



C++ 移行ガイド

Sun™ Studio 8

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-5805-10
2004 年 3 月 , Revision A

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

この配布には、第三者が開発したソフトウェアが含まれている可能性があります。

フォント技術を含む第三者のソフトウェアは、著作権法により保護されており、提供者からライセンスを受けているものです。

libdwarf and lldredblack are Copyright 2000 Silicon Graphics Inc. and are available under the GNU Lesser General Public License from <http://www.sgi.com>.

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。

Sun、Sun Microsystems、Forte、Java、iPlanet、NetBeans および docs.sun.com は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。

サンロゴマークおよび Solaris は、米国 Sun Microsystems 社の登録商標です。

Netscape および Netscape Navigator は、米国ならびに他の国における Netscape Communications Corporation の商標または登録商標です。

すべての SPARC の商標はライセンス規定に従って使用されており、米国および他の各国における SPARC International, Inc. の商標または登録商標です。SPARC の商標を持つ製品は、Sun Microsystems, Inc. によって開発されたアーキテクチャに基づいています。

このマニュアルに記載されている製品および情報は、米国の輸出規制に関する法規の適用および管理下にあり、また、米国以外の国の輸出および輸入規制に関する法規の制限を受ける場合があります。核、ミサイル、生物化学兵器もしくは原子力船に関連した使用またはかかる使用者への提供は、直接的にも間接的にも、禁止されています。このソフトウェアを、米国の輸出禁止国へ輸出または再輸出すること、および米国輸出制限対象リスト(輸出が禁止されている個人リスト、特別に指定された国籍者リストを含む)に指定された、法人、または団体に輸出または再輸出することは一切禁止されています。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含み、明示的であるか黙示的であるかを問わず、あらゆる説明および保証は、法的に無効である限り、拒否されるものとします。

原典： C++ Migration Guide
Part No: 817-5069-10
Revision A

© 2004 by Sun Microsystems, Inc.



目次

はじめに xi

1. 概要 1

C++ 言語 1

コンパイラの動作モード 2

標準モード 2

互換モード 3

バイナリ互換の問題 4

言語の変更 4

新旧バイナリの混在 5

導入 6

要件 6

インタフェースの構築 8

条件式 9

関数ポインタと `void*` 10

将来の変更について 11

不正な符号化の症状 12

2. 互換モードの使用方法 15

- 互換モード 15
- 互換モードで有効なキーワード 16
- 言語の意味 16
 - コピーコンストラクタ 17
 - static 記憶クラス 17
 - 演算子 new および delete 18
 - new const 18
 - 条件式 18
 - デフォルトのパラメータ値 19
 - 末尾にコンマを使用する 19
 - const 値またはリテラル値を渡す 19
 - 関数へのポインタと void* 間の変換 20
 - enum 型 20
 - メンバー初期化リスト 20
 - const と volatile 修飾子 21
 - 入れ子の型 21
 - クラステンプレートの定義と宣言 21
- テンプレートコンパイルモデル 22

3. 標準モードの使用法 23

- 標準モード 23
- 標準モードのキーワード 23
- テンプレート 25
 - 型名の解決 25
 - 新しい規則への移行 26
 - 明示的なインスタンス化と特殊化 26
 - クラステンプレートの定義と宣言 28
 - テンプレートレポジトリ (テンプレートの格納場所) 28

テンプレートと標準ライブラリ	29
クラス名の挿入	30
for 文中の変数	32
関数へのポインタと void* 間の変換	33
文字列リテラルと char*	33
条件式	36
新しい形式の new と delete	36
new と delete の配列形式	37
例外の指定	37
置き換え関数	39
インクルードするヘッダー	40
ブール型	40
extern "C" 関数へのポインタ	41
言語リンケージ	42
移植性の低い解決策	44
関数のパラメータとしての関数へのポインタ	45
実行時の型識別 (RTTI)	47
標準の例外	47
静的オブジェクトの破棄の順序	48
4. 入出力ストリームとライブラリヘッダーの使用方法	51
入出力ストリーム	51
タスク (コルーチン) ライブラリ	54
Rogue Wave Tools.h++	54
C ライブラリヘッダー	55
標準ヘッダーの実装	58
5. C から C++ への移行	59

予約キーワードと事前定義済みのキーワード 59

汎用ヘッダーファイルの作成 61

C 関数へのリンク 61

C および C++ のインライン関数 62

索引 63

表目次

表 2-1	互換モードで有効なキーワード	16
表 3-1	標準モードで有効なキーワード	24
表 3-2	トークンとトークン代替文字列	24
表 3-3	例外関連の型名	47
表 5-1	予約キーワード	59
表 5-2	演算子と句読文字に対する C++ の予約語	60

