- generic、sparc、sparcvis2、sparcvis3、sparcfmaf、sparcima でコンパイルされたオブジェクトライブラリファイル (.o) をリンクして、一度に実行できます。ただし、実行できるのは、リンクされているすべての命令セットをサポートしているプロセッサのみです。
- 特定の設定で、生成された実行可能ファイルが実行されなかったり、従来のアーキテクチャーよりも実行速度が遅くなったりする場合があります。また、4 倍精度 (REAL*16 および long double) 浮動小数点命令は、これらの命令セットアーキテクチャーのいずれにも実装されないため、コンパイラは、そのコンパイラが生成したコードではそれらの命令を使用しません。

### A.2.105.3 x86 での -xarch のフラグ

次の表に、x86 プラットフォームでの -xarch フラグを示します。

表 A–23 x86 上の -xarch フラグ

| フラグ | 意味 |
|---|---|
| amd64 | -m64 -xarch=sse2 と同等です (Solaris のみ)。64 ビットメモリーモデルを得るために -xarch=amd64 を使用する古いメイクファイルとスクリプトでは、-m64 だけを使用すれば十分です。 |
| amd64a | -m64 -xarch=sse2a と同等です (Solaris のみ)。 |
| pentium_pro | 命令セットを 32 ビット Pentium Pro アーキテクチャーに限定します。 |
| pentium_proa | AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット Pentium Pro アーキテクチャーに追加します。 |
| sse | SSE 命令セットを Pentium Pro アーキテクチャーに追加します。 |
| ssea | AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット SSE アーキテクチャーに追加します。 |
| sse2 | SSE2 命令セットを Pentium Pro アーキテクチャーに追加します。 |
| sse2a | AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット SSE2 アーキテクチャーに追加します。 |
| sse3 | SSE3 命令セットを SSE2 命令セットに追加します。 |
| sse3a | AMD 拡張命令 (3dnow など) を SSE3 命令セットに追加します。 |

表 A–23 x86 上の -xarch フラグ (続き)

| フラグ | 意味 |
|---|---|
| ssse3 | SSSE3 命令セットで、Pentium Pro、SSE、SSE2、および SSE3 の各命令セットを補足します。 |
| sse4_1 | SSE4.1 命令セットで、Pentium Pro、SSE、SSE2、SSE3、および SSSE3 の各命令セットを補足します。 |
| sse4_2 | SSE4.2 命令セットで、Pentium Pro、SSE、SSE2、SSE3、SSSE3、および SSE4.1 の各命令セットを補足します。 |
| amdsse4a | AMD SSE4a 命令セットを使用します。 |
| aes | Intel Advanced Encryption Standard 命令セットを使用します。 |
| avx | Intel Advanced Vector Extensions 命令セットを使用します。 |

x86 プラットフォームで、プログラムの一部が –m64 を使用してコンパイルまたはリンクされる場合は、プログラムのすべての部分もこれらのオプションのいずれかを使用してコンパイルされる必要があります。各種 Intel 命令セットアーキテクチャー (SSE、SSE2、SSE3、SSSE3 など) の詳細は、Intel-64 および IA-32 の『Intel Architecture Software Developer's Manual』を参照してください。

26 ページの「1.2 x86 の特記事項」および 27 ページの「1.4 バイナリの互換性の妥当性検査」も参照してください。

### A.2.105.4 相互の関連性

このオプションは単体でも使用できますが、-xtarget オプションの展開の一部でもあります。したがって、特定の -xtarget オプションで設定される -xarch のオーバーライドにも使用できます。-xtarget=ultra2 は -xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1 に展開されます。次のコマンドでは、-xarch=v8plusb は、-xtarget=ultra2 の展開で設定された -xarch=v8plusa より優先されます。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

-xarch=generic64、-xarch=native64、-xarch=v9、-xarch=v9a、または -xarch=v9b による -compat[=4] の使用はサポートされていません。

### A.2.105.5 警告

このオプションを最適化と併せて使用する場合、適切なアーキテクチャーを選択すると、そのアーキテクチャー上での実行パフォーマンスを向上させることができます。ただし、適切な選択をしなかった場合、パフォーマンスが著しく低下するか、あるいは、作成されたバイナリプログラムが目的のターゲットプラットフォーム上で実行できない可能性があります。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ -xarch の値を指定してください。

## A.2.106 -xautopar

複数プロセッサのための自動並列化を有効にします。依存性の解析 (ループの繰り返し内部でのデータ依存性の解析) およびループ再構成を実行します。最適化が -xO3 以上でない場合、最適化は -xO3 に引き上げられ、警告が出されます。

独自のスレッド管理を行なっている場合には、-xautopar を使用しないでください。

より高速な実行を実現するには、このオプションにマルチプロセッサシステムが必要です。シングルプロセッサシステムでは、通常、生成されたバイナリの実行速度は低下します。

並列化されたプログラムをマルチスレッド環境で実行するには、実行前に環境変数 OMP_NUM_THREADS を 1 より大きな値に設定する必要があります。設定しない場合は、デフォルトは 2 です。より多くのスレッドを使用するには、OMP_NUM_THREADS をより高い値に設定します。1 つのスレッドだけで実行する場合は、OMP_NUM_THREADS を 1 に設定します。一般に、OMP_NUM_THREADS には、実行中のシステムで使用可能な仮想プロセッサ数を設定します。Oracle Solaris の psrinfo(1) コマンドを使用して判断できます。

-xautopar を使用してコンパイルとリンクを 1 度に実行する場合、リンクには自動的にマイクロタスキングライブラリおよびスレッドに対して安全な C 実行時ライブラリが含まれます。-xautopar を使用して別々にコンパイルし、リンクする場合、-xautopar でリンクする必要があります。

### A.2.106.1 関連項目

289 ページの「A.2.152 -xopenmp[= *i*]」

## A.2.107 -xbinopt={prepare| off}

(SPARC) このオプションは廃止されており、コンパイラの将来のリリースで削除される予定です。240 ページの「A.2.103 -xannotate[=yes| no]」 を参照してください。

コンパイラに、あとで最適化、変換、および解析を行うためにバイナリを準備するよう指示します。binopt(1) のマニュアルページを参照してください。このオプションは、実行可能ファイルまたは共有オブジェクトの構築に使用できます。コンパイルとリンクを別々に行う場合は、両方の手順に -xbinopt を指定する必要があります。

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

一部のソースコードがコンパイルに使用できない場合も、このオプションを使用してそのほかのコードがコンパイルされます。このとき、最終的なバイナリを作成するリンク手順で、このオプションを使用する必要があります。この場合、このオプションでコンパイルされたコードだけが最適化、変換、分析できます。

### A.2.107.1 デフォルト

デフォルトは -xbinopt=off です。

#### 相互の関連性

このオプションを有効にするには、最適化レベル -xO1 以上で使用する必要があります。このオプションを使用すると、構築したバイナリのサイズが少し増えます。

-xbinopt=prepare と -g を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。

## A.2.108 -xbuiltin[={ %all|%default|%none}]

標準ライブラリ呼び出しの最適化を有効または無効にします。

-xbuiltin オプションは、標準ライブラリ関数を呼び出すコードをさらに最適化するために使用します。このオプションを使用すると、コンパイラは、パフォーマンスに有益な場所で組み込み関数またはインラインシステム関数を置き換えることができます。コンパイラのコメント出力を読み取って、どの関数がコンパイラによって置き換えられたかを判定する方法については、er_src(1) のマニュアルページを参照してください。

-xbuiltin=%all を使用した場合、置き換えによって errno の設定が信頼できなくなることがあります。プログラムが errno の値に依存している場合、このオプションは避けてください。

-xbuiltin=%default では、errno を設定しない関数だけがインライン化されます。errno の値はどの最適化レベルでも常に正確であり、高い信頼度でチェックできます。-xbuiltin=%default を -xO3 以下で使用した場合、コンパイラはどの呼び出しがインライン化に有益かを判定し、それ以外はインライン化しません。

-xbuiltin=%none オプションはデフォルトのコンパイラの動作に影響を与え、コンパイラは組み込み関数に対して特別な最適化は行いません。

### A.2.108.1 デフォルト

-xbuiltin を指定しない場合、デフォルトは、-xO1 以上の最適化レベルでのコンパイル時は -xbuiltin=%default、-xO0 では -xbuiltin=%none です。-xbuiltin を引数なしで指定した場合、デフォルトは -xbuiltin=%all であり、コンパイラは組み込み関数の置き換えや標準ライブラリ関数のインライン化をはるかに積極的に行います。

-xbuiltin オプションでは、システムのヘッダーファイルで定義されている大域関数だけがインライン化され、ユーザーが定義した静的関数はインライン化されません。大域関数に割り込もうとするユーザーコードによって、定義されていない動作になることがあります。

### 相互の関連性

マクロ `-fast` の拡張には、`-xbuiltin=%all` が取り込まれます。

### 例

次のコンパイラコマンドでは、標準ライブラリ呼び出しを特殊処理するように要求します。

```
example% CC -xbuiltin -c foo.cc
```

次のコンパイラコマンドでは、標準ライブラリ呼び出しを特殊処理しないように要求します。マクロ `-fast` の拡張には `-xbuiltin=%all` が取り込まれていることに注意してください。

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

## A.2.109 -xcache=*c*

オプティマイザ用のキャッシュ特性を定義します。この定義によって、特定のキャッシュが使用されるわけではありません。

---

注 – このオプションは単独でも使用できますが、`-xtarget` オプションを展開したものの一部です。主に、`-xtarget` オプションで指定された値を上書きするために使用されます。

---

オプションのプロパティー `[/ti]` は、キャッシュを共有できるスレッドの数を設定します。

### A.2.109.1 値

*c* は、次の表に示されている値のいずれかである必要があります。

表 A–24 -xcache の値

| 値 | 意味 |
|---|---|
| `generic` | コンパイラに、どのプロセッサ上でも大きなパフォーマンス低下を招かず、ほとんどの x86 および SPARC プロセッサ上で良好なパフォーマンスが得られるキャッシュ属性を使用するよう指示します。(デフォルト)<br>これらの最高のタイミング特性は、新しいリリースごとに、必要に応じて調整されます。 |

表 A–24 -xcache の値 *(続き)*

| 値 | 意味 |
|---|---|
| `native` | ホスト環境に対して最適化されたパラメータを設定します。 |
| *s1*/*l1* /*a1*[/*t1*] | レベル 1 のキャッシュ属性を定義します。 |
| *s1*/*l1* /*a1*[/*t1*] :*s2*/*l2* /*a2*[/*t2*] | レベル 1 とレベル 2 のキャッシュ属性を定義します。 |
| *s1*/*l1* /*a1*[/*t1*] :*s2*/*l2* /*a2*[/*t2*] :*s3*/*l3* /*a3*[/*t3*] | レベル 1、レベル 2、レベル 3 のキャッシュ属性を定義します。 |

キャッシュ属性 s*i* /l*i*/a*i* /t*i* の定義を次の表に示します。

| 属性 | 定義 |
|---|---|
| s*i* | レベル *i* のデータキャッシュのサイズ (K バイト単位) |
| l*i* | レベル *i* のデータキャッシュのラインサイズ (バイト単位) |
| a*i* | レベル *i* のデータキャッシュの結合規則 |
| t*i* | レベル *i* でキャッシュを共有するハードウェアスレッドの数 |

たとえば、*i*=1 は、レベル 1 のキャッシュ属性の *s1*/*l1*/*a1* を意味します。

## デフォルト

`-xcache` が指定されていない場合は、デフォルトの `-xcache=generic` が使用されます。この値は、コンパイラに、どのプロセッサ上でも大きなパフォーマンス低下を招かず、ほとんどの SPARC プロセッサ上で良好なパフォーマンスが得られるキャッシュ属性を使用するよう指示します。

*t* の値を指定しない場合のデフォルトは 1 です。

## 例

`-xcache=16/32/4:1024/32/1` は、次の値を指定します。

レベル 1 のキャッシュ　16K バイト、32 バイトの行サイズ、4 ウェイアソシアティブ

レベル 2 のキャッシュ　1024K バイト、32 バイトの行サイズ、ダイレクトマップ結合

### 関連項目

`-xtarget=`*t*

## A.2.110 -xchar[= o]

この オプションは、 char 型が符号なしで定義されているシステムからのコード移植を簡単にするためのものです。そのようなシステムからの移植以外では、このオプションは使用しないでください。符号付きまたは符号なしであると明示的に示すように書き直す必要があるのは、符号に依存するコードだけです。

### A.2.110.1 値

次の表の値のいずれかを *o* に置き換えることができます。

表 A–25 -xchar の値

| 値 | 意味 |
|---|---|
| `signed` | char 型で定義された文字定数および変数を符号付きとして処理します。このオプションは、コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。 |
| `s` | `signed` と同義です。 |
| `unsigned` | char 型で定義された文字定数および変数を符号なしとして処理します。このオプションは、コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。 |
| `u` | `unsigned` と同等です |

### デフォルト

`-xchar` を指定しない場合は、コンパイラでは `-xchar=s` が指定されます。

`-xchar` を値なしで指定した場合は、コンパイラでは `-xchar=s` が指定されます。

### 相互の関連性

`-xchar` オプションは、`-xchar` でコンパイルしたコードでだけ、`char` 型の値の範囲を変更します。このオプションは、システムルーチンまたはヘッダーファイル内の `char` 型の値の範囲は変更されません。特に、`limits.h` で定義された `CHAR_MAX` と `CHAR_MIN` の値は、このオプションが指定されても変更されません。したがって、`CHAR_MAX` および `CHAR_MIN` は、通常の `char` で符号化可能な値の範囲は表示されなくなります。

### 警告

-xchar=unsigned を使用するときは、マクロでの値が符号付きの場合があるため、char を定義済みのシステムマクロと比較する際には特に注意してください。この状況は、マクロを通してアクセスされる、エラーコードを返すすべてのルーチンでもっとも一般的です。エラーコードは、一般的には負の値になっています。したがって、char をそのようなマクロによる値と比較するときは、結果は常に false になります。負の数値が符号なしの型の値と等しくなることはありません。

ライブラリを通してエクスポートされるインタフェースのルーチンをコンパイルするために、-xchar を使用しないでください。Oracle Solaris ABI は char 型を符号付きとして指定し、それに応じてシステムライブラリが動作します。char を符号なしにする影響は、システムライブラリで十分にテストされていません。このオプションを使用する代わりに、char 型の符号の有無に依存しないようにコードを変更してください。char 型の符号は、コンパイラやオペレーティングシステムの間でさまざまに変化します。

## A.2.111 -xcheck[= i]

-xcheck=stkovf を使用してコンパイルすると、シングルスレッドプログラム内のメインスレッドや、マルチスレッドプログラム内のスレーブスレッドスタックのスタックオーバーフローのための実行時チェックが追加されます。スタックオーバーフローが検出された場合は、SIGSEGV が生成されます。スタックオーバーフローによって発生した SIGSEGV をほかのアドレス空間違反と異なる方法で処理する方法については、sigaltstack(2) のマニュアルページを参照してください。

### A.2.111.1 値

*i* は、次の表に示されている値のいずれかである必要があります。

表 A–26 -xcheck の値

| 値 | 意味 |
|---|---|
| %all | チェックをすべて実行します。 |
| %none | チェックを実行しません。 |
| stkovf | スタックオーバーフローのチェックをオンにします。 |
| no%stkovf | スタックオーバーフローのチェックをオフにします。 |
| init_local | ローカル変数を初期化します。詳細は、『C ユーザーズガイド』を参照してください。 |
| no%init_local | 局所変数を初期化しません (デフォルト)。 |

### デフォルト

-xcheck を指定しない場合は、コンパイラではデフォルトで -xcheck=%none が指定されます。

引数を指定せずに -xcheck を使用した場合は、コンパイラではデフォルトで -xcheck=%none が指定されます。

-xcheck オプションは、コマンド行で累積されません。コンパイラは、コマンドで最後に指定したものに従ってフラグを設定します。

## A.2.112 -xchip=*c*

オプティマイザが使用するターゲットとなるプロセッサを指定します。

–xchip オプションは、ターゲットとなるプロセッサを指定してタイミング属性を指定します。このオプションは、次の属性に影響を与えます。

- 命令の順序 (スケジューリング)
- コンパイラが分岐を使用する方法
- 意味が同じもので代用できる場合に使用する命令

---

注 – このオプションは単独でも使用できますが、-xtarget オプションを展開したものの一部です。主に、-xtarget オプションで指定された値を上書きするために使用されます。

---

### A.2.112.1 値

*c* は、次の 2 つの表に示されている値のいずれかである必要があります。

表 A–27 SPARC プロセッサでの -xchip の値

| | |
|---|---|
| generic | ほとんどの SPARC プロセッサ上での良好なパフォーマンス |
| native | コンパイラが実行されているホスト SPARC システム上での良好なパフォーマンス |
| sparc64vi | SPARC64 VI プロセッサ |
| sparc64vii | SPARC64 VII プロセッサ |
| sparc64viiplus | SPARC64 VII+ プロセッサ |
| ultra | UltraSPARC プロセッサ |
| ultra2 | UltraSPARC II プロセッサ |
| ultra2e | UltraSPARC IIe プロセッサ |

表 A–27 SPARC プロセッサでの -xchip の値 *(続き)*

| | |
|---|---|
| `ultra2i` | UltraSPARC IIi プロセッサ |
| `ultra3` | UltraSPARC III プロセッサ |
| `ultra3cu` | UltraSPARC III Cu プロセッサ |
| `ultra3i` | UltraSparc IIIi プロセッサ |
| `ultra4` | UltraSPARC IV プロセッサ |
| `ultra4plus` | UltraSPARC IVplus プロセッサ |
| `ultraT1` | UltraSPARC T1 プロセッサ. |
| `ultraT2` | UltraSPARC T2 プロセッサ |
| `ultraT2plus` | UltraSPARC T2+ プロセッサ |
| `T3` | SPARC T3 プロセッサ |
| `T4` | SPARC T4 プロセッサ |

表 A–28 x86/x64 プロセッサでの -xchip の値

| | |
|---|---|
| `generic` | ほとんどの x86 プロセッサ上での良好なパフォーマンス |
| native | コンパイラが実行されているホスト x86 システム上での良好なパフォーマンス |
| `core2` | Intel Core2 プロセッサ |
| `nehalem` | Intel Nehalem プロセッサ |
| `opteron` | AMD Opteron プロセッサ |
| `penryn` | Intel Penryn プロセッサ |
| `pentium` | Intel Pentium プロセッサ |
| `pentium_pro` | Intel Pentium Pro プロセッサ |
| `pentium3` | Intel Pentium 3 形式プロセッサ |
| `pentium4` | Intel Pentium 4 形式プロセッサ |
| `amdfam10` | AMD AMDFAM10 プロセッサ |
| `sandybridge` | Intel Sandy Bridge プロセッサ |
| `westmere` | Intel Westmere プロセッサ |

### デフォルト

ほとんどのプロセッサ上で、generic は、どのプロセッサでもパフォーマンスの著しい低下がなく、良好なパフォーマンスが得られる最良のタイミング属性を使用するようコンパイラに命令するデフォルト値です。

## A.2.113 -xcode=*a*

(*SPARC* のみ ) コードのアドレス空間を指定します。

---

注 – 共有オブジェクトの構築では、-xcode=pic13 または -xcode=pic32 を指定することをお勧めします。pic13 または pic32 なしに構築された共有オブジェクトは、正しく機能せず、構築できない可能性があります。

---

### A.2.113.1 値

*a* は、次の表に示されている値のいずれかである必要があります。

表 A–29 -xcode の値

| 値 | 意味 |
|---|---|
| abs32 | 32 ビット絶対アドレスを生成します。高速ですが、範囲が制限されます。コード、データ、および bss を合計したサイズは 2**32 バイトに制限されます。 |
| abs44 | SPARC: 44 ビット絶対アドレスを生成します。中程度の速さで中程度の範囲を使用できます。コード、データ、および bss を合計したサイズは 2**44 バイトに制限されます。64 ビットのアーキテクチャーでのみ利用できます。動的 (共有) ライブラリで使用しないでください。 |
| abs64 | SPARC: 64 ビット絶対アドレスを生成します。低速ですが、範囲は制限されません。64 ビットのアーキテクチャーでのみ利用できます。 |
| pic13 | 位置独立コード (小規模モデル) を生成します。高速ですが、範囲が制限されます。-Kpic と同等です。32 ビットアーキテクチャーでは最大 2**11 個の固有の外部シンボルを、64 ビットでは 2**10 個の固有の外部シンボルをそれぞれ参照できます。 |
| pic32 | 位置独立コード (ラージモデル) を生成します。pic13 ほど高速でない可能性がありますが、フルレンジ対応です。-KPIC と同等です。32 ビットアーキテクチャーでは最大 2**30 個の固有の外部シンボルを、64 ビットでは 2**29 個の固有の外部シンボルをそれぞれ参照できます。 |

-xcode=pic13 と -xcode=pic32 のどちらを使用するかを判定するには、elfdump -c を使用して大域オフセットテーブル (GOT) のサイズを確認し、セクションヘッダー sh_name: .got を検索します。sh_size 値が GOT のサイズです。GOT のサイズが 8,192 バイトに満たない場合は -xcode=pic13、そうでない場合は -xcode=pic32 を指定します。詳細は、elfdump(1) のマニュアルページを参照してください。

一般に、-xcode の使用方法の決定に際しては、次のガイドラインに従ってください。

- 実行可能ファイルを構築する場合は、-xcode=pic13 と -xcode=pic32 のどちらも使わない。
- 実行可能ファイルへのリンク専用のアーカイブライブラリを構築する場合は、-xcode=pic13 と -xcode=pic32 のどちらも使わない。
- 共有ライブラリを構築している場合は、-xcode=pic13 から始めてください。GOT のサイズが 8,192 バイトを超えたら、-xcode=pic32 を使用します。
- 共有ライブラリへのリンクのためのアーカイブライブラリを構築している場合は、-xcode=pic32 のみを使用してください。

## デフォルト

32 ビットアーキテクチャーの場合は -xcode=abs32 です。64 ビットアーキテクチャーの場合のデフォルトは -xcode=abs44 です。

共有動的ライブラリを作成する場合、64 ビットアーキテクチャーでは -xcode のデフォルト値である abs44 と abs32 を使用できません。-xcode=pic13 または -xcode=pic32 を指定してください。SPARC の場合、-xcode=pic13 および -xcode=pic32 では、わずかですが、次の 2 つのパフォーマンス上の負担がかかります。

- -xcode=pic13 および -xcode=pic32 のいずれかでコンパイルしたルーチンは、共有ライブラリの大域または静的変数へのアクセスに使用されるテーブル (_GLOBAL_OFFSET_TABLE_) を指し示すようレジスタを設定するために、入口で命令を数個余計に実行します。
- 大域または静的変数へのアクセスのたびに、_GLOBAL_OFFSET_TABLE_ を使用した間接メモリー参照が 1 回余計に行われます。-xcode=pic32 でコンパイルした場合は、大域および静的変数への参照ごとに命令が 2 個増えます。

こうした負担があるとしても、-xcode=pic13 および -xcode=pic32 を使用すると、ライブラリコードを共有できるため、必要となるシステムメモリーを大幅に減らすことができます。-xcode=pic13 あるいは -xcode=pic32 でコンパイルした共有ライブラリを使用するすべてのプロセスは、そのライブラリのすべてのコードを共有できます。共有ライブラリ内のコードに非 pic (すなわち、絶対) メモリー参照が 1 つでも含まれている場合、そのコードは共有不可になるため、そのライブラリを使用するプログラムを実行する場合は、その都度、コードのコピーを作成する必要があります。

.o ファイルが -xcode=pic13 と -xcode=pic32 のどちらを使用してコンパイルされたかを知るためのもっとも簡単な方法は、nm コマンドの使用です。

