C++ プログラミングガイド

Sun WorkShop 6

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

Part No. 806-4838-01
2000 年 6 月 Revision A

原典： *C++ Programming Guide*
Part No: 806-3571-10
Revision A

製品名の変更について

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

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

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

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

製品名の変更に加えて、次の 2 つの製品について大きな変更があります。

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

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

目次

はじめに

このマニュアルでは、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ます。このマニュアルは C++ および Solaris™/UNIX® に関する実用的な知識を持ち、Sun WorkShop の主な開発機能について理解することを目的としたアプリケーション開発者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「Sun C++ コンパイラの紹介」では、Sun C++ コンパイラの特徴について説明しています。

第 2 章「プログラムの構成」では、ヘッダーファイル、インライン関数の定義、およびテンプレートの定義について説明しています。

第 3 章「プラグマ」では、特定の情報をコンパイラに渡すプラグマ (指令) の使用方法について説明しています。

第 4 章「テンプレート」では、テンプレートの定義と使用方法について説明しています。

第 5 章「例外処理」では、Sun C++ 5.0 コンパイラで現在実装されている例外処理につて説明しています。

第 6 章「実行時の型識別」では、RTTI について説明し、本コンパイラで使用している RTTI オプションを紹介しています。

第 7 章「キャスト演算」では、新しいキャスト演算について説明しています。

第 8 章「パフォーマンス」では、C++ 関数のパフォーマンスを改善する方法について説明しています。

第 9 章「マルチスレッド化されたプログラム」では、マルチスレッド化されたプログラムの構築方法を説明します。また、例外の使用方法と、スレッド間で C++ 標準ライブラリのオブジェクトを共有する方法についても取り上げます。

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。

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 (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
	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 (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
	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 (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
	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

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

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

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

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

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

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

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

第1章

Sun C++ コンパイラの紹介

本書および関連マニュアル『C++ ユーザーズガイド』で説明する Sun WorkShop™ 6 C++ コンパイラ CC は、SPARC™ および IA プラットフォームの Solaris 2.5.1、2.6、および Solaris 8 オペレーティング環境で使用できます。Sun WorkShop 6 C++ コンパイラは、C++国際標準に記載された言語とライブラリを実装しています。

C++ 言語

C++ は、Bjarne Stroustrup による『The C++ Programming Language』で初めて登場し、その後、より公式な解説書として Margaret Ellis と Bjarne Stroustrup 共著による『The Annotated C++ Reference Manual』(『注解 C++ リファレンス・マニュアル』、トッパン刊、通称「ARM」) が刊行されました。現在は、C++ の国際標準が存在します。

C++ は C プログラミング言語の進化形として設計されています。C++ では、C の効率の良い低水準のプログラミング機能を継承しながら、次の機能が追加されています。

- 型検査の強化
- 豊富なデータ抽象化機能
- オブジェクト指向プログラミングのサポート
- 同期例外処理
- 大規模な標準ライブラリ

特にオブジェクト指向プログラミングをサポートしたことにより、モジュール性に優れて拡張性のあるプログラムモジュール間インタフェースを設計することができます。また、一連の拡張性のあるデータ型やアルゴリズムを含む標準ライブラリにより、汎用性のあるアプリケーションを効率的に開発できます。

データの抽象化

C++ では、言語で事前定義されたデータ型のように機能するデータ型をプログラマが独自に定義できます。このような抽象データ型を定義することで、問題に対処するためのひな型を作ることができます。

オブジェクト指向の特徴

C++ におけるデータの抽象化の基本単位である「クラス」にはデータが含まれ、そのデータに対する演算が定義されています。

クラスは 1 つ以上のクラスの特性を継承し、それらのクラスの下位のクラスになれます。この特性は、「継承」または「派生」と呼ばれます。引き継がれたクラス (親クラス) は C++ では「基底」クラスと呼ばれ、他のプログラミング言語では「スーパー」クラスと呼ばれます。子クラスは C++ では「派生」クラスと呼ばれ、他のプログラミング言語では「サブクラス」と呼ばれます。派生クラスは、その基底クラスのすべてのデータを含みます (通常はすべての演算も含む)。派生クラスには、基底クラスにはない新しいデータが追加されていたり、基底クラスとは異なる演算が含まれている場合もあります。

基底クラスを派生クラスに置き換えた形のクラス階層を作ることもできます。たとえば `Window` クラスは、派生クラスとして、`ScrollingWindow` クラスを持つことができ、`ScrollingWindow` クラスは `Window` クラスの特性に加えてその内容がスクロール可能であるという特性を備えています。そのため、`ScrollingWindow` クラスは、`Window` クラスが存在する位置であればどこにでも使用できます。このように代替できる特性を、多相性 (「多くの形式を持つ」という意味) と呼びます。

継承を使用し、多相性を持つ抽象データ型を使って設計されたプログラムを「オブジェクト指向のプログラム」と呼びます。

型検査

通常コンパイラ (実際にはインタプリタ) は、演算とデータの型が適合しているか確かめるために型検査を行います。C++ の型検査は C より厳格ですが、Pascal ほどではありません。Pascal が誤ったデータ型を必ず拒否するのに対し、C++ コンパイラはエラーを出力する場合もありますが、ほとんどの場合は適した型に変換します。

もちろん、このような C++ コンパイラによる自動変換のほかに、C と同様に明示的な「型変換」もできます。

関数名の多重定義についても同様です。C++ では複数の関数に同じ名前を付けることができます。コンパイラは、関数呼び出しの際にパラメータの型を検査して呼び出す関数を決定します。正しい関数がコンパイル時に判定できないと、コンパイラは「曖昧である」という意味のエラーを出力します。

クラスとデータの抽象化

C 言語でプログラムを書いた経験のある方ならば、クラスとは `struct` の概念を拡張したものであると考えるとわかりやすいでしょう。`struct` には `char` や `int` のような事前定義されたデータ型が含まれ、他の `struct` 型が含まれる場合もあります。C++ では、`struct` 型はデータを格納するデータ型だけでなく、データを操作する演算にも使用できます。C++ のキーワードである `class` は、C の `struct` と似ています。使い方としては、多くのプログラマは、C と互換性のある `struct` 型を表す場合に `struct` を使用し、C では使用できない C++ 機能を持つ `struct` 型を表す場合に `class` を使用しています。

C++ には、「データの抽象化」の手段としてクラスが用意されています。プログラムのデータとしてどのような型 (クラス) がいいかを最初に決定し、次にそれぞれの型が必要とする演算を決定します。つまり、C++ クラスはユーザーが定義するデータ型と言えます。

たとえば 大きな整数の演算を行う `BigNum` というクラスを定義する場合、クラス `BigNum` のオブジェクトと一緒に使用したときだけ意味を持つ + 演算子を定義できます。`n1` と `n2` が `BigNum` という型のオブジェクトである場合、次の式は `BigNum` で定義した + 演算子によって値が決まります。

```
n1 + n2
```

`operator+()` が定義されていないと、クラスの型に対して + 演算は実行できません。+ 演算子は、`int`、`long`、`float` などの組み込み数値型に対してのみ事前定義されています。
このように複数の定義を持つ演算子を「多重定義演算子」と呼びます。

C++ クラスのデータ記憶要素は、「データメンバー」と呼ばれます。演算には、関数と多重定義された組み込み演算子 (特殊な関数) の両方が含まれます。関数は、クラスの一部として宣言されたメンバー関数であることもあれば、クラスの外で宣言される非メンバー関数であることもあります。メンバー関数は、クラスのメンバーにだけ作

用します。クラスの非公開メンバーまたは限定公開メンバーに直接アクセスする必要がある場合は、非メンバー関数は「フレンド関数」として宣言されなければなりません。

クラスメンバーへのアクセスのレベルは、`public` (公開)、`private` (非公開)、`protected` (限定公開) という「メンバーアクセス指示子」を使って指定できます。公開メンバーは、プログラム内のすべての関数で使用できます。非公開メンバーを使用できるのは、クラスのメンバー関数とフレンド関数だけです。限定公開メンバーを使用できるのは、基底クラスのメンバー関数とフレンド関数、および派生クラスのメンバー関数とフレンド関数だけです。同じアクセス指示子を基底クラスに適用すると、影響を受ける基底クラスのすべてのメンバーに対するアクセスを限定できます。

C との互換性

C++ は、C と高い互換性が保たれるように設計されています。 C のプログラマは C++ を独自のペースで学習し、必要に応じて C++ の機能を統合することができます。C++ では C の長所や便利な点がさらに強化されていますが、特に重要な点は、システムの構成要素に直接関りを持つ型や演算子など、コンピュータのハードウェアへの効率的なインタフェースが、C と同様 C++ でも継承されていることです。

しかし重要な相違点もいくつかあります。普通の C プログラムを C++ でコンパイルするには、多少の修正が必要です。C から C++ へプログラムを移行する場合に注意すべき事項については、『 C++ 移行ガイド』で説明しています。

C と C++ では、プログラムモジュール間のインタフェースを設計する場合に最も大きな違いがありますが、C++ にはそのインタフェースの設計に使用する C の機能がすべて継承されています。たとえば、C++ モジュールを C モジュールにリンクできるため、C++ のプログラムで C のライブラリを使用することができます。

C と C++ ではこの他多くの細かな相違点があります。C++ には次に示す機能があります。

- 定義済み定数を使用すると、プリプロセッサを使用せず、プログラムで名前付き定数を使用することができます。
- 関数プロトタイプを使用する必要があります。
- 格納と解放を行う `new` 演算子と `delete` 演算子は、指定した型の動的オブジェクトを生成します。

- 参照は自動的に間接参照されるポインタであり、変数の別名のような役割を果たします。参照は関数のパラメータとして使うことができます。
- 型変換用の特殊な組み込み演算子名が用意されています。
- プログラマの定義による自動型変換を許可しています。
- 変数宣言は、文が出現するあらゆる位置で行えます。変数宣言はブロックの先頭だけでなく、`if`、`switch`、`loop` 文のヘッダー内でも行えます。
- 新しい注釈用区切り文字を使用すると、その行の末尾までが注釈として認識されます。
- 列挙名やクラス名も自動的に型名になります。
- 関数のパラメータにデフォルト値を代入できます。
- インライン関数を使用すると、関数の呼び出しが関数の本体に置換されます。これにより、マクロを使用することなくプログラムの効率を上げることができます。

第2章

プログラムの構成

C++ プログラムのファイル構成には、C プログラムの通常の場合よりも注意が必要です。この章では、ヘッダーファイル、インライン関数定義、およびテンプレート定義の設定方法について説明します。

ヘッダーファイル

ヘッダーファイルは、C と C++ 両方のさまざまなバージョンに適応させなければならないことがよくあります。このような場合に効率的なヘッダーファイルを作成するには工夫が必要です。テンプレートを用意する場合は、2 度以上のインクルードが可能 (べき等) で、他のファイルを必要としないものにしてください。

言語に対する適合性のあるヘッダーファイル

ヘッダーファイルは、C と C++ プログラムの両方にインクルードできるようにしなければならない場合があります。しかし、従来型の C としても知られる「Kernighan and Ritchie C」(K&R C)、ANSI C の C++ (ARM、日本語版は『注解 C++ リファレンスマニュアル』の C++)、および ISO C++ は、1 つのヘッダーファイル内の同じプログラム要素に対して異なる宣言や定義を規定している場合があります (言語およびバージョンによる違いの詳細は『C++ 移行ガイド』を参照)。これらの標準のすべてに受け入れられるようにヘッダーファイルを設定するには、プリプロセッサマクロの `__STDC__` と `__cplusplus` が存在するかどうか、また、存在する場合はその値が何であるかに応じて、条件付きコンパイルを使用しなければならない場合があります。