コード例目次

コード例 3-1	クラス名挿入の問題 1	30
コード例 3-2	クラス名挿入の問題 2	31
コード例 3-3	標準ヘッダー <code><new></code>	38
コード例 4-1	標準の名前形式の <code>iostream</code> を使用	52
コード例 4-2	従来の名前形式の <code>iostream</code> を使用	52
コード例 4-3	従来の入出力ストリームによる前方宣言	53
コード例 4-4	標準の入出力ストリームによる前方宣言	53
コード例 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	54

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。

書体と記号について

次の表と記述は、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ています。

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。	<code>.login</code> ファイルを編集します。 <code>ls -a</code> を使用してすべてのファイルを表示します。 <code>machine_name% You have mail.</code>
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。	<code>machine_name% su</code> <code>Password:</code>
AaBbCc123 またはゴシック	コマンド行の変数部分。実際の名前または実際の値と置き換えてください。	<code>rm filename</code> と入力します。 <code>rm ファイル名</code> と入力します。

書体または記号	意味	例
『 』	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。	<code>machinename% grep '^#define \\\nXV_VERSION_STRING'</code>
➤	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
UNIX の C シェル	<code>machine_name%</code>
UNIX の Bourne シェルと Korn シェル	<code>machine_name\$</code>
スーパーユーザー (シェルの種類を問わない)	<code>#</code>

コンパイラコレクションのツールとマニュアルページへのアクセス

コンパイラコレクションのコンポーネントとマニュアルページは、標準の `/usr/bin/` と `/usr/share/man` の各ディレクトリにインストールされません。コンパイラとツールにアクセスするには、`PATH` 環境変数にコンパイラコレクションのコンポーネントディレクトリを必要とします。マニュアルページにアクセスするには、`PATH` 環境変数にコンパイラコレクションのマニュアルページディレクトリが必要です。

`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、『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。

注 – この節に記載されている情報は Sun ONE Studio コンパイラコレクションコンポーネントが `/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。ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

コンパイラとツールへのアクセス方法

`PATH` 環境変数を変更して、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るかどうか判断するには以下を実行します。

▼ `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

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` 変数はコンパイラと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、次の指示に従って、`PATH` 環境変数を設定してください。

▼ PATH 環境変数を設定してコンパイラとツールにアクセスする

1. C シェルを使用している場合は、ホームの `.cshrc` ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの `.profile` ファイルを編集します。
2. 次のパスを `PATH` 環境変数に追加します。

```
/opt/SUNWspro/bin
```

マニュアルページへのアクセス方法

マニュアルページにアクセスするために `MANPATH` 変数を変更する必要があるかどうかを判断するには以下を実行します。

▼ 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

1. 次のように入力して、`dbx` マニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

`dbx(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。

▼ MANPATH 変数を設定してマニュアルページにアクセスする

1. C シェルを使用している場合は、ホームの `.cshrc` ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの `.profile` ファイルを編集します。
2. 次のパスを `PATH` 環境変数に追加します。

```
/opt/SUNWspro/man
```

コンパイラコレクションのマニュアルへのアクセス

マニュアルには、以下からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークの製品にインストールされているマニュアルの索引から入手できます。
`/opt/SUNWspro/docs/ja/index.html`

製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

- マニュアルは、`docs.sun.com` の Web サイトで入手できます。以下に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (`docs.sun.com` Web サイトでは入手できません)。
 - 『Standard C++ Library Class Reference』
 - 『標準 C++ ライブラリ・ユーザーズガイド』
 - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
 - 『Tools.h++ ユーザーズガイド』
- リリースノートは、`docs.sun.com` で入手できます。

インターネットの `docs.sun.com` Web サイト (<http://docs.sun.com>) から、サンのマニュアルを参照したり、印刷したり、購入することができます。マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 - Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは以下の表に示す場所から参照することができます。製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアル (サードパーティ製マニュアルは除く)	形式 : HTML (日本語版は PDF のみ) 場所 : http://docs.sun.com
サードパーティ製マニュアル:	形式 : HTML 場所 : <code>file:/opt/SUNWspro/docs/ja/index.html</code> のマニュアル索引
『Standard C++ Library Class Reference』 『標準 C++ ライブラリ・ユーザーズガイド』 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 『Tools.h++ ユーザーズガイド』	
Readme および マニュアル ページ	形式 : HTML 場所 : <code>file:/opt/SUNWspro/docs/ja/index.html</code> のマニュアル索引
リリースノート	製品 CD 内の HTML ファイル