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

位置独立コードを含む .o ファイルには、_GLOBAL_OFFSET_TABLE_ への未解決の外部参照が含まれます。このことは、英文字の U で示されます。

-xcode=pic13 または -xcode=pic32 を使用すべきかどうかを判断するには、nm を使用して、共有ライブラリで使用または定義されている明確な大域および静的変数の個数を確認します。_GLOBAL_OFFSET_TABLE_ のサイズが 8,192 バイトより小さい場合は、-Kpic を使用できます。そうでない場合は、-xcode=pic32 を使用する必要があります。

## A.2.114 -xdebugformat=[stabs|dwarf]

コンパイラは、デバッガ情報の形式をスタブ (「シンボルテーブル」) 形式から「DWARF Debugging Information Format」仕様の *dwarf* 形式に移行しました。デフォルト設定は -xdebugformat=dwarf です。

デバッグ情報を読み取るソフトウェアを保守している場合は、今回からそのようなツールを stab 形式から dwarf 形式へ移行するためのオプションが加わりました。

このオプションは、ツールを移植する場合に新しい形式を使用する方法として使用してください。デバッガ情報を読み取るソフトウェアを保守しているか、または特定のツールがこれらのいずれかの形式のデバッガ情報を必要としていないかぎり、このオプションを使用する必要はありません。

表 A–30 -xdebugformat のフラグ

| 値 | 意味 |
|---|---|
| stabs | -xdebugformat=stabs は、stab 標準形式を使用してデバッグ情報を生成します。 |
| dwarf | -xdebugformat=dwarf は、dwarf 標準形式を使用してデバッグ情報を生成します。 |

-xdebugformat を指定しない場合は、コンパイラでは -xdebugformat=dwarf が指定されます。このオプションには引数が必要です。

このオプションは、-g オプションによって記録されるデータの形式に影響します。-g を指定しなくても、一部のデバッグ情報が記録されますが、その情報の形式もこのオプションによって制御されます。したがって、-g が使用されていないときでも、-xdebugformat は有効です。

dbx とパフォーマンスアナライザソフトウェアは、stab 形式と dwarf 形式を両方とも認識するので、このオプションを使用しても、ツールの機能にはまったく影響を与えません。

---

注 – スタブ形式では、dbx で現在使用されているすべてのデバッグデータを表せません。また、一部のコードは、スタブを使用してデバッグデータを正常に生成できない可能性があります。

---

詳細は、dumpstabs(1) および dwarfdump(1) のマニュアルページも参照してください。

## A.2.115 -xdepend=[yes| no]

ループの繰り返し内部でのデータ依存性を解析し、ループ交換、ループ融合、スカラー交換、「デッドアレイ」代入の回避などのループの再構成を実行します。

SPARC プロセッサ上では、-xdepend はデフォルトで、-xO3 以上のすべての最適化レベルを示す -xdepend=on になります。それ以外の場合は、–xdepend のデフォルトは –xdepend=off です。-xdepend の明示的な設定を指定すると、すべてのデフォルト設定は上書きされます。

x86 プロセッサ上では、-xdepend はデフォルトで -xdepend=off になります。-xdepend を指定し、最適化が -xO3 以上でない場合は、コンパイラは最適化を -xO3 に上げ、警告を発行します。

引数なしで -xdepend を指定すると、-xdepend=yes と同等であることを意味します。

依存性の解析は -xautopar に含まれています。依存性の解析はコンパイル時に実行されます。

依存性の解析はシングルプロセッサシステムで役立つことがあります。ただし、シングルプロセッサシステム上で -xdepend を使用する場合は、-xautopar も同時に指定してはいけません。マルチプロセッサシステムに対して -xdepend の最適化が実行されてしまうためです。

### A.2.115.1 関連項目

-xprefetch_auto_type

## A.2.116 -xdumpmacros[= value[,value...]]

マクロがプログラム内でどのように動作しているかを調べたいときに、このオプションを使用します。このオプションは、定義済みマクロ、解除済みマクロ、実際の使用状況といった情報を提供します。マクロの処理順序に基づいて、標準エラー

(stderr) への出力が出力されます。-xdumpmacros オプションは、ファイルの終わりまで、または dumpmacros プラグマまたは end_dumpmacros プラグマによって上書きされるまで有効です。332 ページの「B.2.5 #pragma dumpmacros」 を参照してください。

### A.2.116.1 値

次の表に、*value* の有効な引数の一覧を示します。接頭辞 no% は関連付けられた値を無効にします。

表 A–31 -xdumpmacros の値

| 値 | 意味 |
|---|---|
| [no%]defs | すべての定義済みマクロを出力します。 |
| [no%]undefs | すべての解除済みマクロを出力します。 |
| [no%]use | 使用されているマクロの情報を出力します |
| [no%]loc | defs、undefs、use の位置 (パス名と行番号) も出力します。 |
| [no%]conds | 条件付き指令で使用されたマクロの使用情報を出力します。 |
| [no%]sys | システムヘッダーファイル内のマクロについて、すべての定義済みマクロ、解除済みマクロ、使用状況を出力します。 |
| %all | オプションを-xdumpmacros=defs,undefs,use,loc,conds,sys に設定します。この引数は、[no%] 形式の引数と併用すると効果的です。たとえば -xdumpmacros=%all,no%sys は、出力からシステムヘッダーマクロを除外しますが、そのほかのマクロに関する情報は依然として出力します。 |
| %none | あらゆるマクロ情報を出力しません。 |

オプションの値は累積されるので、-xdumpmacros=sys -xdumpmacros=undefs を指定することは、-xdumpmacros=undefs,sys と同じ効果があります。

---

注 – サブオプション loc、conds、sys は、オプション defs、undefs、use の修飾子です。loc、conds、sys は、単独では効果はありません。たとえば -xdumpmacros=loc,conds,sys は、まったく効果を持ちません。

---

#### デフォルト

引数なしで -xdumpmacros を指定するときのデフォルトは、-xdumpmacros=defs,undefs,sys です。-xdumpmacros を指定しないときのデフォルトは、-xdumpmacros=%none です。

## 例

-xdumpmacros=use,no%loc オプションを使用すると、使用した各マクロの名前が一度だけ出力されます。より詳しい情報が必要であれば、-xdumpmacros=use,loc オプションを使用します。マクロを使用するたびに、そのマクロの名前と位置が印刷されます。

次のファイル t.c を考慮します。

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b) a+b
#else
#define COMPUTE(a,b) a-b
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

次の例は、defs、undefs、sys、および loc の引数に基づいた、ファイル t.c の出力を示しています。

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_9 1
#define __SUNPRO_CC 0x590
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b

example% CC -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
command line: #define __SunOS_5_9 1
command line: #define __SUNPRO_CC 0x590
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUNPRO_CC_COMPAT 5
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
```

```
command line: #undef BAR
t.c, line 2: #undef FOO
t.c, line 3: #define COMPUTE(a, b) a + b
```

次の例では、use、loc、および conds の引数によって、マクロ動作がファイル t.c に出力されます。

```
example% CC -c -xdumpmacros=use t.c
used macro COMPUTE

example% CC -c -xdumpmacros=use,loc t.c
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE

example% CC -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
used macro MM

example% CC -c -xdumpmacros=use,conds,loc t.c
t.c, line 1: used macro FOO
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
t.c, line 9: used macro COMPUTE
t.c, line 9: used macro NN
t.c, line 9: used macro MM
```

次は、ファイル y.c の例です。

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

次の例は、y.c 内のマクロに基づく、-xdumpmacros=use,loc からの出力を示しています。

```
example% CC -c -xdumpmacros=use,loc y.c
y.c, line 4: used macro Z
y.c, line 4: used macro Y
y.c, line 4: used macro X
```

### 関連項目

プラグマ dumpmacros/end_dumpmacros は、-xdumpmacros コマンド行オプションのスコープより優先されます。

## A.2.117 -xe

構文エラーと意味エラーの有無チェックのみ行います。-xe を指定すると、オブジェクトコードは出力されません。-xe の出力は、stderr に送られます。

コンパイルによってオブジェクトファイルを生成する必要がない場合には、-xe オプションを使用してください。たとえば、コードの一部を削除することによってエラーメッセージの原因を切り分ける場合には、-xe を使用することによって編集とコンパイルを高速化できます。

### A.2.117.1 関連項目

-c

## A.2.118 -xF[=v[, v...]]

リンカーによる関数と変数の最適な順序の並べ替えを有効にします。

このオプションは、コンパイラに、関数やデータ変数を細分化された別々のセクションに配置するよう指示します。これにより、リンカーは、リンカーの -M オプションで指定されたマップファイル内の指示を使用してこれらのセクションを並べ替えることにより、プログラムのパフォーマンスを最適化できるようになります。通常は、この最適化によって効果が上がるのは、プログラムの実行時間の多くがページフォルト時間に費やされている場合だけです。

変数を並べ替えると、実行時のパフォーマンスに悪影響を与える次の問題の解決に役立ちます。

- メモリー内で互いに近い位置にある関連性のない変数によって発生するキャッシュやページの競合
- メモリー内で互いに近い位置にない関連性のある変数によって発生する、必要以上に大きな作業セットサイズ
- 有効なデータ密度を低下させる weak 変数の未使用のコピーによって発生する、必要以上に大きな作業セットサイズ

最適なパフォーマンスを得るために変数と関数の順序を並べ替えるには、次の処理が必要です。

1. -xF によるコンパイルとリンク
2. 『パフォーマンスアナライザ』のマニュアルにある関数のマップファイルを生成する方法に関する指示に従うか、または『リンカーとライブラリ』にあるデータのマップファイルを生成する方法に関する指示に従います。
3. リンカーの -M オプションを使用して新しいマップファイルを再リンクします。
4. アナライザで再実行して、パフォーマンスが向上したかどうかを検証します。

### A.2.118.1 値

*v* には、次の表に示されている値の 1 つ以上を指定できます。no% 接頭辞によって、関連付けられた値が無効になります。

表 A–32 -xF の値

| 値 | 意味 |
|---|---|
| [no%]func | 関数を個別のセクションにフラグメント化します。 |
| [no%]gbldata | 大域データ (外部リンケージを持つ変数) を個別のセクションにフラグメント化します。 |
| [no%]lcldata | ローカルデータ (内部リンケージを持つ変数) を個別のセクションにフラグメント化します。 |
| %all | 関数、大域データ、局所データを細分化します。 |
| %none | 何も細分化しません。 |

### デフォルト

-xF を指定しない場合のデフォルトは、-xF=%none です。引数を指定しないで -xF を指定した場合のデフォルトは、-xF=%none,func です。

### 相互の関連性

-xF=lcldata を指定するとアドレス計算最適化が一部禁止されるので、このフラグは実験として意味があるときにだけ使用するとよいでしょう。

### 関連項目

analyzer(1) および ld(1) のマニュアルページ

## A.2.119 -xhelp=flags

各コンパイラオプションの簡単な説明を表示します。

## A.2.120 -xhwcprof

(*SPARC* のみ) ハードウェアカウンタによるプロファイリングのコンパイラでのサポートを有効にします。

-xhwcprof を有効にすると、コンパイラは、プロファイル対象のロード命令およびストア命令と、それらが参照するデータ型および構造体メンバーをツールが関連付けるのに役立つ情報を、-g で生成されたシンボル情報と組み合わせて生成します。プロファイルデータをターゲットの命令領域ではなく、データ領域に関連付けます。このオプションは、命令プロファイリングだけでは簡単には得られない動作に対する見通しを提供します。

指定した一連のオブジェクトファイルは、-xhwcprof を指定してコンパイルできます。ただし、-xhwcprof がもっとも有効なのは、アプリケーション内のすべてのオブジェクトファイルに適用された場合です。この場合は、アプリケーションのオブジェクトファイルに分散しているすべてのメモリー参照の識別や関連付けが完全に対処されます。

コンパイルとリンクを別々に行う場合は、-xhwcprof をリンク時にも使用してください。将来 -xhwcprof に拡張する場合は、リンク時に -xhwcprof を使用する必要があります。

-xhwcprof=enable または -xhwcprof=disable のインスタンスは、同じコマンド行にある以前の -xhwcprof のインスタンスをすべて無効にします。

-xhwcprof はデフォルトでは無効です。引数を指定せずに -xhwcprof と指定することは、-xhwcprof=enable と指定することと同じです。

-xhwcprof を指定するには、最適化を有効にして、DWARF のデバッグデータ形式を選択する必要があります。現在は、DWARF 形式 ( -xdebugformat=dwarf) がデフォルトです。

-xhwcprof と -g を組み合わせると、コンパイラの一時ファイル記憶領域の必要量は、-xhwcprof と -g のいずれかを指定したときに増える量の合計よりも増えます。

次のコマンドは example.cc をコンパイルし、ハードウェアカウンタによるプロファイリングのサポートを指定し、DWARF シンボルを使用してデータ型と構造体メンバーのシンボリック解析を指定します。

```
example% CC -c -O -xhwcprof -g -xdebugformat=dwarf example.cc
```

ハードウェアカウンタによるプロファイリングについての詳細は、『パフォーマンスアナライザ』のマニュアルを参照してください。

## A.2.121 -xia

区間演算ライブラリをリンクし、適切な浮動小数点環境を設定します。

---

注 – C++ 区間演算ライブラリは、Fortran コンパイラで実装されているとおり、区間演算と互換性があります。

---

x86 プラットフォーム上では、このオプションを指定するには SSE2 命令セットのサポートが必要です。

### A.2.121.1 展開

-xia オプションは、-fsimple=0 -ftrap=%none -fns=no -library=interval に拡張するマクロです。区間演算を使用するようにしていて、-fsimple か -ftrap、-fns、-library のどれかを指定して -xia による設定を無効にした場合、コンパイラが不正な動作をすることがあります。

### A.2.121.2 相互の関連性

区間演算ライブラリを使用するには、<suninterval.h> を取り込みます。

区間演算ライブラリを使用する場合は、Cstd と iostreams のいずれかのライブラリを含める必要があります。これらのライブラリを含める方法については、-library を参照してください。

### A.2.121.3 警告

区間を使用し、-fsimple、-ftrap、または -fns に異なる値を指定すると、プログラムの動作が不正確になる場合があります。

C++ 区間演算は実験に基づくもので発展性があります。詳細は、リリースごとに変更される可能性があります。

### A.2.121.4 関連項目

-library

## A.2.122 -xinline[= func-spec[,func-spec...]]

どのユーザー作成ルーチンをオプティマイザによって -xO3 レベル以上でインライン化するかを指定します。

### A.2.122.1 値

*func-spec* は、次の表に示されている値のいずれかである必要があります。

表 A–33 -xinline の値

| 値 | 意味 |
|---|---|
| %auto | 最適化レベル -xO4 以上で自動インライン化を有効にします。この引数は、オプティマイザが選択した関数をインライン化できることをオプティマイザに知らせます。%auto の指定がない場合、-xinline=[no%] *func-name...* によってコマンド行で明示的インライン化が指定されていると、自動インライン化は通常無効になります。 |

表 A–33 -xinline の値 (続き)

| 値 | 意味 |
|---|---|
| *func_name* | オプティマイザに関数をインライン化するように強く要求します。関数が extern "C" で宣言されていない場合は、*func_name* の値を符号化する必要があります。実行可能ファイルに対し nm コマンドを使用して符号化された関数名を検索できます。extern "C" で宣言された関数の場合は、名前はコンパイラで符号化されません。 |
| no%*func_name* | リスト上のルーチン名の前に no% を付けると、そのルーチンのインライン化が禁止されます。*func-name* の符号化された名前に関する規則は、no%*func-name* にも適用されます。 |

-xipo [=1|2] を使用しないかぎり、コンパイルされているファイルのルーチンだけがインライン化の対象とみなされます。オプティマイザでは、どのルーチンがインライン化に適しているかを判断します。

### A.2.122.2 デフォルト

-xinline オプションを指定しないと、コンパイラでは -xinline=%auto が使用されます。

-xinline= が引数なしで指定されている場合は、最適化レベルには関係なく、どの関数もインライン化されません。

### A.2.122.3 例

自動インライン化を有効にしながら、int foo() が宣言されている関数のインライン化を無効にするには、次のコマンドを使用します。

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

int foo() として宣言された関数のインライン化を強く要求し、その他のすべての関数をインライン化の候補にするには、次のコマンドを使用します。

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

int foo() として宣言された関数のインライン化を強く要求し、ほかのどの関数のインライン化も許可しないようにするには、次のコマンドを使用します。

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

### A.2.122.4 相互の関連性

-xinline オプションは -xO3 未満の最適化レベルには影響を与えません。-xO4 以上では、-xinline オプションを指定しなくてもオプティマイザでどの関数をインライン化する必要があるかを判断します。-xO4 では、コンパイラはどの関数が、インライン化されたときにパフォーマンスを改善するかを判断しようとします。

ルーチンは、次のいずれかの条件が当てはまる場合はインライン化されます。

- 最適化は -xO3 以上
- インライン化に効果があって安全
- コンパイル中のファイルの中に関数がある、または-xipo[=1|2] を使用してコンパイルしたファイルの中に関数がある

### A.2.122.5 警告

-xinline を指定して関数のインライン化を強制すると、実際にパフォーマンスを低下させる可能性があります。

### A.2.122.6 関連項目

274 ページの「A.2.130 -xldscope={*v*}」

## A.2.123 -xinstrument=[ no%]datarace

スレッドアナライザで分析するためにプログラムをコンパイルして計測するには、このオプションを指定します。スレッドアナライザの詳細は、tha(1) のマニュアルページを参照してください。

そうすることで、パフォーマンスアナライザを使用して計測されたプログラムを collect -r races で実行し、データ競合の検出実験を行うことができます。計測されたコードをスタンドアロンで実行できますが、低速で実行されます。

-xinstrument=no%datarace を指定して、スレッドアナライザ用のソースコードの準備をオフにすることができます。これはデフォルト値です。

-xinstrument を引数なしで指定することはできません。

コンパイルとリンクを別々に行う場合は、コンパイル時とリンク時の両方で -xinstrument=datarace を指定する必要があります。

このオプションは、プリプロセッサトークン __THA_NOTIFY を定義します。#ifdef __THA_NOTIFY を指定して、libtha(3) ルーチンへの呼び出しを保護できます。

このオプションにも、-g を設定します。

## A.2.124 -xipo[={0|1|2}]

内部手続きの最適化を実行します。

-xipo オプションが内部手続き解析パスを呼び出すことでプログラムの一部の最適化を実行します。このオプションを指定すると、リンク手順でのすべてのオブジェクトファイル間の最適化を行い、しかもこれらの最適化は単にコンパイルコマンドのソースファイルにとどまりません。ただし、-xipo によるプログラム全体の最適化には、アセンブリ (.s) ソースファイルは含まれません。

-xipo オプションは、大量のファイルを使用してアプリケーションをコンパイルしてリンクするときに特に便利です。このフラグを指定してコンパイルされたオブジェクトファイルには、ソースプログラムファイルとコンパイル済みプログラムファイル間で内部手続き解析を有効にする解析情報が含まれています。ただし、解析と最適化は、-xipo を指定してコンパイルされたオブジェクトファイルに限定され、オブジェクトファイルまたはライブラリは対象外です。

### A.2.124.1 値

-xipo オプションには、次の表に示されている値を指定できます。

表 A–34 -xipo の値

| 値 | 意味 |
|---|---|
| 0 | 内部手続きの最適化を実行しません |
| 1 | 内部手続きの最適化を実行します |
| 2 | 内部手続きの別名解析と、メモリーの割り当ておよび配置の最適化を実行し、キャッシュ性能を向上します |

### A.2.124.2 デフォルト

-xipo を指定しないと、-xipo=0 が使用されます。

-xipoだけを指定すると、-xipo=1 が使用されます。

### A.2.124.3 例

次の例では同じ手順でコンパイルしてリンクします。

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

オプティマイザは、最後のリンク手順で 3 つのすべてのソースファイルにわたるファイル間のインライン化を実行します。ソースファイルのコンパイルのすべてを 1 回のコンパイルで実行する必要はなく、それぞれに -xipo オプションが指定された、複数回の個別のコンパイルにより実行できます。

次の例では別々の手順でコンパイルしてリンクします。

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog part1.o part2.o part3.o
```

コンパイルステップで作成されるオブジェクトファイルは、それらのファイル内でコンパイルされる追加の分析情報を保持します。そのため、リンクステップにおいてファイル相互の最適化を実行できます。

### A.2.124.4 -xipo= を使用しない内部手続き解析を行う場合

内部手続き解析では、コンパイラは、リンクステップでオブジェクトファイル群を操作しながら、プログラム全体の解析と最適化を試みます。コンパイラは、この一連のオブジェクトファイルで定義されているすべての `foo()` 関数またはサブルーチンについて、次の 2 つのことを前提にします。

- 実行時、このオブジェクトファイル群の外部で定義されている別のルーチンによって `foo()` が明示的に呼び出されない。
- オブジェクトファイル群内のルーチンからの `foo()` 呼び出しが、そのオブジェクトファイル群の外部に定義されている別のバージョンの `foo()` によって置き換えられることがない。

特定のアプリケーションについて最初の前提が当てはまらない場合は、`-xipo=2` を使用してコンパイルしないでください。

2 つ目の前提が当てはまらない場合は、`-xipo=1` と `-xipo=2` のどちらでもコンパイルしないでください。

1 例として、独自のバージョンの malloc() で関数 `malloc()` を置き換え、`-xipo=2` を指定してコンパイルするケースを考えてみましょう。独自のコードとリンクされる、`malloc()` を参照しているライブラリ内の関数はすべて、`-xipo=2` も指定してコンパイルされる必要があり、かつそれらのオブジェクトファイルがリンク手順に関与している必要があります。この方法はシステムライブラリには使用できない可能性があるため、独自のバージョンの `malloc()` を `-xipo=2` を使用してコンパイルしないでください。

もう 1 つの例として、別々のソースファイルにある `foo()` および `bar()` という 2 つの外部呼び出しを含む共有ライブラリを構築するケースを考えてみましょう。また、`bar()` は `foo()` を呼び出すと仮定します。`foo()` が実行時に割り込まれる可能性がある場合は、`foo()` または `bar()` のソースファイルを `-xipo=1` または `-xipo=2` を使用してコンパイルしないでください。それ以外の場合、`foo()` が `bar()` にインライン化され、それによって正しくない結果になる可能性があります。

### A.2.124.5 相互の関連性

`-xipo` オプションでは最低でも最適化レベル `-x04` が必要です。

### A.2.124.6 警告

コンパイルとリンクを個別に実行する場合、`-xipo` をコンパイルとリンクの両方で指定しなければなりません。

-xipo なしでコンパイルされたオブジェクトは、-xipo でコンパイルされたオブジェクトと自由にリンクできます。

次の例に示すように、ライブラリは、-xipo を使用してコンパイルされている場合でもファイル間の内部手続き解析に関与しません。

```
example% CC -xipo -xO4 one.cc two.cc three.cc
example% CC -xar -o mylib.a one.o two.o three.o
...
example% CC -xipo -xO4 -o myprog main.cc four.cc mylib.a
```

この例では、内部手続き最適化は one.cc、two.cc、および three.cc 間と main.cc と four.cc 間で実行されますが、main.cc または four.cc と mylib.a のルーチン間では実行されません。最初のコンパイルは未定義のシンボルに関する警告を生成する場合がありますが、相互手続きの最適化は、コンパイル手順でありしかもリンク手順であるために実行されます。

-xipo オプションは、ファイルを介して最適化を実行する際に必要な情報を追加するため、非常に大きなオブジェクトファイルを生成します。ただし、この補足情報は最終的な実行可能バイナリファイルの一部にはなりません。実行可能プログラムのサイズの増加は、そのほかに最適化を実行したことに起因します。

### A.2.124.7 関連項目

-xjobs

## A.2.125 -xipo_archive=[a]

新しい -xipo_archive オプションは、実行可能ファイルを生成する前に、-xipo を付けてコンパイルされ、アーカイブライブラリ (.a) の中に存在しているオブジェクトファイルを使用してリンカーに渡すオブジェクトファイルを最適化します。コンパイル中に最適化されたライブラリに含まれるオブジェクトファイルはすべて、その最適化されたバージョンに置き換えられます。