マクロ __STDC__ は、K&R C では定義されていませんが、ANSI C と C++ では定義されています。このマクロは、K&R C コードを ANSI C または C++ のコードと分ける際に使用してください。このマクロは、プロトタイプ宣言された関数定義とプロトタイプ宣言されていない関数定義を分けるのに最も便利です。

```
#ifdef __STDC__
int function(char*,...);        // C++ および ANSI C の宣言
#else
int function();                 // K&R C
#endif
```

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

注 – C++ の初期のバージョンでは、マクロ __cplusplus ではなく c_plusplus が定義されていました。マクロ c_plusplus は、現在は定義されていません。

__cplusplus の定義は、C と C++ を分ける際に使用してください。このマクロは、次の例に示すように関数宣言に対する extern "C" インタフェースの指定を保護するのに最適です。extern "C" の指定の矛盾を防ぐため、extern "C" リンケージ指定のスコープ内には #include 指令を入れないでください。

```
#include "header.h"
                                //... 他のインクルードファイル ...
#if defined(__cplusplus)
extern "C" {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

ARM C++ では、__cplusplus マクロの値は 1 です。ISO C++ では、このマクロの値は 199711L (この規格が制定された年月を long 定数として表現したもの) です。このマクロの値は、ARM C++ を ISO C++ と分ける際に使用してください。このマクロの値は、テンプレート構文内の変更を保護するのに最適です。

```
// テンプレート関数の特殊化
#if __cplusplus < 199711L
int power(int,int);                         // ARM C++
#else
template <> int power(int,int);             // ISO C++
#endif
```

べき等なヘッダーファイル

ヘッダーファイルはべき等でなければなりません。つまり、ヘッダーファイルを何度インクルードしても、その効果は 1 度だけインクルードする場合と同じでなければなりません。この特性は、テンプレートを作成する場合に特に重要です。ヘッダーファイルをべき等にするには、ヘッダーファイルの本体が 2 度以上出現することを防ぐプリプロセッサ条件を設定すると最も効果的です。

```
#ifndef HEADER_H
#define HEADER_H
/* ヘッダーファイルの内容 */
#endif
```

自己完結するヘッダーファイル

ヘッダーファイルには、完全なコンパイルに必要な定義がすべて含まれている必要があります。ヘッダーファイルは、必要な定義を含むヘッダーファイルをすべてその中にインクルードし、「自己完結」である、つまり他のファイルを必要としないように設定してください。

```
#include "another.h"
/* another.h に依存する定義 */
```

通常、ヘッダーファイルは、べき等であるとともに自己完結でなければなりません。

```
#ifndef HEADER_H
#define HEADER_H
#include "another.h"
/* another.h に依存する定義 */
#endif
```

不要なヘッダーファイルのインクルード

C++ で書かれたプログラムでは通常 C プログラムよりも宣言の数が多く、そのためコンパイル時間が C プログラムよりも長くなります。宣言の数は、いくつかの手法を使用することにより減らせます。

手法の 1 つは、ヘッダーファイルをべき等にするように定義されたマクロを使用して、ヘッダーファイル自体を条件付きでインクルードするものです。ただしこの手法は、ファイル間の依存を増やします。

```
#ifndef HEADER_H
#include "header.h"
#endif
```

注 – システムヘッダーファイルには、_Xxxx (X は大文字) という書式の識別子が含まれていることがよくあります。これらの識別子は予約されているため、保護のためのマクロでは、この書式の識別子は使用しないでください。

コンパイル時間を減らすには、定義を含むヘッダーファイルをインクルードせずに、不完全なクラスと構造体 (struct) 宣言またはクラス (class) 宣言を使用することもできます。この手法は、完全な定義が不要で、識別子が typedef でも template でもなく、class または struct である場合に使用できます (標準ライブラリは、クラスではなくテンプレートである typedef を多数含む)。たとえば、次のように記述する代わりに、

```
#include "class.h"
a_class* a_ptr;
```

次のように記述します。

```
class a_class;
a_class* a_ptr;
```

(a_class が実際に typedef である場合、この手法は無効です)

もう 1 つの方法として、Erich Gamma 著の『オブジェクト指向における再利用のためのデザインパターン』 (ソフトバンク) に述べられているように、インタフェースクラスとファクトリを使用することもできます。

インライン関数の定義

インライン関数の定義を構成する方法は 2 つあります。その 1 つは定義をインライン展開するもので、もう 1 つは定義を取り込むものです。どちらの手法にも、利点と欠点があります。

関数定義のインライン展開

定義のインライン展開による構成は、メンバー関数にしか使用できません。インライン展開するには、クラス定義内の関数宣言の後に関数の本体を直接入れます。

```
class Class
{
    int method() { return 3; }
};
```

この構成は、関数のプロトタイプの繰り返しを防ぎ、ソースファイルの容量を減らし、不整合が起きる可能性を減らします。しかし、この構成は、通常はインタフェースとして扱われる部分に、実装の詳細を挿入する場合があります。この場合は、関数がインライン展開でなくなったときに、大幅な編集が必要になります。

この構成は、関数の本体がごく少量である (つまり空の中括弧) であるか、あるいは関数が常にインラインである場合にのみ使用してください。

関数定義取り込み型の構成

定義取り込み型の構成は、すべてのインライン関数に使用できます。関数の本体を、プロトタイプの繰り返し (必要な場合) とともに入れてください。関数定義は、ソースファイル内に直接入れることも、ソースファイルとともに取り込むこともできます。

```
class Class {
    int method();
};
inline int Class::method() {
    return 3;
}
```

この構成は、インタフェースと実装が分離されます。このため関数がインラインで実装されなくなったときには、定義をヘッダーファイルからソースファイルに簡単に移動できます。欠点は、この構成ではクラスのプロトタイプが繰り返されることです。この繰り返しのため、ソースファイルの容量が増え、矛盾が発生しやすくなります。

テンプレート定義

テンプレート定義は、2 つの方法で構成できます。1 つは定義取り込み型で、もう 1 つは定義分離型です。定義取り込み型の構成の方が、テンプレートのコンパイルをより広範囲に渡って制御できます。

テンプレート定義取り込み型の構成

テンプレートを使用するファイルの中にテンプレートの宣言と定義が含まれていれば、そのファイルは定義取り込み型の構成です。次にその例を示します。

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

テンプレートを使用するファイルがテンプレートの宣言と定義の両方を含むファイルをインクルードしている場合、そのテンプレートを使用するファイルも定義取り込み型の構成になります。次にその例を示します。

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

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

注 – ここでは、テンプレートヘッダーをべき等にすることが重要です (9 ページの「べき等なヘッダーファイル」を参照)。

テンプレート定義分離型の構成

テンプレート定義を構成するには、次の例に示すように、テンプレート定義ファイルに定義を入れることもできます。

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

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

テンプレート定義ファイルにインクルードするヘッダーファイルは必ずべき等でなければなりません。

実際にはヘッダーファイルのインクルードをまったく必要としないこともよくあります (9 ページの「べき等なヘッダーファイル」を参照)。

注 – テンプレート定義ファイルには一般的にソースファイルの拡張子 (`.c`、`.C`、`.cc`、`.cpp`、`.cxx`) が使用されますが、テンプレート定義ファイルは実際にはヘッダーファイルです。コンパイラは、必要に応じてそれらを自動的にインクルードします。テンプレート定義ファイルは、個別にコンパイルするべきものではありません。

テンプレート宣言とテンプレート定義を別々のファイルに置く場合は、その定義ファイルの構成、名前、置き場所に細心の注意を払う必要があります。定義の置き場所を、コンパイラに対して明示的に知らせることが必要になる場合もあります。テンプレート定義の検索に関する規則については、『C ++ ユーザーズガイド』を参照してください。

第3章

プラグマ

この章では、プラグマについて説明します。「プラグマ」とは、コンパイラに特定の情報を渡すために使用するコンパイラ指令です。プラグマを使用すると、コンパイル内容を詳細に渡って制御できます。たとえば、`pack` プラグマを使用すると、構造体の中のデータの配置を変えることができます。プラグマは「指令」とも呼ばれます。

プリプロセッサキーワード `pragma` は C++ 標準の一部です。ただし、プラグマの書式、内容、および意味はコンパイラごとに異なります。プラグマは C++ 標準には定義されていません。したがってプラグマに依存するコードには移植性はありません。

プラグマの書式

次に、Sun WorkShop C++ コンパイラのプラグマのさまざまな書式を示します。

```
#pragma keyword
#pragma keyword (a [ , a] ... )[, keyword ( a [, a ] ... )] ,...
#pragma sun keyword
```

変数 *keyword* は特定の指令を示し、*a* は引数を示します。

Sun WorkShop C++ コンパイラが認識する一般的なプラグマのキーワードを次に示します。

- `align` - デフォルトを無効にして、パラメータ変数のメモリー境界を、指定したバイト境界に揃えます。
- `init` - 指定した関数を初期化関数にします。

- `fini` - 指定した関数を終了関数にします。
- `ident` - 実行可能ファイルの `.comment` 部に、指定した文字列を入れます。
- `pack (n)` - 構造体オフセットの配置を制御します。*n* の値は、すべての構造体メンバーに合った最悪の場合の境界整列を指定する数字で、0、1、2、4、8 のいずれかにします。
- `unknown_control_flow` - 手続き呼び出しの通常の制御フロー属性に違反するルーチンのリストを指定します。
- `weak` - 弱いシンボル結合を定義します。

プラグマ一覧

この節では、Sun WorkShop C++ コンパイラが認識するプラグマキーワードについて説明します。

#pragma align

`#pragma align` *integer* (*variable* [, *variable*] . . .)

`align` を使用すると、指定したすべての変数 *variable* (変数) のメモリー境界を *integer* (整数) バイト境界に揃えることができます (デフォルト値より優先されます) 。ただし、次の制限があります。

- *integer* は、1 ~ 128 の範囲にある 2 の二乗、つまり、1、2、4、8、16、32、64、128 のいずれかでなければなりません。
- *variable* には、大域変数か静的変数を指定します。局所変数またはクラスメンバー変数は指定できません。
- 指定されたメモリーの境界値がデフォルト値より小さいと、デフォルト値が使用されます。
- この `#pragma` 行は、指定した変数の宣言より前になければなりません。前にないと、この `#pragma` 行は無視されます。

- この #pragma 行で指定されていても、プラグマ行に続くコードの中で宣言されない変数は、すべて無視されます。次に、正しく宣言されている例を示します。

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

#pragma align を名前空間内で使用するときは、符号化された名前を使用する必要があります。たとえば、次のコード中の、#pragma align 文には何の効果もありません。この問題を解決するには、#pragma align 文の a、b、および c を符号化された名前に変更します。

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

#pragma init

#pragma init (*identifier* [,*identifier*]...)

init を使用すると、*identifier* (識別子) を「初期設定関数」にします。この関数は void 型で、引数を持ちません。この関数は、実行開始時にプログラムのメモリーイメージを構築する時に呼び出されます。共有オブジェクトの初期設定子の場合、共有オブジェクトをメモリーに入れるとき、つまりプログラムの起動時または dlopen() のような動的ロード時のいずれかに実行されます。初期設定関数の呼び出し順序は、静的と動的のどちらの場合でもリンカーが処理した順序になります。

ソースファイル内で #pragma init で指定された関数は、そのファイルの中にある静的コンストラクタの後に実行されます。*identifier* は、この #pragma で指定する前に宣言しておく必要があります。

#pragma fini

#pragma fini (*identifier* [,*identifier*]...)

fini を使用すると、*identifier* (識別子) を「終了関数」にします。この関数は void 型で、引数を持ちません。この関数は、プログラム制御によってプログラムが終了する時、または関数内の共有オブジェクトがメモリーから削除されるときに呼び出されます。初期設定関数と同様に、終了関数はリンカーが処理した順序で実行されます。

ソースファイル内で #pragma fini で指定された関数は、そのファイルの中にある静的デストラクタの後に実行されます。*identifier* は、この #pragma で指定する前に宣言しておく必要があります。

#pragma ident