関連するコンパイラコレクションマニュアル

以下の表は、file:/opt/SUNWspro/docs/ja/index.html および <http://docs.sun.com> から参照できるマニュアルの一覧です。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルタイトル	内容の説明
数値計算ガイド	浮動小数点計算の数値精度に関する問題について説明します。

関連する Solaris マニュアル

次の表では、docs.sun.com の Web サイトで参照できる関連マニュアルについて説明します。

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 8 Reference Manual Collection	マニュアルページの節を参照。	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris 8 Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris 8 Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。

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。

『プログラミング言語 C++』第3版、アジソン・ウェスレイ・パブリッシャーズ・ジャパン、Bjarne Stroustrup 著、1998 年

『C++ 標準ライブラリチュートリアル & リファレンス』アスキー、Nicolai Josuttis 著、2001 年

『Generic Programming - STL による汎用プログラミング』アスキー、Matthew Austern 著、2000 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

『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

『デザインパターン - オブジェクト指向における再利用のための』ソフトバンク、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著、1995 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++』改訂2版、アスキー、Scott Meyers 著、1998 年

『More Effective C++ - 最新 35 のプログラミング技法』アスキー、Scott Meyers 著、1998 年

『Efficient C++ パフォーマンスプログラミングテクニック』ピアソン・エデュケーション、Dov Bulka、David Mayhew 共著、2000 年

開発者向けのリソース

<http://www.sun.com/developers/studio> にアクセスし、Compiler Collection というリンクをクリックして、以下のようなリソースを利用できます。リソースは頻繁に更新されます。

- プログラミング技術と最適な演習に関する技術文書
- プログラミングに関する簡単なヒントを集めた知識ベース
- **Compiler Collection** のコンポーネントのマニュアル、ソフトウェアと共にインストールされるマニュアルの訂正
- サポートレベルに関する情報
- ユーザーフォーラム
- ダウンロード可能なサンプルコード
- 新しい技術の紹介
- <http://www.sun.co.jp/developers/> でも開発者向けのリソースが提供されています。

技術サポートへの問い合わせ

製品についての技術的なご質問がございましたら、以下のサイトからお問い合わせください (このマニュアルで回答されていないものに限りです)。

<http://sun.co.jp/service/contacting>

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。また、C++ 5.0、5.1、5.2、5.3、5.4、5.5 をまとめて "C++ 5" 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 で機能します。コンパイラには、ほぼすべての C++ 4 コードを変更せずに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、5.1、5.2、5.3、5.4、5.5 コンパイラのいずれかの標準モード (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、5.1、5.2、5.3、5.4、5.5 コンパイラで使用できます。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 C++ 5.4 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。このもっとも顕著な例は、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。従来の初期 C++ プログラム

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

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。Forte Developer 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 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があります。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

C++ コンパイラには、標準モードと互換モードという 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 コンパイラは、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 (`-compat[=4]`) 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 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。

注 – この文書では、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 製および 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総称して Intel アーキテクチャ "IA" と呼んでいます。

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 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。

注 - ほとんどの場合、互換モード (`-compat[=4]`) 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、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ファイルやライブラリをリンクすることはできません。詳細は、5 ページの「新旧バイナリの混在」を参照してください。

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、構造型または集合型の配置、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。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 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ンプレート関数は、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

古いバイナリ (C++ 4 コンパイラ、または C++ 5 コンパイラの互換モードでコンパイルしたオブジェクトファイルやライブラリ) と新しいバイナリ (C++ 5 コンパイラの標準モードでコンパイルしたオブジェクトファイルやライブラリ) は絶対にリンクできないというわけではありません。libExbridge ライブラリを使用することにより、SPARC プラットフォーム上でそれらをリンクすることは可能です。

注 - libExbridge を使用したモードの混在は、x86 プラットフォームではなく、SPARC プラットフォームでのみサポートされます。x86 プラットフォームで libExbridge を使用するには、ABI の変更とコードの再コンパイルが必要になります。コードを再コンパイルできる場合、libExbridge は必要ありません。単一プログラムに互換モードを混在させることは推奨できません。代わりに標準モードですべてのコードをコンパイルしてください。短期と長期の両方で実行すると、さらに好都合です。モードを混合して問題が解決することではなく、libExbridge を使用しても長期的には確実な解決につながりません。

導入

アプリケーションが構築されているか実行されているすべてのシステムに、最新の SUNWlibc パッチをインストールします。新しいパッチの libExbridge.so.1 ライブラリは、同じパッチの libC.so.5 および libCrun.so.1 のバージョンに依存します。そのため、libExbridge.so.1 をインストールしないでシステムにコピーするだけでは、有効とはなりません。

