C++ ユーザーズガイド

Sun WorkShop 6

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-4837-01
2000 年 6 月 Revision A

原典 : *C++ User's Guide*
Part No: 806-3572-10
Revision A

製品名の変更について

Sun は新しい開発製品戦略の一環として、Sun の開発ツール群の製品名を Sun WorkShop™ から Forte™ Developer に変更いたしました。製品自体の内容に変更はなく、従来通りの高品質をお届けいたします。

これまでの Sun の主力製品である基本プログラミングツールに、Forte Fusion™ や Forte™ for Java™ といった Forte 開発ツールの得意とする、マルチプラットフォームおよびビジネスアプリケーション実装の機能を盛り込むことで、より広範囲できめ細かな製品ラインが完成されました。

WorkShop 5.0 で使用されていた名称と、Forte Developer 6 で使用される新しい名称の対応については、以下の表をご覧ください。

旧名称	新名称
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

製品名の変更に加えて、次の 2 つの製品について大きな変更があります。

- Forte for High Performance Computing には Sun Performance WorkShop Fortran に含まれていたすべてのツール、および C++ コンパイラが含まれます。したがって、High Performance Computing のユーザーは開発用に 1 つの製品だけを購入すれば済むことになります。
- Forte Fortran Desktop Edition は以前の Sun Performance WorkShop Personal Edition と同じです。ただし、この製品に含まれる Fortran コンパイラでは、自動並列化されたコード、および明示的な指令に基づいた並列コードは生成できません。この機能は Forte for High Performance Computing に含まれる Fortran コンパイラでは使用できます。

Sun の開発製品を引き続きご利用いただきましてありがとうございます。今後もみなさまのご要望にお応えする製品をお届けできるよう努力してまいります。

目次

表目次

はじめに

このマニュアルでは、Sun WorkShop™ 6 C++ コンパイラの使用方法を説明し、コマンド行オプションについての詳細な情報を記載します。このマニュアルは C++ および Solaris™ / UNIX® に関する実用的な知識を持つプログラマを対象にしています。

マルチプラットフォーム対応

この Sun WorkShop リリースは、Solaris 2.6、7、および 8 のオペレーティング環境 (SPARC™ プラットフォームおよび Intel プラットフォーム) をサポートしています。

注 – IA アーキテクチャとは、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon プロセッサおよび、これらと互換性のある AMD および Cyrix 製のマイクロプロセッサチップを含む、Intel 32 ビットプロセッサアーキテクチャを意味しています。

Sun WorkShop 開発ツールへのアクセス方法

Sun WorkShop 製品コンポーネントとマニュアルページは標準ディレクトリ `/usr/bin` および `/usr/share/man` にはインストールされません。そのため `PATH` および `MANPATH` 環境変数を変更して Sun WorkShop コンパイラとツールにアクセスできるようにする必要があります。

`PATH` 環境変数を設定する必要があるかどうか判断するには以下を実行します。

1. **次のように入力して、PATH 変数の現在値を表示します。**

```
% echo $PATH
```

2. **出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。**

パスがある場合は、PATH 変数は Sun WorkShop 開発ツールにアクセスできるように設定されています。パスがない場合は、この節の指示に従って、PATH 環境変数を設定してください。

MANPATH 環境変数を設定する必要があるかどうか判断するには以下を実行します。

1. **次のように入力して、workshop マニュアルページを表示します。**

```
% man workshop
```

2. **出力された場合、内容を確認します。**

workshop(1) マニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って MANPATH 環境変数を設定してください。

注 – この節に記載されている情報は Sun WorkShop 6 製品が /opt ディレクトリにインストールされていることを想定しています。 Sun WorkShop ソフトウェアが /opt ディレクトリにインストールされていない場合は、システム管理者に連絡してください。

PATH 変数および MANPATH 変数は、C シェルを使用している場合はホームディレクトリの下の .cshrc ファイルに設定する必要があります。Bourne シェルか Korn シェルを使用している場合は、ホームディレクトリの下の .profile ファイルに設定する必要があります。

- Sun WorkShop コマンドを使用するには、 PATH 変数に以下を追加してください。

  /opt/SUNWspro/bin

- man コマンドで、Sun WorkShop マニュアルページにアクセスするには、MANPATH 変数に以下を追加してください。

  /opt/SUNWspro/man

PATH 変数についての詳細は、csh(1)、sh(1) および ksh(1) のマニュアルページを参照してください。MANPATH 変数についての詳細は、 man(1) のマニュアルページを参照してください。このリリースにアクセスするために PATH および MANPATH 変数を設定する方法の詳細は、『Sun WorkShop インストールガイド』を参照するか、システム管理者にお問い合わせください。

内容の紹介

このマニュアルは次の章から構成されています。

- 第 1 章「C++ コンパイラの紹介」では、Sun C++ コンパイラの概要を説明します。
- 第 2 章「C++ コンパイラの使用方法」では、コンパイラの起動方法と一般的なコンパイルの流れについて説明します。
- 第 3 章「C++ コンパイラオプション」では、C++ コンパイラのオプションを詳しく説明します。ここには、オプションが機能別に分類されています。
- 第 4 章「テンプレートのコンパイル」では、テンプレートのコンパイル、定義の検索、インスタンスのリンケージなど、テンプレートの使用について説明します。
- 第 5 章「ライブラリの使用」では、多くの C++ ライブラリの使用方法を説明します。
- 第 6 章「ライブラリの構築」では、ライブラリ構築の流れを説明します。
- 「用語集」では、このマニュアルで使用されている用語を定義します。

書体と記号について

このマニュアルで使用している書体と記号について説明します。

表 P-1　このマニュアルで使用している書体と記号

書体または記号	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、コーディング例。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `machine_name% You have mail.`
`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表わします。	`machine_name% su` `Password:`
AaBbCc123 または ゴシック	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	`rm` *filename* と入力します。 `rm` **ファイル名** と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 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、バックスラッシュは、継続を示します。	`machinename% grep '^#define \` `  XV_VERSION_STRING'`
➤	階層メニューのサブメニューを選択することを示します。	作成： 「返信」➤「送信者へ」

シェルプロンプトについて

シェルプロンプトの例を以下に示します。

表 P-2　シェルプロンプト

シェル	プロンプト
UNIX の C シェル	machine_name%
UNIX の Bourne シェルと Korn シェル	machine_name$
スーパーユーザー (シェルの種類を問わない)	#

関連マニュアル

以下の方法で、関連マニュアルにアクセスすることができます。

- インターネットの docs.sun.com の Web サイトからアクセスできます。特定の本のタイトルで検索するか、主題、マニュアルコレクションまたは製品別にブラウズすることができます。

  http://docs.sun.com

- ローカルシステムまたはローカルネットワークにインストールされた Sun WorkShop 製品からアクセスできます。Sun WorkShop 6 HTML 文書 (マニュアル、オンラインヘルプ、マニュアルページ、各コンポーネントの README ファイル、リリースノート) が、インストールした Sun WorkShop 6 製品から参照可能です。HTML 文書にアクセスするには、次のいずれかを実行します。
  - Sun WorkShop または Sun WorkShop™ TeamWare ウィンドウで、「ヘルプ」➤「オンラインマニュアルについて」を選択します。
  - Netscape™ Communicator 4.0 またはその互換バージョンのブラウザで、以下のファイルを開きます。

    /opt/SUNWspro/docs/ja/index.html

    参照できる Sun WorkShop 6 HTML 文書の一覧がブラウザに表示されます。一覧にあるマニュアルを開くには、マニュアルのタイトルをクリックしてください。

表 P-3 は、Sun WorkShop 6 関連マニュアルをマニュアルコレクション別に一覧にしたものです。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Forte Developer 6 / Sun WorkShop 6 リリースマニュアル	Sun WorkShop 6 マニュアルの概要	Sun WorkShop 6 で使用可能なマニュアルとそのアクセス方法について説明しています。
	Sun WorkShop の新機能	Sun WorkShop 6 の現在のリリースと以前のリリースでの新機能についての情報を記載しています。
	Sun WorkShop 6 リリースノート	インストールの詳細と Sun WorkShop 6 最終リリースの直前に判明した情報を記載しています。このマニュアルはコンポーネントごとの README ファイルにある情報を補足するものです。
Forte Developer 6 / Sun WorkShop 6	プログラムのパフォーマンス解析	新しい標本コレクタと標本アナラザの使い方について説明しています (上級者向けのプロファイリング事例と説明付き)。コマンド行解析ツール `er_print`、ループツール、ループレポートユーティリティおよび UNIX プロファイルツール `prof`、`gprof`、`tcov` についての情報も含んでいます。
	dbx コマンドによるデバッグ	dbx コマンドを使ってプログラムをデバッグする方法について説明しています。参考情報として、同じデバッグ処理を Sun WorkShop デバッグウィンドウを使って実行する方法も記載しています。
	Sun WorkShop の概要	Sun WorkShop 統合プログラミング環境の基本的なプログラム開発機能について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
Forte C 6 / Sun WorkShop 6 Compilers C	C ユーザーズガイド	C コンパイラオプション、サン固有の機能 (プラグマ、 `lint` ツール、並列化、64 ビットオペレーティングシステムへの移行および ANSI/ISO 準拠 C) について説明しています。
Forte C++ 6 / Sun WorkShop 6 Compilers C++	C++ ライブラリ・リファレンス	C++ ライブラリについて説明しています。C++ 標準ライブラリ、Tools.h++ クラスライブラリ、Sun WorkShop Memory Monitor、`Iostream` および複素数の情報も含みます。
	C++ 移行ガイド	コードを本バージョンの Sun WorkShop C++ コンパイラに移行する方法について説明しています。
	C++ プログラミングガイド	新しい機能を使ってより効率的なプログラムを記述する方法について説明しています。テンプレート、例外処理、実行時の型識別、キャスト演算、パフォーマンス、およびマルチスレッド対応のプログラムに関する情報も記載されています。
	C++ ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	Sun WorkShop Memory Monitor ユーザーズガイド	C および C++ のメモリー管理で生じた問題を Sun WorkShop Memory Monitor で解決する方法について説明しています。このマニュアルはインストールした製品 (`/opt/SUNWspro/docs/ja/index.html`) からのみ参照可能で、`docs.sun.com` Web サイトで参照することはできません。
Forte for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	Fortran ライブラリ・リファレンス	Fortran コンパイラによって提供されるライブラリルーチンの詳細について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
	Fortran プログラミングガイド	入出力、ライブラリ、プログラム分析、デバッグおよびパフォーマンスに関連する内容を記述しています。
	Fortran ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	FORTRAN 77 言語リファレンス	Fortran 77 言語の包括的な参照情報を記載しています。
	Fortran 95 区間演算プログラミングリファレンス	Fortran 95 コンパイラによってサポートされる組み込み `INTERVAL` データについて説明しています。
Forte TeamWare 6 / Sun WorkShop TeamWare 6	Sun WorkShop TeamWare ユーザーズガイド	Sun WorkShop TeamWare コード管理ツールの使用方法について説明しています。
Forte Developer 6/ Sun WorkShop Visual 6	Sun WorkShop Visual ユーザーズガイド	C++ と Java™ の GUI (グラフィカルユーザーインタフェース) を Sun WorkShop Visual を使用して作成する方法について説明しています。
Forte / Sun Performance Library 6	Sun Performance Library Reference (英語のみ)	コンピュータによる線形代数および高速フーリエ変換を実行するサブルーチンと関数の最適化ライブラリについて説明しています。
	Sun Performance Library User's Guide (英語のみ)	線形代数で発生した問題の解決に使用されるサブルーチンと関数のコレクションである Sun Performance Library のサン固有の機能の使用方法について説明しています。
数値計算ガイド	数値計算ガイド	浮動小数点演算における数値の精度に関する問題について説明しています。
標準ライブラリ 2	Standard C++ Library Class Reference (英語のみ)	標準 C++ の詳細について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
	標準 C++ ライブラリ・ユーザーズガイド	標準 C++ ライブラリの使用方法について説明しています。
Tools.h++ 7	Tools.h++ 7.0 ユーザーズガイド	Tools.h++ クラスライブラリの詳細について説明しています。
	Tools.h++ 7.0 クラスライブラリ・リファレンスマニュアル	C++ クラスを使用して、プログラム効率を向上させる方法について説明しています。

表 P-4 は、docs.sun.com の Web サイトからアクセスできる Solaris 関連マニュアルの一覧です。

表 P-4　Solaris 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris ソフトウェア開発	リンカーとライブラリ	Solaris リンクエディタと実行時リンカーの操作およびそれらが操作するオブジェクトについて説明しています。
	プログラミングユーティリティ	Solaris オペレーティング環境で使用可能な特殊組み込みプログラミングツールに関する開発者向けの情報を記載しています。

マニュアルページ

C++ ライブラリに関するマニュアルページは『C++ ライブラリ・リファレンス』に記載されています。表 P-5 には、それ以外の C++ に関連するマニュアルページを示します。

表 P-5　C++ 関連のマニュアルページ

タイトル	内容
`c++filt`	ファイルを順番通りに読み、C++ の符号化された名前と思われるシンボルを復号化した後、標準出力に書き出す
`dem`	指定した複数の C++ 名の復号化
`fbe`	アセンブリ言語のソースファイルからオブジェクトファイルの作成
`fpversion`	システムの CPU と FPU に関する情報の出力
`gprof`	プログラムの実行プロファイルの作成
`ild`	プログラムの修正部分だけをリンクし、修正オブジェクトコードを以前に構築された実行可能ファイルに挿入することを可能にする
`inline`	インライン手続きの呼び出しの展開
`lex`	字句解析プログラムの生成
`rpcgen`	RPC プロトコルを実装するため C/C++ コードの生成
`sigfpe`	特定の SIGFPE コードに対するシグナル処理を許可
`stdarg`	変更可能な引数のリストを処理
`varargs`	変更可能な引数のリストを処理
`version`	オブジェクトファイルまたはバイナリファイルのバージョン識別情報の表示
`yacc`	文脈自由文法を、LALR(1) 構文解析アルゴリズムを実行する単純オートマトン用の一連の表に変換

README (最新情報) ファイル

README ファイルには以下のような、コンパイラに関する重要な情報が記載されています。

- 新しい機能および変更された機能
- ソフトウェアの非互換性に関する情報
- 現行ソフトウェアのバグ
- マニュアルの訂正

README ファイルを表示するには次のように入力します。

```
%example CC -xhelp=readme
```

HTML 形式の README ファイルを参照するには、Netscape Communicator 4.0 またはその互換バージョンのブラウザで、以下のファイルを開きます。

```
/opt/SUNWspro/docs/ja/index.html
```

注 – Sun WorkShop ソフトウェアが `/opt` ディレクトリにインストールされていない場合は、インストール先のディレクトリをシステム管理者に確認し、そのディレクトリを上記の `/opt` に置き換えてください。

参照できる Sun WorkShop 6 HTML 文書の一覧がブラウザに表示されます。README を参照するには、該当するタイトルをクリックしてください。

市販の書籍

C++ について書かれている書籍の一部を紹介します。

『注解 C++ リファレンス・マニュアル』トッパン、Margaret A. Ellis、Bjarne Stroustrup 共著、1990 年

『C++ プライマー』第 3 版、トッパン、Stanley B. Lippman、Josee Lajoie 共著、1998 年

『Effective C++—50 Ways to Improve Your Programs and Designs』Second Edition、Scott Meyers 著、Addison-wesley、1998 年

『The C++ Standard Library』Nicolai Josuttis 著、Addison-Wesley、1999 年

『Generic Programming and the STL』Matthew Austern 著、Addison-Wesley、1999 年

『Standard C++ IOStreams and Locales』Angelica Langer、Klaus Kreft 共著、Addison-Wesley、2000 年

『Thinking in C++』Volume 1、Second Edition、Bruce Eckel 著、Prentice Hall、1995 年

『Design Patterns: Elements of Reusable Object-Oriented Software』Erich Gamma、Richard Helm、Ralph Johnson,、John Vlissides 共著、Addison-Wesley、1998 年

『More Effective C++ - 35 Ways of Improve Your Programs and Designs』Scott Meyers 著、Addison-Wesley、1996 年

第1章

C++ コンパイラの紹介

本章では、Sun™ C++ および C++ コンパイラの概要を説明しています。

標準の準拠

この C++ コンパイラ (CC) は、『ISO International Standard for C++, ISO IS 14882:1998, Programming Language - C++』に準拠しています。このリリースに含まれる README (最新情報) ファイルには、この標準と異なる仕様に関する記述が含まれています。

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 によって SPARC International のために出版された SPARC アーキテクチャマニュアル(トッパン刊) の バージョン 8 と SPARC Architecture Manual Version (英語版のみ) のバージョン 9 (ISBN 0-13-099227-5) に定義されています。

このマニュアルでは、「標準」は、上記の標準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」や「拡張」は、これらの標準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、それぞれの標準を規定する組織によって改定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標準が改定されたり、まったく書き換えられた場合は、機能によっては、Sun 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。

オペレーティング環境

C++ コンパイラ (CC) は、Sun WorkShop™ や C コンパイラなどのサンの開発ツールと統合されています。Sun C++ コンパイラとその実行時ライブラリは、Sun Visual WorkShop™ C++ に含まれています。Sun Visual WorkShop C++ の構成要素を使用すれば、マルチプロセッサの Solaris™ 2.6、Solaris 7、Solaris 8 オペレーティング環境でスレッドを使用したアプリケーションを開発できます。

Sun WorkShop 6 C++ コンパイラは、Solaris 2.6、Solaris 7、Solaris 8 のいずれかのオペレーティング環境を実行する SPARC および IA プラットフォームで使用できます。

注 – 機能が特定のオペレーティング環境やハードウェアプラットフォーム固有である場合は、その旨を明示します。しかし、コンパイラの機能性や動作性には、システム間での違いはほとんどありません。マルチプロセッサ機能は、SPARC プラットフォームの Solaris 2.6、Solaris 7、Solaris 8 ソフトウェアにおける Sun WorkShop に含まれています。この機能には、Sun WorkShop ライセンスが必要です。

詳細は C++ の README ファイルを参照してください。

READMEs ディレクトリ

READMEs ディレクトリには、新しい機能やソフトウェアの互換性の問題、既知の問題点、および、このマニュアルの印刷後に明らかになった情報などについて記述したファイル (README ファイルと呼びます) が含まれています。標準インストールでは、README ファイルは /opt/SUNWspro/READMEs/ja にあります。

README ファイルは、どのコンパイラでも -xhelp=readme コマンド行オプションで簡単に表示できます。たとえば CC -xhelp=readme と入力すると、C++ の README ファイルが表示されます。

Netscape Communicator 4.0 (または、互換バージョン) ブラウザで HTML 版の README を表示するには、次のファイルを開きます。

/opt/SUNWspro/docs/ja/index.html

Sun WorkShop ソフトウェアが /opt ディレクトリにインストールされていない場合、システムのどこにインストールされているのかをシステム管理者に尋ねてください。ブラウザは SunWorkShop 6 HTML 文書の一覧を表示します。README を開くには、一覧の上の対応するタイトルをクリックします。

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 (*topic* には、参照したいコマンドやライブラリ関数の名前を指定)。

```
example% man topic
```

C++ のマニュアルで参考情報としてマニュアルページ名を記載する場合は、名前とセクション番号が示されています。CC(1) は、man CC で表示されます。その他のセクションのマニュアルページ、たとえば、ieee_flags(3M) は、man コマンドに -s オプションを使用すると表示されます。

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

ライセンス

C++ コンパイラでは、ネットワークライセンスを使用します。これについては、『Sun WorkShop のインストールとライセンス』を参照してください。

ライセンスがあれば、コンパイラを起動できます。ライセンスがない場合は、ライセンスの要求が待ち行列に入れられ、ライセンスを入手してからコンパイラを使用できるようになります。同じマシン上で同じユーザーであれば、1 ライセンスで同時に何回でもコンパイルできます。