```
#pragma ident string
```

ident を使用すると、実行可能ファイルの .comment 部に、*string* に指定した文字列を記述することができます。

#pragma pack(*n*)

```
# pragma pack ([n])
```

pack は、構造体メンバーの配置制御に使用します。

n を指定する場合、0 であるか 2 の累乗でなければなりません。0 以外の値を指定すると、コンパイラは *n* バイトの境界整列と、データ型に対するプラットフォームの自然境界のどちらか小さい方を使用します。たとえば次の指令は、自然境界整列が 4 バイトまたは 8 バイト境界である場合でも、指令の後 (および後続の pack 指令の前) に定義されているすべての構造体のメンバーを 2 バイト境界を超えないように揃えます。

```
#pragma weak pack(2)
```

n が 0 であるか省略された場合、メンバー整列は自然境界整列の値に戻ります。

n の値がプラットフォームの最も厳密な境界整列と同じかそれ以上の場合には、自然境界整列になります。次の表に、各プラットフォームの最も厳密な境界整列を示します。

表 3-1　プラットフォームの最も厳密な境界整列

プラットフォーム	最も厳密な境界整列
IA	4
SPARC 一般、V7、V8、V8a、V8plus、V8plusa	8
SPARC V9、V9a、V9b	16

`pack` 指令は、次の `pack` 指令までに存在するすべての構造体定義に適用されます。別々の翻訳単位にある同じ構造体に対して異なる境界整列が指定されると、プログラムは予測できない状態で異常終了する場合があります。特に、コンパイル済みライブラリのインタフェースを定義するヘッダーをインクルードする場合は、その前に `pack` を使用しないでください。プログラムコード内では、`pack` 指令は境界整列を指定する構造体の直前に置き、`#pragma pack( )` は構造体の直後に置くことをお勧めします。

SPARC プラットフォーム上で `#pragma pack` を使用して、型のデフォルトの境界整列よりも密に配置するには、アプリケーションのコンパイルとリンクの両方で `-misalign` オプションを指定する必要があります。次の表に、整数データ型のメモリーサイズとデフォルトの境界整列を示します。

表 3-2　メモリーサイズとデフォルトの境界整列 (単位はバイト数)

型	SPARC V8 サイズ、境界整列	SPARC V9 サイズ、境界整列	IA サイズ、境界整列
bool	1、1	1、1	1、1
char	1、1	1、1	1、1
short	2、2	2、2	2、2
wchar_t	4、4	4、4	4、4
int	4、4	4、4	4、4
long	4、4	8、8	4、4
float	4、4	4、4	4、4
double	8、8	8、8	8、4

表 3-2　メモリーサイズとデフォルトの境界整列 (単位はバイト数) (続き)

型	SPARC V8 サイズ、境界整列	SPARC V9 サイズ、境界整列	IA サイズ、境界整列
long double	16、8	16、16	12、4
データへのポインタ	4、4	8、8	4、4
関数へのポインタ	4、4	8、8	4、4
メンバーデータへのポインタ	4、4	8、8	4、4
メンバー関数へのポインタ	8、4	16、8	8、4

#pragma unknown_control_flow

```
#pragma unknown_control_flow (name,[,name]...)
```

unknown_control_flow を使用すると、手続き呼び出しの通常の制御フロー属性に違反するルーチンの名前のリスト *name*,[,*name*]... を指定できます。たとえば、setjmp() の直後の文は、他のどんなルーチンを呼び出してもそこから返ってくることができます。これは、longjmp() を呼び出すことによって行います。

このようなルーチンを使用すると標準のフローグラフ解析ができないため、呼び出す側のルーチンを最適化すると安全性が確保できません。このような場合に #pragma unknown_control_flow を使用すると安全な最適化が行えます。

#pragma weak

```
#pragma weak name1 [=name2]
```

name1 と *name2* はシンボル名を示します。

weak を使用すると、弱い (weak) 大域シンボルを定義できます。このプラグマは主にソースファイルの中でライブラリを構築するために使用されます。リンカーは弱いシンボルを認識できなくてもエラーメッセージを出しません。

weak プラグマは、次の 2 つの書式でシンボルを指定できます。

- **文字列**

文字列は、C++ の変数または関数の符号化された名前でなければなりません。無効な符号化名が指定された場合、その名前を参照したときの動作は予測できません。無効な符号化名を参照した場合、バックエンドがエラーを生成するかどうかは不明です。エラーを生成するかどうかに関わらず、無効な符号化名を参照したときのバックエンドの動作は予測できません。

- **識別子**

  識別子は、コンパイル単位内であらかじめ宣言された C++ の関数のあいまいでない識別子でなければなりません。識別子書式は変数には使用できません。無効な識別子への参照を検出した場合、フロントエンド (`ccfe`) はエラーメッセージを生成します。

`#pragma weak` *name*

name はシンボル名を示します。

`#pragma weak` *name* という書式の指令は、*name* を弱い (`weak`) シンボルに定義します。*name* のシンボル定義が見つからなくても、リンカーはエラーメッセージを生成しません。また、弱いシンボルの定義を複数見つけた場合でも、リンカーはエラーメッセージを生成しません。リンカーは単に最初に検出した定義を使用します。

他のコンパイル単位に関数または変数の強い (strong) 定義が存在する場合、*name* はその定義にリンクされます。*name* の強い定義が存在しない場合、リンカーはシンボルの値を 0 にします。

次の指令は、`ping` を弱いシンボルに定義しています。`ping` という名前のシンボルの定義が見つからない場合でも、リンカーはエラーメッセージを生成しません。

```
#pragma weak ping
```

`#pragma weak` *name1* = *name2*

`#pragma weak` *name1* = *name2* という書式の指令は、シンボル *name1* を *name2* への弱い参照として定義します。*name1* がどこにも定義されていない場合、*name1* の値は *name2* の値になります。*name1* が別の場所で定義されている場合、リンカーはその定

義を使用し、*name2* への弱い参照は無視します。次の指令では、bar がプログラムのどこかで定義されている場合、リンカーはすべての参照先を bar に設定します。そうでない場合、リンカーは bar への参照を foo にリンクします。

```
#pragma weak bar = foo
```

識別子書式では、*name2* は現在のコンパイル単位内で宣言および定義しなければなりません。次に例を示します。

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

文字列書式を使用する場合、シンボルはあらかじめ宣言されている必要はありません。次の例において、_bar と bar の両方が extern "C" である場合、その関数はあらかじめ宣言されている必要はありません。しかし、bar は同じオブジェクト内で定義されている必要があります。

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

関数の多重定義

識別子書式を使用するとき、プラグマのあるスコープ中には指定した名前を持つ関数は、1 つしか存在してはなりません。多重定義された関数を使用して #pragma weak の識別子書式を使用しようとするとエラーになります。次に例を示します。

```
int bar(int);
float bar(float);
#pragma weak bar         // あいまいな関数名により、エラー
```

このエラーを回避するには、文字列書式を使用します。次に例を示します。

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // float bar(int) を弱いシンボルにする
```

詳細は、『リンカーとライブラリ』を参照してください。

第4章

テンプレート

テンプレートの目的は、プログラマが一度コードを書くだけで、そのコードが型の形式に準拠して広範囲の型に適用できるようにすることです。この章では関数テンプレートに関連したテンプレートの概念と用語を紹介し、より複雑な (そして、より強力な) クラステンプレートと、テンプレートの使用方法について説明しています。また、テンプレートのインスタンス化、デフォルトのテンプレートパラメータ、およびテンプレートの特殊化についても説明しています。この章の最後には、テンプレートの潜在的な問題が挙げられています。

関数テンプレート

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

関数テンプレートの宣言

テンプレートは使用する前に宣言しなければなりません。次の例に見られるように、「宣言」によってテンプレートを使用するのに十分な情報は与えられますが、テンプレートの実装には他の情報も必要です。

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

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

関数テンプレートの定義

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

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

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

Sun WorkShop 6 C++ は、関数パラメータリスト内にテンプレートの型名でないパラメータを含む式をサポートしていません。次に例を示します。

```
// 関数パラメータリスト中にテンプレートの型名でない
// パラメータを含む式は、サポートされません。
template<int I> void foo( mytype<2*I> ) { ... }
template<int I, int J> void foo( int a[I+J] ) { ... }
```

関数テンプレートの使用

テンプレートは、いったん宣言すると他のすべての関数と同様に使用することができます。テンプレートを「使用」するには、そのテンプレートの名前とテンプレート引数を指定します。コンパイラは、テンプレート型引数を、関数引数の型から推測します。たとえば、以前に宣言されたテンプレートを次のように使用できます。

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

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

```
template<class T> T func(); // 関数引数なし
int k = func<int>(); // テンプレート引数を明示的に指定
```

クラステンプレート

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

クラステンプレートの宣言

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

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

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

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

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

クラステンプレートの定義

クラステンプレートの定義では、次の例のようにクラスデータと関数メンバーを宣言しなければなりません。

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

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

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

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

クラステンプレートメンバーの定義

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

関数メンバーの定義

テンプレート関数メンバーの定義は、テンプレートパラメータの指定と、それに続く関数定義から構成されます。関数識別子は、クラステンプレートのクラス名とそのテンプレートの引数で修飾されます。次の例は、`template <class Elem>` というテンプレートパラメータ指定を持つ `Array` クラステンプレートの 2 つの関数メンバー定義を示しています。それぞれの関数識別子は、テンプレートクラス名とテンプレート引数 `Array<Elem>` で修飾されています。

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

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

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

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

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

静的データメンバーの定義

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

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

クラステンプレートの使用

テンプレートクラスは、型が使用できる場所ならどこででも使用できます。テンプレートクラスを指定するには、テンプレート名と引数の値を設定します。次の宣言例では、Array テンプレートに基づいた変数 int_array を作成します。この変数のクラス宣言とその一連のメソッドは、Elem が int に置き換わっている点以外は、Array テンプレートとまったく同じです (28 ページの「テンプレートのインスタンス化」を参照)。

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

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

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

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

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

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

テンプレートのインスタンス化

テンプレートの「インスタンス化」には、特定の組み合わせのテンプレート引数に対応した具体的なクラスまたは関数 (「インスタンス」) を生成することが含まれます。たとえば、コンパイラは *Array<int>* と *Array<double>* に対応した別々のクラスを生成します。これらの新しいクラスの定義では、テンプレートクラスの定義の中のテンプレートパラメータがテンプレート引数に置き換えられます。前述の「クラステンプレート」の節に示す *Array<int>* の例では、すべての *Elem* が *int* に置き換えられます。

テンプレートの暗黙的インタンス化

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

全クラスインスタンス化

コンパイラは、あるテンプレートクラスを暗黙的にインスタンス化するとき、使用されるメンバーだけをインスタンス化します。コンパイラがあるクラスを暗黙的にインスタンス化するときにすべてのメンバー関数をインスタンス化するには、コンパイラオプションの `-template=wholeclass` を使用します。このオプションを無効にするには、
`-template=no%wholeclass` を指定します。

