C++ 移行ガイド

Sun WorkShop 6

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

Part No. 806-4839-01
2000 年 6 月 Revision A

原典： *C++ Migration Guide*
Part No: 806-3570-10
Revision A

Adobe PostScript™

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

コード例目次

はじめに

このマニュアルでは、C++ コンパイラのバージョン 4.0、4.0.1、4.1、4.2 から移行するときに知っておく必要がある情報について説明します。この情報は、上記以前のバージョン (3.0 および 3.0.1) から移行する場合にも有効です。バージョン 3.0 および 3.0.1 からの移行に特有情報については、個別に記載します。このマニュアルは C++ に関する実用的な知識と Solaris™ オペレーティング環境および UNIX® コマンドに関する一般的な知識を持つプログラマー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と付録から構成されています。

第 1 章「概要」では、C++ 言語の変更、コンパイラの互換モード、バイナリ互換の問題、条件式、関数ポインタと void* について説明します。

第 2 章「互換モードの使い方」では、C++ コンパイラのバージョン 4.0、4.1、および 4.2 用に記述されたコードをコンパイルする方法について説明します。

第 3 章「標準モードの使い方」では、標準モード (デフォルトのモード) でコンパイルできるようにコードを更新する方法について説明します。

第 4 章「入出力ストリームとライブラリヘッダーの使い方」では、ライブラリとヘッダーファイルの変更点について説明します。

第 5 章「C++ 3.0 からの移行」では、C++ 3.0 コンパイラからの移行について説明します。

第 6 章「C から C++ への移行」では、C のプログラムを C++ へ移行する方法について説明します。

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`

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。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。

マニュアルコレクション

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。

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
	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。
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。

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
	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。
	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。このマニュアルには、旧リリース (Sun WorkShop Visual 5.0) から変更のない機能が記載されています。

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

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
	Sun WorkShop Visual の新機能	Sun WorkShop Visual 6.0 で追加または変更された機能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。
	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

表 P-5　C++ 関連のマニュアルページ

タイトル	内容
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

『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

『Effective C++—50 Ways to Improve Your Programs and Designs』Second Edition、Scott Meyers 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

概要

このマニュアルでは、C++ 4.0、4.01、4.1、4.2 をまとめて「C++ 4」と呼びます。C++ 4 でコンパイルし、動作していた C++ ソースコードは、C++ 言語の定義の変更に起因するいくつかの例外はありますが、そのまま C++ 5.0 および Sun WorkShop™ 6 C++ コンパイラの両方でも機能します。コンパイラには、ほぼすべての C++ 4 コードを変更なしで使用できる互換モード (`-compat[=4]`) が用意されています。

注 – C++ 5.0 コンパイラまたは Sun WorkShop 6 C++ コンパイラの標準モード (デフォルトのモード) でコンパイルしたオブジェクトコードと、それ以前のバージョンの C++ コンパイラでコンパイルした C++ コードとの間には互換性はありません。ただし、古いオブジェクトコードでも、そのライブラリが他への依存関係を持たない場合は C++ 5.0 コンパイラおよび Sun WorkShop 6 C++ コンパイラの両方で使用できます。詳細は、4 ページの「バイナリ互換の問題」を参照してください。

C++言語

C++ は、Bjarne Stroustrup 著『C++ Programming Language』(1986 年刊) で初めて登場し、その後、より公式の解説書として、Margaret Elis、Bjarne Stroustrup 共著『注解 C++リファレンス・マニュアル』(通称 ARM、トッパン刊) が刊行されました。Sun C++ 4 コンパイラは、主としてその ARM の定義に基づき、それに、当時登場しつつあった C++標準の一部の仕様が追加されたものです。C++ 4、特に C++ 4.2 コンパイラに追加された仕様の大部分は、ソースレベルまたはバイナリレベルの互換性の問題を起こさないものに限られていました。

現在 C++ は、国際標準の 1 つ、ISO/IEC 14882:1998 Programming Languages - C++ によって標準が規定されています。標準モードの Sun WorkShop 6 C++ コンパイラでは、ほぼすべての言語仕様が、この国際標準の規定どおりに実装されています。現在のリリースに付属している `README` (最新情報) ファイルには、この標準と異なる仕様の説明が含まれています。

C++ 言語の定義は変更されています。このため、古いソースコードをそのままではコンパイルできないことがあります。この最も顕著な例は、C++の標準ライブラリ全体が名前空間の `std` に定義されるようになったという点です。今や、標準のヘッダー名は `.h` なしの `<iostream>` であり、名前の `cout` および `endl` は大域名前空間ではなく名前空間 `std` に存在するため、標準に厳密に準拠したコンパイラでは従来の C++プログラムをコンパイルすることはできません。

```
#include <iostream.h>
int main() { cout << "Hello, world!" << endl; }
```

Sun WorkShop 6 C++ コンパイラには拡張機能として、標準モードでも従来の C++プログラムがコンパイルできるように `<iostream.h>` ヘッダーが用意されています。言語の変更は、ソースコードの修正を必要とするばかりでなく、バイナリレベルの互換性の問題を起こします。そのため、バージョン 5.0 より前の Sun C++ コンパイラには、C++ 標準に準拠するための変更は行われていません。

新しい C++ 言語機能には、プログラムのバイナリ表現の変更を伴うものあります。この問題については、4 ページの「バイナリ互換の問題」でさらに説明します。

コンパイラの動作モード

Sun WorkShop 6 C++ コンパイラには、「標準」モードと「互換」モードの 2 つの動作モードがあります。

標準モード

標準モードでは、C++ 国際標準の大部分の仕様が実装されており、C++ 4 で受け入れられていた言語とソースレベルの互換性の問題がいくつかあります。

さらに重要なことに、C++ 5.0 コンパイラおよび Sun WorkShop 6 C++ コンパイラの両方は、標準モードでは C++ 4.2 とは異なるアプリケーションバイナリインタフェース (ABI) を使用します。このため、一般的に、標準モードで生成されたコードと 4.2 コンパイラで生成されたコードとの間には互換性がなく、リンクすることはできません。この問題については、4 ページの「バイナリ互換の問題」でさらに説明します。

既存のコードを標準モードでコンパイルするには、いくつかの修正が必要です。修正が必要な理由としては、以下が挙げられます。

- 64 ビットのプログラムに対して互換モードは使用できない。
- 互換モードでは、C++ の重要な標準機能を使用できない。
- C++ 標準に準拠した新しいコードを互換モードでコンパイルできないことがある。つまり、将来アプリケーションに新しいコードを追加できなくなる。
- 4.2 のコードと標準モードの C++コードとをリンクできないため、2 つのバージョンのオブジェクトライブラリの維持管理が必要になる可能性がある。
- 将来、互換モードは廃止される。

互換モード

コンパイラには、C++ 4 から標準モードへの移行のために互換モードが用意されています。互換モードでは、C++ 4 コンパイラと、バイナリレベルでは完全な互換性が、ソースレベルではほぼ完全な互換性が保たれます (「互換性」とは、「上位互換性」を意味します。古いソースコードおよびバイナリコードは、新しいコンパイラで動作できますが、この逆に、新しいコンパイラ用に作成されたコードが古いコンパイラで動作するとは限りません)。互換モードと標準モードの間にはバイナリレベルの互換性はありません。互換モードは、IA と SPARC のプラットフォーム上で動作している Solaris 2.6、Solaris 7、および Solaris 8 オペレーティング環境で使用できます。しかし、SPARC V9 (64 ビット) プロセッサでは使用できません。つまり、新しいコンパイラでコンパイルしたコードと別のバージョンのコンパイラでコンパイルしたコードはリンクできないという制限があります。

互換モードを使用する理由としては、以下が挙げられます。

- たとえば、ソースコードがないために、4.2 コンパイラでコンパイルした C++ オブジェクトライブラリを 5.0 コンパイラの標準モードで再コンパイルできない。

- 製品をすぐに出荷する必要があるのだが、その製品のソースコードがコンパイラの標準モードではコンパイルできない。

バイナリ互換の問題

「アプリケーションバイナリインタフェース」 (ABI) には、コンパイラによって生成されるオブジェクトプログラムのマシンレベルの特性が定義されています。このマシンレベルの特性とは、基本型のサイズや境界整列条件、構造体型または集合体型の配置、関数の呼び出し方法、プログラムに定義されている構成要素の実名、その他多数の機能を指します。Solaris オペレーティング環境用 C++ ABI の大部分は、C 言語用の ABI である基本 Solaris ABI と同じです。

言語の変更

C++ には、C 言語用 ABI にはない多数の機能 (クラスメンバー関数、多重定義関数と演算子、型保証リンケージ、例外、テンプレートなど) が導入されています。C++ では、主要なバージョンが出るたびに、それまでの ABI では実装することが不可能な言語機能が追加されています。そのため、クラスオブジェクトの配置方法、一部関数の呼び出し方法、型保証リンケージ (「名前の符号化」) の実装方法などの変更が ABI に必要になりました。

C++ 4.0 コンパイラには、ARM に定義されている言語仕様が実装されています。その後、C++ 4.2 コンパイラの発表までの間に、C++ 委員会によって一部の ABI の変更を必要とする多数の新しい言語機能が導入されました。その後の言語に対する機能の追加あるいは変更に伴ってさらに ABI の変更が必要になることは確実なため、C++ の 4.2 時点では、サンは、ABI に対する変更を必要としないものだけを新機能として実装する道を選びました。これは、バージョンの異なるコンパイラでコンパイルした複数のバイナリファイルを維持管理するために必要な、ユーザーの作業をできるかぎり抑えることを意図したものでした。今回のリリースに先立ってC++ 標準が制定されたため、サンは、完全な C++ 言語の実装を可能にする新しい ABI を設計しました。この ABI は、C++ 5.0 コンパイラおよび Sun WorkShop 6 C++ コンパイラでデフォルトとして使用されています。

ABI に影響する言語の変更としては、たとえば、格納を行う関数 `new` と解放を行う関数 `delete` の名前、識別形式、意味の変更があります。識別形式が同じあっても、テンプレート関数と非テープレート関数は、異なる関数であるという新しい規則も、

ABI に影響する言語の変更の 1 つです。そのため「名前の符号化」の変更が必要となり、古いコンパイル済みコードとの非互換性の問題が生まれました。`bool` 型が導入されたことでも、特に標準ライブラリのインタフェースという点で ABI の変更が必要になりました。これらの変更のため、不必要に非効率的な実行時コードの原因となっていた古い ABI のいろいろな部分が改善されています。

新旧バイナリの混在

4.2 コンパイラでコンパイルしたオブジェクトファイルやライブラリは、Sun WorkShop 6 C++ コンパイラでコンパイルしたオブジェクトファイルやライブラリとは絶対にリンクできないというわけではありません。リンクできないのは、ファイルやライブラリが C++ インタフェースを持っている場合です。

C++ でコーディングされているにもかかわらず、外部に対して C インタフェースしか用意されていないライブラリがときどきあります。C インタフェースを持っているということは、インターフェース先は元のプログラムが C++ で作成されていることを知らないということです。もっと具体的に言えば、C インタフェースを持つということは、以下のことがすべて当てはまることを意味します。