次の表に、*a* に指定可能な値を示します。

表 A–35 -xipo_archive のフラグ

| 値 | 意味 |
|---|---|
| writeback | 実行可能ファイルを生成する前に、アーカイブライブラリ (.a) に存在する -xipo でコンパイルしたオブジェクトファイルを使ってリンカーに渡すオブジェクトファイルを最適化します。コンパイル中に最適化されたライブラリに含まれるオブジェクトファイルは、すべてその最適化されたバージョンに置き換えられます。<br><br>アーカイブライブラリの共通セットを使用する並列リンクでは、最適化されるアーカイブライブラリの独自のコピーを、各リンクでリンク前に作成する必要があります。 |
| readonly | 実行可能ファイルを生成する前に、アーカイブライブラリ (.a) に存在する -xipo でコンパイルしたオブジェクトファイルを使ってリンカーに渡すオブジェクトファイルを最適化します。<br><br>-xipo_archive=readonly オプションを指定すると、リンク時に指定されたアーカイブライブラリのオブジェクトファイルで、モジュール間のインライン化と内部手続きデータフロー解析が有効になります。ただし、モジュール間インライン化によってほかのモジュールに挿入されたコードを除く、アーカイブライブラリのコードのモジュール間最適化は有効になりません。<br><br>アーカイブライブラリ内のコードにモジュール間の最適化を適用するには、-xipo_archive=writeback を指定する必要があります。この設定によって、コードが抽出された元のアーカイブライブラリの内容が変更されます。 |
| none | これはデフォルト値です。アーカイブファイルの処理は行いません。コンパイラは、モジュール間のインライン化やその他のモジュール間の最適化を、-xipo を使用してコンパイルされ、リンク時にアーカイブライブラリから抽出されたオブジェクトファイルに適用しません。これを行うには、-xipo と、-xipo_archive=readonly または -xipo_archive=writeback のいずれかをリンク時に指定する必要があります。 |

-xipo_archive の値が指定されない場合、-xipo_archive=none に設定されます。

-xipo_archive をフラグなしで指定することはできません。

## A.2.126 -xivdep[= p]

#pragma ivdep プラグマの解釈を無効化または設定します (ベクトル依存を無視)。

ivdep プラグマは、最適化の目的でループ内で検出された、配列参照へのループがもたらす依存関係の一部またはすべてを無視するようにコンパイラに指示します。これにより、指定しない場合には実行不可能なマイクロベクトル化、分散、ソフト

ウェアパイプライン化などの各種ループ最適化を、コンパイラが実行できるようになります。これは、依存関係が重要ではない、または依存関係が実際に発生しないことをユーザーが把握している場合に使用されます。

#pragma ivdep 指令の解釈は、–xivdep オプションの値に応じて異なります。

次のリストは、*p* の値とそれらの意味です。

| | |
|---|---|
| loop | ループキャリーのベクトル依存と想定されるものを無視 |
| loop_any | ループキャリーのベクトル依存をすべて無視 |
| back | 逆方向のループキャリーのベクトル依存と想定されるものを無視 |
| back_any | 逆方向のループキャリーのベクトル依存をすべて無視 |
| none | 依存を無視しない (ivdep プラグマの無効化) |

これらの解釈は、ほかのベンダーの ivdep プラグマの解釈との互換性のために提供されます。

## A.2.127 -xjobs=n

コンパイラが処理を行うために生成するプロセスの数を設定するには、-xjobs オプションを指定します。このオプションを使用すると、マルチプロセッサマシン上での構築時間を短縮できます。現時点では、-xjobs とともに使用できるのは -xipo オプションだけです。-xjobs=*n* を指定すると、内部手続きオプティマイザは、さまざまなファイルをコンパイルするために呼び出すことができるコードジェネレータインスタンスの最大数として、*n* を使用します。

### A.2.127.1 値

-xjobs には必ず値を指定する必要があります。それ以外の場合は、エラー診断が発行され、コンパイルは中止されます。

一般に、*n* に指定する確実な値は、使用できるプロセッサ数に 1.5 を掛けた数です。使用可能なプロセッサの数の何倍もの値を使用すると、生成されるジョブ間のコンテキスト切り替えのオーバーヘッドのために、パフォーマンスが低下する場合があります。また、あまり大きな数を使用すると、スワップ領域などシステムリソースの限界を超える場合があります。

### A.2.127.2 デフォルト

コマンド行に複数の -xjobs のインスタンスがある場合、一番右にあるインスタンスが実行されるまで相互に上書きします。

### A.2.127.3 例

次の例に示すコマンドは 2 つのプロセッサを持つシステム上で、-xjobs オプションを指定しないで実行された同じコマンドよりも高速にコンパイルを実行します。

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

## A.2.128 -xkeepframe[=[ %all,%none,name,no% name]]

指定した機能 (*name*) のスタック関連の最適化を禁止します。

`%all` すべてのコードのスタック関連最適化を禁止します。

`%none` すべてのコードのスタック関連最適化を許可します。

このオプションは累積的で、コマンド行で複数回指定できます。たとえば、`–xkeepframe=%all –xkeepframe=no%func1` は、`func1` を除くすべての関数のスタックフレームが保持されるはずであることを示します。また、`–xkeepframe` は `–xregs=frameptr` より優先されます。たとえば、`–xkeepframe=%all –xregs=frameptr` は、すべての関数のスタックが保持されるはずですが、`–xregs=frameptr` の最適化は無視されることを示します。

このオプションがコマンド行で指定されていないと、コンパイラはデフォルトの `-xkeepframe=%none` を使用します。このオプションが値なしで指定されると、コンパイラは `-xkeepframe=%all` を使用します。

## A.2.129 -xlang=language [,language]

該当する実行時ライブラリをインクルードし、指定された言語に適切な実行時環境を用意します。

### A.2.129.1 値

*language* は `f77`、`f90`、`f95`、`c99` のいずれかとします。

`f90` 引数と `f95` 引数は同じです。`c99` 引数は、`cc -xc99=%all` を付けてコンパイルされ、CC を付けてリンクされようとしているオブジェクトに対して ISO 9899:1999 C プログラミング言語の動作を呼び出します。

### A.2.129.2 相互の関連性

`-xlang=f90` と `-xlang=f95` の各オプションは `-library=f90` を意味し、`-xlang=f77` オプションは `-library=f77` を意味します。ただし、`-library=f77` と `-library=f90` の各オプションは、`-xlang` オプションしか正しい実行時環境を保証しないので、言語が混合したリンクには不十分です。

言語が混合したリンクの場合、ドライバは次の順序で言語階層を使用してください。

1. C++
2. Fortran 95 (または Fortran 90)
3. Fortran 77
4. C または C99

Fortran 95、FORTRAN 77、および C++ のオブジェクトファイルを一緒にリンクする場合は、最上位言語のドライバを使用します。たとえば、C++ と Fortran 95 のオブジェクトファイルをリンクするには、次の C++ コンパイラコマンドを使用します。

```
example% CC -xlang=f95...
```

Fortran 95 と Fortran 77 のオブジェクトファイルをリンクするには、次のように Fortran 95 ドライバを使用します。

```
example% f95 -xlang=f77...
```

`-xlang` オプションと `-xlic_lib` オプションを同じコンパイラコマンドで使用することはできません。`-xlang` を使用していて、しかも Sun Performance Library でリンクする必要がある場合は、代わりに `-library=sunperf` を使用してください。

### A.2.129.3 警告

`-xlang` と一緒に `-xnolib` を使用しないでください。

Fortran 並列オブジェクトを C++ オブジェクトと混合している場合は、リンク行に `-mt` フラグを指定する必要があります。

### A.2.129.4 関連項目

`-library`、`-staticlib`

## A.2.130 -xldscope={*v*}

extern シンボルの定義に対するデフォルトのリンカースコープを変更するには、`-xldscope` オプションを指定します。デフォルトを変更すると、実装がよりうまく隠蔽されるため、共有ライブラリと実行可能ファイルをより高速かつ安全に使用できるようになります。

### A.2.130.1 値

次の表に、*v* に指定可能な値を示します。

表 A–36 -xldscope の値

| 値 | 意味 |
|---|---|
| `global` | 大域リンカースコープは、もっとも制限の少ないリンカースコープです。シンボル参照はすべて、そのシンボルが定義されている最初の動的ロードモジュール内の定義と結合します。このリンカースコープは、extern シンボルの現在のリンカースコープです。 |
| `symbolic` | シンボリックリンカースコープは、大域リンカースコープよりも制限的です。リンク対象の動的ロードモジュール内からのシンボルへの参照はすべて、そのモジュール内に定義されているシンボルと結合します。モジュール外については、シンボルは大域なものとなります。このリンカースコープはリンカーオプション `-Bsymbolic` に対応します。C++ ライブラリでは `-Bsymbolic` を使用できませんが、`-xldscope=symbolic` 指定子は問題なく使用できます。リンカーの詳細については、`ld`(1) のマニュアルページを参照してください。 |
| `hidden` | 隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも制限されたリンカースコープです。動的ロードモジュール内の参照はすべて、そのモジュール内の定義に結合します。シンボルはモジュールの外側では認識されません。 |

### A.2.130.2 デフォルト

`-xldscope` を指定しない場合は、コンパイラでは `-xldscope=global` が指定されます。値を指定しないで `-xldscope` を指定すると、コンパイラがエラーを出力します。コマンド行にあるこのオプションの複数のインスタンスは、右端のインスタンスに達するまで互いに上書きされます。

### A.2.130.3 警告

クライアントがライブラリ内の関数をオーバーライドできるようにする場合は必ず、ライブラリの構築時に関数がインラインで生成されないようにしてください。コンパイラは次の状況で関数をインライン化します。

- 関数名は、`-xinline` で指定されます。

- インライン化が自動的に実行される -xO4 以上を使用してコンパイルした場合。
- インライン指示子またはファイル間の最適化を使用した場合。

たとえば、ABC というライブラリにデフォルトの allocator 関数があり、ライブラリクライアントがその関数を使用でき、ライブラリの内部でも使用されるものとします。

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

-xO4 以上でライブラリを構築すると、コンパイラはライブラリ構成要素内での ABC_allocator の呼び出しをインライン化します。ライブラリクライアントが ABC_allocator を独自の allocator と置き換える場合、置き換えられた allocator は ABC_allocator を呼び出したライブラリ構成要素内では実行されません。最終的なプログラムには、この関数の相異なるバージョンが含まれることになります。

__hidden 指示子または __symbolic 指示子で宣言されたライブラリ関数は、ライブラリの構築時にインラインで生成することができます。これらの指示子がクライアントによって上書きされることは想定されていません。63 ページの「4.1 リンカースコープ」を参照してください。

__global 指示子で宣言されたライブラリ関数はインラインで宣言しないでください。また、-xinline コンパイラオプションを使用することによってインライン化されることがないようにしてください。

### A.2.130.4 関連項目

-xinline、-xO

## A.2.131 -xlibmieee

例外時に libm が数学ルーチンに対し IEEE 754 値を返します。

libm のデフォルト動作は XPG に準拠します。

### A.2.131.1 関連項目

『数値計算ガイド』

## A.2.132 -xlibmil

選択された libm 数学ライブラリルーチンを最適化のためにインライン化します。

注 – このオプションはC++ インライン関数には影響しません。

このオプションは、現在使用されている浮動小数点オプションとプラットフォームのためのもっとも高速な実行可能ファイルを生成する、libm ルーチンのインラインテンプレートを選択します。

### A.2.132.1 相互の関連性

このオプションの機能は、-fast オプションを指定した場合にも含まれます。

### 関連項目

-fast、『数値計算ガイド』

## A.2.133 -xlibmopt

最適化された数学ルーチンのライブラリを使用します。このオプションを使用するときは -fround=nearest を指定することによって、デフォルトの丸めモードを使用する必要があります。

パフォーマンスが最適化された数学ルーチンのライブラリを使用し、より高速で実行できるコードを生成します。通常の数学ライブラリを使用した場合とは、結果が少し異なることがあります。このような場合、異なる部分は通常は最後のビットです。

このライブラリオプションをコマンド行に指定する順序は重要ではありません。

### A.2.133.1 相互の関連性

このオプションの機能は、-fast オプションを指定した場合にも含まれます。

### A.2.133.2 関連項目

-fast、-xnolibmopt、-fround

## A.2.134 -xlic_lib=sunperf

非推奨。使用しないでください。代わりに、-library=sunperf を指定します。詳細は、214 ページの「A.2.49 -library=*l*[ ,*l*...]」 を参照してください。

## A.2.135 -xlicinfo

このオプションは、コンパイル時には暗黙的に無視されます。

## A.2.136 -xlinkopt[=*level*]

(*SPARC* のみ) コンパイラに、オブジェクトファイル内のすべての最適化に加えて、結果として得られる実行可能ファイルまたは動的ライブラリに対してリンク時の最適化を実行するよう指示します。このような最適化は、リンク時にオブジェクトのバイナリコードを解析することによって実行されます。オブジェクトファイルは書き換えられませんが、最適化された実行可能コードは元のオブジェクトコードとは異なる場合があります。

`-xlinkopt` をリンク時に有効にするには、少なくともコンパイルコマンドで `-xlinkopt` を使用する必要があります。`-xlinkopt` を指定しないでコンパイルされたオブジェクトバイナリについても、オプティマイザは限定的な最適化を実行できます。

`-xlinkopt` は、コンパイラのコマンド行にある静的ライブラリのコードは最適化しますが、コマンド行にある共有 (動的) ライブラリのコードは最適化しません。`-xlinkopt` は、共有ライブラリを構築する (`-G` でコンパイルする) 場合にも使用できます。

### A.2.136.1 値

*level* には、実行する最適化のレベルを 0、1、2 のいずれかで設定します。最適化レベルを次の表に示します。

表 A–37 -xlinkopt の値

| 値 | 意味 |
|---|---|
| 0 | リンクオプティマイザは無効になっています (デフォルト)。 |
| 1 | リンク時の命令キャッシュカラーリングと分岐の最適化を含む、制御フロー解析に基づき最適化を実行します。 |
| 2 | リンク時のデッドコードの除去とアドレス演算の簡素化を含む、追加のデータフロー解析を実行します。 |

コンパイル手順とリンク手順を別々にコンパイルする場合は、両方の手順に `-xlinkopt` を指定する必要があります。

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

レベルパラメータは、コンパイラがリンクしている場合にのみ使用されます。この例では、オブジェクトバイナリが暗黙のレベル 1 でコンパイルされていても、リンクオプティマイザのレベルは 2 です。

### A.2.136.2 デフォルト

レベルパラメータなしで `-xlinkopt` を使用することは、`-xlinkopt=1` を指定することと同じです。

### A.2.136.3 相互の関連性

このオプションは、プログラム全体のコンパイル時に、プロファイルのフィードバックとともに使用されると、もっとも効果的です。プロファイリングは、コードの使用頻度がもっとも高い部分ともっとも低い部分を明らかにし、オプティマイザにそれに基づいて処理を集中するよう指示します。これは、リンク時に実行されるコードの最適な配置が命令のキャッシュミスを低減できるような、大きなアプリケーションにとって特に重要です。このオプションは通常、次のように使用されます。

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

プロファイルのフィードバックの使用についての詳細は、305 ページの「A.2.164 –xprofile=*p*」を参照してください。

### A.2.136.4 警告

`-xlinkopt` でコンパイルする場合は、`-zcombreloc` リンカーオプションは使用しないでください。

このオプションを指定してコンパイルすると、リンク時間がわずかに増えます。オブジェクトファイルも大きくなりますが、実行可能ファイルのサイズは変わりません。`-xlinkopt` と `-g` を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。

## A.2.137 -xloopinfo

このオプションは、どのループが並列化されているかを表示し、通常は `-xautopar` オプションとともに使用されます。

## A.2.138 –xM

指定された C++ プログラムに対して C++ プリプロセッサのみを実行し、そのプリプロセッサに対し、メイクファイルの依存関係を生成し、結果を標準出力に送信するよう要求します。`make` ファイルと依存関係についての詳細は、`make`(1) のマニュアルページを参照してください。

ただし、-xM を指定すると、インクルードされているヘッダーの依存関係のみを報告し、関連付けられているテンプレート定義ファイルの依存関係を報告しません。メイクファイルの中で .KEEP_STATE 機能を使用して、make ユーティリティーが使用する .make.state ファイルの中にあるすべての依存関係を使用することもできます。

### A.2.138.1 例

次の例:

```
#include <unistd.h>
void main(void)
{}
```

この出力を生成します。

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

### A.2.138.2 相互の関連性

-xM と -xMF を指定する場合、-xMF で指定したファイルに、コンパイラはすべてのメイクファイルの依存関係情報を書き込みます。プリプロセッサがこのファイルへの書き込みを行うたびに、このファイルは上書きされます。

### A.2.138.3 関連項目

メイクファイルと依存関係についての詳細は、make(1S) のマニュアルページを参照してください。

## A.2.139 -xM1

/usr/include ヘッダーファイルの依存関係を報告せず、またコンパイラで提供されるヘッダーファイルの依存関係を報告しない点を除き、-xM のようなメイクファイルの依存関係を生成します。

-xM1 と -xMF を指定する場合、-xMF で指定したファイルに、コンパイラはすべてのメイクファイルの依存関係情報を書き込みます。プリプロセッサがこのファイルへの書き込みを行うたびに、このファイルは上書きされます。

## A.2.140 -xMD

-xM と同様にメイクファイルの依存関係を生成しますが、コンパイルを続行します。-xMD は、指定されている場合は -o 出力ファイル名から派生したメイクファイルの依存関係情報の出力ファイルを生成します。または、ファイル名接尾辞を .d に置換 (または追加) して、入力元ファイル名を生成します。-xMD と -xMF を指定した場合、プリプロセッサは、メイクファイルのすべての依存関係情報を -xMF で指定されたファイルに書き込みます。複数のソースファイルで -xMD -xMF または -xMD -o *filename* を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

## A.2.141 -xMF

メイクファイルの依存関係の出力先ファイルを指定するには、このオプションを使用します。1 つのコマンド行で、複数の入力ファイルに対して -xMF で個別のファイル名を指定することはできません。複数のソースファイルを使用して -xMD -xMF または -xMMD -xMF を使用してコンパイルすることは許可されておらず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

## A.2.142 -xMMD

システムヘッダーファイルを除き、メイクファイルの依存関係を生成するには、このオプションを使用します。このオプションでは -xM1 と同じ機能が提供されますが、コンパイルは続行されます。-xMMD は、-o 出力ファイル名 (指定されている場合) または入力ソースファイル名から派生したメイクファイルの依存関係情報のための出力ファイルを生成して、ファイル名接尾辞を .d に置き換え (または追加) します。-xMF を指定する場合、コンパイラは代わりに、ユーザーが指定したファイル名を使用します。複数のソースファイルで -xMMD -xMF または -xMMD -o *filename* を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

## A.2.143 -xMerge

(*SPARC* のみ) データセグメントをテキストセグメントにマージします。

オブジェクトファイル内のデータは読み取り専用であり、ld -N を使用してリンクしないかぎりプロセス間で共有されます。

3 つのオプション -xMerge -ztext -xprofile=collect を一緒に使用するべきではありません。-xMerge を指定すると、静的に初期化されたデータを読み取り専用記憶領域に強制的に配置します。 -ztext を指定すると、位置に依存するシンボルを読み取り

専用記憶領域内で再配置することを禁止します。-xprofile=collect を指定すると、書き込み可能記憶領域内で、静的に初期化された、位置に依存するシンボルの再配置を生成します。

### A.2.143.1 関連項目

ld(1) のマニュアルページ

## A.2.144 -xmaxopt[=v]

このオプションは、pragma opt のレベルを指定されたレベルに制限します。*v* は、off、1、2、3、4、5 のいずれかです。デフォルト値は -xmaxopt=off であり、pragma opt は無視されます。引数を指定せずに -xmaxopt を指定したときのデフォルトは -xmaxopt=5 です。

-xO と -xmaxopt の両方を指定する場合、-xO で設定する最適化レベルが -xmaxopt 値を超えてはいけません。

## A.2.145 -xmemalign=*ab*

(*SPARC* のみ) データの境界整列に関するコンパイラの前提を制御するには、-xmemalign オプションを使用します。境界整列が潜在的に正しくないメモリーアクセスにつながる生成コードを制御し、境界整列が正しくないアクセスが発生したときのプログラム動作を制御すれば、より簡単に SPARC にコードを移植できます。

想定するメモリー境界整列の最大値と、境界整列に失敗したデータがアクセスされた際の動作を指定します。*a* (境界整列) と *b* (動作) の両方に値を指定する必要があります。*a* は想定される最大メモリー境界整列を指定し、*b* は境界整列されていないメモリーアクセスに対応する動作を指定します。

コンパイル時に境界整列が判別できるメモリーへのアクセスの場合、コンパイラはそのデータの境界整列に適したロードおよびストア命令を生成します。

境界整列がコンパイル時に決定できないメモリーアクセスの場合、コンパイラは、境界整列を想定して、必要なロード/ストア命令のシーケンスを生成します。

実行時の実際のデータ境界整列が指定された整列に達しない場合、境界整列に失敗したアクセス (メモリー読み取りまたは書き込み) が行われると、トラップが発生します。このトラップに対して発生する可能性のある応答は次の 2 つです。

- OS がトラップを SIGBUS シグナルに変換します。プログラムがこのシグナルを捕捉しなかった場合、プログラムは異常終了します。プログラムがシグナルを捕捉しても、境界整列に失敗したアクセスが成功するわけではありません。

- 境界整列に失敗したアクセスが正常に成功したかのように OS がアクセスを解釈し、プログラムに制御を戻すことによってトラップを処理します。

### A.2.145.1 値

`-xmemalign` の境界整列と動作の値を次に示します。

*a* の値:

1 最大で 1 バイトの境界整列を想定します。

2 最大で 2 バイトの境界整列を想定します。

4 最大で 4 バイトの境界整列を想定します。

8 最大で 8 バイトの境界整列を想定します。

16 最大で 16 バイトの境界整列を想定します。

*b* の値:

`i` アクセスを解釈し、実行を継続する

`s` シグナル SIGBUS を発生させます。

`f` 64 ビット SPARC アーキテクチャーの場合: 4 以下の境界整列に対してシグナル SIGBUS を発生させます。それ以外の場合は、アクセスを解釈し、実行を継続します。

その他のすべてのアーキテクチャーでは、このフラグは `i` と同等です。

*b* を `i` か `f` のいずれかに設定してコンパイルしたオブジェクトファイルにリンクする場合は、必ず、`-xmemalign` を指定する必要があります。48 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

### A.2.145.2 デフォルト

次のデフォルトの値は、`-xmemalign` オプションがまったく指定されていない場合にのみ適用されます。

- `-xmemalign=8i` は、すべての 32 ビット SPARC アーキテクチャー (`-m32`) に適用されます。
- `-xmemalign=8s` は、すべての 64 ビット SPARC アーキテクチャー (`-m64`) に適用されます。

`-xmemalign` オプションが存在するが、値が指定されていない場合は、次のデフォルト値が使用されます。

- `-xmemalign=1i` は、すべてのアーキテクチャーに適用されます。

### A.2.145.3 例

さまざまな境界整列の状況を処理する場合の -xmemalign の使用方法を次に示します。

-xmemalign=1s　すべてのメモリーアクセスの整列が正しくないため、トラップ処理が遅すぎる。

-xmemalign=8i　コード内で、不定期の、意図的な、境界整列されていないアクセスが発生する場合があるが、それ以外は正しい。

-xmemalign=8s　プログラム内で、境界整列されていないアクセスは発生しない。

-xmemalign=2s　奇数バイトへのアクセスが存在しないか検査したい

-xmemalign=2i　奇数バイトへのアクセスが存在しないか検査し、プログラムを実行したい

## A.2.146 -xmodel=[*a*]

(*x86* のみ) -xmodel オプションを使用すると、コンパイラは Oracle Solaris x86 プラットフォーム用の 64 ビットオブジェクトの形式を変更できるようになります。このオプションは、このようなオブジェクトのコンパイルに対してのみ指定してください。