C++ と一緒にほかのユーティリティを実行する場合には、購入したパッケージによっては、複数のライセンスが必要になる場合があります。

C++ コンパイラの新機能

この Sun Workshop 6 C++ コンパイラには次の新しい機能があります。

- 以下の内容を含む、ISO C++ 標準への準拠
  - クラステンプレートの部分的な特殊化
  - 明示的な関数テンプレート引数
  - メンバーテンプレート
  - 部分集合体の初期化
  - 外部インライン関数
  - 静的変数の破棄順序
- すべての `-instances` オプションに対する別定義のテンプレート編成が可能
- 先読み命令

C++ コンパイラのバージョン 5.0 で導入された機能は次のとおりです。

- 次の C++ ISO 標準の実装
  - 名前空間と Koenig の検索
  - `bool` 型
  - `new` 配列および `delete` 配列
  - テンプレートの拡張サポート
  - C++ 標準ライブラリ
  - 仮想関数の戻り型
- C++ 4.0、4.0.1、4.1、4.2 との互換性
- Sun WorkShop Memory Monitor によるガベージコレクションとメモリーリークの検出
- Solaris 7 および Solaris 8 オペレーティング環境での SPARC V9 サポート
- ISO C++ へのスムーズな移行に役立つバイナリおよびソース互換機能
- マルチスレッドに対して安全な C++ 標準ライブラリ

C++ コンパイラパッケージには、次のものも含まれています。

- オンラインの `README` ファイル
  新たに追加された機能、変更された機能、最新のソフトウェア、マニュアルの修正点、マニュアルの出版後に判明した情報が掲載されています。

- マニュアルページ
  ユーザーコマンドやライブラリ関数について簡潔に説明しています。
- C++ の名前を復号化するツール群 (`dem` および `c++filt`)
- `Tools.h++` クラスライブラリ
  プログラミングを単純化できます。

C++ ユーティリティ

現在、ほとんどの C++ ユーティリティは従来の UNIX ツールに統合され、オペレーティングシステムに含まれています。

- `lex` -- テキストの単純な字句解析に使用するプログラムを生成する。
- `yacc` -- 構文に応じて入力ストリームを解析するための C 関数を生成する。
- `prof` -- プログラム内のモジュールの実行プロファイルを作成する。
- `gprof` -- プログラムの実行時パフォーマンスについての手続き単位のプロファイル。
- `tcov` -- プログラムの実行時パフォーマンスについての文単位のプロファイル

これら UNIX ツールについての詳細は、『プログラムのパフォーマンス解析』や関連するマニュアルページを参照してください。

各国語のサポート

本バージョンの 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と日本語です。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

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、オペレーティング環境のマニュアル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++ コンパイラでは次のことができます。

- 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後、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ます。
- 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。

コンパイル方法の概要

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しい説明については、第 3 章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++ アプリケーションを構築して実行するには、基本的に次の手順が必要です。

1. エディタで C++ソースファイルを作成する。このソースファイルには、10 ページの表 2-1 に列挙されている接尾辞のいずれかを使用します。
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.C
```

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出力ファイル名にすることが一般的です)。

あるいは、コンパイルの後に 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、プログラムを実行するには、実行可能ファイルの名前を入力します。

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

コンパイラの起動

この後の節では、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。

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。*source-files* にはソースファイル、*object-files* にはオブジェクトファイル、*libraries* にはライブラリを指定します。

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 (他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。

- すべてのリンカーのオプション、-I、-L、-pti、および -R オプションは上書きされずに累積される
- -U オプションは、すべて -D オプションの後に処理される

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
```

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。
.i	C++	プリプロセッサの出力ファイルを C++ ソースファイルとして扱い、.c 接尾辞と同じ処理をする。
.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。

表 2-1　C++ コンパイラが認識できるファイル名接尾辞 (続き)

接尾辞	言語	処理
`.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する (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` `.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。

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。1 つのソースファイルには、手続き (メインプログラム、関数、モジュールなど) をいくつでも組み込めます。ただし、関連する手続きを 1 つのファイルに集めることに利点があるように、1 つのファイルに 1 つの手続きが入るようにアプリケーションを編成することにも利点があります。このことに関しては、『C++ プログラミングガイド』に説明があります。

バージョンが異なるコンパイラでのコンパイル

Sun WorkShop 6 C++ 以降のコンパイラはテンプレートキャッシュディレクトリにテンプレートキャッシュのバージョンを示す文字列を付けます。

コンパイラは、キャッシュディレクトリのバージョンを調べ、キャッシュのバージョンに問題があれば、エラーメッセージを発行します。将来の Sun WorkShop 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、次のエラーを発行します。

```
SunWS_cache: エラー : /SunWS_cache のテンプレートデータベースは
このコンパイラと互換性がありません
```

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。

Sun WorkShop C++ コンパイラ 5.0 で作成されたテンプレートキャッシュディレクトリにはバージョン識別子が付けられていません。しかし、Sun WorkShop 6 C++ コンパイラは、5.0 のキャッシュディレクトリをエラーや警告なしに処理できます。これは、Sun WorkShop 6 C++ コンパイラが 5.0 のキャッシュディレクトリを、Sun WorkShop 6 C++ コンパイラが使用できるディレクトリ形式に変換するためです。

Sun WorkShop C++ コンパイラ 5.0 は、Sun WorkShop 6 C++ コンパイラ (または、これ以降のリリース) で作成されたキャッシュディレクトリを使用できません。Sun WorkShop C++ コンパイラ 5.0 は形式の違いを認識できず、Sun WorkShop 6 C++ コンパイラ (または、これ以降のリリース) 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、エラーを発行します。

コンパイラをアップグレードするときは、テンプレートキャッシュディレクトリが格納されているディレクトリごとに `CCadmin -clean` を実行する習慣を付けることをお勧めします (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。

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
```

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。

オブジェクトファイル (file1.o、file2.o、file3.o) はコンパイルの後も消去されないので、ファイルを簡単に再リンクしたり、再コンパイルすることができます。

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。

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、次の手順でそれを他のファイルとリンクします。

```
example% CC  -c file1.cc                ←新しいオブジェクトファイルを作成する
example% CC  -o prgrm file1.o file2.o file3.o ←実行可能ファイルを作成する
```

リンク時には、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 (ルーチンがない) で失敗します。

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、次の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が非常に重要です。

- `-fast`
- `-g`
- `-g0`
- `-library`
- `-misalign`
- `-mt`
- `-p`
- `-xa`
- `-xarch=`*isa*
- `-xcg92` および `-xcg89`
- `-xpg`
- `-xprofile`
- `-xtarget=`*t*
- `-xvector` または `-xvector=yes`

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`、`-fast`、`-xarch` オプションの場合、コンパイルとリンクを同時に行えば渡されるはずのリンカーオプションも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。

次の例では、-xcg92 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。このオプションは -xtarget=ss1000 用のマクロであり、-xarch=v8 -xchip=super -xcache=16/64/4:1024/64/1 と展開されます。

```
example% CC -c -xcg92 sbr.cc
example% CC -c -xcg92 smain.cc
example% CC -xcg92 sbr.o smain.o
```

プログラムがテンプレートを使用する場合は、リンク時にその中のいくつか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。

SPARC V9 のためのコンパイル

64 ビットオブジェクトのコンパイル、リンク、実行には、V9 SPARC の Solaris 7 または Solaris 8 環境で 64 ビットカーネルが動作していなければなりません。64 ビットのコンパイルは、-xarch=v9 オプション、-xarch=v9a オプションまたは、-xarch=v9b オプションで指定します。

コンパイラの診断

-verbose オプションを使用すると、プログラムのコンパイル中に役立つ情報を表示できます。このオプションについては、第 3 章を参照してください。

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。

基本的な区別は次のとおりです。

- 認識できないオプション (先頭にダッシュ (-) かプラス符号 (+) の付いたもの) には、警告が生成されます。

- オプション以外のもの (先頭にダッシュ (-) もプラス符号 (+) も付いていないもの) には警告は生成されません (ただし、それらはリンカーに渡されます。リンカーも認識できないと、リンカーから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。しかし、これは本来の意図とは異なります。

```
example% CC -bit move.cc    <- -bit は CC オプションとして認識されない

CC: 警告 : ld が起動される場合は、オプション -bit は ld に渡されます。それ以
外は無視されます。
```

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      <- ユーザーは -fast と入力するつもりだった
move.C:
ld: 重大なエラー : ファイル fast: ファイルをオープンできません :
ファイルもディレクトリもありません。
ld: 重大なエラー: ファイル処理エラー。a.out へ書き込まれる出力がありません。
```

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。CC コマンドは、これらの構成要素をそれぞれ自動的に起動します (コマンド行オプションを使用して自動起動されないように指定することもできます)。図 2-1 に C++ コンパイルの流れを示します。図 2-1 に、構成要素がコンパイラから呼び出される順番を示します。

これらの構成要素はいずれもエラーを生成する可能性があり、構成要素はそれぞれ異なる処理を行うため、エラーを生成した構成要素を識別することがエラーの解決に役立つことがあります。

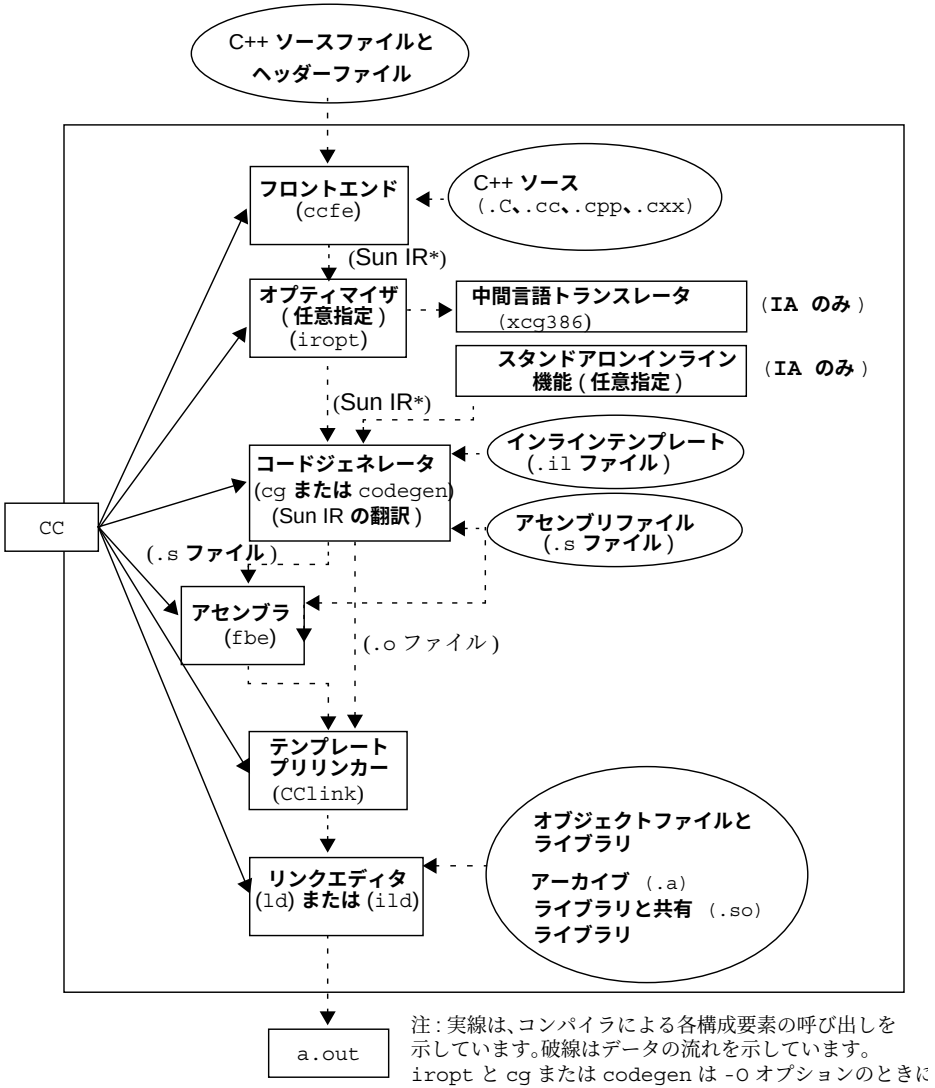

図 2-1　コンパイルの流れ

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。ファイル接尾辞の意味については、10 ページの表 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

構成要素	内容	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	(SPARC) コードオプティマイザ (最適化)	-xO[2-5]、-fast
xcg386	(IA) 中間言語トランスレータ	必ず起動
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
mwinline	(IA) 関数の自動的なインライン展開	-xO4
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	
codegen	(IA) コード生成	
CClink	テンプレートのリンクの前処理	
ld	従来のリンクエディタ	
ild	インクリメンタルリンクエディタ	-g、-xildon

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 -xO*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。その場合には、最適化レベルを下げるか、複数ルーチンからなるソースファイルを 1 つのルーチンからなるファイルに分割します。

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
```

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なければなりません)。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
```

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、非常に多く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。これを制御するには、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 16M バイトに制限しています。

```
example$ ulimit -d 16000
```

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 16M バイトに制限しています。

```
example% limit datasize 16M
```

どちらの例でも、オプティマイザは データ空間が 16M 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なければなりません。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なければなりません。

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。

スワップ領域が 32M バイトなら次のコマンドを使用します。

sh シェルの場合

```
example$ ulimit -d 16000
```

csh シェルの場合

```
example% limit datasize 16M
```

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。

メモリー条件

ワークステーションには、少なくとも 24M バイトのメモリーが必要です。32M バイトを推奨します。

実際のメモリーを調べるには、次のコマンドを使用します。

```
example% /usr/sbin/dmesg | grep mem
mem = 655360K (0x28000000)
avail mem = 602476544
```

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 make を使用すれば、複雑なコンパイラコマンドを簡略化できます。

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
```

上記のコマンド CCfx は、次のコマンドを実行するのと同じことです。

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

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 -xsb'
```

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 変数が上の例のように設定され、メー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、次のコンパイルが行われます (*files* は、複数のファイル名を示します)。

```
CC -xO2 -xsb files...
```

make の使用

make ユーティリティは、サン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。

make での CCFLAGS の使用

メー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++ コンパイル行がない) 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。

メークファイルへの接尾辞の追加

メークファイルに別のファイルの接尾辞を追加すると、C++ にその接尾辞を取り込むことができます。次の例は、C++ ファイルに対する有効な接尾辞として .cpp を追加します。次のように、メークファイルに SUFFIXES マクロを追加してください。

```
SUFFIXES: .cpp .cpp~
```

(この行は、メークファイル内のどこにでも入れることができます。)

次の内容をメークファイルに追加します。インデントされている行は、必ずタブでインデントしてください。

```
.cpp:
    $(LINK.cc) -o $@ $< $(LDLIBS)
.cpp~:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(LINK.cc) -o $@ $*.cpp $(LDLIBS)
.cpp.o:
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp~.o:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp.a:
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
.cpp~.a:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
```

標準ライブラリヘッダーファイルに対する make の使用

標準のライブラリファイルは istream、fstream のような名前で、.h 接尾辞は付いていません。また、テンプレートのソースファイルは、istream.cc、fstream.cc といった名前になります。

このため、Solaris 2.6 または Solaris 7 オペレーティング環境では、<istream> などの標準のライブラリヘッダーがプログラムにインクルードされていて、メークファイルに .KEEP_STATE がある場合は問題になります。たとえば、<istream> がインクルードされている場合、make ユーティリティは istream が実行可能ファイルであるとみなし、istream.cc から istream を構築するときにデフォルトの規則を使用します。このため、非常に誤解を生みやすいエラーメッセージが返されます (istream と istream.cc は両方とも C++ インクルードファイルのディレクトリにインストールされます)。1 つの解決策としては、make ユーティリティを使用せずに、dmake をシリアルモードで使用します (つまり、dmake -m serial を実行)。また、当面の回避策としては、make に -r オプションを指定します。-r オプションはデフォルトの make 規則を無効にします。しかし、この解決策は構築プロセスまで破壊する可能性があります。第 3 の解決策は、.KEEP_STATE ターゲットを使用しないことです。

第3章

C++ コンパイラオプション

この章では、Solaris 2.6、Solaris 7、および Solaris 8 で動作する CC コンパイラのコマンド行オプションについて詳しく説明します。ここで説明する機能は、特に断りがない限りすべてのプラットフォームに適用されます。特定のプラットフォームだけに有効な機能は SPARC または IA と明示します。詳細は、「はじめに」の「マルチプラットフォーム対応」を参照してください。

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　オプションの構文形式の例

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

* `option` はオプション名、*value* は値、*pathname* はパス名、*filename* はファイル名を示します。

各オプションの説明での表記規則は xx ページの「書体と記号について」を参照してください。

大括弧、括弧、中括弧、パイプ記号、省略記号は、オプションの説明に使用しているもので、オプションの一部ではありません。

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後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 (他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)。これはリンカーオプションには当てはまりません。
- `-I`、`-L`、`-pti`、`-R` オプションで指定した内容は累積され、上書きはされません。

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。

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。

コード生成オプション

コード生成オプションの要約をアルファベット順に示します。

表 3-2　コード生成オプション

処理	オプション
コンパイラの主要リリースとの互換モードを設定する。	`-compat`
C++ インライン関数を展開しない。	`+d`
仮想テーブル生成を制御する。	`+e(0|1)`
デバッグ用にコンパイルする。	`-g`
位置に依存しないコードを生成する。	`-KPIC`
位置に依存しないコードを生成する。	`-Kpic`
マルチスレッドコード用のコンパイルとリンクを行う。	`-mt`
コードのアドレス空間を指定する。	`-xcode=`*a*
データセグメントとテキストセグメントをマージする。	`-xMerge`
リンカーオプション	`-z` *arg*

デバッグオプション

デバッグオプションの要約をアルファベット順に示します。

表 3-3　デバッグオプション

処理	オプション
C++ インライン関数を展開しない。	`+d`
ドライバがコンパイラに渡すオプションを表示するが、コンパイルはしない。	`-dryrun`
C++ ソースファイルにプリプロセッサを実行し、結果を `stdout` に出力するが、コンパイルはしない。	`-E`
デバッグ用にコンパルする。	`-g`
デバッグ用にコンパイルするが、インライン機能は無効にしない。	`-g0`
インクルードされたファイルのパス名を出力する。	`-H`
コンパイルで作成される一時ファイルを保存する。	`-keeptmp`
以前のリリースからの移行に関する情報の参照先を表示する。	`-migration`
ソースの前処理だけを行う。`.i` ファイルに出力する。	`-P`
オプションをコンパイル中の各処理に直接渡す。	`-Qoption`
README ファイルの内容を表示する。	`-readme`
実行可能ファイルからシンボルテーブルを取り除く。	`-s`
一時ファイルのディレクトリを指定する。	`-temp=`*dir*
コンパイラのメッセージの詳細度を制御する。	`-verbose=`*vlst*
コンパイラオプションの要約を一覧表示する。	`-xhelp=flags`
インクリメンタルリンカーを無効にする。	`-xildoff`
インクリメンタルリンカーを有効にする。	`-xildon`
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する。	`-xs`
WorkShop ソースコードブラウザ用のテーブル情報を作成する。	`-xsb`
ソースブラウザ情報を作成するだけでコンパイルはしない。	`-xsbfast`

浮動小数点オプション

浮動小数点オプションの要約をアルファベット順に示します。

表 3-4　浮動小数点オプション

処理	オプション
SPARC 非標準浮動小数点モードを有効または無効にする。	`-fns[=(no|yes)]`
(IA) 浮動小数点精度モードを設定する。	`-f`precision=*p*
起動時に IEEE 丸めモードを有効にする。	`-fround=`*r*
浮動小数点最適化の設定を行う。	`-fsimple=`*n*
(IA) 浮動小数点式の精度を強制的に使用する。	`-fstore`
起動時に IEEE トラップモードを有効にする。	`-ftrap=`*tlst*
(IA) 浮動小数点式の精度を強制しない。	`-nofstore`
例外時に `libm` が数学ルーチンに対し IEEE 754 の値を返す。	`-xlibmieee`