- 外部から呼び出されるすべての関数は C リンケージを持ち、パラメータと戻り値に C の型だけを使用する。
- インタフェースのすべての関数へのポインタは C リンケージを持ち、パラメータと戻り値に C の型だけを用する。
- 外部から認識できるすべての型は、C の型である。
- 外部から使用可能なすべてのオブジェクトは、C の型である。
- アプリケーションとライブラリの間で C++ の例外の受け渡しができない。
- `cin`、`cout`、`cerr`、`clog` を使用することはできない。

ライブラリが C インタフェースの条件を満たす場合、そのライブラリは、C ライブラリを使用可能なあらゆる場所で使用できます。つまり、そうしたライブラリのコンパイルと、そのライブラリとリンクするオブジェクトファイルのコンパイルには、異なるバージョンの C++ コンパイラを使用することができます。

ただし、上記の条件の 1 つでも満たされない場合は、ファイルとライブラリをリンクすることはできません。リンクが成功したとしても、プログラムは正しく動作しません。

条件式

C++ 標準では、条件式に関する規則に 1 つの変更が加えられています。この違いは、以下のような式にのみ影響を与えます。

```
e ? a : b = c
```

問題になるのは、グループ化のための括弧がない状態で、コロンの後で代入が行われる場合です。

4.2 コンパイラでは、従来の C++ の規則が使用されており、上記の式は、以下のように書かれているかのように扱われました。

```
(e ? a : b) = c
```

つまり、`c` の値は、`e` の値に従って `a` または `b` のいずれかに代入されます。

このコンパイラでは、互換モードと標準モードのどちらでも、新しい C++ の規則が使用され、上記の式は以下のように書かれているかのように扱われます。

```
e ? a : (b = c)
```

つまり、`c` は `e` が偽の場合にだけ `b` に代入されます。

解決策: 必ず括弧を使用して、自分の意図を正確に指示してください。括弧を使用すると、どのコンパイラでコンパイルしたときでも、コードは同じ意味を持つようになります。

関数ポインタと `void*`

C では、「関数へのポインタ」と `void*` の間の暗黙的な変換は行われません。ARM には、値が入れば関数ポインタと `void*` の間で暗黙的な変換をするという規則が導入されました。この規則は、C++ 4.2 で実装されています。しかし、この暗黙的な変換

は、予測できない関数の多重定義動作の原因となり、コードの移植性を損うため、その後 C++ から削除されました。さらに現在では、関数へのポインタと `void*` の間の暗黙的な変換も、キャストの場合を含め行われません。

このコンパイラは、互換モードでも標準モードでも `+w2` オプションが使用された場合は、関数へのポインタと `void*` の間で暗黙的または明示的な変換が行われると警告を出します。C++ 5.0 は、どちらのモードでも、多重定義された関数呼び出しを解決するときにそのような暗黙的な変換を行いません。4.2 コンパイラに準拠しているそのようなコードは、エラー (一致する関数がない) になります。多重定義を適切に解決するために暗黙的な変換が必要な場合は、キャストを追加する必要があります。たとえば、次のようにします。

```
int g(int);
typedef void (*fptr)();
int f(void*);
int f(fptr);
void foo()
{
    f(g);            // この行は異なる動作になる
}
```

4.2 コンパイラでは、上記のコード例でコメントを付けた行は `f(void*)` を呼び出します。現在では一致する関数がないため、`+w2` オプションを使用するとエラーメッセージが出されます。`f((void*)g)` のような明示的なキャストを追加することもできますが、コードが無効なため警告メッセージが出されます。関数ポインタと `void*` 間の変換は、すべてのバージョンの Solaris オペレーティング環境上で有効です。しかし、すべてのプラットフォームに移植可能というわけではありません。

C++ には、`void*` に対応する「汎用関数ポインタ」はありません。C++ のすべての関数ポインタのサイズと表現は、サポートされるどのプラットフォームでも同じです。したがって、都合のよい任意の関数ポインタ型を使って関数ポインタの値を保持できます。この解決策は、ほとんどのプラットフォームに対して移植性があります。従来と同じように、ポインタ値を使って関数を呼び出す場合は、ポインタ値を元の型に変換する必要があります。37 ページの「extern “C” 関数へのポインタ」を参照してください。

将来の変更について

現在のコンパイラが C++ 標準に準拠していない場合もあります (たとえば、同じエントリを参照する宣言に関する問題)。このような場合、プログラムは正しくリンクされない可能性があります。この問題を回避するには、以下の規則に従ってください。後のリリースでこの問題が修正されても、名前は同じように符号化されます。

- 関数宣言では不必要な `const` キーワードを使用しない。

  値パラメータ `const` の宣言は、関数の識別形式や関数が呼び出される方法に影響するとは想定されていません。したがってそのような影響を持つ値は、`const` であるとは宣言しないでください。

```
int f(const int); // const は意味がないので、このようには宣言しない。
int f(int);       // 代わりにこのように宣言する。
int f(const int i) { ... } // このようには宣言しない。
int f(int i) { ... }       // 代わりにこのように宣言する。
```

- 同じ関数宣言内で、型定義 (`typedef`) した型の元の名前と定義名の両方を使用しない。

```
typedef int int32;
int* foo(int*, int32*); // このように宣言しない。
// 同じ関数宣言内で int* と int32* の両方を使用しない。
// 次のように一貫して宣言する。
int* foo(int*, int*);
int32* foo(int32*, int32*);
```

- 関数へのポインタであるパラメータまたは戻り型には `typedef` だけを使用する。

```
void function( void (*)(), void (*)() ); // このようには宣言しない。
typedef void (*pvf)();
void function( pvf, pvf ); // 代わりにこのように宣言する。
```

- 関数宣言では `const` 配列を使用しない。

```
void function( const int (*)[4] ); // このようには宣言しない。
```

残念ながら、この宣言に対する直接的な回避方法はありません。

この符号化に関する問題の影響をどうしても避けられない場合 (たとえば、自分が所有していないヘッダーやライブラリでこの問題が発生する場合) は、次の例のように弱いシンボルを使用することで、宣言と定義を一致させることができます。

```
extern "C" void __1c_missing_symbol();
#pragma weak __1c_missing_symbol = __1c_existing_symbol
```

このような宣言では、必ず符号化名を使用してください。

第2章

互換モードの使い方

この章では、C++ 4 コンパイラ用に記述されたコードをコンパイルする方法について説明します。

互換モード

次のコンパイラオプションは、ともに互換モードを指示します。

```
-compat
-compat=4
```

例:

```
example% CC -compat -O myfile.cc mylib.a -o myprog
```

以降で説明するように、互換モードにおいては、C++ 4 と Sun WorkShop 6 C++ コンパイラの間に多少の使用方法の違いがあります。

互換モードで有効なキーワード

互換モードでは、新しい C++ キーワードの一部がキーワードとして認識されます。ただし、これらのキーワードの大部分は、次の表に示すようにコンパイラオプションを使用して無効にすることができます。しかし、コンパイラオプションを使用するよりも、ソースコードを変更してこれらのキーワードを使用しないようにすることをお勧めします。

表 2-1　互換モードで有効なキーワード

キーワード	無効にするためのオプション
`explicit`	`-features=no%explicit`
`export`	`-features=no%export`
`mutable`	`-features=no%mutable`
`typename`	なし

`typename` キーワードを無効にすることはできません。表 3-1 に示すその他の新しい C++ キーワードは、互換モードではデフォルトで無効になります。

言語の意味

Sun WorkShop 6 C++ コンパイラでは、C++ 言語の一部の規則について適用機能が強化されています。旧式の構文に対する基準も厳しくなっています。

新しいバージョンの C++ コンパイラをリリースする際に、旧式の構文のサポートを中止するということは、以前からの方針 (マニュアルに記載) でした。しかし、C++ 4 で旧式の構文に関する警告を有効にしてコンパイルすると、かなり前の C++ コンパイラでは受け入れられてきたが、実際には不正であったコードが見つかることがあります。旧式の構文とは主として、アクセス規則 (非公開や限定公開) に違反した構文や、型一致規則に違反した構文、コンパイラが生成した一時変数を参照パラメータの対象として使用した構文などです。

新たに適用されることになった規則は、次のとおりです。

コピーコンストラクタ

オブジェクトを初期化するとき、あるいはクラスの型の値を渡したり返したりするとき、コピーコンストラクタはアクセス可能でなければならない。

```
class T {
    T(const T&);           // 非公開
};
void f1(T t) { }           // エラー、T が渡されない
T f2() { T t; return t; } // エラー、T が返されない
```

解決策: コピーコンストラクタをアクセス可能にしてください。通常は、公開にします。

static 記憶クラス

static 記憶クラスは、型ではなくオブジェクトおよび関数に適用される。

```
static class C {...};   // エラー、ここでは static を使用できない
static class D {...} d; // OK、d は static になる
```

解決策: この例では、クラス C の static キーワードは意味がないので削除してください。

operator new と operator delete

new でオブジェクトを割り当てるときは、対応する operator delete がアクセス可能でなければならない。

```
class T {
    void operator delete(void*); // 非公開
  public:
    void* operator new(size_t);
};
T* t = new T;              // エラー、operator delete にアクセスできない
```

解決策: 演算子 delete をアクセス可能にしてください。通常 delete は公開アクセスであると想定されています。

クラスメンバーでない operator new または operator delete を static 宣言することはできない。

```
static void* operator new(size_t); // エラー
```

解決策: 関数を大域型にしてください。

delete 式にカウントを入れることはできない。

```
delete [5] p; // エラー、delete [] p; を使用すること
```

new const

const オブジェクトを new で割り当てる場合は、オブジェクトを初期化する必要がある。

```
const int* ip1 = new const int;    // エラー
const int* ip2 = new const int(3); // OK
```

条件式

C++ 標準は条件式の規則に変更を導入しました。Sun WorkShop 6 C++ コンパイラは、標準モードと互換モードの両方で新しい規則を使用します。詳細は、6 ページの「条件式」を参照してください。

デフォルトのパラメータ値

多重定義演算子や関数へのポインタにデフォルトのパラメータ値を設定することはできない。

```
T operator+(T t1, T t2 = T(0) );   // エラー
void (*fptr)(int = 3);             // エラー
```

解決策: 他の方法でコーディングする必要があります。一般的には、関数または関数へのポインタ宣言を追加することが考えられます。

main() の戻り値の型

main 関数が返す値の型は、int でなければならない。

```
void main(){ ... } // エラー
```

解決策: main の戻り値の型を int にしてください。return 文を追加する必要はありません。追加した場合、return 文は int 値を返す必要があります。

末尾にコンマを使用する

関数引数リストの末尾にコンマを使用することはできない。

```
f(int i, int j, ){ ... } // エラー
```

解決策: 余分なコンマを削除してください。

const 値またはリテラル値を渡す

const 以外の参照パラメータに const 値またはリテラル値を渡すことはできない。

```
void f(T&);
extern const T t;
void g() {
    f(t);  // エラー
}
```

解決策: 上記の関数によってパラメータが変更されない場合は、const を参照するように const 宣言 (この例では const T&) を変更してください。関数によってパラメータが変更される場合、パラメータに const やリテラル値を渡すことはできません。代りに、const 以外の一時変数を明示的に作成し、それを渡します。詳細は、29 ページの「文字列リテラルと char*」を参照してください。