要件

以下の場合、(上記の説明に当てはまるような) 古いバイナリと新しいバイナリをリンクすることができます。

例外の使用

コードが例外を使用する場合、つまり、コードに throw または catch キーワード (関数上の例外仕様を含む) が含まれている場合、次の要件があります。

- 標準モードと互換モードの両コードが例外を使用する場合、送出点から捕獲点までのすべてのアクティブ関数が同じモードでコンパイルされる限り、コードは動作します。つまり、送出点からキャッチクローズにスタックをウォークアップする際、キャッチブロックを保持するすべての C++ 関数は、同一モードでコンパイルする必要があります。
- ライブラリ libC および libCrun を静的にリンクしてはなりません。コンパイラのデフォルトである共有 (.so) バージョンのライブラリをリンクする必要があります。libExbridge.so.1 をリンクすると、libCrun.so.1 と libC.so.5 は自動的にリンクされます。詳細については7 ページの「libExbridge ライブラリの使用」を参照してください。
- -Bsymbolic リンカーオプションを使用して共有ライブラリを作成したり、
-Bsymbolic リンカーオプションを使用して構築したライブラリをリンクすべきではありません。

使用しているバージョンの C++ コンパイラが -xldscope でスコープするリンカーをサポートする場合、その機能を使用して、ライブラリの記号の可視性を制御します。詳細については、C++ User's Guide または CC(1) の -xldscope を参照してください。

使用しているバージョンの C++ コンパイラが `-xldscope` でスコープするリンカーをサポートしない場合、リンカーのマッピングファイルを使用して、ライブラリの記号の可視性を制御します。詳細については、『[Linker and Libraries Guide](#)』を参照してください。

libExbridge ライブラリの使用

libExbridge の好ましい使用方法としては、コード内でリンクします。そのためには、`-lExbridge` オプションを CC コマンドに追加します。`-l` オプションは、lib をライブラリ名に付加します。

libExbridge とのリンクができない場合、次の指示に従ってください。

- C シェルに対し、次のようにライブラリを事前読み込みします。

```
example% setenv LD_PRELOAD /usr/lib/libExbridge.so.1
example% my_application arg1 arg2
```

- Bourne シェルまたは Korn シェルに対し、次のようにライブラリを事前読み込みします。LD_PRELOAD を設定した後にセミコロンは配置しません。

```
$ LD_PRELOAD=/usr/lib/libExbridge.so.1 my_application arg1 arg2
```

LD_PRELOAD を設定すると、その後に起動したすべてのプログラムが影響を受けます。libExbridge を事前読み込みすると、シェル、またはシェルを生成するプログラムの呼び出しが失敗することがあります。`/usr/bin/sh` は、事前読み込みされたライブラリの `.init` セクションに呼び出されても正しく初期化されない独自の `malloc` を定義します。

そのため、C シェルの使用時は、アプリケーションを実行するシェルスクリプトに環境変数を設定しておくのがベストです。上に示した Bourne シェルと Korn シェルの構文は、同一行のコマンドに対してのみ環境変数を作成します。

libExbridge を事前読み込みして、アプリケーションがシェルを生成する場合、アプリケーションに不具合が生じることがあります。そのような場合、ライブラリを事前読み込みする代わりに、libExbridge を使用してアプリケーションを再リンクする必要があります。

インタフェースの構築

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。

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 の型だけを使用する。
- 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。
- 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。例外処理は混在しません。

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。

C コンパイラ (cc) を使用して、アプリケーションを C インタフェースライブラリにリンクする際に、そのライブラリが C++ 実行時サポートを必要とする場合は、以下に示すいずれかの方法を使用し、libC (互換モード) または libCrun (標準モード) のいずれかに依存性を作成する必要があります。C インタフェースライブラリに C++ 実行時サポートが必要ない場合は、libC や libCrun をリンクする必要はありません。