言語オプション

言語オプションの要約をアルファベット順に示します。

表 3-5　言語オプション

処理	オプション
コンパイラの主要リリースとの互換モードを設定する。	`-compat`
さまざまな C++ 言語機能を有効または無効にする。	`-features=`*alst*

ライブラリオプション

ライブラリリンクオプションの要約をアルファベット順に示します。

表 3-6　ライブラリオプション

処理	オプション
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する。	`-B`*binding*
実行可能ファイル全体に対し動的ライブラリを使用できるかどうか指定する。	`-d(y|n)`
実行可能ファイルではなく動的共有ライブラリを構築する。	`-G`
生成される動的共有ライブラリに名前を割り当てる。	`-h`*name*
`ld(1)` がどのような `LD_LIBRARY_PATH` 設定も無視する。	`-i`
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する。	`-L`*dir*
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` を追加する。	`-l`*lib*
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む。	`-library=`*llst*
マルチスレッドコード用のコンパイルとリンクを行う。	`-mt`
ライブラリのパスを実行可能ファイルに組み込まない。	`-norunpath`
共有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。	`-R`*plst*
静的にリンクする C++ ライブラリを指定する。	`-staticlib=`*llst*
アーカイブライブラリを作成する。	`-xar`
例外時に `libm` が数学ルーチンに対し IEEE 754 の値を返す。	`-xlibmieee`
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する。	`-xlibmil`
最適化された数学ルーチンを使用する。	`-xlibmopt`
(SPARC) Sun Performance Library™ とリンクする。	`-xlic_lib=sunperflib`

表 3-6　ライブラリオプション (続き)

処理	オプション
デフォルトのシステムライブラリとのリンクを無効にする。	`-xnolib`
コマンド行の `-xlibmil` を取り消す。	`-xnolibmil`
数学ルーチンライブラリを使用しない。	`-xnolibmopt`
書き込み不可で割り当て可能なセクションに対する再配置が残っている場合には致命的エラーとする。	`-ztext`

ライセンスオプション

ライセンスオプションの要約をアルファベット順に示します。

表 3-7　ライセンスオプション

処理	オプション
ライセンスの待ち行列化を無効にする。	`-noqueue`
(SPARC) Sun Performance Library™ とリンクする。	`-xlic_lib=sunperf`
ライセンスサーバー情報を表示する。	`-xlicinfo`

廃止オプション

次のオプションはすでに廃止されているか、将来廃止されます。

表 3-8　廃止オプション

処理	オプション
コンパイラは無視する。将来のリリースのコンパイラがこのオプションを別の意味で使用する可能性もある。	`-ptr`
将来のリリースで削除される。	`-vdelx`

出力オプション

次に、出力オプションについてアルファベット順に要約します。

表 3-9　出力オプション

処理	オプション
コンパイルのみ。オブジェクト (`.o`) ファイルを作成するが、リンクはしない。	`-c`
ドライバがコンパイラに渡すオプションを表示するが、コンパイルはしない。	`-dryrun`
C++ ソースファイルにプリプロセッサを実行し、結果を `stdout` に出力するが、コンパイルはしない。	`-E`
実行可能ファイルではなく動的共有ライブラリを構築する。	`-G`
インクルードされたファイルのパス名を出力する。	`-H`
以前のリソースからの移行に関する情報の参照先を表示する。	`-migration`
出力ファイルや実行可能ファイルの名前を *filename* にする。	`-o` *filename*
ソースの前処理だけを行い、`.i` ファイルに出力する。	`-P`
`CC` ドライバが、型が *sourcetype* の出力を作成する。	`-Qproduce` *sourcetype*
実行可能ファイルからシンボルテーブルを取り除く。	`-s`
コンパイラメッセージの詳細度を制御する。	`-verbose=`*vlst*
必要に応じて追加の警告を出力する。	`+w`
警告メッセージを抑止する。	`-w`
コンパイラオプションの要約を一覧表示する。	`-xhelp=flags`
README ファイルの内容を表示する。	`-xhelp=readme`
メークファイルの依存情報を出力する。	`-xM`
依存情報を生成するが、`/usr/include` は除く。	`-xM1`
WorkShop ソースコードブラウザ用のテーブル情報を作成する。	`-xsb`
ソースブラウザ情報を作成するだけでコンパイルはしない。	`-xsbfast`

表 3-9　出力オプション (続き)

処理	オプション
コンパイル処理ごとの実行時間を報告する。	`-xtime`
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に変換する。	`-xwe`
リンカーオプション	`-z` *arg*

パフォーマンスオプション

パフォーマンスオプションの要約をアルファベット順に示します。

表 3-10　パフォーマンスオプション

処理	オプション
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する。	`-fast`
実行可能ファイルからシンボルテーブルを取り除く。	`-s`
ターゲットのアーキテクチャ命令セットを指定する。	`-xarch=`*isa*
(SPARC) オプティマイザのターゲットキャッシュ属性を定義する。	`-xcache=`*c*
一般的な SPARC アーキテクチャ用のコンパイルを行う。	`-xcg89`
SPARC V8 アーキテクチャ用のコンパイルを行う。	`-xcg92`
ターゲットのプロセッサチップを指定する。	`-xchip=`*c*
リンカーによる関数の順序変更を有効にする。	`-xF`
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する。	`-xlibmil`
(SPARC) 最適化された数学ルーチンライブラリを使用する。	`-xlibmopt`
コマンド行の `-xlibmil` を取り消す。	`-xnolibmil`
数学ルーチンライブラリを使用しない。	`-xnolibmopt`
最適化レベルを *level* にする。	`-xO`*level*
(SPARC) 一時レジスタの使用を制御する。	`-xregs=`*r*
(SPARC) メモリーに関するトラップを起こさないものとする。	`-xsafe=mem`

表 3-10 パフォーマンスオプション (続き)

処理	オプション
(SPARC) コードサイズを増やす最適化は行わない。	`-xspace`
ターゲットの命令セットと最適化のシステムを指定する。	`-xtarget=`*t*
可能であればループの最適化を行う。	`-xunroll=`*n*

プリプロセッサオプション

プリプロセッサオプションの要約をアルファベット順に示します。

表 3-11 プリプロセッサオプション

処理	オプション
シンボル *name* をプリプロセッサに定義する。	`-D`*name* [`=`*def*]
C++ ソースファイルにプリプロセッサを実行し、結果を `stdout` に出力するが、コンパイルはしない。	`-E`
ソースの前処理だけを行い、`.i` ファイルに出力する。	`-P`
プリプロセッサシンボル *name* の初期定義を削除する。	`-U`*name*
メークファイルの依存情報を出力する。	`-xM`
依存情報を生成するが、`/usr/include` は除く。	`-xM1`

プロファイルオプション

プロファイルオプションの要約についてアルファベット順に示します。

表 3-12 プロファイルオプション

処理	オプション
`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する。	`-p`
プロファイル処理のためのコードを生成する。	`-xa`
`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する。	`-xpg`
実行時プロファイルデータを収集したり、このデータを使って最適化する。	`-xprofile=tcov`

リファレンスオプション

次のオプションはコンパイラ情報を簡単に参照するためのものです。

表 3-13 リファレンスオプション

処理	オプション
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する。	`-migration`
コンパイラオプションの要約を一覧表示する。	`-xhelp=flags`
README ファイルの内容を表示する。	`-xhelp=readme`

ソースオプション

ソースオプションの要約をアルファベット順に示します。

表 3-14 ソースオプション

処理	オプション
`#include` ファイルの検索パスに *pathname* を追加する。	`-I`*pathname*
メークファイル依存情報を出力する。	`-xM`
依存情報を生成するが、`/usr/include` は除く。	`-xM1`

テンプレートオプション

テンプレートオプションの要約をアルファベット順に示します。

表 3-15 テンプレートオプション

処理	オプション
テンプレートインスタンスの位置とリンケージを制御する。	`-instances=`*a*
テンプレートソースの検索ディレクトリを追加指定する。	`-pti`*path*
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する。	`-template=`*w*

スレッドオプション

スレッドオプションの要約をアルファベット順に示します。

表 3-16 スレッドオプション

処理	オプション
マルチスレッドコード用のコンパイルとリンクを行う。	`-mt`
(SPARC) メモリーに関するトラップを起こさないものとする。	`-xsafe=mem`

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。オプションが他のオプションで置き換えられたり、他のオプションと同じである場合、詳細については他のオプション説明を参照してください。

表 3-17 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後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。
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。 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。 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=4` 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。

表 3-17 オプションの見出し (続き)

見出し	内容
相互の関連性	他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。たとえば「`-xO` が 3 より小さい場合は、`-xinline` オプションを使用すべきではありません」などです。
警告	オプションの使用について注意がある場合はここに示します。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他のオプションや文書を示します。
置き換え、同じなどの言葉	そのオプションが廃止され、他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。 一般的な意味と目的が同じであるオプションが 2 つある場合は、望ましいオプションを示します。たとえば、「`-xO` と同じです」は、`-xO` が望ましいオプションであることを示します。

オプションの一覧

-386

(IA) `-xtarget=386` と同じです。このオプションは、下位互換のためだけに用意されています。

-486

(IA) `-xtarget=486` と同じです。このオプションは、下位互換のためだけに用意されています。

-a

-xa と同じです。

-B*binding*

ライブラリのリンク形式を、シンボリックか、動的 (共有ライブラリ) にするか、静的 (共有でないライブラリ) のいずれかからを指定します。

-B オプションを使用すれば、同じコマンド行で指定を何回も切り替えることができます。このオプションはリンカー (ld) に渡されます。

注 – Solaris 7 および Solaris 8 プラットフォームでは、必ずしもすべてのライブラリが静的ライブラリとして使用できるわけではありません。

値

binding には次のいずれかの値を指定します。

binding の値	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	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なりません。)

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 が -Bdynamic でなければなりません。

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。他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

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。

共有ライブラリを互換モード (-compat[=4]) で作成する場合、そのライブラリに例外が含まれる場合は、-Bsymbolic を使用しないでください。必要な例外の捕獲に失敗してしまう可能性があります。

関連項目

-nolib、staticlib、ld(1)、153 ページの「標準ライブラリの静的リンク」、『リンカーとライブラリ』

-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。

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*

-cg[89|92]

-xcg[89|92] と同じです。

-compat [=(4|5)]

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 と __cplusplus 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。1 つは互換モードで、4.2 コンパイラで定義された ARM の意味解釈と言語が有効です。もう1 つは標準モードです。このモードでは、構文は ANSI/ISO 標準に従っていなければなりません。これらのモードには互換性はありません。ANSI/ISO 標準では、名前の符号化、vtable の配置、その他の ABI の細かい点で互換性のない変更がかなり必要であるためです。これらのモードは、次に示す -compat オプションで指定します。

値

-compat オプションには次の値を指定できます。

値	意味
-compat=4	(互換モード) 言語とバイナリの互換性を 4.0.1、4.1、4.2 コンパイラに合わせます。__cplusplus プリプロセッサマクロを 1 に、__SUNPRO_CC_COMPAT プリプロセッサマクロを 4 にそれぞれ設定します。
-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。__cplusplus プリプロセッサマクロを 1997IIL に、__SUNPRO_CC_COMPAT プリプロセッサマクロを 5 にそれぞれ設定します。

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。

-compat だけを指定すると、-compat=4 が使用されます。

__SUNPRO_CC は、-compat の設定に関係なく 0x510 に設定されます。

相互の関連性

-compat [=4} を使用する場合には、-xarch=v9、-xarch=v9a、-xarch=v9b はサポートされません。

関連項目

『C++ 移行ガイド』

+d

C++ インライン関数を展開しません。

相互の関連性

デバッグオプション -g を指定すると、このオプションが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。

関連項目

-g0、-g

–D*name*[=*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。

値

次の表は、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。

表 3-18 SPARC と IA 用の事前定義シンボル

名前	注
__ARRAYNEW	「配列」形式の operator new と operator delete を有効にしてコンパイルした場合に使用される。詳細は -features=[no%]arraynew を参照
__BOOL	ブール型を有効にした場合に使用される。詳細は -features=[no%]bool を参照
__BUILTIN_VA_ARG_INCR	varargs.h、stdarg.h、sys/varags.h のキーワードが __builtin_alloca、__builtin_va_alist、__builtin_va_arg_incr の場合に使用される。
__cplusplus	
__DATE__	
__FILE__	
__LINE__	
__STDC__	
__sun	
sun	「相互の関連性」を参照。

表 3-18 SPARC と IA 用の事前定義シンボル (続き)

名前	注
`__SUNPRO_CC=0x510`	`__SUNPRO_CC` の値はコンパイラのリリース番号を表す。
`__SUNPRO_CC_COMPAT=(4|5)`	39 ページの「`-compat [=(4|5)]`」を参照。
`__SVR4`	
`__TIME__`	
`__'uname –s'_'uname –r'`	*uname -s* は `uname -s` の出力で、*uname -r* は `uname -r` の出力。無効な文字 (ピリオドなど) は下線で置き換えられる（例: `-D__SunOS_5_7`、`-D__SunOS_5_8`)。
`__unix`	
`unix`	「相互の関連性」を参照。

表 3-19 UNIX 用の事前定義シンボル

名前	注
`_WCHAR_T`	

表 3-20 SPARC 用の事前定義シンボル

名前	注
`__sparc`	32 ビットコンパイルモードのみ
`sparc`	「相互の関連性」を参照。

表 3-21 SPARC v9 用の事前定義シンボル

名前	注
`__sparcv9`	64 ビットコンパイルモードのみ

表 3-22 IA 用の事前定義シンボル

名前	注
`__i386`	
`i386`	「相互の関連性」を参照。

デフォルト

=*def* を使用しないと、*name* は 1 になります。

相互の関連性

`+p` が使用されている場合は、`sun`、`unix`、`sparc`、`i386` は定義されません。

関連項目

`-U`

-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。

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。

関連項目

ld(1)、『リンカーとライブラリ』

-dalign

(SPARC) 可能な場合には、ダブルワードのロードとストア命令を生成してパフォーマンス向上を図ります。

このオプションは、double 型のデータがすべて double の境界から始まることを前提としています。

警告

あるプログラム単位を -dalign でコンパイルした場合は、プログラムのすべての単位を -dalign でコンパイルしなければなりません。そうしないと予期しない結果が生じることがあります。

-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。

-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。

例

このオプションは、プル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、次のようなプログラム foo.cc があるとします。

コード例 3-1 foo.cc

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
```

このプログラム結果は次のようになります。

コード例 3-2 -E オプションを使用したときの foo.cc の出力

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power ( int , int ) ;


int main ( ) {
int x ;
x = power ( 2 , 10 ) ;
}
```

警告

テンプレートを使用する場合は、このオプションの結果を C++ コンパイラの入力に使用することはできません。

関連項目

-P

+e(0|1)

互換モード (-compat[=4]) のときに仮想テーブルの生成を制御します。標準モード (デフォルトモード) のときには無効な指定として無視されます。

値

+e オプションには次の値を指定できます。

値	意味
0	仮想テーブルを生成せず、必要とされているテーブルへの外部参照を生成します。
1	仮想関数を使用して定義したすべてのクラスごとに仮想テーブルを生成します。

相互の関連性

このオプションを使用してコンパイルする場合は、-features=no%except オプションも使用してください。使用しなかった場合は、例外処理で使用される内部型の仮想テーブルがコンパイラによって生成されます。

関連項目

『C++ 移行ガイド』

-fast

コンパイルオプションの最適な組み合わせを選択し、実行速度を最適化します。

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。

拡張

このオプションは、次のコンパイラオプションを組み合せて、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 3-23 `-fast` 展開

オプション	SPARC	IA
`-dalign`	○	—
`-fns`	○	—
`-fsimple=2`	○	—
`-ftrap=%none`	○	○
`-nofstore`	—	○
`-xlibmil`	○	○
`-xlibmopt`	○	○
`-xO5`	○	○
`-xtarget=native`	○	○

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、指定された他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のものに戻されます。

誤

```
example% CC -xarch=v9 -fast test.cc
```

正

```
example% CC -fast -xarch=v9 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、インラインテンプレートファイルの使用よりも、その後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。

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。

```
examle% CC -xO3 -fast
```

警告

コンパイラで -fast オプションを指定すると、そのコードの移植性は失われます。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を使用しているプログラムには、-fast を指定しないでください。計算結果が違ったり、プログラムが途中で終了する、あるいは予期しない SIGFPE シグナルが発生する可能性があります。

以前のリリースの SPARC では、-fast マクロは -fsimple=1 に展開されました。現在では、-fsimple=2 に展開されます。

以前のリリースでは、-fast マクロは -xO4 に展開されました。現在では、-xO5 に展開されます。

注 – 以前の SPARC リリースでは `-fast` マクロに `-fnonstd` が含まれていましたが、このリリースでは含まれていません。`-fast` では、非標準浮動小数点モードは初期化されません。『数値計算ガイド』と `ieee_sun`(3M) のマニュアルページを参照してください。

関連項目

`-dalign`、`-fns`、`-fsimple`、`-ftrap=%none`、`-libmil`、`-nofstore`、`-xO5`、`-xlibmopt`、`-xtarget=native`

-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。

値

互換モード (`-compat[=4]`) と標準モード (デフォルトのモード) の両方で、*a* に次の値の 1 つを指定できます。

表 3-24 互換モードと標準モードでの `-feature` オプション

a の値	意味
`%all`	指定されているモードに対して有効なすべての `-feature` オプションを有効にします。
`[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 [しません]。
`[no%]anachronisms`	廃止されている構文を許可します [しません]。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。
`[no%]bool`	ブール型とリテラルを許可します [しません]。有効にした場合、マクロ `_BOOL=1` が定義されます。無効にした場合、マクロは定義されません。
`[no%]conststrings`	リテラル文字列を読み取り専用メモリーに入れます [入れません]。

表 3-24 互換モードと標準モードでの `-feature` オプション (続き)

a の値	意味
[no%]except	C++ 例外を許可します [しません]。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
[no%]export	キーワード `export` を認識します [しません]。
[no%]iddollar	識別子の最初以外の文字に `$` を許可します [しません]。
[no%]localfor	`for` 文に対して新しい局所スコープ規則を使用します [しません]。
[no%]mutable	キーワード `mutable` を認識します [しません]。
%none	指定されているモードに対して無効にできるすべての機能を無効にします。

標準モード (デフォルトのモード) では、*a* にはさらに次の値の 1 つを指定できます。

表 3-25 標準モードだけに使用できる `-features` オプション

[no%]strictdestrorder	静的記憶領域にある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 [従いません]

互換モード (`-compat[=4]`) では、*a* にはさらに次の値の 1 つを指定できます。

表 3-26 互換モードだけに使用できる `-features` オプション

a の値	意味
[no%]arraynew	`operator new` と `operator delete` の配列形式を認識します [しません] (たとえば、`operator new [] (void*)`)。これを有効にすると、マクロ `__ARRAYNEW=1` が定義されます。有効にしないと、マクロは定義されません。
[no%]explicit	キーワード `explicit` を認識します [しません]。

表 3-26 互換モードだけに使用できる `-features` オプション (続き)

a の値	意味
`[no%]namespace`	キーワード `namespace` と `using` を許可します [しません]。
`[no%]rtti`	実行時の型識別 (RTTI) を許可します [しません]。`dynamic c_cast<>` および `typeid` 演算子を使用する場合は、RTTI を有効にする必要があります。詳細は、『C++ プログラミングガイド』の「実行時の型識別」を参照してください。

注 – `[no%]castop` は、C++ 4.2 コンパイラ用に作成されたメークファイルとの互換性を維持するために使用できますが、C++ 5.0 および Sun Workshop 6 C++ コンパイラには影響はありません。新しい書式の型変換 (`const_cast`、`dynamic_cast`、`reinterpret_cast`、`static_cast`) は常に認識され、無効にすることはできません。

デフォルト

`-features` を指定しないと、以下が使用されます。

- 互換モード (`-compat[=4]`)