関数へのポインタと void* 間の変換

C++ コンパイラは互換モードと標準モードのどちらでも、+w2 オプションが指定された場合に、 関数へのポインタと void* 間での暗黙的および明示的変換に対して警告を出します。 コンパイラは、どちらのモードでも、多重定義された関数呼び出しを解釈処理するときには、暗黙的な変換を認識しません。詳細は、6 ページの「関数ポインタと void*」を参照してください。

enum 型

enum 型のオブジェクトを初期化したり、enum 型のオブジェクトに値を代入する場合は、その値も enum 型でなければならない。

```
enum E { one = 1 };
E e1 = 1;        // エラー、int で E を初期化することはできない
E e2 = E(1);     // キャストで解決
```

解決策: キャストを使用してください。

マクロの再定義

#undef を間に入れずにマクロを別の値に再定義することはできない。

```
#define count 1
#define count 2 // エラー
```

解決策: #define 文の一方を削除するか、2 つ目の #define 文の前に #undef count 文を挿入してください。

メンバー初期化リスト

メンバー初期化リスト中に暗黙の基底クラス名を入れる旧式の C++ 構文を使用することはできない。

```
struct B { B(int); };
struct D : B {
    D(int i) : (i) { }     // エラー、B(i) を使用すること
};
```

const と volatile 修飾子

関数に引数を渡すとき、および変数を初期化するとき、ポインタに対する const と volatile 修飾子は正しく対応している必要がある。

```
void f(char *);
const char* p = "hello";
f(p);              // エラー、const ではない char* に const char* を
                      渡すことはできない
```

解決策: ポイント先の文字列が関数によって変更されない場合は、パラメータを const char* で宣言してください。ポイント先の文字列が変更される場合は、文字列の const ではないコピーを作成して、そのコピーを渡してください。

入れ子の型

クラス修飾子がないと、包含するクラスの外側から入れ子の型にアクセスすることはできない。

```
struct Outer {
    struct Inner { int i; };
    int j;
};
Inner x;            // エラー、Outer::Inner を使用すること
```

クラステンプレートの定義と宣言

クラステンプレートの定義と宣言中では、山かっこ < > で囲まれた型引数が付いたクラスの名前は無効でしたが、バージョン 4 とバージョン 5.0 の C++ コンパイラはエラーを報告しませんでした。たとえば、次のコードでは、MyClass に付けられた <T> は定義と宣言のどちらでも無効です。

```
template<class T> class MyClass<T> { ... }; // 定義
template<class T> class MyClass<T>;          // 宣言
```

解決策: 次の例のように、山かっこで囲まれた型引数をクラス名から削除します。

```
template<class T> class MyClass { ... }; // 定義
template<class T> class MyClass;         // 宣言
```

テンプレートコンパイルモデル

互換モードでのテンプレートコンパイルモデルは 4.2 のコンパイルモデルとは異なります。新しいモデルについての詳細は、3-6 ページの 3.3.5 節「テンプレートレポジトリ」を参照してください。

第3章

標準モードの使い方

この章では、Sun WorkShop 6 C++ コンパイラのデフォルトである標準モードの使い方について説明します。

標準モード

標準モードは、C++ コンパイラのデフォルトの動作モードです。このため、標準モードを指示するオプションを指定する必要はありません。指定する場合は、以下のオプションを使用してください。

```
-compat=5
```

例：

```
example% CC -O myfile.cc mylib.a -o myprog
```

標準モードのキーワード

C++ 標準では、新しいキーワードがいくつか追加されています。これらのキーワードを識別子として使用すると、多数の、ときとして意味不明のエラーメッセージが出力されます (プログラマがキーワードを識別子として使用したのかどうかを判断することは非常に困難です。ほとんどの場合、コンパイラのエラーメッセージは役に立ちません)。

次の表に示すように、新しいキーワードの大部分は、コンパイルオプションを使用して無効にできます。論理的に関連のあるオプションは、グループ単位で有効または無効にすることもできます。

表 3-1　標準モードで有効なキーワード

キーワード	無効にするコンパイラオプション
bool、 true、 false	-features=no%bool
explicit	-features=no%explicit
export	-features=no%export
mutable	-features=no%mutable
namespace、using	なし
typename	なし
and、and_eq、bitand、compl、not、not_eq、or、bitor、xor、xor_eq	-features=no%altspell (下記の注を参照)

ISO C 標準の追補には、特殊なトークンを生成するための新しいマクロを定義した C 標準のヘッダー <iso646.h> が導入されています。C++ 標準では、これらの文字列は予約語として定義されています (代替文字列が有効な場合、プログラムに <iso646.h> をインクルードしても何の働きもしません) 。これらのトークンの意味は、次の表に示すとおりです。

表 3-2　トークンとトークン代替文字列

トークン	代替文字列
&&	and
&&=	and_eq
&	bitand
~	compl
!	not
!=	not_eq
\|\| | or |
\| | bitor |
~ | xor |
~= | xor_eq |

テンプレート

C++ 標準にはテンプレートに関する新しい規則がいくつか導入されています。そのため、既存のコードが標準から外れたものになってしまう可能性があります。特に、新しいキーワード `typename` を使用しているコードがこれに該当します。Sun WorkShop 6 C++ コンパイラでは、それらの規則はまだ強制はされていませんが、キーワード自体は認識されます。ほとんどの場合 4.2 コンパイラでは、不正なテンプレートコードが一部受け入れられることになり、4.2 コンパイラで動作していたテンプレートコードは、5.0 コンパイラでもおそらく動作します。将来的には新しい規則が適用されるため、開発スケジュールが許すかぎり、既存のコードは新しい C++ 規則に準拠させてください。

型名の解決

C++ 標準には、識別子が型名であるかどうかを判定するための新しい規則が導入されています。次の例で、それらの規則について説明します。

```
typedef int S;
class B { ... typedef int U; ... }
template< class T > class C : public B {
    S    s; // OK
    T    t; // OK
    U    x; // 1. C++ 標準では無効
    T::V z; // 2. C++ 標準では無効
};
```

新しい言語規則では、テンプレート中の型名を解決するために、基底クラス名が自動的に検索されることはないと規定されています。また、キーワードの `typename` で宣言されていないかぎり、基底クラスやテンプレートパラメータクラスからとられた名前が型名になることはないとも規定されています。

上記の例の最初の無効な行 (1.) では、修飾クラス名とキーワードを使用せずに B から U を型として継承しようとしています。2 行目の無効な行 (2.) では、テンプレートパラメータからとられた型 V が使用されますが、キーワードの `typename` が省略されています。この型が基底クラスやテンプレートパラメータのメンバーに依存することはないため、`s` の定義は有効です。同様に、`t` の定義では、型の `T` (型である必要があるテンプレートパラメータ) がそのまま使用されるため、有効になります。

正しい実装は次のとおりです。

```
typedef int S;
class B { ... typedef int U; ... }
template< class T > class C : public B {
    S             s; // OK
    T             t; // OK
    typename B::U x; // OK
    typename T::V z; // OK
};
```

新しい規則への移行

コードを変更するときに問題になるのは、以前は typename がキーワードではなかったということです。既存のコードで typename を識別子として使用している場合は、まず識別子を別の名前に変更する必要があります。

新旧のコンパイラのどちらでもコードがコンパイルされるようにするには、プロジェクト全体で使用されるヘッダーファイルに次の例に示すような文を追加します。

```
#ifdef TYPENAME_NOT_RECOGNIZED
#define typename
#endif
```

これらの行を追加することにより、条件付きで typename が何ものにも置き換えられなくなります。typename を認識しない古いコンパイラ (Sun C++ 4.2 など) を使用する場合は、メークファイル中のコンパイラオプションに -DTYPENAME_NOT_RECOGNIZED を追加してください。

明示的なインスタンス化と特殊化

ARM と 4.2 コンパイラには、テンプレート定義を使ってテンプレートを明示的にインスタンス化する標準的な方法がありませんでした。C++ 標準と Sun WorkShop 6 C++ コンパイラの標準モードには、テンプレート定義を使って明示的にインスタンス化す

る構文 (キーワード template の後に型を宣言する) が追加されています。たとえば、次のコードの最後の行では、デフォルトのテンプレート定義を使って、クラス MyClass を型 int でインスタンス化しています。

```
template<class T> class MyClass {
    ...
};
template class MyClass<int>; // 明示的なインスタンス化
```

明示的な特殊化の構文は変更されました。特殊化を明示的に宣言したり、全部の定義をする場合は、宣言の前に template<> を付加してください (空の小なり括弧と大なり括弧が必要です)。たとえば、次のようにします。

```
// MyClass の特殊化
class MyClass<char>;          // 古い形式の宣言
class MyClass<char> { ... }; // 古い形式の定義
template<> class MyClass<char>;          // 標準の宣言
template<> class MyClass<char> { ... }; // 標準の定義
```

これらの形式は、引数のテンプレートに対してプログラマが異なる定義 (特殊化) をどこかで行なっていることを意味します。したがって、コンパイラは、これらの引数に対してはデフォルトのテンプレート定義を使用しません。

コンパイラの標準モードは、古い構文も旧式の構文として受け付けます。4.2 コンパイラは、新しい特殊化構文を受け付けますが、新しい構文を使用したコードをいつも正しく処理するとは限りません (この機能が 4.2 コンパイラに組み込まれた後に標準が変更されたため)。テンプレート特殊化コードの移植性を最大限に保つためには、プロジェクトのヘッダーファイルに次の例のような文を追加します。

```
#ifdef OLD_SPECIALIZATION_SYNTAX
#define Specialize
#else
#define Specialize template<>
#endif
```

その上で、たとえば、次のような文を指定します。

```
Specialize class MyClass<char>; // 宣言
```

クラステンプレートの定義と宣言

クラステンプレートの定義と宣言では、山かっこ < > で囲まれた型引数が付いたクラスの名前は無効でしたが、バージョン 4 とバージョン 5.0 の C++ コンパイラはエラーを報告しませんでした。たとえば、次のコードでは、MyClass に付けられた <T> は定義と宣言のどちらでも無効です。

```
template<class T> class MyClass<T> { ... }; // 定義
template<class T> class MyClass<T>;          // 宣言
```

この問題を解決するには、次の例のように山かっこで囲まれた型引数をクラス名から削除します。

```
template<class T> class MyClass { ... }; // 定義
template<class T> class MyClass;         // 宣言
```

テンプレートレポジトリ (テンプレートの格納場所)

サンの C++ テンプレートは、テンプレートインスタンス用のレポジトリ (格納場所) を使用します。C++ 4.2 では、このレポジトリは、Templates.DB というディレクトリに置かれていました。Sun C++ 5.0 コンパイラおよび Sun WorkShop 6 C++ コンパイラでは、デフォルトでは、このディレクトリは SunWS_cache と SunWs_config です。SunWS_cashe には作業ファイルが含まれています。SunWS_config には、構成ファイル、特にテンプレートオプションファイル (SunWS_config/CC_tmp1_opt) が含まれています (『C++ ユーザーズガイド』を参照)。

何らかの理由でレポジトリ用のディレクトリの名前を指定したメークファイルがある場合は、手動で修正する必要があります。また、レポジトリの内部構造は変更されているため、Templates.DB の内容にアクセスするメークファイルを使用することはできなくなっています。

また、標準に従った C++ プログラムではテンプレートが頻繁に使用されるはずです。そのため、複数のプログラムやプロジェクトでディレクトリを共有する場合には注意が必要です。できれば「同じプログラムまたはライブラリに属するファイルは 1 つのディレクトリでコンパイルする」という最も簡単な構成にしてください。これでテンプレートレポジトリは 1 つのプログラムに適用されます。同じディレクトリで別のプログラムをコンパイルする場合は、CCadmin -clean を使用して、レポジトリを事前に整理してください。詳細は、『C++ ユーザーズガイド』を参照してください。