テンプレートの明示的インスタンス化

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

テンプレート関数の明示的インスタンス化

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

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

テンプレート引数は、コンパイラが推測できる場合は省略できます。

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

テンプレートクラスの明示的インスタンス化

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

```
template class Array<char>;
```

```
template class String<19>;
```

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

テンプレートクラス関数メンバーの明示的インスタンス化

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

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

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

テンプレートクラスの静的データメンバーの明示的インスタンス化

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

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

テンプレートの編成

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

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

そして、次のように定義できます。

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

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

```
Array<int> int_array( 100 );   // intの配列を作成し、
sort( int_array );             // それをソートする。
```

デフォルトのテンプレートパラメータ

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

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

テンプレートパラメータにデフォルト値を指定する場合、それに続くパラメータもすべてデフォルト値でなければなりません。テンプレートパラメータに指定できるデフォルト値は 1 つです。

テンプレートの特殊化

次の twice の例のように、テンプレート引数を例外的に特定の形式で組み合わせると、パフォーマンスが大幅に改善されることがあります。あるいは、次の sort の例のように、テンプレート記述がある引数の組み合わせに対して適用できないこともあ

ります。テンプレートの特殊化によって、実際のテンプレート引数の特定の組み合わせに対して代替実装を定義することが可能になります。テンプレートの特殊化はデフォルトのインスタンス化を無効にします。

テンプレートの特殊化宣言

前述のようなテンプレート引数の組み合わせを使用するには、その前に特殊化を宣言しなければなりません。次の例は `twice` と `sort` の特殊化された実装を宣言しています。

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

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

コンパイラがテンプレート引数を明確に確認できる場合には、テンプレート引数を省略することができます。次にその例を示します。

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

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

テンプレートの特殊化定義

宣言するテンプレート特殊化はすべて定義しなければなりません。次の例は、前の節で宣言された関数を定義しています。

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

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

テンプレートの特殊化の使用とインスタンス化

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

部分特殊化

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

```
template<class T, class U> class A { ... }; // 本来のテンプレート
template<> class A<int, double> { ... };    // 特殊化
```

次のコード例に、本来のテンプレートの部分特殊化を示します。

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

- 例 1 は、最初のテンプレートパラメータが `int` 型である特殊なテンプレート定義です。

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

テンプレートの問題

この節では、テンプレートを使用する場合の問題について説明しています。

非局所型名前の解決とインスタンス化

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

名前の解決は複雑です。したがって、汎用性の高い標準的な環境で提供されているもの以外は、非局所型名前に依存することは避ける必要があります。言い換えれば、どこでも同じように宣言され、定義されている非局所型名前だけを使用するようにしてください。この例では、テンプレート関数の `converter` が、非局所型名前である `intermediary` と `temporary` を使用しています。これらの名前は `use1.cc` と

use2.cc では異なる定義を持っているため、コンパイラが異なれば結果は違うものになるでしょう。テンプレートが正しく機能するためには、すべての非局所型名前 (intermediary と temporary) がどこでも同じ定義を持つ必要があります。

use_common.h

```
// 共通のテンプレート定義
template <class Source, class Target>
Target converter( Source source )
        { temporary = (intermediary)source;
        return (Target)temporary; }
```

use1.cc

```
typedef int intermediary;
int temporary;

#include "use_common.h"
```

use2.cc

```
typedef double intermediary;
unsigned int temporary;

#include "use_common.h"
```

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

テンプレート引数としての局所型

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

コード例 4-1 テンプレート引数としての局所型の問題の例

array.h

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

array.cc

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

file1.cc

```
#include "array.h"
struct Foo { int data; };
Array<Foo> File1Data;
```

file2.cc

```
#include "array.h"
struct Foo { double data; };
Array<Foo> File2Data;
```

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

テンプレート関数のフレンド宣言

テンプレートは、使用前に宣言されていなければなりません。フレンド宣言では、テンプレートを宣言するのではなく、テンプレートの使用を宣言します。フレンド宣言の前に、実際のテンプレートが宣言されていなければなりません。次の例では、作成

済みオブジェクトファイルをリンクしようとするときに、operator<< 関数が未定義であるというエラーが生成されます。その結果、operator<< 関数はインスタンス化されません。

コード例 4-2 フレンド宣言の問題の例

array.h

```
// operator<< 関数に対して未定義エラーを生成する
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

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

array.cc

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

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

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

main.cc

```
#include <iostream>
#include "array.h"

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

コンパイラは、次の宣言を array クラスの friend である正規関数の宣言として読み取っているので、コンパイル中にエラーメッセージを表示しません。

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

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

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

// 次の 2 行は operator<< をテンプレート関数として宣言する
template<class T> class array;
template<class T>
std::ostream& operator<<(std::ostream&, const array<T>&);

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

テンプレート定義内での修飾名の使用

C++ 標準は、テンプレート引数に依存する修飾名を持つ型を、typename キーワードを使用して型名として明示的に示すことを規定しています。これは、それが型であることをコンパイラが認識できる場合も同様です。次の例の各コメントは、それぞれの修飾名が typename キーワードを必要とするかどうかを示しています。

```
struct simple {
    typedef int a_type;
    static int a_datum;
};
int simple::a_datum = 0; // 型ではない
template <class T> struct parametric {
    typedef T a_type;
    static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;      // 型ではない
template <class T> struct example {
    static typename T::a_type variable1;              // 必要
    static typename parametric<T>::a_type variable2;  // 必要
    static simple::a_type variable3;                  // 不要
};
template <class T> typename T::a_type                 // 必要
                   example<T>::variable1 = 0;         // 型ではない
template <class T> typename parametric<T>::a_type     // 必要
                   example<T>::variable2 = 0;         // 型ではない
template <class T> simple::a_type                     // 不要
                   example<T>::variable3 = 0;         // 型ではない
template class example<simple>
```

テンプレート宣言の入れ子

「>>」という文字を持つものは右シフト演算子と解釈されるため、あるテンプレート宣言を別のテンプレート宣言内で使用する場合は注意が必要です。隣接する「>」文字との間に、少なくとも 1 つの空白文字を入れるようにしてください。

以下に誤った書式の例を示します。

```
// 誤った書式の文
Array<String<10>> short_string_array(100); // >> は右シフトを示す。
```

上記の文は、次のように解釈されます。

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

正しい構文は次のとおりです。

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

第5章

例外処理

本章では、Sun C++ コンパイラに現在実装されている例外処理、および C++ 国際規格の規定について説明します。

例外処理に関する追加情報については、『注解 C++ リファレンス・マニュアル』(Margaret A. Ellis、Bjarne Stroustrup 共著、トッパン刊) を参照してください。

例外処理とは

例外とは、プログラムの通常の流れの中で発生し、プログラムの継続を阻止する変則性のことです。これらの変則性 (ユーザーエラー、論理エラーまたはシステムエラー) は、関数で検出できます。変則性を検出した関数がその変則性に対処できない場合は、例外を送出し、例外を処理する関数がそれを捕獲します。

C++ では、例外が送出されたときには、これを無視することはできません。つまり、何らかの通知をするか、プログラムを停止しなければなりません。ユーザーによって作成された例外ハンドラが存在しない場合は、プログラムはコンパイラのデフォルト機構によって強制終了します。

例外処理は、ループや `if` 文のような、プログラムの通常のフロー制御に比べると手間がかかります。そのため、例外機構は通常の動作の処理ではなく、実際に異常と認められる状況でのみ使用するようにしてください。

例外は、局所的に処理できない状況を処理する場合に特に便利です。プログラム全体にエラー状態を伝えるのではなく、エラーを処理できる場所へ直接制御を移すことができます。

たとえば、ファイルを開き、いくつかの関連データを初期化する処理が、ある関数に与えられているとします。ファイルが開けないか壊れている場合、この関数は処理を実行できません。このような問題を処理するための機能が関数に十分与えられていな

い場合、関数は問題を示す例外オブジェクトを送出し、プログラムの前方へ制御を移すことができます。例外ハンドラは、自動的にファイルのバックアップを行う、ユーザーに対して他のファイルで試すか尋ねる、プログラムを正常に停止する、などの処理を行えます。例外ハンドラの指定がないと、状態とデータを関数呼び出しの階層の全体に渡し、関数呼び出しごとに状態の検査を行う必要があります。例外ハンドラを指定する場合、エラー検査によって制御フローがわかりにくくなることはありません。関数が戻る場合、呼び出し元はその関数が正常に終了したと見なせます。

例外ハンドラにはいくつか欠点があります。関数またはその関数が呼び出す他の関数が例外を送出したため関数が戻らない場合には、データが矛盾した状態のままになることがあります。プログラマは、例外が送出される可能性があるのはいつか、例外がプログラムの状態に悪い影響を与えるかどうかを把握する必要があります。

マルチスレッド環境で例外を使用する方法については、79 ページの「マルチスレッドプログラムでの例外の使用」を参照してください。

例外処理キーワードの使用

C++ の例外ハンドラには次の 3 つのキーワードがあります。

- `try`
- `catch`
- `throw`

try

`try` ブロックとは、例外が発生する可能性のある、中括弧 `{ }` で囲まれた C++ 文の集まりです。このグループ化のため、例外ハンドラは `try` ブロック内で生成された例外だけを扱うことができます。各 `try` ブロックには、対応する `catch` ブロックが 1 つ以上存在します。

catch

`catch` ブロックとは、特別に送出された例外を処理するために使用される C++ 文の集まりです。複数の `catch` ブロック (つまりハンドラ) が `try` ブロックの後に置かれます。`catch` ブロックは次の項目からなります。

1. キーワード catch
2. try ブロックから送出される可能性のある例外の型に対応した、括弧 ( ) に囲まれた catch パラメータ
3. 例外を処理するための、中括弧 { } で囲まれた文の集まり

throw

throw 文は、次の例外ハンドラに例外とその値を送出するために使用されます。通常の throw ブロックは、キーワード throw と式から構成されます。式の結果の型によって、どの catch ブロックに制御が移るかが決まります。catch ブロック内では、現在の例外と値は throw キーワードだけ (式は不要) で再送出できます。

この例では、try ブロック中の関数呼び出しは f() に制御を渡します。f() は Overflow 型の例外を送出します。この例外は、Overflow 型の例外を処理する catch ブロックによって処理されます。

```
class Overflow {
                        // ...
public:
   Overflow(char,double,double);
};

void f(double x)
{
                        // ...
   throw Overflow('+',x,3.45e107);
}

int main() {
    try {
                        // ...
        f(1.2);
                        // ...
    }
    catch(Overflow& oo) {
                        // Overflow 型の例外をここで処理する
    }
}
```