このオプションは、64 ビット対応の x64 プロセッサで -m64 も指定した場合にのみ有効です。

次の表に、*a* に指定可能な値を示します。

表 A–38　-xmodel のフラグ

| 値 | 意味 |
|---|---|
| small | このオプションは、実行されるコードの仮想アドレスがリンク時にわかっていて、すべてのシンボルが 0 ~ 2^31 - 2^24 - 1 の範囲の仮想アドレスに配置されることがわかっているスモールモデルのコードを生成します。 |
| kernel | すべてのシンボルが 2^64 - 2^31 ~ 2^64 - 2^24 の範囲で定義されるカーネルモデルのコードを生成します。 |
| medium | データセクションへのシンボリック参照の範囲に関する前提がないミディアムモデルのコードを生成します。テキストセクションのサイズとアドレスには、スモールコードモデルと同じ制限があります。静的データが大量にあるアプリケーションでは、-m64 を指定してコンパイルするときに、-xmodel=medium が必要になることがあります。 |

このオプションは累積的ではないため、コンパイラは、コマンド行にある -xmodel の右端のインスタンスに従ってモデル値を設定します。

-xmodel を指定しない場合、コンパイラは -xmodel=small とみなします。引数を指定せずに-xmodel を指定すると、エラーになります。

すべての変換ユニットをこのオプションを使用してコンパイルする必要はありません。アクセスするオブジェクトが範囲内にあれば、選択したファイルをコンパイルできます。

すべての Linux システムが、ミディアムモデルをサポートしているわけではありません。

## A.2.147 -xnolib

デフォルトのシステムライブラリとのリンクを無効にします。

通常 (このオプションを指定しない場合)、C++ コンパイラは、C++ プログラムをサポートするためにいくつかのシステムライブラリとリンクします。このオプションを指定すると、デフォルトのシステムサポートライブラリとリンクするための -l*lib* オプションが ld に渡されません。

通常、コンパイラは、システムサポートライブラリにこの順序でリンクします。

- デフォルトの -compat=5 の場合、ライブラリは次のとおりです。

  ```
  -lCstd -lCrun -lm -lc
  ```

- Linux 上の -compat=g の場合、ライブラリは次のとおりです。

  ```
  –lstdc++ –lCrunG3 –lm –lc
  ```

- Oracle x86 上の -compat=g の場合、ライブラリは次のとおりです。

  ```
  –lstdc++ –lgcc_s –lCrunG3 –lm –lc
  ```

-l オプションの順序は重要です。-lm オプションは -lc の前にある必要があります。

---

注 – -mt コンパイラオプションを指定した場合、コンパイラは通常 -lm でリンクする直前に -lthread でリンクします。

---

デフォルトでどのシステムサポートライブラリがリンクされるかを知りたい場合は、コンパイルで-dryrun オプションを指定します。たとえば、次のコマンドを実行するとします。

```
example% CC foo.cc -m64 -dryrun
```

この出力には次の内容が示されます。

```
-lCstd -lCrun -lm -lc
```

### A.2.147.1 例

C アプリケーションのバイナリインタフェースを満たす最小限のコンパイルを行う (つまり、必要な C サポートのみを含む C++ プログラムを生成する) には、次のコマンドを使用します。

```
example% CC -xnolib test.cc –lc
```

libm を一般的なアーキテクチャー命令セットを含むシングルスレッドアプリケーションに静的にリンクするには、次のコマンドを使用します。

```
example% CC -xnolib test.cc -lCstd -lCrun -Bstatic -lm -Bdynamic -lc
```

### A.2.147.2 相互の関連性

-xnolib を指定する場合は、必要なすべてのシステムサポートライブラリを手動で一定の順序にリンクする必要があります。システムサポートライブラリは最後にリンクしなければいけません。

-xnolib を指定すると、-library は無視されます。

### A.2.147.3 警告

多くの C++ 言語機能では、libCrun (標準モード) を使用する必要があります。

このリリースのシステムサポートライブラリは安定していないため、リリースごとに変更される可能性があります。

### A.2.147.4 関連項目

-library、-staticlib、-l

## A.2.148 -xnolibmil

コマンド行の -xlibmil を取り消します。

最適化された数学ライブラリとのリンクを変更するには、このオプションを -fast と一緒に使用してください。

## A.2.149 -xnolibmopt

数学ルーチンのライブラリを使用しません。

### A.2.149.1 例

次の例のように、このオプションはコマンド行で -fast オプションを指定した場合は、そのあとに使用してください。

```
example% CC –fast –xnolibmopt
```

## A.2.150 -xnorunpath

221 ページの「A.2.60 -norunpath」 と同じです

## A.2.151 -xOlevel

最適化レベルを指定します。大文字 O のあとに数字の 1、2、3、4、5 のいずれかが続きます。一般的に、プログラムの実行速度は最適化のレベルに依存します。最適化レベルが高いほど、実行時のパフォーマンスは向上します。しかし、最適化レベルが高ければ、それだけコンパイル時間が増え、実行可能ファイルが大きくなる可能性があります。

ごくまれに、-x02 の方がほかの値より実行速度が速くなることがあり、-x03 の方が -x04 より早くなることがあります。すべてのレベルでコンパイルを行なってみて、こうしたことが発生するかどうか試してみてください。

メモリー不足になった場合、オプティマイザは最適化レベルを落として現在の手続きをやり直すことによってメモリー不足を回復しようとします。ただし、以降の手続きについては、-x0*level* オプションで指定された最適化レベルを使用します。

以降の節では、5 つの -x0 *level* 最適化レベルが SPARC プラットフォームと x86 プラットフォーム上でどのように動作するかについて説明します。

### A.2.151.1 値

**SPARC** プラットフォームの場合

- -x01 では、最小限の最適化 (ピープホール) が行われます。これはコンパイルの後処理におけるアセンブリレベルでの最適化です。-x02 や -x03 を使用するとコンパイル時間が著しく増加する場合や、スワップ領域が不足する場合だけ -x01 を使用してください。
- -x02 では、次の基本的な局所的および大域的な最適化が行われます。
  - 帰納的変数の削除
  - 局所的および大域的な共通部分式の削除
  - 計算の簡略化
  - コピーの伝播

- 定数の伝播
- ループ不変式の最適化
- レジスタ割り当て
- 基本ブロックのマージ
- 末尾再帰の削除
- デッドコードの削除
- 末尾呼び出しの削除
- 複雑な式の展開

  このレベルでは、外部変数や間接変数の参照や定義は最適化されません。

`-xO3` では、`-xO2` レベルで行う最適化に加えて、外部変数に対する参照と定義も最適化されます。このレベルでは、ポインタ代入の影響は追跡されません。`volatile` で正しく保護されていないデバイスドライバか、またはシグナルハンドラ内から外部変数を変更するプログラムをコンパイルする場合は、`-xO2` を使用します。一般に、このレベルを使用すると、`-xspace` オプションと組み合わせないかぎり、コードサイズが大きくなります。

- `-xO4` は、`-xO3` の最適化の実行に加えて、同じファイルに含まれている関数の自動インライン化を実行します。インライン化を自動的に行なった場合、通常は実行速度が速くなりますが、遅くなることもあります。一般に、このレベルを使用すると、`-xspace` オプションと組み合わせないかぎり、コードサイズが大きくなります。
- `-xO5` では、最高レベルの最適化が行われます。これを使用するのは、コンピュータのもっとも多くの時間を小さなプログラムが使用している場合だけにしてください。このレベルで使用される最適化アルゴリズムでは、コンパイル時間が増えたり、実行時間が改善されないことがあります。このレベルの最適化によってパフォーマンスが改善される確率を高くするには、プロファイルのフィードバックを使用します。305 ページの「A.2.164 –xprofile=*p*」を参照してください。

**x86** プラットフォームの場合

- `-xO1` では、基本的な最適化を行います。このレベルには、計算の簡略化、レジスタ割り当て、基本ブロックのマージ、デッドコードとストアの削除、およびピープホールの最適化が含まれます。
- `-xO2` では、局所的な共通部分の削除、局所的なコピーと定数の伝播、末尾再帰の削除、およびレベル 1 で行われる最適化を実行します。
- `-xO3` では、局所的な共通部分の削除、大域的なコピーと定数の伝播、ループ強度低下、帰納的変数の削除、およびループ不変式の最適化、およびレベル 2 で行われる最適化を実行します。

- -xO4 では、レベル 3 で行う最適化レベルに加えて、同じファイルに含まれる関数の自動的なインライン化も行われます。インライン化を自動的に行なった場合、通常は実行速度が速くなりますが、遅くなることもあります。このレベルでは一般用のフレームポインタ登録 (edp) も解放します。一般にこのレベルを使用するとコードサイズが大きくなります。
- -xO5 では、最高レベルの最適化が行われます。このレベルで使用される最適化アルゴリズムでは、コンパイル時間が増えたり、実行時間が改善されないことがあります。

### A.2.151.2 相互の関連性

-g または -g0 を使用するとき、最適化レベルが -xO3 以下の場合、最大限のシンボリック情報とほぼ最高の最適化が得られます。

-g または -g0 を使用するとき、最適化レベルが -xO4 以上の場合、最大限のシンボリック情報と最高の最適化が得られます。

-g によるデバッグでは、-x0*level* が抑制されませんが、-x0*level* はいくつかの方法で -g を制限します。たとえば、-x0*level* オプションを使用すると、dbx から渡された変数を表示できないなど、デバッグの機能が一部制限されます。しかし、dbx where コマンドを使用して、シンボリックトレースバックを表示することは可能です。詳細は、『dbx コマンドによるデバッグ』を参照してください。

-xipo オプションは、-xO4 または -xO5 と一緒に使用した場合にのみ効果があります。

-xinline オプションは -xO3 未満の最適化レベルには影響を与えません。-xO4 では、-xinline オプションを指定したかどうかは関係なく、オプティマイザはどの関数をインライン化するかを判断します。-xO4 では、コンパイラはどの関数が、インライン化されたときにパフォーマンスを改善するかを判断しようとします。-xinline を指定して関数のインライン化を強制すると、実際にパフォーマンスを低下させる可能性があります。

### A.2.151.3 デフォルト

デフォルトでは最適化は行われません。ただし、これは最適化レベルを指定しない場合にかぎり有効です。最適化レベルを指定すると、最適化を無効にするオプションはありません。

最適化レベルを設定しないようにする場合は、最適化レベルを示すようなオプションを指定しないようにしてください。たとえば、-fast は最適化を -xO5 に設定するマクロオプションです。最適化レベルを暗黙に示すほかのすべてのオプションは、最適化レベルが設定されているという警告メッセージを発行します。最適化を設定せずにコンパイルする唯一の方法は、コマンド行またはメイクファイルから最適化レ ベルを指定するオプションをすべて削除することです。

### A.2.151.4 警告

大規模な手続き (数千行のコードからなる手続き) に対して -xO3 または -xO4 を指定して最適化をすると、不合理な大きさのメモリーが必要になります。マシンのパフォーマンスが低下することがあります。

この低下が発生しないようにするには、limit コマンドを使用して、1 つのプロセスで使用できる仮想メモリーの量を制限します。csh(1) のマニュアルページを参照してください。たとえば、仮想メモリーを 4G バイトに制限するには、次のコマンドを使用します。

```
example% limit datasize 4G
```

このコマンドにより、データ領域が 4G バイトに達したときに、オプティマイザがメモリー不足を回復しようとします。

マシンが使用できるスワップ領域の合計容量を超える値は、制限値として指定することはできません。制限値は、大規模なコンパイル中でもマシンの通常の使用ができるぐらいの大きさにしてください。

最良のデータサイズ設定値は、要求する最適化のレベルと実メモリーの量、仮想メモリーの量によって異なります。

実際のスワップ領域を検索するには、**swap -l** と入力します。

実際の実メモリーを検索するには、**dmesg | grep mem** と入力します。

### A.2.151.5 関連項目

-xldscope -fast、-xprofile=*p*、csh(1) のマニュアルページ

## A.2.152 -xopenmp[= i]

OpenMP 指令による明示的並列化を使用するには、-xopenmp オプションを指定します。

### A.2.152.1 値

次の表に *i* の値を示します。

表 A–39 -xopenmp の値

| 値 | 意味 |
|---|---|
| parallel | OpenMP プラグマの認識を有効にします。-xopenmp=parallel での最小の最適化レベルは -xO3 です。コンパイラは、必要に応じて最適化を低いレベルから -xO3 に変更し、警告を発行します。<br>このフラグは、プリプロセッサトークン _OPENMP も定義します。 |

表 A–39 -xopenmp の値 (続き)

| 値 | 意味 |
|---|---|
| noopt | OpenMP プラグマの認識を有効にします。最適化レベルが -O3 より低い場合は、最適化レベルは上げられません。<br><br>CC -O2 -xopenmp=noopt のように、-O3 より低い最適化レベルを明示的に設定した場合、コンパイラはエラーを発生させます。-xopenmp=noopt で最適化レベルを指定しない場合は、OpenMP プラグマが認識され、それに応じてプログラムが並列化されますが、最適化は実行されません。<br><br>このフラグは、プリプロセッサトークン _OPENMP も定義します。 |
| none | このフラグはデフォルトであり、OpenMP プラグマの認識を無効にします。コンパイルの最適化レベルは変更せず、どのプリプロセッサトークンも事前定義しません。 |

### A.2.152.2 デフォルト

-xopenmp を指定しない場合、コンパイラのデフォルトは -xopenmp=none です。

-xopenmp を引数なしで指定した場合、コンパイラのデフォルトは -xopenmp=parallel です。

### A.2.152.3 相互の関連性

dbx を指定して OpenMP プログラムをデバッグする場合、-g と -xopenmp=noopt を指定してコンパイルすれば、並列化部分にブレークポイントを設定して変数の内容を表示することができます。

OpenMP プログラムを実行するときに使用するスレッドの数を指定するには、OMP_NUM_THREADS 環境変数を使用します。OMP_NUM_THREADS が設定されていない場合、使用されるスレッドの数のデフォルトは 2 です。より多くのスレッドを使用するには、OMP_NUM_THREADS をより高い値に設定します。1 つのスレッドだけで実行する場合は、OMP_NUM_THREADS を 1 に設定します。一般に、OMP_NUM_THREADS は、実行中のシステム上の使用可能な仮想プロセッサの数に設定します。これは、Oracle Solaris の psrinfo(1) コマンドを使用して特定できます。詳細は、『Oracle Solaris Studio OpenMP API ユーザーガイド』を参照してください。

入れ子並列を有効にするには、OMP_NESTED 環境変数を TRUE に設定する必要があります。入れ子並列は、デフォルトでは無効です。詳細は、『Oracle Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

### A.2.152.4 警告

-xopenmp のデフォルトは、将来変更される可能性があります。警告メッセージを出力しないようにするには、適切な最適化を明示的に指定します。

コンパイルとリンクを別々に実行する場合は、コンパイル手順とリンク手順の両方に -xopenmp を指定してください。共有オブジェクトを作成する場合、このことは重要です。実行可能ファイルのコンパイルに使用されたコンパイラは、xopenmp を使用して -.so を構築したコンパイラより古いものであってはいけません。これは、OpenMP 指令を含むライブラリをコンパイルする場合に特に重要です。48 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

最良のパフォーマンスを得るには、OpenMP 実行時ライブラリ libmtsk.so の最新パッチが、システムにインストールされていることを確認してください。

### A.2.152.5 関連項目

多重処理アプリケーションを構築するための OpenMP Fortran 95、C、および C++ アプリケーションプログラムインタフェース (API) の完全な概要については、『『Oracle Solaris Studio OpenMP API ユーザーズガイド』』を参照してください。

## A.2.153 -xpagesize=*n*

スタックとヒープの優先ページサイズを設定します。

### A.2.153.1 値

次の値は、SPARC で有効です。4k、8K、64K、512K、2M、4M、32M、256M、2G、16G、または default。

次の値は、x86/x64 で有効です。4K、2M、4M、1G、または default。

ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。有効なページサイズを指定しない場合は、要求は実行時にサイレントに無視されます。

Oracle Solaris オペレーティングシステムでページのバイト数を判断するには、getpagesize(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するために、pmap(1) または meminfo(2) を使用できます。

---

注 – このオプションを使用してコンパイルすると、LD_PRELOAD 環境変数を同等のオプションを使用して mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを使用して Oracle Solaris のコマンド ppgsz(1) を実行するのと同じ効果があります。詳細は、Oracle Solaris の各マニュアルページを参照してください。

---

### A.2.153.2 デフォルト

`-xpagesize=default` を指定する場合は、Oracle Solaris オペレーティングシステムがページサイズを設定します。

### A.2.153.3 展開

このオプションは `-xpagesize_heap` と `-xpagesize_stack` のマクロです。これらの 2 つのオプションは `-xpagesize` と同じ次の引数を使用します。`4k`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G`、`default` のいずれか。両方に同じ値を設定するには `-xpagesize` を指定します。別々の値を指定するには個々に指定します。

### A.2.153.4 警告

`-xpagesize` オプションは、コンパイル時とリンク時に使用しないかぎり無効です。48 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

## A.2.154 -xpagesize_heap=*n*

メモリー上のヒープのページサイズを設定します。

### A.2.154.1 値

*n* の値は、`4k`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G`、または `default` です。ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。有効なページサイズを指定しない場合は、要求は実行時にサイレントに無視されます。

Oracle Solaris オペレーティングシステムでページのバイト数を判断するには、`getpagesize`(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するために、`pmap`(1) または `meminfo`(2) を使用できます。

---

注 – このオプションを使用してコンパイルすると、`LD_PRELOAD` 環境変数を同等のオプションを使用して `mpss.so.1` に設定するか、またはプログラムを実行する前に同等のオプションを使用して Oracle Solaris のコマンド `ppgsz`(1) を実行するのと同じ効果があります。詳細は、Oracle Solaris の各マニュアルページを参照してください。

---

### A.2.154.2 デフォルト

`-xpagesize_heap=default` を指定する場合は、Oracle Solaris オペレーティングシステムがページサイズを設定します。

### A.2.154.3 警告

-xpagesize_heap オプションは、コンパイル時とリンク時に使用しないかぎり無効です。

## A.2.155 -xpagesize_stack=*n*

メモリー上のスタックのページサイズを設定します。

### A.2.155.1 値

*n* の値は、4k、8K、64K、512K、2M、4M、32M、256M、2G、16G、または default です。ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。有効なページサイズを指定しない場合は、要求は実行時にサイレントに無視されます。

Oracle Solaris オペレーティングシステムでページのバイト数を判断するには、getpagesize(3C) コマンドを使用します。Oracle Solaris オペレーティングシステムは、ページサイズ要求に従うという保証を提供しません。ターゲットプラットフォームのページサイズを判断するために、pmap(1) または meminfo(2) を使用できます。

---

注 – このオプションを使用してコンパイルすると、LD_PRELOAD 環境変数を同等のオプションを使用して mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを使用して Oracle Solaris のコマンド ppgsz(1) を実行するのと同じ効果があります。詳細は、Oracle Solaris の各マニュアルページを参照してください。

---

### A.2.155.2 デフォルト

-xpagesize_stack=default を指定する場合は、Oracle Solaris オペレーティングシステムがページサイズを設定します。

### A.2.155.3 警告

-xpagesize_stack オプションは、コンパイル時とリンク時に使用しないかぎり無効です。

## A.2.156 -xpch=v

このコンパイラオプションは、プリコンパイル済みヘッダー機能を有効にします。プリコンパイル済みヘッダー機能によって、ソースファイルが、大量のソースコードが含まれる一連の共通のインクルードファイルを共有しているアプリケーションのコンパイル時間が短縮される可能性があります。コンパイラは 1 つの

ソースファイルから一連のヘッダーファイルに関する情報を収集し、そのソースファイルを再コンパイルしたり、同じ一連のヘッダーファイルを持つほかのソースファイルをコンパイルしたりするときにその情報を使用します。コンパイラが収集する情報は、プリコンパイル済みヘッダーファイルに格納されます。この機能を利用するには、`-xpch` と `-xpchstop` オプションを、`#pragma hdrstop` 指令と組み合わせて使用できます。

### A.2.156.1 プリコンパイル済みヘッダーファイルの作成

`-xpch=`*v* を指定する場合、*v* には `collect:`*pch-filename* または `use:`*pch-filename* を指定できます。`-xpch` を初回に使用するときは、`collect` モードを指定する必要があります。`-xpch=collect` を指定するコンパイルコマンドは、ソースファイルを 1 つしか指定できません。次の例では、`-xpch` オプションがソースファイル `a.cc` に基づいて `myheader.Cpch` というプリコンパイル済みヘッダーファイルを作成します。

```
CC -xpch=collect:myheader a.cc
```

有効なプリコンパイル済みヘッダーファイル名には、常に `.Cpch` という接尾辞が含まれます。*pch-filename* を指定する場合は、その接尾辞を追加するか、またはコンパイラによって追加されるようにすることができます。たとえば、`cc -xpch=collect:foo a.cc` と指定すると、プリコンパイル済みヘッダーファイルには `foo.Cpch` という名前が付けられます。

プリコンパイル済みヘッダーファイルを作成する場合、プリコンパイル済みヘッダーファイルを使用するすべてのソースファイルで共通な、一連のインクルードファイルを含むソースファイルを選択します。インクルードファイルの並びは、これらのソースファイル全体で同一でなければいけません。`collect` モードでは、1 つのソースファイル名だけが有効な値である点に注意してください。たとえば、`CC -xpch=collect:foo bar.cc` は有効ですが、`CC -xpch=collect:foo bar.cc foobar.cc` は、2 つのソースファイルを指定しているので無効です。

#### プリコンパイル済みヘッダーファイルの使用

プリコンパイル済みヘッダーファイルを使用するには、`-xpch=use:`*pch-filename* を指定します。プリコンパイル済みヘッダーファイルを作成するために使用されたソースファイルと同じインクルードファイルの並びを持つソースファイルであれば、いくつでも指定できます。たとえば、`use` モードで、次のようなコマンドがあるとします。`CC -xpch=use:foo.Cpch foo.c bar.cc foobar.cc`.