複数のプログラムで同じディレクトリを共用すると、同じ名前に対して異なる定義が必要になる可能性があります。レポジトリを共有した場合、こうした状況に正しく対処することはできません。

テンプレートと標準ライブラリ

C++ の標準ライブラリには、多数のテンプレートと、それらのテンプレートを使用するための多数の新しい標準ヘッダー名が含まれています。サンの C++ の標準ライブラリでは、テンプレートヘッダーに宣言が置かれ、標準ライブラリのテンプレートはそれぞれ別のファイルに置かれています。このため、プロジェクトファイル名に新しいテンプレートヘッダーと同じものがある場合は、誤ったテンプレートファイルが選択され、多数の意味不明のメッセージが出力される可能性があります。たとえば、ユーザーが `vector` というテンプレートを独自に作成していて、標準ライブラリのテンプレートは `vector.cc` というファイルに含まれているとしましょう。ファイルの位置とコマンド行オプションによっては、標準ライブラリの `vector.cc` が必要なときに、ユーザーが作成した `vector.cc` が選択されたり、その逆のことが起きたりする可能性があります。コンパイラの将来のリリースで `export` のようなキーワードが制定され、それを使用するテンプレートが実装された場合この状況はさらに悪くなります。

現在および将来こうした問題が発生するのを防ぐために、以下の 2 つのことをお勧めします。

- 独自のテンプレートファイル名に標準ヘッダー名を使用しない。

  標準ライブラリはすべて名前空間の `std` に含まれているため、ユーザー作成のテンプレートやクラスと直接的に名前の衝突が起こることはありません。しかし、`using` 宣言や指令による間接的な衝突の可能性はあるため、標準ライブラリのテンプレート名と同じ名前は使用しないでください。次に、テンプレートに関係する標準ヘッダーを示します。

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

- 標準ライブラリのテンプレートは、独立したファイルではなくヘッダーファイル (.h) に格納する。こうすることで、標準ライブラリのファイル名の衝突を防ぐことができます。詳細については『C++ ユーザーズガイド』を参照してください。

クラス名の挿入

C++ 標準では、クラスの名前がクラス自身に「挿入」されます。これは、以前の C++ 規則からの変更です。それまでは、クラス名はクラス中に名前としては入っていませんでした。

ほとんどの場合、この微妙な変更が既存のプログラムに影響することはありません。しかし場合によっては、この変更のために、それまで有効だったプログラムが無効になったり、意味が変わったりすることがあります。たとえば、次の場合がそうです。

コード例 3-1　クラス名挿入の問題 1

```
const int X = 5;

class X {
    int i;
public:
    X(int j = X) : // X のデフォルト値は何か?
    i(j) { }
};
```

デフォルトパラメータ値としての X の意味を判定するために、コンパイラは、名前 X を見つけるまで現在のスコープを探し、次にその外のスコープを次々に探します。

- 古い C++ 規則では、クラス X の名前はこのクラススコープにはありません。そのため、ファイルスコープの整数名 X によってクラス名 X が隠されてしまいます。したがって、デフォルト値として 5 が返されます。
- 新しい C++ 規則では、クラス X の名前がこのクラス自身の中にあります。コンパイラはクラスの中で X を見つけ、エラーを出します。コンパイラが見つける X は型名であって、整数値ではないためです。

同じスコープで同じ名前の型とオブジェクトを持つことはプログラミング手法として望ましくないため、このエラーはめったに起こらないはずです。このようなエラーになる場合は、次のように、変数を適切なスコープで修飾してください。

```
X(int j = ::X)
```

次の例は、スコープに関する別の問題です (標準ライブラリのコードを改造したもの)。

コード例 3-2　クラス名挿入の問題 2

```
template class<T> class iterator { ... };

template class<T> class list {
  public:
    class iterator { ... };
    class const_iterator : public ::iterator<T> {
      public:
        const_iterator(const iterator&); // どの反復子か
};
```

const_iterator のコンストラクタに対するパラメータの型は何でしょうか。古い C++ 規則では、コンパイラは、クラス const_iterator のスコープに iterator という名前がないため、次の外側のスコープであるクラス list<T> を探します。次のスコープにはメンバー型として iterator があるため、パラメータの型は list<T>::iterator です。

新しい C++ 規則では、クラスの名前がそれ自身のスコープに挿入されます。具体的には、基底クラスの名前がその基底クラスに挿入されます。コンパイラは、派生クラスのスコープで名前を探し、基底クラスの名前を見つけます。const_iterator コンストラクタに対するパラメータの型にはスコープ修飾子がないため、その名前が const_iterator 基底クラスの名前です。したがって、パラメータの型は、list<T>::iterator ではなく、大域的な ::iterator<T> です。

目的の結果を得るには、いずれかの名前を変更するか、次のようにスコープ修飾子を使用してください。

```
const_iterator(const list<T>::iterator&);
```

for 文中の変数

ARM の規則では、for 文のヘッダーで宣言された変数は、for 文を含むスコープに挿入されると規定していました。しかし、C++ 委員会では、この規則は妥当ではなく、変数のスコープは for 文の終わりで終了すべきであると考えました。また、この規則が当てはまらない場合がいくつかあり、その結果として、コンパイラによって、コードの動作が異なるという事態も生じました。C++ 委員会が for 文中の変数に関する規則を変更したのは、こうした理由によります。ただし、C++ 4.2 コンパイラも含めて、多くのコンパイラでは、引き続き古い規則が採用されています。次の例の if 文は、古い規則では有効ですが、新しい規則では無効になります。これは、k がスコープ外にあるためです。

```
for( int k = 0; k < 10; ++k ) {
    ...
}
if( k == 10 ) ...        // 有効か?
```

互換モードでは、C++ コンパイラはデフォルトで古い規則を適用します。新しい規則の使用をコンパイラに指示するには、-features=localfor コンパイラオプションを使用してください。

標準モードでは、C++ コンパイラはデフォルトで新しい規則を適用します。古い規則の使用をコンパイラに指示するには、-features=no%localfor コンパイラオプションを使用してください。

上記の for 文のヘッダーにある宣言を外に出すと、次の例のように、どのコンパイラのどのモードでも正しく動作するコードを作成することができます。

```
int k;
for( k = 0; k < 10; ++k ) {
    ...
}
if( k == 10 ) ...       // 常に有効なコード
```

関数へのポインタと void* 間の変換

C++ コンパイラは互換モードと標準モードのどちらでも、+w2 オプションが指定された場合に、 関数へのポインタと void* 間での暗黙的および明示的変換に対して警告を出します。 コンパイラは、どちらのモードでも、多重定義された関数呼び出しを解釈処理するときには、暗黙的な変換を認識しません。詳細は、6 ページの「関数ポインタと void*」を参照してください。

文字列リテラルと char*

標準 C++では文字列リテラルは const char[] 型として扱われ、char* と宣言された関数パラメータは文字列リテラルには渡されません。この変更の経緯を順を追って説明します。標準の C では、const キーワードと定数オブジェクトの概念が導入されました。これらのどちらも従来の C 言語 (K&R 形式の C) にはなかったものです。次の例に見られるような無意味な結果が出されないようにするには、論理的には「Hello world」などの文字列リテラルは const で宣言するべきです。

```
#define GREETING "Hello world";
char* greet = GREETING; // コンパイラからのエラー出力はない
greet[0] = 'J';
printf("%s", GREETING); // システムによっては「Gello world」と出力される
```

C、C++ とも、文字列リテラルを変更した結果がどうなるかは未定義です。同じ文字列リテラルに対して同じ書き込み可能記憶域を使用する実装の場合は、上の例のように奇妙な出力となります。

当時存在していたコードの多くが上記の例の 2 行目のようになっていたため、1989 年に C 標準委員会は文字列リテラルを const にはしませんでした。そのため、C++ 言語は当初 C 言語の規則に従いました。しかし後日、C++ 標準委員会は、C++ においては型の安全性が重要と判断し、この文字列リテラルに関する規則を変更しました。

標準の C++ では、文字列リテラルは定数であり、const char[] 型です。上記の例の 2 行目は標準の C++ では無効です。同じように、char* で宣言した関数パラメータは、文字列リテラルとして渡すべきではありません。ところが C++ 標準では、文字列リテラル const char[] から char* への変換は不適切であると規定されています。この例をいくつか示します。

```
char *p1 = "Hello";   // 従来は問題なかったが、現在は不適切
const char* p2 = "Hello"; // OK
void f(char*);
f(p1);       // p1 は const として宣言されていないので常に OK
f(p2);       // エラー、const char* を char* に渡している
f("Hello"); // 従来は問題なかったが、現在は不適切
void g(const char*);
g(p1);       // 常に OK
g(p2);       // 常に OK
g("Hello"); // 常に OK
```

引数として渡された文字配列が直接的にも間接的にも関数によって変更されることがない場合は、パラメータを const char* または const char[] と宣言してください。このようにすると、プログラムのいたるところで const 修飾子を追加する必要があることに気づくでしょう。修飾子を追加するほど、さらに多くの修飾子が必要になります (「const 中毒 (const poisoning)」と呼ばれることがある現象)。

標準モードのコンパイラは、文字列リテラルから char* への変換が適切でないと警告を出します。妥当と思われるあらゆる場所に const を使用していれば、既存のプログラムは新しい規則でもおそらく変更なしにコンパイルされます。

関数を多重定義するために、標準モードでは、文字列リテラルは常に const とみなされます。

```
void f(char*);
void f(const char*);
f("Hello"); // どの f が呼び出されるか
```

上の例を互換モード (または 4.2 コンパイラ) でコンパイルすると、関数 f(char*) が呼び出されます。

標準モードでは、コンパイラは、リテラル文字列をデフォルトで読み取り専用記憶域に置きます。この文字列を変更しようとすると (char* への自動変換によって変更されることがある)、プログラムはメモリー違反で異常終了します。

次の例では、標準モードの C++ コンパイラも 4.2 コンパイラと同様に、文字列リテラルを書き込み可能記憶域に置きます。プログラムは動作しますが、技術的にはその動作がどうなるかは未定義です。標準モードのコンパイラは文字列リテラルをデフォルトで読み取り専用記憶域に置くため、プログラムはメモリー違反で異常終了します。そのため、文字列リテラルの変換に対するすべての警告に注意し、変換が起こらないようにプログラムを修正する必要があります。そうすれば、プログラムはどの C++ 実装でも正しく動作します。

```
void f(char* p) { p[0] = 'J'; }

int main()
{
    f("Hello"); // const char[] から char* への変換
}
```

コンパイラの動作は、コンパイラオプションを使って変更できます。

- `-features=conststrings` コンパイラオプションを指定すると、コンパイラは、互換モードでも文字列リテラルを読み取り専用記憶域に置きます。
- `-features=no%conststrings` コンパイラオプションを指定すると、コンパイラは、標準モードでも文字列リテラルを書き込み可能記憶域に置きます。

C 形式の文字列ではなく標準の C++ の `string` クラスを使用した方が便利なこともあります。標準の C++ の `string` オブジェクトは個別に `const` かどうか宣言したり、参照、ポインタ、値のどれによっても関数に渡せるため、`string` クラスには文字列リテラルに関係する問題はありません。