例外ハンドラの実装

例外ハンドラを実装する場合の基本作業を次に示します。

- 他の多くの関数から呼び出されている関数は、エラーが検出されるたびに例外が送出されるようコーディングしてください。通常、この `throw` 式はオブジェクトを 1 つ送出します。このオブジェクトは、例外の型を識別し、送出された例外に関する固有の情報を渡す際に使用します。
- その関数を使用するプログラム中で `try` 文を使用して例外に備えてください。例外が発生すると思われる関数呼び出しを `try` ブロック内で中括弧で囲んでください。
- `try` ブロックのすぐ後に、`catch` ブロックを 1 つ以上記述してください。
  各 `catch` ブロックは、受け取ることのできるオブジェクトの型またはクラスを識別します。オブジェクトが例外によって送出されると、次のことが行われます。
  - 例外により送出されるオブジェクトが `catch` 式の型と一致する場合は、この `catch` ブロックに制御が移ります。
  - 例外により送出されるオブジェクトが先頭の `catch` ブロックと一致しない場合は、以降の `catch` ブロックから、型の一致するものが順次検索されます。51 ページの「例外とハンドラの一致」を参照してください。
  - 現在のスコープに送出された例外に対応する `catch` ブロックが存在しない場合、制御は現在のスコープ外に移り、そのスコープ内に定義されている自動 (局所的な非静的) オブジェクトはすべて破棄されます。次に、(関数のスコープである可能性がある) 周囲のスコープが対応するハンドラを持っているかどうかが検査されます。この処理は、対応する `catch` ブロックを持つスコープが見つかるまで続けられます。対応する `catch` ブロックが関数 `main` に到達するまでに見つかった場合、その `catch` ブロックに制御が移ります。
  - 対応する `catch` ブロックがない場合は、プログラムは事前定義済みの関数 `terminate()` を呼び出して正常終了します。`terminate()` はデフォルトで `abort()` を呼び出し、`abort()` は残っているオブジェクトをすべて破棄してプログラムを終了します。`set_terminate()` 関数を呼び出すと、このデフォルトの動作を変えることができます。

同期例外処理

例外処理は、配列の範囲検査などの同期例外だけをサポートするように設計されています。同期例外という言葉は、例外が `throw` 式からのみ発生することを意味します。

C++ 標準では、終端モデルを使用した同期例外処理をサポートしています。終端とは、一度例外が送出されたらその場所には制御が戻らないことを意味します。

非同期例外処理

例外処理は、キーボード割り込みなどの非同期例外を直接処理するようには設計されていません。ただし、注意して行えば、非同期イベントの場合にも動作するように設定することもできます。たとえば、例外処理をシグナルと一緒に動作させるには、大域変数を設定し、この変数の値を定期的な間隔でポーリングして値が変化したときに例外を送出するようなシグナルハンドラを作成します。シグナルハンドラからは例外を送出できません。

制御の流れの管理

C++ では、例外ハンドラは例外を訂正してその例外の発生場所に戻ることはしません。その代わり例外が発生すると、制御は例外を送出した関数から抜け、続いて例外を予期していた `try` ブロックから抜け、その例外と例外宣言が一致する `catch` ブロックに移ります。

この `catch` ブロックが例外を処理します。`catch` ブロックは、同じ例外を再送出するか、別の例外を送出するか、ラベルにジャンプするか、関数から戻るか、あるいは正常に終了します。`catch` ブロックが `throw` なしで正常に終了した場合、制御の流れは後続のすべての (`try` ブロックに関連付けられた) `catch` ブロックを飛び越えます。

例外が送出および捕獲され、その例外を送出した関数の外に制御が移ると、「スタックの巻き戻し」が実行されます。スタックの巻き戻しを行なっている間、終了したブロックのスコープ内で生成された自動オブジェクトは、そのデストラクタの呼び出しによって安全に破棄されます。

`try` ブロックが例外なしで終了した場合、関連するすべての `catch` ブロックは無視されます。

注 – 例外ハンドラは、 `return` 文を使用してエラーの発生した場所へ制御を戻すことはできません。この場合、発行された `return` 文はその `catch` ブロックが入っている関数から戻ります。

`try` ブロックとハンドラからの分岐

`try` ブロックやハンドラから外への分岐は許可されています。しかし、`catch` ブロックの中への分岐は、例外の開始を飛び越すことに等しいので許可されていません。

例外の入れ子

他の例外が処理されていない間に別の例外を送出することを、例外の入れ子と呼びます。例外の入れ子は、特定の状況でしか行えません。例外が送出される位置から一致する `catch` 節の入力位置までは、例外は処理されません。この間に呼び出される関数 (破棄される自動オブジェクトのデストラクタなど) は、例外が関数を回避しないかぎり、新しい例外を送出できます。他の例外が処理されていない間に例外によって関数が終了すると、その直後に `terminate()` 関数が呼び出されます。

例外ハンドラがいったん入力されると例外は処理済みと見なされ、例外が再び送出できるようになります。

送出されたまま未処理の状態にある例外は `uncaught_exception()` 関数で確認できます。50 ページの「uncaught_exception() 関数の呼び出し」を参照してください。

送出する例外の指定

関数宣言には、例外指定を 1 つ含めることができます。例外指定とは、関数が直接的にまたは間接的に送出する可能性のある例外のことです。

次の 2 つの宣言は、関数 `f1` は例外を生成し、その例外は `X` 型のハンドラが受け取ることおよび、型 `W`、`Y`、または `Z` のハンドラによって捕獲できる例外だけを関数 `f2` が生成することを伝えています。

```
void f1(int) throw(X);
void f2(int) throw(W,Y,Z);
```

上記の例を少し変えて書くと、次のようになります。

```
void f3(int) throw(); // 空の括弧
```

このように定義すると、関数 f3 は例外を 1 つも生成しなくなります。例外指定で許可されていない例外によって関数が終了する場合、事前に定義済みの関数 unexpected() が呼び出されます。デフォルトでは unexpected() は、プログラムを終了させる terminate() を呼び出します。このデフォルト動作は set_unexpected() 関数を呼び出すことで変更できます。48 ページの「terminate() と unexpected() 関数の変更」を参照してください。

予期しない例外は、コンパイル時ではなくプログラムの実行時に検査されます。許可されていない例外が送出されそうな場合でも、実行時にその例外が実際に送出されないかぎりエラーは出力されません。

しかしコンパイラは、場合によっては不必要な検査を省くことができます。

たとえば次の例では、f は検査されません。

```
void foo(int) throw(x);
void f(int) throw(x);
{   foo(13);
}
```

例外を指定しておかないと、あらゆる例外が送出される可能性があります。

実行時のエラーの指定

例外に関連する実行時エラーメッセージには、次のものがあります。

- 例外処理のハンドラがありません
- 予期しない例外を送出
- 例ハンドラは例外の再送出しかできません
- スタックの巻き戻し中は、デストラクタは独自の例外を処理しなければなりません
- メモリーが足りません

実行時にエラーを検出すると、その例外の型と上記の 5 つのメッセージの 1 つがエラーメッセージとして表示されます。デフォルトでは、その後で事前定義済み関数の terminate() が呼び出されます。terminate() は abort() を呼び出します。

コンパイラは、コード生成を最適化する時に、例外指定で提供された情報を利用します。たとえば、例外を送出しない関数は最適化の対象から外されます。また、関数の例外指定に対する実行時検査はできる限り省略されます。このため、正しく例外が指定された関数を宣言することによって、コード生成の効率が向上します。

terminate() と unexpected() 関数の変更

次に、set_terminate() と set_unexpected() を使用して terminate() 関数と unexpected() 関数の動きを変更する方法について説明します。

マルチスレッド環境で例外を使用する方法については、79 ページの「マルチスレッドプログラムでの例外の使用」を参照してください。

set_terminate()

terminate() のデフォルトの動作は、次のように関数 set_terminate() を呼び出すことによって変更できます。マルチスレッド環境で例外を使用する方法については、79 ページの「マルチスレッドプログラムでの例外の使用」を参照してください。

```
// 宣言は標準ヘッダー <exception> に含まれる
namespace std {
   typedef void (*terminate_handler)();
   terminate_handler  set_terminate(terminate_handler f) throw();
   void terminate();
}
```

terminate() 関数は、次のような場合に呼び出されます。

- 例外処理機構がユーザー関数 (自動オブジェクトのデストラクタを含む) を呼び出したがその関数が未捕獲の例外を残したまま、別の未捕獲の例外のために終了した
- 送出された例外のハンドラを例外処理機構が見つけられない

- 非局所的なオブジェクトが静的なオブジェクトとして存在している間に、その生成または破棄が、例外により終了した
- atexit() で登録された関数の実行が例外により終了した
- オペランドを持たない throw 式が例外を再送出しようとしたが、例外は現在処理されていない
- unexpected()関数以前に違反のあった例外指定が許可しない例外を送出したが、std::bad_exception がその例外指定に含まれていない
- unexpected() のデフォルト版が呼び出されている

terminate() は set_terminate() に引数として渡された関数を呼び出します。このような関数はパラメータを持たず、値を返すこともなく、プログラム (または現在のスレッド) を必ず停止します。set_terminate() への最後の呼び出しで渡された関数が呼び出され、最後に呼び出された set_terminate() に引数として渡された以前の関数が戻り値になります。そのため、terminate() を使用して、今までに登録された関数を順次呼び出すようにプログラミングすることができます。
terminate() のデフォルトの関数は、メインスレッドに対して abort() を呼び出し、他のスレッドに対して thr_exit() を呼び出します。thr_exit() はスタックを巻き戻したり、自動オブジェクトに対する C++ デストラクタを呼び出すことはありません。

注 – terminate() 以外の方法を使用する場合、呼び出し元に戻ってはなりません。

---

set_unexpected()

unexpected() のデフォルトの動作は、関数 set_unexpected() を呼び出すことによって変更できます。

```
// 宣言は標準ヘッダー <exception> に含まれる
namespace std {
   class exception;
   class bad_exception;
   typedef void (*unexpected_handler)();
   unexpected_handler
      set_unexpcted(unexpected_handler f) throw();
   void unexpected();
}
```

unexpected() 関数は、関数がその例外指定にない例外によって終了しようとする場合に呼び出されます。unexpected() のデフォルト版は、terminate() を呼び出します。

ユーザーが変更した unexpected() は、例外指定が許可している例外も送出することがあります。このような場合の例外処理は、その例外が実際に元の関数から送出されたかのように継続します。変更後の unexpected() がそれ以外の例外を送出した場合は、その例外は標準の例外 std::bad_exception に置換されます。元の関数の例外指定が std::bad_exception を許可しない場合は、直後に関数 terminate() が呼び出されます。それ以外では、元の関数が実際に std::bad_exception を送出したかのように例外処理が継続します。

unexpected() は set_unexpected() に引数として渡された関数を呼び出します。このような関数は、パラメータを持たず、値を返すこともありません。このような関数はその呼び出し元に戻ってはなりません。set_unexpected() への最後の呼び出しで渡された関数が呼び出されるようになります。以前の set_unexpected() の呼び出し時に引数として渡された関数が戻り値になります。そのため set_unexpected() を使用し、今までに登録された関数を順次呼び出すようにプログラミングすることができます。

注 – unexpected() 以外の方法を使用する場合、呼び出し元に戻ってはなりません。

---

uncaught_exception() 関数の呼び出し

捕獲されていない (アクティブな) 例外とは、送出されたままハンドラにまだ受け付けられていない例外を意味します。関数 uncaught_exception() は、捕獲されていない例外が存在する場合は true を返し、存在しない場合は false を返します。

ある例外が捕獲されないために関数が終了し、他の例外がまだアクティブな状態のままプログラムが停止してしまうことがあります。uncaught_exception() 関数は、このような問題を防ぐために役立ちます。この問題は、スタックの巻き戻しの間に呼び出されたデストラクタが例外を送出する時に最も多く発生します。対策としては、デストラクタ内で例外を送出する前に uncaught_exception() が false を返すように設定します (また、デストラクタが例外を送出しないですむようにプログラムを設計しても、このような停止を防ぐことができます)。

例外とハンドラの一致

次のいずれかにあてはまる場合に、T 型のハンドラは E 型の throw と一致します。

- T が E と同じ型である
- T が、E の const か volatile である
- E が、T の const か volatile である
- T が E の参照か、E が T の参照である
- T が E の公開基底クラスである
- T と E の両方ともポインタ型で、かつ E は標準のポインタ変換を使用して T に変換できる