既存のプリコンパイル済みヘッダーファイルは、次の状況が当てはまる場合にのみ使用してください。当てはまらない場合は、プリコンパイル済みヘッダーファイルを再作成してください。

- プリコンパイル済みヘッダーファイルにアクセスするために使用するコンパイラは、プリコンパイル済みヘッダーファイルを作成したコンパイラと同じであること。あるバージョンのコンパイラによって作成されたプリコンパイル済み

ヘッダーファイルが、インストール済みのパッチによる違いも含め、別のバージョンのコンパイラでは使用できない可能性があります。

- -xpch オプション以外で -xpch=use とともに指定するコンパイラオプションは、プリコンパイル済みヘッダーファイルが作成されたときに指定されたオプションと一致すること。
- -xpch=use で指定する一連のインクルードヘッダー群は、プリコンパイル済みヘッダーファイルが作成されたときに指定されたヘッダー群と同じであること。
- -xpch=use で指定するインクルードヘッダーの内容が、プリコンパイル済みヘッダーファイルが作成されたときに指定されたインクルードヘッダーの内容と同じであること。
- 現在のディレクトリ (すなわち、コンパイルが実行中で指定されたプリコンパイル済みヘッダーファイルを使用しようとしているディレクトリ) が、プリコンパイル済みヘッダーファイルが作成されたディレクトリと同じであること。
- -xpch=collect で指定したファイル内の前処理指令 (#include 指令を含む) の最初のシーケンスが、-xpch=use で指定したファイル内の前処理指令のシーケンスと同じであること。

プリコンパイル済みヘッダーファイルを複数のソースファイル間で共有するために、これらのソースファイルには、最初のトークンの並びとして一連の同じインクルードファイルを使用していなければいけません。この最初のトークンの並びは、活性文字列 *(viable prefix)* として知られています。活性文字列は、同じプリコンパイル済みヘッダーファイルを使用するすべてのソースファイル間で一貫して解釈される必要があります。

ソースファイルの活性文字列は、コメントと、次のプリプロセッサ指令のいずれかだけで構成できます。

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

これらの指令はいずれかがマクロを参照していてもかまいません。#else、#elif、および #endif 指令は、活性文字列内で一致している必要があります。

プリコンパイル済みヘッダーファイルを共有する各ファイルの活性文字列内では、対応する各 #define 指令と #undef 指令は同じシンボルを参照する必要があります。#define の場合は、それぞれが同じ値を参照する必要があります。各活性文字列内での順序も同じである必要があります。対応する各プラグマも同じで、その順序もプリコンパイル済みヘッダーを共有するすべてのファイルで同じでなければいけません。

プリコンパイル済みヘッダーファイルに組み込まれるヘッダーファイルは、次の制約に違反してはいけません。これらの制限に違反するプログラムをコンパイルした場合、結果は予測できません。

- ヘッダーファイルには、関数や変数の定義を含めることはできません。
- ヘッダーファイルに、`__DATE__` や `__TIME__` が含まれていてはいけません。これらのプリプロセッサマクロを使用すると、予測できない結果が生成される場合があります。
- ヘッダーファイルに、`#pragma hdrstop`が含まれていてはいけません。
- ヘッダーファイルは、活性文字列内で `__LINE__` と `__FILE__` を使用できません。インクルードヘッダー内の `__LINE__` と `__FILE__` を使用できます。

## メイクファイルを変更する方法

この節では、`-xpch` を構築に組み込むためにメイクファイルを変更する場合の考えられるアプローチについて説明します。

- `make` と `dmake` の `KEEP_STATE` 機能と `CCFLAGS` 補助変数を使用すれば、暗黙的な make 規則を使用できます。プリコンパイル済みヘッダーは、個別の独立した手順として生成されます。

```
.KEEP_STATE:
CCFLAGS_AUX = -O etc
CCFLAGS = -xpch=use:shared $(CCFLAGS_AUX)
shared.Cpch: foo.cc
     $(CCC) -xpch=collect:shared $(CCFLAGS_AUX) foo.cc
a.out: foo.o ping.o pong.o
     $(CCC) foo.o ping.o pong.o
```

また、`CCFLAGS` 補助変数を使用する代わりに、独自のコンパイル規則を定義することもできます。

```
.KEEP_STATE:
.SUFFIXES: .o .cc
%.o:%.cc shared.Cpch
        $(CCC) -xpch=use:shared $(CCFLAGS) -c $<
shared.Cpch: foo.cc
        $(CCC) -xpch=collect:shared $(CCFLAGS) foo.cc -xe
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

- プリコンパイル済みヘッダーを通常のコンパイルの二次効果として、`KEEP_STATE` を使用せずに生成することは可能ですが、このアプローチには明示的なコンパイルコマンドが必要です。

```
shared.Cpch + foo.o: foo.cc bar.h
        $(CCC) -xpch=collect:shared foo.cc $(CCFLAGS) -c
ping.o: ping.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared ping.cc $(CCFLAGS) -c
pong.o: pong.cc shared.Cpch bar.h
```

```
        $(CCC) -xpch=use:shared pong.cc $(CCFLAGS) -c
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

### A.2.156.2 関連項目


- 297 ページの「A.2.157 -xpchstop=*file* 」
- 334 ページの「B.2.9 #pragma hdrstop」


## A.2.157 -xpchstop=file

-xpch オプションを使用してコンパイル済みヘッダーファイルを作成するときに考慮される最後のインクルードファイルを指定するには、-xpchstop=*file* オプションを使用します。コマンド行で -xpchstop を使用することは、cc コマンドで指定する各ソースファイル内のファイルを参照する最初のインクルード指令のあとに、hdrstop プラグマを配置することと同じです。

次の例では、-xpchstop オプションで、プリコンパイル済みヘッダーファイルの活性文字列が projectheader.h をインクルードして終わるよう指定してます。したがって、privateheader.h は活性文字列の一部ではありません。

```
example% cat a.cc
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% CC -xpch=collect:foo.Cpch a.cc -xpchstop=projectheader.h -c
```

### A.2.157.1 関連項目

-xpch、 pragma hdrstop

## A.2.158 -xpec[={yes|no}]

(*Solaris* のみ) 移植可能な実行可能コード (Portable Executable Code、PEC) バイナリを生成します。このオプションは、プログラム中間表現をオブジェクトファイルとバイナリに入れます。このバイナリは、あとでチューニングやトラブルシューティングのために、使用される場合があります。

-xpec で構築したバイナリは通常、-xpec なしで構築したバイナリより 5 ~ 10 倍の大きさになります。

-xpec を指定しない場合は、-xpec=no に設定されます。-xpec をフラグなしで指定した場合は、コンパイラは xpec を -xpec=yes に設定します。

## A.2.159 -xpg

gprof プロファイラによるプロファイル処理用にコンパイルします。

-xpg オプションでは、gprof でプロファイル処理するためのデータを収集する自動プロファイルコードをコンパイルします。このオプションを指定すると、プログラムが正常に終了したときに gmon.out を生成する実行時記録メカニズムが呼び出されます。

---

注 – -xpg を指定した場合は、-xprofile を使用しても利点はありません。これら 2 つは、他方で使用できるデータを生成せず、他方で生成されたデータを使用できません。

---

プロファイルは、64 ビットの Solaris プラットフォーム上では prof(1) または gprof(1) を使用して、また 32 ビットの Solaris プラットフォーム上では gprof だけを使用して生成され、概略のユーザー CPU 時間が含まれます。これらの時間は、メインの実行可能ファイル内のルーチンと、実行可能ファイルがリンクされるときにリンカー引数として指定された共有ライブラリ内のルーチンの PC サンプルデータから導出されます。そのほかの共有ライブラリ (dlopen(3DL) を使用してプロセスの起動後に開かれたライブラリ) のプロファイルは作成されません。

32 ビットの Solaris システム上では、prof(1) を使用して生成されたプロファイルは実行可能ファイル内のルーチンに制限されます。32 ビット共有ライブラリは、実行可能ファイルを -xpg にリンクし、gprof(1) を使用することによってプロファイリングできます。

x86 システムでは、-xpg には -xregs=frameptr との互換性がないため、これらの 2 つのオプションは同時に使用できません。 -xregs=frameptr は -fast に含まれている点にも注意してください。

Oracle Solaris 10 ソフトウェアには、-p を使用してコンパイルされたシステムライブラリは含まれません。その結果、Solaris 10 プラットフォームで収集されたプロファイルには、システムライブラリルーチンの呼び出し回数が含まれません。

### A.2.159.1 警告

コンパイルとリンクを個別に行い、-xpg を使用してコンパイルする場合は、必ず -xpg を使用してリンクしてください。48 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

gprof プロファイリングのために -xpg でコンパイルされたバイナリは、binopt(1) では使用しないでください。互換性がないため、内部エラーになる可能性があります。

### A.2.159.2 関連項目

`-xprofile=`*p*、`analyzer(1)` のマニュアルページ、および『パフォーマンスアナライザ』のマニュアル

## A.2.160 -xport64[=(*v*)]

このオプションを指定すると、64 ビット環境に移植するコードをデバッグできます。このオプションは、具体的には、V8 などの 32 ビットアーキテクチャーを V9 などの 64 ビットアーキテクチャーにコード移植する際によく見られる、型 (ポインタを含む) の切り捨て、符号の拡張、ビット配置の変更といった問題について警告します。

このオプションは、コンパイルも 64 ビットモード (`-m64`) で実行していないかぎり無効です。

### A.2.160.1 値

次に、*v* に指定できる値を示します。

表 A–40 -xport64 の値

| 値 | 意味 |
|---|---|
| `no` | 32 ビット環境から 64 ビット環境へのコードの移植に関連した警告を生成しません。 |
| `implicit` | 暗黙の変換に関してのみ警告を生成する。明示的なキャストが存在する場合には警告を生成しない。 |
| `full` | 32 ビット環境から 64 ビット環境へのコードの移植に関連したすべての警告を生成します。これには、64 ビット値の切り捨て、ISO 値保護規則に基づく 64 ビットへの符号拡張、およびビットフィールドの配置の変更に対応する警告が含まれます。 |

### A.2.160.2 デフォルト

`-xport64` を指定しない場合のデフォルトは、`-xport64=no` です。`-xport64` を指定したが、フラグは指定しない場合、デフォルトは `-xport64=full` です。

### A.2.160.3 例

この節では、型の切り捨て、符号の拡張、およびビット配置の変更の原因になる可能性のあるコードの例について説明します。

## 64 ビット値の切り捨てのチェック

SPARC V9 などの 64 ビットアーキテクチャーに移植すると、データが切り捨てられる可能性があります。切り捨ては、初期化時に代入によって暗黙的に行われることもあれば、明示的なキャストによって行われることもあります。2 つのポインタの違いは typedef ptrdiff_t であり、32 ビットモードでは 32 ビット整数型、64 ビットモードでは 64 ビット整数型です。大きいサイズから小さいサイズの整数型に切り捨てると、次の例にあるような警告が生成されます。

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; //warn

example% CC -c -m64 -Qoption ccfe -xport64=full test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

明示的なキャストがデータ切り捨ての原因である場合に 64 ビットコンパイルモードでの切り捨ての警告を無効にするには、-xport64=implicit を使用します。

```
example% CC -c -m64 -Qoption ccfe -xport64=implicit test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

64 ビットアーキテクチャーへの移植でよく発生するもう 1 つの問題として、ポインタの切り捨てがあります。これは常に、C++ におけるエラーです。-xport64 を指定した場合は、このような切り捨ての原因となる int へのポインタのキャストなどの操作を行うと、64 ビット SPARC アーキテクチャーではエラー診断が実行されます。

```
example% cat test2.c
char* p;
int main() {
  p =(char*) (((unsigned int)p) & 0xFF); // -m64 error
  return 0;
}
example% CC -c -m64 -Qoption ccfe -xport64=full test2.c
"test2.c", line 3: Error: Cannot cast from char* to unsigned.
1 Error(s) detected.
example%
```

## 符号拡張のチェック

符号なし整数型の式において、通常の ISO C 値保護規則が符号付き整数値の符号拡張に対処している状況があるかどうかを、-xport64 オプションを使用してチェックすることもできます。こういった符号拡張は、実行時に微妙なバグの原因となる可能性があります。

```
example% cat test3.c
int i= -1;
```

```
void promo(unsigned long l) {}

int main() {
    unsigned long l;
    l = i;  // warn
    promo(i);       // warn
}
example% CC -c -m64 -Qoption ccfe -xport64=full test3.c
"test3.c", line 6: Warning: Sign extension from "int" to 64-bit integer.
"test3.c", line 7: Warning: Sign extension from "int" to 64-bit integer.
2 Warning(s) detected.
```

### ビットフィールドの配置変更のチェック

長いビットフィールドに対する警告を生成するには、-xport64 を使用します。こういったビットフィールドが存在していると、ビットフィールドの配置が大きく変わることがあります。ビットフィールド配置方式に関する前提事項に依存しているプログラムを、64 ビットアーキテクチャーに問題なく移植できるためには、あらかじめ確認作業を行う必要があります。

```
example% cat test4.c
#include <stdio.h>

union U {
   struct S {
       unsigned long b1:20;
       unsigned long b2:20;
   } s;

   long buf[2];
} u;

int main() {
   u.s.b1 = 0XFFFFF;
   u.s.b2 = 0XFFFFF;
   printf(" u.buf[0] = %lx u.buf[1] = %lx\n", u.buf[0], u.buf[1]);
   return 0;
}
example%
```

64 ビット SPARC システム (-m64) 上の出力:

```
example% u.buf[0] = fffffffff000000 u.buf[1] = 0
```

## A.2.160.4 警告

警告が生成されるのは、-m64 を使用して 64 ビットモードでコンパイルした場合だけです。

## A.2.160.5 関連項目

218 ページの「A.2.50 -m32|-m64」

## A.2.161 -xprefetch[=a[,a...]]

先読みをサポートするアーキテクチャーで先読み命令を有効にします。

明示的な先読みは、測定値によってサポートされた特殊な環境でのみ使用すべきです。

次の表に、*a* の指定可能な値を示します。

表 A–41 -xprefetch の値

| 値 | 意味 |
|---|---|
| `auto` | 先読み命令の自動生成を有効にします。 |
| `no%auto` | 先読み命令の自動生成を無効にします。 |
| `explicit` | (SPARC) 明示的な先読みマクロを有効にします。 |
| `no%explicit` | (SPARC) 明示的な先読みマクロを無効にします。 |
| `latx:`*factor* | 指定された factor によってコンパイラで使用されるロードするための先読みと、ストアするための先読みを調整します。このフラグは、`-xprefetch=auto` とのみ組み合わせることができます。係数には必ず正の浮動小数点または整数を指定します。 |
| `yes` | 廃止。使用しないでください。代わりに `-xprefetch=auto,explicit` を使用します。 |
| `no` | 廃止。使用しないでください。代わりに `-xprefetch=no%auto,no%explicit` を使用します。 |

`-xprefetch` および `-xprefetch=auto` を指定すると、コンパイラは生成するコードに自由に先読み命令を挿入します。その結果、先読みをサポートするアーキテクチャーでパフォーマンスが向上します。

大量の計算を行うコードを大規模なマルチプロセッサ上で実行している場合は、`-xprefetch=latx:`*factor* を使用するとパフォーマンスが向上する可能性があります。このオプションは、指定の係数により、先読みからロードまたはストアまでのデフォルトの応答時間を調整するようにコード生成プログラムに指示します。

先読みの応答時間とは、先読み命令を実行してから先読みされたデータがキャッシュで利用可能となるまでのハードウェアの遅延のことです。コンパイラは、先読み命令と先読みされたデータを使用するロードまたはストア命令の距離を決定する際に先読み応答時間の値を想定します。

注 – 先読みからロードまでのデフォルト応答時間は、先読みからストアまでのデフォルト応答時間と同じでない場合があります。

コンパイラは、幅広いマシンとアプリケーションで最適なパフォーマンスを得られるように先読み機構を調整します。しかし、コンパイラの調整作業が必ずしも最適であるとはかぎりません。メモリーに負担のかかるアプリケーション、特に大型のマルチプロセッサでの実行を意図したアプリケーションの場合、先読みの応答時間の値を引き上げることにより、パフォーマンスを向上できます。この値を増やすには、1 よりも大きい係数を使用します。.5 と 2.0 の間の値は、おそらく最高のパフォーマンスを提供します。

外部キャッシュの中に完全に常駐するデータセットを持つアプリケーションの場合は、先読み応答時間の値を減らすことでパフォーマンスを向上できる場合があります。値を減らすには、1 未満の係数を使用します。

`-xprefetch=latx:`*factor* オプションを使用するには、1.0 に近い係数の値から始め、アプリケーションに対してパフォーマンステストを実施します。そのあと、テストの結果に応じて係数を増減し、パフォーマンステストを再実行します。係数の調整を継続し、最適なパフォーマンスに到達するまでパフォーマンステストを実行します。係数を小刻みに増減すると、しばらくはパフォーマンスに変化がなく、突然変化し、再び平常に戻ります。

### A.2.161.1 デフォルト

デフォルトは `-xprefetch=auto,explicit` です。基本的に非線形のメモリーアクセスパターンを持つアプリケーションには、このデフォルトが良くない影響をもたらします。デフォルトを無効にするには、`-xprefetch=no%auto,no%explicit` を指定します。

`no%auto` か `no` の引数で明示的にオーバーライドされない限り、デフォルト `auto` が想定されます。たとえば、`-xprefetch=explicit` は `-xprefetch=explicit,auto` と同じことです。

デフォルト `explicit` は、引数に `no%explicit` か `no` を指定して明示的に無効にするまで継続されます。たとえば、`-xprefetch=auto` は `-xprefetch=auto,explicit` と同じことです。

`-xprefetch` だけを指定すると、`-xprefetch=auto,explicit` が使用されます。

自動先読みを有効にしても、応答時間係数を指定しないと、`-xprefetch=latx:1.0` が想定されます。

### A.2.161.2 相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

sun_prefetch.h ヘッダーファイルには、明示的な先読み命令を指定するためのマクロが含まれています。先読み命令は、実行コード中のマクロの位置にほぼ相当するところに挿入されます。

明示的な先読み命令を使用するには、使用するアーキテクチャーが適切なもので、sun_prefetch.h をインクルードし、かつコンパイラコマンドに -xprefetch が指定されていないか、-xprefetch、-xprefetch=auto,explicit、あるいは -xprefetch=explicit が指定されていなければいけません。

マクロを呼び出し、sun_prefetch.h ヘッダーファイルをインクルードしていても、-xprefetch=no%explicit を指定した場合は、明示的な先読みが実行可能ファイル内に組み込まれません。

latx:*factor* の使用は、自動先読みが有効になっている場合にのみ有効です。latx:*factor* は、-xprefetch=auto,latx:*factor* とともに使用していないかぎり無視されます。

### A.2.161.3 警告

明示的な先読みは、測定方法によってサポートされる特殊な環境でのみ使用してください。

コンパイラは、幅広いマシンやアプリケーションにわたって最適なパフォーマンスを得るために先読みメカニズムを調整するため、-xprefetch=latx:*factor* は、パフォーマンステストで明らかな利点があることが示された場合にのみ使用してください。想定される先読みの応答時間は、リリースごとに変更される可能性があります。したがって、別のリリースに切り替えたら、その都度応答時間係数の影響を再テストすることを推奨します。

## A.2.162 -xprefetch_auto_type= a

ここで *a* は [no%]indirect_array_access です。

このオプションは、コンパイラが -xprefetch_level オプションで示されたループのための間接先読みを、直接メモリーアクセスのための先読みが生成されるのと同じ方法で生成するかどうかを決定するために使用します。

-xprefetch_auto_type の値が指定されていない場合、-xprefetch_auto_type=no%indirect_array_access に設定されます。

-xdepend、-xrestrict、および -xalias_level などのオプションは、メモリー別名を明確化する情報の生成に役立つため、間接先読み候補の計算の攻撃性に影響し、このため、自動的な間接先読みの挿入が促進されることがあります。

## A.2.163 -xprefetch_level[=i]

-xprefetch_level=*i* オプションを使用して、-xprefetch=auto で決定した先読み命令の自動挿入の攻撃性を調整することができます。コンパイラは、-xprefetch_level のレベルが高くなるたびに、より積極的になります。つまり、より多くの先読みを導入します。

-xprefetch_level に適した値は、アプリケーションでのキャッシュミス数によって異なります。-xprefetch_level の値を高くするほど、キャッシュミスが多いアプリケーションの性能が向上する可能性が高くなります。

### A.2.163.1 値

*i* は、次の表に示す 1、2、3 のいずれかである必要があります。

表 A–42 -xprefetch_level の値

| 値 | 意味 |
|---|---|
| 1 | 先読み命令の自動的な生成を有効にします。 |
| 2 | -xprefetch_level=1 の対象以外にも、先読み挿入対象のループを追加します。-xprefetch_level=1. にある先読み以外に、追加の先読みを挿入できます。 |
| 3 | -xprefetch_level=2 の対象以外にも、先読み挿入対象のループを追加します。-xprefetch_level=2 にある先読み以外に、追加の先読みを挿入できます。 |

### A.2.163.2 デフォルト

デフォルトは、-xprefetch=auto を指定した場合は -xprefetch_level=1 になります。

### A.2.163.3 相互の関連性

このオプションは、-xprefetch=auto を使用して、3 以上の最適化レベル (-xO3) で、かつ先読みをサポートする 64 ビット SPARC プラットフォーム (-m64) 上でコンパイルされた場合にのみ有効です。

## A.2.164 –xprofile=p

プロファイルのデータを収集したり、プロファイルを使用して最適化したりします。

*p* には、collect[ :*profdir*]、use[ :*profdir*]、または tcov[ :*profdir*] を指定する必要があります。

このオプションを指定すると、実行頻度のデータが収集されて実行中に保存されます。このデータを以降の実行で使用すると、パフォーマンスを向上させることができます。プロファイルの収集は、マルチスレッド対応のアプリケーションにとって安全です。独自のマルチタスク (`-mt`) を実行するプログラムのプロファイリングによって、正確な結果が生成されます。このオプションは、最適化レベルを `-xO2` かそれ以上に指定するときにのみ有効になります。コンパイルとリンクを別々の手順で実行する場合は、リンク手順とコンパイル手順の両方で同じ `-xprofile` オプションを指定する必要があります。

`collect`[`:`*profdir*`]` 実行頻度のデータを集めて保存します。のちに `-xprofile=use` を指定した場合にオプティマイザがこれを使用します。コンパイラによって文の実行頻度を測定するためのコードが生成されます。

`-xMerge`、`-ztext`、および `-xprofile=collect` を一緒に使用しないでください。`-xMerge` を指定すると、静的に初期化されたデータを読み取り専用記憶領域に強制的に配置します。 `-ztext` を指定すると、位置に依存するシンボルを読み取り専用記憶領域内で再配置することを禁止します。`-xprofile=collect` を指定すると、書き込み可能記憶領域内で、静的に初期化された、位置に依存するシンボルの再配置を生成します。

プロファイルディレクトリ名として *profdir* を指定すると、この名前が、プロファイル化されたオブジェクトコードを含むプログラムまたは共有ライブラリの実行時にプロファイルデータが保存されるディレクトリのパス名になります。*profdir* パス名が絶対パスではない場合、プログラムがオプション `-xprofile=use:`*profdir* でコンパイルされるときの現在の作業用ディレクトリの相対パスとみなされます。

`–xprofile=collect:` *prof_dir* または `–xprofile=tcov:` *prof_dir* でプロファイルディレクトリ名を指定しない場合、プロファイルデータは実行時に、*program*`.profile` という名前のディレクトリに保存されます (*program* はプロファイリングされたプロセスのメインプログラムのベース名)。この場合は、環境変数 `SUN_PROFDATA` および `SUN_PROFDATA_DIR` を使用して、実行時にプロファイルデータが保存される場所を制御できます。設定する場合、プロファイルデータは `$SUN_PROFDATA_DIR/$SUN_PROFDATA` で指定されたディレクトリに書き込まれます。プロファイルディレクトリ名がコンパイル時に指定されても、実行時に `SUN_PROFDATA_DIR` および `SUN_PROFDATA` の効果はありません。これらの環境変数は、`tcov` で書き込まれるプロファイルデータファイルのパスと名前を同様に制御します。`tcov`(1) マニュアルページを参照してください。

これらの環境変数が設定されていない場合、プロファイルデータは現在のディレクトリ内のディレクトリ *profdir*.profile に書き込まれます (*profdir* は実行可能ファイルの名前または -xprofile=collect: *profdir* フラグで指定される名前)。*profdir* がすでに .profile で終わっている場合、-xprofile は .profile を *profdir* に追加しません。プログラムを複数回実行すると、実行頻度データは *profdir*.profile ディレクトリに蓄積されていくので、以前の実行頻度データは失われません。

別々の手順でコンパイルしてリンクする場合は、-xprofile=collect を指定してコンパイルしたオブジェクトファイルは、リンクでも必ず -xprofile=collect を指定してください。

次の例では、プログラムが構築されたディレクトリと同じディレクトリ内にある myprof.profile ディレクトリ内のプロファイルデータを収集して使用します。

```
demo: CC -xprofile=collect:myprof.profile -xO5 prog.cc -o prog
demo: ./prog
demo: CC -xprofile=use:myprof.profile -xO5 prog.cc -o prog
```

次の例では、/bench/myprof.profile ディレクトリ内のプロファイルデータを収集し、収集したプロファイルデータをあとでフィードバックコンパイルで最適化レベル -xO5 で使用します。

```
demo: CC -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.cc -o prog
...run prog from multiple locations..
demo: CC -xprofile=use:/bench/myprof.profile
\    -xO5 prog.cc -o prog
```

use[:*profdir* ]

プロファイリングされたコードが実行されたときに実行された作業のために最適化するときは、–xprofile=collect[: *profdir*] または –xprofile=tcov[: *profdir*] でコンパイルされたコードから収集された実行頻度データを使用します。*profdir* は、–xprofile=collect[: *profdir*] または –xprofile=tcov[: *profdir*] でコンパイルされたプログラムを実行して収集されたプロファイルデータを含むディレクトリのパス名です。

tcov と –xprofile=use[:*profdir* の両方で使用できるデータを生成するには、 –xprofile=tcov[:*profdir*] オプションを使用して、コンパイル時にプロファイルディレクトリを指定する必要があります。–xprofile=tcov: *profdir* と –xprofile=use: *profdir* の両方で同じプロファイルディレクトリを指定する必要があります。混乱を最小限に抑えるには、*profdir* を絶対パス名として指定します。

*profdir* パス名は省略可能です。 *profdir* が指定されていない場合、実行可能バイナリの名前が使用されます。`-o` が指定されていない場合、`a.out` が使用されます。 *profdir* が指定されていない場合、コンパイラは、`profdir .profile/feedback`、または *a.out.profile/feedback* を探します。例:

```
demo: CC -xprofile=collect -o myexe prog.cc
demo: CC -xprofile=use:myexe -xO5 -o myexe    prog.cc
```

`-xprofile=collect` オプションを付けてコンパイルしたときに生成され、プログラムの前の実行で作成されたフィードバックファイルに保存された実行頻度データを使用して、プログラムが最適化されます。

`-xprofile` オプションを除き、ソースファイルおよびコンパイラのほかのオプションは、フィードバックファイルを生成したコンパイル済みプログラムのコンパイルに使用したものと完全に同一のものを指定する必要があります。同じバージョンのコンパイラは、収集構築と使用構築の両方に使用する必要があります。

`-xprofile=collect:`*profdir* を付けてコンパイルした場合は、*-xprofile=use:* `profdir` のコンパイルの最適化に同じプロファイルディレクトリ名 *profdir* を使用する必要があります。

収集 (`collect`) 段階と使用 (`use`) 段階の間のコンパイル速度を高める方法については、`-xprofile_ircache` も参照してください。

`tcov[:`*profdir*`]`

`tcov`(1) を使用する基本のブロックカバレッジ分析用の命令オブジェクトファイル。

オプションの *profdir* 引数を指定すると、コンパイラは指定された場所にプロファイルディレクトリを作成します。プロファイルディレクトリに保存されたデータは、`tcov`(1) または `-xprofile=use:`*profdir* を付けたコンパイラで使用できます。オプションの *profdir* パス名を省略すると、プロファイル化されたプログラムの実行時にプロファイルディレクトリが作成されます。プロファイルディレクトリに保存されたデータは、`tcov`(1) でのみ使用できます。プロファイルディレクトリの場所は、環境変数 `SUN_PROFDATA` および `SUN_PROFDATA_DIR` を使用して指定できます。

*profdir* で指定された場所が絶対パス名でない場合は、コンパイル時に、コンパイルの時点での現在の作業用ディレクトリの相対パスであると解釈されます。いずれかのオブジェクトファイルに対して *profdir* が指定されている場合は、同じプログラム内

のすべてのオブジェクトファイルに対して同じ場所を指定する必要があります。場所が *profdir* で指定されているディレクトリには、プロファイル化されたプログラムを実行するときにすべてのマシンからアクセスできる必要があります。プロファイルディレクトリはその内容が必要なくなるまで削除できません。コンパイラでプロファイルディレクトリに保存されたデータは、再コンパイルする以外復元できません。

1 つ以上のプログラムのオブジェクトファイルが `-xprofile=tcov:/test/profdata` を使用してコンパイルされた場合は、`/test/profdata.profile` という名前のディレクトリがコンパイラによって作成され、プロファイリングされたオブジェクトファイルを表すデータを格納するために使用されます。実行時に同じディレクトリを使用して、プロファイル化されたオブジェクトファイルに関連付けられた実行データを保存できます。

`myprog` という名前のプログラムが `-xprofile=tcov` を使用してコンパイルされ、ディレクトリ `/home/joe` 内で実行された場合は、実行時にディレクトリ `/home/joe/myprog.profile` が作成され、実行時プロファイルデータを格納するために使用されます。

## A.2.165 -xprofile_ircache[ =path]

(*SPARC* のみ) `collect` 段階で保存されたコンパイルデータを再利用することによって `use` 段階のコンパイル時間を改善するには、`-xprofile_ircache`[= *path*] を `-xprofile=collect`| `use` とともに使用します。

大きなプログラムでは、中間データが保存されるため、`use` 段階のコンパイル時間の効率を大幅に向上させることができます。保存されたデータが必要なディスク容量を相当増やすことがある点に注意してください。

`-xprofile_ircache`[=*path*] を使用すると、*path* はキャッシュファイルが保存されているディレクトリを上書きします。デフォルトでは、これらのファイルはオブジェクトファイルと同じディレクトリに保存されます。`collect` と `use` 段階が 2 つの別のディレクトリで実行される場合は、パスを指定しておくと便利です。次の例は一般的なコマンドシーケンスを示します。

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out    // run collects feedback data
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

## A.2.166 -xprofile_pathmap

(*SPARC* のみ) `-xprofile=use` コマンドも指定する場合は、`-xprofile_pathmap=`*collect-prefix*`:`*use-prefix* オプションを使用します。次の両方の条件が当てはまり、かつコンパイラが `-xprofile=use` を使用してコンパイルされたオブジェクトファイルのプロファイルデータを見つけることができない場合は、`-xprofile_pathmap` を使用します。

- 前回オブジェクトファイルが `-xprofile=collect` でコンパイルされたディレクトリとは異なるディレクトリで、オブジェクトファイルを `-xprofile=use` を指定してコンパイルしている。
- オブジェクトファイルはプロファイルで共通ベース名を共有しているが、異なるディレクトリのそれぞれの位置で相互に識別されている。

*collect-prefix* は、オブジェクトファイルが `-xprofile=collect` を使用してコンパイルされたときのディレクトリツリーの UNIX パス名の接頭辞です。

*use-prefix* は、オブジェクトファイルが `-xprofile=use` を使用してコンパイルされるディレクトリツリーの UNIX パス名の接頭辞です。

`-xprofile_pathmap` の複数のインスタンスを指定すると、コンパイラは指定した順序でインスタンスを処理します。`-xprofile_pathmap` のインスタンスで指定された各 *use-prefix* は、一致する *use-prefix* が識別されるか、または最後に指定された *use-prefix* がオブジェクトファイルのパス名に一致しないことが確認されるまで、オブジェクトファイルのパス名と比較されます。

## A.2.167 -xreduction

自動的な並列化を実行するときにループの縮約を解析します。このオプションは、`-xautopar` が指定する場合のみ有効です。それ以外の場合は、コンパイラは警告を発行します。

縮約の認識が有効になっている場合、コンパイラは内積、最大値発見、最小値発見などの縮約を並列化します。これらの縮約によって非並列化コードによる四捨五入の結果と異なります。

## A.2.168 -xregs=r[,r...]

生成コード用のレジスタの使用法を指定します。

*r* には、`appl`、`float`、`frameptr` サブオプションのいずれか 1 つ以上をコンマで区切って指定します。

サブオプションの前に no% を付けるとそのサブオプションは無効になります。例:
-xregs=appl,no%float

–xregs サブオプションは、特定のハードウェアプラットフォームでしか使用できません。

表 A–43 -xregs のサブオプション

| 値 | 意味 |
|---|---|
| appl | (SPARC) コンパイラがアプリケーションレジスタをスクラッチレジスタとして使用してコードを生成することを許可します。アプリケーションレジスタは次のとおりです。<br>g2、g3、g4 ( 32 ビットプラットフォーム)<br>g2、g3 (64 ビットプラットフォーム)<br>すべてのシステムソフトウェアとライブラリを -xregs=no%appl を使用してコンパイルしてください。システムソフトウェア (共有ライブラリを含む) は、アプリケーション用のレジスタの値を保持する必要があります。これらの値は、コンパイルシステムによって制御されるもので、アプリケーション全体で整合性が確保されている必要があります。<br>SPARC ABI では、これらのレジスタはアプリケーションレジスタと記述されています。これらのレジスタを使用すると必要なロードおよびストア命令が少なくてすむため、パフォーマンスが向上します。ただし、アセンブリコードで記述された古いライブラリプログラムとの間で衝突が起きることがあります。 |
| float | (SPARC) コンパイラが浮動小数点レジスタを整数値用のスクラッチレジスタとして使用してコードを生成することを許可します。浮動小数点値を使用する場合は、このオプションとは関係なくこれらのレジスタを使用します。浮動小数点レジスタへのすべての参照をコードからなくす場合は、-xregs=no%float を使用するとともに、コードで浮動小数点型をどのような方法でも使用しないようにする必要があります。 |

表 A–43 -xregs のサブオプション (続き)

| 値 | 意味 |
|---|---|
| frameptr | (*x86* のみ) コンパイラでフレームポインタレジスタ (IA32 上では %ebp、AMD64 上では %rbp) を汎用レジスタとして使用できるようにします。<br><br>デフォルトは -xregs=no%frameptr です。<br><br>–features=no%except によって例外も無効になっているのでなければ、C++ コンパイラは –xregs=frameptr を無視します。-xregs=frameptr は -fast の一部ですが、-features=no%except も同時に指定されないかぎり C++ コンパイラによって無視されます。<br><br>-xregs=framptr を使用すると、コンパイラは浮動小数点レジスタを自由に使用できるので、プログラムのパフォーマンスが向上します。ただし、この結果としてデバッガおよびパフォーマンス測定ツールの一部の機能が制限される場合があります。スタックトレース、デバッガ、およびパフォーマンスアナライザは、–xregs=frameptr を使用してコンパイルされた機能についてレポートできません。<br><br>さらに、Posix pthread_cancel() の C++ 呼び出しは、クリーンアップハンドラの検索に失敗します。<br><br>C または Fortran 関数から直接または間接的に呼び出された C++ 関数が例外をスローする可能性がある場合は、C、Fortran、および C++ が混在したコードを -xregs=frameptr を使用してコンパイルしてはいけません。このような言語が混在するソースコードを –fast でコンパイルする場合は、コマンド行の –fast オプションのあとに –xregs=no%frameptr を追加します。<br><br>64 ビットのプラットフォームでは利用できるレジスタが多いため、–xregs=frameptr でコンパイルすると、64 ビットコードよりも 32 ビットコードのパフォーマンスが向上する可能性が高くなります。<br><br>-xpg も指定されている場合、コンパイラは -xregs=frameptr を無視し、警告を表示します。また、-xkeepframe によって -xregs=frameptr が上書きされます。 |

SPARC のデフォルトは -xregs=appl,float です。

x86 のデフォルトは -xregs=no%frameptr です。

x86 システムでは、-xpg には -xregs=frameptr との互換性がないため、これらの 2 つのオプションは同時に使用できません。 -xregs=frameptr は -fast に含まれている点にも注意してください。

アプリケーションにリンクする共有ライブラリを目的に作成されたコードは、-xregs=no%appl,float を使用してコンパイルしてください。少なくとも、共有ライブラリとリンクするアプリケーションがこれらのレジスタの割り当てを認識するように、共有ライブラリがアプリケーションレジスタを使用する方法を明示的に示す必要があります。

たとえば、大局的な方法で (重要なデータ構造体を示すためにレジスタを使用するなど) レジスタを使用するアプリケーションは、ライブラリと確実にリンクするため、`-xregs=no%appl` なしでコンパイルされたコードを含むライブラリがアプリケーションレジスタをどのように使用するかを正確に特定する必要があります。

## A.2.169 -xrestrict[=f]

ポインタ値の関数パラメータを制限付きポインタとして扱います。*f* は、次の表に示されている値のいずれかである必要があります。

表 A–44 -xrestrict の値

| 値 | 意味 |
|---|---|
| `%all` | ファイル全体のすべてのポインタ型引数を制限付きとして扱います。 |
| `%none` | ファイル内のどのポインタ型引数も制限付きとして扱いません。 |
| `%source` | メインソースファイル内に定義されている関数のみ制限付きにします。インクルードファイル内に定義されている関数は制限付きにしません。 |
| *fn*[,*fn*...] | 1 つ以上の関数名のコンマ区切りのリスト。関数リストが指定された場合は、指定された関数内のポインタ型引数を制限付きとして扱います。詳細については、次の314 ページの「A.2.169.1 制限付きポインタ」を参照してください。 |

このコマンド行オプションは独立して使用できますが、最適化時に使用するのがもっとも適しています。

たとえば、次のコマンドは、ファイル `prog.c` 内のすべてのポインタパラメータを制限付きポインタとして扱います。

```
%CC -xO3 -xrestrict=%all prog.cc
```

次のコマンドは、ファイル `prog.c` 内の関数 `agc` 内のすべてのポインタパラメータを制限付きポインタとして扱います。

```
%CC -xO3 -xrestrict=agc prog.cc
```

C プログラミング言語の C99 標準では `restrict` キーワードが導入されましたが、このキーワードは現在の C++ 標準に含まれていません。一部のコンパイラでは、C99 の `restrict` キーワードのための C++ 言語拡張が適用されており、`__restrict` または `__restrict__` とも表記されます。ただし、Oracle Solaris Studio の C++ コンパイラには現在、この拡張はありません。`-xrestrict` オプションは、ソースコードで `restrict` キーワードを部分的に置き換えます。(このキーワードを使用しても、関数のすべて

のポインタ引数を restrict と宣言する必要があるわけではありません。)このキーワードは、主に最適化の機会に影響を与え、関数に渡すことができる引数を制限します。ソースコードから restict または __restrict のすべてのインスタンスを削除しても、プログラムの見た目の動作には影響しません。

デフォルトは %none で、-xrestrict と指定すると -xrestrict=%sourcel と指定した場合と同様の結果が得られます。

### A.2.169.1 制限付きポインタ

コンパイラが効率よくループを並列化できるようにするには、左辺値が記憶領域の特定の領域を示している必要があります。別名とは、記憶領域の決まった位置を示していない左辺値のことです。オブジェクトへの 2 個のポインタが別名であるかどうかを判断することは困難です。これを判断するにはプログラム全体を解析することが必要であるため、非常に時間がかかります。次の例にある関数 vsq() について考えてみます。

```
extern "C"
void vsq(int n, double *a, double *b) {
    int i;
    for (i=0; i<n; i++) {
            b[i] = a[i] * a[i];
    }
}
```

ポインタ a および b が異なるオブジェクトをアクセスすることをコンパイラが知っている場合には、ループ内の異なる繰り返しを並列に実行することができます。しかし、ポインタ a および b でアクセスされるオブジェクトが重なりあっていれば、ループを安全に並列実行できなくなります。

コンパイル時に、コンパイラは、関数 vsq() を単純に解析しても a と b でアクセスされるオブジェクトが重なりあっているかどうかを知ることはできません。コンパイラがこの情報を得るために、プログラム全体の解析が必要になることがあります。コマンド行オプション -xrestrict[=*func1*,...,*funcn*] を使用して、ポインタ値の関数パラメータを制限付きポインタとして扱うように指定できます。関数リストが指定されている場合は、指定された関数内のポインタパラメータが制限付きとして扱われます。それ以外の場合は、ソースファイル全体にあるすべてのポインタパラメータが制限付きとして扱われます (推奨されません)。たとえば、-xrestrict=vsq を使用すると、関数 vsq() についての例では、ポインタ a および b が修飾されます。

ポインタ引数を制限付きとして宣言する場合は、ポインタが個別のオブジェクトを指定すると明示することになります。コンパイラは、a および b が個別の記憶領域を指していると想定します。この別名情報によって、コンパイラはループの並列化を実行することができます。

-xrestrict は正しく使用するようにしてください。区別できないオブジェクトを指しているポインタを制限付きポインタにしてしまうと、ループを正しく並列化できなくなり、不定な動作をすることになります。たとえば、b[i] と a[i+1] が同じオブ

ジェクトである場合のように、関数 vsq() のポインタ a と b が重なりあっているオブジェクトを指しているとします。このとき a および b が制限付きポインタとして宣言されていなければ、ループは順次実行されます。a と b が誤って制限付きポインタとして修飾された場合、コンパイラはループの実行を並列化する可能性があります。b[i+1] は b[i] が計算されたあとでのみ計算されるべきであるため、これは安全ではありません。

## A.2.170 -xs

オブジェクト (.o) ファイルなしに dbx でデバッグできるようにします。

このオプションを指定すると、すべてのデバッグ情報が実行可能ファイルにコピーされます。このオプションは、dbx のパフォーマンスやプログラムの実行時のパフォーマンスにほとんど影響を与えませんが、取得するディスク容量は増えます。

このオプションは、-xdebugformat=stabs で指定した場合だけ影響があります。この場合のデフォルトは、デバッグデータを実行可能ファイルにコピーしません。デフォルトのデバッグ形式である -xdebugformat=dwarf を使用する場合は、デバッグデータは常に実行可能ファイルにコピーされます。コピーを防止するオプションはありません。

## A.2.171 -xsafe=mem

(*SPARC* のみ) メモリー保護違反が発生しないことをコンパイラで想定できるようにします。

このオプションを使用すると、コンパイラでは SPARC V6 アーキテクチャーで違反のないロード命令を使用できます。

### A.2.171.1 相互の関連性

このオプションは、最適化レベルの -xO5 と、次のいずれかの値の -xarch を組み合わせた場合にだけ有効です。m32 と m64 の両方で sparc、sparcvis、-sparcvis2、または -sparcvis3。

### A.2.171.2 警告

アドレスの位置合わせが合わない、またはセグメンテーション侵害などの違反が発生した場合は違反のないロードはトラップを引き起こさないので、このオプションはこのような違反が起こる可能性のないプログラムでしか使用しないでください。ほとんどのプログラムではメモリーに関するトラップは起こらないので、大多数のプログラムでこのオプションを安全に使用できます。例外条件の処理にメモリーベースのトラップを明示的に使用するプログラムでは、このオプションは使用しないでください。

## A.2.172 -xspace

SPARC: コードサイズが大きくなるような最適化を行いません。

## A.2.173 -xtarget=t

命令セットと最適化処理の対象システムを指定します。

コンパイラにハードウェアシステムを正確に指定すると、プログラムによってはパフォーマンスが向上します。プログラムのパフォーマンスが重要な場合は、対象となるハードウェアを正確に指定してください。これは特に、新しい SPARC プロセッサ上でプログラムを実行する場合に当てはまります。しかし、ほとんどのプログラムおよび旧式の SPARC システムではパフォーマンスの向上はわずかであるため、汎用的な指定方法で十分です。

*t* には次の値のいずれかを指定します。`native`、`generic`、`native64`、`generic64`、*system-name*。

`-xtarget` に指定する値は、`-xarch`、`-xchip`、`-xcache` の各オプションの値に展開されます。実行中のシステムで `-xtarget=native` の展開を調べるには、`-xdryrun` コマンドを使用します。

たとえば、`-xtarget=ultraT2` は次のものと同等です: `-xarch=sparcvis2 -xchip=ultraT2 -xcache=8/16/4:4096/64/16`

---

注 – 特定のホストプラットフォームで `-xtarget` を展開した場合、そのプラットフォームでコンパイルすると `-xtarget=native` と同じ `-xarch`、`-xchip`、または `-xcache` 設定にならない場合があります。

---

### A.2.173.1 プラットフォームごとの -xtarget の値

この節では、`–xtarget` 値をプラットフォームごとに説明します。次の表は、すべてのプラットフォーム向けの `–xtarget` の値を一覧表示します。

表 A–45 すべてのプラットフォームでの -xtarget の値

| 値 | 意味 |
|---|---|
| `native` | 次と同等です。<br>`–m32 –xarch=native –xchip=native –xcache=native`<br>ホスト 32 ビットシステムに最高のパフォーマンスを与えます。 |

表 A–45 すべてのプラットフォームでの -xtarget の値 (続き)

| 値 | 意味 |
|---|---|
| native64 | 次と同等です。<br>–m64 –xarch=native64 –xchip=native64 –xcache=native64<br>ホスト 64 ビットシステムに最高のパフォーマンスを与えます。 |
| generic | 次と同等です。<br>–m32 –xarch=generic –xchip=generic –xcache=generic<br>ほとんどの 32 ビットシステムに最高のパフォーマンスを与えます。 |
| generic64 | 次と同等です。<br>–m64 –xarch=generic64 –xchip=generic64 –xcache=generic64<br>ほとんどの 64 ビットシステムに最高のパフォーマンスを与えます。 |
| *system-name* | 指定するシステムで最高のパフォーマンスが得られます。<br>対象となる実際のシステムを表すシステム名を、次のリストから選択してください。 |

## SPARC プラットフォームの -xtarget の値

SPARC または UltraSPARC V9 での 64 ビット Solaris ソフトウェアのコンパイルは、-m64 オプションで指定します。-xtarget を native64 または generic64 以外のフラグとともに指定する場合は、-xtarget=ultra ... -m64 のように、-m64 オプションも指定する必要があります。それ以外の場合、コンパイラは 32 ビットメモリーモデルを使用します。

表 A–46 SPARC アーキテクチャーでの -xtarget の展開

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| ultra | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra1/140 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra1/170 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra1/200 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra2 | sparcvis | ultra2 | 16/32/1:512/64/1 |
| ultra2/1170 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra2/1200 | sparcvis | ultra | 16/32/1:1024/64/1 |
| ultra2/1300 | sparcvis | ultra2 | 16/32/1:2048/64/1 |
| ultra2/2170 | sparcvis | ultra | 16/32/1:512/64/1 |

表 A–46 SPARC アーキテクチャーでの -xtarget の展開 (続き)

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| ultra2/2200 | sparcvis | ultra | 16/32/1:1024/64/1 |
| ultra2/2300 | sparcvis | ultra2 | 16/32/1:2048/64/1 |
| ultra2e | sparcvis | ultra2e | 16/32/1:256/64/4 |
| ultra2i | sparcvis | ultra2i | 16/32/1:512/64/1 |
| ultra3 | sparcvis2 | ultra3 | 64/32/4:8192/512/1 |
| ultra3cu | sparcvis2 | ultra3cu | 64/32/4:8192/512/2 |
| ultra3i | sparcvis2 | ultra3i | 64/32/4:1024/64/4 |
| ultra4 | sparcvis2 | ultra4 | 64/32/4:8192/128/2 |
| ultra4plus | sparcvis2 | ultra4plus | 64/32/4:2048/64/4:32768/64/4 |
| ultraT1 | sparcvis2 | ultraT1 | 8/16/4/4:3072/64/12/32 |
| ultraT2 | sparc | ultraT2 | 8/16/4:4096/64/16 |
| ultraT2plus | sparcvis2 | ultraT2plus | 8/16/4:4096/64/16 |
| T3 | sparcvis3 | T3 | 8/16/4:6144/64/24 |
| T4 | sparc4 | T4 | 16/32/4:128/32/8:4096/64/16 |
| sparc64vi | sparcfmaf | sparc64vi | 128/64/2:5120/64/10 |
| sparc64vii | sparcima | sparc64vii | 64/64/2:5120/256/10 |
| sparc64viiplus | sparcima | sparc64viiplus | 64/64/2:11264/256/11 |

## x86 プラットフォーム上の -xtarget の値

64 ビット x86 プラットフォームでの 64 ビット Solaris ソフトウェアのコンパイルは、-m64 オプションで指定します。native64 または generic64 以外のフラグで -xtarget を指定する場合は、-m64 オプションも次のように指定する必要があります: -xtarget=opteron ... -m64。それ以外の場合、コンパイラは 32 ビットメモリーモデルを使用します。

表 A–47 x86 プラットフォームでの -xtarget の値

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| opteron | sse2 | opteron | 64/64/2:1024/64/16 |
| pentium | 386 | pentium | generic |
| pentium_pro | pentium_pro | pentium_pro | generic |

表 A–47 x86 プラットフォームでの -xtarget の値 (続き)

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| pentium3 | sse | pentium3 | 16/32/4:256/32/4 |
| pentium4 | sse2 | pentium4 | 8/64/4:256/128/8 |
| nehalem | sse4_2 | nehalem | 32/64/8:256/64/8:8192/64/16 |
| penryn | sse4_1 | penryn | 2/64/8:4096/64/16 |
| woodcrest | ssse3 | core2 | 32/64/8:4096/64/16 |
| barcelona | amdsse4a | amdfam10 | 64/64/2:512/64/16 |
| sandybridge | avx | sandybridge | 32/64/8:256/64/8:8192/64/16 |
| westmere | aes | westmere | 32/64/8:256/64/8:12288/64/16 |

### A.2.173.2 デフォルト

SPARC および x86 で、-xtarget を指定しないと、-xtarget=generic が想定されます。

### A.2.173.3 展開

-xtarget オプションは、市販で購入したプラットフォーム上で使用する -xarch、-xchip、-xcache の組み合わせを素早く、簡単に指定するためのマクロです。-xtarget の意味は = のあとに指定した値を展開したものにあります。

### A.2.173.4 例

-xtarget=ultra は、-xchip=ultra -xcache=16/32/1:512/64/1 -xarch=sparcvis を示します。

### A.2.173.5 相互の関連性

-m64 オプションで示された 64 ビット SPARC V9 アーキテクチャー用のコンパイル。-xtarget=ultra または ultra2 の設定は必要でないか、または十分ではありません。-xtarget が指定されている場合は、-xarch、-xchip、または -xcache 値のすべての変更を -xtarget のあとに指定する必要があります。例:

```
–xtarget=ultra3 -xarch=ultra
```

### A.2.173.6 警告

別々の手順でコンパイルしてリンクする場合は、コンパイル手順とリンク手順で同じ -xtarget の設定値を使用する必要があります。

## A.2.174 -xthreadvar[=*o*]

スレッドローカルな変数の実装を制御するには -xthreadvar を指定します。コンパイラのスレッドローカルな記憶機能を利用するには、このオプションを __thread 宣言指定子と組み合わせて使用します。__thread 指示子を使用してスレッド変数を宣言したあとは、-xthreadvar を指定することにより、動的 (共有) ライブラリ内の位置依存コード (PIC 以外のコード) でスレッドローカルな記憶領域を使用できるようにします。__thread の使用方法についての詳細は、65 ページの「4.2 スレッドローカルな記憶装置」を参照してください。

### A.2.174.1 値

次の表に、*o* の指定可能な値を示します。

表 A–48 -xthreadvar の値

| 値 | 意味 |
|---|---|
| [no%]dynamic | 動的ロード用の変数をコンパイルします。-xthreadvar=no%dynamic を指定すると、スレッド変数へのアクセスは非常に早くなりますが、動的ライブラリ内のオブジェクトファイルは使用できません。すなわち、実行可能ファイル内のオブジェクトファイルだけが使用可能です。 |

### A.2.174.2 デフォルト

-xthreadvar を指定しない場合、コンパイラが使用するデフォルトは位置独立コード (PIC) が有効になっているかどうかによって決まります。位置独立コードが有効になっている場合、オプションは -xthreadvar=dynamic に設定されます。位置独立コードが無効になっている場合、オプションは -xthreadvar=no%dynamic に設定されます。

引数を指定しないで -xthreadvar を指定する場合、オプションは -xthreadvar=dynamic に設定されます。

### A.2.174.3 相互の関連性

-mt オプションは、__thread を使用しているファイルのコンパイルおよびリンクを実行するときに指定する必要があります。

### A.2.174.4 警告

動的ライブラリに位置独立でないコードが含まれている場合は、-xthreadvar を指定する必要があります。

リンカーは、動的ライブラリ内の位置依存コード (非 PIC) スレッド変数と同等のスレッド変数はサポートできません。非 PIC スレッド変数は非常に高速なため、実行可能ファイルのデフォルトとして指定できます。

### A.2.174.5 関連項目

`-xcode`、 `-KPIC`、 `-Kpic`

## A.2.175 –xtime

`CC` ドライバが、さまざまなコンパイル過程の実行時間を報告します。

## A.2.176 -xtrigraphs[={yes|no}]

ISO/ANSI C 標準の定義に従って文字表記シーケンスの認識を有効または無効にします。

コンパイラが文字表記シーケンスとして解釈している疑問符 (?) の入ったリテラル文字列がソースコードにある場合は、`-xtrigraph=no` サブオプションを使用して文字表記シーケンスの認識をオフにすることができます。

### A.2.176.1 値

`-xtrigraphs` に指定可能な値を次に示します。

表 A–49 -xtrigraphs の値

| 値 | 意味 |
|---|---|
| `yes` | コンパイル単位全体の 3 文字表記の認識を有効にします。 |
| `no` | コンパイル単位全体の 3 文字表記の認識を無効にします。 |

### A.2.176.2 デフォルト

コマンド行に `-xtrigraphs` オプションを指定しなかった場合、コンパイラは `-xtrigraphs=yes` を使用します。

`-xtrigraphs` だけを指定すると、コンパイラは `-xtrigraphs=yes` を使用します。

### A.2.176.3 例

`trigraphs_demo.cc` という名前のソースファイル例を考えてみましょう。

```
#include <stdio.h>

int main ()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

次の例は、-xtrigraphs=yes を使用してこのコードをコンパイルしたときの出力を示しています。

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as (]
```

次の例は、-xtrigraphs=no を使用してこのコードをコンパイルしたときの出力を示しています。

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

### A.2.176.4 関連項目

3 文字表記については、『C ユーザーズガイド』の ANSI/ISO C への移行に関する章を参照してください。

## A.2.177 -xunroll=n

このオプションはコンパイラに、可能な場合はループを展開して最適化するよう指示します。

### A.2.177.1 値

*n* が 1 の場合、コンパイラはループを展開しません。

*n* が 1 より大きな整数の場合は、-unroll=*n* によってコンパイラがループを *n* 回展開します。

## A.2.178 -xustr={ascii_utf16_ushort|no}

コンパイラにオブジェクトファイル内で UTF-16 文字列に変換させたい文字列リテラルまたは文字リテラルがコードに含まれる場合、を使用します。このオプションが指定されていない場合、コンパイラは 16 ビット文字列リテラルを生成することも認識することもしません。このオプションにより、U"*ASCII-string*" 文字列リテラルを unsigned short int の配列として認識できるようになります。このような文字列はまだどの標準にも含まれていないため、このオプションによって非標準 C++ の認識が可能になります。

すべてのファイルを、このオプションによってコンパイルしなければならないわけではありません。

### A.2.178.1 値

ISO10646 UTF--16 文字列リテラルを使用する国際化アプリケーションをサポートする必要がある場合、`-xustr=ascii_utf-16_ushort` を指定します。`-xustr=no` を指定すれば、コンパイラが U"*ASCII_string*" 文字列リテラルまたは文字リテラルを認識しなくなります。このオプションのコマンド行の右端にあるインスタンスは、それまでのインスタンスをすべて上書きします。

`-xustr=ascii_ustf16_ushort` の場合は、U"*ASCII-string*" 文字列リテラルを同時に指定しなくても指定できます。そのようにしても、エラーにはなりません。

### A.2.178.2 デフォルト

デフォルトは `-xustr=no` です。引数を指定しないで `-xustr` を指定した場合、コンパイラはこの指定を受け付けず、警告を出力します。C または C++ 標準で構文の意味が定義された場合は、デフォルトが変更される可能性があります。

### A.2.178.3 例

次の例では、前に U が付いた引用符内の文字列リテラルを示します。また、`-xustr` を指定するコマンド行も示されます。

```
example% cat file.cc
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() {return foo;}
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

8 ビットの文字列リテラルに `U` を付加して、`unsigned short` 型を持つ 16 ビットの UTF-16 文字を形成できます。例:

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

## A.2.179 -xvector[= a]

ベクトルライブラリ関数呼び出しの自動生成、または SIMD (Single Instruction Multiple Data) をサポートする x86 プロセッサ上での SIMD 命令の生成を可能にします。このオプションを使用するときは `-fround=nearest` を指定することによって、デフォルトの丸めモードを使用する必要があります。

`-xvector` オプションを指定するには、最適化レベルが `-xO3` かそれ以上に設定されていることが必要です。最適化レベルが指定されていない場合や –x03 よりも低い場合はコンパイルは続行されず、メッセージが表示されます。

*a* に指定可能な値を次の表に示します。`no%` 接頭辞は関連付けられたサブオプションを無効にします。

表 A–50 -xvector のサブオプション

| 値 | 意味 |
|---|---|
| [no%]lib | (*Solaris* のみ) コンパイラで、ループ内の数学ライブラリ呼び出しを同等のベクトル数学ルーチンへの 1 回の呼び出しに変換できるようにします (このような変換が可能な場合)。大きなループカウントを持つループでは、これによりパフォーマンスが向上します。このオプションを無効にするには no%lib を使用します。 |
| [no%]simd | (*x86* のみ) コンパイラに、特定のループのパフォーマンスを向上させるためにネイティブ x86 SSE SIMD 命令を使用するよう指示します。ストリーミング拡張機能は、x86 で最適化レベルが 3 かそれ以上に設定されている場合にデフォルトで使用されます。このオプションを無効にするには、no%simd を使用します。<br><br>コンパイラは、ストリーミング拡張機能がターゲットのアーキテクチャーに存在する場合、つまりターゲットの ISA が SSE2 以上である場合にのみ SIMD を使用します。たとえば、最新のプラットフォームで -xtarget=woodcrest、–xarch=generic64、 -xarch=sse2、 -xarch=sse3、または -fast を指定して使用できます。ターゲットの ISA にストリーミング拡張機能がない場合、このサブオプションは無効です。 |
| %none | このオプションを完全に無効にします。 |
| yes | このオプションは非推奨です。代わりに -xvector=lib を指定してください。 |
| no | このオプションは非推奨です。代わりに -xvector=%none を指定してください。 |

### A.2.179.1 デフォルト

デフォルトは、x86 では -xvector=simd で、SPARC プラットフォームでは -xvector=%none です。 -xvector をサブオプションなしで指定した場合、コンパイラでは x86 Solaris では -xvector=simd,lib が、SPARC Solaris では -xvector=lib、Linux プラットフォームでは -xvector=simd が使用されます。

### A.2.179.2 相互の関連性

コンパイラは、リンク時に libmvec ライブラリを取り込みます。

コンパイルとリンクを個別のコマンドで実行する場合は、リンク時の CC コマンドでも必ず -xvector を使用してください。

## A.2.180 -xvis[={yes|no}]

(*SPARC* のみ) VIS Software Developers Kit (VSDK) で定義されているアセンブリ言語のテンプレートを使用する場合、または VIS 命令と `vis.h` ヘッダーファイルを使用するアセンブラインラインコードを使用する場合は、`-xvis=[ yes|no]` コマンドを使用します。

VIS 命令セットは、SPARC v9 命令セットの拡張です。UltraSPARC プロセッサが 64 ビットの場合でも、多くの場合、特にマルチメディアアプリケーションではデータサイズが 8 ビットまたは 16 ビットに制限されています。VIS 命令では 1 つの命令で 4 ワードの 16 ビットデータを処理できるため、画像処理、線形代数、信号処理、オーディオ、ビデオ、ネットワーキングなどの新しいメディアを処理するアプリケーションのパフォーマンスが大幅に向上します。

### A.2.180.1 デフォルト

デフォルトは `-xvis=no` です。`-xvis` と指定すると `-xvis=yes` と指定した場合と同様の結果が得られます。

## A.2.181 -xvpara

OpenMP の使用時に正しくない結果をもたらす可能性のある、並列プログラミングに関連した潜在的な問題に関する警告を発行します。`-xopenmp` および OpenMP API 指令とともに使用します。

次の状況が検出された場合は、コンパイラは警告を発行します。

- 異なるループ繰り返し間でデータに依存関係がある場合に、MP 指令を使用して並列化されたループ。
- OpenMP データ共有属性節に問題がある場合。たとえば、「shared」と宣言された変数に、OpenMP 並列領域からアクセスするとデータ競合が発生する可能性がある場合や、並列領域の中に値を持つ変数を「private」と宣言し、並列領域よりあとでその変数を使用する場合です。

すべての並列化命令が問題なく処理される場合、警告は表示されません。

---

注 – Solaris Studio のコンパイラは OpenMP API の並列化をサポートしています。そのため、MP プラグマ命令は非推奨で、サポートされません。OpenMP API への移植については、『OpenMP API ユーザーズガイド』を参照してください。

---

## A.2.182 –xwe

ゼロ以外の終了状態を返すことによって、すべての警告をエラーとして扱います。

### A.2.182.1 関連項目

186 ページの「A.2.15 -errwarn[=t]」

## A.2.183 -Yc,path

構成要素 *c* がある場所の新しいパスを指定します。

コンポーネントの場所が指定されている場合、そのコンポーネントの新しいパス名は *path/component-name* です。このオプションは `ld` に渡されます。

### A.2.183.1 値

次の表に、*c* に指定可能な値を示します。

表 A–51 -Y のフラグ

| 値 | 意味 |
|---|---|
| P | `cpp` のデフォルトのディレクトリを変更します。 |
| 0 | `ccfe` のデフォルトのディレクトリを変更します。 |
| a | `fbe` のデフォルトのディレクトリを変更します。 |
| 2 | `iropt` のデフォルトのディレクトリを変更します。 |
| c (SPARC) | `cg` のデフォルトのディレクトリを変更します。 |
| O | `ipo` のデフォルトのディレクトリを変更します。 |
| k | `CClink` のデフォルトのディレクトリを変更します。 |
| l | `ld` のデフォルトのディレクトリを変更します。 |
| f | `c++filt` のデフォルトのディレクトリを変更します。 |
| m | `mcs` のデフォルトのディレクトリを変更します。 |
| u *(x86)* | `ube` のデフォルトのディレクトリを変更します。 |
| h *(x86)* | `ir2hf` のデフォルトのディレクトリを変更します。 |
| A | すべてのコンパイラ構成要素を検索するときのディレクトリを指定します。*path* にコンポーネントが見つからない場合は、コンパイラがインストールされているディレクトリに戻って検索が行われます。 |
| P | デフォルトのライブラリ検索パスにパスを追加します。デフォルトのライブラリ検索パスの前にこのパスが調べられます。 |
| S | 起動用のオブジェクトファイルのデフォルトのディレクトリを変更します。 |

### A.2.183.2 相互の関連性

コマンド行に複数の -Y オプションを配置できます。2 つ以上の -Y オプションが 1 つの項目に適用されている場合には、最後に指定されたものが有効です。

### A.2.183.3 関連項目

リンカーとライブラリ

## A.2.184 -z[ ]arg

リンクエディタのオプション。詳細は、ld(1) のマニュアルページおよび Oracle Solaris の『リンカーとライブラリ』を参照してください。

237 ページの「A.2.98 -Xlinker *arg*」も参照してください。

◆ ◆ ◆ 付録 B

# プラグマ

この付録では、プラグマについて説明します。プラグマとは、プログラマがコンパイラに特定の情報を渡すために使用するコンパイラ指令です。プラグマを使用すると、コンパイル内容を詳細に渡って制御できます。たとえば、`pack` プラグマを使用すると、構造体の中のデータの配置を変えることができます。プラグマは「指令」とも呼ばれます。

プリプロセッサキーワード `pragma` は C++ 標準の一部ですが、書式、内容、および意味はコンパイラごとに異なります。プラグマは C++ 標準には定義されていません。

---

注 – したがってプラグマに依存するコードには移植性はありません。プラグマに依存するコードは移植性がありません。

---

## B.1 プラグマの書式

次に、C++ コンパイラのプラグマのさまざまな書式を示します。

```
#pragma keyword
#pragma keyword ( a [ , a ] ...) [ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

変数 *keyword* は特定の指令を示し、*a* は引数を示します。

### B.1.1 プラグマの引数としての多重定義関数

ここで示すいくつかのプラグマは、引数として関数名をとります。その関数が多重定義されている場合、プラグマは、その引数として、その直前の関数宣言を使用します。次の例を考えてみましょう。

```
int bar(int);
int foo(int);
```

```
int foo(double);
#pragma does_not_read_global_data(foo, bar)
```

この例の foo は、プラグマの直前の foo の宣言である foo(double) を意味し、bar は、単に宣言されている bar である bar(int) を意味します。ここで、foo が再び多重定義されている次の例を考えてみます。

```
int foo(int);
int foo(double);
int bar(int);
#pragma does_not_read_global_data(foo, bar)
```

この例の bar は、単に宣言されている bar である bar(int) を意味します。しかし、プラグマは、どのバージョンの foo を使用すべきか分かりません。この問題を解決するには、プラグマが使用すべき foo の定義の直後にプログラムを置く必要があります。

次のプラグマは、この節で説明した方法で選択を行います。

- does_not_read_global_data
- does_not_return
- does_not_write_global_data
- no_side_effect
- opt
- rarely_called
- returns_new_memory

# B.2 プラグマの詳細

この節では、C++ コンパイラにより認識されるプラグマキーワードについて説明します。

## B.2.1 #pragma align

#pragma align *integer*(*variable*[,*variable*...])

align を使用すると、指定したすべての変数のメモリー境界を *integer* バイト境界に揃えることができます (デフォルト値より優先されます)。ただし、次の制限があります。

- *integer* は 1 と 128 の間の 2 の累乗である必要があります。有効な値は 1、2、4、8、16、32、64、および 128 です。
- *variable* は大域または静的変数です。これを局所変数またはクラスメンバー変数にすることはできません。
- 指定された境界がデフォルトより小さい場合は、デフォルトが優先します。

- プラグマ行は、指定した変数の宣言より前になければいけません。それ以外の場合は無視されます。
- この #pragma 行で指定されていても、プラグマ行に続くコードの中で宣言されない変数は、すべて無視されます。次に、正しく宣言されている例を示します。

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

#pragma align を名前空間内で使用するときは、符号化された名前を使用する必要があります。たとえば、次のコード中の、#pragma align 文には何の効果もありません。この問題を解決するには、#pragma align 文の a、b、および c を符号化された名前に変更します。

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

## B.2.2 #pragma does_not_read_global_data

```
#pragma does_not_read_global_data(funcname[, funcname])
```

このプラグマは、指定したルーチンが直接的にも間接的にも大域データを読み込まないことをコンパイラに表明することで、そのようなルーチンの呼び出しに関するコードをさらに最適化することが可能です。具体的には、代入文やストア命令をそうした呼び出しの前後に移動することができます。

このプラグマを使用できるのは、指定した関数のプロトタイプを宣言したあとに限定されます。大域アクセスに関する表明が真でない場合は、プログラムの動作は未定義になります。

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.3 #pragma does_not_return

```
#pragma does_not_return(funcname[, funcname])
```

このプラグマは、指定した関数への呼び出しが復帰しないことをコンパイラに表明することで、コンパイラはその仮定と一致する最適化を実行できます。たとえば、レジスタの有効期間が呼び出し元で終了し、これがより多くの最適化を可能にします。

指定した関数が復帰した場合は、プログラムの動作は未定義になります。

次の例のとおり、このプラグマを使用できるのは、指定した関数のプロトタイプを宣言したあとだけです。

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.4 #pragma does_not_write_global_data

`#pragma does_not_write_global_data(`*funcname*`[, `*funcname*`])`

このプラグマは、指定したルーチンが直接的にも間接的にも大域データを書き込まないことをコンパイラに表明することで、そのようなルーチンの呼び出しに関するコードをさらに最適化することが可能です。具体的には、代入文やストア命令をそうした呼び出しの前後に移動することができます。

このプラグマを使用できるのは、指定した関数のプロトタイプを宣言したあとに限定されます。大域アクセスに関する表明が真でない場合は、プログラムの動作は未定義になります。

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.5 #pragma dumpmacros

`#pragma dumpmacros (`*value*`[,`*value*`...])`

マクロがプログラム内でどのように動作しているかを調べたいときに、このプラグマを使用します。このプラグマは、定義済みマクロ、解除済みマクロ、実際の使用状況といった情報を提供します。マクロの処理順序に従って、標準エラー (`stderr`) に出力します。`dumpmacros` プラグマは、ファイルが終わるまで、または `#pragma end_dumpmacro` に到達するまで、有効です。333 ページの「B.2.6 #pragma end_dumpmacros」を参照してください。次の表に、*value* の可能な値を示します。

| 値 | 意味 |
|---|---|
| `defs` | すべての定義済みマクロを出力します |
| `undefs` | すべての解除済みマクロを出力します |

| 値 | 意味 |
|---|---|
| use | 使用されているマクロの情報を出力します |
| loc | defs、undefs、use の位置 (パス名と行番号) も出力します |
| conds | 条件付き指令で使用したマクロの使用情報を出力します |
| sys | システムヘッダーファイルのマクロについて、すべての定義済みマクロ、解除済みマクロ、使用状況も出力します |

注 – サブオプション loc、conds、sys は、オプション defs、undefs、use の修飾子です。loc、conds、sys は、単独では効果はありません。たとえば #pragma dumpmacros(loc,conds,sys) には、何も効果はありません。

dumpmacros プラグマとコマンド行オプションの効果は同じですが、プラグマはコマンド行オプションをオーバーライドします。257 ページの「A.2.116 -xdumpmacros[=*value*[,*value*...]]」を参照してください。

dumpmacros プラグマは入れ子にならないので、次のコードでは #pragma end_dumpmacros が処理されるとマクロ情報の出力が停止します。

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(defs, undefs)
...
#pragma end_dumpmacros
```

dumpmacros プラグマの効果は累積的です。次のものは、

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

次と同じ効果を持ちます。

```
#pragma dumpmacros(defs, undefs, loc)
```

オプション #pragma dumpmacros(use,no%loc) を使用した場合、使用したマクロそれぞれの名前が一度だけ出力されます。オプション #pragma dumpmacros(use,loc) を使用した場合、マクロを使用するたびに位置とマクロ名が出力されます。

## B.2.6 #pragma end_dumpmacros

```
#pragma end_dumpmacros
```

このプラグマは、dumpmacrospragma が終わったことを通知し、マクロ情報の出力を停止します。dumpmacros プラグマ終了時に end_dumpmacros プラグマを使用しなかった場合、dumpmacros プラグマはファイルが終わるまで出力を生成し続けます。

## B.2.7 #pragma error_messages

`#pragma error_messages (on|off|default, tag… tag)`

このエラーメッセージプラグマは、ソースプログラムの中から、コンパイラが発行するメッセージを制御可能にします。プラグマは警告メッセージにのみ効果があります。`-w` コマンド行オプションは、すべての警告メッセージを無効にすることでこのプラグマを上書きします。

- `#pragma error_messages (on, tag… tag)`

  `on` オプションは、先行する `#pragma error_messages` オプション (`off` オプションなど) のスコープを終了して、`-erroff` オプションの効果をオーバーライドします。

- `#pragma error_messages (off, tag… tag)`

  `off` オプションは、コンパイラプログラムが指定トークンから始まる特定のメッセージを発行することを禁止します。この特定のエラーメッセージに対するプラグマの指定は、別の `#pragma error_messages` によって無効にされるか、コンパイルが終了するまで有効です。

- `#pragma error_messages (default, tag… tag)`

  `default` オプションは、指定タグについて、先行する `#pragma error_messages` 指令を無効にします。

## B.2.8 #pragma fini

`#pragma fini (identifier[,identifier...])`

`fini` を使用すると、*identifier* を「終了関数」にします。この関数は `void` 型で、引数を持ちません。この関数は、プログラム制御によってプログラムが終了する時、または関数内の共有オブジェクトがメモリーから削除されるときに呼び出されます。初期設定関数と同様に、終了関数はリンカーが処理した順序で実行されます。

ソースファイル内で `#pragma fini` で指定された関数は、そのファイルの中にある静的デストラクタのあとに実行されます。identifier は、この #pragma で指定する前に宣言しておく必要があります。

このような関数は `#pragma fini` 指令の中に登場するたびに、1 回呼び出されます。

## B.2.9 #pragma hdrstop

`hdrstop` プラグマをソースファイルヘッダーに埋め込むと、活性文字列の終わりが指示されます。たとえば次のファイルがあるとします。

```
example% cat a.cc
#include "a.h"
#include "b.h"
```

```
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.cc
#include "a.h"
#include "b.h"
#include "c.h"
```

活性文字列は c.h で終わるので、各ファイルの c.h の後に #pragma hdrstop を挿入します。

#pragma hdrstop を挿入できる場所は、CC コマンドで指定したソースファイルの活性文字列の終わりだけです。#pragma hdrstop をインクルードファイル内に指定しないでください。