条件式

C++ 標準は条件式の規則に変更を導入しました。Sun WorkShop 6 C++ コンパイラは、標準モードと互換モードの両方で新しい規則を使用します。詳細は、6 ページの「条件式」を参照してください。

新しい形式の new と delete

新しい形式の new と delete については、次の注意事項があります。

- 配列の形式
- 例外の指定
- 置き換え関数
- ヘッダーファイル

互換モードでは、デフォルトで古い規則が適用されます。標準モードでは、デフォルトで新しい規則が適用されます。古い実行時ライブラリ (libC.so) は古い定義と動作に依存し、新しい標準ライブラリ (libCstd.so) は新しい定義と動作に依存するため、デフォルトを変更することはお勧めできません。

新しい規則を適用した場合、コンパイラは事前に _ARRAYNEW マクロを 1 に定義します。古い規則を適用した場合、このマクロは定義されません。次の使用例を参照してください。この意味については、次の節で詳しく説明します。

```
// 置き換え関数
#ifdef _ARRAYNEW
    void* operator new(size_t) throw(std::bad_alloc);
    void* operator new[](size_t) throw(std::bad_alloc);
#else
    void* operator new(size_t);
#endif
```

new と delete の配列形式

C++ 標準では、配列の割り当てあるいは割り当て解除を行うときに呼び出される operator new と operator delete の新しい形式が追加されています。従来は、これらの operator 関数は 1 つの形式しかありませんでした。また、配列の割り当てでは、大域形式の operator new と operator delete が使用され、クラス固有の形式は使用されませんでした。新しい形式を使用するには、ABI の変更が必要になるため、C++ 4.2 コンパイラでは、新しい形式はサポートされていません。

次の関数に加えて、

```
void* operator new(size_t);
void operator delete(void*);
```

C++ 標準では、以下の関数が追加されています。

```
void* operator new[](size_t);
void operator delete[](void*);
```

新旧いずれの場合も、実行時ライブラリにある形式とは別の形式を記述することができます。このように 2 つの形式が用意されているのは、配列と個々のオブジェクトに対して異なるメモリープールを使用できるようにするためと、配列に対してクラスが独自の形式の `operator new` を提供できるようにするためです。

新旧どちらの規則でも、`new T` と記述すると (`T` は特定の型)、`operator new(size_t)` 関数が呼び出されます。ただし、新しい規則で `new T[n]` と記述すると、`operator new[](size_t)` 関数が呼び出されます。

同様にどちらの規則でも `delete p` と記述すると、`operator delete(void*)` が呼び出されます。ただし、新しい規則で `delete [] p;` と記述すると、`operator delete[](void*)` が呼び出されます。

これらの関数について、クラス固有の配列形式を記述することもできます。

例外の指定

古い規則では、割り当てに失敗すると、どの形式の `operator new` でも NULL ポインタ を返します。新しい規則では、割り当てに失敗すると、通常の形式の `operator new` では例外を送出し、値は返しません。このほか、例外を送出する代わりにゼロを返す特殊な形式の `operator new` もあります。どの形式の `operator new` および `operator delete` にも、「例外指定」があります。次は、標準ヘッダーの `<new>` にある宣言です。

コード例 3-3　標準ヘッダー <new>

```
namespace std {
    class bad_alloc;
    struct nothrow_t {};
    extern const nothrow_t nothrow;
}
// 単一オブジェクト形式
void* operator new(size_t size) throw(std::bad_alloc);
void* operator new(size_t size, const std::nothrow_t&) throw();
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
// 配列形式
void* operator new[](size_t size) throw(std::bad_alloc);
void* operator new[](size_t size, const std::nothrow_t&) throw();
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

次の例に示すような安全対策のためのコードは、新しい規則では意図したとおりには動作しません。割り当てに失敗すると、new 式から自動的に呼び出される operator new によって例外が送出され、ゼロを判定する検査は行われません。

```
T* p = new T;
if( p == 0 ) {              // 新しい規則ではエラー
    ...                     // 割り当て失敗の処理
}
...                         // p を使用する
```

このような場合には、次の 2 つの方法で解決できます。

- 以下のように、コードを記述し直して例外を捕獲できるようにする。

```
T* p = 0;
try {
    p = new T;
}
catch( std::bad_alloc& ) {
    ...         // 割り当て失敗の処理
}
...             // p を使用する
```

- 以下のように nothrow 形式の operator new を使用する。

```
T* p = new (std::nothrow) T;
... 以降は元のコードと同じ
```

コード中で例外を使用したくない場合は、2 番目の形式を使用してください。コード中で例外を使用するときは、最初の形式をお勧めします。

operator new が成功するかどうかを確認していない場合は、既存のコードを変更せずにそのまま使用してもかまいません。不正なメモリー参照が発生する箇所まで処理が進むことはなく、プログラムは割り当てに失敗した時点で異常終了します。

置き換え関数

別の形式の operator new と operator delete を使用している場合、その関数は、例外の指定を含めてコード例 3-3 と同じ識別形式である必要があります。また、実装されている意味も同じである必要があります。通常の形式の operator new では、失敗時に bad_alloc 例外を送出する必要があります。これに対して nothrow 形式では、失敗時に例外を送出せずに、ゼロを返す必要があります。operator delete では、どの形式についても、例外を送出してはいけません。標準ライブラリのコードでは、大域的な operator new と operator delete が使用されており、コードが正しく実行されるかどうかは、その動作に依存します。他社のライブラリについても、同様の依存関係が存在する可能性があります。

Sun WorkShop 6 C++ の実行時ライブラリの大域形式の operator new[]() は、C++ 標準で規定されているように、単一オブジェクト形式の operator new() を呼び出すだけです。Sun WorkShop 6 C++ の標準ライブラリの大域形式の operator new() を置き換える場合、大域形式の operator new[]() を置き換える必要はありません。

C++ 標準では、あらかじめ定義されている、以下の「配置」形式の operator new の置き換えを禁止しています。

```
void* operator new(std::size_t, void*) throw();
void* operator new[](std::size_t, void*) throw();
```

上記の置き換えは 4.2 コンパイラでは許可されますが、標準モードでは置換できません。4.2コンパイラでは、別のパラメータリストを使用して独自の置き換えを記述することもできます。

インクルードするヘッダー

互換モードの場合は、通常どおり `<new.h>` をインクルードしてください。標準モードでは、代わりに `<new>` (`.h` なし) をインクルードしてください。簡単に移行できるよう、標準モードでは、ヘッダーの `<new.h>` を使用すると、名前空間 `std` の名前を大域の名前空間が使用できます。このヘッダーには、例外の古い名前を新しい名前に対応させる `typedef` も用意されています。

ブール型

ブール型 (`bool`、`true`、`false`) は、コンパイラで `bool` キーワードの認識が有効になっているかどうかによって制御されます。

- 互換モードでは、`bool` キーワードの認識はデフォルトで無効です。`bool` キーワードの認識を有効にするには、コンパイラオプション `-features=bool` を使用します。
- 標準モードでは、`bool` キーワードの認識はデフォルトで有効です。`bool` キーワードの認識を無効にするには、コンパイラオプション `-features=no%bool` を使用します。

互換モードでは、キーワードを有効にすることをお勧めします。これは、コード中でキーワードが現在どのように使用されているか明らかになるためです。

注 – 既存のコードで使用されているブール型の定義に互換性があるとしても、実際の型が異なるため、名前の符号化に影響が生じます。その場合は、関数のパラメータにブール型を使用して、古いコードをすべて再コンパイルする必要があります)。

標準モードで `bool` キーワードを無効にすることは、お勧めしません。これは、C++ の標準ライブラリが、 `bool` 型に依存しているためです。後で `bool` を有効にすると、名前の符号化などのことで、さらに問題が生じます。

bool キーワードが有効な場合、コンパイラは、あらかじめ _BOOL マクロを 1 に定義します。キーワードが無効な場合、このマクロは定義されません。次に例を示します。

```
// 互換性のあるブール型の定義
#if !defined(_BOOL) && !defined(BOOL_TYPE)
    #define BOOL_TYPE            // 局所インクルード対策
   typedef unsigned char bool; // 標準モードでは、bool は 1 バイトを使用
    const bool true = 1;
    const bool false = 0;
#endif
```

互換モードでは、新しい組み込み型の bool 型とまったく同じように動作するブール型を定義することはできません。組み込み型の bool 型が C++ に追加されているのは、このためです。

extern "C" 関数へのポインタ

関数は、次のような言語リンケージによって宣言できます。

```
extern "C" int f1(int);
```

リンケージを指定しないと、C++ のリンケージが使用されます。C++ リンケージは、明示的に指定することもできます。

```
extern "C++" int f2(int);
```

複数の宣言をグループにまとめることもできます。

```
  extern "C" {
        int g1(); // C リンケージ
        int g2(); // C リンケージ
        int g3(); // C リンケージ
    } // セミコロンなし
```

この手法は、標準ヘッダーでも幅広く使用されています。

言語リンケージ

「言語リンケージ」とは、関数の呼び出しに関する方法を意味します。たとえば、引数の場所、戻り値の検出場所の指定などがこれに当たります。言語リンケージを宣言するということは、その言語で関数が記述されないという意味です。言語リンケージを宣言すると、指定した言語で記述されているかのように関数を呼び出すことができます。つまり、C++ 関数が C リンケージを持つように宣言するとは、C 言語で記述された関数から C++ 関数を呼び出せるようにするということです。

関数の宣言に適用された言語リンケージは、戻り値型、および関数または関数へのポインタを持つすべてのパラメータに適用されます。

互換モードでは、言語リンケージは関数の型の構成要素ではないという、ARM の規則が実装されています。特に、ポインタのリンケージや割り当てられた関数とは無関係に、関数へのポインタを宣言することができます。これは、C++ 4.2 コンパイラと同じ動作です。

標準モードでは、言語リンケージはその関数の型の構成要素であり、かつ、関数へのポインタの型の構成要素であるという新しい規則が実装されています。このため、リンケージは関数とポインタとの間で一致していなければなりません。

次の例は、C リンケージと C++ リンケージを持つ関数および関数へのポインタの組み合わせとして考えられる 4 つの場合すべてを表しています。互換モードでは、コンパイラは 4.2 コンパイラと同じように、あらゆる組み合わせを受け入れます。標準モードのコンパイラでは、一致していない組み合わせは旧式とみなされます。

```
extern "C" int fc(int) { return 1; }      // fc の C リンケージ
int fcpp(int) { return 1; }               // fcpp の C++ リンケージ
// fp1 と fp2 の C++ リンケージ
int (*fp1)(int) = fc;                     //不一致
int (*fp2)(int) = fcpp;                   // OK
// fp3 と fp4 の C リンケージ
extern "C" int (*fp3)(int) = fc;          // OK
extern "C" int (*fp4)(int) = fcpp;        //不一致
```

リンケージに関連して問題が発生した場合は、C リンケージ関数と組み合わせ可能なポインタが C リンケージで宣言され、C++ リンケージ関数と組み合わせられるポインタにリンケージ指定がないか、または、C++ リンケージで宣言されていることを確認してください。