参照やポインタ型の例外を送出すると、「ポインタのからまり」という問題が発生する可能性があります。これは、例外処理が完了する前にポインタの宛先または参照先のオブジェクトが破棄された場合に起こります。オブジェクトが送出される場合、コピーコンストラクタによりオブジェクトのコピーが必ず作成され、このコピーが catch ブロックに渡されます。そのため、局所的なオブジェクトまたは一時的なオブジェクトを送出しても安全です。

(X) 型と (X&) 型の両方のハンドラとも X 型の例外と一致しますが、意味は異なります。(X) 型のハンドラを使用すると、そのオブジェクトのコピーコンストラクタを (再び) 起動することになり、そのオブジェクトを切り捨てる可能性があります。ハンドラの型から派生した型のオブジェクトが送出される場合、オブジェクトは切り捨てられます。そのため、通常は参照によりクラスオブジェクトを捕獲する方が実行速度が速くなります。

try ブロックのハンドラは現われる順序で使用されます。派生クラスのハンドラを確実に起動するには、派生クラスのハンドラ (または派生クラスの参照へのポインタ) を基底クラスのハンドラより前に置いてください。

例外におけるアクセス制御の検査

コンパイラは、例外のアクセス制御に関して次の検査を行います。

- `catch` 節の仮引数が、`catch` 節のある関数の引数と同じ規則に従っているか
- `throw` による送出の起きた関数のコンテキストでオブジェクトがコピーされたり破棄されることがある場合、オブジェクトが送出されるか

現在では、アクセス制御は照合に影響を与えません。

51 ページの「例外とハンドラの一致」に記載した照合規則以外、他のアクセス制御は実行時に検査されません。

`try` ブロック内に関数を入れる

クラス `T` の基底クラスまたはメンバーのコンストラクタが例外によって終了した場合、`T` コンストラクタがその例外を検出または処理する方法は通常ありません。例外は、`T` コンストラクタの本体に入る前 (つまり `T` 内の `try` ブロックに入る前) に送出されることになります。

C++ には、`try` ブロック内に関数全体を入れる新機能があります。通常の関数の場合、その効果は関数の本体を `try` ブロック内に入れることと変わりません。しかし、コンストラクタの場合、こうすることで `try` ブロックはコンストラクタのクラスの基底クラスとメンバーの初期設定子を回避する例外をすべてトラップするようになります。関数全体が `try` ブロックで囲まれる場合、そのブロックは「関数 `try` ブロック」と呼ばれます。

次の例では、基底クラス `B` またはメンバー `E` のコンストラクタから送出される例外はすべて T コンストラクタの本体に入る前に捕獲され、一致する `catch` ブロックにより処理されます。

catch ブロックは関数の外にあるため、関数 try ブロックのハンドラ内で return 文は使用できません。exit() を呼び出して例外を送出するか、あるいは terminate() を呼び出してプログラムを停止するしかありません。

```
class B { ... };
class E { ... };
class T : public B {
public:
    T();
private:
    E e;
};
T::T()
try  : B(args), e(args)
{
    ... // コンストラクタの本体
}
catch( X& x ) {
   ... // 例外 X を処理する
}
catch( ... ) {
   ... // 他の例外を処理する
}
```

例外を無効にする

プログラム内で例外を使用しないことがわかっている場合は、コンパイラオプション -features=no%except を使用して、例外処理を行うためのコードが生成されないように設定できます。このオプションを使用すると、コードのサイズが幾分小さくてすむほか、コードの実行が速くなります。しかし、例外を無効にしてコンパイルされたファイルが例外を使用するファイルにリンクされる場合は、例外を無効にしてコンパイルされたファイル内の一部の局所的なオブジェクトは例外発生時に破棄されません。デフォルトでは、コンパイラは例外処理を行うためのコードを生成します。時間と容量のオーバーヘッドが重要でないかぎり、例外を有効にすることをお勧めします。

実行時関数と事前定義された例外の使用

標準ヘッダー <exception> には、C++ 標準に規定されたクラスおよび例外に関連する関数が含まれています。このヘッダーにアクセスできるのは、標準モードで (コンパイラのデフォルトモード、あるいはオプション -compat=5 を使用して) コンパイルする場合だけです。次に、<exception> ヘッダーファイル宣言を示します。

```
// 標準ヘッダー <exception>
namespace std {
    class exception {
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    };
    class bad_exception: public exception { ... };
    // 予期されない例外処理
    typedef void (*unexpected_handler)();
    unexpected_handler
        set_unexpected(unexpected_handler) throw();
    void unexpected();
    // 停止処理
    typedef void (*terminate_handler)();
    terminate_handler set_terminate(terminate_handler) throw();
    void terminate();
    bool uncaught_exception() throw();
}
```

標準クラス exception は、選択されている言語構造または C++ 標準ライブラリによって送出されるすべての例外の基底クラスです。 exception 型のオブジェクトについては、例外を生成することなく生成、コピー、破棄が可能です。仮想メンバー関数 what() は、例外を説明する文字列を返します。

C++ リリース 4.2 で使用される例外との互換性を保つため、標準モードで使用できるヘッダー <exception.h> も用意されています。このヘッダーファイルには、標準 C++ コードへの移行のために、標準の C++ の一部ではない宣言も含まれています。開発スケジュールが許す場合は、<exception.h> ではなく <exception> を使用して C++ 標準に準拠するようにコードを更新してください。

```
// 移行のために使用されるヘッダー <exception.h>
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

互換モード (-compat[=4]) では、ヘッダー <exception> は使用できません。このモードでは、ヘッダー <exception.h> は C++ リリース 4.2 が提供するものと同じヘッダーを参照します。このヘッダーはここでは掲載していません。

シグナルによる例外と setjmp/longjmp の混在

例外どうしの間に関連性がない限り、例外が発生するプログラムでも setjmp/longjmp を使用できます。

例外と setjmp/longjmp に対する規則は、これらを別々に使用する場合とまったく同様に適用できます。さらに、ポイント A からポイント B への longjmp が有効であるのは、ポイント A から送出されてポイント B で捕獲される例外が同じ効果を持つ場合だけです。特に、try ブロックまたは catch ブロックの中へ、あるいは、try ブロックまたは catch ブロックから外への longjmp は、直接的にも間接的にも、使用してはなりません。また、自動変数または一時変数の初期化または明示的な破棄の後ろへの longjmp も使用してはなりません。

シグナルハンドラからは例外を送出できません。

例外を含む共有ライブラリの作成

共有ライブラリが `dlopen()` によって開かれている場合は、`RTLD_GLOBAL` を使用して例外処理を実行する必要があります。

注 – 例外機能を含む共有ライブラリを構築する場合、オプション `-Bsymbolic` を `ld` に渡さないでください。捕獲されるべき例外が見つからなくなる場合があります。

第6章

実行時の型識別

本章では、実行時の型識別 (RTTI) の使用について説明します。この機能は、コンパイル時には確認できない型情報を、プログラムの実行時に調べる場合に使用してください。

静的な型と動的な型

C++ では、クラスへのポインタには、ポインタ宣言の中に書かれた型である「静的な型」と、参照された実際の型によって決定される「動的な型」があります。オブジェクトの動的な型は、静的な型から派生したものであればいずれのクラス型でもかまいません。

次の例で、ap には静的な型 A* と動的な型 B* があります。

```
class A {};
class B: public A {};
extern B bv;
extern A* ap = &bv;
```

RTTI によって、プログラマがポインタの動的な型を決定できるようになります。

RTTI オプション

互換モード (-compat[=4]) で RTTI を行うには、実装のためにかなりの資源が必要になります。このモードでは、デフォルトで RTTI は無効になっています。RTTI 実装を有効にし、かつ関連する typeid キーワードの認識を有効にするには、

-features=rtti オプションを使用してください。RTTI 実装を無効にし、かつ関連する typeid キーワードの認識を無効にするには、-features=no%rtti オプション (デフォルト) を使用してください。

標準モード (デフォルトモード) の場合、RTTI がプログラムのコンパイルや実行に大きな影響を与えることはありません。標準モードでは、RTTI は常に有効になります。

typeid 演算子

typeid 演算子はクラス type_info のオブジェクトへの参照を生成します。これはそのオブジェクトの最も派生の進んだ型を記述するものです。typeid() 関数を使用するためには、ソースコードが <typeinfo> ヘッダーファイルをインクルード (#include) していなければなりません。この演算子とクラスの組み合わせは、比較を行うときにその利点を発揮します。そのような比較では、次の例に示すように最上位の const と volatile 修飾子は無視されます。この例では、A と B はデフォルトのコンストラクタを持つ型です。

```
#include <typeinfo>
#include <assert.h>
void use_of_typeinfo( )
{
      A a1;
      const A a2;
      assert( typeid(a1) == typeid(a2) );
      assert( typeid(A)  == typeid(const A) );
      assert( typeid(A)  == typeid(a2) );
      assert( typeid(A)  == typeid(const A&) );
      B b1;
      assert( typeid(a1) != typeid(b1) );
      assert( typeid(A)  != typeid(B) );
}
```

typeid 演算子はヌルポインタが与えられたとき bad_typeid 例外割り込みを生成します。

type_info クラス

クラス type_info は typeid 演算子によって生成された型情報を記述します。type_info によって提供される基本関数は等式、不等式、before、および name です。<typeinfo.h> 中の定義は次のようになります。

```
class type_info {
    public:
        virtual ~type_info( );
        bool operator==( const type_info &rhs ) const;
        bool operator!=( const type_info &rhs ) const;
        bool before( const type_info &rhs ) const;
        const char *name( ) const;
    private:
        type_info( const type_info &rhs );
        type_info &operator=( const type_info &rhs );
};
```

before 関数は、2 つの型の実装時の照合順序を比較します。name 関数は、変換と表示に適した実装定義のヌルで終わる多バイト文字列を返します。

コンストラクタは非公開メンバー関数なので、プログラマが型「type_info」の変数を作成することはできません。「type_info」オブジェクトを得るには「typeid」演算子を使用してください。

第7章

キャスト演算

この章では、C++ 標準で新しいキャスト演算子である、const_cast、reinterpret_cast、static_cast、dynamic_cast について説明します。キャストは、オブジェクトまたは値の型を別の型に変換します。

新しいキャスト演算

C++ 標準では、以前のキャスト演算よりキャストの制御が優れた新しいキャスト演算を定義しています。dynamic_cast<> 演算子では、多様なクラスへのポインタの実際の型を調べることができます。古い形式のキャストを検索するには構文解析が必要ですが、新しい形式のキャストはテキストエディタを使用してすべて検索できます (_cast を検索する)。

それ以外では、これらの新しいキャストはすべて、古いキャスト表記で許可されたキャスト演算の一部を実行します。たとえば、const_cast<int*>v は (int*)v と書くことができます。これらの新しいキャストは、利用できる演算の種類を簡潔に分類してプログラマの意図をより明確に示し、コンパイラがより効率のよい検査を行うようにします。

キャスト演算子は常に有効です。無効にはできません。

const キャスト

式 const_cast<*T*>(v) を使用して、ポインタまたは参照の const 修飾子または volatile 修飾子を変更することができます(新しい形式のキャストの内、const 修飾子を削除できるのは const_cast<> のみ)。T はポインタ、参照、またはメンバー型へのポインタでなければなりません。