293 ページの「A.2.156 -xpch=*v*」および 297 ページの「A.2.157 -xpchstop=*file*」を参照してください。

## B.2.10 #pragma ident

```
#pragma ident string
```

ident を使用すると、実行可能ファイルの .comment 部に、*string* に指定した文字列を記述できます。

## B.2.11 #pragma init

```
#pragma init(identifier[,identifier...])
```

init を使用すると、*identifier* (識別子) を「初期設定関数」にします。この関数は void 型で、引数を持ちません。この関数は、実行開始時にプログラムのメモリーイメージを構築する時に呼び出されます。共有オブジェクトの初期設定子の場合、共有オブジェクトをメモリーに入れるとき、つまりプログラムの起動時または dlopen() のような動的ロード時のいずれかに実行されます。初期設定関数の呼び出し順序は、静的と動的のどちらの場合でもリンカーが処理した順序になります。

ソースファイル内で #pragma init で指定された関数は、そのファイルの中にある静的コンストラクタのあとに実行されます。identifier は、この #pragma で指定する前に宣言しておく必要があります。

このような関数は #pragma init 指令の中に登場するたびに、1 回呼び出されます。

## B.2.12 #pragma ivdep

ivdep プラグマは、最適化の目的でループ内で検出された、配列参照へのループがもたらす依存関係の一部またはすべてを無視するようにコンパイラに指示します。これによってコンパイラは、マイクロベクトル化、分散、ソフトウェアパイプラインなど、それ以外の場合は不可能なさまざまなループ最適化を実行できます。これは、依存関係が重要ではない、または依存関係が実際に発生しないことをユーザーが把握している場合に使用されます。

#pragma ivdep 指令の解釈は、–xivdep オプションの値に応じて異なります。

## B.2.13 #pragma must_have_frame

#pragma must_have_frame(*funcname*[,*funcname*])

このプラグマは、(System V ABI で定義されているとおり) 完全なスタックフレームを必ず持つように、指定した関数リストをコンパイルすることを要求します。このプラグマで関数を列挙する前に、関数のプロトタイプを宣言する必要があります。

```
extern void foo(int);
extern void bar(int);
#pragma must_have_frame(foo, bar)
```

このプラグマを使用できるのは、指定した関数のプロトタイプの宣言後のみに限定されます。プラグマは関数の最後より先に記述する必要があります

```
void foo(int) {
  .
  #pragma must_have_frame(foo)
  .
  return;
  }
```

329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.14 #pragma no_side_effect

#pragma no_side_effect(*name*[,*name*...])

no_side_effect は、関数によって持続性を持つ状態が変更されないことを通知するためのものです。このプラグマは、指定された関数がどのような副作用も起こさないことをコンパイラに宣言します。つまり、これらの関数は、渡された引数だけに依存する結果値を返します。さらに、これらの関数と、そこから呼び出される関数は、次のように動作します。

- 呼び出し時点で呼び出し側が認識できるプログラム状態の一部に、読み出しまたは書き込みのためにアクセスすることはありません。

- 入出力を実行しません。
- 呼び出し時点で認識できるプログラム状態のどの部分も変更しません。

コンパイラは、この情報を最適化に使用します。

関数に副作用があると、この関数を呼び出すプログラムの実行結果は未定義になります。

*name* 引数で、現在の翻訳単位に含まれている関数の名前を指定します。プラグマは関数と同じスコープ内になければならず、また、関数宣言後に位置していなければいけません。プラグマは、関数定義の前に位置していなければいけません。

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.15 #pragma opt

`#pragma opt` *level* (*funcname[, funcname]*)

*funcname* には、現在の翻訳単位内で定義されている関数の名前を指定します。*level* の値は、指定した関数に対する最適化レベルです。0、1、2、3、4、5 いずれかの最適化レベルを割り当てることができます。*level* を 0 に設定すると、最適化を無効にできます。関数は、プラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。プラグマは、最適化する関数の定義を処理する必要があります。

プラグマ内に指定される関数の最適化レベルは、`-xmaxopt` の値に下げられます。`-xmaxopt=off` の場合、プラグマは無視されます。

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.16 #pragma pack( n)

`#pragma pack([`*n*`])`

`pack` は、構造体メンバーの配置制御に使用します。

*n* を指定する場合は、0 または 2 の累乗にする必要があります。0 以外の値を指定すると、コンパイラは *n* バイトの境界整列と、データ型に対するプラットフォームの自然境界のどちらか小さい方を使用します。たとえば次の指令は、自然境界整列が 4 バイトまたは 8 バイト境界である場合でも、指令のあと (および後続の `pack` 指令の前) に定義されているすべての構造体のメンバーを 2 バイト境界を超えないように厳密にそろえます。

```
#pragma pack(2)
```

*n* が 0 であるか省略された場合、メンバー整列は自然境界整列の値に戻ります。

*n* の値がプラットフォームのもっとも厳密な境界整列と同じかそれ以上の場合には、自然境界整列になります。次の表に、各プラットフォームのもっとも厳密な境界整列を示します。

表 B–1 プラットフォームのもっとも厳密な境界整列

| プラットフォーム | もっとも厳密な境界整列 |
|---|---|
| x86 | 4 |
| SPARC 全般 | 8 |
| 64 ビット SPARC V9 (`-m64`) | 16 |

pack 指令は、次の pack 指令までに存在するすべての構造体定義に適用されます。別々の翻訳単位にある同じ構造体に対して異なる配置制御が指定されると、プログラムは予測できない状態で異常終了する場合があります。特に、コンパイル済みライブラリのインタフェースを定義するヘッダーをインクルードする場合は、その前に pack を使用しないでください。プログラムコード内では、pack 指令は境界整列を指定する構造体の直前に置き、`#pragma pack()` は構造体の直後に置くことをお勧めします。

SPARC プラットフォーム上で `#pragma pack` を使用して、型のデフォルトの境界整列よりも密に配置するには、アプリケーションのコンパイルとリンクの両方で `-misalign` オプションを指定する必要があります。次の表に、整数データ型のメモリーサイズとデフォルトの境界整列を示します。

表 B–2 メモリーサイズとデフォルトの境界整列 (単位はバイト数)

| 型 | 32 ビット SPARC<br>サイズ、境界整列 | 64 ビット SPARC<br>サイズ、境界整列 | x86<br>サイズ、境界整列 |
|---|---|---|---|
| `bool` | 1, 1 | 1, 1 | 1, 1 |
| `char` | 1, 1 | 1, 1 | 1, 1 |
| `short` | 2, 2 | 2, 2 | 2, 2 |
| `wchar_t` | 4, 4 | 4, 4 | 4, 4 |
| `int` | 4, 4 | 4, 4 | 4, 4 |
| `long` | 4, 4 | 8, 8 | 4, 4 |
| `float` | 4, 4 | 4, 4 | 4, 4 |
| `double` | 8, 8 | 8, 8 | 8, 4 |
| `long double` | 16, 8 | 16, 16 | 12, 4 |

表 B–2　メモリーサイズとデフォルトの境界整列 (単位はバイト数)　(続き)

| 型 | 32 ビット SPARC<br>サイズ、境界整列 | 64 ビット SPARC<br>サイズ、境界整列 | x86<br>サイズ、境界整列 |
|---|---|---|---|
| データへのポインタ | 4, 4 | 8, 8 | 4, 4 |
| 関数へのポインタ | 4, 4 | 8, 8 | 4, 4 |
| メンバーデータへのポインタ | 4, 4 | 8, 8 | 4, 4 |
| メンバー関数へのポインタ | 8, 4 | 16, 8 | 8, 4 |

## B.2.17 #pragma rarely_called

`#pragms rarely_called(`*funcname[, funcname]*`)`

このプラグマは、指定の関数がほとんど呼び出されないことをコンパイラに示唆することで、プロファイル収集フェーズのオーバーヘッドを生じさせることなくそのようなルーチンの呼び出し側でのプロファイルフィードバックスタイルの最適化をコンパイラが実行できるようにします。このプラグマはヒントの提示ですので、コンパイラは、このプラグマに基づく最適化を行わないこともあります。

`#pragma rarely_called` プリプロセッサ指令を使用できるのは、指定の関数のプロトタイプが宣言されたあとだけです。次は、`#pragma rarely_called` の例です。

```
extern void error (char *message);
#pragma rarely_called(error)
```

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.18 #pragma returns_new_memory

`#pragma returns_new_memory(`*name*`[,`*name*`...])`

このプラグマは、指定した関数が新しく割り当てられたメモリーのアドレスを返し、そのポインタがほかのポインタの別名として使用されないことをコンパイラに宣言します。この情報によって、オプティマイザはポインタ値の追跡性が向上し、メモリー位置を明確化できるため、結果としてスケジューリングとパイプラインが改善されます。

このプラグマの宣言が実際には誤っている場合は、該当する関数を呼び出したプログラムの実行結果は保証されません。

*name* 引数で、現在の翻訳単位に含まれている関数の名前を指定します。プラグマは関数と同じスコープ内になければならず、また、関数宣言後に位置していなければいけません。プラグマは、関数定義の前に位置していなければいけません。

プラグマがその引数として多重定義関数を処理する方法の詳細は、329 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

## B.2.19 #pragma unknown_control_flow

```
#pragma unknown_control_flow(name[,name...])
```

`unknown_control_flow` を使用すると、手続き呼び出しの通常の制御フロー属性に違反するルーチンの名前のリストを指定できます。たとえば、`setjmp()` の直後の文は、ほかのどんなルーチンを呼び出してもそこから返ってくることができます。これは、`longjmp()` を呼び出すことによって行います。

このようなルーチンを使用すると標準のフローグラフ解析ができないため、呼び出す側のルーチンを最適化すると安全性が確保できません。このような場合に #pragma unknown_control_flow を使用すると安全な最適化が行えます。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

## B.2.20 #pragma weak

```
#pragma weak name1 [= name2]
```

`weak` を使用すると、弱い (weak) 大域シンボルを定義できます。このプラグマは主にソースファイルの中でライブラリを構築するために使用されます。リンカーは弱いシンボルを認識できなくてもエラーメッセージを出しません。

`weak` プラグマは、次の 2 つの書式でシンボルを指定できます。

- 文字列書式。文字列は C++ 変数または関数の符号化名である必要があります。無効な符号化名が指定された場合、その名前を参照したときの動作は予測できません。無効な符号化名を参照した場合、コンパイラがエラーを生成しないこともあります。エラーを生成するかどうかにかかわらず、無効な符号化名を使用したときのコンパイラの動作は予測できません。
- 識別子書式。識別子は以前コンパイル単位で宣言された C++ 関数の明確な識別子である必要があります。識別子書式は変数には使用できません。無効な識別子への参照を検出した場合、フロントエンド (`ccfe`) はエラーメッセージを生成します。

### B.2.20.1 #pragma weak name

#pragma weak *name* という書式の指令は、*name* を弱い (weak) シンボルに定義します。*name* のシンボル定義が見つからない場合、リンカーは指示を出しません。また、弱いシンボル定義を複数見つけた場合も、リンカーは警告しません。リンカーは単に最初に検出した定義を使用します。

プラグマ name の強い定義が存在しない場合、リンカーはシンボルの値を 0 にします。

次の指令は、ping を弱いシンボルに定義しています。ping という名前のシンボルの定義が見つからない場合でも、リンカーはエラーメッセージを生成しません。

```
#pragma weak ping
```

#### #pragma weak name1 = name2

#pragma weak *name1* = *name2* という書式の指令は、シンボル *name1* を *name2* への弱い参照として定義します。*name1* がどこにも定義されていない場合、*name1* の値は *name2* の値になります。*name1* が別の場所で定義されている場合、リンカーはその定義を使用し、*name2* への弱い参照は無視します。次の指令では、bar がプログラムのどこかで定義されている場合、リンカーはすべての参照先を bar に設定します。そうでない場合、リンカーは bar への参照を foo にリンクします。

```
#pragma weak bar = foo
```

識別子書式では、*name2* は現在のコンパイル単位内で宣言および定義しなければいけません。例:

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

文字列書式を使用する場合、シンボルはあらかじめ宣言されている必要はありません。次の例において、_bar と bar の両方が extern "C" である場合、その関数はあらかじめ宣言されている必要はありません。しかし、bar は同じオブジェクト内で定義されている必要があります。

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

#### 関数の多重定義

識別子書式を使用するとき、プラグマのあるスコープ中には指定した名前を持つ関数は、1 つしか存在できません。多重定義関数に識別子書式の #pragma weak を使おうとすると、エラーになります。例:

```
int bar(int);
float bar(float);
#pragma weak bar        // error, ambiguous function name
```

このエラーを回避するには、文字列書式を使用します。例を次に示します。

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

詳細は、Oracle Solaris の『リンカーとライブラリ』を参照してください。

# 用語集

| | |
|---|---|
| **ABI** | 「アプリケーションバイナリインタフェース」を参照。 |
| **ANSI C** | ANSI (米国規格協会) による C プログラミング言語の定義。ISO (国際標準化機構) 定義と同じです。「ISO」を参照。 |
| **ANSI/ISO C++** | 米国規格協会と国際標準化機構が共同で作成した C++ プログラミング言語の標準。「ISO」を参照。 |
| **cfront** | C++ を C ソースコードに変換する C++ から C へのコンパイルプログラム。変換後の C ソースコードは、標準の C コンパイラでコンパイルできます。 |
| **ELF** ファイル | ELF は Executable and Linking Format の略語で、コンパイラによって生成されるファイル。 |
| **ISO** | 国際標準化機構。 |
| **K&R C** | Brian Kernighan と Dennis Ritchie によって開発された、ANSI C 以前の事実上の C プログラミング言語標準。 |
| **VTABLE** | 仮想関数を持つクラスごとにコンパイラが作成するテーブル。 |
| アプリケーションバイナリインタフェース | コンパイルされたアプリケーションとそのアプリケーションが動作するオペレーティングシステム間のバイナリシステムインタフェース。 |
| インクリメンタルリンカー | 変更された `.o` ファイルだけを古い実行可能ファイルにリンクして新しい実行可能ファイルを作成するリンカー。 |
| インスタンス化 | C++ コンパイラが、テンプレートから使用可能な関数やオブジェクト (インスタンス) を生成する処理。 |
| インスタンス変数 | 特定のオブジェクトに関連付けられたデータ項目。クラスの各インスタンスは、クラス内で定義されたインスタンス変数の独自のコピーを持っています。フィールドとも呼びます。「クラス変数」も参照。 |
| インライン関数 | 関数呼び出しを実際の関数コードに置き換える関数。 |
| 右辺値 | 代入演算子の右辺にある変数。右辺値は読み取れますが、変更はできません。 |
| 演算子の多重定義 | 同じ演算子表記を異なる種類の計算に使用できること。関数の多重定義の特殊な形式の 1 つです。 |
| オプション | 「コンパイラオプション」を参照。 |

| | |
|---|---|
| 型 | シンボルをどのように使用するかを記述したもの。基本型は `integer` と `float` です。ほかのすべての型は、これらの基本型を配列や構造体にしたり、ポインタ属性や定数属性などの修飾子を加えることによって作成されます。 |
| 関数の多相性 | 「関数の多重定義」参照。 |
| 関数の多重定義 | 扱う引数の型と個数が異なる複数の関数に、同じ名前を与えること。関数の多相性ともいいます。 |
| 関数のテンプレート | ある関数を作成し、それをひな型として関連する関数を作成するための仕組み。 |
| 関数プロトタイプ | 関数とプログラムの残りの部分とのインタフェースを記述する宣言。 |
| キーワード | プログラミング言語で固有の意味を持ち、その言語において特殊な文脈だけで使用可能な単語。 |
| 局所変数 | ブロック内のコードからはアクセスできるが、ブロック外のコードからはアクセスできないデータ項目。たとえば、メソッド内で定義された変数は局所変数であり、メソッドの外からは使用できません。 |
| クラス | 名前が付いた一連のデータ要素 (型が異なっていてもよい) と、そのデータを処理する一連の演算からなるユーザーの定義するデータ型。 |
| クラステンプレート | 一連のクラスや関連するデータ型を記述したテンプレート。 |
| クラス変数 | クラスの特定のインスタンスではなく、特定のクラス全体を対象として関連付けられたデータ項目。クラス変数はクラス定義中に定義されます。静的フィールドとも呼びます。「インスタンス変数」も参照。 |
| 継承 | プログラマが既存のクラス (基底クラス) から新しいクラス (派生クラス) を派生させることを可能にするオブジェクト指向プログラミングの機能。継承の種類には、公開、限定公開、および非公開があります。 |
| コンストラクタ | クラスオブジェクトを作成するときにコンパイラによって自動的に呼び出される特別なクラスメンバー関数。これによって、オブジェクトのインスタンス変数が初期化されます。コンストラクタの名前は、それが属するクラスの名前と同じでなければなりません。「デストラクタ」を参照。 |
| コンパイラオプション | コンパイラの動作を変更するためにコンパイラに与える命令。たとえば、`-g` オプションを指定すると、デバッガ用のデータが生成されます。同義語: フラグ、スイッチ。 |
| 最適化 | コンパイラが生成するオブジェクトコードの効率を良くする処理のこと。 |
| サブルーチン | 関数のこと。Fortran では、値を返さない関数を指します。 |
| 左辺値 | 変数のデータ値が格納されているメモリーの場所を表す式。あるいは、代入演算子の左辺にある変数のインスタンス。 |
| 事後束縛 | 「動的束縛」を参照。 |
| 事前束縛 | 「静的束縛」を参照。 |

| | |
|---|---|
| 実行時型識別機構 **(RTTI)** | プログラムが実行時にオブジェクトの型を識別できるようにする標準的な方法を提供する仕組み。 |
| 実行時束縛 | 「動的束縛」を参照。 |
| シンボル | 何らかのプログラムエントリを示す名前やラベル。 |
| シンボルテーブル | プログラムのコンパイルで検出されたすべての識別子と、それらのプログラム中の位置と属性からなるリスト。コンパイラは、このテーブルを使って識別子の使い方を判断します。 |
| スイッチ | 「コンパイラオプション」を参照。 |
| スコープ | あるアクションまたは定義が適用される範囲。 |
| スタック | あと入れ先出し法によってデータをスタック のいちばん上にのみ追加、またはいちばん上からのみ削除できるデータ記憶方式。 |
| スタブ | オブジェクトコードに生成されるシンボルテーブルのエントリ。デバッグ情報を含む `a.out` ファイルと ELF ファイルには同じ形式のスタブが使用されます。 |
| 静的束縛 | 関数呼び出しと関数本体をコンパイル時に結び付けること。事前束縛とも呼びます。 |
| 束縛 | 関数呼び出しを特定の関数定義に関連付けること。一般的には、名前を特定のエントリに関連付けることを指します。 |
| 多相性 | ポインタや参照が、自分自身の宣言された型とは異なる動的な型を持つオブジェクトを参照できること。 |
| 多重継承 | 複数の基底クラスから 1 つの派生クラスを直接継承すること。 |
| 多重定義 | 複数の関数や演算子に同じ名前を指定すること。 |
| 抽象クラス | 1 つまたは複数の抽象メソッドを持つクラス。したがって、抽象クラスはインスタンス化できません。抽象クラスは、ほかのクラスが抽象クラスを拡張し、その抽象メソッドを実装することで具体化されることを目的として、定義されています。 |
| 抽象メソッド | 実装を持たないメソッド。 |
| データ型 | 文字、整数、浮動小数点数などを表現するための仕組み。変数に割り当てられる記憶域とその変数に対してどのような演算が実行可能かは、この型によって決まります。 |
| データメンバー | クラスの要素であるデータ。関数や型定義と区別してこのように呼ばれます。 |
| デストラクタ | クラスオブジェクトを破棄したり、演算子 `delete` をクラスポインタに適用したときにコンパイラによって自動的に呼び出される特別なクラスメンバー関数。デストラクタの名前は、それが属するクラスの名前と同じで、かつ、名前の前にチルド (~) が必要です。「コンストラクタ」を参照。 |
| テンプレートオプションファイル | テンプレートのコンパイル用オプションやソースの位置などの情報が含まれている、ユーザーが用意するファイル。テンプレートオプションファイルの使用は推奨されていないため、使用すべきではありません。 |

| | |
|---|---|
| テンプレートデータベース | プログラムが必要とするテンプレートの処理とインスタンス化に必要なすべての構成ファイルを含むディレクトリ。 |
| テンプレートの特殊化 | デフォルトのインスタンス化では型を適切に処理できないときに、このデフォルトを置き換える、クラステンプレートメンバー関数の特殊インスタンス。 |
| 動的キャスト | ポインタや参照の型を、宣言されたものから、それが参照する動的な型と矛盾しない任意の型に安全に変換するための方法。 |
| 動的束縛 | 関数呼び出しと関数本体を実行時に結び付けること。これは、仮想関数に対してのみ行われます。事後束縛または実行時束縛とも呼ばれます。 |
| 動的な型 | ポインタや参照でアクセスするオブジェクトの実際の型。この型は、宣言された型と異なることがあります。 |
| トラップ | ほかの処置をとるためにプログラムの実行などの処置を遮ること。これによって、マイクロプロセッサの演算が一時的に中断され、プログラム制御がほかのソースに渡されます。 |
| 名前空間 | 大域空間を一意の名前を持つスコープに分割して、大域的な名前のスコープを制御する仕組み。 |
| 名前の符号化 | C++ では多くの関数が同じ名前を持つことがあるため、名前だけでは関数を区別できません。コンパイラは、関数名とパラメータの組み合わせで構成される一意の名前を各関数について作成すること、つまり名前の符号化によってこの問題を解決します。この方法によって、型保証されたリンケージが可能になります。「名前修飾」とも呼びます。 |
| バイナリ互換 | あるリリースのコンパイラでコンパイルしたオブジェクトファイルを別のリリースのコンパイラを使用してリンクできること。 |
| 配列 | 同じデータ型の値をメモリーに連続して格納するデータ構造。各値にアクセスするには、配列内のそれぞれの値の位置を指定します。 |
| 符号化する | 「名前の符号化」を参照。 |
| フラグ | 「コンパイラオプション」を参照。 |
| プラグマ | コンパイラに特定の処置を指示するコンパイラのプリプロセッサ命令、または特別な注釈。 |
| べき等 | ヘッダーファイルの属性。ヘッダーファイルを 1 つの翻訳単位に何回インクルードしても、一度インクルードした場合と同じ効果を持つこと。 |
| 変数 | 識別子で命名されているデータ項目。各変数は `int` や `void` などの型とスコープを持っています。「クラス変数」、「インスタンス変数」、「局所変数」も参照。 |
| マルチスレッド | シングルまたはマルチプロセッサシステムで並列アプリケーションの開発を可能にするソフトウェア技術。 |
| メソッド | 一部のオブジェクト指向言語でメンバー関数の代わりに使用される用語。 |
| メンバー関数 | クラスの要素である関数。データ定義や型定義と区別されます。 |

| | |
|---|---|
| リンカー | オブジェクトコードとライブラリを結び付けて、完全な実行可能プログラムを作成するツール。 |
| 例外 | プログラムの通常の流れの中で起こる、プログラムの継続を妨げるエラー。たとえば、メモリーの不足やゼロ除算などを指します。 |
| 例外処理 | エラーの捕捉と防止を行うためのエラー回復処理。具体的には、プログラムの実行中にエラーが検出されると、あらかじめ登録されている例外ハンドラにプログラムの制御が戻り、エラーを含むコードは実行されなくなることを指します。 |
| 例外ハンドラ | エラーを処理するために作成されたコード。ハンドラは、対象とする例外が起こると自動的に呼び出されます。 |
| ロケール | 地理的な領域または言語に固有な一連の規約。日付、時刻、通貨単位などの形式。 |
| 基底クラス | 「継承」を参照。 |
| 派生クラス | 「継承」を参照。 |

# 索引

## D



## E



## F



## G

## M



## N



## O



## P



## Q



## R

## S



## T



## U

## V



## W



## X

## Z



## あ



## い



## え



## お

## け



## こ



## さ



## し

## わ