```
-features=%none,anachronisms,except
```

- 標準モード (デフォルトモード)

```
-features=%all,no%iddollar
```

関連項目

『C++ 移行ガイド』

-flags

-xhelp=flags と同じです。

-fnonstd

(IA) 浮動小数点ハードウェアの非標準的な初期設定を行います。

このオプションを指定すると、次の操作も可能になります。

- 浮動小数点演算オーバーフローのハードウェアによるトラップ
- ゼロによる除算
- 無効演算の例外

これらの結果は SIGFPE シグナルに変換されますが、プログラムに SIGFPE ハンドラがない場合は、メモリーダンプを行なってプログラムを終了します。ただし、コアダンプのサイズがゼロに設定されている場合を除きます。

デフォルト

-fnonstd を指定しないと、IEEE 754 浮動小数点演算例外が起きても、プログラムは異常終了しません。アンダーフローは段階的です。

関連項目

-fns、-ftrap=common、『数値計算ガイド』

-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。

-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 を含む他のマクロオプション (-fast など) の後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 (「例」を参照)。

一部の SPARC デバイスでは、非標準浮動小数点モードで「段階的アンダーフロー」が無効にされ、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 デバイスでは、`-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。

値

`-fns` オプションには次の値を指定できます。

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` とみなされます。

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、その後に続く `-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 でコンパイルした場合は、そのプログラムのすべてのルーチンを -fns オプションでコンパイルする必要があります。そうしないと、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、SPARC プラットフォームでメインプログラムをコンパイルするときしか有効ではありません。IA プラットフォームでは、このオプションは無視されます。

-fns=yes (または-fns のみ) を使用したときに、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すると、次のメッセージが返されることがあります。

関連項目

『数値計算ガイド』、ieee_sun(3M)

-fprecision=*p*

(IA) 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 オプションを指定すると、FPU (Floating Point Unit) 制御ワードの丸め精度モードのビットが設定されます。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。

値

p には次のいずれかを指定します。

p の値	意味
single	IEEE 単精度値に丸めます。
double	IEEE 倍精度値に丸めます。
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 フラグを指定しないと、丸め精度モードは extended になります。

警告

このオプションは、IA プラットホームでメインプログラムをコンパイルするときしか有効ではありません。SPARC プラットホームでは、このオプションは無視されます。

-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、次に示す IEEE 754 丸めモードを設定します。

- 定数式を評価する時にコンパイラが使用できる。
- プログラム初期化中の実行時に設定される。

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。

値

r には次のいずれかを指定します。

r の値	意味
nearest	最も近い数値に丸め、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* でコンパイルした場合は、そのプログラムのすべてのルーチンを同じ -fround=*r* オプションでコンパイルする必要があります。そうしないと、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。

-fsimple[=*n*]

浮動小数点最適化の設定を選択します。

このオプションで浮動小数点演算に影響する前提を設けることにより、オプティマイザで行う浮動小数点演算が簡略化されます。

値

n を指定する場合、0、1、2 のいずれかにしなければなりません。

n の値	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。

1	安全な簡略化を行います。その結果生成されたコードは、IEEE 754 に厳密には合致していませんが、大多数のプログラムの数値結果は変わりません。`-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。 • 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。 • 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。 • 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。 • 演算はゼロの符号を区別しない。 `-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	これは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、`-fsimple=0` が使用されます。

`-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`

-fstore

(IA) 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。

このオプションを無効にするには、-nofstore オプションを使用します。

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

-ftrap=*t*[,...*t*]

起動時に IEEE 754 トラップモードを有効に設定します。

このオプションは、プログラムの初期化時に設定される IEEE 754 トラップモードを設定しますが、SIGFPE ハンドラはインストールしません。トラップの設定と SIGFPE ハンドラのインストールを同時に行うには、ieee_handler 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。

値

t には次の値のいずれかを指定できます。

t の値	意味
[no%]division	ゼロによる除算をトラップします [しません]。
[no%]inexact	正確でない結果をトラップします [しません]。
[no%]invalid	無効な操作をトラップします [しません]。

t の値	意味
[no%]overflow	オーバーフローをトラップします [しません]。
[no%]underflow	アンダーフローをトラップします [しません]。
%all	上のすべてをトラップします。
%none	上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。これは、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。

IEEE トラップを有効にする場合は、-ftrap=common の設定をお勧めします。

デフォルト

-ftrap を指定しないと、-ftrap=%none が使用されます (トラップは自動的には有効にされません)。

例

1 つ以上の値を指定すると、それらは左から右に処理されます。したがって、-ftrap=%all,no%inexact と指定すると、inexact を除くすべてのトラップが設定されます。

相互の関連性

モードは、実行時に ieee_handler(3M) で変更できます。

警告

このオプションを使用してルーチンを 1 つコンパイルした場合は、プログラムのルーチンもすべて同じオプションを使用してコンパイルしてください。そうしないと、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には `-ftrap=common` を使用してください。

関連項目

`ieee_handler`(3M) のマニュアルページ

–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 `-Kpic` オプションでコンパイルされます。

テンプレートを使用する共有ライブラリを作成する場合は、通常、テンプレートデータベースでインスタンス化されているテンプレート関数を、共有ライブラリに組み込む必要があります。このオプションを使用すると、これらのテンプレートが必要に応じて共有ライブラリに自動的に追加されます。

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 ld 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。

関連項目

-dy、-Kpic、-xcode=pic13、-xildoff、-ztext、ld(1) のマニュアルページ、『C++ ライブラリ・リファレンス』

-g

コンパイラとリンカーに、デバッガでデバッグ可能な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。
- 「支援関数」を生成する。デバッガはこれを呼び出して、デバッガの機能のいくつかを実現する。
- 関数のインライン生成を無効にする。
- 特定のレベルの最適化を無効にする。

相互の関連性

このオプションと -xO*level* 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、109 ページの「–xOlevel」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 -xO3 以下の場合、可能な限りのシンボリック情報とほぼ最高の最適化が得られます。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。

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、可能な限りのシンボリック情報と最高の最適化が得られます。

このオプションを指定すると、+d オプションが自動的に指定されます。

このオプションを指定すると、-xildon が指定されてデフォルトのリンカーがインクリメンタルリンカーのオプションになるため、コンパイル、編集、デバッグのサイクルを効率的に実行できます。

次の条件のどれかが真でない場合は、ld ではなく ild が起動されます。

- -G オプションを指定している

- -xildoff オプションを指定している
- コマンド行でソースファイルを指定している

関連項目

+d、-g0、-xildoff、-xildon、-xs、および 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)

-g0

デバッグ用のコンパイルとリンクを行いますが、インライン展開は行いません。

このオプションは、+d が有効化されないという点を除いて、-g と同じです。

関連項目

+d、-g、-xildon、『dbx コマンドによるデバッグ』

-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。

-help

-xhelp=flags と同じです。

-h*name*

生成する動的共有ライブラリに名前 *name* を割り当てます。これはローダー用のオプションで、ld に渡されます。通常、-h の後に指定する *name* (名前) は、-o の後に指定する名前と同じでなければなりません。-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。

コンパイル時のローダーは、指定された名前を作成中の動的共有ライブラリに割り当て、そのライブラリのイントリンシック名 (固有名) としてライブラリの中に記録します。
-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。

例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

-i

リンカー ld は LD_LIBRARY_PATH の設定を無視します。

-I*pathname*

#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、インクルードファイルの相対ファイル名 (スラッシュ以外の文字で始まるファイル名) リストに、*pathname* (パス名) を追加します。

プリプロセッサは、インクルードファイルを次の順序で検索します。

1. #include "foo.h" の形式のインクルードファイル (" " で囲まれたもの) については、そのソースファイルを含むディレクトリ内を検索する
2. <foo.h> の形式のインクルードファイル (< > で囲まれたもの) については、そのソースファイルを含むディレクトリは検索しない
3. -I オプションで指定された名前を持つディレクトリ (もしあれば)
4. コンパイラ付属の C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特別な目的のファイルが格納されているディレクトリ

5. /usr/include ディレクトリ

注 – 標準ヘッダーの処理は上記とは異なります。詳細は、158 ページの「標準ヘッダーの実装」を参照してください。

相互の関連性

-pti*path* (パス) を使用しない場合、コンパイラは -I*pathname* (パス) 内の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* を使用してください。

-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。

値

a には次のいずれかを指定します。

a の値	意味
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。
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)。
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、それらに対して静的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	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=extern` が使用されます。

関連項目

第 4 章「テンプレートのコンパイル」

-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ておきます。

このオプションを `-verbose=diags` と一緒に使用すると、デバックに便利です。

関連項目

`-v`、`-verbose`

-KPIC

(SPARC) `-xcode=pic32` と同じです。

(IA) `-Kpic` と同じです。

-Kpic

(SPARC) `-xcode=pic13` と同じです。

(IA) 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 PC 相対アドレス指定モードで生成されます。

-L*dir*

ライブラリを検索するディレクトリに、*dir* (ディレクトリ) 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 *dir* が先に検索されます。

-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。通常のライブラリは、名前が lib*lib*.a か lib*lib*.so の形式です (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。コマンド行には、ライブラリをいくつでも指定できます。指定したライブラリは、-L*dir* で指定された順に検索されます。

-l*lib* オプションはファイル名の後に指定してください。

相互の関連性

正しい順序でライブラリが検索されるようにするには、安全のため、必ずソースとオブジェクトの後に -l*x* を使用してください。

警告

libthread とリンクする場合は、ライブラリを正しい順序でリンクするために -lthread ではなく -mt を使用してください。

POSIX スレッドを使用する場合は、-mt オプションと -lpthread オプションを使ってリンクする必要があります。-mt オプションが必要な理由は、libCrun (標準モード) と libC (互換モード) がマルチスレッド対応アプリケーションに対して libthread を必要とするためです。

関連項目

-L*dir*、-mt、『C++ ライブラリ・リファレンス』、『Tools.h++ 7.0.7 クラスライブラリ・リファレンスマニュアル』

-libmieee

-xlibmieee と同じです。

-libmil

-xlibmil と同じです。

-library=*l*[,...*l*]

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。

値

互換モード (-compat[=4]) の場合、*l* には次のいずれかを指定します。

表 3-27 互換モードでの -library オプション

l の値	意味
[no%]rwtools7	Tools.h++ バージョン 7 を使用します [しません]。
[no%]rwtools7_dbg	デバッグ可能な Tools.h++ バージョン 7 を使用します [しません]。
[no%]complex	複素数の演算に libcomplex を使用します [しません]。
[no%]libC	C++ サポートライブラリ libC を使用します [しません]。
[no%]gc	ガベージコレクション libgc を使用します [しません]。
[no%]gc_dbg	デバッグ可能なガベージコレクション libgc を使用します [しません]。
%all	-library=%all は -library=%none,rwtools7,complex,gc,libC を指定することと同じです
%none	C++ ライブラリを一切使用しません。

標準モード (デフォルトモード) の場合、*l* には次のいずれかを指定します。

表 3-28 標準モードでの `-library` オプション

l の値	意味
`[no%]rwtools7`	`Tools.h++` バージョン 7 を使用します [しません]。
`[no%]rwtools7_dbg`	デバッグ可能な `Tools.h++` バージョン 7を使用します [しません]。
`[no%]iostream`	古い `iostream` ライブラリ `libiostream` を使用します [しません]。
`[no%]Cstd`	C++ 標準ライブラリ `libCstd` を使用します [しません]。コンパイラ付属の `Cstd` ヘッダーファイルをインクルードします [しません]。
`[no%]Crun`	C++ 実行時ライブラリ `libCrun` を使用します [しません]。
`[no%]gc`	ガベージコレクション `libgc` を使用します [しません]。
`[no%]gc_dbg`	デバッグ可能なガベージコレクション `libgc` を使用します [しません]。
`%all`	`-library=%all` は `-library=%none,rwtools7,gc,iostream,Cstd,Crun` を指定することと同じです。
`%none`	`libCrun` の場合を除いてC++ ライブラリを使用しません。

デフォルト

- 互換モード (`-compat[=4]`)
  - `-library` を指定しないと、`-library=%none,libC` が使用されます。
  - `-library=%none` または`-library=no%libC` で特に除外されない限り、`libC` ライブラリは常に含まれます。
- 標準モード (デフォルトモード)
  - `-library` を指定しないと、`-library=%none,Cstd,Crun` が使用されます。
  - `-library=%none` または`-library=no%Cstd` で特に除外されない限り、`libCstd` ライブラリは常に含まれます。
  - `-library=no%Crun` で特に除外されない限り、`libCrun` ライブラリは常に含まれます。

例

標準モードで libCrun 以外の C++ ライブラリを除外してリンクするには、次のコマンドを使用します。

```
example% CC -library=%none
```

標準モードで RogueWave の Tools.h++ バージョン 7 ライブラリと iostream 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
```

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。

-library オプションを使用すると、指定したライブラリに対する -l オプションが正しい順序で送信されるようになります。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。

RogueWave の Tools.h++ ライブラリは同時に 1 つしか使用できません。

RogueWave の Tools.h++ バージョン 7 ライブラリは、従来の iostream を使って構築されています。このため、標準モードで Rouged Wave の Tools.h++ のライブラリを使用する場合は、libiostream も含める必要があります。詳細は、『C++ 移行ガイド』を参照してください。

libCstd と libiostream の両方を含めた場合は、プログラム内で新旧両方の形式の iostream (例: cout と std::cout) を使用して、同じファイルにアクセスしないよう注意してください。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。

libC とも libCrun ともリンクしないプログラムは、C++ のすべての機能を使用できないことがあります。

`-xnolib` を指定すると、`-library` は無視されます。

警告

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。

関連項目

`-I`、`-l`、`-R`、`-Y P`、`-staticlib=`*l*、`-xnolib`、
23 ページの「標準ライブラリヘッダーファイルに対する make の使用」
『C++ ライブラリ・リファレンス』、『Tools.h++7.0 ユーザーズガイド』、
『Tools.h++ 7.0 クラスライブラリ・リファレンスマニュアル』、
『Standard C++ Class Library Reference』(英語版のみ)

`-library=no%cstd` オプションを使用して、ユーザー独自の C++ 標準ライブラリの使用を有効にする方法については、156 ページの「C++ 標準ライブラリの置き換え」を参照してください。

-migration

以前のバージョンのコンパイラ用に作成されたソースコードの移行に関する情報の参照先を表示します。

-misalign

(SPARC) 通常はエラーになる、メモリー中の境界整列の誤ったデータを許可します。以下に例を示します。

```
char b[100];
int f(int * ar) {
return *(int *) (b +2) + *ar;
}
```

このオプションは、プログラムの中に正しく境界整列されていないデータがあることをコンパイラに知らせます。したがって、境界整列が正しくない可能性があるデータに対しては、ロードやストアを非常に慎重に (つまり、1 度に 1 バイトずつ) 行う必要があります。このオプションを使用すると、実行速度が大幅に低下することがあります。低下する程度はアプリケーションによって異なります。

相互の関連性

SPARC プラットフォームで `#pragma pack` を使って、型のデフォルト境界整列よりも高い密度でデータをパックする場合は、アプリケーションのコンパイルとリンクに `-misalign` オプションを指定する必要があります。

境界整列が正しくないデータは、実行時に `ld` のトラップ機構によって処理されます。`-misalign` オプションとともに最適化フラグ (`-xO[1|2|3|4|5]` またはそれと同等のフラグ) を使用すると、ファイル境界整列の正しくないデータを正しい境界に整列に合わせるための命令がオブジェクトに挿入されます。この場合には、実行時不正境界整列トラップは生成されません。

警告

できれば、プログラムの境界整列が正しい部分と境界整列が誤った部分をリンクしないでください。

コンパイルとリンクを別々に行う場合は、`-misalign` 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で指定する必要があります。

-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。

このオプションでは、次のことが行われます。

- `-D_REENTRANT` をプリプロセッサに渡します。
- `-lthread` を正しい順序で ld に渡します。
- 標準モード (デフォルトモード) では、`libthread` が `libCrun` よりも前にリンクされるようにします。
- 互換モード (`-compat`) では、`libthread` が `libC` よりも前にリンクされるようにします。

アプリケーションやライブラリがマルチスレッド化されている場合は、-mt オプションが必要です。

警告

libthread とリンクする場合には、-lthread ではなく -mt オプションを使用してライブラリのリンク順序が正しくなるようにしてください。

POSIX スレッドを使用する場合は、-mt オプションと -lpthread オプションを使ってリンクする必要があります。-mt オプションが必要な理由は、libCrun (標準モード) と libC (互換モード) がマルチスレッド対応のアプリケーションに対して libthread を必要とするためです。

コンパイルとリンクを別々に実行する場合で、コンパイルで -mt を使用した場合は、次の例に示すようにリンクでも -mt を使用してください。そうしないと、予期しない結果が発生する可能性があります。

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

関連項目

-xnolib、『C++ プログラミング・ガイド』、『マルチスレッドのプログラミング』、『リンカーとライブラリ』、『C++ ライブラリ・リファレンス』

-native

-xtarget=native と同じです。

-noex

-features=no%except と同じです。

-nofstore

(IA) 浮動小数点式の精度を変換しない。

このオプションを指定すると、次のどちらの場合でも、コンパイラは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せん。つまり、レジスタの値はそのままです。

- 式や関数を変数に代入する
- 式や関数をそれより短い浮動小数点型にキャストする

関連項目

`-fstore`

-nolib

`-xnolib` と同じです。

-nolibmil

`-xnolibmil` と同じです。

-noqueue

ライセンスを待ち行列に入れません。

ライセンスを確保できない場合、コンパイラはコンパイル要求を待ち行列に入れず、コンパイルもしないで終了します。メークファイルのテストには、ゼロ以外の状態が返されます。

-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、標準以外の位置にインストールされたコンパイラで生成した実行可能ファイルを、プログラムのユーザーがその非標準の位置で使う必要がない場合に便利です。

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 (デフォルトのインストール先は `/opt/SUNWspro/lib`) 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なりません。そうすることにより、実行時リンカーはその共有ライブラリを見つけることができます。

-O

`-xO2` と同じです。

-O*level*

`-xO`*level* と同じです。

-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。

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レ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` 内のレポジトリにそれぞれ書き込みます。

```
example% CC -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レ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、次のコマンドでは、コンパイラは `./sub1/SunWS_Cache` と `./sub2/SunWS_cache` から読み取り、必要な場合は `./SunWS_cache` に書き込みます。

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

詳細は、133 ページの「テンプレートレポジトリ」を参照してください。

警告

このファイル名には、コンパイラが作成するファイルの型に合った接尾辞を指定してください。また、cc ドライバはソースファイルには上書きしないため、ソースファイルとは異なるファイルを指定する必要があります。

+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。

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

-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。

関連項目

-E

-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作成します。-p は実行内容を記録し、正常終了時に mon.out というファイルを生成します。

関連項目

-xpg、-xprofile、analyzer(1) のマニュアルページ、*Sun WorkShop* の『プログラムのパフォーマンス解析』

-pentium

(IA) -xtarget=pentium と置き換えられています。

-pg

-xpg と同じです。

-PIC

(SPARC) -xcode=pic32 と同じです。

(IA) -Kpic と同じです。

-pic

(SPARC) -xcode=pic13 と同じです。

(IA) -Kpic と同じです。

-pta

-template=wholeclass と同じです。

-pti*path*

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。

関連項目

`-I`*pathname*

-pto

`-instances=static` と同じです。

-ptr

このオプションは廃止されたため、コンパイル時には無視されます。

警告

`-ptr` オプションは存在しても無視されますが、すべてのコンパイルコマンドから削除するようにしてください。これは将来のリリースで、`-ptr` が以前とは異なる動作のオプションとして再実装される可能性があるためです。

関連項目

レポジトリのディレクトリについては、133 ページの「テンプレートレポジトリ」を参照してください。

-ptv

`-verbose=template` と同じです。

-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。

値

phase には、以下の値のいずれか 1 つを指定します。

SPARC	IA
ccfe	ccfe
iropt	cg386
cg	codegen
cclink	cclink
ld	ld

例

次に示すコマンド行では、ld が CC ドライバによって起動されたとき、-Qoption で指定されたオプションの -i と -m が ld に渡されます。