```
extern "C" {
    int fc(int);
    int (*fp1)(int) = fc; // どちらも C リンケージを持つ
}
int fcpp(int);
int (*fp2)(int) = fcpp;   // どちらも C++ リンケージを持つ
```

ポインタと関数が一致しない場合は、関数を包含するコード (ラッパー) を記述することによって、コンパイラのエラーを回避することができます (Solaris では、C と C++ の関数リンケージは同じですが、一般的には、リンケージが一致していないとエラーになります。これは新しい言語規則として適用されているためです)。

次の例では、 `composer` は、C リンケージを持つ関数へのポインタをとる C 関数です。

```
extern "C" void composer( int(*)(int) );
extern "C++" int foo(int);
composer( foo ); // 不一致
```

関数 `foo` (C++ リンケージを持つ) を 関数 `composer` に渡すには、次のように `foo` に C インタフェースを提供する `foo_wrapper` という C リンケージ関数を作成します。

```
extern "C" void composer( int(*)(int) );
extern "C++" int foo(int);
extern "C" int foo_wrapper(int i) { return foo(i); }
composer( foo_wrapper ); // OK
```

この手法は、コンパイラのエラーを回避するためだけでなく、C と C++ の関数が実際には異なるリンケージを持っている場合にも使用できます。

移植性の低い解決策

サンの実装している C と C++ の関数リンケージはバイナリ互換です。すべての C++ の実装がこうなっているわけではありませんが、比較的共通のことです。互換性がなくなってもかまわないのであれば、キャストを使って C++ リンケージ関数を C リンケージ関数と同じように使用できます。

たとえば静的メンバー関数がよい例です。リンケージに関する C++ 言語の新しい規則が関数の型の一部となるまでは、クラスの静的メンバー関数を C リンケージを持つ関数として扱うのが一般的でした。これによって、クラスメンバー関数のリンケージを宣言できないという制限を回避していました。たとえば、次の例を考えてみましょう。

```
// 既存のコード
typedef int (*cfuncptr)(int);
extern "C" void set_callback(cfuncptr);
class T {
    ...
    static int memfunc(int);
};
...
set_callback(T::memfunc); // 新しい規則では無効
```

上記の問題を解決するには、前の項でお勧めしたように `T::memfunc` を呼び出す関数ラッパーを作成してから、すべての `set_callback` 呼び出しを変更して `T::memfunc` の代りにラッパーを使用します。こうすると、完全な移植性を持つ正しいコードになります。

もう 1 つの解決策として、次の例のように多重定義した set_callback 呼び出しを作成して、C++ リンケージを持つ関数を受け取り、元の関数を呼び出すこともできます。

```
// 変更したコード
extern "C" {
    typedef int (*cfuncptr)(int); // C 関数へのポインタ
    void set_callback(cfuncptr);
}
typedef int (*cppfuncptr)(int); // C++ 関数へのポインタ
inline void set_callback(cppfuncptr f) // 多重定義したもの
    { set_callback((cfuncptr)f); }
class T {
    ...
    static int memfunc(int);
};
...
set_callback(T::memfunc); // 元のコードと同じ
```

この例では、既存のコードをわずかに変更しただけです。ここには、コールバックを設定する set_callback を新たに追加しました。既存のコードは元の set_callback を呼び出していましたが、ここでは多重定義したものを呼び出し、それが元のものを呼び出します。多重定義したものはインライン関数なので、実行時のオーバーヘッドはまったくありません。

この方法は Sun C++ では動作しますが、すべての C++ の実装で動作するとは限りません。これは、他のシステムでは C 関数と C++ 関数の呼び出し順序が異なる場合があるためです。

関数のパラメータとしての関数へのポインタ

言語リンケージに関する新しい規則の追加に伴う微妙な問題があります。それは、上記の例の composer 関数のような、パラメータとして関数へのポインタをとる関数の問題です。

```
extern "C" void composer( int(*)(int) );
```

言語リンケージに関する規則のうち、変更されていない規則として、言語リンケージを持つ関数が宣言されていて、その後に「同じ関数」が言語リンケージなしで定義されている場合は、前の言語リンケージが適用されるという規則があります。

```
extern "C" int f(int);
int f(int i) { ... } // "C" リンケージを持つ
```

上記の関数 f は C リンケージを持ちます。この宣言 (インクルードされるヘッダーファイルに含まれている可能性もある) の後の定義は、リンケージ指定を継承します。しかし、次の例に示すように、この関数が関数へのポインタ型のパラメータをとる場合はどうなるのでしょう。

```
extern "C" int g( int(*)(int) );
int g( int(*pf)(int) ) { ... } // "C" または "C++" リンケージのどちらか?
```

古い規則と 4.2 コンパイラでは、このコードには、g という関数が 1 つ存在するだけです。新しい規則では、1 行目は、C リンケージを持つ関数へのポインタをとる、C リンケージを持つ関数 g を宣言し、2 行目は、C++ リンケージを持つ関数へのポインタをとる関数を定義していることになります。2 つの関数は同じではありません。2 つ目の関数は C++ リンケージを持ちます。リンケージは関数へのポインタの型の構成要素であるため、2 つの行は、それぞれが g という名前の多重定義関数を参照します。このため、これらの関数が同じ関数であることに依存するコードは、問題になります。コンパイルまたはリンクが失敗する可能性が非常に高くなります。

プログラミングするときの習慣として、リンケージは、宣言だけでなく関数の定義でも指定するようにしてください。

```
extern "C" int g( int(*)(int) );
extern "C" int g( int(*pf)(int) ) { ... }
```

型に関する混乱は、関数パラメータに typedef を使用することでさらに少なくすることができます。

```
extern "C" typedef int (*pfc)(int); // C リンケージ関数へのポインタ
extern "C" int g(pfc);
extern "C" int g(pfc pf) { ... }
```

実行時の型識別 (RTTI)

4.2 コンパイラと同様に、5.0 の互換モードでは実行時の型識別 (RTTI) はデフォルトで無効です。標準モードでは、RTTI は有効であり、無効にすることはできません。古い ABI では RTTI を有効にすると、データのサイズ、および実行効率の面で著しい負担がかかっていました (古い ABI では、RTTI を直接に実装することができず、非効率的な間接的な方法をとる必要があったためです)。標準モードでは、新しい ABI を使用することにより、この負担は無視できるほどになっています (これは ABI で改善された機能の 1 つです)。

標準の例外

C++ 4.2 コンパイラには、C++ 標準の草案段階で提案されていた標準例外に関連する名前が、例外名として採用されています。それ以降、C++ 標準では、例外名が変更されてきました。C++ 5.0 コンパイラおよび Sun WorkShop 6 C++ コンパイラ両方の標準モードでは標準の例外の名前として、次の表に示す名前が使用されています。

表 3-3　例外関連の型名

古い名前	標準名	説明
`xmsg`	`exception`	標準例外の基底クラス
`xalloc`	`bad_alloc`	割り当て要求の失敗で送出
`terminate_function`	`terminate_handler`	終了ハンドラ関数の型
`unexpected_function`	`unexpected_handler`	予期しない例外ハンドラ関数の型

クラスの使用方法が異なるように、こられクラスの公開メンバー (`xmsg` と `exception` および `xalloc` と `bad_alloc`) の使用方法は異なります。

静的オブジェクトの破棄の順序

「静的オブジェクト」とは、静的な記憶期間を持つオブジェクトのことです。静的オブジェクトは、大域オブジェクトでも名前空間中のオブジェクトでもかまいません。また、関数に局所的な静的変数でも、クラスの静的なデータメンバーでもかまいません。

C++ 標準は、静的オブジェクトの破棄は構築時とは逆の順序で行われるべきであると規定しています。さらに、`atexit()` 関数で登録された関数の破棄との兼ね合いについても規定しています。

以前のバージョンの Sun WorkShop C++ コンパイラは、1 つのモジュールで生成された大域静的オブジェクトを、生成時とは逆の順序で破棄していました。しかし、プログラム全体に渡って正しい順序で破棄されるかどうかは確約されてはいませんでした。

WorkShop 6 C++ コンパイラ以降、静的オブジェクトは、必ず構築されたときと逆の順序で破棄されます。たとえば、次のような型 `T` の静的オブジェクトが 3 つあると仮定します。

- 1 つ目のオブジェクトは `file1` 内の大域スコープにある。
- 2 つ目のオブジェクトは `file2` 内の大域スコープにある。
- 3 つ目のオブジェクトは関数内の局所スコープにある。

`file1` と `file2` にある 2 つの大域オブジェクトのどちらが最初に生成されるかどうかはわかりません。しかし、最初に生成された方の大域オブジェクトは、他方の大域オブジェクトが破棄された後に破棄されます。

局所静的オブジェクトは、その関数が呼び出されたときに生成されます。両方の大域静的オブジェクトが生成された後に関数が呼び出された場合、局所オブジェクトは、両方の大域オブジェクトが破棄される前に破棄されます。

C++ 標準は、`atexit()` 関数で登録された関数と静的オブジェクトの破棄との関連性について規定を追加しました。つまり、静的オブジェクト `X` が生成された後に関数 `F` が `atexit()` で登録された場合、`F` は、`X` が破棄される前に、プログラムの終了時に呼び出される必要があります。逆に言うと、`X` が生成される前に関数 `F` が `atexit()` で登録された場合、`F` は、`X` が破棄された後に、プログラムの終了時に呼び出される必要があります。

次に、この規則の例を示します。