```
class A
{
public:
  virtual void f();
  int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
  const A a1;
  const_cast<A&>(a1).f( );              // const を削除
ip = const_cast<int*> (cvip);           // const と volatile を削除
}
```

解釈を変更するキャスト

式 reinterpret_cast<*T*>*(v)* は式 *v* の値の解釈を変更します。この式は、ポインタ型と整数型の間、2 つの無関係なポインタ型の間、ポインタ型からメンバー型へ、ポインタ型から関数型へ、という各種の変換に使用できます。

reinterpret_cast 演算子を使用すると、未定義の結果または実装に依存しない結果を出すことがあります。次に、確実な動作について説明します。

- データオブジェクトまたは関数へのポインタ (メンバーへのポインタは除く) は、それを十分保持できる大きさの任意の整数型に変換できます (long 型は十分大きいため、Sun WorkShop C++ がサポートするアーキテクチャでは常にポインタ値を保持できます)。元の型に戻しても、値は元の値と同じになります。
- (非メンバー) 関数へのポインタは、別の (非メンバー) 関数型へのポインタに変換できます。元の型に戻しても、値は元の値と同じになります。

- 新しい型が元の型よりも厳しい整列条件を持たない場合、オブジェクトへのポインタは別のオブジェクト型へのポインタに変換できます。元の型に戻しても、値は元の値と同じになります。
- reinterpret_cast 演算子を使用して型「*T1* のポインタ」の式を型「*T2* のポインタ」に変換できる場合、型 *T1* の左辺値は型「*T2* の参照」に変換できます。
- *T1* と *T2* の両方が関数型であるか両方がオブジェクト型である場合、「型 *T1* の X のメンバーを指すポインタ」型の右辺値は、「型 *T2* の Y のメンバーを指すポインタ」型の右辺値に明示的に変換できます。
- (変換が許可されている場合) ある型のヌルポインタは別の型のヌルポインタに変換された後もヌルポインタのままです。
- reinterpret_cast 演算子を使用して、const を const でない型にキャストすることはできません。このようにキャストするには const キャストを使用します。
- reinterpret_cast 演算子を使用して、ポインタと、同じクラス階層に存在する別のクラスの間の変換を行うことはできません。このように変換するには、静的キャストまたは動的キャストを使用します (reinterpret_cast は必要があっても調整は行わない)。次にこの例を示します。

```
class A { int a; public: A(); };
class B : public A { int b, c; }
void use_of_reinterpret_cast( )
{
     A a1;
     long l = reinterpret_cast<long>(&a1);
     A* ap = reinterpret_cast<A*>(l);       // 安全
     B* bp = reinterpret_cast<B*>(&a1);     // 安全ではない
     const A a2;
    ap = reinterpret_cast<A*>(&a2);       // エラー、const が削除された
}
```

静的キャスト

式 `static_cast<T>(v)` は式の値 *v* を型 *T* の値に変換します。この式は、暗黙的に実行されるすべての型変換に使用できます。さらに、いかなる値でも `void` にキャストすることができ、いかなる暗黙的型変換でも、そのキャストが旧式のキャストと同様に正当である限り、反転させることができます。

```
class B               { ... };
class C : public B { ... };
enum E { first=1, second=2, third=3 };
void use_of_static_cast(C* c1 )
    {
      B* bp = c1;                    // 暗黙的な変換
      C* c2 = static_cast<C*>(bp); // 暗黙的な変換を反転させる
      int i = second;                // 暗黙的な変換
      E e = static_cast<E>(i);      // 暗黙的な変換を反転させる
    }
```

`static_cast` 演算子を使用して、`const` を `const` 以外の型にするようなキャストを行うことはできません。階層の下位に (基底から派生ポインタまたは参照へ) キャストするには `static_cast` を使用できますが、変換は検証されず、結果は使用できない場合があります。抽象基底クラスから下位へのキャストには、`static_cast` は使用できません。

動的キャスト

クラスへのポインタ (または参照) は、そのクラスから派生されたすべてのクラスを実際に指す (参照する) ことができます。場合によっては、オブジェクトの完全派生クラス、またはその完全なオブジェクトの他のサブオブジェクトへのポインタを得る方が望ましいことがあります。動的キャストによってこれが可能になります。

注 – 互換モード (`-compat[=4]`) でコンパイルする場合、プログラムが動的キャストを使用している場合は、`-features=rtti` を付けてコンパイルする必要があります。

動的な型のキャストは、あるクラス *T1* へのポインタ (または参照) を別のクラス *T2* のポインタ (または参照) に変換します。*T1* と *T2* は、同じ階層内になければなりません。両クラスとも (公開派生を介して) アクセス可能でなければならず、変換はあいまいであってはなりません。また、変換が派生クラスからその基底クラスの 1 つに対するものでないかぎり、*T1* と *T2* の両方が入った階層の最小の部分は多相性がなければなりません (少なくとも仮想関数が 1 つ存在すること)。

式 `dynamic_cast`<*T*>*(v)* では、*v* はキャストされる式であり、*T* はキャストの対象となる型です。*T* は完全なクラス型 (定義が参照できるもの) へのポインタまたは参照であるか、あるいは「*cv* `void` へのポインタ」でなければなりません。ここで *cv* は空の文字列、`const`、`volatile`、`const volatile` のいずれかです。

階層の上位にキャストする

階層の上位にキャストする場合で、*v* が指す (参照する) 型の基底クラスを *T* が指す (あるいは参照する) 場合、変換は `static_cast`<*T*>(*v*) で行われるものと同じです。

`void*` にキャストする

T が `void*` の場合、結果はオブジェクト全体のポインタになります。つまり、*v* はあるオブジェクト全体の基底クラスの 1 つを指す可能性があります。この場合、`dynamic_cast<void*>`*(v)* の結果は、*v* をオブジェクト全体の型 (種類は問わない) に変換した後で `void*` に変換した場合と同じです。

`void*` にキャストする場合、階層に多相性がなければなりません (仮想関数が存在すること)。結果は実行時に検証されます。

階層の下位または全体にキャストする

階層の下位または全体にキャストする場合、階層に多相性がなければなりません (仮想関数を持つ必要がある)。結果は実行時に検証されます。

階層の下位または全体にキャストする場合、*v* から *T* に変換できないことがあります。たとえば、試行された変換があいまいであったり、*T* に対するアクセスが不可能であったり、あるいは必要な型のオブジェクトを *v* が指さない (あるいは参照しない) 場合がこれに当たります。実行時検査が失敗し、*T* がポインタ型である場合、キャスト式の値は型 T のヌルポインタです。*T* が参照型の場合、何も返されず (C++ にはヌル参照は存在しない)、標準例外 `std::bad_cast` が送出されます。

たとえば、次の例は公開クラスを継承しているため成功します。

```
class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, public B { };

void simple_dynamic_casts( )
{
  AB  ab;
  B*  bp  = &ab;        // キャストは不要
  A*  ap  = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // 成功
  ap = dynamic_cast<A*>(bp);         assert( ap != NULL );
  bp = dynamic_cast<B*>(ap);         assert( bp != NULL );
  ap = dynamic_cast<A*>(&abr);       assert( ap != NULL );
  bp = dynamic_cast<B*>(&abr);       assert( bp != NULL );
}
```

しかし、次の例は、基底クラス B にアクセスできないために失敗します。

```
class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, private B { };

void attempted_casts( )
{
  AB ab;
  B* bp  = (B*)&ab;     // B が非公開であるため、C の形式のキャストが必要
  A* ap  = dynamic_cast<A*>(bp); // 失敗。B にはアクセス不能
  assert(ap == NULL);
  AB& abr = dynamic_cast<AB&>(*bp);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // 失敗。B にはアクセス不能
  }
  catch(const bad_cast&) {
    return; // ここで失敗した参照キャストが捕獲される
  }
  assert(0); // ここまでは到達しない
}
```

1 つの基底クラスについて仮想継承と多重継承が存在する場合には、実際の動的キャストは一意の照合を識別することができなければなりません。もし照合が一意でないならば、そのキャストは失敗します。たとえば、下記の追加クラス定義が与えられたとします。

```
class AB_B :     public AB,        public B  { };
class AB_B__AB : public AB_B,      public AB { };
```

上記の定義の後には次の関数が続きます。

```
void complex_dynamic_casts( )
    {
      AB_B__AB ab_b__ab;
      A*ap = &ab_b__ab;
                    // OK: A を静的に特定できる
      AB*abp = dynamic_cast<AB*>(ap);
                    // 失敗: あいまい
      assert( abp == NULL );
                    // 静的エラー: AB_B* ab_bp = (AB_B*)ap;
                    // 動的キャストではない
      AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // 動的キャストは成功
      assert( ab_bp != NULL );
    }
```

dynamic_cast のエラー時のヌル (NULL) ポインタの戻り値は、コード中の 2 つのブロック (1 つは型推定が正しい場合にキャストを処理するためのもの、もう 1 つは正しくない場合のもの) の間の条件として役立ちます。