```
example% CC -Qoption ld -i,-m test.c
```

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、ccfe に -features=bool,castop を渡そうと次のように指示するとします。

```
-Qoption ccfe -features=bool,castop
```

しかしこの指定は、意図に反して次のように解釈されてしまいます。

```
-Qoption ccfe -features=bool -Qoption ccfe castop
```

正しい指定は次のとおりです。

```
-Qoption ccfe -features=bool,-features=castop
```

-qoption *phase option*

-Qoption と同じです。

-qp

-p と同じです。

-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。

sourcetype に指定する接尾辞の定義は次のとおりです。

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	コードジェネレータ cg が作成するオブジェクトファイル
.s	cg が作成するアセンブラソース

-qproduce *sourcetype*

-Qproduce と同じです。

-R*pathname* [:...*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。

複数のパス名も指定できます (例: -R/path1:/path2)。

このオプションは ld に渡されます。

デフォルト

-R オプションを指定しないと、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命令によって異なります (-xarc を指定しないと、-xarch=generic が使用されます)。

-xarch の値	デフォルトのライブラリ検索パス
v9、v9a、v9b	*install_dirctory*/SUNWspro/lib/v9
上記以外の値	*install_directory*/SUNWspro/lib

標準インストールでは、*install-directory* は /opt です。

相互の関連性

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。

関連項目

-norunpath、『リンカーとライブラリ』

-readme

-xhelp=readme と同じです。

-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。

-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。

-sb

`-xsb` で置き換えられています。

-sbfast

`-xsbfast` と同じです。

-staticlib=*l*[,...*l*]

`-library` オプションに指定されている C++ ライブラリのうち (デフォルトを含む) どれを静的にリンクするのかを指定します。

値

l には、以下の値のいずれか 1 つを指定します。

l の値	意味
`[no%]` *library*	*library* に指定できる値については 67 ページの「`-library=l[,...l]`」を参照してください。
`%all`	`-library` オプションに指定したすべてのライブラリが静的にリンクされます。
`%none`	静的にリンクするライブラリはありません。

デフォルト

`-staticlib` を指定しないと、`-staticlib=%none` が使用されます。

例

-library のデフォルト値は Crun であるため、次のコマンド行は、libCrun を静的にリンクします。

```
example% CC -staticlib=Crun ← 正しい
```

これに対し、次のコマンド行は libgc をリンクしません。これは、-library オプションで明示的に指定しない限り、libgc はリンクされないためです。

```
example% CC -staticlib=gc ← 誤り
```

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 ← 正しい
```

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 ← 誤り
```

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 ←正しい
```

相互の関連性

-staticlib オプションは、-library オプションで明示的に選択された C++ ライブラリ、または、デフォルトで暗黙的に選択された C++ ライブラリだけに機能します。互換モードでは (-compat=[4])、libC がデフォルトで選択されます。標準モードでは (デフォルトのモード)、Cstd と Crun がデフォルトで選択されます。

-xarch=v9、-xarch=v9a、-xarch=v9b のいずれかを使用する場合、静的ライブラリとしては使用できない C++ ライブラリがあります。

警告

ライブラリで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。

関連項目

-library、153 ページの「標準ライブラリの静的リンク」

-temp=*dir*

一時ファイルのディレクトリを定義します。

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を *dir* に指定します。

関連項目

-keeptmp

-template=*w*[,...*w*]

さまざまなテンプレートオプションを実行可能または実行不能にします。

値

w は次のいずれかでなければなりません。

w の値	意味
[no%]wholeclass	コンパイラに対し、使用されている関数だけインスタンス化するのではなく、テンプレートクラス全体をインスタンス化する [しない] ように指示します。クラスの少なくとも 1 つのメンバーを参照しなければなりません。そうでない場合は、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。
[no%]extdef	別のソースファイルからテンプレート定義を検索します [しません]。

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。

-time

`-xtime` と同じです。

-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。

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。他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。

`-U` オプションは、コマンド行に複数指定できます。

相互の関連性

すべての `-U` オプションは、(存在する場合は) すべての `-D` オプションの後で処理されます。

-unroll=*n*

`-xunroll=`*n* と同じです。

-V

`-verbose=version` と同じです。

-v

`-verbose=diags` と同じです。

-vdelx

互換モード (`-compat[=4]`) のみ

`delete[]` を使用する式に対し、実行時ライブラリ関数 `_vector_delete_` の呼び出しを生成する代わりに `_vector_deletex_` の呼び出しを生成します。関数 `_vector_delete_` は、削除するポインタおよび各配列要素のサイズという 2 つの引数をとります。

関数 `_vector_deletex_` は `_vector_delete_` と同じように動作しますが、3 つめの引数としてそのクラスのデストラクタのアドレスをとります。この引数はサン以外のベンダーが使用するためのもので、関数では使用しません。

デフォルト

コンパイラは、`delete[]` を使用する式に対して `_vector_delete_` の呼び出しを生成します。

警告

これは旧式フラグであり、将来のリリースでは削除されます。サン以外のベンダーからソフトウェアを購入し、ベンダーがこのフラグの使用を推奨していない限り、このオプションは使用しないでください。

-verbose=*v*[,...*v*]

コンパイラの冗長性を制御します。

値

v には、次に示す値の 1 つを指定します。

v の値	意味
[no%]diags	各コンパイル段階が渡すコマンド行を表示します [しません]。
[no%]template	テンプレートインスタンス化冗長モード (検証モードともいう) を起動します [しません]。冗長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 ドライバに対し、呼び出したプログラムの名前とバージョン番号を表示するよう指示します [しません]。
%all	上のすべてを呼び出します。
%none	-verbose=%none は -verbose=no%template,no%diags,no%version を指定することと同じです。

v の値は複数指定することができます。たとえば、-verbose=template,diags とすることができます。

デフォルト

-verbose を指定しないと、-verbose=%none が使用されます。

+w

意図しない結果になるおそれがあるコードを識別します。

次のような問題のありそうな構造について、追加の警告を生成します。

- 移植性がない
- 間違っていると考えられる
- 効率が悪い

デフォルト

このオプションを指定しないと、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。

関連項目

-w、 +w2

+w2

+w で発行される警告に加えて、技術的な違反についての警告を発行します。+w2 で行われる警告は、危険性はないが、プログラムの移植性を損なう可能性がある違反に対するものです。

警告

+w2 を指定してコンパイルすると、Solaris および C++標準ヘッダーファイルに関する警告が発行されることがあります。

関連項目

+w

-w

ほとんどの警告メッセージを抑止します。

コンパイラが出す警告を出力しません。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。

関連項目

+w

-xa

プロファイル用のコードを生成します。

コンパイル時に TCOVDIR 環境変数を設定すれば、カバレージ (.d) ファイルを置くディレクトリを指定できます。この変数を設定しなければ、カバレージ (.d) ファイルは ソースファイルと同じディレクトリにソースファイルとして残ります。

このオプションは、古いカバレージファイルとの下位互換を保つためだけに使用してください。

相互の関連性

-xprofile=tcov オプションと -xa オプションは、1 つの実行可能ファイルで同時に使用できます。つまり、-xprofile=tcov でコンパイルされたファイルと -xa でコンパイルされたファイルからなるプログラムをリンクすることはできますが、両方のオプションを使って 1 つのファイルをコンパイルすることはできません。

-xa オプションと -g を一緒に使用することはできません。

警告

コンパイルとリンクを別々に行う場合で、-xa でコンパイルした場合は、 リンクも -xa で行わなければなりません。そうしないと予期できない結果になることがあります。

関連項目

-xprofile=tcov、tcov(1) のマニュアルページ、
『プログラムのパフォーマンス解析』

-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++ のアーカイブをコンパイルするときには通常、テンプレートデータベース中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の中にあらかじめ入れておく必要があります。このオプションはそれらのテンプレートを必要に応じてアーカイブに自動的に追加します。

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

第 6 章「ライブラリの構築」

-xarch=*isa*

対象となる命令セットアーキテクチャ (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の命令だけに制限します。つまり、指定した命令セットだけを許可します。このオプションを使用すると、ターゲットアーキテクチャに固有の命令が使用できない可能性があります。

値

SPARC プラットフォームの場合

表 3-29 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 3-29 SPARC プラットフォームでの `-xarch` の値

isa の値	意味
`generic`	**ほとんど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。** これはデフォルトです。このオプションは、どのプロセッサでも大きくパフォーマンスを落とさず、またほとんどのプロセッサで良好なパフォーマンスを得られるような最良の命令セットを使用します。「最良な命令セット」の内容は、新しいリリースごとに調整される可能性があります。
`native`	**現在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。** これは `-fast` オプションのデフォルトです。現在コンパイラを実行しているシステムのプロセッサに最も適した設定を選択します。
`v7`	SPARC-V7 ISA **用にコンパイルします。** V7 ISA 上で良好なパフォーマンスを得るためのコードを生成します。これは、V8 ISA 上で最良なパフォーマンスを得るための最良の命令セットと同じですが、整数の `mul` と `div` 命令、および `fsmuld` 命令は含まれていません。 例: SPARCstation 1、SPARCstation 2
`v8a`	V8a **版の** SPARC-V8 ISA **用にコンパイルします。** 定義上、V8a は V8 ISA を意味します。ただし、`fsmuld` 命令は含まれていません。 V8a ISA 上で良好なパフォーマンスを得るためのコードを生成します。 例: microSPARC I チップアーキテクチャに基づくすべてのシステム
`v8`	SPARC-V8 ISA **用にコンパイルします。** V8 アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 例: SPARCstation 10

表 3-29 SPARC プラットフォームでの `-xarch` の値 (続き)

isa の値	意味
`v8plus`	**V8plus 版の SPARC-V9 ISA 用にコンパイルします。** 定義上、V8plus は V9 ISA を意味します。ただし、V8plus ISA 仕様で定義されている 32 ビットサブセットに限定されます。さらに、VIS (Visual Instruction Set) と実装に固有な ISA 拡張機能は含まれていません。 • V8plus ISA 上で良好なパフォーマンスを得るためのコードを生成します。 • 生成されるオブジェクトコードは SPARC-V8+ ELF32 形式であり、Solaris UltraSPARC 環境でのみ実行できます。つまり、V7 または V8 のプロセッサ上では実行できません。 例: UltraSPARC チップアーキテクチャに基づくすべてのシステム
`v8plusa`	**V8plusa 版の SPARC-V9 ISA 用にコンパイルします。** 定義上、V8plusa は V8plus アーキテクチャ + VIS (Visual Instruction Set) バージョン 1.0 + UltraSPARC 拡張機能を意味します。 • UltraSPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。ただし、V8plus 仕様で定義されている 32 ビットサブセットに限定されます。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式であり、Solaris UltraSPARC 環境でのみ実行できます。つまり、V7 または V8 のプロセッサ上では実行できません。 例: UltraSPARC チップアーキテクチャに基づくすべてのシステム
`v8plusb`	**UltraSPARC-III 拡張機能を持つ、V8plusb 版の SPARC-V8plus ISA 用にコンパイルします。** UltraSPARC アーキテクチャ + VIS (Visual Instruction Set) バージョン 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。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式です。このコードは Solaris UltraSPARC-III 環境でのみ実行できます。 • UltraSPARC-III アーキテクチャ上で良好なパフォーマンスを得るための最良のコードを使用します。

表 3-29 SPARC プラットフォームでの `-xarch` の値 (続き)

isa の値	意味
`v9`	SPARC-V9 ISA **用にコンパイルします。** V9 SPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 • 生成される `.o` オブジェクトファイルは ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • `-xarch=v9` は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。
`v9a`	UltraSPARC **拡張機能を持つ** SPARC-V9 ISA **用にコンパイルします。** SPARC-V9 ISA に VIS (Visual Instruction Set) と UltraSPARC プロセッサに固有の拡張機能を追加します。V9 SPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 • 生成される `.o` オブジェクトファイルは ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • `-xarch=v9a` は、64 ビット対応 Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。

表 3-29 SPARC プラットフォームでの `-xarch` の値 (続き)

isa の値	意味
`v9b`	UltraSPARC-III **拡張機能を持つ** SPARC-V9 ISA **用にコンパイルします。** V9a 版の SPARC-V9 ISA に UltraSPARC-III 拡張と VIS バージョン 2.0 を追加します。Solaris UltraSPARC-III 環境で良好なパフォーマンスを得るためのコードを生成します。 • 生成される `.o` オブジェクトファイルは SPARC-V9 ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC-III プロセッサ上でしか実行できません。 • `-xarch=v9b` は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。

また、次のことにも注意してください。

- SPARC 命令セットアーキテクチャ V7、V8 および V8a はバイナリ互換です。
- `v8plus` でコンパイルされたオブジェクトバイナリ (`.o`) ファイルと `v8plusa` でコンパイルされた `.o` ファイルは、SPARC `V8plusa` 互換のプラットフォーム上でのみリンクおよび同時に実行できます。
- `v8plus`、`v8plusa`、および `v8plusb` でそれぞれコンパイルされたオブジェクトバイナリ(`.o`)ファイル は、SPARC `V8plusb` 互換のプラットフォーム上でのみリンクおよび同時に実行できます。
- `-xarch` の値 `v9`、`v9a` および `v9b` は、UltraSPARC 64 ビット Solaris 環境でのみ、指定できます。
- `v9` と `v9a` でそれぞれコンパイルされたオブジェクトバイナリ (`.o`) ファイルは、SPARC `V9a` 互換プラットフォーム上でのみリンクおよび同時に実行できます。
- `v9`、`v9a`、および `v9b` でそれぞれコンパイルされたオブジェクトバイナリ (`.o`) ファイルは、SPARC `V9b` 互換プラットフォーム上でのみリンクおよび同時に実行できます。

いずれの場合でも、初期のアーキテクチャでは、生成された実行可能ファイルの実行速度がかなり遅くなる可能性があります。また、4 倍精度 (`REAL*16` と `long double`) の浮動小数点命令は多くの命令セットアーキテクチャで使用できますが、この命令はコンパイラが使用するコードには含まれません。

IA プラットフォームの場合

表 3-30 に、IA プラットフォームでの -xarch キーワードの詳細を示します。

表 3-30 IA プラットフォームでの -xarch 値

isa の値	意味
generic	ほとんど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。これはデフォルトです。このオプションは、どのプロセッサでも大きくパフォーマンスを落とさず、またほとんどのプロセッサで良好なパフォーマンスを得られるような最良の命令セットを使用します。「最良な命令セット」の内容は、新しいリリースごとに調整される可能性があります。
386	このリリースでは、generic と 386 は同じです。
486	Intel PentiumPro チップ用にコンパイルします。
pentium	このリリースでは、486 と pentium は同じです。
pentium_pro	このリリースでは、486 と pentium_pro は同じです。

デフォルト

-xarch=*isa* を指定しないと、-xarc=generic が使用されます。

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 オプションによって設定された -xarch の値を変更するためにも使用できます。たとえば、-xtarget=ultra2 は -xarch=v8 -xchip=ultra2 -xcache=15/32/1:512/64/1 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=v8 より優先されます。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

compat[=4] とともに -xarch=v9、-xarch=v9a、-xarch=v9b のいずれかを使用することはできません。

警告

このオプションを最適化の指定と一緒に使用する場合、適切な選択をすれば、指定したアーキテクチャで実行可能ファイルの良好なパフォーマンスが得られます。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。

-xcache=*c*

(SPARC) オプティマイザで使用するキャッシュ属性を定義します。

オプティマイザが使用できるキャッシュの属性を定義します。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` オプションが展開されたものの一部です。このオプションの主な目的は、`-xtarget` オプションにより指定される値を変更することです。

値

c には次の値のいずれかを指定します。

c の値	意味
`generic`	ほとんどの SPARC プロセッサで良好なパフォーマンスが得られるキャッシュ属性を定義します。
s1/l1/a1	レベル 1 のキャッシュ属性を定義します。
s1/l1/a1:s2/l2/a2	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:s2/l2/a2:s3/l3/a3	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 *si*/*li*/*ai* の定義は次のとおりです。

属性	定義
si	レベル *i* のデータキャッシュのサイズ (K バイト)
li	レベル *i* のデータキャッシュのラインサイズ (バイト)
ai	レベル *i* のデータキャッシュの結合規則

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 を指定しないと、-xcache=generic がデフォルトで使用されます。この値を指定すると、ほとんどの SPARC プロセッサで良好なパフォーマンスが得られ、どのプロセッサでも顕著なパフォーマンスの低下がないキャッシュ属性がコンパイラで使用されます。

例

-xcache=16/32/4:1024/32/1 の設定内容は、次のとおりです。

レベル 1 のキャッシュ	レベル 2 のキャッシュ
16K バイト ラインサイズ 32 バイト 4 ウェイアソシアティブ	1024K バイト ラインサイズ 32 バイト ダイレクトマッピング

関連項目

-xtarget=*t*

-xcg89

-xtarget=ss2 と同じです。

警告

コンパイルとリンクを別々に実行する場合で、コンパイルで `-xcg89` を使用した場合は、リンクでも同じオプションを使用してください。そうしないと、予期しない結果が発生する可能性があります。

-xcg92

`-xtarget=ss1000` と同じです。

警告

コンパイルとリンクを別々に実行する場合で、コンパイルで `-xcg92` を使用した場合は、リンクでも同じオプションを使用してください。そうしないと、予期しない結果が発生する可能性があります。

-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。

ターゲットとなるプロセッサを指定することによって、タイミング属性を指定します。

このオプションは次のものに影響を与えます。

- 命令の順番 (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` オプションが展開されたものの一部です。このオプションの主な目的は、`-xtarget` オプションにより指定される値を変更することです。

値

c には次の値のいずれかを指定します。

表 3-31 `-xchip` オプション

プラットフォーム	*c* の値	意味
SPARC	`generic`	SPARC プロセッサ上で良好なパフォーマンスを得るための、タイミング属性を使用します。
SPARC	`old`	SuperSPARC チップより以前のプロセッサのタイミング属性を使用します。
SPARC	`super`	SuperSPARC チップのタイミング属性を使用します。
SPARC	`super2`	SuperSPARC II チップのタイミング属性を使用します。
SPARC	`micro`	MicroSPARC チップのタイミング属性を使用します。
SPARC	`micro2`	MicroSPARC II チップのタイミング属性を使用します。
SPARC	`hyper`	HyperSPARC チップのタイミング属性を使用します。
SPARC	`hyper2`	HyperSPARC II チップのタイミング属性を使用します。
SPARC	`powerup`	Weitek PowerUp チップのタイミング属性を使用します。
SPARC	`ultra`	UltraSPARC I チップのタイミング属性を使用します。
SPARC	`ultra2`	UltraSPARC II チップのタイミング属性を使用します。
SPARC	`ultra2i`	UltraSPARC IIi チップのタイミング属性を使用します。
SPARC	`ultra3`	UltraSPARC III チップのタイミング属性を使用します。
IA	`generic`	一般的な IA プロセッサが持つタイミング属性を使用します。
IA	`386`	Intel 386 チップのタイミング属性を使用します。

表 3-31 `-xchip` オプション

プラットフォーム	*c* の値	意味
IA	`486`	Intel 486 チップのタイミング属性を使用します。
IA	`pentium`	Intel Pentium チップのタイミング属性を使用します。
IA	`pentium_pro`	Intel Pentium Pro チップのタイミング属性を使用します。

デフォルト

ほとんどの SPARC プロセッサでは、デフォルト値の `generic` を使用すれば、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がコンパイラで使用されます。

-xcode=*a*

(SPARC) コードのアドレス空間を指定します。

値

a には次の値のいずれかを指定します。

表 3-32 `-xcode` オプション

a の値	意味
`abs32`	32 ビット絶対アドレスを生成します。高速ですが範囲が限定されます。コード + データ + bss サイズは 2**32 バイトに限定されます。
`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 + bss サイズは 2**44 バイトに限定され、64 ビットアーキテクチャ `-xarch=(v9|v9a|v9b)` でのみ使用可能です。