- **アーカイブ C インタフェースライブラリ** アーカイブ C インタフェースライブラリを使用する場合は、ライブラリの使用方法を提示する必要があります。

- **標準モードC** インタフェースライブラリが、C++ コンパイラ (cc) の標準モード (デフォルトのモード) で構築されていた場合は、C インタフェースライブラリを使用するときに、コマンド行で cc コマンドに `-lCrun` を追加する必要があります。
- **互換モードC** インタフェースライブラリが、C++ コンパイラ (cc) の互換モード (`-compat`) で構築されていた場合は、C インタフェースライブラリを使用するときに、コマンド行で cc コマンドに `-lC` を追加する必要があります。
- **共有 C インタフェースライブラリ** 共有 C インタフェースライブラリを使用する場合は、ライブラリの構築時に `libC` または `libCrun` のいずれかに依存性を作成する必要があります。共有ライブラリが正しい依存性を持っている場合は、ライブラリを使用するときに、cc コマンドに `-lC` や `-lCrun` を追加する必要はありません。
- **標準モードC** インタフェースライブラリが、標準モード (デフォルトのモード) で構築されている場合は、ライブラリを構築するときに、コマンド行で cc コマンドに `-lCrun` を追加する必要があります。
- **互換モードC** インタフェースライブラリが、互換モード (`-compat`) で構築されている場合は、ライブラリを構築するときに、コマンド行で cc コマンドに `-lC` を追加する必要があります。

条件式

C++ 標準は条件式の規則に変更を導入しました。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。

現在のコンパイラは、標準モードと互換モードの両方で新しい C++ 規則を使用します。新しい規則では、上記の式がユーザーの作成したものであ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、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。標準モードでは、コンパイラは、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。4.2 コンパイラに準拠しているそのようなコードは、標準モードではエラー (一致する関数がない) になります。互換モードでは、従来の古い形式で警告が出力されます。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。例:

```
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、標準モードでは一致する関数がないため、エラーメッセージが出力されます。`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 C++ 標準に違反する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。41 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記号を使用することで、宣言と定義を一致させることができます。

```
int cpp_function( int arg ) { return arg; }  
#pragma_weak "__1c_missing_mangled_name" = cpp_function
```

このような宣言では、必ず符号化名を使用してください。

不正な符号化の症状

問題が発生する領域として、11 ページの「将来の変更について」で説明した機能がコードに含まれている場合は、コンパイラが矛盾して名前を符号化することがあります。症状としては、プログラムがリンクに失敗し、記号が見つからないことを表すエラーメッセージがリンカーによって出力されます。リンカーのエラーメッセージには符号化されていない名前があり、その名前は実際に定義されている関数またはオブ

ジェクトを指します。しかし、コンパイラが記号定義とは異なる方法で記号への参照を符号化したため、リンカーはそれらの名前を一致させることができません。次の例を検討してください。

```
main.cc
-----
int foo(int);    // 宣言に 'const' なし
int main()
{
    return foo(1);
}

file1.cc
-----
int foo(const int k) // 引数の定義に 'const' を追加
{
    return k;
}

example% CC main.cc file1.cc
main.cc:
file1.cc:
未定義の                最初に参照している
記号                    ファイル
int foo(int)                main.o
ld: fatal:記号参照エラー。a.out に書き込まれる出力はありません。
```

コンパイラによって生成された名前をオブジェクトファイルで調べると、この障害の原因が分かります。

```
% nm main.o | grep foo
[2]      |          0|          0|NOTY  |GLOB  |0      |UNDEF  |__1cDfoo6Fi_i_
% nm file1.o | grep foo
[2]      |         16|         40|FUNC  |GLOB  |0      |2      |__1cDfoo6Fki_i_
```

main.o では、関数 foo への参照がコンパイラによって生成されています。この関数は、file1.o で関数 foo を定義するときに符号化した名前とは異なる方法で符号化されています。11 ページの「将来の変更について」で説明したように、この問題の回避方法は、foo のパラメータの宣言で const を使用しないことです。

const パラメータを使用して矛盾なく foo を宣言するプログラムと、const 以外のパラメータを使用して矛盾なく foo を宣言するプログラムは、正常にコンパイルされてリンクされます。

コンパイラのバグを修正すると、リンクしている一部のプログラムがリンクしなくなります。たとえば、`file1.o` が含まれているサードパーティのバイナリライブラリがあるとします。コンパイラのバグを修正した場合は、`foo` の宣言がなければ、プログラムはそのライブラリ内の `foo` にリンクできます。バグを修正しなかった場合は、`const` パラメータを使用して `foo` を宣言し、正常にそのライブラリにリンクできません。

ただし、名前の符号化に関連する既知のコンパイラのバグは、「一意の」符号化名を生成します。つまり、無効な符号化名が誤って別の関数またはオブジェクトの符号化名を指すことはありません。この移行ガイドで説明しているように、記号を追加すると、不正な符号化名によって生じた問題を常に修正できます。

第2章

互換モードの使用方法

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 C++ 5 コンパイラの間に多少の使用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	無効にするコンパイラオプション
explicit	-features=no%explicit
export	-features=no%export
mutable	-features=no%mutable
typename	<i>cannot disable</i>

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

C++ 5 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。

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にもかかわらず、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。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。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。

注 - これらの規則は、C++ コンパイラの互換モードおよび標準モードの両方に適用されます。

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  
public:  
    T();  
};  
T    f1(T t) { return t; } // エラー、T が返されない  
void f2()    { f1( T() ); } // エラー、T が渡されない
```