```
void using_dynamic_cast( A* ap )
    {
      if ( AB *abp = dynamic_cast<AB*>(ap) )
          {    // abp は NULL ではない。
               // したがって ap は AB オブジェクトへのポインタである。
               // abp を使用する。
            process_AB( abp ); }
      else
          {    // abp は NULL である。
               // したがって ap は AB オブジェクトへのポインタではない。
               // abp は使用しない。
            process_not_AB( ap );
    }
```

互換モード (-compat[=4]) では、-features=rtti コンパイラオプションによって実行時の型情報が有効になっていないと、コンパイラは dynamic_cast を static_cast に変換し、警告メッセージを出します (57 ページの「RTTI オプション」を参照)。

実行時型情報が無効にされている場合、すなわち -features=no%rtti の場合には 、コンパイラは dynamic_cast を static_cast に変換し、警告を発行します。参照型への動的キャストを行う場合は、そのキャストが実行時に無効であると判明したときに送出される例外が必要です。

動的キャストは必然的に、仮想関数による変換のような適切な設計パターンより遅くなります。Erich Gamma 著 (ソフトバンク)『オブジェクト指向における再利用のためのデザインパターン』を参照してください。

第8章

パフォーマンス

C++ 関数のパフォーマンスを高めるには、コンパイラが C++ 関数を最適化しやすいように関数を記述することが必要です。言語一般、特に C++ のソフトウェアパフォーマンスについて関連する書籍は多数あります。たとえば、Tom Cargill 著、Addison-Wesley、1992 年発行、『C++ Programming Style』、Jon Louis Bentley 著、Prentice-Hall、1982 年発行、『Writing Efficient Programs』、Dov Bulka と David Mayhew 共著、Addison-Wesley、2000 年発行、『Efficient C++: Performance Programming Techniques』、Scott Meyers 著、Addison-Wesley、1998 年発行、『Effective C++ - 50 Ways to Improve Your Programs and Designs, Second Edition』などを参照してください。この章では、これらの書籍にある内容を繰り返すのではなく、Sun C++ コンパイラにとって特に有効なパフォーマンス向上の手法について説明します。

一時オブジェクトの回避

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

プログラムの明瞭さを保ちつつ、一時オブジェクトの数が最小になるように関数を記述してください。このための手法としては、暗黙の一時オブジェクトに代わって明示的な変数を使用すること、値パラメータに代わって参照パラメータを使用することなどがあります。また、`+` と `=` だけを実装して使用するのではなく、`+=` のような演算を

実装および使用することもよい手法です。たとえば、次の例の最初の行は、`a + b` の結果に一時オブジェクトを使用していますが、2 行目は一時オブジェクトを使用していません。

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

インライン関数の使用

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

関数定義を変更する可能性があり、呼び出し元をすべて再コンパイルするには手間がかかると予測される場合は、インライン関数は使用しないでください。そうでない場合は、関数をインライン展開するコードが関数を呼び出すコードよりも小さいか、あるいはアプリケーションの動作がインライン関数によって大幅に高速化される場合にのみ使用してください。

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

- ループ、`switch` 文、`try` および `catch` 文のような難しい制御構造が関数に含まれる場合。実際には、これらの関数では、その難しい制御構造はごくまれにしか実行されません。このような関数をインライン展開するには、難しい制御構造が入った内側部分と、内側部分を呼び出すかどうかを決定する外側部分の 2 つに関数を分割します。コンパイラが関数全体をインライン展開できる場合でも、このようによく使用する部分とめったに使用しない部分を分けることで、パフォーマンスを高めることができます。

- インライン関数本体のサイズが大きいか、あるいは複雑な場合。見たところ単純な関数本体は、本体内でほかのインライン関数を呼び出していたり、あるいはコンストラクタやデストラクタを暗黙に呼び出していたりするために複雑な場合があります (派生クラスのコンストラクタとデストラクタでこのような状況がよく起きる)。このような関数ではインライン展開でパフォーマンスが大幅に向上することはめったにないため、インライン展開しないことをお勧めします。
- インライン関数呼び出しの引数が大きいか、あるいは複雑な場合。インラインメンバー関数を呼び出すためのオブジェクトが、そのインライン関数呼び出しの結果である場合は、パフォーマンスが大幅に下がります。複雑な引数を持つ関数をインライン展開するには、その関数引数を局所変数を使用して関数に渡してください。

デフォルト演算子の使用

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

デフォルト演算子はインライン関数であるため、インライン関数が適切でない場合にはデフォルト演算子を使用しないでください (前の節を参照)。デフォルト演算子は、次のような場合に適切です。

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

- ユーザーが記述するデストラクタが空の場合

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

値クラスの使用

構造体や共用体などの C++ クラスは、値によって渡され、値によって返されます。POD (Plain-Old-Data) クラスの場合、C++ コンパイラは構造体を C コンパイラと同様に渡す必要があります。これらのクラスのオブジェクトは、直接渡されます。ユーザー定義のコピーコンストラクタを持つクラスのオブジェクトの場合、コンパイラは実際にオブジェクトのコピーを構築し、コピーにポインタを渡し、ポインタが戻った後にコピーを破棄する必要があります。これらのクラスのオブジェクトは、間接的に渡されます。この 2 つの条件の中間に位置するクラスの場合は、コンパイラによってどちらの扱いにするかが選択されます。しかし、そうすることでバイナリ互換性に影響が発生するため、コンパイラは各クラスに矛盾が出ないように選択する必要があります。

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

互換モード (`-compat[=4]`) では、クラスに次の要素が含まれる場合、クラスは間接的に渡されます。

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

これらの要素が含まれない場合は、クラスは直接渡されます。

標準モード (デフォルトモード) では、クラスに次の要素が含まれる場合、クラスは間接的に渡されます。

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

これらの要素が含まれない場合は、クラスは直接渡されます。

クラスを直接渡す

クラスが直接渡される可能性を最大にするには、次のようにしてください。

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

各種のプロセッサでクラスを直接渡す

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

表 8-1　アーキテクチャ別の構造体と共用体の渡し方

アーキテクチャ	説明
SPARC V7 および V8	構造体と共用体は、呼び出し元で記憶領域を割り当て、ポインタをその記憶領域に渡すことによって渡されます (つまり、構造体と共用体はすべて参照により渡されます)。
SPARC V9	16 バイト (32 バイト) 以下の構造体は、レジスタ中で渡され (返され) ます。共用体と他のすべての構造体は、呼び出し元で記憶領域を割り当て、ポインタをその記憶領域に渡すことによって渡され (返され) ます (つまり、小さな構造体はレジスタ中で渡され、共用体と大きな構造体は参照により渡されます)。この結果、小さな値のクラスは基本の型と同じ効率で渡されることになります。
IA プラットフォーム	構造体と共用体を渡すには、スタックで領域を割り当て、引数をそのスタックにコピーします。構造体と共用体を返すには、呼び出し元のフレームに一時オブジェクトを割り当て、一時オブジェクトのアドレスを暗黙の最初のパラメータとして渡します。

メンバー変数のキャッシュ

C++ メンバー関数では、メンバー変数へのアクセスが頻繁に行われます。

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

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

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

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

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

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

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

上のコードが次の指令で呼び出されると、意図しない結果になります。

```
x*=x;
```

第9章

マルチスレッド化されたプログラム

この章ではマルチスレッド化されたプログラム (以降、マルチスレッドプログラムと呼ぶ) の作成方法について説明します。また、例外の使用方法と、スレッド間で C++ 標準ライブラリオブジェクトを共有する方法についても説明します。

マルチスレッドの詳細については、『マルチスレッドプログラミングガイド』、『C++ ライブラリ・リファレンス』、『Tools.h++ 7.0 ユーザーズガイド』、および『Standard C++ Library Class Reference』を参照してください。

マルチスレッドプログラムの構築

C++ コンパイラに付属されているライブラリはすべてマルチスレッドに対して安全です。マルチスレッド化されたアプリケーション (以降、マルチスレッドアプリケーションと呼ぶ) を作成したい場合、またはアプリケーションからマルチスレッド対応のライブラリにリンクしたい場合、`-mt` オプションを指定してプログラムをコンパイルおよびリンクしなければなりません。このオプションは、`-D_REENTRANT` をプリプロセッサに渡し、`-lthread` を正しい順序で `ld` に渡します。互換モード (`-compat[=4]`) の場合、`-mt` オプションを指定すると、`libthread` は必ず `libC` の前にリンクされます。標準モード (デフォルトモード) の場合、`-mt` オプションを指定すると、`libthread` は必ず `libCrun` の前にリンクされます。

`-lthread` を使ってアプリケーションを直接リンクしないでください。直接リンクすると、`libthread` が誤った順序でリンクされてしまいます。

コンパイルとリンクを別々の手順で行なった場合の、マルチスレッドアプリケーションの正しい作成方法の例を次に示します。

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

次の例は、マルチスレッドアプリケーションの誤った作成方法です。

```
example% CC -c -mt myprog.o
example% CC myprog.o -lthread <- libthread が正しくリンクされない
```

マルチスレッド対応コンパイルの確認

アプリケーションが libthread にリンクされているかどうかは、ldd コマンドで確認できます。

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

スレッドとシグナルに対する C++ サポートライブラリの使用

C++ サポートライブラリ libCrun、libiostream、libCstd、および libC は、マルチスレッドに対しては安全ですが、async については安全ではありません。つまり、マルチスレッドアプリケーションでは、サポートライブラリにある関数でも、シグナルハンドラでは使用するべきではありません。使用すると、デッドロックが発生する可能性があります。

マルチスレッドアプリケーションではシグナルハンドラで以下を使用することは、安全ではありません。

- 入出力ストリーム

- new 式と delete 式
- 例外

マルチスレッドプログラムでの例外の使用

現在の例外処理実装はマルチスレッド対応になっています。つまり、あるスレッド内の例外は、別のスレッドの例外を妨害することはありません。しかし、例外をスレッド間でやりとりすることはできません。つまり、あるスレッドが送出した例外を別のスレッドで受け取ることはできません。

terminate() 関数や unexpected() 関数は、各スレッドごとに独自のものを設定することができます。set_terminate() や set_unexpected() をあるスレッドで呼び出しても、その例外は呼び出し元のスレッドにだけ影響します。terminate() がデフォルトで呼び出す関数は、対象がメインスレッドであれば abort() で、その他のスレッドであれば thr_exit() です。47 ページの「実行時のエラーの指定」を参照してください。

注 – スレッドの取り消し (pthread_cancel(3T)) を行うと、スタック上の自動 (局所的で非静的) オブジェクトが破棄されます。スレッドが取り消されると、局所的デストラクタの実行の前に、ユーザーが pthread_cleanup_push() で登録した掃除用 (クリーンアップ) ルーチンが起動されます。特定のクリーンアップルーチンが登録された後に呼び出された関数の局所型オブジェクトは、ルーチンの実行前に破棄されます。

スレッド間での C++ 標準ライブラリオブジェクトの共有

C++ 標準ライブラリ (libCstd) はマルチスレッドに対して安全なので、ライブラリの内部はマルチスレッド環境で正しく動作します。しかし、自分で設定したスレッド間での共有ライブラリオブジェクトは、必ずロックしなければなりません (iostreams と locale オブジェクトを除きます)。

たとえば、文字列をインスタンス化すると、新しいスレッドを作成し、その文字列を参照によりスレッドに渡します。次に、その文字列への書き込みアクセスをロックしなければなりません。これは、スレッド間でその 1 つの文字列オブジェクトを明示的に共有しているからです (この作業を行うためにライブラリにより提供される機能については、次に説明します)。

一方、新しいスレッドに文字列を値で渡す場合、ロックについて考慮する必要はありません。これは、2 つの異なるスレッドにある文字列が Rogue Wave の「コピー・オン・ライト (書き込み時コピー)」技法により 1 つの表現を共有している場合も同じです。

ライブラリはこのロックを自動的に行います。自分でロックを行う必要があるのは、スレッド間での参照を渡したり、大域オブジェクトや静的オブジェクトを使うなどして、あるオブジェクトを複数のスレッドで明示的に使用可能にする場合だけです。

次に、C++ 標準ライブラリ内部で使用されているロック (同期) 機構について説明します。この機構によってスレッドが複数ある場合でも誤動作が発生しない仕組みになっています。

この機能へのインタフェース (ファイル、マクロ、クラス、クラスメンバーの名前を含む) は、実装同様、ライブラリ内部の詳細であるため、予告なしに変更される場合があります。後方互換性は保証されていません。

2 つの同期クラス `_RWSTDMutex` と `_RWSTDGuard` が、マルチスレッド安全を実現するための機構を提供しています。

`_RWSTDMutex` クラスは、次のメンバー関数により、プラットフォームに依存しないロック機構を提供しています。

- `void acquire()` — 自分自身に対するロックを獲得するか、ロックが獲得できるまでブロックする
- `void release()` —自分自身に対するロックを解放する

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

_RWSTDGuard クラスは、_RWSTDMutex クラスのオブジェクトをカプセル化する、便利なラッパークラスです。_RWSTDGuard オブジェクトは、自分自身のコンストラクタ中でカプセル化された相互排他ロック (mutex) を獲得しようと試み (エラー時にスローされる std::exception から派生した ::thread_error 型の例外をスローし)、デストラクタで相互排他ロックを解放します (デストラクタは例外をスローしない)。

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

また、マクロ _RWSTD_MT_GUARD (mutex) (以前の _STDGUARD) を使用すると、マルチスレッドアプリケーションを構築する際に _RWSTDGuard クラスオブジェクトの生成を条件によって有効または無効にできます。_RWSTDGuard オブジェクトは、自分が定義されているブロック中のコードが、複数のスレッドによって同時に実行されないように保護します。スレッドが 1 つしかない場合は、_RWSTD_MT_GUARD マクロは空の式に展開されます。

次に、これらの機構の使用例を示します。

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

//
// 複数のスレッドで共有される整数
//
int I;

//
// I への更新を同期させるために使用する相互排他ロック (mutex)
//
_RWSTDMutex I_mutex;

//
// I を 1 増やす。_RWSTDMutex を直接使用する
//

void increment_I ()
{
   I_mutex.acquire();  // mutex をロックする
   I++;
   I_mutex.release();  // mutex のロックを解放する
}

//
// I を 1 減らす。_RWSTDGuard を使用する
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex);  // I_mutex に対するロックを獲得する
   --I;
   //
   // I に対するロックは、guard デストラクタが呼び出されたときに解放される
   //
}
```

索引

う

え

お

か

き

く

こ

さ

し

れ