表 3-32 -xcode オプション

a の値	意味
abs64	(SPARC) 64 ビット絶対アドレスを生成します。低速ですが全範囲を使用でき、64 ビットアーキテクチャ -xarch=(v9|v9a|v9b) でのみ使用可能です。
pic13	位置に依存しないコード (小規模モデル) を生成します。高速ですが範囲が限定されます。-Kpic と同等。32 ビットアーキテクチャでは最大 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	位置に依存しないコード (大規模モデル) を生成します。低速ですが全範囲を使用できます。-KPIC と同等。32 ビットアーキテクチャでは最大 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。

デフォルト

SPARC V8 と V7 の場合は -xcode=abs32 です。

SPARC と UltraSPARC (-xarc=(v9|v9a|v9b) のとき) の場合は -xcode=abs64 です。

-xcrossfile[=*n*]

(SPARC) 複数のソースファイルに渡る最適化とインライン化を可能にします。

値

n には次のどちらかの値を指定します。

n の値	意味
0	複数のソースファイルに渡る最適化とインライン化を実行しません。
1	複数のソースファイルに渡る最適化とインライン化を実行します。

通常、コンパイラの解析の範囲は、コマンド行で指定した個々のファイルごとに行われます。たとえば、-xO4 オプションを指定した場合、自動インライン化は同じソースファイル内で定義および参照されているサブプログラムにのみ行われます。

`-xcrossfile` または `-xcrossfile=1` を指定すると、コンパイラはコマンド行で指定されたすべてのファイルを一括して分析し、それらが単一のソースファイルであるかのように扱います。

デフォルト

`-xcrossfile` を指定しない場合、`-xcrossfile=0` が仮定され、複数のソースファイルに渡る最適化とインライン化は行われません。

`-xcrossfile` は `-xcrossfile=1` と同じです。

相互の関連性

`-xcrossfile` 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得られます。

警告

このオプションを使ってコンパイルされたファイルは、インライン化されたコードを含む可能性があるため、相互に依存しています。したがって、プログラムにリンクするときは、1 つの単位として使用しなければなりません。あるルーチンを変更したために、関連するファイルを再コンパイルした場合は、すべてのファイルを再コンパイルする必要があります。結果として、このオプションを使用すると、`makefile` の構成に影響を与えます。

-xF

この `-xF` オプションを指定してコンパイルした後で実行してアナライザを使用すると、最適化された関数の順序を示すマップファイルを作成できます。続いて実行するリンカーには、`-M`*mapfile* (マップファイル) オプションでそのマップを使用するよう指示して、実行可能ファイルを作成することができます。これによって、実行可能ファイルの各関数が別々のセクションに置かれます。

メモリー上でサブプログラムの順序を並べ替えることで効果が上がるのは、アプリケーション時間の多くの割合がアプリケーションテキストのページフォルト時間に費やされている場合だけです。それ以外の場合は、順序を変えてもアプリケーションの全体的なパフォーマンスが向上しないことがあります。

相互の関連性

-xF オプションは、-features=no%except (-noex) のときにだけ有効です。

関連項目

analyzer(1)、debugger(1)、および ld(1) のマニュアルページ

-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。

-xhelp=readme

README (最新情報) ファイルの内容を表示します。

README ファイルのページングには、環境変数 PAGER で指定されているコマンドが使用されます。PAGER が設定されていない場合、デフォルトのページングコマンド more が使用されます。

-xildoff

インクリメンタルリンカーを無効にします。

デフォルト

-g オプションを使用していない場合は、この -xildoff オプションがデフォルトになります。さらに -G オプションを使用しているか、コマンド行にソースファイルを指定している場合も、このオプションがデフォルトになります。このオプションを無効にするには、-xildon オプションを使用してください。

関連項目

-xildon 、ild(1) および ld(1) のマニュアルページ、
『dbx コマンドによるデバッグ』

-xildon

インクリメンタルリンカーを有効にします。

-G ではなく -g を使用し、コマンド行にソースファイルを指定していない場合は、このオプションが有効になります。このオプションを無効にするには、-xildoff オプションを使用してください。

関連項目

-xildoff と、ild(1) および ld(1) のマニュアルページ
『dbx コマンドによるデバッグ』

-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。

関連項目

『数値計算ガイド』

-xlibmil

選択された libm ライブラリルーチンを最適化のためにインライン展開します。

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。

一部の libm ライブラリルーチンにはインラインテンプレートがあります。このオプションを指定すると、これらのテンプレートが選択され、現在選択されている浮動小数点オプションとプラットフォームに対して最も高速な実行可能コードが生成されます。

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』

-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。

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。

相互の関連性

-xlibmopt 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、-xnolibmopt

-xlic_lib=sunperf

(SPARC) Sun Performance Library™ とリンクします。
-l と同様、このオプションは、ソースまたはオブジェクトファイル名に続けて、コマンド行の最後に指定する必要があります。

関連項目

README ファイル『performance_library』

-xlicinfo

ライセンスサーバー情報を表示します。

このオプションは、ライセンスサーバー名と、検査済みのライセンスを所持するユーザーのユーザー ID を返します。このオプションを指定するとコンパイラは起動されず、ライセンスも検査されません。

矛盾するオプションを使用すると、コマンド行の最後のものが優先され、警告が出されます。

例

コンパイルせずにライセンス情報を表示します。

```
example% CC -c -xlicinfo any.cc
```

コンパイルしますが、ライセンス情報は表示しません。

```
example% CC -xlicinfo -c any.cc
```

-Xm

-features=iddollar と同じです。

-xM

メークファイルの依存情報を出力します。

例

プログラム foo.cc には次の文が含まれています。

```
#include "foo.h"
```

foo.c を -xM でコンパイルすると、次の行が出力に含まれます。

```
foo.o:foo.h
```

関連項目

メークファイルおよび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。

-xM1

依存情報を生成しますが、/usr/include は除きます。

/usr/include ヘッダーファイルについての依存情報を出力しない点以外は、-xM オプションと同じ機能です。

-xMerge

(SPARC) データセグメントをテキストセグメントと併合 (マージ) します。

オブジェクトファイルのデータは読み取り専用です。また、このデータは ld -N を指定してリンクしない限りプロセス間で共有されます。

関連項目

ld(1) のマニュアルページ

-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。

- 標準モード (デフォルトモード)

```
-lCstd -lCrun -lm -lw -lcx -lc
```

- 互換モード (-compat)

```
-lC -lm -lw -lcx -lc
```

-l オプションの順序は重要です。-lm、-lw、-lcx オプションは -lc より前になければなりません。

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 -xarch=v9 -dryrun
```

上記の出力には次の行が含まれます。

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

-xarch=v9 を指定したときは、-lcs がリンクされないことに注意してください。

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場合、つまり、C サポートだけが必要な C++ プログラムの場合は、次のように指定します。

```
example% CC -xnolib test.cc -lc
```

一般的なアーキテクチャ命令を持つシングルスレッドアプリケーションに libm を静的にリンクするには、次のように指定します。

標準モードの場合

```
example% CC -xnolib test.cc -lCstd -lCrun \
-Bstatic -lm -Bdynamic -lw -lcx -lc
```

互換モードの場合

```
example% CC -compat -xnolib test.cc -lC \
-Bstatic -lm -Bdynamic -lw -lcx -lc
```

相互の関連性

-xarch=v9、-xarch=v9a、-xarch=v9b のいずれかでリンクする場合には、使用できない静的システムライブラリがあります (libm.a や libc.a など)。

-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なりません。

-xnolib を指定すると、-library は無視されます。

警告

C++ 言語の多くの機能では、libC (互換モード) または 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。

-lcx は 64 ビットコンパイルモードにはありません。

関連項目

-library、-staticlib、-l

-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。

-xnolibmopt

数学ルーチンのライブラリを使用しないようにします。

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、その後に使用してください。

```
example% CC -fast -xnolibmopt
```

-xO*level*

最適化レベルを指定します。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、実行速度が速くなります。

`-xO`*level* を指定しないと、非常に基本的なレベルの最適化しか行われません。つまり、最適化は、式の局所的な共通部分を削除することと、デッドコードを分析することに限定されます。最適化レベルを指定してコンパイルすると、プログラムのパフォーマンスが著しく向上することがあります。ほとんどのプログラムの場合、`-xO2` (または同等のオプション `-O` および `-O2`) を使用することをお勧めします。

一般に、プログラムをより高い最適化レベルでコンパイルすれば、実行時のパフォーマンスはそれだけ向上します。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` の方が他の値より実行速度が速くなることがあり、`-x03` の方が `-x04` より速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、`-xO`*level* オプションで指定された最適化レベルを使用します。

`-xO` には 5 つのレベルがあります。以降では各レベルが SPARC および IA プラットフォームでどのように動作するかを説明します。

値

SPARC プラットフォームの場合

- `-xO1` 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。`-xO2` や `-xO3` 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 `-xO1` を使用してください。

- `-xO2` 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。一般に、このレベルを使用するとコードサイズが最小になります。

注 – `-O` は `-xO2` を指定することと同じです。

- `-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` で適切に保護されていないデバイスドライバをコンパイルする場合か、シグナルハンドラの中から外部変数を修正するプログラムをコンパイルする場合は、`-xO2` を使用してください。一般に `-xO3` を使用すると、コードサイズが大きくなります。スワップ領域が不足する場合は、`-xO2` を使用してください。

- `-xO4` では、`-xO3` レベル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インライン展開も自動的に行われます。インライン展開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コードサイズが大きくなります。

- `-xO5` では、最高レベルで最適化が行われます。これを使用するのは、コンピュータの最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。114 ページの「-xprofile=p」 を参照してください。

IA プラットフォームの場合

- -xO1 では、引数がメモリーから事前にロードされます。その結果、デフォルトの最適化の第 1 段階で行われる単純な最適化に加え、クロスジャンプ (末尾のマージ) も行われます。
- -xO2 では、レベル 1 で行われる最適化に加えて、高レベルと低レベルの命令のスケジューリング、スピル解析、ループメモリー参照の削除、レジスタ寿命解析、レジスタ割り当ての強化、大域的な共通部分式の削除が行われます。
- -xO3 では、レベル 2 で行われる最適化に加えて、ループ力の縮小とインライン展開が行われます。
- -xO4 では、レベル 3 で行われる最適化に加えて、アーキテクチャ固有の最適化が行われます。
- -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。

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。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。

-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 を指定してデバッグを行なっても -xO*level* には影響はありません。しかし、-xO*level* によって -g がある程度の制限を受けます。たとえば、-xO*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。

-xcrossfile 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。

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、途方もない大きさのメモリーが必要になり、マシンのパフォーマンスが低下することがあります。

こうしたパフォーマンスの低下を防ぐには、`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大きさを制限します (`csh(1)` のマニュアルページを参照)。たとえば、使用できる仮想メモリーを 16M 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 16M
```

このコマンドにより、データ領域が 16M 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。

実際のスワップ空間に関する情報を得るには、**`swap -l`** と入力します。

実際の実メモリーに関する情報を得るには、**`dmesg | grep mem`** と入力します。

関連項目

`-fast`、`-xcrossfile=`*n*、`-xprofile=`*p*、`csh`(1) のマニュアルページ

-xpg

`-xpg` オプションでは、`gprof` で自動プロファイル処理するためのデータを収集するコードが生成されます。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。

警告

コンパイルとリンクを別々に行う場合は、-xpg でコンパイルしたときは -xpg でリンクする必要があります。

関連項目

-xprofile=*p*、analyzer(1) のマニュアルページ、
『プログラムのパフォーマンス解析』

-xprefetch[=*a*[,*a*]]

(SPARC) 先読み機能をサポートするアーキテクチャで先読み命令を有効にします。たとえば、UltraSPARC-II (-xarch=v8plus、v8plusa、v8plusb、v9、v9a、v9b のいずれか) の場合です。

a は次のどれかです。

値	意味
auto	先読み命令の自動的な生成を有効にします。
no%auto	先読み命令の自動的な生成を無効にします。
explicit	明示的な先読みマクロを有効にします。
no%explicit	明示的な先読みマクロを無効にします。
yes	-xprefetch=yes は -xprefetch=auto,explicit と同じです。
no	-xprefetch=no は -xprefetch=no%auto,no%explicit と同じです。

デフォルト

-xprefetch を指定しないと、-xprefetch=no%auto,explicit が使用されます。

-xprefetch だけを指定すると、-xprefetch=auto,explicit が使用されます。

-xprefetch だけを指定した場合や引数として auto または yes を指定した場合以外は、デフォルトで no%auto が使用されます。たとえば、-xprefetch=explicit は -xprefetch=explicit,no%auto と同じことです。

no%explicit か no を指定した場合以外は、デフォルトで explicit が使用されます。たとえば、-xprefetch=auto は -xprefetch=auto,explicit と同じことです。

相互の関連性

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=explict か -xprefetch=yes が指定されていなければなりません。

マクロが呼び出され、sun_prefetch.h ヘッダーファイルがインクルードされていても、-xprefetch=no%explict か -xprefetch=no が指定されていると、明示的な先読み命令は実行コードに組み込まれません。

-xprefetch、-xprefetch=auto、-xprefetch=yes の場合には、コンパイラは生成するコードに先読み命令を自動的に挿入します。これにより、先読みをサポートするアーキテクチャではパフォーマンスが向上する場合があります。

警告

明示的な先読み命令の使用は、パフォーマンスが実際に向上する特別な場合に限定してください。

-xprofile=*p*

実行時プロファイルデータを収集したり、それを使って最適化します。

このオプションを使用すると、実行頻度のデータが集められて、実行時に保存されます。保存されたデータは後続する処理の実行時に使用され、これによってパフォーマンスが向上します。このオプションは、最適化のレベルが指定されている場合にのみ有効です。

値

p は次のいずれかでなければなりません。

表 3-33 `-xprofile` オプション

p の値	意味
`collect` [:*name*]	実行頻度のデータを集めて保存します。後に `-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。コンパイラは、コードを生成して実行頻度を計ります。*name* には分析するプログラム名を指定します。*name* は省略可能です。指定しなかった場合、`a.out` と仮定されます。 `-xprofile=collect:`*name* でコンパイルしたプログラムは、実行時に、実行時のフィードバック情報を書き込むサブディレクトリ *name*`.profile` を作成します。データは、このサブディレクトリのファイル `feedback` に書き込まれます。プログラムを繰り返し実行すると、実行頻度のデータが `feedback` ファイルに累積されます。つまり、前の実行で出力されたデータはなくなりません。
`use` [:*name*]	有効な最適化を行うために実行頻度データを使います。*name* には分析する実行可能ファイル名を指定します。*name* は省略可能で、省略すると実行可能ファイル名は `a.out` とみなされます。 プログラムは、前の実行で `feedback` ファイルに生成され、保存された実行頻度データを使って最適化されます。このファイルは、`-xprofile=collect` でコンパイルしたプログラムを前に実行したときに書き込まれたものです。 ソースファイルと他のコンパイラオプションは、`feedback` ファイルを生成したコンパイル済みプログラムをコンパイルしたときとまったく同じでなければなりません。`-xprofile=collect:`*name* でコンパイルしたのであれば、同じプログラム名 *name* を最適化コンパイルの `-xprofile=use:`*name* にも指定しなければなりません。

表 3-33 `-xprofile` オプション (続き)

p の値	意味
`tcov`	「新しい」形式の `tcov` を使った基本ブロックカバレージ解析。 `tcov` の基本ブロックプロファイルの新しい形式です。`-xa` オプションと類似した機能を持つが、ヘッダーファイルにソースコードが含まれているプログラムや、C++ テンプレートを使用するプログラムのデータを集めます。コード生成は `-xa` オプションと類似していますが、`.d` ファイルは生成されません。その代わりにファイルが 1 つ生成されます。このファイルの名前は最終的な実行可能ファイルに基づきます。たとえば、 `/foo/bar` にある `myprog` を実行する場合、データファイルは `/foo/bar/myprog.profile/tcovd` に保存されます。 `tcov` を実行する場合は、新しい形式のデータが使用されるように `-x` オプションを指定します。`-x` オプションを指定しないと、デフォルトで古い形式の `.d` ファイルが使用され、予期しない結果が出力されます。 `-xa` オプションとは異なり、`TCOVDIR` 環境変数はコンパイル時間には影響しません。ただし、`TCOVDIR` 環境変数の値はプログラムの実行時に使用されます。

相互の関連性

`-xprofile=tcov` オプションと `-xa` オプションは、1 つの実行可能ファイルで同時に使用できます。つまり、`-xprofile=tcov` でコンパイルされたファイルと `-xa` でコンパイルされたファイルからなるプログラムをリンクすることはできますが、両方のオプションを使って 1 つのファイルをコンパイルすることはできません。

`-xO4` を使用したために、関数のインライン化が行われている場合は、`-xprofile=tcov` によって生成されたコードカバレージ報告は信用できない可能性があります。

警告

コンパイルとリンクを別々に実行する場合で、コンパイルで `-xprofile` オプションを使用した場合は、リンクでも `-xprofile` を使用する必要があります。

関連項目

-xa、tcov(1) のマニュアルページ
『プログラムのパフォーマンス解析』

-xregs=*r*[,...*r*]

(SPARC) 一時レジスタの使用を制御します。

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限りません。

値

r には次の値のいずれかを指定します。各値の意味は -xarch の設定によって異なります。

r の値	内容
[no%]appl	V8 および V8a の場合、レジスタ %g2、%g3、%g4 の使用を許可します [しません]。 v8plus、v8plusa、および v8plusb の場合、レジスタ g2、g3、g4、g5 の使用を許可します [しません]。 V9、V9a、および v9b の場合、レジスタ %g2 と %g3 の使用を許可します [しません]。 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 load や store 命令が少なくなるため、パフォーマンスが向上します。ただし、これらのレジスタの使用は、他の目的でレジスタを使用するプログラムとの矛盾を起こすことがあります。
[no%]float	SPARC ABI で指定されているように、浮動小数点レジスタの使用を許可します [しません]。 プログラム中に浮動小数点コードが含まれていない場合でも、これらのレジスタを使用できます。 浮動小数点コードが含まれているソースプログラムには、このオプションを使用できません。

デフォルト

-xregs を指定しないと、-xregs=appl,float が使用されます。

例

使用可能なすべての一時レジスタを使ってアプリケーションプログラムをコンパイルするには、次のように指定します。

```
-xregs=appl,float
```

コンテキストの切り替えの影響を受けやすい非浮動小数点コードをコンパイルするには、次のように指定します。

```
-xregs=no%appl, no%float
```

関連項目

SPARC V7 および V8 の ABI、SPARC V9 の ABI

-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。

-xs オプションは、dbx の自動読み込みを無効にします。このオプションは、.o ファイルを残しておくことができない場合に使用してください。このオプションにより、-s オプションがアセンブラに渡されます。

「非自動読み込み」とは、シンボルテーブルの古い読み込み方法です。dbx の全シンボルテーブルが実行ファイル内に置かれます。また、リンカーによるリンクや dbx による初期化の速度が遅くなります。

「自動読み込み」は、シンボルテーブルの新しい読み込み方法 (デフォルト) です。各 .o ファイルに情報が含まれるため、dbx はシンボルテーブルが必要な場合にのみシンボルテーブル情報を読み込みます。このため、リンカーによるリンクや dbx による初期化の速度が速くなります。

-xs を指定する場合で、実行ファイルを別のディレクトリに移動して dbx を使用するときは、オブジェクト (.o) ファイルを移動する必要はありません。

-xs を指定せずに実行ファイルを別のディレクトリに移動して dbx を使用する場合は、ソースファイルとオブジェクト (.o) ファイルの両方を移動する必要があります。