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 `delete` は公開アクセスであると想定されています。

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。

演算子new および delete

operator new と operator delete

```
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。

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。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、9 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。

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、33 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、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。詳細は、10 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 { zero=0, one=1 };
E foo(E e)
{
    e = 0;    // エラー
    e = E(0); // OK
    return e;
}
```

解決策:キャストを使用してください。

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 char* を 非 const char* に渡している
```

解決策:ポインタ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。ポインタ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;  
};  
Outer x;           // エラー、Outer::Inner を使用する
```

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、山かつこ <> 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、28 ページの「テンプレートレポジトリ (テンプレートの格納場所)」を参照してください。

第3章

標準モードの使用方法

この章では、C++ コンパイラのデフォルトのモードである標準モードの使用方法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、多数の予測不能なエラーが発生します。プログラマがキーワードを識別子として使用した部分を判定することは非常に困難であり、そのような場合、コンパイラのエラーメッセージは役立たないことがあります。

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++ 標準は、これらの文字列を直接的に予約語として導入しました。代替文字列が有効である場合、プログラムに `<iso646.h>` を含めても効果はありません。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
<code>&&</code>	<code>and</code>
<code>&&=</code>	<code>and_eq</code>
<code>&</code>	<code>bitand</code>
<code>~</code>	<code>compl</code>
<code>!</code>	<code>not</code>
<code>!=</code>	<code>not_eq</code>
<code> </code>	<code>or</code>
<code> </code>	<code>bitor</code>
<code>~</code>	<code>xor</code>
<code>~=</code>	<code>xor_eq</code>

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。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せん。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;
template< class T > class B { typedef int U; };
template< class T > class C :public B<T> {
    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、テンプレートパラメータに依存する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 `typename` 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;
template< class T > class B { typedef int U; };
template< class T > class C :public B<T> {
    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。

```
#ifndef 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 (C++ 4.1 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 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)。例:

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

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。

```
#ifndef 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。C++ 5 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が。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」というもっと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、テンプレートに関する標準ヘッダーを示します。

algorithm	bitset	complex	deque	exception
fstream	functional	io manip	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。例:

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。そのスコープには、メンバーの型 `iterator` があります。そのため、パラメータの型は `list<T>::iterator` になります。

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めます。`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 ) ...// このコードは OK ですか
```

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 OK
```

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、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。標準モードでは、コンパイラは、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、10 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*

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] = 'G';
printf("%s", GREETING); //システムによっては"Gello world"と出力される
```

C、C++ とともに、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、上の例のように予測不能な出力となります。

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";           // 以前は OK でしたが、現在は反対されています
const char* p2 = "Hello";    // OK
void f(char*);
f(p1);                        // p1 が const で宣言されていないため、常に OK です。
f(p2);                        // 常にエラーになります。const char* が char* に渡されます
f("Hello");                  // 以前は OK でしたが、現在は反対されています
void g(const char*);
g(p1);                        // 常に OK です
g(p2);                        // 常に OK です
g("Hello");                  // 常に OK です
```

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ります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。例:

```
void f(char*);
void f(const char*);
f("Hello"); // どの f が呼び出されますか
```

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。標準モードでコンパイルした場合、関数 `f(const 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 (`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=noconststrings` 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。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、9 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。互換モードの実行時ライブラリ (libc) は古い定義と動作に依存し、標準モードの実行時ライブラリ (libCstd) 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推奨できません。

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;
}
// 単一オブジェクトの形式
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();
// 配列の形式
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 ) {                // OK ではありません
    ...                      // ハンドル割り当ての障害
}
...                          // p を使用します
```

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& ) {
    ...           // ハンドル割り当ての障害
}
...             // p を使用します
```

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

```
T* p = new (std::nothrow) T;
... オリジナルから変わらないコードのメモ
```

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 `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 `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。

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。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。詳細については47 ページの「標準の例外」を参照してください。

ブール型

`bool` キーワード (`-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。例:

```
// 合理的に互換性のあるブール型を定義します
#if !defined(_BOOL) && !defined(BOOL_TYPE)
    #define BOOL_TYPE          // ローカルインクルードガード
    typedef unsigned char bool; // 標準モードの bool は 1 バイトを使用します
    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 には C リンケージがあります
int fcpp(int) { return 1; }        // fcpp には C++ リンケージが
// fp1 および fp2 には C++ リンケージがあります
int (*fp1)(int) = fc;              // ミスマッチ
int (*fp2)(int) = fcpp;            // OK
// fp3 および fp4 には C リンケージがあります
extern "C" int (*fp3)(int) = fc;    // OK
extern "C" int (*fp4)(int) = fcpp;  // ミスマッチ
```