```
// T はデストラクタを持つ型。
void bar();
void foo()
{
  static T t2;
  atexit(bar);
  static T t3;
}
T t1;
int main()
{
  foo();
}
```

プログラムの開始時には、t1 が生成され、その後で main が実行されます。main は foo() を呼び出します。foo() 関数は、次の作業をこの順序どおりに実行します。

1. t2 を生成する。
2. atexit() で bar() を登録する。
3. t3 を生成する。

main の終了時には、exit が自動的に呼び出されます。終了手順は次の順序どおりに行われる必要があります。

1. t3 を破棄する (t3 は bar() が atexit() で登録された後に生成された)。
2. bar() を実行する。
3. t2 を破棄する (t2 は bar() が atexit() で登録される前に生成された)。
4. t1 を破棄する。t1 は最初に生成されたため、最後に破棄される。

このように静的デストラクタと atexit() 処理を交互に行うには、Solaris 実行時ライブラリ libc.so が必要です。この処理は Solaris 8 から実行できます。WorkShop 6 でコンパイルされた C++ プログラムは、実行時にライブラリ中で特別なシンボルを検索し、そのシンボルの有無から現在プログラムが動作しているバージョンの Solaris で上記の処理が実行できるかどうかを判断します。シンボルが存在する場合、静的デストラクタと atexit() で登録された関数は交互に処理されます。シンボルが存在しない場合、デストラクタは適切な順序で実行されますが、atexit() で登録された関数と関連付けて実行されることはありません。

この判断はプログラムが実行されるたびに行われます。プログラムが構築されたバージョンの Solaris は問題ではありません。現在プログラムが動作しているバージョンの Solaris が上記の処理をサポートしていて、Solaris 実行時ライブラリ `libc.so` が動的にリンクされていれば (デフォルトではリンクされる)、プログラム終了時に `atexit()` で登録された関数が実行されます。

静的オブジェクトの破棄の順序をどれだけ正しくサポートできるかは、コンパイラによって異なります。コードの移植性を向上させるには、静的オブジェクトが破棄される順序に影響を受けないようにプログラムを作成します。

プログラム中に破棄の順序に依存するコードがあり、かつ、古いコンパイラで作業する必要がある場合、標準モードでは、C++ 標準の規定によってプログラムが破壊されてしまう可能性があります。`-features=no%strictdestrorder` コマンドオプションを使用すると、厳密な破棄の順序を無効にできます。

第4章

入出力ストリームとライブラリヘッダーの使い方

この章では、C++ 5.0 コンパイラに実装されたライブラリとヘッダーファイルの変更について説明します。この変更は、C++ 4 コンパイラ用に記述したコードを Sun WorkShop 6 C++ コンパイラ用に変更する際に考慮する必要があります。

入出力ストリーム

C++ 4.2 コンパイラには、これまで正式な定義ではなかった「従来型」の入出力ストリームが実装されています。この実装形式は、`Cfront` (1990 年) とともにリリースされたバージョンと互換性があり、いくつかの問題点が解決されています。

標準 C++ では、新しい入出力ストリームと拡張された入出力ストリーム (標準入出力ストリーム) が定義されています。新しい入出力ストリームは、綿密に定義されており、機能が豊富で、多言語化対応のコードの記述に利用できます。

互換モードでは、C++ 4.2 コンパイラと同じ従来型の入出力ストリームが提供されます。互換モードでコンパイルしたとき (`-compat[=4]`)、4.2 コンパイラで動作していた既存の入出力ストリームコードはまったく同じように動作します。

注 – コンパイラが提供する従来型の入出力ストリーム実行時ライブラリには 2 つのバージョンがあります。一方のバージョンはコンパイラの互換モードでコンパイルされるもので、C++ 4.2 で使用されるライブラリと同じです。もう一方のバージョンは、ソースコードは同じですが、コンパイラの標準モードでコンパイルされます。ソースコードのインタフェースは同じですが、ライブラリのバイナリコードには標準モードの ABI が使用されています。詳細については 4 ページの「バイナリ互換の問題」を参照してください。

標準モードでは、デフォルトで標準の入出力ストリームが使用されます。標準形式のヘッダー名 (「.h」なし) を使用すると、標準ヘッダーが使用されます。その場合、すべての宣言は名前空間 std にあります。

標準ヘッダーには「.h」で終わる形式が用意されているものもあります。次の 4 ファイルがそうです。これを使用すると、すべてのヘッダー名が using 宣言によって大域名前空間に存在するようになります。

- <fstream.h>
- <iomanip.h>
- <iostream.h>
- <strstream.h>

これらのヘッダーはサンの拡張であるため、これに依存するコードは移植性がない可能性があります。これらのヘッダーを使用すると、従来の入出力ストリームの代りに標準の入出力ストリームを使用している場合でも、既存の (簡単な) 入出力ストリームコードを変更せずにコンパイルできます。たとえば、表 4-2 のコードは、従来の入出力ストリームでもサンの標準入出力ストリームの実装でもコンパイルできます。

表 4-1 標準の名前形式の iostream を使用

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
}
```

表 4-2 従来の名前形式の iostream を使用

```
#include <iostream.h>
int main()
{
    cout << "Hello, world!" << endl;
}
```

すべての従来の入出力ストリームコードが標準入出力ストリームと互換性があるとは限りません。従来の入出力ストリームコードをコンパイルできない場合は、コードを変更するか、従来の入出力ストリームだけを使用する必要があります。

従来の入出力ストリームを標準モードで使用する場合は、コンパイラオプション `-library=iostream` を `CC` コマンド行に指定します。このオプションを使用すると、従来の入出力ストリームのヘッダーファイルが入った特別なディレクトリが探索され、従来の入出力ストリーム実行時ライブラリがプログラムとリンクされます。このオプションは、プログラムに必要なすべてのコンパイルだけでなく、最後のリンク段階でも使用する必要があります。そうしないと、一貫性のない結果となります。

注 – 古い形式と新しい形式の入出力ストリーム (標準の入力ストリームと出力ストリーム `cin`、`cout`、`cerr` を含む) が同じプログラムに混在していると、重大な問題が発生することがあるのでお勧めできません。

従来の入出力ストリームを使用すると、入出力ストリームヘッダーの 1 つをインクルードする代わりに、入出力ストリームクラス用の独自の前方宣言を作成できます。

表 4-3 従来の入出力ストリームによる前方宣言

```
// 従来の入出力ストリームでのみ有効
#include <iosfwd>
using std::istream;
using std::ostream;
class MyClass;
istream& operator>>(istream&, MyClass);
ostream& operator<<(ostream&, MyClass);
```

従来の名前 (`istream`、`ofstream`、`streambuf` など) が標準の入出力ストリームのクラスの名前と同じでないため、この方法は標準の入出力ストリームには適用できません。従来の名前はクラステンプレートの特殊化を型定義 (`typedef`) したものです。

標準入出力ストリームを使用すると、入出力ストリームクラスに対して独自の前方宣言は作成できません。正しく前方宣言が行われるようにするには、代わりに標準ヘッダー <iosfwd> をインクルードします。

表 4-4 標準の入出力ストリームによる前方宣言

```
// 標準の入出力ストリームにのみ有効
#include <iosfwd>
using std::istream;
using std::ostream;
class MyClass;
istream& operator>>(istream&, MyClass);
ostream& operator<<(ostream&, MyClass);
```

標準型と従来型の両方の入出力ストリームで機能するコードを作成するには、前方宣言を使用する代わりに、ヘッダーファイル全体をインクルードします。次に例を示します。

表 4-5 従来型と標準型の両方の入出力ストリームに有効なコード

```
// Sun WorkShop C++ の従来型と標準型の両方の入出力ストリームで有効
#include <iostream.h>
class MyClass;
istream& operator>>(istream&, MyClass);
ostream& operator<<(ostream&, MyClass);
```

タスク (コルーチン) ライブラリ

<task.h> ヘッダーを介してアクセスするコルーチンライブラリはサポートされていません。コルーチンライブラリに比べて、Solaris のスレッドと、言語開発ツール (特にデバッガ) およびオペレーティングシステムとの間の統合が改善されています。

RogueWave Tools.h++

Sun WorkShop 6 C++ コンパイラには Rogue Wave の Tools.h++ バージョン 7 が含まれています。RogueWave Tools.h++ バージョン 7 ライブラリは、従来の入出力ストリームで構築されています。したがって、RogueWave のツールライブラリを標準

モードでインクルードする場合は、入出力ストリームもインクルードする必要があります。古い形式と新しい形式の入出力ストリームを同じプログラムで使用しないように注意が必要です。`Tools.h++` を標準モードで使用するときは、プログラム全体に含まれるすべての部分をコンパイルおよびリンクする際には、`-library=iostream` オプションを使用してください。

- RogueWave `Tools.h++` ライブラリを標準モードで使用する場合は、次のコンパイラオプションを使用します。

  ```
  -library=rwtools7,iostream
  ```

- RogueWave `Tools.h++` ライブラリを互換モードで使用する場合は、次のコンパイラオプションを使用します。

  ```
  -compat  -library=rwtools7
  ```

`Tools.h++` へのアクセス方法の詳細については、『C++ ユーザーズガイド』または `CC`(1) マニュアルページを参照してください。

C ライブラリヘッダー

互換モードでは、従来どおり C のヘッダーを使用できます。標準ヘッダーは、 使用中のリリースの Solaris に含まれる `/usr/include` ディレクトリにあります。

C++ 標準では、C のヘッダーの定義が変更されています。

ここで説明する C ライブラリヘッダーとは、ISO C 標準 (1990 年の ISO 9899) と、それ以降の追補 (1994 年) で定義されている以下の 17 のヘッダーです。

```
<assert.h> <ctype.h>  <errno.h>  <float.h>  <iso646.h> <limits.h>
<locale.h> <math.h>   <setjmp.h> <signal.h> <stdarg.h> <stdio.h>
<stdlib.h> <string.h> <time.h>   <wchar.h>  <wctype.h>
```

`/usr/include` ディレクトリとその下位ディレクトリに存在する、その他の数百のヘッダーは、C 言語標準に規定されていないため、この言語の変更による影響を受けることはありません。

これらのヘッダーは、旧リリースの Sun C++ と同様の方法で C++ プログラムにインクルードして使用することができますが、以下のことに注意してください。

C++ 標準では、型、オブジェクト、これらのヘッダー中で使用する関数名は、大域的な名前空間だけでなく、std 名前空間にも記述するよう規定しています。つまり、Solaris 2.6 および Solaris 7 オペレーティング環境に付属しているヘッダーはそのままでは使用できないということです。標準モードでコンパイルする場合は、Sun WorkShop 6 C++ コンパイラに付属しているヘッダーを使用してください。ヘッダーが正しくないと、プログラムのコンパイルやリンクが失敗する可能性があります。

Solaris 2.6 および Solaris 7 オペレーティング環境では、ヘッダーにはパス名ではなく標準のヘッダー名を使用してください。次の文は正しい例です。

```
#include <stdio.h> // 正しい
```

次の文は正しくない例です。

```
#include "/usr/include/stdio.h" // 誤り
#include </usr/include/stdio.h> // 誤り
```

Solaris 8 オペレーティング環境では、/usr/include にある標準の C ヘッダーは C++ にも有効であり、C++ コンパイラによって自動的に使用されます。したがって、次の文を使用した場合、Solaris 2.6 と 7 オペレーティング環境でコンパイルしたときは、C++ コンパイラ付属の stdio.h が使用されます。

```
#include <stdio.h>
```

ところが、Solaris 8 オペレーティング環境でコンパイルしたときは、Solaris 付属の stdio.h が使用されます。Solaris 8 オペレーティング環境では、include 文で明示的なパス名を使用することに関する制限はありません。しかし、パス名 (</usr/include/stdio.h> など) を使用すると、コードの移植性が失われます。

C++ 標準では、17 個ある C 標準のヘッダーのすべてに、別のバージョンのヘッダーが追加されています。つまり、<*NAME*.h> 形式のすべてのヘッダーについて、ヘッダー名の最後の .h を落とし、先頭に c を追加した <c*NAME*> という形式の名前を持つヘッダーがあります (*NAME* は名前を示します)。例：<cstdio>、<cstring>、<cctype>

新しいバージョンのヘッダーには、従来の形式のヘッダーで使用されていた名前が含まれていますが、新しいバージョンのヘッダーは std 名前空間にのみ存在します。次に、C++ 標準に則った使用例を示します。

```
#include <cstdio>
int main() {
    printf("Hello, ");            // エラー、printf が不明
    std::printf("world!\n");      // OK
}
```

<stdio.h> の代わりに <cstdio> が使用されているため、printf という名前は、名前空間 std にあるだけで、大域的な名前空間にはありません。printf の名前を修飾するか、using 宣言を追加する必要があります。

```
#include <cstdio>
using std::printf;
int main() {
    printf("Hello, ");         // OK
    std::printf("world!\n");   //OK
}
```

/usr/include 中の C 標準のヘッダーには、C 標準では使用が許可されていない宣言が多数含まれています。これらの宣言が存在しているのは、慣習上の理由によります。つまり、UNIX システムにおいては、これらのヘッダー中で慣例的に標準外の宣言が使用されてきたということ、また、他の標準 (POSIX や XOPEN など) においては、そうした宣言が必要になるためです。互換性を維持するため、Sun C++ の <*NAME*.h> ヘッダーには、そうした名前が残されていますが、これらの名前は、大域的な名前空間にしか存在せず、新しいバージョンの <c*NAME*> ヘッダーには存在しません。