-xsafe=mem

(SPARC) メモリーに関するトラップを発生させません。

このオプションによって、V9 マシン上で投機的なロード命令を使用することができます。

相互の関連性

このオプションは、`-xarch` で `v8plus`、`v8plusa`、`v8plusb`、`v9`、`v9a`、`v9b` のいずれかを指定し、最適化レベルの `-xO5` と組み合わせた場合にだけ有効です。

警告

このオプションは、プログラムでメモリーに関するトラップが起こらないとみなせる場合しか使用しないでください。ほとんどのプログラムの場合、この仮定は適切であり、このように仮定してもかまいません。しかし、メモリーに関するトラップを明示的に強制して例外条件を処理するプログラムの場合は、このように仮定することはできません。

-xsb

WorkShop ソースブラウザ用の情報を生成します。

このオプションを指定すると、`CC` ドライバが、ソースブラウザのために `SunWS_cache` サブディレクトリにシンボルテーブル情報を追加生成します。

関連項目

`-xsbfast`

-xsbfast

ソースブラウザ情報を生成するだけで、コンパイルはしません。

このオプションでは、`ccfe` 段階だけを実行して、ソースブラウザのために `SunWS_cache` サブディレクトリにシンボルテーブル情報を追加生成します。オブジェクトファイルは生成されません。

関連項目

-xsb

-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*

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を重視する場合は、ハードウェアを適切に指定することが極めて重要です。特に、プログラムをより新しい SPARC システム上で実行する場合には重要になります。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。

値

SPARC プラットフォームの場合

SPARC プラットフォームでは、*t* には次のいずれかの値を指定します。

t の値	意味
native	ホストシステムで最高のパフォーマンスが得られます。 コンパイラは、ホストシステム用に最適化されたコードを生成し、コンパイラが動作しているマシンで使用できるアーキテクチャ、チップ、キャッシュの属性を決定します。
generic	汎用アーキテクチャ、チップ、キャッシュで最高のパフォーマンスが得られます。 コンパイラは、-xtarget=generic を -xarch=generic -xchip=generic -xcache=generic に展開します。 これはデフォルト値です。
platform-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。 29 ページの表 3-6 から SPARC プラットフォームの名前を選択します。

次の表は、-xtarget に指定できる SPARC プラットフォームの名前とその展開値を示しています。

表 3-34 -xtarget の SPARC プラットフォーム名

-xtarget	-xarch	-xchip	-xcache
generic	generic	generic	generic
cs6400	v8	super	16/32/4:2048/64/1
entr150	v8	ultra	16/32/1:512/64/1
entr2	v8	ultra	16/32/1:512/64/1
entr2/1170	v8	ultra	16/32/1:512/64/1
entr2/1200	v8	ultra	16/32/1:512/64/1
entr2/2170	v8	ultra	16/32/1:512/64/1
entr2/2200	v8	ultra	16/32/1:512/64/1
entr3000	v8	ultra	16/32/1:512/64/1
entr4000	v8	ultra	16/32/1:512/64/1
entr5000	v8	ultra	16/32/1:512/64/1
entr6000	v8	ultra	16/32/1:512/64/1
sc2000	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1

表 3-34 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget	-xarch	-xchip	-xcache
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1

表 3-34 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget	-xarch	-xchip	-xcache
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1

表 3-34 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget	-xarch	-xchip	-xcache
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/630	v7	old	64/32/1
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
ultra	v8	ultra	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/200	v8	ultra	16/32/1:512/64/1
ultra2	v8	ultra2	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:512/64/1
ultra2/1200	v8	ultra	16/32/1:1024/64/1
ultra2/1300	v8	ultra2	16/32/1:2048/64/1
ultra2/2170	v8	ultra	16/32/1:512/64/1
ultra2/2200	v8	ultra	16/32/1:1024/64/1
ultra2/2300	v8	ultra2	16/32/1:2048/64/1
ultra2i	v8	ultra2i	16/32/1:512/64/1
ultra3	v8	ultra3	64/32/4:8192/256/1

IA プラットフォームの場合

IA の場合は、-xtarget に次の値を指定できます。

- native または generic
- 386
  Intel 80386 マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。
- 486
  Intel 80486 マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。
- pentium
  Pentium または Pentium Pro マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。
- pentium_pro
  Pentium Pro マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。

デフォルト

SPARC でも IA でも、-xtarget を指定しないと、-xtarget=generic が使用されます。

展開

-xtarget オプションは、購入したプラットフォーム上で使用する -xarch、-xchip、-xcache の組み合わせを簡単に指定するためのマクロです。-xtarget 自体の意味は、展開することです。

例

-xtarget=sun4/15 は -xarch=v8a -xchip=micro -xcache=2/16/1 を意味します。

相互の関連性

-xarch=v9|v9a|v9b オプションを指定して SPARC V9 アーキテクチャ用にコンパイルする場合、-xtarget=ultra または ultra2 の指定は必要でないか、十分ではありません。-xtarget を指定する場合は、-xarch=v9|v9a|v9b オプションは -xtarget よりも後になければなりません。たとえば、次のように指定するとします。

```
-xarch=v9 -xtarget=ultra
```

上記の指定は次のように展開され、-xarch の値が v8 に戻ります。

```
-xarch=v9 -xarch=v8 -xchip=ultra -xcache=16/32/1:512/64/1
```

正しくは、次のように、-xarch を -xtarget よりも後に指定します。

```
-xtarget=ultra -xarch=v9
```

-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。

-xunroll=*n*

可能な場合は、ループを展開します。

このオプションでは、コンパイラがループを最適化 (展開) するかどうかを指定します。

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。

-xvector[=(yes|no)]

(SPARC) SPARC ベクトルライブラリ関数の自動呼び出しを有効にします。

デフォルト

コンパイラのデフォルトは -xvector=no です。-xvector だけを指定した場合、-xvector=yes が仮定されます。

警告

コンパイルとリンクを別々に行う場合は、どちらにも同じ -xvector 設定を使用する必要があります。

-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。

-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と Solaris 関連のマニュアル『リンカーとライブラリ』を参照してください。

-ztext

書き込み不可で割り当て可能なセクションに対して再配置が残っている場合に、致命的エラーとします。

このオプションはリンカーに渡されます。

第4章

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なければなりません。コンパイラは、テンプレートレポジトリを使って、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しなければなりません。

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、通知されません。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。

テンプレートコマンド

テンプレートレポジトリの管理は CCadmin(1) コマンドで行います。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 -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。

テンプレートインスタンスの配置とリンケージ

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すべての開発に適しており、テンプレートのコンパイルとしては総合的に最も優れています。
- 静的インスタンスは非常に小さなプログラムやデバッグに適しており、用途は限られています。
- 大域インスタンスは、ある種のライブラリ構造に適しています。
- 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、上記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、用途は限られています。

特別な理由がない限り、デフォルトの外部インスタンス方式を使用してください。詳細は、『C++ プログラミングガイド』を参照してください。

外部インスタンスリンケージ

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レ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。

テンプレートインスタンスは、レ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、外部リンケージで現在のコンパイル単位から参照されます。

外部リンケージは、-instances=extern オプションで指定します。このオプションはデフォルトです。

インスタンスはテンプレートレ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したい場合には、CC コマンドに -xar オプションを指定してください。ar コマンドは使用できません。次に例を示します。

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

詳細は、第 6 章を参照してください。

静的インスタンス

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レポジトリに保存されません。

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。これには、次の欠点があります。

- 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ます (したがって、静的インスタンスのリンケージは、テンプレートがインスタンス化される回数が少ない小さなプログラムだけに適しています)。
- 静的変数を持つテンプレートにはその変数のコピーがたくさんあります。これは必然的に C++ 標準に違反することになります。したがって、静的インスタンスはテンプレート内の静的変数には使用できません。

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)。

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。

大域インスタンス

大域インスタンスの場合では、すべてのインスタンスが現在のコンパイル単位の中に置かれます。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。テンプレートはテンプレートデータベースに保存されません。

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以外でも認識したり、使用したりできます。その結果、複数のコンパイル単位におけるインスタンス化でリンク作業中に複数のシンボル定義のエラーが生じることがあります。したがって、大域インスタンスは、インスタンスが繰り返されないことがわかっている場合に限り適しています。

大域インスタンスは、-instances=global オプションで指定します。

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内に置かれるため、テンプレートは再コンパイルのたびに再インスタンス化され、テンプレートレポジトリには保存されません。

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。同じプログラムで複数の明示的なインスタンス化があると、リンカーで複数シンボル定義エラーになります。したがって、明示的インスタンス方式は、明示的なインスタンス化でライブラリを構成する場合のように、インスタンスが繰り返されないことがわかっている場合に限り適しています。

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

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。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、テンプレートレポジトリには保存されません。

明示的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。同じプログラムで複数の明示的インスタンス化があると、リンカーで複数のシンボル定義エラーになります。したがって、半明示的インスタンスは、明示的なインスタンス化によってライブラリを構成する場合のように、明示的インスタンスが繰り返されないことがわかっている場合にだけ適しています。

明示的インスタンスの本体内から使用される暗黙的インスタンスは、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の外では認識できません。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。これには、次の 2 つの欠点があります。

- 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ます (したがって、半明示的インスタンスのリンケージは、テンプレート本体で複数のインスタンス化が起こらないプログラムだけに適しています)。
- 静的変数を持つテンプレートにはその変数のコピーがたくさんあります。これは必然的に C++ 標準に違反することになります。したがって、半明示的インスタンスはテンプレート内の静的変数には使用できません。

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

テンプレートレポジトリ

テンプレートレポジトリには、別々のコンパイルで共有できるテンプレートインスタンスが格納されます。したがって、テンプレートインスタンスは必要なときだけコンパイルされます。テンプレートレポジトリには、外部インスタンスを使用したときにテンプレートのインスタンス化に必要な、ソースファイル以外のすべてのファイルが含まれています。このレポジトリはその他の種類のインスタンスには使用されません。

レポジトリの構造

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

テンプレートレ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レポジトリにそれらを保存します。たとえば、次の例では、オブジェクトファイルを ./sub/a.o に書き込み、テンプレートインスタンスを ./sub/SunWS_cache にあるレポジトリに書き込みます。

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

コンパイラがテンプレートをインスタンス化するときにこのキャッシュディレクトリが存在しない場合は、このディレクトリが作成されます。

複数のテンプレートレポジトリからの読み取り

コンパイラは、読み取るオブジェクトファイルに対応したテンプレートレポジトリから読み取りを行います。たとえば次の例では、/sub1/SunWS_cache と ./sub2/SunWS_cache から読み取り、必要に応じて ./SunWS_cache へ書き込みます。

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

テンプレートレポジトリの共有

レ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なりません。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 (最も保守的で) 最も簡単な方法は、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。

テンプレート定義の検索

定義分離テンプレート編成 (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。したがって、定義検索の必要がない定義取り込み型テンプレートファイル編成を使用するようにしてください。詳細については『C++ プログラミングガイド』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、コンパイラは別のソースファイルを検索しません。

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。この方法では、テンプレート定義ファイルがテンプレート宣言ファイルと同じベース名を持ち、しかも現在の `include` パスにも存在している必要があります。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、または他の何らかの認識可能なソースファイル拡張子 (`.C`、`.c`、`.cc`、`.cpp`、`.cxx`) 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。

定義検索パス

`-I` で設定する通常の検索パスの代わりに、オプションの `-pti`*directory* (ディレクトリ) 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`*directory*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。しかし、`-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* フラグより `-I` フラグを使用してください。

テンプレートインスタンスの自動一貫性

テンプレートレポジトリマネージャは、レ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には、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、テンプレートレ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。

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。Sun Workshop 6 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。

テンプレートオプションファイル

テンプレートオプションファイルとは、テンプレート定義を特定したり、インスタンスを再コンパイルする際に必要なオプションを含む、ユーザーが用意するファイルです (省略も可)。このファイルを使ってテンプレートの特殊化と明示的なインスタンス化を制御することもできます。しかし、現在特殊化の宣言と明示的なインスタンス化に必要な構文はソースコード中で使用できるため、テンプレートオプションをこの用途に使用すべきではありません。

注 – テンプレートオプションファイルは、C++ コンパイラの将来のリリースではサポートされなくなる可能性があります。

オプションファイルの名前は `CC_tmpl_opt` で、`SunWS_config` ディレクトリ内にあります。このディレクトリ名は `SUNWS_CONFIG_NAME` 環境変数で変更できます。

オプションファイルは ASCII テキストファイルで、多くのエントリを含んでいます。エントリはキーワードから始まり、テキストが続き、セミコロン (;) で終わります。エントリは複数行に渡ってもかまいませんが、キーワードは必ず 1 行中に納めるようにしてください。分割してはなりません。

コメント

コメントは # 文字で始まり、その行の終わりまで続きます。コメント内のテキストは無視されます。

```
# コメント中のテキストは行末まで無視される。
```

インクルード

オプションファイルをインクルードすれば、複数のテンプレートデータベース間でオプションファイルを共有できます。この機能は特に、テンプレートを含むライブラリを構築するときに便利です。処理中、指定されたオプションファイルは原文どおりに

現在のオプションファイルにインクルードされます。オプションファイル内では、複数の include 文をどこにでも指定できます。オプションファイルは入れ子にすることもできます。*options-file* はオプションファイル名を示します。

```
include "options-file";
```

ソースファイルの拡張子

コンパイラがデフォルトの Cfront 形式のソースファイル検索機構を使用している場合、extensions エントリを使用すれば、コンパイラが検索するソースファイルの拡張子を指定できます。次に、このエントリの構文を示します。

```
extensions "ext-list";
```

ext-list には有効なソースファイルの拡張子を、空白文字で区切って指定します。

```
extensions ".CC .c .cc .cpp";
```

このエントリがオプションファイルに存在しない場合、コンパイラが検索する拡張子は、.cc、.c、.cpp、.C および .cxx です。

定義ソースの位置

定義ソースファイルの位置は、オプションファイル中の definition エントリで明示的に指定できます。definition エントリは、テンプレートの宣言と定義のファイル名が標準の Cfront 形式の規約に準拠していない場合に使用してください。次に、このエントリの構文を示します。

```
definition name in "file-1",[ "file-2" ..., "file-n"] [nocheck "options"];
```

name フィールドには、このエントリでの指定を適用するテンプレートを指定します。1 つの *name* に使用できる definition エントリは 1 つだけです。*name* に指定する名前は単純な名前である必要があります。つまり、修飾名は使用できません。また、丸

かっこ、戻り型、およびパラメータリストも使用できません。戻り型やパラメータに関わらず、名前そのものだけが重要です。結果として、definition エントリは複数の (おそらくは、多重定義された) テンプレートに適用される可能性があります 。

「*file-n*」リストフィールドには、テンプレート定義が含まれているファイルを指定します。ファイルの検索には、定義検索パスが使用されます。ファイル名は引用符 (") で囲む必要があります。複数のファイルを指定する理由は、指定した単純なテンプレート名がファイルごとに定義されている複数の異なるテンプレート名を参照していたり、1 つのテンプレートの定義がファイルごとに異なっている可能性があるためです。たとえば、func が 3 つのファイルで定義されている場合、これら 3 つのファイルを definition エントリのリストに指定する必要があります。

nocheck フィールドについては、この節の最後で説明します。

次の例では、コンパイラは foo.cc にあるテンプレート関数 foo を見つけ、この関数をインスタンス化します。ただし foo.cc 中の関数 foo はデフォルトの検索でも検出されるため、この definition エントリは冗長です。

コード例 4-1　冗長な definition エントリ

foo.cc	`template <class T> T foo( T t ) { }`
CC_tmpl_opt	`definition foo in "foo.cc";`

次の例では、静的なデータメンバーの定義と単純名の使用を示します。

コード例 4-2　静的なデータメンバーの定義と単純名の使用

foo.h	`template <class T> class foo { static T* fooref; };`
foo_statics.cc	`#include "foo.h"` `template <class T> T* foo<T>::fooref = 0`
CC_tmpl_opt	`definition fooref in "foo_statics.cc";`

fooref の定義で使用されている名前は単純名であり、修飾名 (foo::fooref など) ではありません。この definition エントリを定義する理由は、ファイル名が認識可能な拡張子ではなく (foo.cc など)、デフォルトの Cfront 形式の検索規則ではファイルを見つけることができないためです。

次の例では、テンプレートメンバー関数の定義を示します。例に示すとおり、メンバー関数は静的メンバー初期設定子とまったく同じように処理されます。

コード例 4-3　テンプレートメンバー関数の定義

foo.h	`template <class T> class foo { T* foofunc(T); };`
foo_funcs.cc	`#include "foo.h"` `template <class T> T* foo<T>::foofunc(T t) {}`
CC_tmpl_opt	`definition foofunc in "foo_funcs.cc";`

次の例では、2 つの異なるソースファイルにあるテンプレート関数の定義を示します。

コード例 4-4　異なるソースファイルにあるテンプレート関数の定義

foo.h	`template <class T> class foo {`
	`  T* func( T t );`
	`  T* func( T t, T x );`
	`};`

foo1.cc	`#include "foo.h"` `template <class T> T* foo<T>::func( T t ) { }`
foo2.cc	`#include "foo.h"` `template <class T> T* foo<T>::func( T t, T x ) { }`
CC_tmpl_opt	`definition func in "foo1.cc", "foo2.cc";`

この例では、コンパイラは多重定義されている関数 `func()` の定義を両方とも見つける必要があります。そこで、`definition` エントリで、どこに適切な関数定義があるのかをコンパイラに指示します。

コンパイルフラグが変更されても、再コンパイルが不必要な場合もあります。オプションファイルの `definition` エントリに `nocheck` フィールドを指定すると、不必要な再コンパイルを回避できます。`nocheck` フィールドでオプションを指定すると、コンパイラとテンプレートデータベースマネージャは、そのオプションを依存関係の

検査対象から除外します。特定のコマンド行フラグを追加または削除したために、コンパイラがテンプレート関数を再インスタンス化する必要がない場合は、nocheck フラグを使用してください。次に、このエントリの構文を示します。

```
definition name in "file-1"[, "file-2" ..., "file-n"] [nocheck "options"];
```

オプション (*options*) は引用符 (") で囲む必要があります。

次の例では、コンパイラは foo.cc にあるテンプレート関数 foo を見つけ、この関数をインスタンス化します。後で再インスタンス化のための検査が必要な場合、コンパイラは -g オプションを無視します。

コード例 4-5　nocheck オプション

foo.cc	template <class T> T foo(T t) {}
CC_tmpl_opt	definition foo in "foo.cc" nocheck "-g";

テンプレートの特殊化エントリ

最近まで、C++ 言語はテンプレートを特殊化するための機構を持っておらず、個々のコンパイラが独自の機能を提供していました。この節では、以前のバージョンの C++ コンパイラの機構を使用したテンプレートの特殊化を説明します。この機構は、互換モード (-compat[=4]) でのみサポートされています。

special エントリは、指定された関数が特殊化であり、この関数に遭遇してもインスタンス化してはならないことをコンパイラに指示します。コンパイル時インスタンス化方法を使用する場合は、オプションファイルの中で special エントリを使用して、特殊化を事前に登録してください。次に、このエントリの構文を示します。

```
special declaration;
```

宣言 (*declaration*) には、戻り型がない正しい C++ 形式の宣言を指定します。次に例を示します。

コード例 4-6　`special` エントリ

`foo.h`	`template <class T> T foo( T t ) { };`
`main.cc`	`#include "foo.h"`
`CC_tmpl_opt`	`special foo(int);`

上記の `special` エントリを含むオプションファイルは、テンプレート関数 `foo()` を `intd` 型にインスタンス化してはならないこと、および、特殊化された関数 `foo()` がユーザーから提供されることをコンパイラに指示します。このエントリをオプションファイルに指定しない場合、関数は不必要に再インスタンス化され、その結果、エラーが発生します。

コード例 4-7　`special` エントリを使用する必要がある場合

`foo.h`	`template <classT> T foo( T t ) { return t + t; }`
`file.cc`	`#include "foo.h"` `int func( ) { return foo( 10 ); }`