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。例:

```
extern "C" {
    int fc(int);
    int (*fp1)(int) = fc; // 両方に C リンケージがあります
}
int fcpp(int);
int (*fp2)(int) = fcpp;   // 両方に C++ リンケージがあります
```

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。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 ); // ミスマッチ
```

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。例:

```
extern "C" int f(int);  
int f(int i) { ... } // "C" リンケージがあります
```

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か。

```
extern "C" int g( int(*) (int) );  
int g( int (*pf) (int) ) { ... } // これは "C" リンケージと "C++" リン  
ケージのどちらですか
```

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、第 1 行において `C` リンケージを保持する関数へのポインタを取得する `C` リンケージとともに関数 `g` を宣言します。第 2 行では、`C++` リンケージを保持する関数へのポインタを取得する関数を定義します。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 コンパイラの標準モードでは、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	内容の説明
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 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。

C++ コンパイラのバージョン 5.1 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 ソフトウェアから実行できます。C++ 5.1、5.2、5.3、5.4 コンパイラ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 ソフトウェア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** ソフトウェア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%strictdestroorder` 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用方法

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 C++ 5 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、コード例 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。

コード例 4-1 標準の名前形式の `iostream` を使用

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

コード例 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。例:

コード例 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
class istream;
class ostream;
class MyClass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

従来の名前 (`istream`、`ofstream`、`stringstream` 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例化を型定義 (`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。

コード例 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&, const 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。例:

コード例 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 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&, const 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。

Rogue Wave Tools.h++

C++ コンパイラには、2 種類の Tools.h++ ライブラリが含まれています。

- **従来入出力ストリーム用** この Tools.h++ ライブラリは、以前のバージョンのコンパイラに含まれていた Tools.h++ ライブラリと互換性があります。
 - **標準モード** 従来入出力ストリーム用の Tools.h++ を標準モード (デフォルトのモード) で使用するには、`-library=rwtools7,iostream` オプションを使用します。
 - **互換モード** 従来入出力ストリーム用の Tools.h++ を互換モード (`-compat[=4]`) で使用するには、`-library=rwtools7` オプションを使用します。
- **標準入出力ストリーム用** この種類の Tools.h++ ライブラリは、従来入出力ストリーム用の Tools.h++ と互換性がありません。この種類のライブラリは、標準モードでのみ利用できます。互換モード (`-compat[=4]`) で利用することはできません。

標準の入出力ストリーム用の `Tools.h++` を使用するには、`-library=rwtools7_std` オプションを使用します。

`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)` の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 ソフトウェアに含まれる `/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。

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

`/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 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、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 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" // 間違い
#include </usr/include/stdio.h> // 間違い
```

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、次のように記述すると、

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

Solaris 7 オペレーティング環境でコンパイルする際には、C++ コンパイラ付属の `stdio.h` が使用されますが、Solaris 8 オペレーティング環境でコンパイルする際には、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。`<NAME.h>` の形式のヘッダーには、`<cNAME>` の形式のヘッダーが追加されています。つまり、末尾の `".h"` が削除され、先頭に `"c"` が付加されます。

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 が未知
    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。互換性を継続するために、Sun C++ バージョンの <NAME.h> ヘッダーには、こうした余分な名前がありますが、使用されるのは大域名前空間のなかだけです。これらの余分な名前は、<cNAME> バージョンのヘッダーでは使用されません。

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。したがって、通常のプログラミングにおいては、<cNAME> は便利でないように思えるかもしれません。しかし、移植性を最大限に活かした標準 C++ コードを記述してみると、<cNAME> ヘッダーは移植性のない宣言をまったく含んでいないことがわかります。<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。

C++ 標準では、同じコンパイル単位で `<NAME.h>` と `<cNAME>` 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。

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、ディレクトリ `include/CC/Cstd` には、`string` と `string.SUNWCCh` の両方があります。それらは、同一のファイル、つまり `<string>` をインクルードしたときに取得したファイルを参照します。

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 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++ はリンクの互換性があります。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++ 言語の参考資料については、xviii ページの「市販の書籍」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 5-1 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 5-1 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
const	false	private	template	volatile

表 5-1 予約キーワード (続き)

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
}
```

produces:

```
yes
yes
```

次の表は、C++ 標準に指定されている、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 5-2 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 関数を呼び出すには、または C++ 関数「マスカレード」を C 関数として呼び出すには、このエンコードを避ける必要があります。このエンコードを避けるには、`extern "C"` 宣言を使用します。例:

```
extern "C" {  
    double sqrt(double); //sqrt(double) has C linkage  
}
```

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。

C および C++ のインライン関数

インライン関数の定義が、C および C++ のどちらのコンパイラでもコンパイル可能なソースコード内にある場合、その関数は以下に示す制限事項に従う必要があります。

- インライン関数の宣言および定義は、条件文 `extern "C"` で囲む必要があります。以下に例を示します。

```
#ifdef __cplusplus
extern "C" {
#endif
inline int twice( int arg ) { return arg + arg; }
#ifdef __cplusplus
}
#endif
```

- インライン関数の宣言および定義は、両方の言語から受ける制約に従う必要があります。
- 関数の意味は、両方のコンパイラにおいて同一である必要があります。プログラムの構築における、両言語で生じる意味上の違いについては、C++ 標準の規格書の付録 D を参照してください。

索引

数字

64 ビットアドレス空間 3

A

ARM (注釈リファレンスマニュアル) 1, 4, 10, 26, 32, 42

C

C++ 言語 1 - 2

意味 16 - 21

規則 16

変更 2, 4

C++ 国際標準 2

C++ 標準ライブラリ 2, 29, 41

char* 33

-compat command 15

-compat コマンド 23

const

new 18

new による割り当て 18

将来の変更 11

ポインタ 21

文字リテラル 33

渡す 19

const 値を const 以外の参照へ渡す 19

C インタフェース 8

C ライブラリヘッダー 55

C リンケージ 43, 46, 61

C、C++ との併用 61

D

delete 18

operator 38, 39

新しい規則 18

新しい形式 36

delete 式のカウンタ 18

E

enum 型 20

extern "C" 41 - 46, 61

F

for 文中の変数 32

for 文の規則 32

I

iostreams 54

L

libExbridge ライブラリ 5

N

new 18

new rules 18

operator 37, 39

新しい形式 36

O

operator

delete 38, 39

new 37, 39

P

PATH 環境変数、設定 xiii

S

static 記憶 17

T

Tools.h++ 54

typedef

将来の変更 12

typename 25, 26

typename 16

V

void* 変換 10, 20, 33

volatile ポインタ 21

あ

アクセスできるマニュアル xv

アプリケーションバイナリインタフェース (ABI)

3, 4 - 8

い

入れ子の型 21

インクルードするヘッダー 40

インライン関数 62

か

型名の解決 25

関数へのポインタ 12, 41 - 46

「関数ポインタの変換」も参照

関数ポインタの変換 10, 20, 33

関数、インライン 62

き

キーワード 16, 17, 23, 25, 26, 40, 59

基底クラス名 20

く

クラス名の挿入 30

け

言語リンケージ 42, 46, 61

こ

互換モード 1, 3, 15 - 22

コピーコンストラクタ 17

コルーチンライブラリ 54

コンパイル、アクセス xiii

し

実行時の型識別 (RTTI) 47
修飾子、constvolatile 21
条件式 9
書体と記号について xi

せ

静的オブジェクト、破棄の順序 48 - 50

て

デフォルトのパラメータ値 19
テンプレート 25 - 30
 C++ 標準ライブラリ 29
 クラス、宣言 28
 クラス、定義 28
 コンパイルモード 22
 特殊化 26
 インスタンス化、明示的 26
 無効な型引数 21
 レボジトリ 28

と

トークン、代替文字列 24

な

名前の符号化 5, 11

に

入出力ストリーム 51 - 53

は

バイナリ互換の問題 4 - 8
 言語の変更 4

新旧バイナリの混在 5

ひ

標準ヘッダーの実装 58
標準モード 2, 23 - 50
 キーワード 23
標準例外 47

ふ

ブール型 40
符号化の問題、回避 11
古い形式の構文 16, 27, 43

へ

ヘッダーファイル 61
ヘッダー、標準 C 57, 58

ほ

ポインタの変換 10, 20, 33

ま

マクロ
 __STDC__ 61
 __cplusplus 61
末尾のコンマ 19
マニュアル索引 xv
マニュアル、アクセス xv
マニュアルページ、アクセス xiii

も

モード
 互換 3, 15 - 22
 標準 2, 23 - 50

標準との互換性の混在 5

文字リテラル 33

戻り値の型

 C インタフェース 8

 関数へのポインタ 12

 クラス 17

よ

予約語 59

れ

レボジトリ、テンプレート 28