これは、新しいヘッダーが以前のプログラムには使用されていないため、互換性や慣習上の問題が生じることはないためです。<cNAME> は、一般的なプログラミングに役立たないと思われるかもしれませんが、最大の移植性を持つ標準的な C++ コードを作成するには、<cNAME> ヘッダーに移植不可能な宣言が含まれないようにする必要があります。<stdio.h> を使用した例を次に示します。

```
#include <stdio.h>
extern FILE* f; // std::FILE でも OK
int func1() { return fileno(f); }       // OK
int func2() { return std::fileno(f); } // エラー
```

次の例では、<cstiod> を使用しています。

```
#include <cstdio>
extern std::FILE* f; // FILE は名前空間 std にだけある
int func1() { return fileno(f); }       // エラー
int func2() { return std::fileno(f); } // エラー
```

上記の例の中の fileno は、互換性を維持するために <stdio.h> に残されている「標準外」の関数です。この関数は、大域的な名前空間にのみ存在し、std 名前空間には存在しません。標準外の関数であるため、<cstdio> にも存在しません。

注 – Solaris 2.6 オペレーティング環境用の、Sun WorkShop 6 C++ コンパイラ版 <wchar.h>、<cwchar>、<wctype.h>、および <cwctype> ヘッダーファイルからは、いくつかの関数が意図的に除外されています。これは、Solaris 2.6 環境ではそれらの関数の実行に必要な機能が装備されていないためです。

C++ 標準では、同じコンパイル単位で <*NAME*.h> と <c*NAME*> の両方のバージョンの C 標準ヘッダーを使用することを許可しています。一般的に、このことが意図的に行われることはないと思われますが、たとえば、使用するプロジェクトヘッダーに <stdlib.h> が含まれていて、作成したコードに <cstdlib> がインクルードされたときには、このことが起こります。Solaris 2.6 オペレーティング環境では、いくつかのヘッダー、特に <wchar.h> と <cwchar>、<wctype.h> と <cwctype> ヘッダーの組み合わせについては機能しません。これらのヘッダーを使用したときにコンパイラのエラーが出力された場合は、<c*NAME*> 版ではなく、<*NAME*.h> 版のヘッダーを使用してください。

標準ヘッダーの実装

『C++ ユーザーズガイド』には、標準ヘッダーの実装方法とともに、その方法が採用された理由が詳細にわたって説明されています。C または C++ 標準のヘッダーをインクルードした場合、コンパイラは実際には指定された名前の後に `.SUNWCCh` が付いた名前を持つファイルを検索します。たとえば、`<string>` であれば `<string.SUNWCCh>`、`<string.h>` であれば `<string.h.SUNWCCh>` を検索します。コンパイラの `include` ディレクトリには、この両方の名前が含まれており、2 つの名前のどちらも同じファイルを示します。たとえば、`include/CC` ディレクトリには、`string` と `string.SUNWCCh` の両方が含まれており、これらの名前は同じファイル (`<string>` をインクルードしたときにアクセスされるファイル) を示します。

エラーメッセージとデバッグ情報には、`.SUNWCCh` という接尾頭辞は追加されません。たとえば、`<string>` をインクルードした場合、エラーメッセージとデバッグ情報には、`string` とだけ示されます。接尾頭辞なしの名前に関するデフォルトのメークファイル規則で問題が起きるのを避けるため、ファイル依存情報には、`string.SUNWCCh` が使用されます。たとえば、SunOS の `find` コマンドを使用して、単にヘッダーファイルだけ探す場合は、`.SUNWCCh` 接尾辞で検索するようにしてください。

第5章

C++ 3.0 からの移行

ここでは、C++ 3.0 または 3.0.1 コンパイラ用に作成したコードを Sun WorkShop 6 C++ 用に移行するときのいくつかの注意事項について説明します。

C++ 3.0 コンパイラ以降に追加されたキーワード

C++ 3.0 コンパイラ以降に C++ に追加されたキーワードは、次のとおりです。これらのキーワードを識別子として使用している場合は、名前を変更してください。20 ページの表 3-1 で説明しているように、一部のキーワードを無効にすることができます。

表 5-1　C++ 3.0 コンパイラ以降に追加されたキーワード

```
bool、false、true
const_cast、dynamic_cast、reinterpret_cast、static_cast
explicit
export
mutable
namespace、using
typename
wchar_t
```

ソースコードの非互換性

C++ 3.0 コンパイラ用に作成したコードを Sun WorkShop 6 C++ コンパイラでコンパイルするには、次の変更が必要です。

- C++ 5.0 コンパイラでは、K&R 形式の関数定義を使用することはできません。プロトタイプ形式の関数定義を使用してください。

```
int f(a) int a; { ... } // エラー
```

- 代入で _new_handler 大域変数を設定することはできません。この目的には、set_new_handler() 関数を呼び出してください。

- クラス内に operator new() がない場合は、常に大域の operator new() が使用されます。C++ 3.0 では、誤って、大域のものではなくクラスの外側のものが使用される場合があります。次の例では、C++ 3.0 は、領域の割り当てに、誤って Outer::operator new を使用しています。

```
class Outer {
public:
    void* operator new(size_t);
    class Inner {
        ...                             // operator new なし
    };
};
Outer::Inner* p = new Outer::Inner; // どちらの operator new か？
```

- typedef 名を、struct、class、union のタグとして使用することはできません。次に例を示します。

```
typedef struct { int x; } S;
struct S b;        // C++ 3.0 では問題なかったが、現在はエラー
S c;               // 常に OK
```

代わりに、構造体、クラス、共用体でタグを使用してください。上記の例のエラーを解決する最も簡単な方法は、typedef 名をさらにタグとして使用する方法です。この方法は、C でも C++ でも許可されます。

```
typedef struct S { int x; } S;
struct S b; // 常に OK
S c;        // 常に OK
```

- 名前をクラス内で使用した場合、その名前を外側のスコープから定義し直すことはできません。大きな問題になる可能性があるため、C++ 標準では、そうした再定義は許可していません。C++ 3.0 コンパイラでは、このようなコードを検出しませんでしたが、現在は再定義はエラーになります。次に例を示します。

```
typedef int T;
class C {
    T iv;              // int 型
    typedef float T;   // T を再定義、エラー
    T fv;              // float 型
};
```

このエラーを解決するには、T 定義のいずれかの名前を変更します。

- C++ 3.0 コンパイラでは、次の例に示すように、定義されていないパラメータをとる関数へのポインタが、状況によっては、「汎用」の関数へのポインタとして機能してしまうという問題がありました。C++ の規則では、関数へのポインタの型は一致している必要があります。

```
typedef (*pfp)(int,char);
typedef (*ufp)(...);
int foo(int,char);
pfp p = (ufp)foo; // 3.0 では許可されたが、現在はエラー
```

- NULL ポインタ定数でコンマを使った式を使用することはできません。リテラルのゼロは NULL ポインタ定数ですが、(*anything*, 0) というような式 (*anything* は任意の値を示す) は NULL ポインタ定数ではありません。

```
int f();
char* g()
{
    return (f(), 0); // 3.0 では問題なかったが、現在はエラー
    // 次のようにするか、
    //     return (f(), (char*)0); // OK
    // または次の 2 つの文にする
    //    f();
    //    return 0;
}
```

- 基底クラスを持つクラスは、集合体の初期化構文を使用して初期化することはできません。C++ 3.0 コンパイラでは、仮想関数が存在していない場合、このことが許されていました。そうしたクラスには、コンストラクタを使用してください。

```
struct Base { int i; };
struct Derived : Base { int j; };
Derived d = {1, 2}; // 3.0 では問題ないが、5.0 ではエラー
```

第6章

C から C++ への移行

この章では、C プログラムを C++ プログラムに移行する方法について説明します。

一般的に、C プログラムを C++ プログラムとしてコンパイルするにあたって必要な修正はほとんどありません。C と C++ はリンク時の互換性があるため、C++ コードとリンクするために、コンパイル済みの C コードを修正する必要はありません。C++ 言語の個別の情報につては、『The C++ Programming Language』(Margaret A. Ellis、Bjarne Stroustrup 共著) を参照してください。

予約キーワードと事前定義済みのキーワード

表 6-1 に、C++ および C の全予約キーワードと、C++ で事前定義されているキーワードを示します。C++ では予約されていて、C では予約されていないキーワードは**太字**で示しています。

表 6-1　予約キーワード

asm	do	if	return	typedef
auto	double	**inline**	short	**typeid**
bool	**dynamic_cast**	int	signed	typename
break	else	long	sizeof	union
case	enum	**mutable**	static	unsigned
catch	**explicit**	**namespace**	**static_cast**	**using**
char	**export**	**new**	struct	**virtual**
class	extern	**operator**	switch	void

表 6-1　予約キーワード

const	**false**	**private**	**template**	volatile
const_cast	float	**protected**	**this**	**wchar_t**
continue	for	**public**	**throw**	while
default	**friend**	register	**true**	
delete	goto	**reinterpret_cast**	**try**	

__STDC__ は、あらかじめ値 0 に定義されています。たとえば、次のコードがあるとします。

```
#include <stdio.h>
main()
{
    #ifdef __STDC__
        printf("yes\n");
    #else
        printf("no\n");
    #endif

    #if    __STDC__ ==0
        printf("yes\n");
    #else
        printf("no\n");
    #endif
}
```

この結果は次のようになります。

```
yes
yes
```

次の表は、ISO C++ 標準委員会の現在の ANSI/ISO 検討資料に指定されている、演算子と句読文字の代替表現のための予約語を示したものです。

表 6-2　演算子と句読文字に対する C++ の予約語

and	bitor	not	or	xor
and_eq	compl	not_eq	or_eq	xor_eq
bitand				

汎用ヘッダーファイルの作成

K&R C、ANSI C、C++ にはそれぞれ異なるヘッダーファイルが必要です。C++ ヘッダーファイルを K&R C 標準と ANSI C 標準に準拠する汎用ヘッダーファイルにするには、マクロ `__cplusplus` を使って、C++ コードと C コードを分離する必要があります。また、マクロ `__STDC__` は ANSI C にも C++ にも定義されています。C++ や ANSI C コードを K&R C コードと分離するには、このマクロを使用します。詳細は、『C++ プログラミングガイド』を参照してください。

注 – 以前の C++ コンパイラが事前定義していたマクロ `c_plusplus` は、現在はサポートされていません。代わりに `__cplusplus` を使用してください。

C 関数へのリンク

コンパイラは、C++ 関数名を符号化して、多重定義を可能にします。C 関数あるいは「C 関数を装った」C++ 関数を呼び出すには、`extern "C"` 宣言を使用して、この符号化が行われないようにする必要があります。次に例を示します。

```
extern "C" {
double sqrt(double); //sqrt(double) は C リンケージを持つ
    }
```

このリンケージ指定によって `sqrt()` を使用するプログラムの意味が変わることはありません。`sqrt()` に対して、コンパイラが C の命名規則を使用することになるだけです。

C リンケージを持つことができるのは、複数の多重定義の C++ 関数のうちの 1 つだけです。つまり、C プログラムから呼び出す C++ 関数に C リンケージを使用することはできますが、使用できるのは、その関数の 1 つのインスタンスだけということになります。

関数定義の中で C リンケージを指定することはできません。そうした宣言は、大域のスコープでのみ行うことができます。

索引

よ

れ

わ