`main.cc`	`#include "foo.h"`
	`int foo( int i ) { return i * i; } // 特殊化`
	`int main( ) { int x = foo( 10 ); int y = func();`
	`return 0; }`

上記の例では、`main.cc` をコンパイルするとき、コンパイラはその定義をあらかじめ確認しているため、特殊化された `foo` を正しく使用します。しかし、`file.cc` をコンパイルするとき、コンパイラは `main.cc` に `foo` が存在することを知らないため、`foo` に対して独自のインスタンス化を行います。この結果、ほとんどの場合は、このリンク中にシンボルが複数回定義されるだけですが、場合によっては (特にライブラリの場合)、間違った関数が使用され、実行時エラーが発生することがあります。特殊化された関数を使用する場合は、その特殊化を登録しておくことをお勧めします。

special エントリは多重定義できます。次に例を示します。

コード例 4-8　special エントリの多重定義

foo.h	template <classT> T foo(T t) {}
main.cc	#include "foo.h" int　foo(int　i) {} char* foo(char* p) {}
CC_tmpl_opt	special foo(int); special foo(char*);

テンプレートクラスを特殊化するには、special エントリにテンプレート引数を指定します。

コード例 4-9　テンプレートクラスの特殊化

foo.h	template <class T> class Foo { ... various members ... };
main.cc	#include "foo.h" int main( ) { Foo<int> bar; return 0; }
CC_tmpl_opt	special class Foo<int>;

テンプレートクラスメンバーが静的なメンバーの場合、special エントリにキーワード static を指定する必要があります。

コード例 4-10 静的テンプレートクラスメンバーの特殊化

foo.h	template <class T> class Foo { public: static T func(T); }; };
main.cc	#include "foo.h" int main( ) { Foo<int> bar; return 0; }
CC_tmpl_opt	special static Foo<int>::func(int);

第5章

ライブラリの使用

ライブラリを使用すると、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を単純化することができます。Sun WorkShop 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。

C ライブラリ

Solaris オペレーティング環境では、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`、`libm`、`libw` 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。それ以外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
```

Sun Workshop 6 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。

C++ コンパイラ付属のライブラリ

Sun 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。これらのライブラリには、互換モード (`-compat=4`) だけで使用できるもの、標準モード (`-compat=5`) だけで使用できるもの、あるいは両方のモードで使用できるものがあります。`libgc` ライブラリと `libdemangle` ライブラリには C インタフェースがあり、どちらのモードでもアプリケーションにリンクできます。

次の表に、Sun 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 5-1　C++ コンパイラに添付されるライブラリ

ライブラリ	内容	使用できるモード
`libCrun`	C++ 実行時	`-compat=5`
`libCstd`	C++ 標準ライブラリ	`-compat=5`
`libiostream`	従来の `iostream`	`-compat=5`
`libC`	C++ 実行時、従来の `iostream`	`-compat=4`
`libcomplex`	複素数ライブラリ	`-compat=4`
`librwtool`	`Tools.h++ 7.0`	`-compat=4`、`-compat=5`
`librwtool_dbg`	デバック可能な `Tools.h++ 7.0`	`-compat=5`
`libgc`	ガベージコレクション	C インタフェース
`libgc_dbg`	デバッグ可能なガベージコレクション	`-compat=4`、`-compat=5`
`libdemangle`	復号化	C インタフェース

C++ライブラリの説明

これらのライブラリについて簡単に説明します。

- `libCrun:` このライブラリには、コンパイラが標準モード (`-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:` これは C++ 標準ライブラリです。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、標準 C++ ライブラリのマニュアルを参照してください。

- `libiostream:` これは標準モード (`-compat=5`)で構築した従来の `iostream` ライブラリです。既存のソースで従来の `iostream` を使用している場合には、`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。

- `libC:` これは互換モード (`-compat=4`) で必要なライブラリです。このライブラリには C++ 実行時サポートだけでなく従来の `iostream` も含まれています。

- `libcomplex:` このライブラリは、互換モード (`-compat=4`) で複素数の演算を行うときに必要です。標準モードの場合は、 `libCstd` の複素数演算の機能が使用されます。

- `libwtool:` (Tools.h++ 7) これは RougeWave の Tools.h++ バージョン 7 ライブラリです。

- `libgc:` これはガベージコレクションライブラリ (Sun WorkShop Memory Monitor の構成要素) です。このライブラリの文書にアクセスするには、Memory Monitor を起動するか、Web ブラウザで次のファイルを参照してください。

  `file:`*install-directory*`/SUNWspro/docs/ja/index.html`

  install-directory は、Sun WorkShop インストールディレクトリへのパスに変更します。デフォルトのインストールでは、*install-directory* は `/opt` です。

- `libdemangle:` このライブラリは C++ 符号化名を復号化するときに使用します。

デフォルトの C++ ライブラリ

これらのライブラリには、`CC` ドライバによってデフォルトでリンクされるものと、明示的にリンクしなければならないものがあります。標準モードでは、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 -lw -lcx -lc
```

互換モード (`-compat`) では、次のライブラリがデフォルトでリンクされます。

-lC -lm -lw -lcx -lc

詳細は、67 ページの「-library=l[,...l]」を参照してください。

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。
- Sun C++ コンパイラに添付された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
  - librwtool、librwtool_dbg
  - libgc、libgc_dbg

-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。次にオプションの例をいくつか示します。

- 次のコマンドでは Tools.h++ バージョン 7 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 test.cc が互換モードでコンパイルされ、libC が静的にリンクされます。互換モードでは libC がデフォルトでリンクされるので、このライブラリを -library オプションで指定する必要はありません。

```
example% CC test.cc -compat=4 -staticlib=libC
```

- 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
```

本来ならデフォルトで使用される libCrun ライブラリと libCstd ライブラリが、リンクされなくなります。

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なシステムライブラリをリンクします。-xnolib (または-nolib) が指定された場合は、-l オプションで明示的に指定されたライブラリだけをリンクします (-xnolib または -nolib が使用された場合は、-library オプションを指定しても無視されます)。

-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、デフォルトで -R/opt/SUNWspro/lib を ld に渡します (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)。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。

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。

iostream ライブラリ

Sun Workshop 6 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、4.2 コンパイラに添付された iostream ライブラリ、およびそれ以前に cfront ベースの 3.0.1 コンパイラに添付された iostream ライブラリを指します。このライブラリの標準はありませんが、既存のコードの多くがこれを使用しています。このライブラリは、互換モードの libC の一部であり、標準モードの libiostream にもあります。
- **標準の** iostream：これは C++ 標準ライブラリ libCstd に含まれていて、標準モードだけで使用されます。これは、バイナリレベルでもソースレベルでも「従来の iostream」とは互換性がありません。

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。

```
// ファイル prog1.cc
#include <iostream.h>

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

次のコマンドは、互換性モードで prog1.cc をコンパイル、リンクして、prog1 という実行可能なプログラムを生成します。従来の iostream ライブラリは、互換性モードのときにデフォルトでリンクされる libC ライブラリに含まれています。

```
example% CC -compat prog1.cc -o prog1
```

次の例では、標準の iostream が使用されています。

```
// ファイル 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。コンパイルは標準モードで行われ、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
```

complex ライブラリ

標準ライブラリには、C++ 4.2 コンパイラに付属していた complex ライブラリに似た、テンプレート化された complex ライブラリがあります。標準モードでコンパイルする場合は、<complex.h> ではなく、<complex> を使用する必要があります。互換性モードで <complex> を使用することはできません。

互換性モードでは、リンク時に complex ライブラリを明示的に指定しなければなりません。標準モードでは、complex ライブラリは libCstd に含まれており、デフォルトでリンクされます。

標準モード用の complex.h ヘッダーはありません。C++ 4.2 では、「complex」はクラス名ですが、標準 C++ では「complex」はテンプレート名です。したがって、旧式のコードを変更せずに動作できるようにする typedef を使用することはできません。このため、複素数を使用する、4.2 用のコードで標準ライブラリを使用するには、多少の編集が必要になります。たとえば、次のコードは 4.2 用に作成されたものであり、互換性モードでコンパイルされます。

```
// ファイル ex1.cc （互換モード）
#include <iostream.h>
#include <complex.h>

int main()
{
    complex x(3,3), y(4,4);
    complex z = x * y;
    cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
}
```

次の例では、ex1.cc を互換モードでコンパイル、リンクし、生成されたプログラムを実行しています。

```
example% CC -compat ex1.cc -library=complex
example% a.out
x=(3, 3), y=(4, 4), z=(0, 24)
```

次は、標準モードでコンパイルされるように ex2.cc と書き直された ex1.cc です。

```
// ファイル ex2.cc (ex1.cc rewritten for standard mode)
#include <iostream>
#include <complex>
int main()
{
     std::complex<double> x(3,3), y(4,4);
     std::complex<double> z = x * y;
     std::cout << "x=" << x << ", y=" << y << ", z=" << z <<
       std::endl;
}
```

次の例では、書き直された ex2.cc をコンパイル、リンクして、生成されたプログラムを実行しています。

```
demo% CC ex2.cc
demo% a.out
x=(3,3), y=(4,4), z=(0,24)
```

C++ライブラリのリンク

次の表は、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をまとめています。詳細は、67 ページの「-library=l[,...l]」を参照してください。

表 5-2　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	コンパイルモード	オプション
従来の iostream	-compat=4	不要
	-compat=5	-library=iostream
complex	-compat=4	-library=complex
	-compat=5	不要
Tools.h++ バージョン 7	-compat=4	-library=rwtool7
	-compat=5	-library=rwtool7,iostream
デバッグ対応 Tools.h++ バージョン 7	-compat=4	-library=rwtool7_dbg
	-compat=5	-library=rwtool7_dbg,iostream
ガベージコレクション	-compat=4	-library=gc
	-compat=5	-library=gc
デバッグ対応ガベージコレクション	-compat=4	-library=gc_dbg
	-compat=5	-library=gc_dbg

標準ライブラリの静的リンク

デフォルトでは、CC ドライバは、デフォルトライブラリのそれぞれについて -l*lib* オプションをリンカーに渡すことによって、libc や libm などの共有ライブラリをいくつか静的にリンクします (互換性モードと標準モードのデフォルトライブラリについては、147 ページの「デフォルトの C++ ライブラリ」を参照)。

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すれば、C++ ライブラリを静的にリンクできます。この方法は、以前説明した方法よりもかなり簡単です。次に例を示します。

```
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。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 %none,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。-l オプションは、自分で渡す必要があります。次の例は、Solaris 2.6、Solaris 7、Solaris 8 のいずれかのオペレーティング環境で libCrun と静的に、libw、libm、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 -lw -lcx -lc
```

-l オプションの順序は重要です。-lc の前に -lCstd、-lCrun、-lm、-lw、-lcx オプションがあることに注意してください。

注 – IA プラットフォームでは、-lcx オプションはありません。

他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、106 ページの「-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。

共有ライブラリの使用

C++ コンパイラには、次の共有ライブラリが含まれています。

- libCrun.so.1
- libC.so.5
- libcomplex.so.5
- librwtool.so.2
- libgc.so.1
- libgc_dbg.so.1

プログラムにリンクされた各共有オブジェクトは、生成される実行可能ファイル (a.out ファイル) に記録されます。この情報は、実行時に ld.so が使用して動的リンク編集を行います。ライブラリコードをアドレス空間に実際に組み込むのは後になるため、共有ライブラリを使用するプログラムの実行時の動作は、環境の変化 (つまり、ライブラリを別のディレクトリに移動すること) に影響を受けます。たとえば、プログラムが /opt/SUNWspro/*release*/lib の libcomplex.so.5 とリンクされている場合、後で libcomplex.so.5 ライブラリを /opt2/SUNWspro/*release*/lib に移動すると、このバイナリコードを実行したときに次のメッセージが表示されます。

```
ld.so.1: a.out: libcomplex.so.5: open に失敗しました :
ファイルもディレクトリもありません。
```

ただし、環境変数 LD_BINARY_PATH に新しいライブラリの ディレクトリを設定すれば、古いバイナリコードを再コンパイルせずに実行できます。

C シェルでは次のように入力します。

```
example% setenv LD_LIBRARY_PATH \
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
```

Bourne シェルでは次のように入力します。

```
example$ LD_LIBRARY_PATH=\
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
example$ export LD_LIBRARY_PATH
```

注 – *release* には Sun WorkShop の各リリース番号を指定します。

LD_BINARY_PATH には、ディレクトリのリストが含まれています。ディレクトリは通常コロンで区切られています。C++ のプログラムを実行すると、動的ローダーがデフォルトディレクトリより前に LD_BINARY_PATH のディレクトリを検索します。

実行可能ファイルにどのライブラリが動的にリンクされるのかを知るには、ldd コマンドを使用します。

```
example% ldd a.out
```

共有ライブラリを移動することはめったにないので、この手順が必要になることはほとんどありません。

注 – 共有ライブラリを dlopen で開く場合は、RTLD_GLOBAL を使用しないと例外が機能しません。

共有ライブラリの詳しい使い方については、『リンカーとライブラリ』を参照してください。

C++ 標準ライブラリの置き換え

コンパイラに添付されている標準ライブラリの代わりに別の標準ライブラリを使用することは危険で、必ずしもよい結果にはつながるわけではありません。しかし、パフォーマンス、機能、または他のシステムとの互換性のために異なるバージョンの C++ 標準ライブラリを使用したい場合には、Sun WorkShop 6 C++ で使用するライブラリを変更することができます。基本的な操作としては、コンパイラに添付されてい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格納されているディレクトリとライブラリ自身の名前を指定します。

置き換える対象

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。たとえば libCstd ライブラリを別のものに置き換える場合は、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。

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、Solaris オペレーティング環境と基本 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++ 版 (<cstdlib>、<cstdio>、<cstring> など) は基本の C 版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。

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。なお、ライブラリの名前を lib で始めると後々便利です。

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。次に例を示します。

```
example% CC -I/opt/mycstd/include -library=no%Cstd ...(コンパイルの場合)
```

-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。次に例を示します。

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd ... (リンクの場合)
```

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。次に例を示します。

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a ... (リンクの場合)
```

-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。

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 Solaris オペレーティング環境に標準で付属しており、/user/include に置かれています。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の名前空間と std 名前空間の両方に存在するという条件が付加されています。Solaris 8 より前のリリースの Solaris オペレーティング環境の C++ コンパイラでは、/usr/include ディレクトリにあるヘッダーはそのまま残して、独自のバージョンのヘッダーを別に用意しています。

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。

- ファイル名に接尾辞がない場合、ヘッダーファイルだけ検索したり、ヘッダーファイル用のメークファイルを作成したりできない

- コンパイラのコマンド行に `-I/usr/include` を指定すると、コンパイラ専用の `include` ディレクトリの前に `/usr/include` が検索されるため、Solaris 2.6 および Solaris 7 オペレーティング環境の正しいバージョンの標準 C ヘッダーが検出されない
- `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
- Solaris 8 より前のリリースの Solaris オペレーティング環境では `.KEEP_STATE` が有効なときのメークファイルのデフォルトの相互依存関係により、標準ヘッダーが実行可能プログラムに置き換えられる可能性がある

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)。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 < > に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、他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 5-3　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`

表 5-3　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
`<fcntl.h>`	`fcntl.h`	標準 C および C++ヘッダー以外
`"string"`	`string`	山かっこではなく、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) を見つけることができなかった場合、コンパイラは、`#include` 指令で指定された名前で検索をやり直します。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。

標準 C++ ヘッダーの置き換え

158 ページの「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、157 ページの「代替ライブラリのインストール」で説明している `SUNWCCh` 版の代替ヘッダーを指定する必要はありません。しかし、これまでに説明したいくつかの問題が発生する可能性もあります。その場合、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` を持つファイルに対してシンボリックリンクを作成することです。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` という名前の他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。

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したい場合、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> の代替ヘッダーとして stdio.h と cstdio を使用したいとします。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6章

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。

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。共有したい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、プログラマは他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、ライブラリ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、通常、接尾辞`.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
- ライブラリを実行時に置き換える場合でも、アプリケーションとリンクし直す必要がない (プログラムの再リンクや再配布をしなくても、Solaris 環境でプログラムが新しい機能を使用できるのは、主にこの仕組みのためです)
- 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
- 共有ライブラリを別の場所に移動すると、システムがライブラリを検索できずに、プログラムを実行できなくなることがある (環境変数 LD_LIBRARY_PATH を使えば、この問題は解決できます)

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わからないため、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。構築の例を次に示します。

```
% CC -c foo.cc # main を含むファイルをコンパイルし、テンプレート
                オブジェクトを作成する
% CC -xar -o foo.a foo.o # すべてのオブジェクトを 1 つのライブラリに集める
```

-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、テンプレートレ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 – -xar フラグは既存のアーカイブの作成や更新のためのもので、保守には使用できません。-xar オプションは ar -cr を実行するのと同じことです。

---

1 つの .o ファイルには 1 つの関数を入れることをお勧めします。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。

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レ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)。さらに、CC コンパイラは、大域変数がライブラリに定義されている場合、動的ライブラリが正しく構築されていないと大域変数を初期化しません。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() が終了した後に呼び出されます。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。CC -G コマンドを使用して動的ライブラリを構築しないと、例外が機能しないことがあります。

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。原因不明のリンクエラーがいくつも出る場合は、-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 libgoo.so.1 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、-o オプションはライブラリのファイル名を指定します。-h オプションは、共有ライブラリの名前を指定しています。-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。

注 – CC -G コマンドは -l オプションを ld に渡しません。共有ライブラリに他の共有ライブラリとの依存関係を持たせたい場合、必要な -l オプションをコマンド行に指定する必要があります。たとえば、共有ライブラリに libCrun.so との依存関係を持たせたい場合、-lCrun をコマンド行に指定する必要があります。

例外を含む共有ライブラリの構築

dlopen() で共有ライブラリを開く場合、例外が機能するようにするには、RTLD_GLOBAL を使用する必要があります。

注 – 例外を含む共有ライブラリを構築するとき、`ld` に`-Bsymbolic` オプションを渡さないでください。必要な例外が捕獲されなくなる可能性があります。

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。

公開ライブラリの構築

他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` は使用しないなど)。

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。V7 と V8 では、これらのレジスタは `%g2`、`%g3`、`%g4` です。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しようとします。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。

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 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。

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` または `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
- 実行時型特定機構 (RunTime Type Information、RTTI) を使用しない

`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` で 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=4` の場合は `libC.so.5`、`-compat=5` の場合は `libCrun.so.1`) に依存していなければなりません。

そのためには、共有ライブラリを構築するときに、`-compat=4` の場合は `-lC`、`-compat=5` の場合は `-lCrun` を次のようにコマンド行に追加します。

```
example% CC -G -compat=4 ... -lC
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。

注 – 共有ライブラリを `dlopen()` で開く場合は、`RTLD_GLOBAL` を使用しないと、例外は機能しません。

マルチスレッド化されたプログラムの構築

マルチスレッド対応ライブラリの構築については、『C++ プログラミングガイド』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。

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。基本型は整数と浮動小数点数であり、他のすべての型は、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。

関数の多相性

「関数の多重定義」を参照。

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。

基底クラス

「継承」を参照。

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、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。フラグやスイッチとも呼ばれます。

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。

事前束縛

「動的束縛」を参照。

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。

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

後入れ先出し法によってデータをスタックの一番上に追加するか、一番上から削除しなければならないデータ記憶方式。

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。

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。

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、他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つけること。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

他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、プログラム制御が他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。そこで、コンパイラは関数名とパラメータを組み合わせた一意の名前を各関数に割り当てます。このことを名前の符号化と呼びます。これによって、型の誤りのないリンケージを行うことができます。名前の符号化は「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。

派生クラス

「継承」を参照。

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を開発・実行するためのソフトウェア技術。

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してこのように呼ばれます。

リンカー

オブジェクトコードとライブラリを結びつけて、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

地理的な領域と言語のどちらか、あるいはその両方に固有な一連の規約。日付、時刻、通貨単位などの形式。

索引

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

あ

と

な

は

ひ

ふ

へ

ま

む